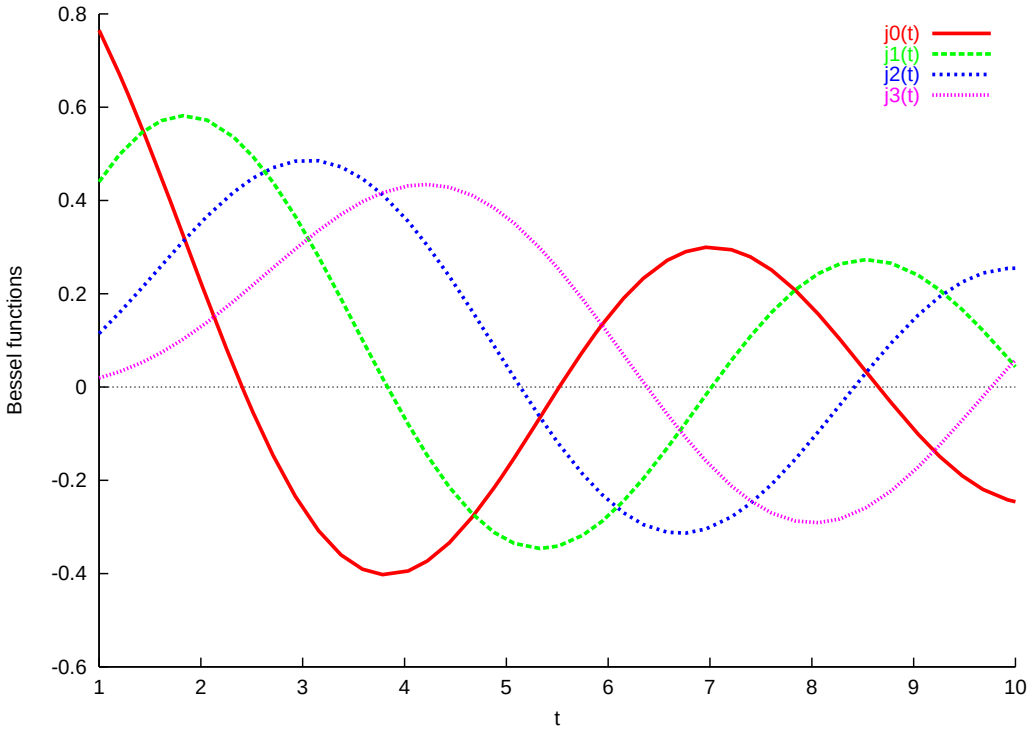
SoftIntegration®

The Ch Language Environment

Version 7.0

User's Guide



How to Contact SoftIntegration

Mail	SoftIntegration, Inc. 216 F Street, #68 Davis, CA 95616
Phone	+ 1 530 297 7398
Fax	+ 1 530 297 7392
Web	http://www.softintegration.com
Email	info@softintegration.com

Copyright ©2012 by SoftIntegration, Inc. All rights reserved.
Revision 7.0.0, May 2012

Permission is granted for registered users to make one copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited.

SoftIntegration, Inc. is the holder of the copyright to the Ch language environment described in this document, including without limitation such aspects of the system as its code, structure, sequence, organization, programming language, header files, function and command files, object modules, static and dynamic loaded libraries of object modules, compilation of command and library names, interface with other languages and object modules of static and dynamic libraries. Use of the system unless pursuant to the terms of a license granted by SoftIntegration or as otherwise authorized by law is an infringement of the copyright.

SoftIntegration, Inc. makes no representations, expressed or implied, with respect to this documentation, or the software it describes, including without limitations, any implied warranty merchantability or fitness for a particular purpose, all of which are expressly disclaimed. Users should be aware that included in the terms and conditions under which SoftIntegration is willing to license the Ch language environment as a provision that SoftIntegration, and their distribution licensees, distributors and dealers shall in no event be liable for any indirect, incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of the Ch language environment, and that liability for direct damages shall be limited to the amount of purchase price paid for the Ch language environment.

In addition to the foregoing, users should recognize that all complex software systems and their documentation contain errors and omissions. SoftIntegration shall not be responsible under any circumstances for providing information on or corrections to errors and omissions discovered at any time in this documentation or the software it describes, even if SoftIntegration has been advised of the errors or omissions. The Ch language environment is not designed or licensed for use in the on-line control of aircraft, air traffic, or navigation or aircraft communications; or for use in the design, construction, operation or maintenance of any nuclear facility.

Ch, ChIDE, SoftIntegration, and One Language for All are either registered trademarks or trademarks of SoftIntegration, Inc. in the United States and/or other countries. Microsoft, MS-DOS, Windows, Windows 2000, Windows XP, Windows Vista, and Windows 7 are trademarks of Microsoft Corporation. Solaris and Sun are trademarks of Sun Microsystems, Inc. Unix is a trademark of the Open Group. HP-UX is either a registered trademark or a trademark of Hewlett-Packard Co. Linux is a trademark of Linus Torvalds. Mac OS X and Darwin are trademarks of Apple Computers, Inc. QNX is a trademark of QNX Software Systems. AIX is a trademark of IBM. All other trademarks belong to their respective holders.

Preface

Ch (pronounced C H) is an embeddable C/C++ interpreter. The Ch language environment was originally designed and implemented by Dr. Harry H. Cheng for teaching introductory computer programming in C for engineering applications and his research projects in mechatronics and computational kinematics. As the user's base increases, Ch has been evolved from a special-application program to a general-purpose language environment with wide applicability. Similar to using a natural language, it was intended to use *One Language for All* programming purposes.

The C language was selected for its compact syntax, expressive power, and wide availability. But, for many applications, an interpreter is more desirable. As a result, a C interpreter has been developed over the years. To distinguish it from many existing C compilers, this C interpreter is called Ch. But, Ch was never meant to be a new language. Conforming to the C standard is its prevailing design goal for Ch. We believe that Ch is the most complete C interpreter in existence. As a complete C interpreter, Ch supports all features of the ISO C90 standard ratified in 1990. Ch supports some major features in C++ for object-based programming. C was originally designed for system programming. It has many deficiencies for applications in engineering and science. Ch extends C for very high-level numerical computing and graphical plotting, shell programming, and embedded scripting.

Our work on extending C for numerical computing overlapped with the ASNI C Standard Committee's effort in revising the C standard. Ch has greatly benefited from our participation in ANSI X3J11 and ISO S22/WG14 C Standard Committees since 1993. Many new features such as complex numbers, variable-length arrays (VLA), binary constants, and function name `__func__` first implemented in Ch had been added in the latest C standard called C99. In its current implementation, Ch supports more new features added in the ISO C99 than most existing C compilers. C programmers are encouraged to use these new features such as complex numbers, variable length arrays (VLA), IEEE floating-point arithmetic, type generic mathematical functions described in this manuscript because they significantly simplify many programming tasks. In addition to these numerical features in C99, Ch also supports computational arrays as first-class objects as in Fortran 90 and MATLAB for linear algebra and matrix computations.

A proposal was submitted to the C Standard Committee to add classes in C++ to the C99 standard. Due to the time constraint and other reasons, the proposal was not adopted in C99. Nevertheless, Ch added classes in C++ mainly based on this proposal. In addition, Ch supports reference type in C++ for convenience of passing values by reference to functions as in Fortran.

Different from many other software packages, Ch bridges the gap between low-level languages and very high-level languages. As a superset of C, Ch retains low-level features of C such as accessing memory for hardware interface. However, Ch is a very high-level language (VHLL) environment. Ch supports shell programming with a built-in string type. Some problems, which might take thousands of lines of C code, can be easily solved with only a few lines of Ch code. Ch enables programmers to increase their productivity significantly.

Furthermore, Ch is designed to be platform-independent. It can run in an heterogeneous computing environment with different computer hardware and operating systems including Windows, Linux, Mac OS X, and Unix. A program developed in one platform can run in any other platforms.

This manuscript is written to help readers learn how to program in C/Ch with new features in C99. Although prior knowledge of programming language is not required, it will be helpful to understand the basics of Ch. The emphasis of this manuscript is on extensions of C99 over C90. Extensions to C in Ch are highlighted. Features not explained in this manuscript follow the interpretation of the ISO C99 standard. The examples include many testing codes used during the development of Ch. Whether you are a novice computer user or experienced professional programmer, we hope that Ch will make your programming tasks more enjoyable and that you will like Ch.

This user's guide is prepared for different editions of Ch. There are Standard, Professional, and Student Editions for Ch. All features in this manual are available for both Professional and Student Editions. Features described in chapters 16, 23, and 24 on computational array, plotting, and numerical analysis, respectively, are not available for Ch Standard Edition.

If you are one of those impatient C/C++ programmers, you may jump to Appendices B and C which highlight the differences between Ch and C/C++ with an overview of new features in Ch extended over C/C++. After that, you can start programming in Ch quickly without lengthy compile/link/execute/debug cycles. You can just run your C/C++ programs in a Ch command shell by typing program names such as `hello.c` or `hello.cpp`. You can also run C/C++ programs in Ch from an Integrated Development Environment (IDE).

Typographical Conventions

The following list defines and illustrates typographical conventions used as visual cues for specific elements of the text throughout this document.

- Interface components are window titles, button and icon names, menu names and selections, and other options that appear on the monitor screen or display. They are presented in boldface. A sequence of pointing and clicking with the mouse is presented by a sequence of boldface words.

  Example: Click **OK**

  Example: The sequence **Start->Programs->Ch7.0->Ch** indicates that you first select **Start**. Then select submenu **Programs** by pointing the mouse on **Programs**, followed by **Ch7.0**. Finally, select **Ch**.

- Keycaps, the labeling that appears on the keys of a keyboard, are enclosed in angle brackets. The label of a keycap is presented in typewriter-like typeface.

  Example: Press `<Enter>`

- Key combination is a series of keys to be pressed simultaneously (unless otherwise indicated) to perform a single function. The label of the keycaps is presented in typewriter-like typeface.

  Example: Press `<Ctrl><Alt><Enter>`

- Commands presented in lowercase boldface are for reference only and are not intended to be typed at that particular point in the discussion.

  Example: "Use the **install** command to install..."

  In contrast, commands presented in the typewriter-like typeface are intended to be typed as part of an instruction.

  Example: "Type `install` to install the software in the current directory."

- Command Syntax lines consist of a command and all its possible parameters. Commands are displayed in lowercase bold; variable parameters (those for which you substitute a value) are displayed in lowercase italics; constant parameters are displayed in lowercase bold. The brackets indicate items that are optional.

  Example: **ls** [**-aAbcCdfFgilLmnopqrRstux1**] [*file* ...]

- Command lines consist of a command and may include one or more of the command's possible parameters. Command lines are presented in the typewriter-like typeface.

  Example: `ls /home/username`

- Screen text is a text that appears on the screen of your display or external monitor. It can be a system message, for example, or it can be a text that you are instructed to type as part of a command (referred to as a command line). Screen text is presented in the typewriter-like typeface.

  Example: The following message appears on your screen

```
usage:  rm [-fiRr] file ...


ls [-aAbcCdfFgilLmnopqrRstux1] [file ... ]
```

- Function prototype consists of return type, function name, and arguments with data type and parameters. Keywords of the Ch language, typedefed names, and function names are presented in boldface. Parameters of the function arguments are presented in italic. The brackets indicate items that are optional.

  Example: **double derivative**(**double** (**func*)(**double**), **double** *x*, ... [**double** **err*, **double** *h*]);

- Source code of programs is presented in the typewriter-like typeface.

  Example: The program **hello.ch** with code

```
int main() {
    printf("Hello, world!\n");
}
```

  will produce the output `Hello, world!` on the screen.

- Variables are symbols for which you substitute a value. They are presented in italics.

  Example: module *n* (where *n* represents the memory module number)

- System Variables and System Filenames are presented in boldface.

  Example: startup file **/home/username/.chrc** or **.chrc** in directory `/home/username` in Unix and **C:**\ >**_chrc** or **_chrc** in directory C:\ > in Windows.

- Identifiers declared in a program are presented in typewriter-like typeface when they are used inside a text.

  Example: variable `var` is declared in the program.

- Directories are presented in typewriter-like typeface when they are used inside a text.

  Example: Ch is installed in the directory `/usr/local/ch` in Unix and `C:/Ch` in Windows.

- Environment Variables are the system level variables. They are presented in boldface.

  Example: Environment variable **PATH** contains the directory /usr/ch.

Other Relevant Documentations

The core Ch documentation set consists of the following titles. These documentation come with the CD and are installed in CHHOME/docs, where CHHOME is the Ch home directory.

- *The Ch Language Environment — Installation and System Administration Guide*, version 7.0, SoftIntegration, Inc., 2011.

  This document covers system installation and configuration, as well as setup of Ch for Web servers.

- *The Ch Language Environment, — User's Guide*, version 7.0, SoftIntegration, Inc., 2011.

  This document presents language features of Ch for various applications.

- *The Ch Language Environment, — Reference Guide*, version 7.0, SoftIntegration, Inc., 2011.

  This document gives detailed references of functions, classes and commands along with sample source code.

- *The Ch Language Environment, — SDK User's Guide*, version 7.0, SoftIntegration, Inc., 2011.

  This document presents Software Development Kit for interfacing with C/C++ functions in static or dynamical libraries.

- *The Ch Language Environment CGI Toolkit User's Guide*, version 3.5, SoftIntegration, Inc., 2003.

  This document describes Common Gateway Interface in CGI classes with detailed references for each member function of the classes.

Table of Contents

Preface **i**

Ch Graphics Gallery **xvii**

Introduction **1**

I The Language Features **5**

1 Getting Started **6**
1.1 Startup . . . 6
1.1.1 Startup in Unix . . . 6
1.1.2 Startup in Windows . . . 7
1.2 Command Mode . . . 8
1.3 Program Mode . . . 10
1.3.1 Command Files . . . 10
1.3.2 Script Files . . . 11
1.3.3 Function Files . . . 12
1.4 Complex Numbers . . . 12
1.5 Computational Arrays . . . 13
1.6 Plotting . . . 16

2 Lexical Elements **18**
2.1 Character Set . . . 18
2.1.1 Trigraphs . . . 18
2.2 Keywords . . . 19
2.2.1 Keywords . . . 19
2.2.2 Reserved Symbols . . . 20
2.3 Identifiers . . . 21
2.3.1 Predefined Identifiers . . . 21
2.4 Punctuators . . . 26
2.5 Comments . . . 26

3 Program Structure **27**
3.1 Directories and Files in the Ch Home Directory . . . 27
3.2 Startup . . . 27
3.2.1 Sample Startup Files . . . 29
3.2.2 Command Line Options . . . 31

3.3 Ch Programs . . . 32
3.3.1 Command Files . . . 32
3.3.2 Script Files . . . 34
3.3.3 Function Files . . . 34
3.4 Program Execution . . . 36
3.4.1 Execution of Programming Statements in Command Mode . . . 37
3.4.2 Program Startup . . . 37
3.4.3 Program Termination . . . 38
3.4.4 Search Order . . . 39
3.4.5 Running Programs with Multiple Files . . . 39
3.4.6 Debug Programs . . . 42
3.5 Scope Rules . . . 44
3.5.1 Scopes of Identifiers . . . 44
3.5.2 Linkages of Identifiers . . . 44
3.5.3 Name Spaces of Identifiers . . . 45
3.5.4 Storage Duration of Objects . . . 45

4 Portable Interactive Command Shell and Shell Programming 47
4.1 Shell Prompts . . . 47
4.2 Interactive Execution of Commands . . . 48
4.2.1 Current Shell . . . 49
4.2.2 Background Job . . . 50
4.3 Interactive Execution of Programming Statements . . . 50
4.4 Built-in Commands . . . 53
4.4.1 Commands For Interactive Shell Only . . . 54
4.5 Repeating Commands at Prompt . . . 56
4.5.1 History Substitution . . . 56
4.5.2 Quick Substitution . . . 58
4.5.3 File Completion . . . 59
4.5.4 Command Completion . . . 60
4.6 Aliases . . . 61
4.7 Variable Substitution . . . 64
4.7.1 Expression Substitution . . . 65
4.7.2 Command Name Substitution . . . 66
4.8 Filename Substitution . . . 67
4.9 Command Substitution . . . 69
4.10 Input/Output Redirection . . . 70
4.11 Pipeline . . . 73
4.12 Running Commands in Background . . . 74
4.13 Run-Time Expression Evaluation . . . 75
4.14 Handling Environment Variables . . . 76
4.15 General-Purpose Ch Programs . . . 78
4.16 Shell Programming . . . 80
4.16.1 Use Shell Commands in Programs . . . 80
4.16.2 Passing Values to Shell Commands . . . 83

5 Preprocessing Directives **87**
5.1 Conditional Inclusion . . . 87
5.2 Source File Inclusion . . . 88
5.3 Macro Replacement . . . 89
5.4 Converting Tokens to Strings . . . 91
5.5 Token Merging in Macro Expansions . . . 91
5.6 Line Control . . . 92
5.7 Error Directive . . . 93
5.8 NULL Directive . . . 93
5.9 Pragma Directive . . . 93
5.10 Predefined Macros . . . 96

6 Types and Declarations **98**
6.1 Data Types . . . 98
6.1.1 Integral Data Types . . . 98
6.1.2 Floating-Point Types . . . 101
6.1.3 Aggregate Floating-Point Types . . . 103
6.1.4 Pointer Data Types . . . 104
6.1.5 Array Types . . . 104
6.1.6 Structure Types . . . 106
6.1.7 Class Types . . . 106
6.1.8 Bit Field . . . 107
6.1.9 Union Types . . . 107
6.1.10 Enum Types . . . 108
6.1.11 Void Type . . . 108
6.1.12 Reference Type . . . 108
6.1.13 String Type . . . 109
6.1.14 Function Types . . . 111
6.2 Type Qualifiers . . . 112
6.2.1 Computational Arrays . . . 112
6.2.2 Restricted Function . . . 112
6.3 Constants . . . 113
6.3.1 Character Constants . . . 113
6.3.2 String Literals . . . 116
6.3.3 Integer Constants . . . 116
6.3.4 Floating-Point Constants . . . 118
6.4 Initialization . . . 119

7 Operators and Expressions **122**
7.1 Arithmetic Operators . . . 125
7.2 Relational Operators . . . 125
7.3 Logical Operators . . . 128
7.4 Bitwise Operators . . . 128
7.5 Assignment Operators . . . 128
7.6 Conditional Operator . . . 129
7.7 Cast Operators . . . 131
7.7.1 Cast Operators . . . 131
7.7.2 Functional Type Cast Operators . . . 131

7.8 Comma Operator . . . 132
7.9 Unary Operators . . . 133
7.9.1 Address and Indirection Operators . . . 133
7.9.2 Increment and Decrement Operators . . . 134
7.9.3 Command Substitution Operator . . . 135
7.10 Member Operators . . . 135

8 Statements and Control Flow **136**
8.1 Simple and Compound Statements . . . 136
8.2 Expression and Null Statements . . . 136
8.3 Selection Statements . . . 137
8.3.1 If Statements . . . 137
8.3.2 If-Else Statements . . . 138
8.3.3 Else-If Statements . . . 138
8.3.4 Switch Statements . . . 138
8.4 Iteration Statements . . . 140
8.4.1 While Loop . . . 140
8.4.2 Do-While Loop . . . 140
8.4.3 For Loop . . . 141
8.4.4 Foreach Loop . . . 142
8.5 Jump Statements . . . 142
8.5.1 Break Statements . . . 143
8.5.2 Continue Statements . . . 143
8.5.3 Return Statements . . . 144
8.5.4 Goto Statements . . . 144
8.6 Labeled Statements . . . 145

9 Pointers **147**
9.1 Pointer Arithmetic . . . 147
9.2 Dynamic Allocation of Memory . . . 149
9.3 Arrays of Pointers . . . 151
9.4 Pointers to Pointers . . . 153

10 Functions **156**
10.1 Call-by-Value versus Call-by-Reference . . . 156
10.2 Function Definitions . . . 157
10.3 Function Prototypes . . . 160
10.4 Recursive Functions . . . 165
10.5 Nested Functions . . . 165
10.5.1 Scopes and Lexical Levels of Nested Functions . . . 167
10.5.2 Prototypes of Nested Functions . . . 170
10.5.3 Nested Recursive Functions . . . 171
10.6 Using Pointers to Pass Arguments of Function by Reference . . . 179
10.7 Variable Number Arguments in Functions . . . 179
10.8 Pointer to Functions . . . 186
10.9 Communication between Functions . . . 188
10.10 The `main()` Function and Command-Line Arguments . . . 190
10.11 Function Files . . . 196

10.12Generic Functions 198

11 Reference Type **200**
11.1 References in Statements 201
11.2 Passing Arguments of Function by References 203
11.3 Passing Variables of Different Data Types to the Same Reference 206

12 Scientific Computing Using Generic Mathematical Functions **210**
12.1 Generic Mathematical Functions in the Entire Domain 211
12.2 Programming Examples 215
12.2.1 Computation of Extreme Values of Floating-Point Numbers 215
12.2.2 Programming with Metanumbers 218

13 Programming with Complex Numbers **221**
13.1 Complex Numbers 221
13.1.1 Complex Constants and Complex Variables 221
13.2 Complex Planes and Complex Metanumbers 222
13.2.1 Data Conversion Rules 224
13.3 I/O for Complex Numbers 226
13.4 Complex Operations 227
13.4.1 Complex Operations with Regular Complex Numbers 227
13.4.2 Complex Operations with Complex Metanumbers 228
13.5 Complex Functions 230
13.5.1 Results of Complex Functions with Regular Complex Numbers 230
13.5.2 Results of Complex Functions With Complex Metanumbers 233
13.6 Lvalues Related to Complex Numbers 234
13.7 Creation of User's Complex Functions 237

14 Pointers and Arrays **239**
14.1 Accessing Array Elements Through Pointers 239
14.2 Dynamic Allocation of Arrays 240
14.2.1 Dynamic Allocation of One-Dimensional Arrays 240
14.2.2 Dynamic Allocation of Two-Dimensional Arrays 241
14.3 Passing One and Multi-Dimensional Arrays of Fixed Length 244
14.3.1 One-Dimensional Arrays 244
14.3.2 Multi-Dimensional Arrays of Fixed Length 245

15 Variable Length Arrays **249**
15.1 Storage Duration and Declaration of Arrays 250
15.1.1 Storage Duration of Objects 250
15.1.2 Declaration of Arrays 250
15.2 Deferred-Shape Arrays 252
15.2.1 Constraints and Semantics 252
15.2.2 Deferred-Shape Arrays Related to Switch Statement 255
15.2.3 Deferred-Shape Arrays Related to Goto Statement 255
15.2.4 Deferred-Shape Arrays as Members of Structures and Unions 257
15.2.5 Sizeof 258
15.2.6 Typedef 259

15.2.7 Other Data Types and Pointer Arithmetic . . . 260
15.3 Assumed-Shape Arrays . . . 260
15.3.1 Constraints and Semantics . . . 260
15.3.2 Sizeof . . . 263
15.3.3 Other Data Types and Pointer Arithmetic . . . 264
15.4 Pointers to Array of Assumed-Shape . . . 265
15.4.1 Declaration . . . 265
15.4.2 Constraints and Semantics . . . 266
15.4.3 Function Prototype Scope . . . 267
15.4.4 Typedef . . . 268
15.4.5 Arrays Allocated by Dynamic Memory Allocation Functions . . . 268
15.4.6 Similarities between Pointers to Fixed-Length Array and Pointers to Assumed-Shape Array 269
15.5 Arrays with Explicit Lower and Upper Bounds . . . 271
15.5.1 Arrays of Fixed Subscript Range . . . 272
15.5.2 Arrays of Variable Subscript Range . . . 275
15.6 Passing Arrays with Explicit Lower and Upper Bounds to Functions . . . 277
15.6.1 Passing Arrays of Fixed Subscript Range . . . 277
15.6.2 Passing Arrays of Variable Subscript Range Using Pointers to Assumed-Shape Array 279

16 Computational Arrays and Matrix Computations 283
16.1 Declaration and Initialization of Computational Arrays . . . 283
16.2 Array Reference . . . 285
16.2.1 Whole Arrays . . . 285
16.2.2 Array Elements . . . 285
16.3 Formatted Input and Output for Computational Arrays . . . 287
16.4 Implicit Data Type Conversion for Computational Arrays . . . 289
16.5 Array Operations . . . 290
16.5.1 Arithmetic Operations . . . 290
16.5.2 Assignment Operations . . . 292
16.5.3 Increment and Decrement Operations . . . 293
16.5.4 Relational Operations . . . 294
16.5.5 Logic Operations . . . 294
16.5.6 Conditional Operation . . . 296
16.5.7 Address Operations . . . 296
16.5.8 Cast Operations . . . 297
16.6 Promotion of Scalars to Computational Arrays in Operations . . . 297
16.7 Passing Computational Arrays to Functions . . . 298
16.7.1 Fully-Specified-Shape Arrays . . . 299
16.7.2 Assumed-Shape Arrays . . . 299
16.7.3 Deferred-Shape Arrays . . . 302
16.7.4 Arrays in Variable Number Arguments . . . 304
16.7.5 Arrays of Reference . . . 304
16.8 Computational Arrays with Value NULL . . . 312
16.9 Functions Return Computational Arrays . . . 313
16.9.1 Functions Return Computational Arrays of Fixed Length . . . 313
16.9.2 Functions Return Computational Arrays of Variable Length . . . 313
16.10 Type Generic Array Functions . . . 315
16.11 Some Commonly Used Array Functions . . . 318

16.12 Pointer to Computational Arrays . . . 320
16.12.1 Pointer to Computational Arrays of Fixed Length . . . 320
16.12.2 Pointer to Computational Arrays of Assumed Shape . . . 327
16.12.3 Using Pointer to Computational Arrays to Pass Arrays to Functions . . . 328
16.13 Relationship between Computational Arrays and C Arrays . . . 329

17 Characters and Strings 330
17.1 Using Functions in string.h Header File . . . 330
17.1.1 Copying Functions . . . 331
17.1.2 Concatenation Functions . . . 331
17.1.3 Comparison Functions . . . 332
17.1.4 Search Functions . . . 332
17.1.5 Miscellaneous Functions . . . 334
17.1.6 String Functions Supported by Ch, but not in C Standard Library . . . 334
17.2 String Type string_t . . . 335
17.3 Handling String Tokens Using foreach Loop . . . 337
17.4 Wide Characters . . . 338
17.5 Wide Strings . . . 339

18 Structures, Unions, Bit Fields, and Enumerations 340
18.1 Structures . . . 340
18.2 Unions . . . 341
18.3 Bit-fields . . . 341
18.4 Enumerations . . . 342

19 Classes and Object-Based Programming 344
19.1 Class Definition and Objects . . . 344
19.2 Member Functions of Class . . . 344
19.3 Public and Private Members of Class . . . 346
19.4 Constructors and Destructors in Class . . . 346
19.5 The new and delete Operators . . . 347
19.6 Static Member of Class . . . 348
19.7 Scope Resolution Operator :: . . . 351
19.8 The Implicit this Pointer . . . 352
19.9 Polymorphism . . . 352
19.9.1 Polymorphic Generic Mathematical Functions . . . 353
19.9.2 Functions with Parameter Type of Array of Reference . . . 353
19.9.3 Polymorphic Functions . . . 354
19.9.4 Polymorphic Member Functions of Class . . . 359
19.10 Nested Classes . . . 362
19.11 Classes inside Member Function . . . 363
19.12 Passing Member Functions to Arguments of Functions . . . 363
19.13 Predefined Identifiers __class__ and __class_func__ . . . 368

20 Input and Output 370
20.1 Streams . . . 370
20.2 Buffered and Unbuffered I/O . . . 370
20.3 I/O Formats . . . 372

20.3.1 Output Format for fprintf Family of Output Function . . . 372
20.3.2 Input Format for fscanf Family of Input Function . . . 377
20.4 Default I/O Formats . . . 381
20.4.1 Default Format for fprintf Family of Output Functions . . . 381
20.4.2 Default Format for fscanf Family of Input Functions . . . 382
20.4.3 I/O Using cout, cin, cerr, and endl . . . 383
20.5 I/O for Metanumbers . . . 384
20.6 I/O Formats for Aggregate Data Types . . . 386
20.7 Verbatim Output Blocks Using fprintf . . . 386
20.8 File Manipulation . . . 390
20.8.1 Opening and Closing a File . . . 390
20.8.2 Reading and Writing a File . . . 392
20.8.3 Random Access . . . 394
20.9 Directory Manipulation . . . 395
20.9.1 Opening and Closing a Directory . . . 395
20.9.2 Reading a Directory . . . 397

21 Safe Ch **401**
21.1 Safe Ch Shell . . . 401
21.1.1 Startup in Windows . . . 401
21.2 Features Disabled in a Sandbox . . . 401
21.3 Restricted Functions . . . 403
21.4 Safe Ch Programs . . . 403
21.5 Applets and Network Computing . . . 404

22 Library, Toolkit, and Package **405**
22.1 Library . . . 405
22.2 Toolkit . . . 409
22.3 Package . . . 410

II The Library for Scientific Computing **413**

23 Two and Three-Dimensional Plotting **414**
23.1 A Class for Plotting . . . 414
23.1.1 Data for Plotting . . . 414
23.1.2 Annotations . . . 422
23.1.3 Multiple Data Sets and Legends . . . 425
23.1.4 Using Predefined Geometric Primitives . . . 432
23.1.5 Subplots . . . 433
23.1.6 Export and Zoom Plots . . . 434
23.1.7 Print Plots . . . 434
23.2 2D Plotting . . . 437
23.2.1 Plot Types, Line Styles, and Markers . . . 437
23.2.2 Polar Plot . . . 443
23.2.3 2D Plotting Functions . . . 447
23.3 3D Plotting . . . 451
23.3.1 Plot Types . . . 451

23.3.2 Plotting in Different Coordinate Systems . . . 452
23.3.3 3D Plotting Functions . . . 455
23.4 Dynamic Web Plotting . . . 458

24 Numerical Analysis **462**
24.1 Mathematical Functions . . . 463
24.1.1 Cross Product . . . 463
24.1.2 Dot Product . . . 463
24.1.3 Uniform Random Numbers . . . 467
24.1.4 Sign Function . . . 467
24.1.5 Greatest Common Divisor . . . 468
24.1.6 Least Common Multiple . . . 468
24.1.7 Complex Equation . . . 468
24.2 Data Analysis and Statistics . . . 470
24.2.1 Get Numbers from Console . . . 470
24.2.2 Assign Data to Arrays . . . 470
24.2.3 Minimum and Maximum . . . 471
24.2.4 Sum . . . 472
24.2.5 Product . . . 473
24.2.6 Mean . . . 474
24.2.7 Median . . . 474
24.2.8 Standard Deviation . . . 474
24.2.9 Covariance and Correlation Coefficients . . . 475
24.2.10 Norm . . . 477
24.2.11 Factorial . . . 477
24.2.12 Combination . . . 478
24.2.13 Sort Data . . . 478
24.2.14 Unwrap . . . 479
24.2.15 Functions Applied to Elements of Arrays . . . 480
24.2.16 Histogram . . . 481
24.3 Data Interpolation and Curve Fitting . . . 482
24.3.1 One-Dimensional Interpolation . . . 482
24.3.2 Two-Dimensional Interpolation . . . 484
24.3.3 General Curve Fitting . . . 484
24.3.4 Curve Fitting Using Polynomial Functions . . . 487
24.4 Minimization or Maximization of Functions . . . 487
24.4.1 Minimization of Function with One Variable . . . 488
24.4.2 Minimization of Function with Multiple Variables . . . 489
24.5 Polynomials . . . 490
24.5.1 Evaluation of Polynomials . . . 491
24.5.2 Derivative of Polynomials . . . 492
24.5.3 Find Roots of Polynomials . . . 493
24.5.4 Find Coefficients of Polynomials . . . 494
24.5.5 Residues for Factorization of Polynomials . . . 495
24.5.6 Characteristic Polynomials of Matrices . . . 497
24.6 Nonlinear Equations . . . 498
24.6.1 Solve a Nonlinear Equation . . . 498
24.6.2 Solve System of Nonlinear Equations . . . 498

24.7 Derivatives and Ordinary Differential Equations . . . 499
24.7.1 Difference . . . 499
24.7.2 Derivatives . . . 499
24.8 Solve Ordinary Differential Equations . . . 500
24.9 Numerical Integration . . . 503
24.9.1 One-Dimensional Integration . . . 503
24.9.2 Two-Dimensional Integration . . . 503
24.9.3 Three-Dimensional Integration . . . 504
24.10 Matrix Functions . . . 505
24.10.1 Characteristics of Matrices . . . 506
24.10.2 Manipulation of Matrices . . . 508
24.10.3 Special Matrices . . . 509
24.10.4 Matrix Analysis . . . 513
24.11 Matrix Decomposition . . . 515
24.11.1 LU Decomposition . . . 515
24.11.2 Singular Value Decomposition . . . 516
24.11.3 Cholesky Decomposition . . . 517
24.11.4 QR Decomposition . . . 518
24.11.5 Hessenberg Decomposition . . . 520
24.11.6 Schur Decomposition . . . 521
24.12 Linear Equations . . . 521
24.12.1 Linear System of Equations . . . 521
24.12.2 Over-Determined or Under-Determined Linear System of Equations . . . 522
24.12.3 Inverse and Pseudo Inverse Matrices . . . 524
24.12.4 Linear Spaces . . . 525
24.13 Eigenvalues and Eigenvectors . . . 526
24.14 Fast Fourier Transforms . . . 528
24.15 Convolution and Filtering . . . 530
24.16 Cross Correlation . . . 537

25 Bibliography **539**

A Known Problems and Platform Specific Features **541**
A.1 Platform Specific Features . . . 541
A.1.1 Solaris . . . 541
A.1.2 Windows NT/2000/XP/Vista/Windows 7 . . . 541
A.2 Functions Not Supported in Specific Platforms . . . 541

B Comparison with C and Implementation-Defined Behaviors **543**
B.1 New C99 Features Supported in Ch . . . 543
B.2 Summary of Extensions to C . . . 544
B.3 Implementation Notes . . . 546
B.3.1 Unlimited Properties . . . 546
B.3.2 Defined Properties . . . 547
B.3.3 Temporarily Features . . . 548
B.3.4 Incompatibility between Ch and C . . . 550
B.4 Tips for Porting C to Ch . . . 551

C Comparison with C++ **554**
C.0.1 Features in Both C++ and Ch . . . 554
C.0.2 Extensions to C++ Classes in Ch . . . 555
C.0.3 C++ Features not Supported in Ch . . . 555
C.0.4 Differences Between C++ and Ch . . . 557

D Comparison with C Shell **558**
D.1 Syntax . . . 558
D.2 Control Flow . . . 561

E Comparison with MATLAB **562**
E.1 Operators . . . 563
E.2 Functions and Constants . . . 565
E.3 Control Flow . . . 574

F Comparison with Fortran **575**
F.1 Reference in Ch versus Equivalence in FORTRAN . . . 575
F.2 Call-by-Reference in Ch and in FORTRAN . . . 576

G Summary of Commonly Used Portable Shell Commands in Ch **579**
G.1 File Systems . . . 579
G.2 Binary Files . . . 580
G.3 Text Files . . . 580
G.4 Comparing Files . . . 581
G.5 Shell Utilities . . . 581
G.6 Archiving Files . . . 581

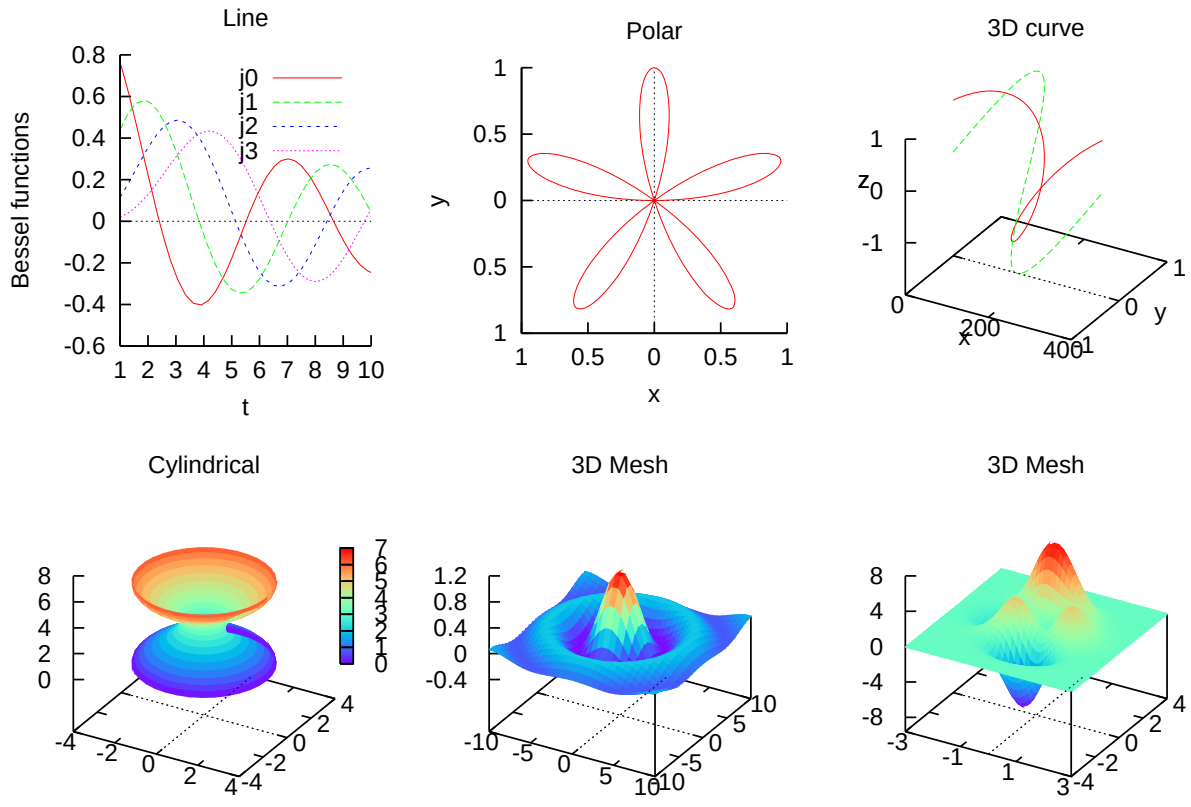
H Summary of vi Text Editor **582**

I Porting Code to the Latest Version **585**
I.1 Porting Code to Ch Version 6.0.0.13581 . . . 585
I.2 Porting Code to Ch Version 6.0.0.13581 . . . 585

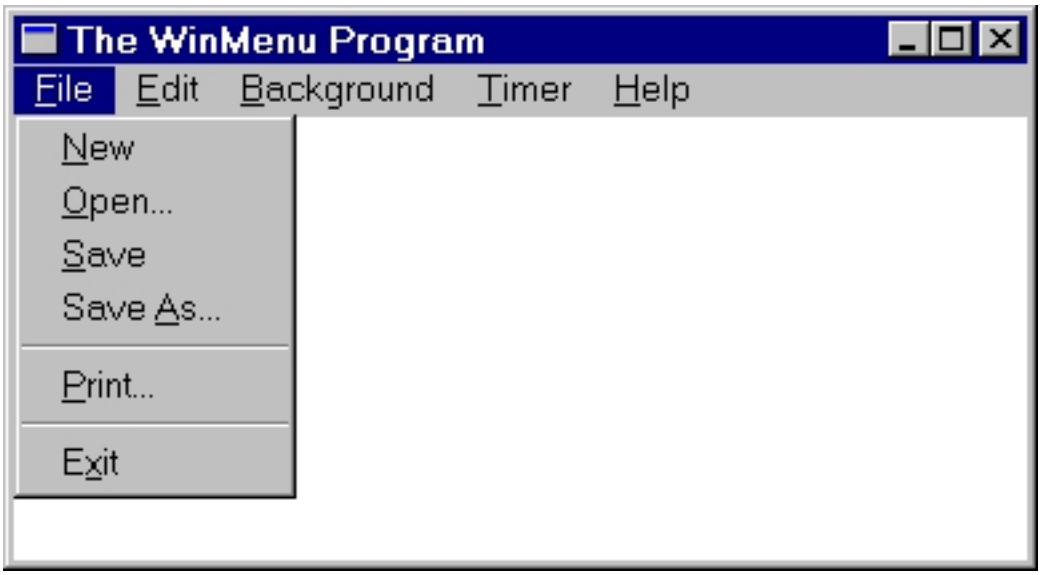
Index **587**

Ch Graphics Gallery

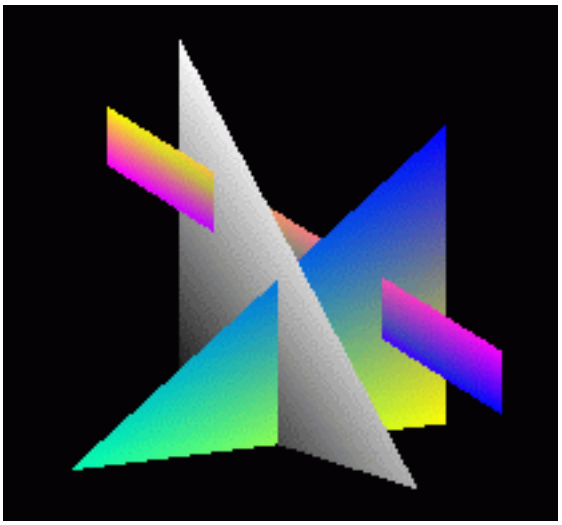
Plotting



Graphical User Interface



Graphics and Animation



Introduction

What is Ch?

Ch is C+. Ch is an embeddable C/C++ interpreter. It is an interpretive implementation of C with salient features from C++, other languages and software packages for scripting, rapid application development, deployment, and integration with legacy systems. Ch is designed for both experienced C/C++ programmers and new comers. Leveraging their C language skills, programmers can learn C once, and use it anywhere for any programming purpose.

Ch is embeddable. Unlike C/C++ compilers, Ch can be embedded as a scripting engine in C/C++ applications and hardware. It can relieve users from developing and maintaining a macro language or interpreter for many applications.

Ch is for 2D/3D graphical plotting and numerical computing. It is especially designed for applications in engineering and science. Ch has built-in graphical support, generic mathematic functions and computational arrays for linear algebra and matrix computations, 2D/3D graphic plotting, and advanced high-level numerical functions for linear systems, differential equation solving, integration, non-linear equations, curve fitting, Fourier analysis, etc. For example, linear system equation $\mathbf{b} = A * \mathbf{x}$ can be written verbatim in Ch. The user does not need to worry about the underlying optimization with fast and accurate numerical algorithms. Ch is the only computing environment in existence that can perform numerical computing with consistent numerical results under the IEEE floating-point arithmetic in the entire real domain and complex domain using an extended complex plane for a Riemman sphere. The extensions to C makes Ch an ideal choice for numerical computing in C/C++ domain.

Ch is for shell programming. Ch is a C-compatible shell whereas the so-called C-shell *csh* is a C-like shell. Ch is a very high-level language (VHLL) environment. Ch in Windows supports frequently used Unix utilities and commands such as *vi, ls, awk, sed, mv,rm, cp, find, grep,* etc. for cross platform shell programming. It can be used to automate repetitive tasks. Some complicated problems, which might take thousand of lines of C code, can be solved in a few lines of Ch code. The interactive Ch command shell is especially suitable for rapid prototyping, teaching, and learning.

Ch has borrowed features and ideas from many other languages and software packages. Ch owes its most to C/C++. The following is a short list of other languages and software packages which in one way or another have influenced the development of Ch.

- Like C shell, Ch can be used as a login shell and for shell programming. But, as a superset of C, Ch is a genuine C shell.
- Like Basic, Ch is designed for and has been used by beginners with limited computer experience.
- Like Perl, Ch can be used for common gateway interface (CGI) in a Web server.
- Like Java, Ch can be used for internet computing. A Ch applet can be executed across a network on different computer platforms on the fly.
- Like JavaScript, Ch scripts can be embedded in HTML files such as active server pages (ASP).
- Like Fortran 77/90, Ch can be used for scientific computing.
- Like MATLAB/Mathematica, Ch can be used for rapid prototyping.

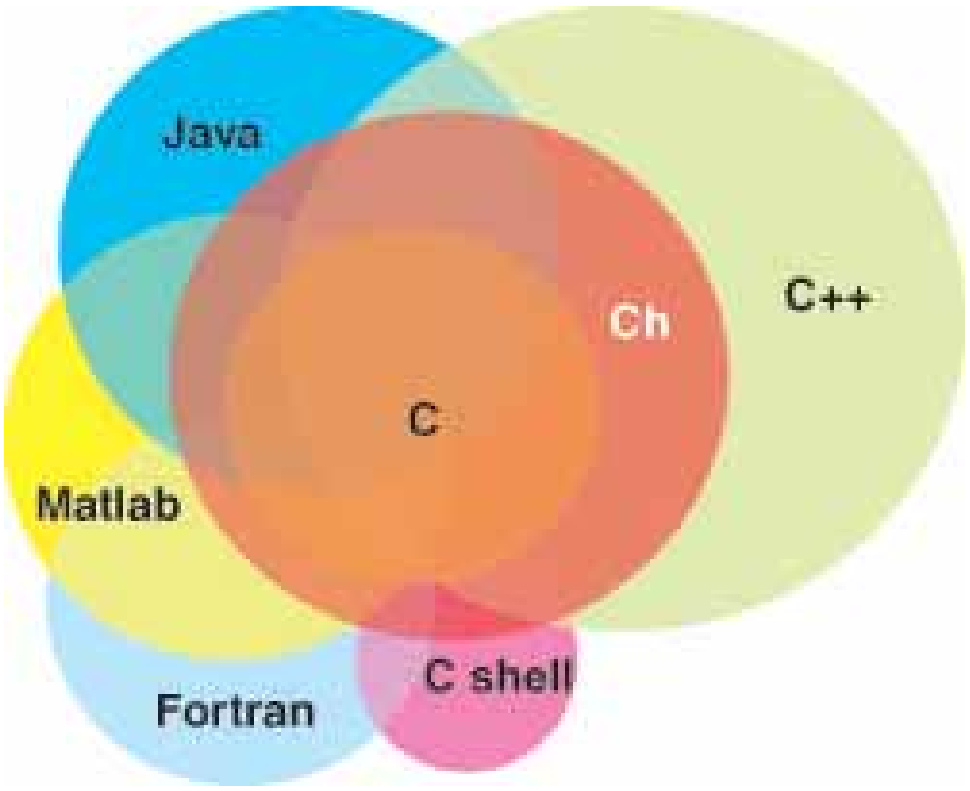


Figure 1: Relation of Ch with some other languages and software packages.

The relation of Ch with some of these languages and software packages is shown in Figure 1.

Major Features

Ch supports all features in the ISO 1990 C standard (C90), wide characters in Addendum 1 for C90, major new features in the latest ISO 1999 C standard (C99) including complex numbers, variable length arrays (VLAs), IEEE 754 floating-point arithmetic, generic functions. Ch supports classes, objects, and encapsulation in C++ for object-based programming, as well as many computer industry standards such as POSIX and socket/Winsock, Windows, X11/Motif, OpenGL, ODBC, GTK+. Ch has many extensions to C. Major features of Ch are summarized as follows:

- *No Learning Curve* Every C programmer can start to use Ch by executing C code in a Ch virtual machine without learning a new language. One can use the features of the C language to complete all tasks. It minimizes the hassles of learning and memorizing many different language syntaxes.
- *Interpretive* C programs can be executed in Ch without tedious compile/link/execute/debug cycles.
- *Interactive* One can run the C code interactively, entering the code line by line. Thus, it is very intuitive for beginners to learn C. It is a very effective tool to teach and learn programming in C with the latest C99 features. Also, one can easily test new functions. It is an ideal environment for real-time interactive computing.

- *Numerical Computing* In addition to supporting all C types such as char, int, float, double, and the new type complex and variable length array (VLA) as introduced in ISO C99, Ch treats a computational array as a first-class object. Many high-level numerical functions, such as differential equation solving, integration, Fourier analysis, along with 2D/3D plotting make Ch a very powerful language environment for solving problems in engineering and science. Programs using 2D/3D plotting features can also be compiled in C++ compilers using SoftIntegration Graphical Library in C++.

- *Very High-Level Language* Ch bridges the gap between low-level languages and very high-level languages (VHLL). As a superset of C, Ch retains low-level features of C such as accessing memory for hardware interface. As a command shell, Ch is a very high-level language. Some problems, which might take thousands of lines of C code, can be easily solved with only a few lines of Ch code.

- *Object-Based* Ch supports classes, objects, and encapsulation in C++ for object-based programming with data abstraction and information hiding, as well as simplified I/O handling. For example, only a single control class is used to implement Ch Control System Toolkit for high-level control system design and analysis. To keep Ch simple, complicated features in C++ are excluded in Ch.

- *Text Handling* Ch has advanced text handling features such as built-in string data type and **foreach**-loop. These features are specially useful for system administration, shell programming, and Web-based applications.

- *Cross platform Shell* Ch provides a universal shell for the convenience of users. It can be used as a login command shell similar to C-Shell, Bourne shell, Bash, tcsh, or Korn shell in Unix, as well as MS-DOS shell in windows. Ch has more built-in enhanced features for shell programming to automate repetitive tasks, rapid prototyping, regression test, and system administration across different platforms.

- *Safe Network Computing* Safe Ch is designed from scratch with different secure layers, such as sandbox, programmer/administrative control, suppressed pointers, restricted functions, automatic memory management for string type, and auto array bound checking effectively address security problems for network computing.

- *Portable* C standard-conforming programs are portable. But the compilation process is platform-dependent. A Ch program can run across different platforms including Windows and Unix. A programmer can develop and maintain programs in one machine, deploy them in all platforms supported by Ch.

- *Libraries* All existing C libraries and modules can be part of the Ch libraries. Therefore, the potential of Ch libraries is almost unlimited. For example, Ch supports POSIX, TCP/IP socket, Winsock, Win32, X11/Motif, GTK+, OpenGL, ODBC, LAPACK, XML, NAG statistics library, Intel OpenCV for computer vision and image processing, National Instruments' NI-DAQ and NI-Motion, PCRE for regular expression, etc.

- *Interface with Binary Modules* Using Ch SDK, Ch can interface binary objects without restarting a new process. It can seamlessly integrate different components. A Ch program can call functions in a static or dynamic library for integration with legacy systems and existing C/C++ code. Vice versa, a function in a binary object can call a Ch function.

- *Web Enabled* With development modules, such as classes for Common Gateway Interface (CGI) for Web servers, Ch allows rapid development and deployment of Web-based applications and services.

- *Embeddable* Ch is embeddable. Embedded Ch can be embedded in other application programs, hardware and handheld devices. This will relieve users from developing and maintaining proprietary scripting languages across different platforms.

Organization of this Documentation

Ch contains all features of C. Chapter 1 gives a brief overview of Ch and on how to run C/C++ programs in the Ch language environment. Chapters 2, 5-10, 14, 17-18, 20 describe features in C. Chapters 12, 13, and 15 present new features added in C99 with IEEE floating-point arithmetic and type generic mathematical functions, complex numbers, and variable length arrays (VLA), respectively. Chapters 11 and 19 present features of reference types and classes available in C++, respectively. Like any C compiler, Ch also contains some unique features with different setup and configuration. Features described in Chapters 3 and 24 are related to setup and configuration of Ch as an interpreter. Two and three dimensional plotting capabilities described in Chapter 23 are available in both Ch and C++. Computational arrays described in Chapter 16 and safe Ch in Chapter 21 are available in Ch only. Based on computational arrays, advanced numerical functions in Chapter 24 are convenient for many applications in engineering and science.

Appendix A lists known problems and platform specific features. Appendix B lists implementation defined behaviors and highlights the extension of Ch over C. Appendix C compares the differences between Ch and C++. Ch is a portable command shell. Appendices D, E, and F, compare Ch with C shell, MATLAB, and Fortran, respectively. Appendix G lists commonly used commands for portable shell programming in Ch across different platforms.

Part I

The Language Features

Chapter 1

Getting Started

This chapter gives an overview of the Ch language environment.

1.1 Startup

1.1.1 Startup in Unix

You can type the command `ch` in any command shell to go into the Ch command shell.

Startup in Mac OS X x86

Once you have downloaded and installed the software, you can click the icon **Ch** on the dashboard on in the Application folder to get into the Ch command shell.

For Ch Professional or Student Edition, you can also click the icon for ChIDE on the dashboard on in the Application folder to luanch ChIDE. You can then click the icon **Ch** in the ChIDE to get into the Ch command shell.

Startup in Linux

In Linux, you can launch Ch by clicking the icon **Ch** under the entry `System Tools` in the startup menu. You can also click `Run Program` in the startup menu. Then, enter **ch** and check `Run in terminal` to lanch Ch.

For Ch Professional or Student Edition, you can also click the icon for ChIDE under the entry `Programming Tools` in the startup menu. You can then click the icon **Ch** in the ChIDE to get into the Ch command shell.

The command

```
ch -d
```

will create an icon for Ch on the desktop. If Ch is installed with a ChIDE, an icon for ChIDE will also be created on the d esktop.

Startup in Unix as a Login Shell

You can login to a Unix computer system through a terminal that may be directly wired to the system, or through a modem to the internet or a local area network. To log into the system, type your user name at the system login prompt, which will begin the execution of program *login*. The program displays the string

password: at the terminal and waits for you to type the password. Once you have typed your password, the program proceeds to verify your login name against the corresponding entry in the file */etc/passwd*. Similarly, your password will be checked. The file */etc/passwd* contains one line for each user of the system. The information in this line specifies, among other things, the login name, home directory, and the program to be executed when the user logs in. The program that will start up after the login process is specified in the entry after the last colon. If nothing follows the last colon, by default, the system will use the Bourne shell /bin/sh. For example, if file */etc/passwd* contains the following three lines for three users of the system: `harry, john`, and `marry`.

```
harry:x:121:15::/home/harry:/bin/ch
john:x:125:20::/home/john:/bin/csh
marry:x:130:25::/usr/data:/usr/data/bin/word_processor
```

The home directory for user `harry` is */home/harry*. When `harry` logs in the system, the Ch shell will start execution. The home directory for user `john` is */home/john*, `john` will get C shell when he logs in. When `marry` logs in the system, the program *word_processor* that may be a special-purpose word processing software package will be invoked.

You can remotely login to another workstation which acts as a client. However, your local workstation may refuse connection to the client; the remote client fails and displays an error message. A proper communication has to be established so that the client will be able to determine which server receives the output of the client. At the same time, your workstation's X server will allow the remote system to send the output. This is accomplished by setting the environment variable *DISPLAY* on the client and adding the client to the name list of remote systems on your workstation's X server by the command *xhost*. For example, if you login to the remote machine *mouse* from the local machine *cat* and want the output of *mouse* to be sent to *cat*, you should execute the command

cat> `xhost mouse`

on the local machine *cat* and execute the command

mouse> `putenv("DISPLAY=cat:0.0")`

on the remote machine *mouse*. If the machine *mouse* is often used remotely, you may want to put the command `putenv("DISPLAY=cat:0.0")` in the startup file *.chrc* in your home directory of the machine *mouse* and set the following alias in the startup file of the local machine *cat*

```
alias("mouse", "xhost mouse; rsh mouse");
```

Then, the command *mouse* will add the remote machine *mouse* to the list of the remote systems of the local X server and start the remote login process. The name of the host machine can be obtained by the command *hostname*.

If Ch is the login shell, you can readily use the Ch language environment. If not, you can type command `ch` at a terminal prompt to launch the Ch language environment.

If the user resizes a Window under xterm command shell in the X-Window system by dragging the window borders using a mouse, the command **resize** can be used to set terminal settings to the current xterm window size. Because the command **resize** does not recognize Ch as a command shell, the user may type function `_resize()` in a Ch shell to set the environment variables **COLUMNS** and **LINES** to the current xterm window sizes.

1.1.2 Startup in Windows

Once you have downloaded and installed the software, there are four ways to get into the Ch language environment. For example, to start Ch Standard Edition 6.3,

1. Click the icon **Ch Standard** on the Desktop screen to get into the regular Ch shell, similar to MS-DOS.

2. Click **Start->Programs->SoftIntegration Ch 6.3 Standard->Ch 6.3**.
3. Click **Start**, followed by **Run**, then type `ch.exe`.
4. Go to the MS-DOS prompt, and type `ch`.

For Ch Professional or Student Edition, you can click the icon for ChIDE on the Desktop screen to launch ChIDE.

1.2 Command Mode

When Ch is launched or a Ch program is executed, by default, it will execute the startup file .chrc in Unix or _chrc in Windows in the user's home directory if it exists. This startup file typically sets up search paths for commands, functions, header files, etc. In Windows, a startup file _chrc with default setup is created in the user's home directory during installation of Ch. However, there is no startup file in a user's home directory in Unix by default. The system administrator may add such a startup file in a user's home directory. However, the user can execute Ch with the option -d as follows

```
> ch -d
```

to copy a sample startup file from the directory CHHOME/config to the user's home directory if there is no startup file in the home directory yet. Note that CHHOME is not the string "CHHOME", instead it uses the file system path under which Ch is installed. In Linux, the above command will also create an icon for Ch on the desktop. If Ch is installed with a ChIDE, an icon for ChIDE will also be created on the desktop.

The Ch language environment can be introduced with a famous programming output statement

```
hello, world
```

that was popularized by Kernighan and Ritchie (1978). The level of difficulty in printing this statement along with other criteria is often used to judge the simplicity and friendliness of a language. Users with previous C or FORTRAN experience may remember that, to print this statement, one has to first go through compilation and link processes to get the executable object code, and then run the program to get the output. For a large program, the `make` utility may have to be used to maintain the program's integrity. No compilation and link processes are necessary for running a Ch program. Ch can be used interactively and provides a quick system response. As a specific example, the prompt of the screen in C-shell is shown below:

```
%
```

The output from the system, as shown in this system prompt, is displayed in italics. To invoke the Ch language environment, one types `ch` on the terminal keyboard. The screen will become:

```
>
```

This prompt indicates that the system is in the Ch language environment and is ready to accept the user's terminal keyboard input. Ch can also be set as the default shell in the file */etc/passwd* so that, whenever the user logs in, the Ch programming environment will be invoked automatically as shown in the previous section.

Any syntactically correct terminal input entered at the Ch command prompt will be executed. Following successful completion of the command(s), the Ch prompt > will be printed. If a command fails, an error message shall be printed.

At the Ch prompt >, any Unix commands such as `cd, ls`, and `pwd` can be executed. In this scenario, Ch is used as a Unix shell in the same manner as Bourne-shell, C-shell, or Korn-shell. For example, to print the current working directory, one can type `pwd`. Then, the screen may appear as follows:

```
> pwd
/usr/local/ch
>
```

where the input typed in from the terminal is in the typewriter font. In Ch, if there is any output from the system resulting from executing a command, it will be printed out. In this case, assume `/usr/local/ch` is the current working directory, it becomes the output from execution of the command `pwd`,

Because Ch is a superset of ISO C, it is more powerful than the conventional Unix shells. If an expression is typed in, it will be evaluated by Ch and the result will be printed out immediately. For example, if the expression `1+3*2` is typed in, the output will be 7. If the input is `8`, the output will also be `8`. Any valid Ch expressions can be evaluated in this command mode. Therefore, Ch can be used as a calculator by novice users. Command **help** can help new users of Ch getting started with some illustrative examples.

```
> help
(display messages ...)
>
```

The first lesson that a C programmer learns may be to use the standard I/O function **printf**() to get the output `hello, world`. Because Ch is a superset of C, the output can be obtained by the I/O function **printf**() as follows:

```
> printf("hello, world")
hello, world
>
```

All variables including system variables such as **_path** can be printed out using function **printf()** in C or **cout** in C++ syntax. In interactive command mode, one can just type a variable name to display the value of the variable. For example,

```
> int i
> i = 10
> i*i
100
> printf("\%d", i)
10
> cout << 2*i
20
>
```

The following four functions in Ch can be used to handle environment variables. The function **putenv()** can add an environment variable to the system. The function **getenv()** returns the value of a given environment variable. The function **remenv()** can remove an environment variable. The function **isenv()** can test if a symbol is an environmental variable. The interactive command execution below demonstrates their application.

```
> putenv("ENVVAR=value")
> getenv("ENVVAR")
value
> isenv("ENVVAR")
```

```
1
> remenv("ENVVAR")
> isenv("ENVVAR")
0
>
```

There are hundreds of commands along with their online documentation in the system. No one knows all of them. Every computer wizard has a small set of working tools that are used all the time, plus a vague idea of what else is out there. Appendix G gives a list of common commands grouped by their functions. Details about these commands as well as command line options can be found using the command `man` followed by the command in query.

1.3 Program Mode

1.3.1 Command Files

A C program can be executed without compilation in the Ch language environment. The command-line argument interface in Ch is C compatible. C programs are called *command files* or simply *commands* in Ch. A command file shall have both read and execute permissions. In Ch, a command file can be executed without compilation. For example, one can create a command file named `hello.c` using a text editor. If the program `hello.c` is as follows:

```
/* A simple program */
#include <stdio.h>
int main() {
    printf("hello, world\n");
    return 0;
}
```

One can type the command `hello.c` to get the output of `hello, world` as follows.

```
> hello.c
hello, world
>
```

In order to use a file as a command in Unix, it has to be executable. To make the program `hello.c` executable, the following command may need to be executed.

```
chmod +x hello.c
```

To run a command file in command mode, the file name must be a valid identifier in Ch or start with a relative or absolute directory path such as '`./`', '`../`', '`~/`', and '`/`'. For example, if we change the above file name from `hello.ch` to `20`, it becomes a number rather than an identifier.

```
> mv hello.ch 20
> 20
20
> ./20
hello, world
>
```

Many Integrated Development Environments (IDE) support Ch. For example, ChIDE can be used to edit, run, or debug a program as shown in Figure 1.1. The user interface of ChIDE can be displayed in over 30 different local languages such as German, French, Chinese, and Japanese.

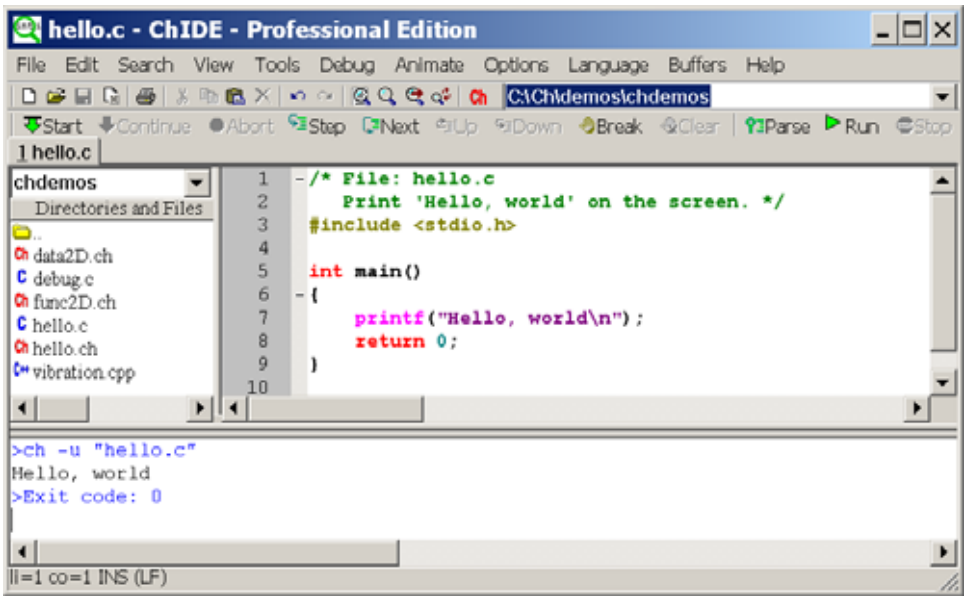


Figure 1.1: Edit and run a program using ChIDE.

1.3.2 Script Files

A program that can run in Ch, but cannot be compiled using a C or C++ compiler, is called a *script*. For example, a program without function main() or starting with `#!/bin/ch` is a script. Statements, functions, and commands can be grouped as a *script file* or *script* in Ch. Like a command file, a script file shall have both read and execute permissions. For example, if the script file `prog` contains the following statements:

```
#!/bin/ch
int i = 90;
/* copy hello.c to hello.ch */
cp hello.c hello.ch
printf("i is equal to %d from the script file\n", i);
```

it can be executed interactively as follows:

```
> prog
i is equal to 90 from the script file
>
```

Or, it can be executed in two separate steps as follows:

```
> chparse prog
> chrun
i is equal to 90 from the script file
>
```

where the command `chparse prog` parses the script file `prog` first, and the built-in command `chrun` then executes the parsed program. After execution of the script file `prog`, the file `hello.c` will be copied

to a Ch program named `hello.ch` by the statement `cp hello.c hello.ch`. With the file extension `.ch`, the program `hello.ch` can be executed in the Ch language environment as command `hello`. Program `prog` can be invoked by other scripting languages such as C-shell or Korn shell.

1.3.3 Function Files

A Ch program can be divided into many separate files. Each file can include many related functions at the top level that are accessible to any part of the program. A file that contains more than one function is usually suffixed with `.ch` to identify itself as part of a Ch program. Besides command files and script files, there are *function files* in Ch. A *function file* in Ch is a program that contains only one function definition. A function file shall be readable. By default, the extension of a function file is .chf. The names of the function file and function definition inside the function file shall be the same. The functions defined using function files are treated as if they were the system built-in functions in a Ch programming environment. For example, if the program `addition.chf` contains the following statements:

```
int addition(int a, int b) {
  int c;
  c = a + b;
  return c;
}
```

it can be invoked automatically to add two integers as shown in the following interactive execution session:

```
> int i = 9
> i = addition(3, i)
12
>
```

Integer value 3 and integer variable i with the value of 9 are added together by the function `addition()` first, the result is then assigned to variable i. In this case, the function `addition()` is treated as if it was a built-in function like **sin**() or **cos**().

1.4 Complex Numbers

A second order polynomial equation

$$ax^2 + bx + c = 0$$

can be solved by the formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}. \tag{1.1}$$

According to the formula (1.1), two real solutions ($x_1 = 2$) and ($x_2 = 3$) for equation

$$x^2 - 5x + 6 = 0$$

They can be obtained by Program 1.1.
Program 1.1 gives the following output:

```
x1 = 3.000000
x2 = 2.000000
```

```
#include <stdio.h>
#include <math.h>

int main() {
    double a = 1, b = -5, c = 6, x1, x2;
    x1 = (-b +sqrt(b*b-4*a*c))/(2*a);
    x2 = (-b -sqrt(b*b-4*a*c))/(2*a);
    printf("x1 = %f\n", x1);
    printf("x2 = %f\n", x2);
}
```

Program 1.1: The solution for $x^2 - 5x + 6 = 0$.

For equation

$$x^2 - 4x + 13 = 0,$$

two complex solutions ($x_1 = 2+i3$) and ($x_2 = 2-i3$) exist. These complex numbers cannot be represented in the double data type. Attempting to solve the equation in the real domain will result in invalid results. Ch reports the invalid results as NaN that stands for Not-a-Number. Refer to Program 1.2 and its corresponding output.

```
#include <stdio.h>
#include <math.h>

int main() {
    double a = 1, b = -4, c = 13, x1, x2;
    x1 = (-b +sqrt(b*b-4*a*c))/(2*a);
    x2 = (-b -sqrt(b*b-4*a*c))/(2*a);
    printf("x1 = %f\n", x1);
    printf("x2 = %f\n", x2);
}
```

Program 1.2: The solution for $x^2 - 4x + 13 = 0$ in the real domain.

Program 1.2 gives the following output:

```
x1 = NaN
x2 = NaN
```

Using complex numbers, equation

$$x^2 - 4x + 13 = 0$$

with two complex solutions of $x_1 = 2 + i3$ and $x_2 = 2 - i3$ can be solved by Program 1.3.
Program 1.3. gives the following output.

```
x1 = complex(2.000000,3.000000)
x2 = complex(2.000000,-3.000000)
```

1.5 Computational Arrays

Arrays in Ch are ISO C compatible. They are intimately tied with pointers. For numerical computing and data analysis, computational arrays are introduced in Ch, available in Ch Professional and Student Edition.

```
#include <stdio.h>
#include <math.h>
#include <complex.h>

int main() {
    double complex a = 1, b = -4, c = 13, x1, x2;
    x1 = (-b +sqrt(b*b-4*a*c))/(2*a);
    x2 = (-b -sqrt(b*b-4*a*c))/(2*a);
    printf("x1 = %f\n", x1);
    printf("x2 = %f\n", x2);
}
```

Program 1.3: The solution for $x^2 - 4x + 13 = 0$ in the complex domain.

Computational arrays can be handled as a first-class object in Ch. For example, the following array formula

$$\mathbf{x} = \mathbf{Ab} + \mathbf{3b} \tag{1.2}$$

with

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 2 \\ 4 & 4 & 6 \\ 7 & 8 & 9 \end{bmatrix} \text{ and } \mathbf{b} = \begin{bmatrix} 5 \\ 6 \\ 8 \end{bmatrix}$$

can be calculated by Program 1.4. The array expression **Ab+3b** is computed using three different methods. Execution of Program 1.4 gives the following output.

```
x = 48.000 110.000 179.000
x = 48.000 110.000 179.000
x = 48.000 110.000 179.000
```

In Program 1.4, arrays `A, b` and `x` are declared as computational arrays of double data type. The macro `array` of type qualifier for computational arrays is defined in header file **array.h**. The program includes this header file to use computational arrays. The values for arrays `A` and `b` are initialized at declaration. The value for array `x` is first calculated in the function `main()`. Next, it is calculated in function `arrayexp1()` and with the result passed back to the main program through a function argument. Finally, it is calculated by function `arrayexp()` which returns a computational array of double data type with three elements. Ch can also handle variable length arrays (VLAs) of C arrays and computational arrays very conveniently. Details about VLAs will be described in later chapters.

For example, the following linear system of equations

$$\mathbf{Ax} = \mathbf{b} \tag{1.3}$$

with

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 2 \\ 4 & 4 & 6 \\ 7 & 8 & 9 \end{bmatrix} \text{ and } \mathbf{b} = \begin{bmatrix} 5 \\ 6 \\ 8 \end{bmatrix}$$

can be solved by Program 1.5. In Program 1.5, function `inverse()` returns a computational array. The program gives the following output.

```
x = -5.000 2.000 3.000
```

```
#include <stdio.h>
#include <array.h>
void arrayexp1(array double A[3][3], array double b[3], array double x[3]) {
  x = A*b+3*b;
}

array double arrayexp2(array double A[3][3], array double b[3])[3] {
  array double x[3];
  x = A*b+3*b;
  return x;
}

int main() {
  array double A[3][3] = {{1,2,2},
                          {4,4,6},
                          {7,8,9}};
  array double b[3] = {5,6,8}, x[3];
  x = A*b+3*b;
  printf("x = %.3f", x);

  arrayexp1(A, b, x);
  printf("x = %.3f", x);

  x = arrayexp2(A, b);
  printf("x = %.3f", x);
}
```

Program 1.4: Calculation of array expression **Ab+3b**.

```
#include <stdio.h>
#include <numeric.h>

int main() {
  array double A[3][3] = {{1,2,2},
                          {4,4,6},
                          {7,8,9}};
  array double b[3] = {5,6,8}, x[3];
  linsolve(x, A, b);
  // or x = inverse(A)*b;
  printf("x = %.3f\n", x);
}
```

Program 1.5: Solution for Ax=b.

```
#include <math.h>
#include <chplot.h>

int main() {
    array double x[100], y[100];       // Use 100 data points
    char *title="sine wave",           // Define labels
         *xlabel="radian",
         *ylabel="amplitude";

    lindata(-M_PI, M_PI, x);          // X-axis data
    y = sin(x);                        // Y-axis data
    plotxy(x,y,title,xlabel,ylabel);   // Call plotting function
}
```

Program 1.6: Plot function $\sin(x)$ with $-\pi < x < \pi$.

1.6 Plotting

A convenient plotting library is available in Ch Professional and Student Edition. Program 1.6 plots function $\sin(x)$ with x in the range of $-\pi < x < \pi$. The output from Program 1.6 is shown in Figure 1.2.

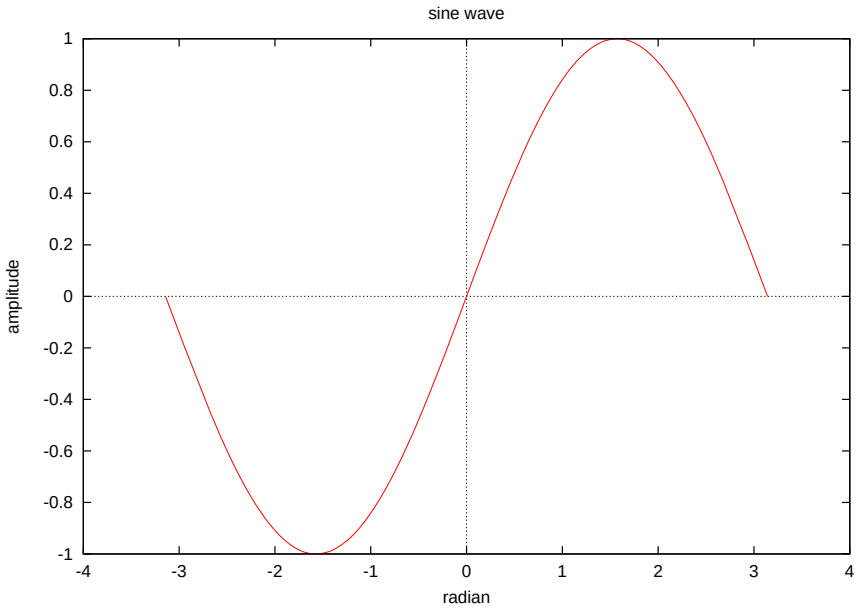


Figure 1.2: Plot for function $\sin(x)$ with $-\pi < x < \pi$.

Chapter 2

Lexical Elements

A Ch source file is a sequence of characters selected from a character set. A token is the minimal lexical element of the language. The categories of tokens are: keywords, identifiers, constants, string literals, and punctuators. Constants and string literals are described in Chapter 6.

2.1 Character Set

The character set used in Ch includes the following members: the 26 uppercase letters of the Latin alphabet

```
A  B  C  D  E  F  G  H  I  J  K  L  M
N  O  P  Q  R  S  T  U  V  W  X  Y  Z
```

the 26 lowercase letters of the Latin alphabet

```
a  b  c  d  e  f  g  h  i  j  k  l  m
n  o  p  q  r  s  t  u  v  w  x  y  z
```

the 10 decimal digits

```
0  1  2  3  4  5  6  7  8  9
```

the following 31 graphic characters

```
!  "  #  %  &  ’  (  )  *  +  ,  -  .  /  :
;  <  =  >  ?  [  \  ]  ^  _  {  |  }  ~ $ ‘
```

the space character, and control characters representing horizontal tab, vertical tab, and form feed, as well as control characters representing alert, backspace, carriage return, and new line. The graphic characters $ (dollar sign) and ‘ (accent grave or back quotation marks) are not part of the C standard. The dollar sign $ is used as an event designator in command mode, and variable substitution in both command mode and programs. The accent grave ‘ is used for command substitution.

2.1.1 Trigraphs

In Ch, all occurrences in a source file of the following sequences of three characters (called *trigraph sequences*)), which all begin with two consecutive question mark characters, are replaced with the corresponding single character shown below, so that the users of Ch can write Ch programs using the ISO 646-1083 Invariant Code Set.

```
??=     #       ??)     ]       ??!     |
??(     [       ??'     ^       ??>     }
??/     \       ??<     {       ??-     ~
```

No other trigraph sequences exist. Each **?** that does not begin one of the trigraphs listed above is not changed.

To prevent interpretation of the sequence of three characters listed above to its replaced character, use the character escape code `'\?'`. For example,

```
> printf("??!")
|
> printf("?\?!")
??!
>
```

the string `"?\?!"`, which includes the character escape code `'\?'`, can be used to represent the string `"??!"`, whereas the trigraph form `"??!"` represents the character `|`. Escape codes are described in section 6.3.1.

2.2 Keywords

2.2.1 Keywords

Following symbols are default keywords in Ch. Their semantics in a program follow the interpretation of the C standard and Unix/C convention.

Language Syntax Keywords

ComplexInf ComplexNaN Inf NaN NULL auto break const complex char case continue class double default delete do else enum extern float for foreach fprintf goto if inline int long new operator printf private public register restrict return scanf static struct short signed sizeof string_t switch this union unsigned volatile void while

Keywords from C++ The semantics of the following keywords are the same as those in C++.
class delete new private public this

Keywords not in C/C++ The following additional keywords have been added in Ch.
ComplexInf ComplexNaN Inf NaN NULL foreach fprintf printf scanf string_t

The symbol **NaN** stands for Not-a-Number. **Inf** represents the mathematical value ∞ of infinity. Complex Not-a-Number and complex infinity under a Riemman sphere are represented by **ComplexNaN** and **ComplexInf**, respectively. The keyword **NULL** in Ch resolves the problem of inconsistent use of macro NULL for pointer value `(void *)0` and integral value of 0 in C. NULL in Ch has the value of `(void*)0` when it is is used as pointer type and the value of 0 when it is used as an integral type.

The semantics for the standard functions **fprintf, printf**, and **scanf** defined in header file **stdio.h** in C are retained in Ch. But **fprintf, printf**, and **scanf** are extended in Ch as described in Chapter 20.

The new built-in data type **string_t** of the first-class object is added to solve memory problems related to strings of characters in C. The keyword **foreach** is added for foreach-loop construction. It is mainly used to handle loops with an index of string type. More information about these two keywords is given in Chapter 17.

Generic Functions

The generic functions available in Ch are listed below.

abs access acos acosh alias asin asinh atan atan2 atanh atexit ceil clock conj cos cosh dlerror dlopen dlrunfun dlsym exp elementtype fgets floor fmod fprintf fread free frexp fscanf getenv gets imag ioctl ldexp log log10 max memcpy memmove memset min modf open polar pow printf read real scanf setrlimit shape sin sinh sprintf sqrt sscanf stradd strcat strchr strcmp strcoll strcpy strerror streval strlen strncat strncpy strparse strtod strtok strtol strtoul strxfrm tan tanh transpose umask vprintf vfprintf vsprintf

A generic function is a built-in system function. It is an extension of the standard C function. Most generic functions are polymorphic. For example, the function call sin(x) uses the built-in system function so that argument x can be any valid data type for function sin(). For example, the following code is valid.

```
#include <array.h> // for the macro array

int i;
float f;
double df;
complex z;
double complex dz;
array double a[2][3];
array double complex az[2][3];
...
f = sin(i);
f = sin(f);
df = sin(df);
z = sin(z);
dz = sin(dz);
a = sin(a);
az = sin(az);
```

Note that the return type in function call of `sin(i)` is different from the argument type. Details about generic functions are further described in section 10.12.

The function **iskey()** defined in header file **chshell.h** can be used to determine if a name is a keyword in Ch. If the argument is not a keyword, 0 is returned; if the argument is the name of a generic function, 1 is returned; and if the argument is a keyword or a reserved symbol, 2 is returned. For example

```
> iskey("abcde")
0
> iskey("abs")
1
> iskey("while")
2
>
```

2.2.2 Reserved Symbols

The following symbols are reserved for possible future extension for features of inheritance and exception handling in C++.

virtual protected try catch

The following symbols are reserved for possible future extension of multi-tasking.

event_t recvevent sendevent beginparalleltask endparalleltask

2.3 Identifiers

An identifier is a sequence of characters that forms the name of a variable, class, type, function, etc. An identifier can consist of the underscore character, the lower case and upper case Latin letters, numerical characters and other characters. The dollar sign character `'$'` cannot be used in an identifier. Lower case and upper case letters are distinct. The first character must not be a number. The maximum length of an identifier is 5119. When preprocessing tokens are converted to tokens, if a preprocessing token could be converted to either a keyword or an identifier, it is converted to a keyword.

2.3.1 Predefined Identifiers

The identifiers listed in Table 2.1 are predefined in Ch. The default values of these predefined identifiers are given in Table 2.2. The major constraints are listed below.

- The delimiters for entries in **_ipath**, **_fpath**, and **_lpath _path**, are `";:"` for Unix and `";"` for Windows, respectively. A space can be used as part of a directory path in both Unix and Windows. Details about these system variables will be described in the next chapter.
- When system variables **_cwd**, **_cwdn**, **_home**, **_lang**, **_lc_all**, **_lc_collate**, **_lc_ctype**, **_lc_monetary**, **_lc_numeric**, **_lc_time**, **_logname**, **_path**, **_shell**, **_term**, **_tz**, **_user** are updated, the corresponding environment variables **HOME**, **LANG**, **LC_ALL**, **LC_COLLATE**, **LC_CTYPE**, **LC_MONETARY**, **LC_NUMERIC**, **LC_TIME**, **LOGNAME**, **PATH**, **PWD**, **SHELL**, **TERM**, **TZ**, **USER** will also be updated.
- **CHHOME** in the path names is not the string "CHHOME", instead it represents the file system path under which Ch is installed. For instance, use `C:\Ch` for **CHHOME** in Windows and `/usr/local/ch` for **CHHOME** in Unix. Similarly, **WINDIR** and **SYSTEMDIR** in path names are the values of system variables **WINDIR** and **SYSTEMDIR**, respectively.

Table 2.1: Predefined identifiers.

Identifier	Data Type	Description
_argc	int	Equivalent to argc in main(int argc, char**argv).
_argv	char**	Equivalent to argv in main(int argc, char*argv[]).
__class__	char []	the class name inside a member function.
__class_func__	char []	the class and function names inside a member function.
_cwd	string_t	Current working directory.
_cwdn	string_t	Current working directory name.
_environ	char**	An array of pointers to C strings. Each array entry points to an environment string.
_errno	int	System call error number.
_formatf	string_t	the default output format for float.
_formatd	string_t	the default output format for double.
_fpath	string_t	Path for function files.
_fpathext	string_t	Function file name extension.
__func__	char []	the function name inside a function.
_histnum	string_t	History number of a command saved.
_histsize	int	Size of history of commands saved.
_home	string_t	Home directory.
_host	string_t	Host name of the computer.
_ignoreeof	int	If it is true, the shell ignores EOF from terminals. This protects against accidentally killing a Ch shell by typing a Ctrl-d.
_ignoretrigraph	int	If it is true, the shell ignores trigraphs.
_ipath	string_t	Path for header files with the preprocessing directive #included.
_lang	string_t	The name of the locale to use for locale categories when both **_lc_all** and the corresponding system variable (beginning with "_lc_") do not specify a locale.
_lc_all	string_t	The name of the locale to be used to override any values for locale categories specified by the setting of **_lang** or any system variable beginning with "_lc_".
_lc_collate	string_t	The name of the locale for collation information.
_lc_ctype	string_t	The name of the locale for character information.
_lc_monetary	string_t	The name of the locale containing monetary-related numeric editing information.
_lc_numeric	string_t	The name of the locale containing numeric editing (i.e., radix character) information.
_lc_time	string_t	The name of the locale for date/time formatting information.
_logname	string_t	The name of the initial working directory of the user from the user database.
_lpath	string_t	Path for searching dynamically loaded lib used in function **dlopen**(const char **pathname*, int *mode*)). If pathname does not contain an embedded /, path in **_lpath** will be searched first. Then, follow the search order of the native function call. For example, the environment variable **LD_LIBRARY_PATH** will be search in SunOS.
_new_handler	void (*)()	Pointer to user defined handler function for operator **new**.

Table 2.1: Predefined identifiers (Contd.).

Identifier	Data Type	Description
_path	string_t	Path for commands.
_pathext	string_t	Command name extension.
_ppath	string_t	Path for adding paths for **_fpath**, **_ipath**, **_ipath** used in **#pragma package** <packagename>
_prompt	string_t	Prompt for interactive Ch shell.
_setlocale	int	If it is true, function `setlocale(CL_ALL, "")` will be called to handle multi-byte functions in header files wchar.h and wctype.h.
_shell	string_t	Name of the shell in use.
_status	int	Exit value indicating the status of the executed command. 0 for successful execution, non-zero for failure.
_term	string_t	Terminal type.
_tz	string_t	Time zone information.
_user	string_t	User account name.
_warning	int	3 with all warning messages. 2 most warning messages. 1 serious warning messages only. 0 no warning message.

Table 2.2: Default values of predefined identifiers.

Identifier	Data Type	Default Values
_argc	int	Command dependent
_argv	char*[]	Command dependent
__class__	static const char []	" "
__class_func__	static const char []	" "
_cwd	string_t	Current working directory, if cwd is not available, **_cwd** uses the value of **_home**
_cwdn	string_t	Current working directory name
_environ	char**	An array of pointers to C strings. Each array entry points to an environment string
_errno	int	0
_formatf	string_t	`".2f"`
_formatd	string_t	`".4lf"`
_fpathext	string_t	"chf"
_fpath	string_t	`"CHHOME/lib/libc;CHHOME/lib/libch;"` `"CHHOME/lib/libopt;CHHOME/lib/libch/numeric;"` for regular Ch; `"CHHOME/lib/libc;CHHOME/lib/libch;"` `"CHHOME/lib/libch/numeric;"` for safe Ch
__func__	static const char []	" "
_histnum	string_t	"0" (changed internally as commands are processed)
_histsize	int	128
_home	string_t	value of environment variable HOME, if it is set. Otherwise, home directory for Unix and current_drive:/ or C:/ for Windows
_host	string_t	Host name of the computer
_ignoreeof	int	0
_ignoretrigraph	int	0
_ipath	string_t	`"CHHOME/include;CHHOME/toolkit/include;"`
_lang	string_t	"C"
_lc_all	string_t	NULL
_lc_collate	string_t	NULL
_lc_ctype	string_t	NULL
_lc_monetary	string_t	NULL
_lc_numeric	string_t	NULL
_lc_time	string_t	NULL
_logname	string_t	Name of the initial working directory
_lpath	string_t	`"CHHOME/lib/dl;CHHOME/toolkit/dl;"`
_new_handler	void (*)()	NULL

Table 2.2: Default values of predefined identifiers (Contd.).

Identifier	Data Type	Default Values

_path	string_t	in regular Ch
		`"CHHOME/bin/;CHHOME/sbin;`
		`CHHOME/toolkit/bin;CHHOME/toolkit/sbin;`
		`/bin;/usr/bin;/sbin;"` for Unix;
		`"CHHOME/bin/;CHHOME/sbin;`
		`CHHOME/toolkit/bin;CHHOME/toolkit/sbin;`
		`/bin;/usr/bin;/sbin;/usr/openwin/bin;"` for SunOS/Solaris;
		`"CHHOME/bin;CHHOME/sbin;`
		`CHHOME/toolkit/bin;CHHOME/toolkit/sbin;`
		`WINDIR;WINDIR/COMMAND;`
		`WINDIR/SYSTEMDIR;"` for Windows 95/98/ME;
		`"CHHOME/bin;CHHOME/sbin;`
		`CHHOME/toolkit/bin;CHHOME/toolkit/sbin;`
		`WINDIR; WINDIR/SYSTEMDIR;"` for Windows NT/2000/XP;
		in safe Ch
		`"CHHOME/sbin;CHHOME/toolkit/sbin;`
		for all diff OS.

_pathext	string_t	"" in Unix and ".com;.exe;.bat;.cmd" in Windows
_ppath	string_t	`"CHHOME/package;"`
_prompt	string_t	**stradd**(**_cwdn**,`"> "`) for regular user, **stradd**(**_cwdn**,`"# "`) for superuser
_setlocale	int	0
_shell	string_t	Name of the shell in use
_status	int	0
_term	string_t	The value of the environment variable **TERM**
_tz	string_t	Local time zone
_user	string_t	User account name
_warning	int	1

2.4 Punctuators

A punctuator is a symbol that has independent syntactic and semantic significance. Depending on context, it may specify an operation to be performed in which case it is known as an *operator*. An *operand* is an entity on which an operator acts. The following punctuators are valid in Ch:

```
! != # ## $ % %= & && &= * *= + ++ += - -- -=
-> . .* ./ / /= < << <<= <= = == > >= >> >>= ?
^ ^= ^^ " ' ‘ { | |= || } ~
```

2.5 Comments

There are two forms of comments in Ch. Comments of a Ch program can be enclosed within a pair of delimiters `/*` and `*/`. These two comment delimiters cannot be nested. Except within a character constant, a string literal, or a comment, the characters **/*** introduce a comment. The contents of a comment are examined only to identify multibyte characters and to find the characters ***/** that terminate it.

The symbol `//` in Ch will comment out a subsequent text terminated at the end of a line. A `//` can be used to comment out `/*` or `*/` and `/* */` can be used to comment out `//`. Except within a character constant, a string literal, or a comment, the characters // introduce a comment that includes all multibyte characters up to, but not including, the next new-line character. The contents of such a comment are examined only to identify multibyte characters and to find the terminating new-line character. For example,

```
"a//b"                  // four-character string literal
// */                   // comment, not syntax error
f = g/**//h;            // equivalent to f = g / h;
//\
i();                    // part of a two-line comment
/\
/ j();                  // part of a two-line comment
/*//*/ l();             // equivalent to l();
m = n//**/o
  + p;                  // equivalent to m = n + p;
```

These two companion methods provide a convenient mechanism to comment out a section of code that contains comments. When a comment does not start at the beginning of a line, the use of `//` is recommended. A combined use of preprocessor directives `#if, #elif, #else`, and `#endif` can also comment out a large section of code.

Comments cannot be used in the place of argument of command statements. For example,

```
> int i=2   // comment ok
> i*4       /* comment ok */
8
> ls        // comment bad
> cmd       /* comment bad */
```

In the above example, comments cannot be applied to command statements **ls** and **cmd**.

Chapter 3

Program Structure

3.1 Directories and Files in the Ch Home Directory

Directories and files in the Ch home directory are shown in Table 3.1. The information about the latest release is kept in file **CHHOME/release/release_Ch**, where **CHHOME** is the home directory for Ch. Note that **CHHOME** is the file system path under which Ch is installed, instead of the string "CHHOME". For instance, `C:\Ch` is used for **CHHOME** in Windows and `/usr/local/ch` for **CHHOME** in Unix.

Table 3.1: Directories and files in the Ch home directory.

Directory Name	Contents
README	Important information
bin	Executable binary files
config	Configuration files
demos	Demo programs
dl	Dynamically loaded libraries
docs	Documentation
extern	Interface with other languages and binary objects
include	Header files used in Ch
lib	Libraries
license	License information
package	Ch packages
sbin	Commands for safe Ch
toolkit	Toolkits
www	Web related programs

3.2 Startup

The user can get into the Ch language environment by clicking the Ch icon on the desktop in Windows to start a Ch command window, or typing commands below

```
ch      -------- for regular shell
ch -S   -------- for safe shell (the same as chs)
chs     -------- for safe shell
```

Table 3.2: Ch startup files.

Startup files in Windows	Descriptions
~/**_chrc**	Included by **CHHOME/config/chrc**.
~/**_chsrc**	Included by **CHHOME/config/chsrc**.
~/**_chlogin**	Included by **CHHOME/config/chlogin**.
~/**_chslogin**	Included by **CHHOME/config/chslogin**.
~/**_chlogout**	Read by login shells at logout.
Startup files in Unix	Descriptions
~**/.chrc**	Included by **CHHOME/config/chrc**. Read at the beginning of execution by regular shell.
~**/.chsrc**	Included by **CHHOME/config/chsrc**. Read at beginning of execution by safe shell.
CHHOME/config/chlogin	Read by lgoin shells after execution of chrc at login for regular shells.
~**/.chlogin**	Included by **CHHOME/config/chlogin**.
CHHOME/config/chslogin	Read by safe ch login shells after execution of chsrc login for safe shells.
~ **/.chslogin**	Incldued by **CHHOME/config/chslogin**.
~ **/.chlogout**	Read by login shells at logout.

in a command window of Windows or command shell of Unix. Assume the environment variable **CHHOME** is the top directory where Ch is installed. It can be `/usr/ch` in Unix or `C:\Ch` in Windows. Startup files in Table 3.2 are executed when the Ch language environment is invoked.

When first started, the Ch shell normally performs commands from **CHHOME/config/chrc** which includes the **.chrc** file in your home directory, provided that it is readable. If the shell is invoked with a name that starts with '-', as when started by the login program in Unix, the shell runs as a login shell. In this case, after executing commands from **CHHOME/config/chrc** which includes the **.chrc** file in your home directory, the shell executes commands from the **.chlogin** file in your home directory; the same permission checks as those for **.chrc** are applied to this file. Typically, the **.chlogin** file contains commands to specify the terminal type and environment.

As a login shell terminates, it performs commands from the **.chlogout** file in your home directory; the same permission checks as those for **.chrc** are applied to this file.

When Ch is started with -d option, it first checks if file **.chrc** exists in your home directory. If not, Ch will copy **CHHOME/config/.chrc** to your home directory.

When Ch is started with option -f for fast startup, files **CHHOME/config/chrc** and ~**/.chrc** are not executed.

The startup procedure for safe Ch shell is the same as that for regular shell. But, startup files **chsrc, .chsrc, .chslogin,** and **chslogout**, instead of **chrc, .chrc, .chlogin,** and **chlogout**, are used.

In Windows, startup files **_chrc** and **_chsrc**, instead of **.chrc** and **.chsrc**, for regular and safe Ch in your home directory will be used, respectively.

By default, the value for system variable **_fpath** for the paths of function files is "`CHHOME/lib/libc`; `CHHOME/lib/libch`; `CHHOME/lib/libopt`; `CHHOME/lib/libch/numeric`" for regular Ch and "`CHHOME/lib/libc`; `CHHOME/lib/libch`; `CHHOME/lib/libch/numeric`" for safe Ch, respectively. Functions defined in function files not located in the above default directories cannot be used in startup files **.chrc, .chsrc, chrc**, and **chsrc.** But, generic functions can be used in the startup files.

```
umask(0022);
_warning = 3;     // print all warning. default is 1 with serious warning message only
_format = 8;      // output format for double "%.'format'lf" and float "%.'format-2'f"
_ignoreeof = 1;   // ignore EOF. defalut is 0
_path = stradd(_path, ".;");
//_ppath = stradd(_ppath, "/my/package/path;");
//_fpath = stradd(_fpath, "/my/function/path;");
//_ipath = stradd(_ipath, "/my/headerfile/path;");
//_lpath = stradd(_ipath, "/my/dynloadlib/path;");
//_pathext = stradd(_pathext, ";.ch");

#define RLIMIT_CORE  4
struct rlimit {int rlim_cur, rlim_max;} rl={0,0};
setrlimit(RLIMIT_CORE, &rl); /* no core dump */

if(_prompt != NULL) { // change the default prompt "cwdn> "
   _prompt = stradd(_user, "@", _host, ":", _cwd, _histnum, "> ");
}
putenv("TERM=xterm");
alias("rm", "rm -i");
alias("mv", "mv -i");
alias("cp", "cp -i");
alias("ls", "ls -F");
alias("go", "cd /very/long/dir");
alias("opentgz", "gzip -cd _argv[1] | tar -xvf -");
```

Program 3.1: Example of the startup file **.chrc**.

3.2.1 Sample Startup Files

Samples of startup files can be found in the directory `CHHOME/config`. After installation of Ch, the system administrator can modify these startup files according to different system configurations. Users can customize their individual startup files in the home directories. For the convenience of users, a sample of the startup file will be copied from the directory `CHHOME/config` to the user's home directory by command `ch -d`.

Program 3.1 is an example of the startup file **.chrc** the user's home directory in Unix. In this example, function **umask**(0*lmn*) allows the user to specify permission settings for new files or directories. The first digit '0' in the parameter indicates an octal number. The subsequent three digits *lmn* represent a three-number octal code used as summing access code for each access group. The most left number *l* is for the owner, the second number *m* for the group, and *n* for everyone else. Read access is 4, write access is 2, execute or search access is 1. The function **umask()** is used to disable the unwanted access. The function call

```
umask(0022);
```

removes the write access for the group and others. The system variable **_warning** indicates how the shell displays the warning messages. The meanings of different values of **_warning** are defined in Table 2.1.

The statement

```
_warning = 3;
```

changed its value from 1, the default value, to 3 to display all warning messages.

The default output format for values of float and double are `".2f"` and `".4lf"`, respectively. These default formats can be changed by resetting the system variables **_formatf** and **_formatd**. The statements

```
_formatf = ".6f";
_formatd = ".6lf";
```

change the default output format for values of float and double to `".6f"` and `".6lf"`, respectively.

The statement

```
_ignoreeof = 1;
```

sets the system variable **_ignoreeof** to true. Therefore, the shell ignores **EOF** from terminals. This protects against accidentally killing a Ch shell by typing a `Ctrl-d`. The statement

```
_path = stradd(_path, ".;");
```

adds the current working directory into the system variable **_path**, so that Ch will search it for commands. By default, the above statement has been commented out in the startup file for Unix. To make files in the current directory executable from the command shell, the above statement shall be uncommented. Similarly, the subsequent commands in this example for the system variables of **_fpath**, **_lpath**, **_ipath**, and **_ppath** add directories to these variables. The system variable **_pathext** of string type contains file extension of commands. To invoke a Ch command, such as `prog.ch` without typing the file extension `.ch` explicitly, one may add the file extension `.ch` to the system variable **_pathext**. The meanings and default values of these system variables can be found in Table 2.1.

The C function **setrlimit()** can be used to control maximum resource consumption. The first argument of this function represents the resource to be controlled. For example, the resource `RLIMIT_CORE` indicates the maximum size of a core file in bytes. The second argument is the **rlimit** structure which represents the resource limits. The **rlim_cur** member of **rlimit** specifies the current or soft limit and the **rlim_max** member specifies the maximum or hard limit. Soft limits may be changed by a process to any value that is less than or equal to the hard limit. A process may lower its hard limit to any value that is greater than or equal to the soft limit. The code below

```
#define RLIMIT_CORE  4
struct rlimit {int rlim_cur, rlim_max;} rl={0,0};
setrlimit(RLIMIT_CORE, &rl)
```

changes both of the soft and hard limits of maximum size of a core file to 0 to prevent the creation of it. The system variable **_prompt** contains the symbol of the prompt for the interactive Ch shell. For the regular user, its default value is the result of command `stradd(_cwdn, "> ")`, i.e. the current working directory name and the symbol '`>`'. The statement

```
_prompt = stradd(_user, "@", _host, ":", _cwd, _histnum, "> ");
```

in Program 3.1 changes the default prompt to the string including the username, the symbol '`@`', the machine name, the current working directory, the command history number, and the symbol '`>`', for example, "`user@machine:/path/dir#>`".

The environment variables maintain the special information of the user's environment. The functions **putenv()** and **getenv()** can put and get the environment information. The statement

```
putenv("TERM=xterm");
```

changes the environment variable **TERM** to `xterm`, where **TERM** is an environment variable that indicates the type of terminal. Some applications, such as **vi**, use this variable to determine what type of terminal the user is using. The last part of this example is about the **alias** command. The **allias** command makes an abbreviation for a frequently used command or series of commands. For example, the command

```
alias("rm", "rm -i");
```

makes the command **rm** equilvalent to **rm -i**. Most commonly used Unix commands such as **rm, mv, cp, ls** are available in Ch for Windows. Ch also contains all MS-DOS commands. Because different commands are used in different operating systems, the startup files for Windows can be slightly different. For example, command `alias("del", "del /P")` can be setup for MS-DOS command **del** in Ch for Windows.

The alias

```
alias("go", "cd /very/long/dir");
```

allows the user to only type the command `go` for changing the current working directory to `/very/long/dir`. The alias

```
alias("opentgz", "gzip -cd _argv[1] | tar -xvf -");
```

can be used to decompress and untar an archive file with file extension `.tgz` or `.tar.gz`. The formal argument `_argv[1]` will be replaced by the actual argument in the typed command. For example, with this alias, the command

```
opentgz file.tar.gz
```

is equivalent to

```
gzip -cd file.tar.gz |tar -xvf -
```

More information about alias can be found in section 4.6.

If Ch is used as a login shell, the command `stty` in the startup file **.chlogin** in the user's home directory sets the terminal characteristics, such as the `erase` character making a backspace, `kill` character cancelling the current command line, `intr` character interrupting the current command, and `susp` character suspending the current command. In this example, the command

```
stty intr '^C' erase '^?' kill '^U' susp '^Z'
```

changes the interrupt character to `Ctrl-C`, the erase character to `Ctrl-H`, the kill character to `Ctrl-U` , and the suspend character to `Ctrl-Z`. The user can use the command `stty -a` to display all current settings.

3.2.2 Command Line Options

A non-interactive Ch shell can execute a command supplied as an argument on its command line with the syntax as follows:

```
ch [-Sacdfghinruw] [argument...]
```

Except for the following command line options, the remaining words from the command line are passed as arguments to the invoked command.

-S Safe shell. Many functions, such as **system()**, are not available for safe shell. Many generic functions are disabled after the execution of **CHHOME/config/chsrc** and **CHHOME/config/chlogin** in the case of the login shell. See Chapter 21 for more details.

-a Portable code such as applets. Platform-dependent functions in `CHHOME/lib/libopt` cannot be used.

- c Read commands from the first filename argument (which must be present and readable). Remaining arguments are passed as arguments to **_argv**. If the program is a Ch command with function **main**(*int argc, char *argv[]*), arguments will also be passed to `argv` of function **main()**.

- d When ch is started, it first checks if file **.chrc** exists in the user's home directory. If not, Ch will copy **CHHOME/config/.chrc** to the user's home directory. When chs is started, it first checks if file **.chsrc** exists in the user's home directory. If not, Ch will copy **CHHOME/config/.chsrc** to the user's home directory. In Windows, startup files **_chrc** and **_chsrc** instead of **.chrc** and **.chsrc**, will be used for regular Ch and safe Ch, respectively.

- f Fast start. Read neither the **chrc** and **.chrc** files, nor the **chlogin** and **.chlogin** files (if a login shell) upon startup.

- g For CGI script debug. It turns the Web browser into a text shell.

- h Display Ch usage message for help.

- i Reserved for forced interactive shell (ignored).

- n Parse (interpret), but do not execute commands. This option can be used to check Ch shell scripts for syntax errors. The warning flag for system variable **_warning** will be set to the highest level. All warning messages will be printed out. Start up files will be parsed only without execution.

- r Redirect `stderr` stream to `stdout`. This option is useful for debugging programs running in Windows operating systems. For example, command `ch -r chcmd > junkfile` will send error messages from `stderr` stream in program `chcmd` to file `junkfile`.

- u Unbuffer the stdout stream mainly for handling I/O in IDE.

- v Print out Ch edition and version number in the stdout stream.

- w The warning flag for system variable **_warning** will be set to the highest level for both parsing and execution of the program. All warning messages will be printed out.

Option `-a` can be used to test if a Ch program is portable across different platforms. For example, the command below will test if program `cmd.ch` is portable.

```
ch -a cmd.ch
```

Option `-g` is very useful for debugging CGI code. If a CGI script starting with the first line of

```
#!/bin/ch -g
```

the Web browser is turned into a text shell. All output including error messages from executing the CGI script will be displayed inside a Web browser.

3.3 Ch Programs

3.3.1 Command Files

A C program can be executed without compilation in a Ch language environment. The command-line argument interface in Ch is C compatible. C programs are called *command files* or simply *commands* in Ch. A file is identified as a Ch program if it has read/execute permission and starts with one of the following tokens:

1. Comment symbols `/*` or `//`
2. A type specifier, type qualifier, or storage-class specifier.
3. Symbol # followed by a preprocessor directive.
4. Symbol # followed by `!/bin/ch` or `!/bin/sch`
5. Identifier `main`.
6. Function name `printf`.
7. Dot '.', which is used for execution of the program in the current shell, when it is entered in a Ch shel prompt.

In a Ch programming environment, a command file can be executed without compilation. The system variable **_pathext** of string type contains file extension of commands. The default value of variable **_pathext** is `""` in Unix and `".com;.exe;.bat;.cmd"` in Windows. To invoke a Ch command with file extension .ch, such as `hello.ch` without typing the file extension `.ch` explicitly, one may add the file extension `.ch` to the system variable **_pathext** in the startup file **.chrc** in Unix or **_chrc** in Windows in the user's home directory. For example, if the `hello-world` program is saved in a file `hello.ch` and **_pathext** contains `.ch`, it can be executed as follows.

```
> hello
hello, world
>
```

A Ch program shall have both read and execute permissions for the user to execute it. The permission of a program can be changed by command `chmod`. For example, command

```
chmod 755 program.ch
```

will change program `program.ch` with read/write/execute permission for the owner of the program and read/execute permission for the group and others.

If a command name is preceded with a relative or absolute path, Ch will search for it in the specified path. Otherwise, Ch will search the paths specified in the system variable **_path**, one after another. The default value of **_path** is listed in Table 2.2. Generic function **stradd()** can be used to add paths into **_path**. For example, the command below adds the path `/home/mydir/bin` to the end of **_path**.

```
> _path = stradd(_path, "/home/mydir/bin;")
/usr/ch/bin/;/usr/ch/sbin;/usr/ch/toolkit/bin;
/usr/ch/toolkit/sbin;/bin;/usr/bin;/sbin;/home/mydir/bin;
>
```

If this path is to be automatically added each time when Ch is started, the command below

```
_path = stradd(_path, "/home/mydir/bin;")
```

should be added in the startup file, such as **.chrc** in Unix or **_chrc** in Window in the user's home directory. More information about how to customize the startup files can be found in section 3.2.

In both Unix and Windows, a path name in system variable **_path** may contain blank spaces, for example, `C:/Program Files/package`. The paths for dynamically loaded libraries with file extension `.dll` in Windows may also be added to **_path**.

Function **system()** can handle programs with file extension `.ch` just like they are included in **_pathext**. For example,

```
system("help.ch")
```

or

```
system("/usr/ch/bin/help.ch")
```

3.3.2 Script Files

The Ch language environment can recognize other shell scripts and programs. To be recognized by other shells and programs such as for WWW Common Gateway Interface, a Ch program shall start with

```
#!/bin/ch
```

followed by command line options such as `-S` for safe shell, and `-f` for fast start up. Although not recommended, the use of spaces before the sign # and after the sign ! are allowed. A program that can run in Ch, but cannot be compiled using a C or C++ compiler, is called a *script*. For example, a program without function **main()** or starting with `#!/bin/ch` is a script. It is treated in the same manner as a command. Inside a script , system variables **_argc** and **_argv** can be used for command line interface. These two command line interface variables are available even for a command file.

3.3.3 Function Files

A Ch program can be divided into many separate files. Each file consists of many related functions at the top level that are accessible to any part of a program. A file that contains more than one function is usually suffixed with `.ch` to identify itself as part of a Ch program. Besides command files and script files, there are *function files* in Ch. A *function file* in Ch is a program started with a function definition. A function file shall be readable. The extension of a function file is specified by the system variable **_fpathext** of string type. The default value of the system variable **_fpathext** is `".chf"`. The names of the function file and function definition inside the function file shall be the same. The functions defined using function files are treated as if they were the system built-in functions in Ch. For example, if program `addition.chf` contains the following statements,

```
/**** function file for adding two integers ****/
int addition(int a, int b) {
  int c;
  c = a + b;
  return c;
}
```

Function `addition()` can be invoked automatically to add two integers. It is suggested that, inside a function file, there is only one function definition which may nest many local functions. A program that invokes the function `addition()` from a function file can be prototyped as

```
extern  int addition(int a, int b);
```

This prototype for a function from a function file is optional in the program.

The preprocessing directive `#endif` described in Chapter 5 shall not fall after the closing parenthesis for the arguments of the function in a function file. For example, the following code is invalid.

```
int fun(int arg1,
#ifdef NeedWidePrototypes
int arg2,
double arg3) {
#else
char arg2,
float arg3) {
#endif
     /* ... */
}
```

Instead, it should be written as

```
int fun(int arg1,
#ifdef NeedWidePrototypes
int arg2,
double arg3
#else
char arg2,
float arg3
#endif
) {
     /* ... */
}
```

The included file before the function definition inside a function file will be processed first, before the function definition is parsed. For example, the following code is valid.

```
#include<stdio.h>
FILE *fopen(const char *filename, const char *type) {
  return _fopen(filename, type);
}
```

The preprocessing directives before the function definition of a function are ignored, when the function file is used as a function prototype for a program. These directives will be parsed when the function is processed at the end of the program. If a function in a function file is invoked in command mode at prompt, all directives except `#include` will be processed and the included header file will be parsed before the function prototype in the function file is used. Therefore, conditional preprocessing directives inside a function file are valid before the function definition, only when the function is used inside a program. Function `func()` defined in the following function file can be used in a program, but not in command mode at prompt.

```
#ifdef HEADER1
#include<header1.h>
#else
#include<header2.h>
#endif
int func() {
  ...
}
```

The function can be used in both program and command mode, if the above code is changed to

```
#include<header.h> // include hearer1.h and hearder2.h conditionally
int func() {
  ...
}
```

Ch will search for function files in the the paths specified in the system variable **_fpath**, one after another. The default value of **_fpath** is listed in Table 2.2. Additional paths for function files can be added to the system variable **_fpath**. For example, the command below adds the path `/home/mydir/lib` to the end of **_fpath**.

```
> _fpath = stradd(_fpath, "/home/mydir/lib;")
/usr/ch/lib/libc;/usr/ch/lib/libch;/usr/ch/lib/libopt;
/usr/ch/lib/libch/numeric;/home/mydir/lib;
>
```

If the system variable **_fpath** is modified in command mode, it will be effective only for functions invoked in the current shell interactively. The function search paths in the current shell will not be used and inherited in subshells. To make function files in this path available to the current Ch shell and all Ch programs, the command below

```
_fpath = stradd(_fpath, "/home/mydir/lib;")
```

should be added in the startup file **_chrc** in Windows or **.chrc** in Unix at the user's home directory. If the search paths for function files have not been properly setup, a warning message such as

```
WARNING: function 'addition()' not defined
```

will be displayed, when the function `addition()` is called.

When a function is called in command mode, the function file will be loaded. If you modify a function file after the function has been called, the subsequent calls in command mode will still use the old version of the function definition that had been loaded. To invoke the modified version of the new function file, you can either remove the function definition, say `addition`, in the system by command

```
> remvar addition
```

or start a new Ch shell.

A .chf file can contain multiple function and class definitions. A .chf file with multiple function and class definitions should not be treated as a function file. Rather, it should be loaded explicitly using a **pragma** directive. For example, the code below

```
#pragma importf <myfunc.chf>
#pragma importf <myclass.chf>
```

will load files `myfunc.chf` and `myclass.chf` with multiple function and class definitions located in a directory specified by the system variable **_fpath**. This **pragma** directive can be placed in a header file that may typically be included in applications.

3.4 Program Execution

The *program startup* occurs when a designated Ch program is invoked by the execution environment. The program is parsed to form an internal data structure first, then it is executed. All objects in static storage are *initialized* (set to their initial values) before program execution. *Program termination* returns control to the execution environment.

3.4.1 Execution of Programming Statements in Command Mode

At a Ch shell prompt, all expressions, programming statements and functions parsed and executed immediately. For example,

```
> int i
> for (i = 0; i < 3; i++) printf("i = %d\n", i)
i = 0
i = 1
i = 2
> int func1(int i){int j; j = i+i; return j;}
> i = func1(10)
20
> int func2(int i){int j; j = i*i;\
return j;}
> i = func2(i)+func1(1)
402
> 2*i
804
>
```

In the example above, the for-loop and definitions of functions `func1()` and `func2()` are typed at a shell prompt. The ending semicolons are not necessary at prompt. All programming statements have to be completed in one command line which may consist of multiple lines separated with a line continuation symbol '\' immediately followed by a carriage return character as shown for the definition of function `func2()`. Otherwise, Ch gives error messages. For example, if the for-loop in the previous example is broken into two lines, the result is wrong. If the definition of function is broken into more than one line, Ch treats it as a syntax error.

```
> int i
> for (i = 0; i < 3; i++)   // break the for-loop into two lines
> printf("i = %d", i)       // and the result is unexpected
i = 3
> int fun1(int i){int j;    // the definition of fun1() is broken
ERROR: missing '}'
WARNING: missing return statement for function fun1() and
default zero is used
>
```

3.4.2 Program Startup

A Ch program is normally executed according to the following sequences. First, a startup file **CHHOME/config/chrc** is executed. All values for global and system variables inside the startup file are retained for use in the current executed program. The program including all modules from so-called preprocess directives are then parsed to form an internal program tree. Each executable statement in the internal program tree is then executed. Finally, either function **main()** or **WinMain()** is executed, if it has been declared.

Function **main()** shall be defined with a return type of int and in one of the following forms, with no parameters:

```
int  main(void) { /* ... */ }
```

or with two parameters referred to here as `argc` and `argv`, though any names may be used, as they are local to the function in which they are declared:

```
int main(int argc, char *argv[]) { /* ... */ }
```

or

```
int main(int argc, char **argv[]) { /* ... */ }
```

Or with three parameters

```
int main(int argc, char *argv[], char **environ) { /* ... */ }
```

Function **WinMain()** in Windows shall be defined according to the Windows API as

```
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{ /* ... */}
```

If they are declared, the parameters to function **main()** will obey the following constraints:

- The value of `argc` is nonnegative.
- `argv[argc]` is a null pointer.
- If the value of `argc` is greater than zero, the array members `argv[0]` through `argv[argc-1]` inclusively contain pointers to strings.
- If the value of `argc` is greater than zero, the string pointed to by `argv[0]` represents the *program name*. If the value of `argc` is greater than one, the strings pointed to by `argv[1]` through `argv[argc-1]` represent the program parameters.
- The parameters `argc` and `argv` and the strings pointed to by the `argv` array shall be modifiable by the program, and retain their last-stored values between program startup and program termination.
- The parameter `environ` is a pointer to the table of environmental variables.

More information about these functions can be found in section 10.10. The constraints and values for system variables **_argc** and **_argv** are the same as parameters `argc` and `argv`, respectively. More information about these two system variables **_argc** and **_argv** are available in section 4.16.

3.4.3 Program Termination

The return type of function **main()** shall be a type compatible with int. A return from the initial call to function **main()** is equivalent to calling the **exit** function with the value returned by function **main()** as its argument. The status value is stored in system variable **_status**.

3.4.4 Search Order

Order of Things in a Program

For a given identifier, the Ch language environment will interpret it according to the following search sequence:

- Check if it is a defined macro.
- Check if it is a keyword.
- Check if it is a defined variable including variable of a function.
- Check if it is followed by an open parenthesis '('. If it is followed by an open parenthesis, attach the name with a file extension from a list of extensions in the system variable **_fpathext**. For each file extension, search each directory specified by the system variable **_fpath** for a function file until it is found.
- Check if it is a command in each directory specified by the system variable **_path**. Then, attach the name with a file extension from a list of extensions in the system variable **_pathext**. For each file extension, search each directory specified by the system variable **_path** for the executable and readable command until it is found.

Order of Things at Prompt

When an identifier is given in interactive mode at the prompt. It is first tested against the list of aliases. Then, follow the search sequence described in the previous section.

The Ch program **which** described in Chapter 4 can be used to tell how a given identifier is interpreted.

3.4.5 Running Programs with Multiple Files

In this section, running programs with mulitple files will be described. Handling of packages in Ch will be presented in section 22.3.

The file name of a Ch program is the command name. The extension of a command can be specified by the system variable **_pathext**. A program, consisting of multiple files, can be organized using `import` and `importf` following preprocessor directive `pragma` described in section 5.9. Unlike included header files which search the directory specified in system variable **_ipath**, the directories specified in system variables **_path** and **_fpath** are searched for the program following `import` and `importf`, respectively. In addition, a string can follow `import` and `importf`. In this case, the file will be searched in the current directory first. If the file pointed to by the string expression does not exist, the pragma statement shall be ignored. For example, assume command `command` consists of four separate files `command.c`, `module1.c`, `module2.c`, `module3.c`, then program `command.c` can be written as follows.

```
/* Program command.c */
#include <stdio.h>
int main() {
  int i =90;
  printf("main() program \n");
  /* ... main program goes here */
}
#pragma importf "module1.c" /* search for module1.c in current
```

```
            directory first then directories specified in _fpath */
#pragma import "module2.c"  /* search for module2.c in current
            directory first then directories specified in _path */
#pragma importf <module3.c> /* search for module3.c in directories
                                    specified in _fpath only */
```

Static variables in files `module1.c`, `module2.c`, and `module3.c` have file scope. Notice the difference between `import` and `importf` in this example. File `module1.c` is searched in the current working directory first, then in the directories specified in **_fpath**. File `module2.c` is searched in the current working directory first, then in the directories specified in **_path**, not in **_fpath**. File `module3.c` is searched only in the directories specified in **_fpath**. Command file `command.c` shall have read/execute permission, whereas files `module1.c, module2.c, module3.c` shall have read permission.

Alternatively, one can add a Ch command called `command.ch` without touching files `command.c`, `module1.c`, `module2.c`, and `module3.c` of the original C code.

```
#!/bin/ch
/* command.ch  */
#pragma import  "command.c"
#pragma importf "module1.c"
#pragma import  "module2.c"
#pragma importf <module3.c>
```

It is recommended that for a command with multiple files, create a directory to hold other files used by the command. For example, for command `xxx`, create directory `xxx_ch` to hold files invoked by the command `xxx`. Therefore, for command `command`, one can create a directory `command_ch` with its parent directory being part of search path in system variable **_path**. The command can be written as follows:

```
/* Program command.c */
#include <stdio.h>
int main() {
  int i =90;
  printf("main() program \n");
  /* ... main program goes here */
}
#pragma import <command_ch/module1.c> /* search for module1.c in
                              directories specified in _path only */
#pragma import <command_ch/module2.c>
#pragma import <command_ch/module3.c>
```

File `command` can then be used as an executable Ch command.

A static variable in a file included by `pragma` has file scope. This works fine in most cases. However, sometime in a C program, a static variable is declared in a header file which is included by different modules. Each module is compiled separately. This means a static variable needs to be accessed by all modules in the corresponding Ch program. For such cases, directive `include` can be used. The previous sample program can be written as follows.

```
/* Program command.c */
#include <stdio.h>
int main() {
  int i =90;
```

```
  printf("main() program \n");
  /* ... main program goes here */
}
#ifdef _CH_
#include "module1.c"  /* search for module1.c in current directory
                  first then directories specified in _ipath */
#include "module2.c"  /* search for module2.c in current directory
                  first then directories specified in _ipath */
#include <module3.c>  /* search for module3.c in directories
                         specified in _ipath only */
#endif
```

Similarly, one can add a Ch command called `command.ch` without touching files `command.c`, `module1.c`, `module2.c`, and `module3.c` of the original C code.

```
#!/bin/ch
/* command.ch  */
#include "module1.c"
#include "module2.c"
#include <module3.c>
```

A program, consisting of multiple files, can also be organized using a dot command which runs in the current shell. Unlike included header files, the directory specified in system variable **_path** is searched for the program following the dot. Similar to including files using preprocessing directive `include`, a static variable in a doc command is visible to all modules in the program. Using dot commands, the above sample program `command.c` can be written as follows.

```
/* Program command.c */
#include <stdio.h>
#ifdef _CH_
. "module1.c"  /* search for module1.c in current directory first
                  then directories specified in _ipath */
. "module2.c"  /* search for module2.c in current directory first
                  then directories specified in _ipath */
. <module3.c>  /* search for module3.c in directories
                  specified in _ipath only */
#endif
int main() {
  int i =90;
  printf("main() program \n");
  /* ... main program goes here */
}
```

If all these files are located in a directory, say, `/my/package/dir`, command *command* can be executed at different directories by changing the line

```
#ifdef _CH_
```

in the above code to

```
#ifdef _CH_ && strcat(_ipath,"/my/package/dir;") \
            && strcat(_path,"/my/package/dir;")
```

Similarly, one can add a Ch command called `command.ch` without touching files `command.c`, `module1.c`, `module2.c`, and `module3.c` of the original C code.

```
#!/bin/ch
/* command.ch  */
. "module1.c"
. "module2.c"
. <module3.c>
```

Chapter 22 describes the details on how to create library and software packages to run in Ch.

3.4.6 Debug Programs

To parse a Ch program without execution for checking the syntax error of the program, the shell command **chparse** followed by the file name can be used. After parsing, the program can be executed by typing shell command **chrun**. For example,

```
> chparse program.c
> chrun
>
```

If the program `hello.c` is as follows,

```
int main() {
  printf("hello, world\n";
}
```

the error of the program can be diagnosed by the command **chparse** as follows:

```
> chparse hello.c
ERROR: missing )
ERROR: Syntax error at line 2
>
```

where the missing parenthesis for the function **printf**() at line 2 is detected.

An entire program can be parsed first. Then it can be executed step by step interactively using the shell command **chdebug**. In the example below, `program.c` is listed by command `more` first. Then, run by command **chdebug**.

```
> more program.c
int main() {
   int i, *p;
   i = 10;
   p = &i;
}
> chdebug program.c

You are debugging file 'program.c'

Type    (1) expression for evaluation
        (2) 'run' to continue
```

```
        (3) hit return key to step to next line
        1:      int main() {

        2:          int i, *p;

        3:          i = 10;

        4:          p = &i;
i
=> 10
i*i
=> 100
&i
=> 1c21a0

        5:      }
p
=> 1c21a0
*p
=> 10

        1:      int main() {

>
```

In the debug mode, the user has three options: evaluating an expression, executing the program non-stop, or step-by-step execution of the program. In a step-by-step execution, the source code including the line number will be displayed before it is executed. When the user types in an expression for evaluation, the result of the expression will be displayed following the symbol `=>`. In this example, it shows that `i*i` is 100 and the address of variable `i` is the same as the value for pointer `p`. Details about pointers are described in Chapter 9.

Using commands **chparse-chrun** and **chdebug**, the program runs in the current shell. A program can be parsed to just check for syntax errors without being executed using option `-n` as shown below.

```
> ch -n program.c
>
```

In this case, the program runs in a subshell.

The macro `assert()` defined in header file **assert.h** can also be used to debug a program. One can setup a break point in a program by set adding the debugging function `_stop("Your debug message\n")` inside the program. The program will stop at this statement to wait for the user's input at the execution phase. The value of a variable or expression can be printed out by typing the name of variable or expression at the point where the program stops. At run time, the implicit pointer 'this' can also be used to access members of a class in member functions of the class when the program is executed in debugging mode.

In Windows, because the `stderr` stream is handled differently, one should use command line option `-r` to debug a Ch program. For example, command

```
ch -r program.c > junkfile
```

will send error messages from `stderr` stream to file `junkfile.`

3.5 Scope Rules

3.5.1 Scopes of Identifiers

An identifier can denote an object; a function; a tag or a member of a class, structure, union, or enumeration; a typedef name; a label name; a macro name; or a macro parameter. The same identifier can denote different entities at different points in the program. A member of an enumeration is called an *enumeration constant*. The macro names in the source file are replaced by the preprocessing token sequences that constitute their macro definitions during the passing phase.

For each different entity that an identifier designates, the identifier is *visible* (i.e., can be used) only within a region of program text called its *scope*. Different entities designated by the same identifier either have different scopes, or are in different name spaces. There are five kinds of scopes: function, file, block, and function prototype, program, system. A *function prototype* is a declaration of a function that declares the types of its parameters.

A label name is the only kind of identifier that has *function scope*. It can be used in a **goto** statement anywhere in the function in which it appears, and is declared implicitly by its syntactic appearance followed by a : and a statement.

Every other identifier has scope determined by the placement of its declaration (in a declarator or type specifier). If the declarator or type specifier that declares the identifier appears outside of any block, the identifier has *program scope*. If the identifier is declared outside of any block with storage-class qualifier **static** as a static variable, the identifier has *file scope*. If the identifier is declared with `__declspec(global)`, the identifier has *system scope* in the current Ch shell. An identifier in system scope can be accessed by multiple programs. If the declarator or type specifier that declares the identifier appears inside a block or within the list of parameter declarations in a function definition, the identifier has *block scope*, which terminates at the end of the associated block. If the declarator or type specifier that declares the identifier appears within the list of parameter declarations in a function prototype (not part of a function definition), the identifier has *function prototype scope*, which terminates at the end of the function declarator. If an identifier designates two different entities in the same name space, the scopes might overlap. If so, the scope of one entity (the *inner scope*) will be a strict subset of the scope of the other entity (the *outer scope*). Within the inner scope, the identifier designates the entity declared in the inner scope; the entity declared in the outer scope is *hidden* (and not visible) within the inner scope. Two identifiers have the same scope if and only if their scopes terminate at the same point.

Unless explicitly stated otherwise, where this manuscript uses the term *identifier* to refer to some entity (as opposed to the syntactic construct), it refers to the entity in the relevant name space whose declaration is visible at the point the identifier occurs.

Class, structure, union, and enumeration tags have scope that begin just after the appearance of the tag in a type specifier that declares the tag. Each enumeration constant has scope that begins just after the appearance of its defining enumerator in an enumerator list. Any other identifier has a scope that begins just after the completion of its declarator.

3.5.2 Linkages of Identifiers

An identifier declared in different scopes or in the same scope more than once can be made to refer to the same object or function by a process called *linkage*. There are four kinds of linkage: global, external, internal, and none.

In the set of source files that constitutes an entire program, each declaration of a particular identifier with *external linkage* denotes the same object or function. Within a source file, each declaration of an identifier with *internal linkage* denotes the same object or function. Each declaration of an identifier with *no linkage*

denotes a unique entity.

If the declaration of a file scope identifier for an object or a function contains `__declspec(global)`, the identifier has global linkage.

If the declaration of a file scope identifier for an object or a function contains the storage-class specifier **static**, the identifier has internal linkage.

For an identifier declared with the storage-class specifier **extern** in a scope in which a prior declaration of that identifier is visible, if the prior declaration specifies internal or external linkage, the linkage of the identifier at the later declaration is the same as the linkage specified at the prior declaration. If no prior declaration is visible, or if the prior declaration specifies no linkage, then the identifier has external linkage.

If the declaration of an identifier for a function has no storage-class specifier, its linkage is determined exactly as if it were declared with the storage-class specifier **extern**. If the declaration of an identifier for an object has file scope and no storage-class specifier, its linkage is external.

The following identifiers have no linkage: an identifier declared to be a function parameter; a block scope identifier for an object declared without the storage-class specifiers **extern**.

It is a syntax error, if the same identifier appears with both internal and external linkage.

3.5.3 Name Spaces of Identifiers

If more than one declaration of a particular identifier is visible at any point in a program, the syntactic context disambiguates uses that refer to different entities. The separate *name spaces* categorized for various identifiers are given as follows:

- *macro names* the macros defined by the preprocessing directive `#define`.

- *label names* (disambiguated by the syntax of the label declaration and use);

- the tags of classes, structures, unions, and enumerations (disambiguated by following any of the keywords **class**, **struct**, **union**, or **enum**);

- the *members* of classes, structures or unions; each class, structure or union has a separate name space for its members (disambiguated by the type of the expression used to access the member via the `.` or `->` operator);

- all other identifiers, called *ordinary identifiers* (declared in ordinary declarators or as enumeration constants).

3.5.4 Storage Duration of Objects

The valid storage-class specifiers are given in Table 3.3.

An object has a *storage duration* that determines its lifetime. There are three storage durations: static, automatic, and allocated.

Variables qualified by `__declspec(global)` can cross different programs when they are executed in the current shell using dot command. A variable qualified by `__declspec(global)` should be only declared once and used by multiple programs in the current shell. Such a global variable is typically declared in the system startup file `chrc` or the user's startup file `_chrc` in Windows or `.chrc` in Unix. Variables of functions and class/struct/union cannot be declared as global variables.

An object whose identifier is declared with external or internal linkage, or with the storage-class specifier **static** has *static storage* duration. For such an object, storage is reserved and its stored value is initialized only once, prior to program startup. The object exists, has a constant address, and retains its last-stored value throughout the execution of the entire program.

Table 3.3: Storage-class Specifiers.

Specifier	Function
auto	local automatic variable
extern	external variable
__declspec(global)	system-wide global variable
__declspec(local)	nested local function
register	(ignored)
static	static variable

An object whose identifier is declared with no linkage and without the storage-class specifier **static** has *automatic storage duration*. For such an object that does not have a variable length array type, storage is guaranteed to be reserved for a new instance of the object on each entry into the block with which it is associated; the initial value of the object is zero. If an initialization is specified for the object, it is performed each time the declaration is reached in the execution of the block; otherwise, the value becomes indeterminate each time the declaration is reached. Storage for the object is no longer guaranteed to be reserved when execution of the block ends in any way. (Entering an enclosed block or calling a function suspends, but does not end, execution of the current block.)

For such an object that does have a variable length array type, storage is guaranteed to be reserved for a new instance of the object each time the declaration is reached in the execution of the program. The initial value of the object is zero. Storage for the object is no longer guaranteed to be reserved when the execution of the program leaves the scope of the declaration.

If an object is referred to when storage is not reserved for it, the behavior is undefined. The value of a pointer that referred to an object whose storage is no longer reserved is indeterminate. During the time that its storage is reserved, an object has a constant address.

The storage can be *allocated* dynamically at run time by the functions **calloc(), malloc()**, and **realloc()**; and subsequently freed by the function **free()**. The storage can also be dynamically *allocated* and deallocated by operators **new** and **delete**, respectively. Details about memory allocation and pointers will be described in Chapter 9.

Chapter 4

Portable Interactive Command Shell and Shell Programming

This chapter describes how Ch can be used interactively in the command mode as a command shell. Like other shells, Ch shell is a command interpreter that reads command lines typed by the user at a prompt and figures out what to do. All operators and functions as well as most commands are available for both interactive shell and shell programs in Ch. Detailed information about operators and functions can be found in Chapters 7 and 10, respectively. From a semantic point of view, Ch shell is similar to C shell. Ch is a superset of C whereas the so-called C shell is quite different from C. Some selected syntax comparisons between C shell and Ch are listed in Appendix D.

4.1 Shell Prompts

Each shell has its own shell prompt. By default, the prompt for a regular Ch shell is '`cwd> `' where `cwd` is the current working directory. It tells the user that the regular Ch shell is ready to process the input from the command line. By default, the prompt for a safe Ch shell is '`safech> `'. For more information about safe Ch, refer to Chapter 21. For a superuser in Unix or an administrator in Windows, shell prompts are '#' and '`safech#`' for regular and safe Ch shells, respectively.

Table 4.1 is the comparison of default shell prompts between Ch and other popular shells. The user can change the default symbol for the Ch shell prompt, or add information such as hostname and current working directory to the shell prompt by setting system variable **_prompt**. For example, in the interactive command shell below

```
> _prompt = "$ "
```

Table 4.1: Comparison of shell prompts.

Shell	General User Prompt	Superuser Prompt
Ch shell in Windows	>	#
Ch shell in Unix	>	#
Safe Ch shell in Windows	safech>	safech#
Safe Ch shell in Unix	safech>	safech#
C shell	%	#
Bourne, Korn, and BASH shells prompt	$	#

```
$
$ _prompt = "% "
%
% _prompt = stradd(_cwd,"> ")
/usr/ch>
```

we set the Ch shell prompt to symbols '$' and '%' first, then set it to the current working directory ending with the symbol '>' by calling function **stradd()**. In this example, the current working directory is `/usr/ch`. By setting the value of **_prompt**, the user can choose any character as the shell prompt. Typically, system variable **_prompt** is set in the startup file **.chrc** in Unix or **_chrc** in Windows in the user's home directory.

4.2 Interactive Execution of Commands

In the command line mode of Ch shell, the user can type commands at a shell prompt. The commands include compiled binary executable files, shell scripts, C and Ch programs, etc. For example,

```
> pwd
/home/myname
> mkdir subdir1
> cd subdir1
> pwd
/home/myname/subdir1
> which ls
ls is aliased to ls -F
>
```

In the above example, program **pwd** displayed the current working directory `/home/myname`. A new directory `subdir1` is created by command **mkdir**. The current directory is changed by the built-in command **cd**. The shell program **which** indicates that **ls** is an alias, which will be described in section 4.6.

To run a command file in command mode, the file name shall be a valid identifier in Ch or starts with a relative or absolute directory path such as '`./`', '`../`', '`~/`', and '`/`'. For example, numerical values such as `20` or 20.e1 are not valid identifiers. A command can be enclosed in a pair of double quotation marks. The option for the command shall not be included inside the quotation marks. The quotation marks can be used to avoid the conflict of a command and a variable identifier in a program. It can also be used in the case that the command is located in a directory with white space. For example,

```
> int ls = 10
> ls*2
20
>  "ls" -1
(display files in the current directory in a single column)
> "C:/Program Files/Windows NT/Accessories/wordpad.exe"
(launch wordpad program)
```

The user can type two or more commands in the same command line by putting semicolons between them. For example, the compound command

```
> cp filename1 filename2; vi filename2
>
```

copies file `filename1` to file `filename2`, and then calls the command **vi** to edit the latter.

4.2.1 Current Shell

The principle and syntax of running a program in the current shell in Ch is the same as those in sh, bash, and ksh shells. By default, a program in Ch shell is executed in a subshell. The built-in dot command

```
. filename
```

executes program *filename* in the current shell, instead of a subshell. When a command is typed in the prompt, either with or without '.', the search paths specified by the system variable **_path** are used to find the directory containing the command. Assume program **cmd** has two statements of

```
int x = 3;
double y = 4;
```

In the example below, this program is executed in a subshell first and then in the current shell.

```
> cmd    // run cmd in a subshell
> x      // print the value of variable x in current shell
ERROR: variable 'x' not defined
ERROR: command 'x' not found
> . cmd  // run cmd in the current shell
> x
3
> x*y
12.0000
> showvar
    x          3
    y          4.0000
```

The first execution of program `cmd` (without symbol '`.`') is in a subshell. So, when the program quits, the variable `x` which hasn't been defined yet in the current shell is not available. The second execution of program `cmd` (with symbol '`.`') is in the current shell. When the program quits, the variable `x` has the value of 3, assigned to `x` within the program `cmd`, in the current shell. Because variables in the current shell can be used interactively at the prompt, sometime, one may place variables to be used interactively at the prompt in a command and execute it in the current shell as a dot command. All variables and their values can be displayed using the shell command **showvar**. The shell command **stackvar** is obsolete and replaced by the command **showvar**.

Ch will search for a command such as `cmd` in the directories specified by the system variable **_path**. If the program *cmd* is not located in one of directories specified by the system variable **_path**, an error message will be displayed. If `cmd` has been used as a variable in the current shell, the command can be used with a preceding absolute or relative path as shown below.

```
> /dir1/dir2/cmd   // run cmd in the directory /dir1/dir2
> ./cmd            // run cmd in the current working directory
> ../cmd           // run cmd in the parent directory
> ~/cmd            // run cmd in the home directory
```

These commands execute the command file `cmd` located in the directory `/dir1/dir2`, current working directory, parent directory and home directory, respectively.

Pathnames for a command can be separated using an '/' in both Unix and Windows. However, the separator '\' can also be used in Windows. For example, the program `notepad` can be launched in Windows in one of the following forms.

```
> notepad
> C:/Windows/notepad
> /Windows/notepad
> "/Windows/notepad"
> C:\Windows\notepad
> \Windows\notepad
```

4.2.2 Background Job

In MS-DOS command shell in Windows, all Win32 programs run as background jobs. Ch is consistent in handling commands in both Unix and Windows. A command can be started in the background using the & metacharacter. so that it will not block the shell to accept new commands. For example, the command

```
> notepad &
```

will launch the program notepad in background.

4.3 Interactive Execution of Programming Statements

As it is mentioned before, besides executable binary files and shell scripts, the Ch shell can also execute C/Ch programs directly without compilation. Interactive execution of C programs without lengthy compile/link/execute/debug cycles is especially appealing for rapid application development and deployment. For example, assume file `hello.c` contains the following statements.

```
#include <stdio.h>
int main(void) {
  printf("Hello, world!\n");
  return 0;
}
```

It can be executed in a Ch shell without compilation as follows.

```
> hello.c   // execute hello.c program without compilation
Hello, world!
>
```

Source files as well as programming statements can be executed in Ch shell directly and interactively. In the interactive command line mode, semicolons at the end of programming statements are not required. For example,

```
> int i
> i = 10
10
> i * 2
20
> printf("i = %d", i)
i = 10
> printf("i in hexadecimal number = %x", i)
i in hexadecimal number = a
>
```

Ch also supports C extensions, such as computational arrays in Ch Professional and Student Editions, in the command line mode as shown below.

```
> array int a[2][3] = {1, 2, 3, 4, 5, 6}
> a
1 2 3
4 5 6
> 2 * transpose(a)
2 8
4 10
6 12
>
```

where `a` is a `2X3` computational array which is treated as a single object. Generic function **transpose()** returns the transpose of the argument of a one-dimension vector or a 2-dimension matrix. Computational arrays are useful for numerical computing in engineering and science. More information about computational arrays can be found in Chapter 16.

To use macros and defined types by **typedef** in a header file at a shell prompt, the user can load the header file by using commands **chparse** and **chrun** mentioned in section 3.4.6. For example, in the commands below

```
> chparse /usr/local/ch/include/stdlib.h
> chrun
> size_t i
> i = 90
90
>
```

the header file **stdlib.h**, where the type **size_t** is typedefed, is loaded before **size_t** is used as a type declarator for variable `i`. In this case, the header file **stdlib.h** runs in the current shell.

If an invalid statement is typed at a shell prompt, Ch will give error messages for debugging purposes.

```
> blah
ERROR: variable 'blah' not defined
ERROR: command 'blah' not found
>
```

When a function is called in the command mode, the search paths specified by the system variable **_fpath** are used to find the directory containing the function definition. To call functions in a program from the command prompt of a Ch shell, the program has to be loaded first. Once a program is loaded by command **chparse**, the program can be executed by command **chrun** as described in section 3.4.6. At the same time, functions in the program can also be called at the prompt. On the other hand, a program can be executed in the current shell first and then functions in the program can be used interactively. For example, in the interactive execution of program `currentshell.cpp` in Program 4.1 below,

```
> . currentshell.cpp
func(5) = 10
15
> func(10)
20
```

```
 #include <stdio.h>
 #include <iostream.h>

 int func(int i) {
   return 2*i;
 }

 class tag {
   private:
     int m_i;
   public:
     tag();
     int memfunc(int);
 };
 tag::tag() {
   m_i=10;
 }
int tag::memfunc(int i) {
   cout << m_i+i << endl;
   return m_i+i;
 }

 int main() {
    class tag c1;

    printf("func(5) = %d\n", func(5));
    c1.memfunc(5);
 }
```

Program 4.1: a C++ program `currentshell.cpp` executed in the current shell.

Table 4.2: Built-in shell commands in Ch.

Command	Description
X:	change to the directory in drive **X** in Windows
cd	change to the home directory
cd -	change to the previous directory
cd --	change to the directory before the previous one
cd ---	change to the directory before the previous two
cd *dir*	change to the directory *dir*
cd *dir name*	change to the directory *dir name* with space
chdir	change to the home directory
chdir -	change to the previous directory
chdir --	change to the directory before the previous one
chdir ---	change to the directory before the previous two
chdir *dir*	change to the directory *dir*
chdir *dir name*	change to the directory *dir name* with space
. *filename*	dot command. Read and execute command *filename* in the current shell, instead of a subshell.
exec *command*	execute command in place of the current shell.

```
> class tag c
> c.memfunc(10)
20
>
```

the dot command `. currentshell.cpp` loads and executes a C++ program currentshell.cpp in the current shell with output of.

```
func(5) = 10
15
```

Command `func(10)` in the command prompt calls the function `func()` in the program loaded in the current shell. Declaration statement `class tag c` instantiates an object `c` of `class tag`. When the member function tt tag::memfunc() is invoked interactively by function call `c.memfunc(10)`, the result of value 20 will be displayed. Details about using class for object-based programming will be described in Chapter 19. Note that when a program running in the current shell crashes, the current shell will be terminated. For the debugging purpose, it is recommended to run such a program in a new Ch shell so that when the current shell is terminated, the Ch shell running in the background will be still available as shown below.

```
> ch
> . currentshell.cpp
```

4.4 Built-in Commands

The built-in commands of Ch are listed in Table 4.2. The directory in which the user is working is called the *current working directory* or *cwd*. To check the current working directory, type the command **pwd** at a shell prompt.

There are three kinds of directory names or pathnames in Ch : *simple*, *absolute* and *relative*. The simple pathnames are file or directory names which don't include any information about the position within the file system hierarchy. The simple pathnames are used to go to subdirectories of the current working directory. Absolute pathnames indicate the absolute position of a directory within the file system hierarchy. They begin with character '`/`' which represents the root directory. In Windows, they can also begin with a letter standing for a drive such as `X:/`. For example, the pathname **/usr/ch** indicates the absolute position of directory **ch** from the root directory. The relative pathnames trace the path from the working directory, instead of the root, to the desired file or directory. For example, the pathname **../ch** indicates the relative position of directory from the current working directory. In relative pathnames, the symbols **.** and **..** indicate the current working directory and the parent directory, respectively.

In Ch shell, built-in commands **cd** and **chdir** can be used to change the user's current working directory to a desired directory. The command **cd** or **chdir**, without a directory name, change the current working directory to the home directory indicated by the system variable **_home**. The command **cd** *dir* or **chdir** *dir* switches the current working directory to the directory *dir*. The command **cd** `-` or **chdir** `-` switches the current working directory to the previous directory. Similarly, **cd** `--` or **chdir** `--` switches to the directory before the previous one; **cd** `---` or **chdir** `---` switches to the directory before the previous two. For example,

```
> pwd
/home
> cd /usr/ch
> pwd
/usr/ch
> chdir -
> pwd
/home
> cd myname
> pwd
/home/myname
> cd ../../usr/ch
> pwd
/usr/ch
>
```

where `/usr/ch` is an absolute pathname, `myname` is a simple pathname, and `../../usr/ch` is a relative pathname.

Examples about execution of programs in the current shell using dot commands are given in sections 4.2 and 4.3

The built-in command **exec** executes other command in place of the current shell, which terminates.

4.4.1 Commands For Interactive Shell Only

All operators, functions and built-in commands are available for both interactive shell and shell programs in Ch. But, not all commands that are valid in the command line mode can be used in shell programs. The commands which are valid only when Ch is invoked as an interactive shell are called *interactive commands*. They are not valid inside a Ch program. All interactive commands are shown in Table 4.3.
In the interactive command line mode of the Ch shell, a variable, including a variable of function type, can be removed by the **remvar** command. In the following example,

Table 4.3: Interactive commands valid only in the interactive shell.

Command	Description
!	repeat the previous executed command.
chdebug *filename*	debug program *filename*.
chparse [-S] *filename*	parse program *filename* only to check syntax. Option -S for safe shell.
chrun	execute the parsed program.
exit	exit Ch shell.
history	show the command history.
remvar	remove a variable.
remkey	remove a keyword .
showvar	display variables and their values in all stacks.

```
> int i        // define variable i
> i = 90
90
> remvar i     // remove variable i
> i
ERROR: variable 'i' not defined
ERROR: command 'i' not found
>
```

command `int i` declared variable `i` in Ch shell, and command `remvar i` removes variable `i`. The command **remvar** is an interactive command which is invalid inside Ch programs. If the user wants to remove a variable, say `var`, inside a shell program, the preprocessing directive `#pragma remvar(var)` should be used.

Similarly, a keyword can be removed by the **remkey** command as shown below.

```
> remkey(sin)   // generic function sin is removed as a keyword
> float sin
> sin =10.0
```

Inside a program, the preprocessing directive `#pragma remkey(key)` should be used to remove the generic function `sin()`.

The command **showvar** can be used to display all global variables and their values in the current shell. The default format will be used to display the value for a variable. Tag names for sruct/class/enum types, function prototypes without function definition, and typedefed variables are not displayed. Members of a structure type and arrays are displayed without indentation. For example,

```
> int x = 3;
> double d = 10.1234
> double a[2][3] = {1,2,3,4,5,6};
> array double b[2][3] = {1,2,3,4,5,6};
> struct tag {int i, int j;} s = {10, 20};
> showvar
    x             3
    d             10.1234
    a [C array]
```

```
1 2 3
4 5 6
    b [Ch array]
1 2 3
4 5 6
    s
.i = 10
.j = 20
```

The command **showvar** can also be used to display all variables and their values in a command executed in the current shell as shown in section 4.2.1.

More information about the event designator `!` and command **history** can be found in section 4.5 below. More information about commands **chdebug**, **chparse** and **chrun** can be found in section 3.4.6. Generic function **alias**() is typically used inside the system startup file **chrc** and the user's startup file **.chrc** in Unix or **_chrc** in Windows. Commands **alias** and **unalias**, which will be described in section 4.6, can be used in the command mode.

4.5 Repeating Commands at Prompt

The features described in this section are valid only at the command line mode in Ch shell. The history and quick substitutions for repeating commands at prompt will be described in this section.

The most convenient way to repeat commands at prompt is to use arrow keys. The previously typed commands can be retrieved easily by the upper '↑' and down '↓' arrow keys on the keyboard for commands typed before and after the current command, respectively. The retrieved command can be modified by first moving the cursor to the location using the left '←' or right '→' arrow keys on the keyword. Then, use delete or backspace key to delete characters or type any graphical characters to insert characters for command line editing like in Emacs text editor.

4.5.1 History Substitution

History substitution allows the user to use words from previously typed commands at a shell prompt. This simplifies spelling corrections and the repetition of complicated commands or arguments. Command lines are saved in the history list, the size of which is controlled by the system variable `_histsize`. The history can be displayed by shell command `history`. The most recent commands are retained. A history substitution starting with a `!` sign may occur only at the beginning of the command line; history substitutions cannot be nested. For example, commands

```
> _histsize       // print the current value of _histsize
20
> _histsize = 4   // change the current value of _histsize to 4
4
> pwd
/usr/ch
> history         // print the history list of commands
123   _histsize       // print the current value of _histsize
124   _histsize = 4   // change the current value of _histsize to 4
125   pwd
126   history         // print the history list of commands
>
```

Table 4.4: Event designators.

Command	Description
`!`	Refer to the previous command. By itself, this substitution repeats the previous command.
`!!`	the same as `!`.
`!n`	Refer to command line *n*.
`!-n`	Refer to the current command line minus *n*.
`!str`	Refer to the most recent command starting with *str*.

print the current value of _histsize first, and then change this value to 4. After another command **pwd**, the command **history** prints the most recent 4 commands including comments in the history list. The number displayed at the front of each command is the command line number. History substitution allows users to repeat previous command lines which are in the history list by using *event designators*.

An event designator is a reference to a command line entry in the history list. Different event designators listed in Table 4.4 make it more convenient to repeat execution of a long command line in the history list. The most commonly used event designator is `!`. The `!` repeats the last command line entered by a user. For example, if a user uses the command **more** to view a file, and misses the part of the file he wants, he can repeat **more** just by typing `!`. The shell types out the command line repeated by `!` first, and then executes it, so that the user can make sure it is the right one. The `!` is the basis for a number of more sophisticated timesaving event designators which are listed in Table 4.4. Command `!n` repeats the command with the number *n* in the command history list. Command `!-n` repeats the command with the number of *m-n*, where *m* is assumed to be the number of the current command. It means the command `!-1` is equivalent to the command `!`. Command `!str` is also a commonly used command for command repetition. If a user wants to edit the most current file again, he doesn't need to remember the number of the previous **vi** command, all he has to do is to type `!vi`.
The following example illustrates how these event designators can be used to repeat a command in the history list.

```
> _histsize = 5
5
> pwd
/usr/local//ch
> !
pwd
/usr/local/ch
> strlen("abc")
3
> history
136   _histsize = 5
137   pwd
138   pwd
139   strlen("abc")
140   history
> !137
pwd
/usr/local/ch
```

```
> !h
history
138   pwd
139   strlen("abc")
140   history
141   pwd
142   history
> !-4
strlen("abc")
3
>
```

4.5.2 Quick Substitution

The quick substitution allows users to make a change on the previous command and at the same time execute the changed command. It is useful to correct typos in commands or to repeat similar commands. Commands ^*old*^*new* and ^*old*^*new*^ can substitute string *old* in the previous command with string *new*. For example,

```
> mkkdir mydir
ERROR: variable 'mkkdir' not defined
ERROR: command 'mkkdir' not found
> ^kk^k
> history
11    mkkdir mydir
12    mkdir mydir
13    history
>
```

To correct the typo `mkkdir`, use the command `^kk^k`. In the following example, quick substitution commands are used for a repetitive task of creating five directories for five different months.

```
> mkdir Jan
> ^Jan^Feb
> ^Feb^March
> ^March^April
> ^April^May
> history
31    mkdir Jan
32    mkdir Feb
33    mkdir March
34    mkdir April
35    mkdir May
36    history
>
```

Quick substitution command ^*old* and ^*old*^ can be used to delete string *old* in the previous command. For example,

```
> cp file file1.c
> ^1
```

Table 4.5: Quick substitution.

Command	Description
^*old*^*new*	substitute string *old* in the previous command with string *new*.
^*old*^*new*^	the same as ^*old*^*new*.
^*old*	delete string *old* in the previous command.
^*old*^	the same as ^*old*.

```
> history
56   cp file file1.c
57   cp file file.c
58   history
>
```

4.5.3 File Completion

Ch shell is able to complete words when given a unique abbreviation. Type part of a word (for example 'ls /usr/local/ch/de') and hit the tab key, The shell completes the file name '/usr/local/ch/de' to '/usr/local/ch/demos/', replacing the incomplete word with the complete word in the input buffer. Note the terminal '/'; completion adds a '/' to the end of completed directories and a space to the end of other completed words, to speed typing and provide a visual indicator of successful completion.

If no match is found (perhaps '/usr/local/ch/demos' doesn't exist), the terminal bell rings. If the word is already complete (perhaps there is a '/usr/local/ch/de' on your system, or perhaps you were thinking too far ahead and typed the whole thing) a '/' or space is added to the end if it isn't already there.

File completion works only at the end of the input buffer.

If there are multiple choices, the shell lists the possible completions using the command `ls -F` and reprints the prompt and unfinished command line, for example:

```
> ls /usr/local/ch/d[^D]
dl/     demos/  docs/
> ls /usr/local/ch/d
```

If the choices are more than 100, it will ask the user to confirm whether all the choices shall be displayed:

```
> ls fil[tab]
Display all 102 choices? (y or n)
```

Ch shell completes on the shortest possible unique match, even if more typing might result in a longer match:

```
> ls
fodder   foo      food     foonly
> rm fo[tab]
```

just beeps, because 'fo' could expand to 'fod' or 'foo', but if we type another 'o',

```
 > rm foo[tab]
 foo food foonly
 > rm foo
```

the completion completes on 'foo', even though 'food' and 'foonly' also match.

If the first command is "cd", the completion shows only the choices of directory only:

```
> ls dir[tab]
dir1/ dir2/ dir3/ dir4@ dirf1 dirf2 dirf3@
> cd dir[tab]
dir1/ dir2/ dir3/ dir4@
```

For a symbolic link to a file or directory, the symbol '@' is attached.

The shell treats `'\ '` as a space and `'\$'` as `'$'` in the file completion:

```
> ls test\ t[tab]
ls "juck tmp"
```

The shell adds double quotes to enclosing the directory that contains space(s) in file completion as shown above.

For built-in command **cd**, the backslash can be omitted for a directory contains spaces in file completion.

```
> cd aa b[tab]
> cd "aa bb"/
```

A directory or file in Windows often contains a space. The shell can complete the file or directory in this case.

```
> cd Prog[tab]
> cd "Program Files"/
```

4.5.4 Command Completion

If a tab key is hit before the end of the first token, Ch shell handles the token with command completion. The shell searches files in the directories specified in the environment variable `PATH` which has the same value as in the system variable **_path**. in both windows and Unix. Only executable files will be selected for command completion in Unix. In Windows, only files with extensions `.com, .exe, .bat, .cmd,` or `.ch` will be selected.

If there is only one matched command, the shell replaces the incomplete command with the complete command in the input buffer. The shell adds a space to the end of other completed command to speed typing and provide a visual indicator of successful completion. For example,

```
> lps[tab]
> lpstat
```

Similar to file completion, if there are multiple choices, the shell lists the possible completions using the command `ls -F` and reprints the prompt and unfinished command line. If the chioces are more than 100, it will ask the user to confirm whether all the choices shall be displayed.

```
> lp[tab]
lp lpstat
> lp
```

If no command match is found in the directories specified in the environment variable `PATH`, the shell searches the current directory for possible matches of directory. If there is only one matched directory, the shell replaces the incomplete command with the complete directory in the input buffer. The shell adds '/'

to the end of the completed directory to speed typing. If there are multiple choices of directory, the shell lists the possible completions using the command `ls -F` and reprints the prompt and unfinished command line. If the chioces are more than 100, it will also ask the user to confirm whether all the choices shall be displayed.

If no match is found at all, the terminal bell rings.

To search commands only in the current directory for command completion, the user shall type the command starting with the current directory `./`. For example,

```
> cd /bin
> ./log[tab]
logger   login    logname
> ./log
```

Type tab directly in command line, the shell will be able to show all the commands.

```
> [tab]
Display all 1296 choices? (y or n)
```

4.6 Aliases

In interactive command mode, the Ch shell maintains a list of aliases that one can create, display, and modify using commands **alias** and **unalias**. The shell checks the first word in each command to see if it matches the name of an existing alias. If it does, the command is reprocessed with the alias definition replacing its name.

Aliases are typically created using the generic function **alias**() in the system startup file **chrc** and the startup file **.chrc** in Unix or **_chrc** in Windows in the user's home directory. The generic function **alias**() is overloaded with the following prototypes.

```
int alias(string_t name, string_t alius);
string_t alias(string_t name);
int alias(void);
```

The different arguments with corresponding return values are listed in Table 4.6. Function call **alias**(`name, alius`) will make symbol `name` an alias to command `alius`. If `name` is a valid identifier, the function returns 0. If `name` is already an alias, the function returns 1. If the value of `name` is NULL, it returns -1. If the second argument `alius` is NULL, the function will unalias symbol `name` from command `alius`. Function call **alias**(`name`) will return the alias for the symbol `name` as a string. If the symbol `name` is not an alias, the function returns NULL. Function call **alias**() will print out all the names as well as their aliases in the standard output; and return the number of aliases. The return values of this generic function are shown below in an interactive execution session. The function **alias()** can be called both in command mode and shell programs. Follow the C convention, characters `'\'` and `'"'` can be passed in an alias using escape character `'\'` as `'\\'` and `'\"'`, respectively. The commands below demonstrates various features of function **alias**().

```
> alias("ls", "ls -a")
0
> alias("ls", "ls -agl")
1
> alias("cp", "cp -i")
```

Table 4.6: Function call **alias**().

Function Call	Return Value
alias("*name1*", "*alius*")	0
alias("*name1*", "*alius*")	1
alias("*name2*", "")	0
alias("*name2*", NULL)	0
alias("*name3*", NULL)	1
alias(NULL, "*alius*")	-1
alias(NULL, NULL)	-1
alias("*name1*")	alius
alias("*name3*")	NULL
alias(NULL)	NULL

Table 4.7: Formal arguments in **alias**().

Formal argument	Description
_argv[0]	The first input word (command).
_argv[n]	The nth argument.
_argv[#]	The entire command line.
_argv[$]	The last argument.
_argv[*]	All the arguments, or a null value if there is just one word in the command.

```
0
> alias()
cp      cp -i
ls      ls -alg
2
> alias("ls", NULL)
0
> alias()
cp      cp -i
1
> alias("cp")
cp -i
>
```

The *argument substitution* is available in aliases. The formal arguments shown in Table 4.7 inside a definition of an alias will be replaced with actual command line arguments when the alias is used. If no argument substitution is called for, the arguments remain unchanged. For example,

```
> echo abc xyz
abc xyz
> alias("myecho1", "echo _argv[1]")
```

Table 4.8: Commands **alias** and **unalias**.

Command	Description
alias *name alius*	make alias
alias *name* "*string with space*"	make alias
alias *name*	display alias for *name*
alias	display all aliases
unalias *name*	unalias *name*

```
> myecho1 abc xyz
abc
> alias("myecho2", "echo _argv[$]")
> myecho1 abc xyz
xyz
```

In the above example, only the first argument of command `myecho1 abc xyz` is used in the alias. The last argument is used in alias `myecho2`. As another example, to search a file in the current directory and its subdirectories and then print it out, the alias `find` below can be used to replace the system command **find** as follows.

```
> alias("find", "find . -name _argv[1] -print")
> find filename
(display files with name 'filename')
```

For the current process, commands **alias** and **unalias** shown in Table 4.8 can be more conveniently used in an interactive command shell. These two commands are valid only in command mode. For example,

```
> alias ls "ls -agl"
> alias cp "cp -i"
> alias
cp      cp -i
ls      ls -alg
> unalias ls
> alias
cp      cp -i
> alias cp
cp -i
>
```

Aliases can be nested. That is, an alias definition can contain the name of another alias. This is useful in pipelines such as

```
> alias("ls", "ls -agl")
> alias("lm", "ls * | more")
```

When command `lm` is invoked, actually the expanded command

```
> ls -agl * | more
```

Table 4.9: Variable substitution.

Command	Description
$*var*	replaced by the value of variable *var*.
${*var*}	replaced by the value of variable *var*.
$(*var*)	replaced by the value of variable *var*.

is invoked instead. The output of `ls` is piped through program `more`. As another example, the command alias `opentgz` below can be used portably to extract a compressed archival file such as `file.tar.gz` or `file.tgz`.

```
> alias("opentgz", "gzip -cd _argv[1] | tar -xvf -")
> opentgz file.tar.gz
(display extracted files from file.tar.gz)
```

The command use a pipeline described in section 4.11.

Nested aliases are expanded before any argument substitution is applied. For example,

```
> alias("t1", "t2 _argv[1] A")
> alias("t2", "echo a b c")
> t1 x y z
a b c x A
> alias("p1", "p2 a b c")
> alias("p2", "echo _argv[1] A")
> p1 x y z
a A
```

4.7 Variable Substitution

A variable name can be replaced by its value through *variable substitution*. Three syntaxes of variable substitution $*var*, $(*var*), and ${*var*} are shown in Table 4.9. The variable name or symbol to be expanded may be enclosed in parentheses or braces, which are optional but serve to protect the variable to be expanded from characters immediately following it which could be interpreted as part of the name. Variable substitution makes code more portable and flexible, because a variable can have different values for different situations. For example, an installation program with variable substitutions can allow users to specify different target directories instead of the default directory. The user may choose to install the software in a directory of his choice.

Variable substitution takes place after the input command line is analyzed and aliases are resolved. It is valid only for interactive command mode, command statements in programs, and command substitution operations described in section 4.9.

The variable in a variable substitution could be a predefined identifier described in section 2.3.1; a user-defined variable of string, pointer to char, or integral data type; an environment variable described in section 4.14; or undefined symbol. For a variable substitution, the Ch shell will first search the Ch name space for the variable name according to its scope rule. If the variable is not defined, then it searches the environment variables of the current process. If no variable with the specified name is found either in Ch space or environment space, no substitution will take place and the variable is ignored.

The variable substitution can be prevented by preceding the '$' with a '\' except within ''s for command substitution where it always occurs, and within ''s where it never occurs. A '$' is passed unchanged if followed by a blank, tab, or end-of-line. For example, assume `myname` is the user's account name, the following commands

```
> _home                    // _home is a predefined identifier
/home/myname
> cd $_home
> pwd
/home/myname
> _fpathext                 // _fpathext is a predefined identifier
.chf
> // copy file1 to file1.ch
> cp file1 file1$_pathext
> cd $CHHOME              // CHHOME is an environment variable
> pwd
/usr/ch
> echo $ CHHOME \$10.5 ${_home}/tmp
$ CHHOME $10.5  /home/myname/tmp
```

are examples of variable substitution using $*var* and ${*var*}. The variables **_home** and **_fpathext** are predefined identifiers in Ch. The variable **_home** contains the home directory of the current user. For different users, values of **_home** are different. In a shell program, the command `cd $_home` is more flexible than command `cd /home/myname`. Similarly, the environment variable **CHHOME** contains the home directory of Ch, which might be different in different machines. Obviously, a shell program using the command `cd $CHHOME` are more portable than the program using command `cd /usr/ch`. To display '$' immediately followed by a digit, it has to be preceded by an `'\'`.

Furthermore, suppose that the user is working with files in a directory with a very long name `/home/myname/project1/subproject2/plan1`. This pathname can be abbreviated as a string `mydir`. Then, when this directory is used in commands, `$mydir` instead of `/home/myname/project1/subproject2/plan1` can be used on the command line. For example,

```
> string_t mydir = "/home/myname/project1/subproject2/plan1"
> cd $mydir
> pwd
/home/myname/project1/subproject2/plan1
>
```

4.7.1 Expression Substitution

Expression substitution allows the evaluation of a Ch expression and the substitution of the result. The format for expression substitution is:

```
$(expression)
```

or

```
${expression}
```

The expression shall be an expression of string, pointer to char, or integral data type. It can be a constant, variable, function call, mathematical expression, and other valid expression.

A single variable can be treated as an expression. While variable substitution can be used to obtain the values of a single variable, expression substitution is typically reserved for more complicated expressions. For an environment variable or undefined symbol, variable substitution should be used. In the example below, the environment variables used in commands are obtained by an expression calling the function **getenv()**.

For example,

```
> getenv("CHHOME")         // CHHOME is an environment variable
/usr/local/ch
> cd $(getenv("CHHOME"))
> pwd
/usr/local/ch
> ls $CHHOME/include
... (list of /usr/local/ch/include)
> ls ${CHHOME}/include
... (list of /usr/local/ch/include)
> ls $(getenv("CHHOME"))/include
... (list of /usr/local/ch/include)
> ls $(stradd(getenv("CHHOME"),"/include"))
... (list of /usr/local/ch/include)
>
```

where commands `ls $CHHOME/include`, `ls ${CHHOME}/include`, `ls $(getenv("CHHOME"))/include`, and `ls $(stradd(getenv("CHHOME"),"/include"))` are equivalent to the command `ls /usr/local/ch/include`. But, the commands including `CHHOME` are more portable. Because the character '/' is not a valid character for an identifier, the braces for variable substitution of variable `CHHOME` is optional.

4.7.2 Command Name Substitution

A command name substitution is useful for execution of a command. The command to be executed is obtained dynamically at run time. The syntax for command name substitution is the same as those for variable name substitution and expression substitution. In this case, the $ shall appears at the beginning of a syntax statement. The data type of the variable or expression following the $ sign shall be string, pointer to char, or pointer to unsigned char. For example,

```
string_t cmd = "/Ch/bin/echo.exe option";
$cmd more options
string_t cmd2 = "\"C:/Program Files/ch/bin/echo.exe\"";
$cmd2 option2
char *cmd3 = "ls";
$cmd2 -agl
```

The string `cmd` contains both the command `/Ch/bin/echo.exe` and its `option`. To use a command with white space, the command has to be placed inside a pair of double quotation marks as shown in the string `cmd2` above for the command `C:/Program Files/ch/bin/echo.exe`. A string `cmd3` in the form of a pointer to char contains the command `ls`.

4.8 Filename Substitution

Using certain special characters called *wild card characters*, users can abbreviate filenames and directory names by filename substitution. Valid wild card characters in Ch are shown in Table 4.10. Symbol '?' is the wild character representing a one-character value; * represents an arbitrary number of characters. For example, to list all the files in the current working directory, type

```
> // list all files in current directory with *
> ls *
abc1.ch   abc2.ch   abc3.ch   abc12.c   efc1.c
>
```

To list all the files with extension of .ch, type

```
> ls *.ch
abc1.ch   abc2.ch   abc3.ch
>
```

To list files whose names contain the string c1, type

```
> ls *c1*
abc1.ch   abc12.c   efc1.c
```

To list files whose names start with the string abc, end with the string .ch, and have only one character between these two strings, type

```
> ls abc?.ch
abc1.ch   abc2.ch   abc3.ch
>
```

The user can specify the home directory as the tilde character ~. For example, regardless of the current working directory, the ~ abbreviation can be used to list the files in the home directory. Substituting ~ in place of the home directory pathname requires less typing and does not change the current working directory. For example, assume that the user's account name is myname,

```
> // print the current user name
> echo $_user
myname
> // list files in  home directory of current user
> ls ~
... (list files in home directory of myname)> pwd
>
```

The user can specify the current working directory name as ./. This substitution is useful in a variety of situations. For instance, when the user wants to copy a file from a distant directory into the working directory, the command **cp** followed by the pathname of the file and the current working directory abbreviation ./ can be used. For example, the following commands

```
> pwd
/home/myname/project2
> cp /home/myname/project1/subproject2/plan1/* .
>
```

Table 4.10: Filename substitution.

wild card character	Description
*	Match any (zero or more) characters.
?	Match any single character.
~	The home directory, as indicated by the value of the variable _home, or that of _user, as indicated by the password entry for user.
./	current working directory.
../	the parent directory of the current working directory.

will copy all files in the directory `/home/myname/project1/subproject2/plan1` into the current working directory. Another use of the current working directory abbreviation `./` is to make sure a program in the current directory is running. A commonly used file name, such as `test`, could be used by more than one program in different directories. If the current directory is not included in the search paths specified by the variable **_path**, or it has a lower priority than other directory which has a file called `test` too, typing the file name `test` will execute program `test` in another directory instead of the program in the current directory. The command `./test.c` will ensure that the execution of the program is in the current directory. The command `which -a test` can be used to list all programs named `test` in order of search paths. In the following example, the current directory is not included in the search paths.

```
> pwd
/home/myname/project1/subproject2/plan1
> which -a test
/bin/test
/usr/bin/test
/usr/local/gnu/bin/test
/pkg/gnu/bin/test
> test    // execute /bin/test
> ./test  // execute /home/myname/project1/subproject2/plan1/test
```

The user can specify the parent directory of the current working directory as `..`. The most common use of the parent directory abbreviation is to switch the current working directory into the parent directory or its subdirectories.

```
> // print current working directory
> pwd
/home/myname/project1/subproject2/plan2
> // go to directory plan1 of the parent directory
> cd ../plan1
> pwd
/home/myname/project1/subproject2/plan1
>
```

In the example, the command `cd ../plan1` changes the current working directory to the subdirectory `plan1` of the parent directory `/home/myname/project1/subproject2`.

4.9 Command Substitution

Command substitution pipes the output of one command into a variable inside a program or the command shell. This is accomplished by enclosing the embedded command in a pair of accent grave marks `‘`, which are sometimes called *back quotation marks*. For example,

```
> string_t s = ‘date‘
Wed Jul 25 10:11:18 PDT 2001
> s
Wed Jul 25 10:11:18 PDT 2001
> char *s1 = ‘date‘
> s1
Wed Jul 25 10:12:15 PDT 2001
> s1[0]
W
> free(s1)  // the memory should be freed
>
```

Commands `string_t s = ‘date‘` and `char *s1 = ‘date‘` pipe the output of command **date** to variables `s` and `s1`, respectively. Unlike aliases of commands, variables `s` and `s1` are not equivalent to the command **date**. They only store the output of the command **date**. It means that the contents of `s` and `s1` won't change as time changes, whereas the output from execution command `date` will change as the time changes. Note that the memory allocated for variable `s1` should be freed later. It is recommended to use the type **string_t** instead of the type `char *` to implement command substitution. More information about **string_t** can be found in section 17.2. Another example for command substitution is that the Unix command

```
vi ‘grep -l "str1 str2" * ‘
```

can be used to edit all files that contain the string *str1 str2* in the current directory using the *vi* text editor.

The variable substitution described in section 4.7 can be used inside a command substitution. The variable can be name in Ch space or environment space. A valid Ch expression can also be used for variable substitution. For example,

```
> string_t s1 = "/bin", s2;
> s2 = ‘ls $s1‘;
... (list of /bin)
> echo ‘ls $s1‘;
... (list of /bin)
```

Note that variable substitution can be prevented by preceding the ‘$’ with a ‘\’ except within ‘’’s for command substitution where it always occurs, and within ‘’’s where it never occurs. A ‘$’ is passed unchanged if followed by a blank, tab, or end-of-line. For example, if the variables `f1` and `f2` have values of `file1` and `file2`, respectively, the expression

```
‘echo $f1 \$f2 |sed ’s/endofline$/converted/’‘
```

is equivalent to

```
‘echo file1 \file2 |sed ’s/endofline$/converted/’‘
```

To make it easier for the user to refer to each item from the word list of a command substitution individually, the output from a command embedded in a pair of accent grace marks ‘ is postprocessed. The consecutive blank space characters, characters for form feed, new line, carriage return, horizontal and vertical tabs are replaced by single blank space characters. This is consistent with the C shell.

The output from a command embedded in a pair of double accent grave marks ‘‘ are intact. For example, the extra blank from the output of the command **date** is retained by the command substitution ‘‘date‘‘.

```
> string_t s = ‘‘date‘‘
> s
Mon Aug  6 11:44:16 PDT 2001
> s = ‘date‘
Mon Aug 6 11:44:24 PDT 2001
>
```

The two blank space characters after word Aug from output of the command ‘‘date‘‘ are retained. They are replaced by a single blank space in command ‘date‘.

4.10 Input/Output Redirection

In Ch, a command's input and output may be redirected using a special notation interpreted by the shell following the convention of Bourne shell. The redirection notations listed in Table 4.11 may appear in a typed-in command in an interactive shell or command line in a Ch program.

Through output redirection, the command `cmd > output` followed by the quotation can be used to create the file `output` in the command mode. By typing the ‘>’ symbol, we redirected the output from the `cmd` command into the file `output`. The system took what the command `cmd` would have printed out to the screen and put it into the file `output` instead.

By using the symbol ‘>’, when the user redirects output into a file that already exists, the output redirection will remove the current contents of that file and replace them with the output of the command. The user can avoid overwriting the contents of a file by using another redirection symbol ‘>>’ which is called the *append* redirection symbol. It adds the data to the end of a file, rather than replacing the file. If the user appends output to a file which doesn't exist, the symbol ‘>>’ acts like ‘>’, creating the file and redirecting the output of a command into it.

A process associates a number with each opened file. This number is called *file descriptor*. When Ch is launched in Unix, it is connected to three files, standard input which has file descriptor 0, standard output which has file descriptor 1, standard error which has file descriptor 2. The standard error is not available in Windows. The user can redirect any file descriptor from 0 through 9 by specifying the file descriptor number before the symbols ‘>’, ‘<’, and ‘>>’. For instance, to redirect standard error, use `2>`. The user can redirect the output from standard output and standard error to the same file `output` by command `cmd > output 2> output`. The user can also specify that the file descriptor `n` be redirected to the same file as another file descriptor `m` by symbol ‘`n>&m`’. For instance, command `cmd > output 2>&1` redirects the output of command **cmd** to file `output`, and then redirects standard error there.

The following commands illustrate how input/output redirection in Ch shell works.

```
> cat datefile
old content
> date > datefile
> cat datefile
```

Table 4.11: Input/Output redirection.

Notation	Description
cmd < *word*	Use file *word* as standard input (file descriptor 0).
cmd > *word*	Use file *word* as standard output (file descriptor 1). If the file does not exist, it is created; otherwise, it is truncated to zero length.
cmd > *word* 2>&1	Redirect the standard output and standard error (diagnostic output) to file *word* (for Unix).
cmd 1 > *word* 2>&1	Redirect the standard output and standard error (diagnostic output) to file *word* (for Windows).
ch -r cmd > *word*	Redirect the standard output and standard error (diagnostic output) to file *word* (for both Unix and Windows). If the file does not exist, it is created; otherwise, it is truncated to zero length.
cmd >> *word*	Use file *word* as standard output. If the file exists, output is appended to it (by first seeking to the EOF); otherwise, the file is created.
cmd >> *word* 2>&1	Redirect the standard output and standard error (diagnostic output) to file *word*. If the file exists, output is appended to it (by first seeking to the EOF); otherwise, the file is created (for Unix).
cmd 1 >> *word* 2>&1	Redirect the standard output and standard error (diagnostic output) to file *word*. If the file exists, output is appended to it (by first seeking to the EOF); otherwise, the file is created (for Windows).
cmd << *word*	After parameter and command substitution is done on *word*, the shell input is read
cmd << *word*	After parameter and command substitution is done on *word*, the shell input is read up to the first line that literally matches the resulting *word*, or to an EOF.

```
Wed Jul 25 17:10:40 PDT 2001
>
> date >> datefile
> cat datefile
Wed Jul 25 17:10:40 PDT 2001
Wed Jul 25 17:11:45 PDT 2001
>
> cat > p1.c
int i, j
(To complete the file and quit from the command cat,
 use Ctrl-D in Unix or Ctrl-Z in Windows)
> p1.c > result_p1 2>&1
> cat result_p1
ERROR: missing ';'
ERROR: syntax error before or at line 2 in file p1.c
==>:
BUG: <== ???
WARNING: cannot execute command 'p1.c'
>
> cat > input_p2
10
(To complete the file and quit from the command cat,
 use Ctrl-D in Unix or Ctrl-Z in Windows)
> cat > p2.c
int i;
scanf("%d", i);
printf("%d\n", i);
(To complete the file and quit from the command cat,
 use Ctrl-D in Unix or Ctrl-Z in Windows)
> p2.c < input_p2
10
>
```

In this example, output of command **date** is redirected to file `datefile` by symbol '`>`' first. The content in `datefile` is overwritten. Then the second output of command **date** is redirected to file `datefile` by symbol '`>>`'. It is appended to the end of file `datefile`, rather than overwriting it. The error messages of the execution of file `p2.c` are redirected into the file `result_p1` by the command `p1.c > result_p1 2>&1`. Using the symbol '`<`', interactive execution of file `p2.c` gets the input from file `input_p2` instead of keyboard which is the default standard input device.

Note that the syntax

```
cmd >& word
```

for redirecting both standard output and standard error to file `word` works in C shell, but not in Ch and Bourne shells. Use

```
ch cmd > word 2>&1
```

in Ch and Bourne shells in Unix.

There is a command option `-r` in Ch, which can be conveniently used to redirect the standard error stderr to the standard output stdout. The command below

```
ch -r cmd > word
```

will redirect *both* standard output and standard error to the file `word`. This syntax works in both Unix and Windows.

In Unix, the function **system()** can be used to redirect the stdout of a command, say `cmd`, to file word1 and the stderr to file `word2` as shown below.

```
system("(cmd > word1) 2> word2");
```

4.11 Pipeline

A pipeline is a sequence of one or more commands separated by the symbol `|` . The standard output of each command except the last one is connected by a pipe to the standard input of the next command. Each command runs as a separate process; the shell waits for the last command to terminate. The exit status of a pipeline is the exit status of the last command in the pipeline. Users can regard it as running the first command with its output redirected to a temporary file, then running the second command with its input redirected from the temporary file, and so on.

The common use of a pipeline is preprocessing or postprocessing the output of a command by connecting one or more filters together. A command that reads from the standard input and writes to the standard output is called a *filter*. For example, the command **grep** which displays the lines in one or more files that contain a specified string. If the user wants to look for the definition of typedefed type **time_t** in the header file directory `/usr/ch/include`, the commands below with a pipeline can be used,

```
> pwd
/usr/ch/include
> grep time_t *.h | grep typedef
time.h:typedef int    time_t;
>
```

The output of command `grep time_t *.h` is piped to `grep typedef`. It means the pipeline `grep time_t *.h | grep typedef` lists all lines which include both strings `time_t` and `typedef` in header files in the directory `/usr/ch/include`. It is more effective than only using `grep time_t *.h` or `grep typedef *.h` which may give the user many useless output. The command below only displays the status of processes whose owner is `myname` and command is **vi**.

```
> ps -elf | grep myname | grep vi
1 S     myname 20851 20850  0 156 20  19d0500  115  1086564 23:36:47
ttyp5 0:01 /bin/vi example.txt
>
```

The command `ps -elf` lists all processes status in detail. Its output is piped to command `grep myname` as input. Then the command `grep myname` lists lines including string `myname`. These lines are piped to command `grep vi` as input. The pipelined command lists all processes status lines which include both strings `myname` and **vi**.

As another example, a compressed archival file `file.tar.gz` can be retrieved by the command below

```
gzip -cd file.tar.gz |tar -xvf -
```

The command **gzip** unzips the file `file.tar.gz` first. The uncompressed file is then piped as the standard ouput for command **tar** to extract files. To copy files in the directory `/home/from` to directory `/home/new/from` without changing the permission mode and access time, the following command can be used.

```
tar cf - /home/from | (cd /home/new; tar xf - )
```

Function **popen()** and **pclose()** can also be used to pipe output from one program to other one. For example, the **ed** command in the following Bourne shell program

```
#!/bin/sh
ed testfile  <<END
a
input from the console command line
abcd
123456
.
w
q
END
```

will edit the file `testfile` using the subsequent lines, up to the first occurrence of delimiter line `END`, as input. The user doesn't have to use "`END`" as the delimiter, any word will do. The editor **ed** will append the file with the three text lines shown below.

```
input from the console command line
abcd
123456
```

ed is a line-oriented text editor. **a**, '**.**', **w** and **q** in the Bourne shell program are commands of **ed**. The command **a** ("append") tells **ed** to start collecting text; the '**.**' is a command to signal the end of the text; the command **w** ("write") stores the information into file `testfile`; the command **q** ("quit") leaves the editor.

The result of the previous bourne shell program can be achieved using a Ch program by piping data to a process as follows:

```
#!/bin/ch
#include <stdio.h>
FILE *fp;
string_t command_args="a\ninput from the console command line\
\nabcd\n123456\n.\nw\nq";
fp = popen("ed testfile", "w");
fwrite(command_args, strlen(command_args), sizeof(char), fp);
pclose(fp);
```

where the command and input lines for editor **ed** in previous Bourne shell program are replaced by a variable of type **string_t**, `command_args`. In the string of the variable `command_args`, these command and input lines are put together and separated by new line characters '`\n`'. Functions **popen()** and **pclose()** are used to initiate and close the piped I/O to the process for command `ed testfile`, respectively. The first argument of function **popen()** is a string containing the shell command line `ed testfile`; the second argument of `w` is the I/O mode indicating the operation is writing data to the process through the pipe. Function **fwrite()** sends the data to a stream pointed to by the stream `fp` for the process of shell command `ed testfile`.

4.12 Running Commands in Background

By default, Ch shell in command mode waits for a command to finish execution before the next prompt is displayed. The command for which Ch shell waits is called a foreground command. A background

command is a process that is started asynchronously within a Ch shell. Before the background command is completed, the Ch shell will display the next prompt and readily accept the input from the command line. In program mode, the control flow of the program will go to the next statement immediately before the background command finishes execution. A command or pipeline ending with an & will be treated as a background command. If the output of a background command is not redirected, it will be displayed at the terminal.

A background command is useful for handling commands that take a long time to execute. It is also useful to start a command with event-driven graphical user interface. For example, command `notepad` in Windows can be started as a background command as shown below.

```
> notepad &
```

4.13 Run-Time Expression Evaluation

The generic function **streval()** can be used to evaluate an expression expressed in string at run time. The expression may invoke functions located in function files specified by the function file path `_fpath`. This function is polymorphic and can only be used as an rvalue. For example,

```
> int i = 1
> float f = 10.1
> string_t s
> char a[10], *p
> i = streval("i*2")  /* i becomes 2 */
2
> s = "f*i"
f*i
> f = streval(s)      /* f becomes 20.20 */
20.20
> strcpy(a, s)
f*i
> strcat(a, "+5")
f*i+5
> f = streval(a)      /* f becomes 45.40 */
45.40
> p = a
f*i+5
> f = streval(p)      /* f becomes 95.80 */
95.80
> streval("23unknown")
Invalid argument for streval()
```

The generic function **strparse()** returns 0, if the expression can be converted to a numerical value at runtime. Otherwise, it returns non-zero as illustrated below.

```
int i =90, status;
status = strparse("i*2");
if(!status)  {
   i = streval("i*2");
   printf("i = %d\n", i);
}
```

In some applications, the string value for an argument of **streval()** is passed through the command interface. For example, program `command` may obtain strings `x` and 10 , or `x+sin(90)*9` and `10`. The parentheses may need to be escaped as shown below.

```
command x 10
command x+sin\(90\)*9 10
```

4.14 Handling Environment Variables

The environment variables in Ch are similar to the environment variables in other Unix shells and MS-DOS shell. There are four environment variable handling functions in Ch. Function **putenv()** can add an environment variable to the system. This function is commonly used in the system startup file **CHHOME/config/chrc** and individual user's startup file **.chrc** in Unix and **_chrc** in Windows in the user's home directory. Given an environment variable, function **getenv()** can get its corresponding value. Function **remenv()** can remove an environment variable. Function **isenv()** can test if a symbol is an environment variable. The interactive command execution below demonstrates their application.

```
> putenv("ENVVAR=value")
0
> getenv("ENVVAR")
value
> isenv("ENVVAR")
1
> remenv("ENVVAR")
> isenv("ENVVAR")
0
>
```

In this example, the value `value` has been set to the environment variable `ENVVAR` by using command `putenv("ENVVAR=value")`. Note that in C there shouldn't be blanks space adjacent to the equal sign '='. After that, the command `getenv("ENVVAR")` obtains `value`, the value of the environment variable `ENVVAR`. Because `ENVVAR` is an environment variable, the function call `isenv("ENVVAR")` returns `1`. After invoking function **remenv()** to remove `ENVVAR` from environment variables, the the function call `isenv("ENVVAR")` returns `0`.

As an example of application, the environment variable **DISPLAY** can be set in the remote machine to the name of a local machine `local.domain.com` so that the remote machine `remote.domain.com` can send graphic output to the machine `local.domain.com` through the network via X-window. On the local machine `local.domain.com`, the command

```
> xhost server.domain.com
server.domain.com being added to access control list
>
```

should be used to add the remote machine into the access control list. On the remote machine `remote.domain.com`, the command

```
> putenv("DISPLAY=local.domain.com:0.0")
>
```

sets the environment variable **DISPLAY** to make the server send graphic output to `local.domain.com`.

Programs described in section 10.10 can be used to print out all environment variables and their corresponding values. In Unix, a system command **env** can also be used to display all environment variables.

Environment variables are passed to all commands and programs running from within the current shell. A Ch subshell inherits a copy of the environment from its parent shell. On the other hand, changes to the values of environment variables in a subshell will not affect its parent shell. For example,

```
> putenv("ENVVAR=value")
0
> getenv("ENVVAR")
value
> cat > changeenv.ch
  #!/bin/ch
  printf("%s\n", getenv("ENVVAR"));
  putenv("ENVVAR=value2");
  printf("%s\n", getenv("ENVVAR"));
(To complete the file and quit from the command cat,
 use Ctrl-D in Unix or Ctrl-Z in Windows)
> changeenv.ch
value
value2
> getenv("ENVVAR")
value
>
```

In this example, program `changeenv.ch` runs in a subshell which has a copy of all environment variables including `ENVVAR` from its parent shell. After changing the value of the copy of `ENVVAR` in the subshell to `value2`, the value of `ENVVAR` in the parent shell is still `value`. As it is described in section 4.4, a program can be executed in the current shell using dot command. In the example below, command `. changeenv.ch`, changes the environment variable `ENVVAR` in the current shell.

```
> getenv("ENVVAR")
value
> . changeenv.ch
value
value2
> getenv("ENVVAR")
value2
>
```

The values of environment variables at the command line in the interactive Ch shell, and command statements in a program can be obtained by variable substitution described in section 4.7. The value of environment variable `ENVVAR` can be obtained by either `$ENVVAR` or `$(getenv(ENVVAR))` as shown below.

```
> getenv("ENVVAR")
value
> echo $ENVVAR
value
> echo $(getenv("ENVVAR"))
```

Table 4.12: General-Purpose Ch Programs.

Command	Description
ch	Ch shell
dirs	List directories in the dir stack, C shell compatible
help	Getting started in Ch
popd	Pop the first directory from the dir stack, and switch to that directory, C shell compatible
popd +*n*	Pop the *n*th directory from the dir stack, and switch to that directory, C shell compatible
pushd	Switch to the second directory in the stack, C shell compatible
pushd *dirname*	Push directory *dirname* onto the dir stack, and switch to that directory, C shell compatible
pushd +*n*	Switch to the *n*th directory in the stack, C shell compatible
chs	Safe Ch shell
which [-a]	Similar to **which** in C shell. Used to show the location of a token

```
value
>
```

As illustrated in Appendix D, an environment variable can be setup in C shell using its shell command `setenv`. In sh, bash, ksh shells, it can be setup by the shell command `export`. As an example, the command

```
putenv("DISPLAY=local.domain.com:0.0")
```

in Ch shall be handled in C shell as

```
setenv DISPLAY local.domain.com:0.0
```

In sh, bash, and ksh shells, it can be setup as

```
DISPLAY=local.domain.com:0.0
export DISPLAY
```

4.15 General-Purpose Ch Programs

The commands listed in Table 4.12 are general-purpose Ch programs. Command **ch** or **chs** is used to start a new regular Ch or safe Ch, respectively. Command **which** is used to tell where a specified executable program is found. Commands **dirs**, **pushd** and **popd**, borrowed from C shell, can maintain a directory stack which allows the users to conveniently switch among several working directories. The user can use these in place of **cd** for changing directories. Command **pushd** is used to travel between two directories on the top of the stack. Command **pushd** `dirname` pushes the `dirname` into the stack, and switches the current working directory there. Command **pushd** `+n` changes the current working directory to the `nth` directory on the stack. Command **popd** removes the top directory from the stack. Command **popd**`+n` removes the `nth` directory from the stack. Command **help** can help new users of Ch get started with some illustrative examples.

The following commands illustrate how these general-purpose programs can be used in Ch shell.

```
> which ch
/usr/ch/bin/ch
> which ch ls
/usr/ch/bin/ch
ls is aliased to ls -F
> which -a ls stdio TERM
ls is aliased to ls -F
/bin/ls
/usr/bin/ls
/usr/ucb/ls
/usr/ch/include/stdio.h
dtterm
> pwd
/usr/ch
> pushd /usr/ch
0  /usr/ch
> pwd
/usr/ch
> pushd /home/myname
0  /home/myname
1  /usr/ch
> pwd
/home/myname
> pushd
0  /usr/ch
1  /home/myname
> pwd
/usr/ch
> pushd
0  /home/myname
1  /usr/ch
> pwd
/home/myname
> dirs
0  /home/myname
1  /usr/ch
> popd
0  /usr/ch
> dirs
0  /usr/ch
>
```

In this example, the **which** command first tells where the executable program of the **ch** command is found. It can handle multiple commands as shown with commands **ch** and **ls**. In this case, **ls** is an alias. If the option `-a` is used, all aliases, executable programs in the paths specified by predefined identifier **_path**, function files in the paths pointed by **_fpath**, environment variable, and header files in the paths pointed by **_ipath** are displayed. Without option `-a`, only the first available command, function file, and header file, or the value of an environment variable are displayed. With option `-v`, if commands cannot be found, all paths

Table 4.13: The first line of shell programs for different shells.

Shell	The first line
Ch shell	`#!/bin/ch`
C shell	`#!/bin/csh`
Bourne shell	`#!/bin/sh`
Korn shell	`#!/bin/ksh`
BASH	`#!/bin/bash`

in **_path** will be displayed. In the above example, besides an alias for **ls**, all possible executable commands in the paths are listed. Symbol `stdio` is a header file `stdio.h` with file extension `.h`. The value for environment variable `TERM` is `dtterm`. After pushing two working directories into the stack, the command **pushd** is used to switch between these two directories.

Commands listed in Table 4.12 are accessible by regular Ch shell only, not by safe Ch shell. However, Ch shell can also be invoked by safe Ch shell

4.16 Shell Programming

4.16.1 Use Shell Commands in Programs

Unlike C shell, the syntax and control flow of Ch are C compatible. C programs can readily run in a Ch shell. However, if the execution speed of a program is not a major concern, it is often more convenient to use shell commands in a Ch program. Based on the existing commands, sometimes, a task which needs thousands of lines of C code can be accomplished with only a few lines of Ch code. Appendix G in gives a list of commands commonly used for shell programming portable across different platforms in Ch.

A Ch shell script typically does not contain function `main()` or `WinMain()` in Windows. In Ch shell, a file with the name extension specified in the system variable **_pathext** will be treated as a Ch script file, regardless of its content. Otherwise, Ch will analyze the content. Typically, the shell for which a script file is written is indicated by the first line of the file. They are shown in Table 4.13 for popular Unix shells. If the first line of a program contains the statement `#!/bin/ch`, it will be treated as a Ch shell script. Ch scripts can be executed even in other shells such as C shell or Bourne shell. If the file extension of a shell script is included in the system variable **_pathext**, the script will be treated as a Ch shell script even if the first line indicates that it is not a Ch program. In this case, the program may not run successfully as a Ch program.

If the file extension of a program is not contained in the system variable **_pathext**, and the program does not start with the statement `#!/bin/ch` or other tokens described in section 3.3, Ch will invoke other shell to execute it.

A command invoked inside a Ch program is called a *command statement*. The constraints for a command statement, say `cmd`, are as follows.

- It shall be a valid identifier or an identifier with file extension, such as `cmd.ext`.
- It shall not be a declared variable within the scope, except that it is preceded with an absolute or relative path as shown below for command `cmd`.

```
/path/cmd
./cmd
```

```
../cmd
~/cmd
```

- If the command name is also a declared variable within the scope, the command can be enclosed inside a pair of double quotation marks. The option for the command shall not be included inside the quotation marks. It can also be used in the case that the command is located in a directory with white space. For example,

```
int ls = 10;
"ls" -1
"/ch/bin/echo" option $PATH
"C:/Program Files/Windows NT/Accessories/wordpad.exe"
```

- A command can be obtained dynamically at run time by command name substitution. The data type of the variable or expression following the $ sign for command name substitution. shall be string, pointer to char, or pointer to unsigned char. For example,

```
string_t s = "cmd";
$s option
```

  In the case, the symbol `cmd` can also be used as a variable name independent of the command name.

- It shall be enclosed with a pair of double quotation marks following a dot '`.`' such as

```
. "cmd"
```

  In this case, the command `cmd` is executed in the current shell as if it is included by the preprocessing directive `include`, except that the system variable **_path**, instead of **_ipath**, is searched for the program. It is similar to

```
#pragma import "cmd"
```

In Windows, if a Ch shell script, say `cmd.ch`, is used in other programs, such as in a Makefile, it may need to be invoked by Ch as follows.

```
ch cmd.ch
```

or

```
ch cmd
```

These two formats of command execution start Ch shells explicitly to execute the script file `cmd.ch`.

The variable substitution described in section 4.7 can be applied to variables used in command statements. The ending semicolon for other programming statements is not required for command statements. For example, assume the shell script `cpfile1.ch` contains the following statements

```
#!/bin/ch
cp /dir/source/*.ch /dir/dest/
```

There is no ending semicolon for the command statement starting with Unix command `cp`, which copies files specified by the first argument to the directory specified by the second argument. Execution of the above shell script

```
#!/bin/ch
#include <stdio.h>
#include <unistd.h>
string_t file, files = ‘ls ./‘;
string_t newfile="newfile";
string_t allfiles= stradd(_home, "/allfiles");
int i;

if (access(newfile, F_OK) == 0) // clear up first
    remove(newfile);
foreach(file; files) {
    if (access(file, R_OK) == 0) {
       i++;
       echo $i $file >> $newfile
       echo $i $file >> $allfiles
    }
}
```

Program 4.2: Handle I/O use shell scripts.

```
> cpfile1.ch
>
```

copies all files with extensions `.ch` from directory `/dir/source` to directory `/dir/dest`.

The **foreach** loop described in section 8.4.4 is very useful for shell programming. For example, the program below will print names of all files in the current directory. File names in the current directory are obtained by command `ls` and sorted by a foreach-loop.

```
#!/bin/ch
string_t file, files = ‘ls ./‘;
foreach(file; files) {
  printf("file = %s\n", file);
}
```

Although the output from a program can be handled by a family of I/O functions **fopen(), fclose(), fprintf(), etc.** described in Chapter 20, it is often time more convenient to use shell commands to send output from a program to files. For example, in Program 4.2, file names with read permission in the current directory are saved in file `newfile`. If the file `newfile` already exists, it will be deleted first by the function **remove()**. Whether `file` exists or not is tested by the function call of `access(file, F_OK).`

The function **access()**, which is defined in the header file **unistd.h**, checks the accessibility of the file named by the pathname pointed to by the first argument. The real user ID in place of the effective user ID and the real group ID in place of the effective group ID are used to allows a setuid process to verify that the user running it would have had permission to access this file. The value of the second argument, which is of type **int**, is either the bitwise inclusive **OR** of the access permissions to be checked (**R_OK**, **W_OK**, and **X_OK**) or the existence test (**F_OK**). These symbolic constants are described in Table 4.14. If the requested access is permitted, zero shall be returned. Otherwise, `-1` shall be returned and `errno` shall be set to indicate the error.

In Program 4.2, whether `file` is readable or not is checked by function call of `access(file, R_OK).` File names are also appended in file `allfiles` located in the user's home directory by command **echo** with

Table 4.14: Symbolic Constants for the function **access()**.

Constant	Description
R_OK	Test for read permission.
W_OK	Test for write permission.
X_OK	Test for execute or search permission.
F_OK	Test for existence of file.

```
#!/bin/ch
string_t result1;
char result2[50];

grep "test" myfile.txt;
if ( _status == 0 ) {
    printf("'test' is found in myfile.txt\n");
}
else {
    printf("Cannot find 'test' in myfile.txt\n");
}
result1=‘wc -l /etc/passwd‘;
echo $result1;
strcpy(result2, ‘wc -l /etc/passwd‘);
printf("%s\n", result2);
```

Program 4.3: Combination of shell commands with C code.

output redirection. A file name is preceded with a sequential number. For example, if the current directory contains files `file1`, `file2, file3`. the output from executing Program 4.2

```
1 file1
2 file2
3 file3
```

will be redirected to file `newfile` in the current directory and at the same time appended in file `allfiles` in the user's home directory.

For shell programming, section 20.9 uses function **stat()** and directory handling functions to obtain detailed information such as the size, access time, user id, etc. of files in a directory and its subdirectories recursively.

Executable programs can be used directly from a Ch script. Shell commands such as sed, awk, wc, grep, etc. can be combined with C code to run in Ch as shown in Program 4.3 The output for `tnum1` and `tnum2` in Program 4.3 is the same.

4.16.2 Passing Values to Shell Commands

This section describes how values from command line arguments can be passed to Ch shell programs. In a Ch shell program, two predefined identifiers **_argc** and **_argv** are used to handle arguments from the command line. They are defined internally with types of **int** and **char **** as follows,

```
int    _argc;
char* _argv[];
```

Identifier **_argc** contains the number of the arguments on the command line which includes the command name itself. Identifier **_argv** maintains the argument list on the command line. The shell stores the command name in **_argv[0]**, the first argument in **_argv[1]**, and so on. For example, assume file `argtest.ch` has the following statements.

```
#!/bin/ch
echo $_argc
echo $(_argv[0])
echo $(_argv[1])
printf("%s\n", _argv[2]);
printf("%s\n", _argv[3]);
```

The execution of `argtest.ch` is shown below.

```
> argtest.ch -option arg2
3
argtest.ch
-option
arg2
(null)
>
```

In this example, the value of **_argc** is 3, which includes file name `argtest.c` and two arguments `arg1` and `arg2`. The filename is stored in the variable **_argv[0]**; the first argument `arg1` is in **_argv[1]**; the second argument `arg2` in **_argv[2]**. Comparison of C shell and Ch for accessing arguments in the command line are listed in Appendix D.

Program 4.4 is an example for handling command-line arguments with **_argc** and **_argv**. It can be used to implement the **which** command. Here, the variables `a_option` and `v_option` indicate that the valid options `-a` and `-v` are on or not. Their values are `false` by default. If there is no command-line argument, the program will print out the error message, because the command **which** at least has one argument, i.e. the name to be searched for. The while-loop in this program handles all arguments which begin with the minus sign `-`, and the for-loop analyzes these arguments character by character. The statement

```
c = _argv[i][j++];
```

makes variable `c` equal the `j`th character in the argument `_argv[i]`. If the characters 'a' and 'v' are found in these arguments, the variables `a_option` and `v_option` are set to true, respectively. If other characters are found, the error messages will be printed out. At the end of Program 4.4, options and the remaining command-line arguments are printed out. Assume that the file name of Program 4.4 is `commandline.ch`, the results from executing Program 4.4 with different options are shown below.

```
> commandline.ch -a -v arg1
option -a is on
option -v is on
arg1
> commandline.ch -av arg1
option -a is on
option -v is on
```

```
#include <stdio.h>
#include <stdbool.h>

int main() {
  int i = 0;            // for index of arguments
  int j = 0;            // for index of characters in arguments
  char c;
  int a_option = false; // default, no -a option
  int v_option = false; // default, no -v option

  if(_argc == 1){        // no argument
    fprintf(stderr, "Usage: which [-av] names \n");
    exit(1);
  }

  _argc--; i++; j = 0;
  while(_argc > 0 && _argv[i][j++] == '-') {
      // for every argument beginning with -
      // empty space is not valid option
      for(c = _argv[i][j++]; c&&c!=' '; c = _argv[i][j++]) { // for -av
        switch(c)
        {
          case 'a':
            a_option = true;      // get all possible matches
            break;
          case 'v':
            v_option = true;      // print message
            break;
          default:
            fprintf(stderr,"Warning: invalid option %c\n", c);
            fprintf(stderr, "Usage: which [-av] names \n");
            break;
        }
      }
      _argc--; i++; j = 0;
  }

  if(a_option)
    printf("option -a is on\n");
  if(v_option)
    printf("option -v is on\n");
  while(_argc > 0) { // print out the remaining arguments
      printf("%s\n",  _argv[i]);
      _argc--; i++;
  }
  return 0;
}
```

Program 4.4: Handle command-line arguments with **_argc** and **_argv**.

```
arg1
> commandline.ch -v arg1 arg2
option -v is on
arg1
arg2
```

The similar program which handles command-line arguments with pointers to pointers is discussed in section 10.10.

Assume program `cpfile2.ch` contains the following statements.

```
#!/bin/ch
cp /dir/source/$(_argv[1]) /dir/dest/
```

Program `cpfile2.ch` can be used to copy files located at directory `/dir/source` to directory `/dir/dest` conveniently. For example, the commands below

```
> cpfile2 file1
> cpfile2 file2
>
```

will copy files `file1` and `file2` from `/dir/source` to directory `/dir/dest`.

Chapter 5

Preprocessing Directives

At its current implementation, Ch is interpretive. It has no separate translation stage such as preprocessing. But, the syntaxes of preprocessing directives in C are supported in Ch. For the sake of convenience, we also call them preprocessing directives in Ch. A preprocessing directive consists of a sequence of preprocessing tokens that begins with a # preprocessing token. The preprocessing directives are listed in Table 5.1. The details about these directives will be described in this chapter.

Table 5.1: Preprocessing directives.

Directive	Description
`#define`	Define a preprocessor macro.
`#elif`	Alternatively include some text based on the value of another expression, if the previous `#if, #ifdef, #ifndef`, or `#elif` test failed.
`#else`	Alternatively include some text, if the previous `#if, #ifdef, #ifndef`, or `#elif` test failed.
`#endif`	Terminate conditional text.
`#error`	Produce a compile-time error with a designated message.
`#if`	Conditionally include text, based on the value of an expression.
`#ifdef`	Conditionally include text, based on whether a macro name is defined
`#ifndef`	Conditionally include text, based on if a name is not defined macro
`#include`	Insert text from another source file.
`#line`	Give a line number for message.
`#pragma`	Ch specific features, not in C standard.
`#undef`	Remove a preprocessor macro definition.
`#`	Replace a maco parameter with a string constant containing the parameter's value
`##`	Create a single token out of two adjacent tokens.
`#`	Null directive.
`defined`	Preprocessing operator that yields 1 if a name is defined as a preprocessing macro and 0 otherwise; used in `#if` and `#elif`.

5.1 Conditional Inclusion

Preprocessing directives of the forms

```
# if   expr1
# elif expr2
```

check whether the controlling expression evaluates to nonzero. The expression that controls conditional inclusion shall be an integer expression except that it shall not contain declared identifiers. It may contain preprocessing operation

```
defined (identifier)
```

which evaluates to 1 if the identifier is currently defined as a macro name (that is, if it is predefined or if it has been the subject of a **#define** preprocessing directive without an intervening **#undef** directive with the same subject identifier), 0 if it is not. Ch has extensions to C in preprocessing directives. All operators and generic functions such as `strcat(), and strcmp(), access()` can be used in arguments of preprocessing directives `#if` and `#elif` in Ch. For example,

```
#if defined(_HPUX_)
  printf("I am using HP-UX\n");
#elif !strcmp(‘uname‘, "Linux")
  printf("I am using Linux\n");
#endif
```

Preprocessing directives of the forms

```
# ifdef  identifier
# ifndef identifier
```

check whether the identifier is or is not currently defined as a macro name. Their conditions are equivalent to **#if defined** *(identifier)* and **#if !defined** *(identifier)*, respectively.

Each directive's condition is checked in order. If it evaluates to false (zero), then the group that it controls is skipped: directives are processed only through the name that determines the directive in order to keep track of the level of nested conditionals; the rest of the directives' preprocessing tokens are ignored, as are the other preprocessing tokens in the group. Only the first group whose control condition evaluates to true (nonzero) is processed. If none of the conditions evaluates to true, and there is a **#else** directive, then the group controlled by the **#else** is processed; if lacking a **#else** directive, then all the groups until the **#endif** are skipped.

5.2 Source File Inclusion

A **#include** directive identifies a header or source file that can be processed by the Ch interpreter. A preprocessing directive of the form

```
#include <h-char-sequence>
```

searches for a header identified uniquely by the specified sequence between the $<$ and $>$ delimiters, and causes the replacement of that directive by the entire contents of the header. The header is searched according to the paths specified in predefined identifier **_ipath** of string type. Each path is delimited by a semicolon. By default, the variable **_ipath** contains string `"CHHOME/include;CHHOME/toolkit/include;"` where CHHOME is the home directory of the Ch software. The variable **_ipath** for the search path is typically setup in a startup file **.chrc** in Unix and **_chrc** in Windows in the user's home directory.

A preprocessing directive of the form

```
#include "q-char-sequence"
```

causes the replacement of that directive by the entire contents of the source file identified by the specified sequence between the " delimiters. The named source file is searched for in the current directory first, then in the directory specified in the system variable **_ipath**.

A preprocessing directive of the form

```
#include pp-tokens
```

that does not match one of the two previous forms is permitted. The preprocessing tokens after **include** in the directive are processed just as in normal text. Each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens. The directive resulting *after all replacements* shall match one of the two previous forms. A **#include** preprocessing directive may appear in a source file that has been read because of a **#include** directive in another file. There is no limit to the nesting level of the #include directives.

The most common uses of #include preprocessing directives are given as follows:

```
#include <stdio.h>
#include "myprog.h"
```

The following code fragment illustrates macro-replaced #include directives:

```
#if VERSION == 3
    #define INCFILE  <version3.h>
#elif VERSION == 2
    #define INCFILE  <version2.h>
#else
    #define INCFILE  <version1.h>
#endif
#include INCFILE
```

5.3 Macro Replacement

A preprocessing directive of the form

```
#define identifier replacement-list new-line
```

defines an object-like macro that causes each subsequent instance of the macro name to be replaced by the replacement list of preprocessing tokens that constitute the remainder of the directive. The new-line is a character that terminates the **#define** preprocessing directive.

The identifier immediately following the **#define** is called the *macro name*. The *macro name* is followed by a sequence of tokens called replacement list. Two replacement lists are identical if and only if the preprocessing tokens in both have the same number, ordering, spelling, and white-space separation, where all white-space separations are considered identical.

The simple form of macro is particularly useful for introducing named constants into a program, so that some numbers such as the length of a table may be written in exactly one place and then referred to elsewhere by name. This makes it easier to change the number later. For example, given the following macro

```
#define BLOCK_SIZE 0x100
```

we can write

```
int size = BLOCK_SIZE;
```

instead of

```
int size = 0x100;
```

A preprocessing directive of the form

```
#define identifier( identifier-list-opt ) replacement-list new-line
```

defines a function-like macro with arguments, similar syntactically to a function call. The parameters are specified by the optional list of identifiers, whose scope extends from their declaration in the identifier list until the new-line character that terminates the **#define** preprocessing directive. Each subsequent instance of the function-like macro name followed by an open parenthesis '**(**' as the next preprocessing token introduces the sequence of preprocessing tokens that is replaced by the replacement list in the definition (an invocation of the macro). The replaced sequence of preprocessing tokens is terminated by the matching closing parenthesis '**)**' preprocessing token, skipping intervening matched pairs of left and right parenthesis preprocessing tokens. Within the sequence of preprocessing tokens making up an invocation of a function-like macro, new-line is considered a normal white-space character.

For example, if a macro `mul` with two arguments is defined by

```
#define mul(x,y) ((x)*(y))
```

then the source program line

```
result = mul(5, a+b);
```

is replaced with

```
result = ((5)*(a+b));
```

Note that the parentheses are important in the macro definition. If the macro `mul()` was defined without parentheses as

```
#define mul(x,y) x*y
```

the statement

```
result = mul(5, a+b);
```

would become

```
result = 5*a+b;
```

A variable argument list macro uses the ellipsis notation in the arguments. An identifier **__VA_ARGS__** that occurs in the replacement list is treated as if it were a parameter, and the variable arguments form the preprocessing tokens used to replace it. For example, the code fragment

```
#define debug(...)    printf(__VA_ARGS__)
#define debug2(fp, ...)   fprintf(fp, __VA_ARGS__)
debug("x = %d\n", x);
debug2(stderr, "x = %d\n", x);
```

results in

```
printf("x = %d\n", x);
fprintf(stderr,  "x = %d\n", x);
```

5.4 Converting Tokens to Strings

The # token appearing within a macro definition is recognized as a unary stringization operator. If, in the replacement list, a parameter is immediately preceded by a # preprocessing token, both are replaced by a single character string literal preprocessing token that contains the spelling of the preprocessing token sequence for the corresponding argument. For example,

```
> #define TEST(a) #a
> printf("%s",TEST(abcd))
abcd
>
```

The macro parameter `abcd` has been converted to the string constant "`abcd`".

Each occurrence of white space between the argument's preprocessing tokens becomes a single space character in the character string literal. White space before the first preprocessing token and after the last preprocessing token composing the argument is deleted. Otherwise, the original spelling of each preprocessing token in the argument is retained in the character string literal. The spelling of string literals and character constants: a \ character is inserted before each " and \ character of a character constant or string literal (including the delimiting " characters), is specially handled. For example,

```
> #define TEST(a) #a
> printf("1%s2",TEST(  a    b  ))
1a b2
> printf("1%s2\n", TEST( a\\b ))
1a\b2
> printf("1%s2\n", TEST(" a \\ b "))
1" a \\ b "2
>
```

Here the argument is turned into the string constant "`a b`". The white spaces before `a` and after `b` are deleted, and the sequence of white spaces between `a` and `b` is replaced by a single character.

5.5 Token Merging in Macro Expansions

Merging of tokens to form new tokens in Ch is controlled by the presence of the merging operator ## in macro definitions. For both object-like and function-like macro invocations, before the replacement list is reexamined for more macro names to replace, each instance of a ## preprocessing token in the replacement list (not from an argument) is deleted and the preceding preprocessing token is concatenated with the following preprocessing token. The new token might be the name of a function, variable or type, or a keyword; it might even be the name of another macro, in which case it will be expanded. The common use of concatenation is concatenating two names into a longer name. It is also possible to concatenate two numbers, or a number and a name, such as '`1.5`' and '`e3`', into a number. In addition, multi-character operators such as '+=' can be formed by concatenation. For example,

```
> #define CONC2(a, b) a ## b
> #define CONC3(a, b, c) a ## b ## c
> CONC2(1, 2)
12
> CONC3(3, +, 4)
```

```
7
>
```

The macro `CONC2(1, 2)` concatenates two numbers, `1` and `2`, into `12`, and `CONC2(3, +, 4)` concatenates these three arguments into `3+4`, which generates `7` in Ch command line.

Ch converts comments to white spaces before macros are even considered. Any "`/* comment sequence */`" sequence will be interpreted as a number of blank spaces. The user can use comments next to a "##" in a macro definition, or in actual arguments that will be concatenated because the comments will be initially converted to blank spaces that will later be discarded by the contatenation operation. For example,

```
> #define CONC2(a, b) a ## b
> CONC2(1, /*this is a comment */2)
12
>
```

The comment in the second argument is discarded in concatenation.

A ## preprocessing token shall not occur at the beginning or at the end of a replacement list for either form of macro definition.

5.6 Line Control

The **#line** directive can be used to alter the line numbers assigned to the source code. This directive gives a new line number to the following line, which is then incremented to derive the line number for subsequent lines. The directive can also specify a new file specification for the program source file. This is useful for referring to original source files that are preprocessed into Ch code by other programs.

A preprocessing directive of the form

```
#line digit-sequence new-line
```

causes the implementation to behave as if the following sequence of source lines begins with a source line that has a line number as specified by the digit sequence (interpreted as a decimal integer). The digit sequence shall not specify zero, nor a number greater than 2147483647. The line number is stored in the predefined macro **__LINE__** internally.

A preprocessing directive of the form

```
#line digit-sequence "s-char-sequence-opt" new-line
```

sets the presumed line number similarly and changes the presumed name of the source file to be the contents of the character string literal. The name of the source file is stored in the predefined macro **__FILE__** internally.

For example, the following program with file name `pre_line.c`.

```
int main () {
   printf("before line directive, line number is %d \n", __LINE__);
   printf("the FILE predefined macro = %s\n", __FILE__);
   #line 200 "newname"
   printf("after line directive, line number is %d \n", __LINE__);
   printf("the FILE predefined macro = %s\n", __FILE__);
   return 0;
}
```

will print out

```
before line directive, line number is 2
the FILE predefined macro = pre_line.c
after line directive, line number is 200
the FILE predefined macro = newname
```

5.7 Error Directive

A preprocessing directive of the form

```
#error pp-tokens-opt new-line
```

causes the implementation to produce a diagnostic message that includes the specified sequence of preprocessing tokens and the interpretation to cease.

For example, when the program `pre_err.c` below is executed in Ch,

```
int main () {
#error from preprocessing error directive
   printf("after error directive\n");
   return 0;
}
```

it will print out

```
ERROR: #error:  from preprocessing error directive
ERROR: syntax error before or at line 2 in file pre_err.c
  ==>: #error from preprocessing error directive
  BUG: #error from preprocessing error directive <== ???
WARNING: cannot execute command 'pre_err.c'
```

5.8 NULL Directive

A preprocessing directive of the form

```
#new-line
```

has no effect on the program. The line is ignored.

5.9 Pragma Directive

A preprocessing directive of the form

```
# pragma pp-tokens-opt new-line
```

is called a pragma directive. The C standard defines `#pragma` as a means to implement platform dependent functionality. According to the C standard, if the preprocessing token STDC does not immediately follow pragma in the directive prior to any macro replacement (`#pragma STDC`), implementation-defined features can be added. Ch defines several `#progma` statements to implement special functionality. The

Table 5.2: Valid pragmas.

Pragma name	Value of argument
`exec expr`	Execute an expression when it is parsed.
`remvar(`*arg*`)`	Remove a global or top level variable *arg*.
`remkey(`*arg*`)`	Remove a kwyword *arg*.
`import "`*filename*`"`	Include the file *filename*. It searches for the file in the current directory first. Then, the directories specified by **_path**.
`import <`*filename*`>`	Include the file *filename*. It searches for the file in only directories specified by **_path**.
`importf "`*filename*`"`	Include the file *filename*. It searches for the file in the current directory first. Then, the directories specified by **_fpath**.
`importf <`*filename*`>`	Include the file *filename*. It searches for the file in only directories specified by **_fpath**.
`pack()`	Automatic alignment of structure fields.
`pack(pop)`	Automatic alignment of structure fields.
`pack(push, `*n*`)`	Turn *n* byte packing of structures on.
`pack(`*n*`)`	Turn *n* byte packing of structures on.
`package <`*pname*`>`	Add `$(_ppath)/pname/bin` to **_path**, `$(_ppath)/pname/lib` to **_fpath**, `$(_ppath)/pname/include` to **_ipath**, `$(_ppath)/pname/dl` to **_lpath**.
`package "`*/u/dir/pname*`"`	Add `/u/dir/pname/bin` to **_path**, `/u/dir/pname/lib` to **_fpath**, `/u/dir/pname/include` to **_ipath**, `/u/dir/pname/dl` to **_lpath**.
`package _fpath <`*path*`>`	Add `$(_ppath)/`*path* to **_fpath**.
`package _fpath "`*/u/dir/path*`"`	Add */u/dir/path* to **_fpath**.
`package _ipath <`*path*`>`	Add `$(_ppath)/`*path* to **_ipath**.
`package _ipath "`*/u/dir/path*`"`	Add */u/dir/path* to **_ipath**.
`package _lpath <`*path*`>`	Add `$(_ppath)/`*path* to **_lpath**.
`package _lpath "`*/u/dir/path*`"`	Add */u/dir/path* to **_lpath**.
`package _path <`*path*`>`	Add `$(_ppath)/`*path* to **_path**.
`package _path "`*/u/dir/path*`"`	Add */u/dir/path* to **_path**.
`_fpath <`*path*`>`	Add `CHHOME/toolkit/lib/`*path* to **_fpath**.
`_fpath "`*/u/dir/path*`"`	Add */u/dir/path* to **_fpath**.
`_ipath <`*path*`>`	Add `CHHOME/toolkit/include/`*path* to **_ipath**.
`_ipath "`*/u/dir/path*`"`	Add */u/dir/path* to **_ipath**.
`_lpath "`*/u/dir/path*`"`	Add */u/dir/path* to **_lpath**.
`_path "`*/u/dir/path*`"`	Add */u/dir/path* to **_path**.

preprocessing token names for the pragma directive defined in Ch, conforming to the C standard, are listed in Table 5.2.

In the example below, variables `var1` and var2 are first declared as int. Later, they are removed and re-declared as float in different scopes.

```
int var1;
int var2;
..
#pragma remvar(var1)
float var1;
int main(){
   #pragma remvar(var2)
   float var2;
   ...
}
```

In the example below, keyword **int** is removed from the system and later used as a variable identifier.

```
#pragma remkey(int)
float int;
int = 10;
```

The expression `expr` in the directive `#pragma exec expr` is evaluated when a program is parsed. It may contain generic functions, but not functions located in function files. For example, assume the home directory obtained by the generic function `getenv()` is `/home/myname`, the directive below

```
#pragma exec _fpath=stradd(_fpath, getenv("HOME"), "/chfunc;");
```

adds the directory `/home/myname/chfunc` in the system variable **_fpath** for function files at both parsing and runtime. As another example, the CPU time in the unit of seconds for parsing a header file and code in a block can be obtained calling the generic function **clock()** at the parsing time as follows.

```
#include <time.h>
#pragma exec clock();
#include <headerfiles.h>
/* other code */
#pragma exec printf("CPU: %f\n", (double)clock()/CLOCKS_PER_SEC);
```

The `pack` in the directive specifies packing alignment for structure, union, and class members. Either one of the following statements

```
#pragma pack()
#pragma pack(pop)
```

sets the alignment automatically. Either one of the following statements

```
#pragma pack(push, n)
#pragma pack(n)
```

specifies the value, in bytes, to be used for packing. Valid values are 1, 2, 4, 8, and 16. The alignment of a member will be on a boundary that is either a multiple of *n* or a multiple of the size of the member, whichever is smaller.

Table 5.3: Macros defined in both C and Ch.

Macro name	Description
__LINE__	The line number of the current source program line which is expressed as a decimal integral constant,
__FILE__	The name of the current source file which is expressed as a string constant.
__DATE__	The calendar date of the translation which is expressed as a string constant form "Mmm dd yyyy". Mmm is as produced by **asctime()**.
__TIME__	The current time which is expressed as a string constant of the form "hh:mm:ss", as returned by **asctime()**.
__STDC__	The decimal constant 1.
__STDC_VERSION__	The decimal constant 199901L.

Table 5.4: Macros defined in Ch.

Macro name	Description
_CH_	The decimal constant 1.
CHDLL	The decimal constant 1 if dynamic link libraries are supported. Otherwise, not defined.
GLOBALDEF	The decimal constant 1 when defined macros are in program scope. Undefine it when defined macros in a program, dot files, and function files are unrelated to each other. By default, it is defined as 1.
_M64_	The decimal constant 1, defined only for 64-bit machines.
SCH	The decimal constant 1, when Ch is invoked as safe shell. Otherwise, it is not defined.

5.10 Predefined Macros

The macro names predefined in both C and Ch are listed in Table 5.3, the macro names predefined only in Ch are listed in Table 5.4, and the platform-independent macro are listed in Table 5.5. The platform-independent macros _HPUX_, _LINUX_, _LINUXPPC_, _SOLARIS_, _WIN32_, _DARWIN_, _FREEBSD, _QNX_, _AIX_ are defined primarily for use in start-up and header files. The user shall avoid using them in portable Ch application programs.

Table 5.5: Platform-dependent macros defined in Ch.

Macro name	Description
ppc	The decimal constant 1, when PowerPC in Mac OS X in Darwin is used. Otherwise, it is not defined.
_i386_	The decimal constant 1 for Intel x86 32 bit machine. Otherwise, it is not defined.
_x86_x64_	The decimal constant 1 for Intel x86 64 bit machine. Otherwise, it is not defined.
_BIG_ENDIAN_	The decimal constant 1, when the big endian machine in Mac OS X in Darwin or Sparc in Solaris is used. Otherwise, it is not defined.
LITTLEENDIAN	The decimal constant 1, when the little endian machine in QNX is used. Otherwise, it is not defined.
AIX	The decimal constant 1, when AIX is used. Otherwise, it is not defined.
DARWIN	The decimal constant 1, when Mac OS X in Darwin is used. Otherwise, it is not defined.
FREEBSD	The decimal constant 1, when FreeBSD OS is used. Otherwise, it is not defined.
HPUX	The decimal constant 1, when HP-UX OS is used. Otherwise, it is not defined.
LINUX	The decimal constant 1, when Linux OS is used. Otherwise, it is not defined.
LINUXPPC	The decimal constant 1, when Linux OS for PowerPC is used. Otherwise, it is not defined.
QNX	The decimal constant 1, when QNX OS is used. Otherwise, it is not defined.
SOLARIS	The decimal constant 1, when Solaris OS is used. Otherwise, it is not defined.
WIN32	The decimal constant 1, when Windows OS is used. Otherwise, it is not defined.
_X86_	The decimal constant 1, when Intel x86 processor is used in Windows and Mac OS X x86. Otherw

For some programs in Solaris, the macro __STDC__ may need to be redefined with value 0 by

```
#define __STDC__ 0
```

Chapter 6

Types and Declarations

Ch is a loosely typed language with a rich set of data types. Unlike languages such as Pascal which prohibits automatic type conversion, one data type in Ch can be automatically converted to another data type if it makes sense in context. The meaning of a value stored in an object or returned by a function is determined by the type of the expression used to access it. An identifier declared to be an object is the simplest expression; the type is specified in the declaration of the identifier. Types are partitioned into *object types* that describe objects, *function types* that describe functions, and *incomplete types* that describe objects but lack the information needed to determine their sizes. The format of a value stored in computer memory depends on the machine architecture in use. How identifiers of different types are declared, and their values internally represented in a computer system for manipulation inside Ch, will be illustrated in this chapter. The discussion is based on the architecture of the RISC processor for SUN Sparc workstations.

6.1 Data Types

6.1.1 Integral Data Types

Integer is a basic data type for any computer language. An integer in Ch can be represented in one of below data types

```
char
signed char
unsigned char
short
signed short
unsigned short
int
signed int
unsigned int
long
long int
signed long
signed long int
unsigned long
unsigned long int
long long
long long int
```

```
signed long long
signed long long int
unsigned long long
unsigned long long int
```

Numerical manipulations of char and int data in Ch follow the rules defined in C.

Char Data Representation

The char data are used to store characters such as letters and punctuation. An array of char can be used to store a string. A character is actually stored as an integer according to a certain numerical code such as the ASCII code. Under this code, certain integers represent certain characters. The standard ASCII code ranges from 0 to 127, which need only 7 bits to represent. In Ch, the char variable is a signed integer ranging from CHAR_MIN to CHAR_MAX. The macros CHAR_MIN and CHAR_MAX, defined in the C standard header **limits.h**, are system constants in Ch. Typically, a char constant or variable occupies 1-byte of unit memory. Bit 8 is a sign bit. The maximum positive integer for a signed 1-byte representation is 127 or 01111111 in the binary form. A negative number is stored as the binary complement form. To generate the two's complement of a number, all the binary bits (8 bits for a char) are inverted and the result is incremented by one. For example, the decimal value 2 is represented in binary with 8 bit char integer as 00000010. The decimal value of -2 is represented by the binary value of 11111110 in a 1-byte two's complement form as follows.

$$\begin{aligned}(-2)_{10} &= \text{complement}(00000010)_2 + (1)_2 \\ &= (11111101)_2 + (1)_2 \\ &= (11111110)_2\end{aligned}$$

where the subscripts 2 and 10 indicate the base of the associated number. The minimum integer values for a signed char is -128 or 10000000 in the binary form. The range of integers for a char is thus -128 to +127.

Unsigned Char Data Representation

In Ch, the unsigned char variable is equivalent to an unsigned int ranging from 0 to UCHAR_MAX. The macro UCHAR_MAX defined in the C standard header **limits.h**, is a system constant in Ch. Typically, an unsigned char variable occupies 1-byte unit memory without the sign bit, so that the parameter UCHAR_MAX is 255 or 11111111 in the binary form.

Short Data Representation

The short variable ranges from SHRT_MIN to SHRT_MAX. The macros SHRT_MIN and SHRT_MAX defined in the C standard header **limits.h**, are system constants in Ch. A short data uses 2 bytes (16 bits) for storage with 1 bit for the sign in Ch. Negative numbers are stored in 2-byte two's complement form. Therefore, the parameters SHRT_MIN and SHRT_MAX are $-32768(2^{15})$ and $32767(2^{15} - 1)$, respectively.

Unsigned Short Data Representation

The unsigned short variable ranges from 0 to USHRT_MAX. The macro USHRT_MAX defined in the C standard header **limits.h**, is a system constant in Ch. An unsigned short variable occupies 2-byte unit memory without the sign bit in Ch, so that the parameter USHRT_MAX is $65535(2^{16}-1)$.

Int Data Representation

An int data is a signed integer in Ch. An int number is a whole number that can be negative, positive, or zero. The int ranges from INT_MIN to INT_MAX. The macros INT_MIN and INT_MAX, defined in the C standard header **limits.h**, are precalculated system constants in Ch. Unlike some C implementations, in which an int data may occupy only 2 bytes, an int data uses 4 bytes (32 bits) for storage with 1 bit for the sign in Ch. Negative numbers are stored in 4-byte two's complement form. The values of INT_MIN and INT_MAX then become -2147483648 (2^{31}) and 2147483647, respectively. For example, the following statements are valid in Ch.

```
> char c[2][3], *cptr;
> int i, *iptr;
> c[0][1] = 'a'; // c[0][1] becomes 'a'
a
> i = c[0] [ 1];   // i becomes 97, ASCII number for 'a'
97
> c[1][2] = i+1  // c[1][2] becomes 'b', ASCII number for 'b' is 98
b
> i += c[1][2];  // i becomes 195 = 97 + 98
195
> iptr = &i;     // iptr points to address of i
4005ec50
> *iptr /= 2;    // i becomes 97 = 195/2
97
>
```

Note that arrays in Ch can be declared and accessed by `c[i][j]`. White spaces and tab characters, such as the ones in the statement `i = c[0] [ 1]`, will be ignored in the Ch program, except when they are characters within a string such as `"ab cd"`.

Unsigned Int Data Representation

The unsigned int variable ranges from 0 to UINT_MAX. The macro UINT_MAX defined in the C standard header **limits.h**, is a system constant in Ch. An unsigned int variable occupies 4-byte unit memory without the sign bit in Ch, so that the parameter UINT_MAX is 4294967295 (2^{32} - 1).

Long Data Representation

In Ch, data of long and long int have the same representation as that in C. For example, in 32-bit machine, long is typically the same as int. For 64-bit machines, long is the same as long long in Linux 64-bit and the same as int in Windows 64-bit.

Long Long Data Representation

In Ch, data of long long and unsigned long long integral types contain 64 bits. They have the similar representation as the data types of int and unsigned int. For example,

```
> long long l
> l = 10LL
> printf("l = %lld", l);
```

```
l = 10
> sizeof(l)
8
> scanf("%lld", &l);
11
> printf("l = %lld", l);
l = 11
>
> unsigned long long ul
> ul = 10ULL
> printf("ul = %llu", ul);
ul = 10
>
```

6.1.2 Floating-Point Types

The integer data type serves well for some software development projects, especially for system programming. However, for scientific computing, floating-point numbers are used extensively. Floating-point numbers correspond to real numbers that include the numbers between integers. These numbers are defined in Ch as float or double, which are equivalent to real and double precision in Fortran, respectively. Floating-point numbers are analogous to the representations of numbers in scientific notion. Floating-point arithmetic is complicated, compared with the integer arithmetic.

The most common implementation of floating-point arithmetic is based upon the IEEE 754 standard. In this standard, a float or double is represented in the form of

$$(-1)^{sign} 2^{exponent-bias} 1.f \tag{6.1}$$

where $1.f$ is the significand. The 1 is implicit and f represents the fractional bits of the normalised number. This normalized floating-point number contains a "hidden" bit '1'. Therefore, this representation has one more bit of precision than would otherwise be the case.

Float Data Representation

The float data type uses 32 bits for its storage. The result of a float data is formulated as

$$(-1)^{sign} 2^{exponent-127} 1.f \tag{6.2}$$

Bit 31 is a sign bit; it is 1 for negative numbers. Bits 23 to 30 are the *exponent* bits. The exponent is offset by 127 to allow a range of numbers spanning 1. Cases when all the exponent bits are 0's and all the exponent bits are 1's are reserved for the metanumbers Inf, $-$Inf, NaN shown in Table 6.1. Bits 0 to 22 define the fractional component of the significand. The leading integer of the normalised significand is always 1 so doesn't need to be stored. In binary fractions, the most significant bit represents 0.5, the next bits representing 0.25, 0.125, etc. Table 6.1. shows the hexadecimal representation of some float numbers. For example, according to Equation (6.2), float numbers 1.0 and -2.0 can be obtained by $(-1)^0 2^{127-127} 1.0 = 1.0$ and $(-1)^1 2^{128-127} 1.0 = 2.0$, respectively. Remember that the fraction of the normalized significand is stored in a binary fraction. The float number 3.0 can be calculated by $(-1)^0 2^{128-127} (1.1)_2 = 2 * (1.1)_2 = 2 * (1.5)_{10} = (3.0)_{10}$ where subscripts indicate the base of the floating-point number. Note that the IEEE 754 standard distinguishes $+0.0$ from -0.0 for floating-point numbers.

The macro FLT_MAX, defined as the maximum representable finite floating-point number in the float data type in the C standard header **float.h**, is a precalculated system constant in Ch. If a number is greater

Table 6.1: Hexadecimal representations of selected real numbers

value	float	double
0.0	00000000	0000000000000000
−0.0	80000000	8000000000000000
1.0	3F800000	3FF0000000000000
−1.0	BF800000	BFF0000000000000
2.0	40000000	4000000000000000
−2.0	C0000000	C000000000000000
3.0	40400000	4080000000000000
−3.0	C0400000	C080000000000000
Inf	7F800000	7FF0000000000000
−Inf	FF800000	FFF0000000000000
NaN	7FFFFFFF	7FFFFFFFFFFFFFFF
FLT_MAX	7F7FFFFF	
DBL_MAX		7FEFFFFFFFFFFFFF
FLT_MIN	007FFFFF	
DBL_MIN		000FFFFFFFFFFFFF
FLT_MINIMUM	00000001	
DBL_MINIMUM		0000000000000001

than FLT_MAX, it is called an *overflow*. Any number greater than FLT_MAX has all 8 exponent bits set to 1's. This shall be represented by the metanumber Inf, which corresponds to the mathematical infinity symbol ∞. This is the result of many operations such as division of a finite number by zero although an inexact exception may be raised in an IEEE machine. Any number less than −FLT_MAX shall be represented by the metanumber −Inf which corresponds to the mathematical negative infinity symbol $-\infty$.

The value of the parameter FLT_MIN is defined in the C standard library header **float.h** as a minimum normalized positive floating-point float number. If a number is less than FLT_MIN, it is called an *underflow*. The IEEE 754 standard provides a *gradual underflow*. When a number is too small for a normalized representation, leading zeros are placed in the significand to produce a denormalized representation. *A denormalized number* is a nonzero number that is not normalized and whose exponent is the minimum exponent for the storage type. The maximum representable positive denormalized float is defined as FLT_MINIMUM in Ch as shown in Table 6.1. There is only one unit in the last place for FLT_MINIMUM so that it is commonly referred to as *ulp*. Almost all floating-point implementations substitute the value zero for a value that is smaller than FLT_MINIMUM for IEEE machines, and FLT_MIN for non-IEEE machines. However, in the arithmetic operations and mathematical functions defined in Ch, there is a qualitative difference between FLT_MINIMUM which is smaller than FLT_MIN and zero. In this documentation, by the value of 0.0 means that it is a zero, not a small number. The Ch expressions of 0., 0.00, and .0 are the same as 0.0. In the same token, the following Ch floating-point constant expressions -0.0, $-0.$, -0.00, and $-.0$ are equivalent. Mathematically, divisions of zero by zero of $0.0/0.0$ and infinity by infinity of ∞/∞ are indeterminate. The results of these operations are represented by the symbol of NaN, which stands for Not-a-Number. It should be mentioned that the IEEE 754 standard distinguishes *quiet* NaN from *signaling* NaN. The signaling NaN should generate a signal or raise an exception. In Ch, all NaNs are treated as quiet NaNs. Furthermore, the IEEE 754 standard does not interpret the sign of NaN. No −NaN will be produced as a result of arithmetic and functions in Ch although it can be created by manipulating the bit pattern of the memory location of a float variable. The expression −NaN is interpreted as NaN in Ch. The metanumbers are treated just as

regular floating-point numbers. The internal hexadecimal representations of the metanumbers for the float type are also given in Table 6.1.

Double Data Representation

For a large range of representable floating-point numbers, a double data can be used in Ch. The double data type uses 64 bits as its storage. The result of the double data is formulated as

$$(-1)^{sign}2^{exponent-1023}1.f \tag{6.3}$$

Bit 63 is a sign bit; it is 1 if the number is negative. Eleven-bit exponent of bits 52 to 62 are biased by 1023; values of all zeros and all ones are reserved for metanumbers. Bits 0 to 51 are fractional components of normalized significand. Like float, the integral value 1 of the normalized significand is hidden. The hexadecimal representation of some typical double numbers are also given in Table 6.1. Note that the width and bias value of the exponent of double is different from those of float. Therefore, a float cannot be converted into a double just by padding zeros in its fraction. On the other hand, when a double data is cast into a float, the result cannot be obtained just by ignoring the values in bits 0 to 31. Note that there is no external distinction between float Inf and double Inf, although their internal representations differ. This is also true for metanumbers −Inf and NaN. Similar to float, parameters DBL_MAX, DBL_MIN, and DBL_MINIMUM are system constants in Ch. The internal memory representations of these special finite double floating-point numbers are also given in Table 6.1. Note that due to the finite precision of the floating-point number representation, the exact values of irrational numbers such as π are not representable in a computer system whether they are represented in float or double.

6.1.3 Aggregate Floating-Point Types

The complex number, an extension of real number, has wide applications in science and engineering. The variables of complex type can be declared by two type specifiers, `complex` and `double complex`. After the declaration, the complex number can be created in Ch by the complex constructor **complex**(*x*, *y*), where *x* is its real part and *y* is its imaginary part. For example,

```
> complex z1;          // a double complex variable
> double complex z;    // a double complex variable
> float complex z2;    // a float complex variable
> z1 = complex(1, 2);  // z1 becomes 1 + i2
complex(1.00,2.00)
>
```

One can declare not only a simple complex variable, but also a pointer to complex, array of complex, and array of pointer to complex, etc. Declarations of these complex variables are similar to the declarations of other data types. For example,

```
> complex *zptr1;
> complex z2[2], z3[2][3]; // declared as pointer to complex variable
> complex *zptr[2][4];     // array of pointer to complex
> zptr1 = &z1;             // zptr1 point to the address of z1
4005e748
> *zptr1 = complex(2, 3);  // z1 becomes 2 + i3
complex(2.00,3.00)
> z1
```

```
complex(2.00,3.00)
>
```

Chapter 13 describes details of complex numbers, including input/output operations, data conversion rules, functions, etc.

6.1.4 Pointer Data Types

Pointer is defined as a variable which contains the address of another variable or dynamically allocated memory. Ch uses pointers explicitly for arrays, structures, functions, classes and simple data types. With operator '`*`' in front of the variable names, the variables of pointer type can be declared similar to variables of other data types. The unary operator '`&`' gives the "address of a variable". For example, the code below

```
int i, *p1, **p2;
p1 = &i;
p2 = &p1;
```

declares two pointers `p1` and `p2`. `p1` stores the address of the integer `i`, and `p2` stores the address of `p1`. More information about pointers can be found in Chapter 9. In addition to pointer to simple data types, pointer to arrays and functions are also available in Ch. More information about pointer to arrays and functions can be found in Chapter 14 and section 10.8, respectively.

6.1.5 Array Types

The number of dimensions in an array is called the *rank* of the array. The number of elements in a dimension is called the *extent* of the array in that dimension. The *shape* of an array is a vector where each element of the vector is the extent in the corresponding dimension.

Computational array in Ch is a first-class object. The type qualifier `array` for computational arrays, available in Ch Professional and Student Editions, is defined as a macro in the header file **array.h**. The declaration of an array is shown below.

```
#include <array.h>
int a1[3][4];       // array of integer
int *a2[3][4];      // array of pointer
array int a3[3][4]; // computational array
```

where `a1` is declared as an array of integer, `a2` is an array of pointer to integer, and `a3` is a computational array. Type qualifier **array** in the declaration will make `a3` a computational array, which can be treated as a first-class object for linear algebra and matrix computation. Arrays of variable length including deferred-shape arrays, assumed-shape arrays, pointer to assumed-shape arrays, and arrays of reference are supported. The following example will clarify the concepts of these various array definitions.

```
void funct(int a[:][:], (*b)[:], c[], d[&], n, m){
/* a: assumed-shape array */
/* b: pointer to array of assumed-shape */
/* c: incomplete array completed by function call */
/* d: array of reference */
/* n, m: integers */
  int e[4][5];     // fixed-length array
  int f[n][m];     // deferred-shape array
```

```
  int (*g)[:];     // pointer to array of assumed-shape
  extern int h[];  // incomplete array completed by external linkage
  int i[] = {1,2}; // incomplete array completed by initialization
  f[1][2] = a[2][3];
}
int A[3][4], B[5][6], C[3], D[4];
funct(A, B, C, D, 10, 20);
funct(B, A, C, D, 85, 85);
```

The argument a is declared as an assumed-shape array to which the arrays with different extents can be passed, the argument b is declared as a pointer to array of assumed-shape, c is declared as an incomplete array which will be completed by the function call, d is an array of reference which can handle arrays of different data types.

In the for-loop below, when the array a with different sizes is redeclared, its memory will be reallocated by function **realloc()** internally.

```
int i;
for (i = 0; i<10; i++) {
  int a[i];
  ...
}
```

The range of subscripts for an index of an array can be adjusted. For example,

```
int a[1:10], b[-5:5], c[0:10][1:10], d[10][1:10];
int e[n:m], f[n1:m1][1:m2];
extern int a[1:], b[-5:], c[0:][1:10];
int funct(int a[1:], int b[1:10], int c[1:][3], int d[1:10][0:20]);
a[10] = a[1]+2; /* OK */
a[0] = 90;      /* Error: index out of range */
```

where the subscript of a ranges from 1 to 10; b from -5 to 5; the first dimension of c ranges from 0 to 10, the second from 1 to 10; the first dimension of d from 0 to 9 and the second from 1 to 10.

Arrays of different shape and data type can be passed to arrays of reference. For example,

```
float a[3][4];
double b[5][6];
void func(double a[&][&]);
func(a);
func(b);
```

where the argument a of the function func is declared as an array of reference, so that arrays with different extents and data types can be passed to it.

Members of struct/union can be pointers to assumed-shape arrays. For example,

```
int a[4][5], b[7][8];
struct tag_t {
  int n;
  int (*A)[:];
} s;
s.A = a; /* s.A[i][j] == a[i][j] */
...
s.A = b; /* s.A[i][j] == b[i][j] */
```

where the member A of struct s is declared as a pointer to assumed-shape arrays to which the arrays with different extents can be assigned.

More information about the relation between pointer and array can be found in Chapter 14, and information about computational arrays can be found in section 6.2.1 and Chapter 16.

6.1.6 Structure Types

The structure types in Ch are similar to those in C++. They are collections of members that can have different types. For example,

```
struct tag_t {
  data_type1 field1;
  data_type2 field2;
} name1;
tag_t name2, *name3;
struct tag_t name4;
name3 = &name2;
```

where the struct with the tag name of tag_t has two members, field1 and field2. Three objects of struct tag_t, name1, name2 and name4, are declared by three different ways. name1 is declared directly after the definition of the struct, name2 is declared only by the tag name, while the name4 is declared with the optional keyword struct. The variable name3 is declared as a pointer to struct, and is assigned the address of name2.

There are two namespaces for struct in C, one for struct tags and one for member variables. But there are one and a half namespaces for struct in C++, one for struct tags and an half for member variables. Struct in Ch are handled the same as those in C++, and tag is implicitly treated as a typedefed name in Ch. Struct tags and struct variables in Ch share the same namespace. Once a tag name is used as a variable explicitly, this implication will be disabled. For example,

```
struct tag1_t{
    struct tag2_t {
    ....
    };
  ...
};
tag1_t s;           /* OK */
int tag1_t;         /* OK, tag name is used */
struct tag1_t s2;   /* OK */
tag1_t s3;          /* Not valid in Ch and C++ */
```

Like C++, members of a structure in Ch can be functions. More information about structures can be found in Chapter 18.

6.1.7 Class Types

Classes in Ch or C++ are a natural evolution of the structure. Like C++, both class and struct in Ch can have members of functions. By default, members of a class are private whereas members of a struct are public.

The following is an example of the definition of class.

```
class Student {
    int id;
    char *name;
  public:
   void setName(char *n);
}

void Student::setName(char *n) {
  ...
}
```

The class `Student` has three members, two private members `id` and `name`, and a public member function `setName()`. Assume `id` holds the ID number of a student, `name` is the name of the student and the function `setName()` is used to set a student name. After defining a class, it can be used in a program shown below.

```
int main() {
    class Student s1;
    s1.setName("Bob");
    ...
}
```

where `s1` is called an *object* or an *instance* of class `Student`. More information about class types can be found in Chapter 19.

6.1.8 Bit Field

Like C, Ch offers the bit-field which has the capability of defining and accessing within a word directly. In the following code fragment,

```
struct tag{
  data_type1 a:4;
  data_type2 b:4;
} name1 {1,1};
struct tag name2 = {1,1};
name2.a = 2;
```

two members of tag, `a` and `b`, only take 8 bits of memory, 4 bits for each. More information about bit field can be found in Chapter 18.

6.1.9 Union Types

A union type describes an overlapping non-empty set of member objects. a union can only hold one of its members at a time. The members are conceptually overlaid in the same memory. Each member of a union is located at the beginning of the union. For example, the code below shows how a union type can be defined.

```
union tag{
  data_type1 fields1;
  data_type2 fields2;
} name1;
tag name2, *name3;
```

```
union tag name4;
name3 = &name2;
```

The members `fields1` and `fields2` share the same memory. Only one member can be used at a time. Like C++, tag is put into typedefed namespace by default. More information about union types can be found in Chapter 18.

6.1.10 Enum Types

An enumerated type is a set of integer values represented by enumeration constants. For example, the code below

```
enum tag_t{bad, good=1, ugly} x;
enum tag_t y;
x = good;
y = x;
```

defines a new enumerated type indicated by the tag name `tag_t`. The variables of `tag_t`, such as `x` and `y`, can be assigned three enumeration constants `bad`, `good` and `ugly`. More information about enum types can be found in Chapter 18.

6.1.11 Void Type

The void type is used mainly for pointers to void, void argument lists and void return values of functions.

A pointer can be pointed to the type of void. Any pointer of other types may be assigned to and from pointers to void, and may be compared with them. Furthermore, the pointer to any object can be converted to type of void without loss of information. But, in order to access the object pointed to by the original pointer properly, the converted pointer has to be converted back to the original pointer type.

The keyword void, which appears in front of the function name when it is defined, indicates that the function has no return value. For example, the function `funct1()` defined below has no return value.

```
void funct1(int i) {int i; ...; };      /* no return value */
```

The keyword void which appears in the argument list of a function indicates that the function has no argument. For example, the function `funct2()` defined below has no argument.

```
int funct2(void){int i; ...; return i;} /* no argument */
```

6.1.12 Reference Type

In Ch, a reference declared with symbol '&' is an alternative name for an object just as in C++. They have the same syntax. For example, the declarations shown below

```
> int i
> int &j = i
> i = 5
5
> j
5
>
```

indicate that variable `j` is a reference of `i`. They share the same memory space inside the system and, therefore, can be used interchangeably. Any change to the value of `i` will affect the value of `j`. Not only reference for simple data types, including char, short, int, float, double, as well as data types qualified by signed, unsigned, long, and complex can be declared in Ch, but also reference for pointer type can be declared.

Although arguments are passed by the way of *call-by-value* to functions in Ch by default, they can be passed by *call-by-reference* by using the symbol '`&`'. For example, in the prototype of function `swap` shown below

```
void swap(int &n, int &m); /* the same as in C++ */
```

arguments `n` and `m` are declared as references to int. This means that any change to `n` and `m` inside the called function `swap()` will affect their original values in the calling function. More information about reference types can be found in Chapter 11.

6.1.13 String Type

String is a first class object in Ch with type specifier `string_t`. An argument of pointer to char in a function can be replaced by an argument of string. String or array of chars, instead of pointer to char, should be used for safe network computing. The memory allocation and deallocation variables of string type are handled by Ch automatically.

```
string_t s1, s2, s, a[3];
s1 = "Hello, ";
s2 = "world!";
s = s2;
int i = strlen(s1);
strcat(s1,s2);   /* s1 becomes "Hello, world!" */
strcpy(a[0],s1); /* a[0] becomes "Hello, world!" */
```

String of reference is supported in Ch as shown in the program below.

```
string_t stringcat(string_t &s1, s2)
{
  string_t s;
  s = strcat(s1, s2);
  /* s = stradd(s1, s2); */
  s1 = s;
  return s1;
}
string_t s1 = "string1", s2 = "string2";
stringcat(s1, s2);
printf("%s\n", s1); // print out string1 and string2
```

Function `stringcat()` adds string `s2` to the end of string `s1`. It is equivalent to the standard function **strcat()** in C.

Pointer to string can also be declared as shown in the code below.

```
string_t stringcat2(string_t *s1, s2) {
  *s1 = strcat(*s1, s2);
  return *s1;
```

```
}
string_t s1 = "string1", s2 = "string2";
stringcat2(&s1, s2);
printf("%s\n", s1); // print out string1 and string2
```

Function `stringcat2()` is similar to function `stringcat()`.

The string type can be used to obtain a string with space through the input function **scanf()** as shown below.

```
> string_t str;
> scanf("%s", &str);
abcd 1234
> str
abcd 1234
```

The relational operators `==, !=, <,` and `>` can be used for comparison of two strings with one of two operands is the built-in string type **string_t**, the other operand can be **string_t**, string literal "someting", pointer to char, pointer to unsigned char. If the contents of the two strings are the same, the operation `==` gives 1 and the operation `!=` gives 0. Otherwise, the operation `==` gives 0 and the operation `!=` gives 1.

The results of `>` and `<` are similar to 1 and -1 from the function **strcmp()**. The operations `>` and `<` compares two strings byte-by-byte, according to the ordering of your machine's character set. The operation `s1>s2` gives 1 if the string s1 is greater than the string s2 between the values of the first pair of bytes that differ in the strings being compared. Otherwise, it gives 0. The operation `s1<s2` gives 1 if the string s1 is less than the string s2 between the values of the first pair of bytes that differ in the strings being compared. Otherwise, it gives 0.

Symbolic computing operations using operands of string are shown in Figure 6.2.

Table 6.2: Symbolic computing operations using string.

Definition	Ch Syntax
addition	s1 + s2
subtraction	s1 − s2
multiplication	s1 ∗ s2
division	s1 / s2

Symbolic computing is demonstrated as follows.

```
int i = 2;
float x = 10, y;
string_t a="sin(x)", b="x", s;
s = i*a/b+1;
s = s+s
printf("s = %s\n", s); /* output is 2*(2*sin(x)/x+1) */
y = streval(s);        /* y = 1.782 = 2*(2*sin(10)/10+1) */
printf("y = %f\n", y); /* output is y = 1.782 */
```

The symbolic computing capability in Ch is still very preliminary at this point of implementation. Note that the commutative property for the addition of two strings is valid in Ch, i.e., s1+s2 equals s2+s1. More information about string type can be found in Chapter 17.

6.1.14 Function Types

Regular functions in Ch follow the C standard. In the spirit of C, the function definition with nested functions in Ch takes the following forms

```
return_type function_name(argument declaration)
{
  statements
  function_definitions
}
```

or

```
return_type function_name(argument declaration)
{
  function_definitions
  statements
}
```

where statements can be any valid Ch statements and local functions can be defined inside other local functions. There is no restriction on the number of function nesting in Ch. For example,

```
int func1() {
   int func2() {
     int func3() { ...}
   }
   /* ...  */
   func2();
}
```

The definition of a local function can be placed anywhere inside a function. If a local function is invoked prior to its definition, a local function prototype shall be used as shown in Program 6.1.

```
void funct1()
{
  __declspec(local) float funct2(); // local function prototype
  funct2();
  float funct2()        // definition of the local function,
  {
    return 9;
  }
}
```

Program 6.1: `Declaration __declspec(local)` qualifies `funct2()` as a local function.

In Program 6.1, because the function `funct2()` is used before it is defined, a function prototype is needed. Since it is a local function, the type qualifier `__declspec(local)` is used to distinguish a local function from the top level regular C functions.

In a function definition, parameters in the argument list can be ignored if they are not used inside functions. For example,

Table 6.3: Type qualifiers.

Qualifier	Function
array	computational array
const	(ignored for now, will be fixed later)
inline	(ignored)
operator	(reserved for possible operator overloading)
restrict	restricted function, ignored if appears inside argument lists
virtual	(ignored for now, reserved for virtual function in C++)
volatile	(ignored)

```
int func(int i, int /* not_used */, int /* no_used */) {
   return i*i;
}
func(10, 20, 30);
```

6.2 Type Qualifiers

Type qualifiers in Ch are listed in Table 6.3. The type qualifiers `array` and `restrict` are used for computational arrays and restricted functions, respectively.

6.2.1 Computational Arrays

An array qualified by type qualifier `array` is called a *computational array*, available in Ch Professional and Student Editions. This type qualifier is defined as a macro in the header file **array.h**. A computational array is treated as a first-class object. For example,

```
array float a[10][10], b[10][10];
a += b+inverse(a)*transpose(a);
```

for $a = a + b + a^{-1} * a^T$. Computational arrays can be arguments of a function. A regular complete C array can be passed to an argument of a computational array, and vise versa. More information about computational arrays can be found in Chapter 16.

6.2.2 Restricted Function

In Ch, if type qualifier `restrict` appears in a function definition or before the declaration for the return type, the function is treated as a restricted function. For the sake of security, restricted functions cannot be called by Safe Ch programs. For example, the user can declare a restricted function `restricted_function()` as follows to prevent it from execution in Safe Ch.

```
restrict int restricted_function(int i);
```

Some functions, such as **fopen()**, in the C Standard library are defined as restricted functions in Ch. If the type qualifier `restrict` appears in the argument lists of functions, it is ignored. More information about Safe Ch can be found in Chapter 21.

6.3 Constants

In this section, we will describe the external representations of data types discussed in the previous section. Besides declared variables and system defined parameters, different data types in Ch can have their corresponding constants at the programmer's disposal. The constants in Ch include four different kinds: characters, strings, integers, and floating-point numbers.

6.3.1 Character Constants

A character constant, stored as an integer, can be written as one character within a pair of single quotes like `'x'`. A character constant can be assigned to the variable of type `char`. For example,

```
> char c = 'x'
> c
x
>
```

Character constants containing more than a single character or escape character are called *multibyte characters*. Ch also allows the wide character constant which is preceded by the letter `L`. The apostrophe, backslash, and some characters that might not be easily readable in the source program, such as newline characters shall be included in character constants by using escape characters described later. More information about characters can be found in Chapter 17.

Wide Characters and Multibyte Characters

Ch can handle extended character sets which include locale-specific characters. These characters are always too large to be represented within a single object of type char whose size is a byte. To accommodate these characters, Ch supports both wide characters and multibyte characters. The "wide character" is an internal representation scheme which makes an extended character code fit in an object of the integral type **wchar_t**, which is defined in the header file **stddef.h**. Strings of extended characters can be represented as objects of type wchar_t[] or pointers of type wchar_t *. For example, the code below declares a wide character `wc` in Ch.

```
wchar_t wc = L'a';
```

The `L` before the character `a` indicates that character `a` is a wide character. On the other hand, "multibyte character", which contains more than a single character or escape, is the external representation scheme supported by Ch. A multibyte character is a sequence of normal characters which correspond to a wide character. The maximum number of bytes used in representing a multibyte character in the current locale is indicated by macro **MB_CUR_MAX** defined in header file **stddef.h**. A wide character string can be represented externally by a multibyte character string. Multibyte characters may appear in comments, string, and character constants.

A multibyte character set may have a *state-dependent* encoding, wherein each sequence of multibyte characters begins in an *initial shift state* and enters other locale-specific *shift states* when specific multibyte characters are encountered in the sequence. While in the initial shift state, all single-byte characters retain their usual interpretation and do not alter the shift state. The interpretation for subsequent bytes in the sequence is a function of the current shift state.

Ch also supports the facilities defined in C to implement conversion between multibyte character and wide character. For example, the function **mbtowc()** declared in the file **stdlib.h** converts a multibyte character to a wide character, and the function **wctomb()** does it contrarily.

Escape Characters

Some special characters, and particular behaviors of the output device are impossible to be typed in a source program directly. Ch supports *escape characters*, which are *escape codes* beginning with the back slash character '\', to represent these characters and behaviors. Escape codes could be *character escape code* which are characters listed in Table 6.4, and *numeric escape code* which are up to three octal digits or any number of hexadecimal digits.

Typically the character escape code `\a` produces a beep from the speaker as the alert. The *active position* is the location on a display device where the next character output by the function **fputc()** or **fputwc()** would appear. The intent of writing a printing character (as defined by the **isprint()** or **iswprint()** function) to a display device is to display a graphic representation of that character at the active position and then advance the active position to the next position on the current line. The code `\b` moves the active position to the previous position on the current line. The code `\f` represents a form feed which moves the active position to the initial position at the start of the next logical page. The code `\n` is the most commonly used escape code which moves the active position to the initial position of the next line, whereas `\r` moves the active position to the initial position of the current line. The codes `\t` and `\v` move the active position to the next horizontal tabulation position and the next vertical tabulation position, respectively. The code `\\` represents a backslash which is not the preceding character of an escape code. The single quote appearing in a character constant might be mistaken as the ending apostrophe of the character constant. If this is the case, the code `\'` can be used to represent a single quote in a character constant. Similarly, the code `\"` can represent a double quote in a string constant, which is described in section 6.3.2. The code `\?` can be used to produce a question mark in the circumstances in which it might be mistaken as part of a trigraph described in section 2.1.1. The codes shown below shows how the character escapes can be used.

```
> printf("abcdefd");
abcdefd
> printf("abcd\befd");  // backspace
abcefd
> printf("abcd\tefd");  // horizontal tab
abcd    efd
> printf("abcd\"efd");  // double quote
abcd"efd
> printf("%c", '\'');   // single quote
'
> printf("??!")         // trigraph
|
> printf("?\?!")        // question mark
??!
>
```

The numeric escape codes come in two varieties, octal escape codes and hexadecimal escape codes. An octal escape code consists of up to 3 octal digits following the backslash character `\`. For example, under the ASCII encodings, the character `'a'` may be written as `'\141'`; the null character, used to terminate strings, can be written as `'\0'`. A hexadecimal escape code consists of any number of hexadecimal digits following characters `'\x'`. For example, the character `'a'` can be written in hexadecimal escape code as `'\x61'`.

Each of these escape sequences produces a unique value which can be stored in a single **char** object. An octal escape code terminates when the first character that is not an octal digit is encountered or when three octal digits have been used. Therefore, the string `"\1111"` represents two characters, `'\111'` and

Table 6.4: Character escape code.

Escape Code	Translation
a	(alert) Produces an audible or visible alert. The active position shall not be changed.
b	(backspace) Moves the active position to the previous position on the current line. If the active position is at the initial position of a line, the behavior is unspecified.
f	(form feed) Moves the active position to the initial position at the start of the next logical page.
n	(new line) Moves the active position to the initial position of the next line.
r	(carriage return) Moves the active position to the initial position of the current line.
t	(horizontal tab) Moves the active position to the next horizontal tabulation position on the current line. If the active position is at or past the last defined horizontal tabulation position, the behavior is unspecified.
v	(vertical tab) Moves the active position to the initial position of the next vertical tabulation position. If the active position is at or past the last defined vertical tabulation position the behavior is unspecified.
\\	(backslash) Produces a backslash character \, the active position is moved to the next.
\'	(single quote) Produces a single quote character ', the active position is moved to the next.
\"	(double quote) Produces a double quote character ", the active position is moved to the next.
\?	(question mark) Produces a question mark character ?, the active position is moved to the next.

`'1'`, and the string `"\182"` represents three characters, `'\1'`, `'8'` and `'2'`. Since a hexadecimal escape sequences can be of any length and terminated only by a non-hexadecimal character, to stop a hexadecimal escape in a string, break the string into pieces. For example, the codes `'\x61'` and `'a'` are two characters; however, the hexadecimal escape code `'\x61a'` contains only one character, rather than two characters of `'a'`.

6.3.2 String Literals

A character string literal is a sequence of zero or more multibyte characters enclosed in double-quotes, as in `"xyz"`. The same considerations apply to each element of the sequence in a character string literal or a wide string literal as if it were in an integer character constant or a wide character constant, except that the single-quote **'** is representable either by itself or by the escape sequence **'**, but the double-quote **"** shall be represented by the escape sequence **"**.

The multibyte character sequences specified by any sequence of adjacent character and wide string literal tokens are concatenated into a single multibyte character sequence. If any of the tokens are wide string literal tokens, the resulting multibyte character sequence is treated as a wide string literal; otherwise, it is treated as a character string literal.

A byte, or code of value zero, is appended to each multibyte character sequence that results from a string literal or literals. The multibyte character sequence is then used to initialize an array of static storage duration and length just sufficient to contain the sequence. For character string literals, the array elements have type **char**, and are initialized with the individual bytes of the multibyte character sequence. These arrays of static storage duration are distinct. For example, the pair of adjacent character string literals

```
"A" "3"
```

produces a single character string literal containing the two characters whose values are `'A'` and `'3'` . More information about strings can be found in Chapter 17.

Wide Strings

A wide string literal is a sequence of zero or more multibyte characters enclosed in double-quotes and prefixed by the letter `L`, such as `L"xyz"`.

For wide string literals, the array elements have type **wchar_t**, and are initialized with the sequence of wide characters. A wide character string can be represented externally by a multibyte character string. Multibyte characters may appear in comments, string and character constants. Like in strings of normal characters, the single null character, '`\0`', acts as a terminator in strings of multibyte characters. A byte with all bits zero shall be interpreted as a null character, it does not occur in the second or subsequent bytes of a multibyte character. The function **mbstowcs()** converts a multibyte string to a wide-character string, and the function **wcstombs()** does it contrarily. More information about wide strings can be found in Chapter 17.

6.3.3 Integer Constants

A decimal integer constant like 12345 is an int. An integer can also be specified in binary, octal or hexadecimal instead of decimal. A leading 0 (zero) on an integer constant indicates an octal integer whereas a leading 0x or 0X means hexadecimal. Ch and C99 also support binary constants with leading 0b or 0B. For example, decimal 30 can be written as 036 in octal, 0X1e or 0x1E in hexadecimal, and 0b11110 or 0B11110 in binary. Note that expressions like 029 and 0b211 are illegal, which can be detected by Ch.

The value of 0 in Ch means that it is an integer zero. Unlike real numbers, there is no 0_- in int. Therefore, the integer value of -0 equals 0 in Ch. The domain [$-$FLT_MAX, FLT_MAX] of real numbers is larger

than the domain [−INT_MIN, INT_MAX] of integer numbers. When a real number smaller than INT_MIN, including −Inf, is converted to an integer, the result is INT_MIN. For a real number larger than INT_MAX, including Inf, the converted integral value is INT_MAX. When NaN is assigned to an integral variable, the system will print a warning message, and the resultant integral value becomes INT_MAX whose memory map is the same as that of NaN.

In addition to decimal, octal, hexadecimal integral constants, binary integral constants and binary format specifier for I/O are supported. A binary constant is started with the prefix `0b` or `0B`. The format specifier is `%b`. For example,

```
/* Bit-map using binary constants */
#include<stdio.h>
int H[] = {
    0b00000000000000000000000000000000,
    0b00000000000000000000000000000000,
    0b00011111110000000000011111100000,
    0b00000110000000000000000110000000,
    0b00000110000000000000000110000000,
    0b00000110000000000000000110000000,
    0b00000110000000000000000110000000,
    0b00000110000000000000000110000000,
    0b00000110000000000000000110000000,
    0b00000110000000000000000110000000,
    0b00000111111111111111111110000000,
    0b00000111111111111111111110000000,
    0b00000110000000000000000110000000,
    0b00000110000000000000000110000000,
    0b00000110000000000000000110000000,
    0b00000110000000000000000110000000,
    0b00000110000000000000000110000000,
    0b00000110000000000000000110000000,
    0b00011111110000000000011111100000,
    0b00000000000000000000000000000000,
    0b00000000000000000000000000000000
}
int main() {
  int i, size;
  int I=0b00000110000000000000000110000000;

  size = sizeof(H)/sizeof(int);

  for (i=0; i<size; i++) {
    printf("H[%2d] = 0X%8x\n", i, H[i]);
  }
  /* H becomes II */
  H[10] = I;
  H[11] = I;
  for (i=0; i<22; i++) {
```

```
    printf("%32b\n", H[i]);
  }
  return 0;
}
```

6.3.4 Floating-Point Constants

Constants of Real Numbers

In K&R C, all floats in expressions are converted into doubles before evaluation. As a result, any operations involving floating-point operands, even with two float operands, will produce a double result. This is not applicable to many scientific computations in which speed and memory of a program are critical. Because of the indiscriminate conversion rules in the early design of C, every floating point constant like 3.5 and 3e7 is taken as double. This default double mode for floating-point constants has been carried over to the C standard and supported in Ch. All floating-point constants such as $2.4, 2e+3, -2.E-3$, and $+2.1e3$ are double constants by default. However, C has provided a mechanism to specify a float constant. The suffix F or f indicates a float constant, D or d for double. For example, constants $3.4e3F, 3E-3f$, and $3e+3F$ are floats whereas constants $3.4e3D, 3E-3d$, and $3e+3D$ are doubles. However, the constant metanumbers $\pm$Inf, and NaN are always taken as floats unless they are values of double variables. These features are supported in Ch as well. According to this design, the range of representable floating-point numbers can be expanded automatically. For example, the values of FLT_MAX and DBL_MAX for SUN SPARCStations are $3.4e38$ and $1.8e308$, respectively. The following Ch program

```
printf("pow(10.0F, 39) < Inf is %d \n", pow(10.0F, 39) < Inf);
printf("pow(10.0, 39) < Inf is %d \n", pow(10.0, 39) < Inf);
```

will print out
pow(10.0F, 39) < Inf is 0
pow(10.0, 39) < Inf is 1
In the first statement of the program, the value of 10^{39} calculated by **pow**(10.0F, 39) has overflowed as Inf because it is larger than FLT_MAX. The value of 10^{39} calculated by **pow**(10.0, 39) in double data is still within the representable range of $-\text{DBL_MAX} < \mathbf{pow}(10.0, 39) < \text{DBL_MAX}$. In the second case, the metanumber Inf is expanded as a double infinity larger than DBL_MAX.

Hexadecimal Floating-Point Constants

The hexadecimal floating-point constants in C99 are supported in Ch. For example,

```
> 0X2P3
16.0000
> 0x1.1p0
1.0625
> 0x1.1p1F
2.12
```

Constants of Complex Numbers

A complex constant can be formed by the complex number constructor **complex**(x, y), where x and y are real and imaginary parts of the complex number, respectively. If both arguments of function **complex()** are float or integer type, the resulting complex number is of float complex. If one or two of arguments is double type, the resulting complex number is a double complex. For example

```
complex z = complex(1, 3);          // complex(1, 3) is float complex
double complex z = complex(1.0, 3);// complex(1.0, 3) is
                                    // double a complex
```

In addition, complex metanumbers `ComplexInf` and `ComplexNaN` corresponding to the complex infinity and the complex Not-a-Number are available in Ch.

Constants of Pointers

The constants 0 and `NULL` can be assigned to a pointer. In Ch, the constant `NULL` is a built-in dual purpose symbol which can be assigned to variables of both integer and pointer types. It is used in place of zero. For example, the code below

```
> int i, *p
> p = &i
4005e758
> p = NULL
00000000
>
```

assign the address of the integer `i` to the pointer `p` first, and then assign the constant `NULL` to it.

6.4 Initialization

The declaration of a variable may be accompanied by an initializer that specifies the value of the variable should have at the beginning of its lifetime. All rules for initialization in C can be applied to Ch, except that arrays of more than three dimensions cannot be initialized in Ch.

If an object that has either automatic or static storage duration, which is not initialized explicitly, then:

- if it has pointer type, it is initialized to a null pointer;
- if it has arithmetic type, it is initialized to (positive or unsigned) zero;
- if it is an aggregate, every member is initialized (recursively) according to these rules;
- if it is a union, the first named member is initialized (recursively) according to these rules.

The difference between Ch and C is that if an object that has automatic storage duration not initialized explicitly, its value is indeterminate in C, whereas Ch will apply the above initialization rules. The initializer for a scalar shall be a single expression, optionally enclosed in braces. The initial value of the object is that of the expression (after conversion); the same type constraints and conversions as for simple assignment apply, taking the type of the scalar to be the unqualified version of its declared type. For example,

```
> int i = 3.0/2
> i
1
>
```

The variable `i` is initialized by the result of the expression `3.0/2` whose type has been converted from float to int.

An array of character type may be initialized by a character string literal, optionally enclosed in braces. Successive characters of the character string literal (including the terminating null character if there is room

or if the array is of unknown size) initialize the elements of the array. Similarly, an array with element type compatible with **wchar_t** may be initialized by a wide string literal, optionally enclosed in braces. For example, the array `str1` with size of 80 bytes is initialized by a string literal `"this is a string"`

```
> char str1[80] = "this is a string"
> str1
this is a string
>
```

If an array of unknown size is initialized, its size is determined by the largest indexed element with an explicit initializer. At the end of its initializer list, the array no longer has incomplete type. For example,

```
> char str2[] = "this is a string"
> str2
this is a string
>
```

The size of the array of char as the same as the length of string "`this is a string`" plus 1, which is for the terminating null.

The initializer for an object that has aggregate or union type shall be a brace-enclosed list of initializers for the elements or named members. For example, The variable `s1`, which is an object of struct, is initialized by the brace-enclosed list `{1 ,2}`. The member `s1.a` is set to 1, and `s2.b` to 2.

```
> struct {int a, b;} s1 = {1, 2};
> s1
.a = 1
.b = 2
>
```

The initialization shall occur in initializer list order, each initializer provided for a particular subobject overriding any previously listed initializer for the same subobject; all subobjects that are not initialized explicitly shall be initialized implicitly.

If the aggregate or union contains elements or members that are aggregates or unions, these rules apply recursively to the subaggregates or contained unions. If the initializer of a subaggregate or contained union begins with a left brace, the initializers enclosed by that brace and its matching right brace initialize the elements or members of the subaggregate or the contained union. Otherwise, only enough initializers from the list are taken to account for the elements or members of the subaggregate or the first member of the contained union; any remaining initializers are left to initialize the next element or member of the aggregate of which the current subaggregate or contained union is a part. For example, the declaration

```
int y[3][3] = {
   { 1, 3, 5 },
   { 2, 4, 6 },
   { 3, 5, 7 },
};
```

is a definition with a fully bracketed initialization. 1, 3, and 5 initialize the first row of `y`, i.e. the array object `y[0]`. Likewise the next two lines initialize `y[1]` and `y[2]`. In the declaration below

```
int y[3][3] = {
  1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

The initializer for `y[0]` does not begin with a left brace, so three items from the list are used. Likewise the next three are take successively for `y[1]` and `y[2]`. It has the same effect as the previous fully bracketed initialization. In the following commands

```
> struct {int a[3], b;} s2[] = {{1}, 2}
> s2[0]
.a = 1 0 0
.b = 0
> s2[1]
.a = 2 0 0
.b = 0
>
```

the declaration is a definition with an inconsistently bracketed initialization. It defines an array with two element structures, `s2[0].a[0]` is 1 and `s[1].a[0]` is 2; all the other elements are zero.

If there are fewer initializers in a brace-enclosed list than there are elements or members of an aggregate, or fewer characters in a string literal used to initialize an array of known size than there are elements in the array, the remainder of the aggregate shall be initialized implicitly the same as objects that have static storage duration.

```
> struct {int a, b;} s3 = {1}
> s3
.a = 1
.b = 0
>
```

The first member is initialized as 1, and the others are initialized implicitly as 0.

In Ch, non-constant expressions, generic functions, and functions defined in function files can be used as initializers for objects of both static and dynamic duration, with one exception. A function defined in a function file cannot be used as an initializer for static variables the function or block scope as illustrated in code below. Function `hypot()` defined in function file `hypot.chf` cannot be used for initialization of identifier `d1` which is a static variable in the function scope.

```
#include <math.h>
int main () {
    static double d = hypot(3,4);  // Error: hypot is not generic function
}
```

Chapter 7

Operators and Expressions

The operators used in Ch are summarized in Table 7.1. An operator has higher precedence than operators at the lower level. Operators at the same level have the same precedence. Operators with the same precedence will associate the operands according to their associativities. Unary operators, ternary conditional operator and comma operator are right associative; all others are left associative.

Table 7.1: Precedence and associativity of operators

Operations	Associativity
`::`	
`( ) [ ]`	left to right
function_name( )	right to left
`. ->`	left to right
`` ` ! ~ ++ -- + - * & (type) ``	right to left
`* / % .* ./`	left to right
`+ -`	left to right
`<< >>`	left to right
`< <= > >=`	left to right
`== !=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`^^`	left to right
`||`	left to right
`?:`	right to left
`= += -= *= /= %= &= |= <<= >>=`	right to left
`,`	left to right

The operation precedence for different operators in Ch is in full compliance with the C standard. The exclusive-or operator `^^`, command substitution operator `` ` ``, array multiplication operator '`.*`', and array division operator '`./`' are introduced in Ch. Following the C standard, the algorithms and resultant data types of operations for floating-point numbers will depend on the data types of operands in Ch. The conversion rules for char, int, float, and double in Ch follow the type conversion rules defined in the C standard. A data

Table 7.2: Negation results

Negation −							
operand	−Inf	−x1	−0.0	0.0	x2	Inf	NaN
result	Inf	x1	0.0	−0.0	−x2	−Inf	NaN

Table 7.3: Addition results

Addition +							
left operand	right operand						
	−Inf	−x1	−0.0	0.0	x2	Inf	NaN
Inf	NaN	Inf	Inf	Inf	Inf	Inf	NaN
y2	−Inf	y2−x1	y2	y2	y2+x2	Inf	NaN
0.0	−Inf	−x1	0.0	0.0	x2	Inf	NaN
−0.0	−Inf	−x1	−0.0	0.0	x2	Inf	NaN
−y1	−Inf	−y1−x1	−y1	−y1	−y1+x2	Inf	NaN
−Inf	−Inf	−Inf	−Inf	−Inf	−Inf	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

type that occupies less memory can be converted to a data type that occupies more memory space without loss of any information. For example, a char integer can be cast into int or float without problem. However, a reverse conversion may result in loss of information. The order of real numbers in Ch ranges from char, int, float, to double. The char data type is the lowest and double the highest. Like C, the algorithms and resultant data types of the operations depend on the data types of operands in Ch. For binary operations, such as addition, subtraction, multiplication, and division, the resultant data type will take the higher order data type of two operands. For example, the addition of two float numbers will result in a float number whereas the addition of a float number and a double number will become a double number.

The operation rules for regular real numbers and metanumbers in Ch are presented in Tables 7.2 to 7.12. In Tables 7.2 to 7.12, x, x1, and x2 are regular positive normalized floating-point numbers in float or double; metanumbers 0.0, −0.0, Inf, −Inf, and NaN are constants or the values of float or double variables. By default, the constant metanumbers are float constants.

Table 7.4: Subtraction results.

Subtraction −							
left operand	right operand						
	−Inf	−x1	−0.0	0.0	x2	Inf	NaN
Inf	Inf	Inf	Inf	Inf	Inf	NaN	NaN
y2	Inf	y2+x1	y2	y2	y2−x2	−Inf	NaN
0.0	Inf	x1	0.0	0.0	−x2	−Inf	NaN
−0.0	Inf	x1	0.0	−0.0	−x2	−Inf	NaN
−y1	Inf	−y1+x1	−y1	−y1	−y1−x2	−Inf	NaN
−Inf	NaN	−Inf	−Inf	−Inf	−Inf	−Inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Table 7.5: Multiplication results

Multiplication $*$							
left operand	right operand						
	$-$Inf	$-$x1	-0.0	0.0	x2	Inf	NaN
Inf	$-$Inf	$-$Inf	NaN	NaN	Inf	Inf	NaN
y2	$-$Inf	$-$y2$*$x1	-0.0	0.0	y2$*$x2	Inf	NaN
0.0	NaN	-0.0	-0.0	0.0	0.0	NaN	NaN
-0.0	NaN	0.0	0.0	-0.0	-0.0	NaN	NaN
$-$y1	Inf	y1$*$x1	0.0	-0.0	$-$y1$*$x2	$-$Inf	NaN
$-$Inf	Inf	Inf	NaN	NaN	$-$Inf	$-$Inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Table 7.6: Division results

Division /							
left operand	right operand						
	$-$Inf	$-$x1	-0.0	0.0	x2	Inf	NaN
Inf	NaN	$-$Inf	NaN	NaN	Inf	NaN	NaN
y2	-0.0	$-$y2/x1	$-$Inf	Inf	y2/x2	0.0	NaN
0.0	-0.0	-0.0	NaN	NaN	0.0	0.0	NaN
-0.0	0.0	0.0	NaN	NaN	-0.0	-0.0	NaN
$-$y1	0.0	y1/x1	Inf	$-$Inf	$-$y1/x2	-0.0	NaN
$-$Inf	NaN	Inf	Inf	$-$Inf	$-$Inf	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

7.1 Arithmetic Operators

For the negation operation shown in Table 7.2, the data type of the result is the same as the data type of the operand, and a real number will change its sign by the negation operation. There is no $-$NaN in Ch. The leading plus sign '+', a unary plus operator, in an expression such as $+57864 - x$ will be ignored. It should be pointed out that the negation of a positive integer zero is still a positive zero. Based on two's complement representation of negative integer numbers discussed before, we cannot represent Inf and NaN in the int data type.

According to the IEEE 754 standard, some operations depend on the rounding mode. For example, in case of rounding toward zero, overflow will deliver FLT_MAX rather than Inf with the appropriate sign. This rounding mode is necessary for Fortran implementation and for machines that lack infinity. If the rounding mode is rounded toward $-\infty$, both $-0.0 + 0.0$ and $0.0 - 0.0$ deliver -0.0 rather than 0.0. For scientific programming, consistency and determinacy are essential. Ch is currently implemented using the default rounding mode of round to nearest so that overflow will result in Inf, and both $-0.0 + 0.0$ and $0.0 - 0.0$ deliver 0.0 as shown in Tables 7.3 and 7.4. Note that the modulus operator % in Ch is C compatible.

For addition, subtraction, multiplication, and division operations shown in Tables 7.3 to 7.6, the resultant data type will be double if any one of two operands is double; otherwise, the result is a float. The mathematically indeterminate expressions such as $\infty - \infty, \infty * 0.0, \infty/\infty$, and $0.0/0.0$ will result in NaNs. The values of ± 0.0 play important roles in the multiplication and division operations. For example, a finite positive value of x2 divided by 0.0 results in a positive infinity $+\infty$ whereas division by -0.0 will create a negative infinity $-\infty$. If any one of the operands of binary arithmetic operations is NaN, the result is NaN.

Element-wise multiplication and division of two computational arrays can be performed using array multiplication operator '`.*`' and array division operator '`./`', respectively. Details about array multiplication operator '`.*`' and array division operator '`./`' for operands of computational array are described in Chapter 16.

7.2 Relational Operators

For relational operations given in Tables 7.7-7.12, the result is always an integer with a logical value of 1 or 0 corresponding to TRUE or FALSE, which are predefined system constants. According to the IEEE 754 standard, there is a distinction between $+0.0$ and -0.0 for floating-point numbers. In Ch, the value of 0.0 means that the value approaches zero from positive numbers along the real line and it is a zero; the value of -0.0 means that the value approaches zero from negative numbers along the real line and it is infinitely smaller than 0.0 in many cases. Signed zeros $+0.0$ and -0.0 in a Ch program behave like correctly signed infinitesimal quantities 0_+ and 0_-, respectively. Although there is a distinction between -0.0 and 0.0 for floating-point numbers in many operations, according to the IEEE 754 standard, the comparison shall ignore the sign of zeros so that -0.0 equals 0.0 in relational operations. Functions such as `signbit(x)` and `copysign(x,y)` can be used to handle signs of expressions. The value of -0.0 could be regarded different from 0.0 for comparison operations in Ch. For the convenience of porting C code to Ch, zero is unsigned in comparison operations. The equality for metanumbers has different implications in Ch. Two identical metanumbers are considered to be equal to each other. As a result, comparing two Infs or two NaNs will get logical TRUE. This is just for the convenience of programming because, mathematically, the infinity of ∞ and not-a-number of NaN are undefined values that cannot be compared with each other. Metanumbers of Inf, $-$Inf, and NaN in Ch are treated as regular floating-point numbers consistently in arithmetic, relational, and logical operations.

The relational operators `==, !=, <,` and `>` for comparison of two strings with one of two operands is the built-in string type **string_t** are described in section 6.1.13.

Table 7.7: Less than comparison results

Less than comparison $<$							
left operand	right operand						
	$-$Inf	$-$x1	$-$0.0	0.0	x2	Inf	NaN
Inf	0	0	0	0	0	0	0
y2	0	0	0	0	y2 $<$ x2	1	0
0.0	0	0	0	0	1	1	0
$-$0.0	0	0	0	0	1	1	0
$-$y1	0	$-$y1 $<$ $-$x1	1	1	1	1	0
$-$Inf	0	1	1	1	1	1	0
NaN	0	0	0	0	0	0	0

Table 7.8: Less than or equal comparison results

Less or equal comparison $<=$							
left operand	right operand						
	$-$Inf	$-$x1	$-$0.0	0.0	x2	Inf	NaN
Inf	0	0	0	0	0	1	0
y2	0	0	0	0	y2 $<=$ x2	1	0
0.0	0	0	1	1	1	1	0
$-$0.0	0	0	1	1	1	1	0
$-$y1	0	$-$y1 $<=$ $-$x1	1	1	1	1	0
$-$Inf	1	1	1	1	1	1	0
NaN	0	0	0	0	0	0	0

Table 7.9: Equal comparison results

Equal comparison $==$							
left operand	right operand						
	$-$Inf	$-$x1	$-$0.0	0.0	x2	Inf	NaN
Inf	0	0	0	0	0	1	0
y2	0	0	0	0	y2 == x2	0	0
0.0	0	0	1	1	0	0	0
$-$0.0	0	0	1	1	0	0	0
$-$y1	0	$-$y1 == $-$x1	0	0	0	0	0
$-$Inf	1	0	0	0	0	0	0
NaN	0	0	0	0	0	0	0

Table 7.10: Greater than or equal comparison results

Greater or equal comparison $>=$							
left operand	right operand						
	$-$Inf	$-$x1	-0.0	0.0	x2	Inf	NaN
Inf	1	1	1	1	1	1	0
y2	1	1	1	1	y2 >= x2	0	0
0.0	1	1	1	1	0	0	0
-0.0	1	1	1	1	0	0	0
$-$y1	1	$-$y1 >= $-$x1	0	0	0	0	0
$-$Inf	1	0	0	0	0	0	0
NaN	0	0	0	0	0	0	0

Table 7.11: Greater than comparison results

Greater than comparison $>$							
left operand	right operand						
	$-$Inf	$-$x1	-0.0	0.0	x2	Inf	NaN
Inf	1	1	1	1	1	0	0
y2	1	1	1	1	y2 > x2	0	0
0.0	1	1	0	0	0	0	0
-0.0	1	1	0	0	0	0	0
$-$y1	1	$-$y1 > $-$x1	0	0	0	0	0
$-$Inf	0	0	0	0	0	0	0
NaN	0	0	0	0	0	0	0

Table 7.12: Not equal comparison results

Not equal comparison !=							
left operand	right operand						
	$-$Inf	$-$x1	-0.0	0.0	x2	Inf	NaN
Inf	1	1	1	1	1	0	1
y2	1	1	1	1	y2 != x2	1	1
0.0	1	1	0	0	1	1	1
-0.0	1	1	0	0	1	1	1
$-$y1	1	$-$y1 != $-$x1	1	1	1	1	1
$-$Inf	0	1	1	1	1	1	1
NaN	1	1	1	1	1	1	1

7.3 Logical Operators

In Ch, there are four logical operators `!, &&, ||`, and `^^` corresponding to logical operations `not, and, inclusive or`, and `exclusive or`, respectively. The operations of `!, ||, &&` in Ch comply with the C standard. The operator `^^` is introduced in Ch due to the consideration of programming convenience so that logical and bitwise exclusive-or operators are orthogonal. Note that, like C, Ch will evaluate the right operand of both the `&&` and `||` operations will be evaluated only if the left operand evaluates to TRUE and FALSE, respectively. This "short circuit" behavior for the `^^` operator does not exist because, for either TRUE or FALSE of the first operand, an exclusive-or operation can return TRUE, depending on the second operand. The precedence of operator `^^` is higher than operator `||`, but lower than `&&`. This operation precedence is similar to that for bitwise operators `&, |`, and `^`, which will be discussed in the next section. Because there are only two values of either TRUE or FALSE for logical operations, the values of ± 0.0 are treated as logical FALSE while the metanumbers $-$Inf, Inf, and NaN are considered as logical TRUE. For example, evaluations of $!(-0.0)$ and !NaN will get the values of 1 and 0, respectively.

7.4 Bitwise Operators

In Ch, there are six bitwise operators `&, |, ^, <<, >>`, and `~`, corresponding to `bitwise and, inclusive or, exclusive or, left shift, right shift`, and `one's complement`, respectively. These operators in Ch are in full compliance with the C standard. They can only be applied to integral data that are char and int at its current implementation of Ch. The returned data type depends on the data types of operands. The result of the unary operator `~` keeps the data type of its operand. Results of binary operators `&, |`, and `^` will have the higher data type of two operands. The binary operators `<<` and `>>` return the data type of the left operand.

However, some undefined behaviors in C are defined in Ch. For operators `<<` and `>>`, the right operand can be any data type so long as it can be converted into int internally whereas the right operand must be a positive integral value in C. In Ch, if the right operand is a negative integral value that may be converted from a floating-point data, the shifting direction will be reversed. For example, the expression of 7 `<<` -2.0 is equivalent to 7 `>>` 2.0 in Ch. Therefore, only one of these two shift operators is needed in Ch. The use of operator `<<` is recommended for Ch programming. A program with dual shift directions for one operator can be cleaner as compared with unidirectional shifts of two operators.

7.5 Assignment Operators

Besides the regular assignment statement, there are nine assignment operators of `+=, -=, *=, =, &=, |=, ^=, <<=`, and `>>=`. These assignment operators are C compatible. An *lvalue* is any object that occurs on the left hand side of an assignment statement. The lvalue refers to a memory such as a variable or pointer, not a function or constant. The Ch expression of `lvalue op= rvalue` is defined as `lvalue = lvalue op rvalue` where `lvalue` is any valid lvalue including complex numbers and it is only evaluated once. For example, `i += 3` is equivalent to `i = i+3`, and **real**`(c) *= 2` is the same as `real(c) = real(c)*2`. But, statement `*ptr++ += 2` is different from statement `*ptr++ = *ptr++ +2` because lvalue `*ptr++` contains an increment operation. The operation rules for operators of `+, -, *, /, &, |, ^, <<`, and `>>` have been discussed in the previous sections.

7.6 Conditional Operator

The conditional operator '`?:`' introduces a conditional expression in Ch. The following conditional expression

```
r = op1 ? op2 : op3;
```

is equivalent to

```
if(op1 != 0)
  r = op2;
else
  r = op3;
```

In a conditional expression, the first and second operands are separated by a question mark '`?`' and the second and third operands separated by a colon '`:`'. The execution of a conditional expression proceeds as follows:

1. The first operand is evaluated.
2. The second operand is evaluated only if the first does not evaluate to 0. The third operand is evaluated only if the first evaluates to 0.
3. The result is the value of the second or third operand, whichever is evaluated.

The first operand of a conditional expression shall have scalar type. For the second and third operands, one of the following shall hold.

1. Both operands have arithmetic type. The result type is determined by the usual arithmetic conversions.
2. Both operands have compatible class, structure or union types. The result is the class, structure or union type.
3. Both operands have **void** type. The result has **void** type.
4. Both operands are pointers to compatible types. The result is a pointer to the composite type.
5. One operand is a pointer and the other is NULL. The result has the type of the operand which is not NULL.
6. One operand is a pointer to an object or incomplete type and the other is a pointer to **void**. The result is a pointer to **void**.
7. Both operands are computational arrays of the same shape. The result is a computational array with the higher order data type of the two operands.

Conditional expressions are right-associative. For example,

```
op1 ? op2 : op3 ? op4 : op5 ? op6: op7
```

is handled as

```
op1 ? op2 : (op3 ? op4 : (op5 ? op6: op7))
```

The following commands are examples of conditional expressions with operands of computational array type.

```
> 5 ? 1 : 2
1
> 0 ? 1 : 0 ? 3 : 4 // right-association
4
> 0 ? 1.0 : 2       // data type conversion
2.0000
> 1 ? (array float [2][3])1 : (array int [2][3])2
1.00 1.00 1.00
1.00 1.00 1.00
> 0 ? (array float [2][3])1 : (array int [2][3])2
2.00 2.00 2.00
2.00 2.00 2.00
```

In Program 7.1, the function `func()` is called in the main function where the argument passed is the result of a conditional expression. Pointers `p1` and `p2` are used as operands in conditional expression

```
p1  =  (p1)? p1 : p2;
```

Then structs `s1` and `s2` are used as operands in conditional expression

```
i  = (1 ? s1 : s2).ii;
```

The output of Program 7.1 is displayed in Program 7.2.

```
struct tag {
  int ii;
  int *pp;
} s1, s2, *ps1, *ps2;

int func(int i) {
  printf("i = %d\n", i);
  return 0;
}

int main() {
  int i = 1;
  int *p1 = NULL, *p2 = &i;

  func(i? 5 : 8);         // passed as argument of function

  p1  =  (p1)? p1 : p2;   // oparands is pointers
  printf("p1 = %p\n", p1);

  ps1 = &s1;
  ps2 = &s2;
  s1.ii = 10;
  s2.pp = &s1.ii;
  i  = (1 ? s1 : s2).ii;      // operands of structure
  p1  = (0 ? ps1 : ps2)->pp;
  printf("i = %d\n", i);
  printf("*p1 = %d\n", *p1);
}
```

Program 7.1: Example of conditional expression with operands of different data type.

```
i = 5
p1 = 40063528
i = 10
*p1 = 10
```

Program 7.2: Output of Program 7.1.

7.7 Cast Operators

7.7.1 Cast Operators

In Ch, the explicit type conversion is not necessary in many cases when C needs it. For example, `aptr[3] = malloc(90)` is valid in Ch. However, sometimes it is necessary to convert a value of one type explicitly to a value of another type. This can be achieved by the traditional C cast operation `(type)expr` where `expr` is a Ch expression and `type` is a data type of a single object such as **char**, **int**, **float**, **double** or any pointer declaration identifiers such as `char *, double *, complex *`, etc. For example, (**int**)9.3, (**float**)ptr, (**double**)9, (**float**$*$)&i, and (**complex**$*$)iptr are valid Ch expressions.

The **sizeof**() function can also use a type identifier. For example, `ptr = malloc(5+sizeof(int*)+sizeof((int)2.3) + sizeof((int)float(90)+7))` is a valid Ch statement.

One important feature of C is its capability for hardware interface by accessing a specific memory location in a computer. This is achieved by pointing a pointer to a specific memory location or register. This hardware interface capability is retained in Ch. For example, the following statements will assign the integer value at the memory location $(68\mathrm{FFE})_{16}$ to variable i and set the byte at the memory address $(\mathrm{FF000})_{16}$ to $(01101001)_2$;

```
char *cptr;
int i, *iptr, j;
iptr = (int *)0X68FFE;   // point to the memory location at 0X68FFE
i = *iptr;               // i equals the value at 0X68FFE;
cptr = (char *)0XFF000;  // point to the memory location at 0XFF000
*cptr = 0B01101001;      // 0B01101001 is assigned to 0XFF000
cptr = (float *)cptr + 1;// cptr  points to 0XFF004, not 0XFF001.
                         // note: (float *)cptr++ is (float *)(cptr++)
j = int(cptr);           // j becomes 0XFF004
```

Note that an integral value cannot be assigned to a pointer variable without an explicit type cast, and vice versa. The lower segment of the memory in a computer is usually reserved for the operating system and system programs. An application program will be terminated with exception handling if these protected segments of memory are messed up by pointers.

7.7.2 Functional Type Cast Operators

There is an additional functional type casting operation in Ch in the form of `type(expr)` for data types of single object or `type(expr1, expr2, ...)` for data types of aggregate such as complex. In this functional type casting operation, `type` shall not be a pointer data type. For example, **int**(9.3), **complex**(**float**(3), 2), and **complex**(**double**(3), 2) are valid Ch expressions. Operation **float**() is the same as **real**() if they are

used as operands. However, function **real**() can be used as an lvalue whereas **float**() cannot. More information about function **real**() can be found in section 13.6. Examples of functional type cast operations are shown below.

```
char char(double)
char char(complex)
char char(pointer_type)
complex complex(float, float)
double complex complex(double, float)
double complex complex(float, double)
double complex complex(double, double)
double double(double)
double double(complex)
double double(pointer_type)
float float(double)
float float(complex)
float float(pointer_type)
int int(double)
int int(complex)
int int(pointer_type)
long long(double)
long long(complex)
long long(pointer_type)
short short(double)
short short(complex)
short short(pointer_type)
signed signed(double)
signed signed(complex)
signed signed(pointer_type)
unsigned unsigned(double)
unsigned unsigned(complex)
unsigned unsigned(pointer_type)
```

7.8 Comma Operator

The comma operator ',' introduces comma expression in Ch. The comma expression consists of two expressions separated by a comma. For example,

```
a = 1, ++a;
```

The comma operator is syntactically left-associative. The following expression

```
a = 1, ++a, a + 10;
```

is equivalent to

```
((a = 1), ++a), a + 10;
```

The left operand of a comma operator is evaluated as a void expression first. Then the right operand is evaluated; the result has its type and value. For example,

```
> a = 1, ++a, a + 10
12
```

The comma operator cannot appear in contexts where a comma is used as a separate item such as the argument list of a function. In these cases, it can be used within parenthesis. For example,

```
int func(int i1, int i2);
int t;
...
func((t = 1, t + 2), 2);
```

7.9 Unary Operators

7.9.1 Address and Indirection Operators

The unary operator & gives the address of an object. The operator &, which is C compatible, can only be applied to a valid lvalue.

When a unary indirection operator `*` is applied to a pointer, it accesses the object to which the pointer points. A pointer and an integer can be added or subtracted. For example, for variables `ptr`, `ptr1`, and `ptr2` of pointer type and n of integral value, the expression `ptr+n` gives the address of the nth object beyond the one `ptr` currently points to. The memory locations of pointers `ptr+n` and `ptr` are `n*sizeof(*ptr)` bytes apart, that is, n is scaled to `n*sizeof(*ptr)` bytes according to declaration of pointer variable `ptr`. Pointer subtraction for pointers with the same data type is permitted. If `ptr1` $>$ `ptr2, ptr1` $-$ `ptr2` gives the number of objects between `ptr2` and `ptr1`. An array of pointers can also be declared. When a pointer is declared, it is initialized to zero. The symbolic constant NULL, instead of zero, can be used in the program. If `ptr` is NULL, the operand `*ptr` in an expression is evaluated as zero. When `*ptr`, with `ptr` equal to NULL, is used as an lvalue, a memory of **sizeof**(*ptr) will be allocated automatically for pointer `ptr`. In both cases, the system will print out warning messages. The automatic memory allocation for a pointer that does not point to a valid location can avoid a system crash.

Two pointers and constant NULL can be used in the relational operations `<, <=, ==, >=, >`, and `!=`. In assignment and relational operations, pointers with different data types can work together without explicit type conversions. For example, the following is a valid Ch program.

```
int *iptr;
float *fptr;
iptr = (int *)malloc(90);
fptr = malloc(80); // fptr = (float *)malloc(80)
if(iptr !=  NULL && iptr != fptr)
  free(iptr);
iptr = fptr;
```

In Ch, not only are all the variables initialized to zero when they are declared, but also the memory allocated by either function **malloc**(), **calloc**() or **realloc**() , is initialized to zero. This can avoid some unexpected errors. In C, the content for the memory allocated by functions **malloc**() and **realloc**() will be random values. Furthermore, the casting operation for three memory allocation functions **malloc**(), **calloc**(), and **realloc**() can be omitted in Ch. If no memory is available, these functions will return NULL and the system will print out error messages. The function **free**(ptr) will deallocate the memory allocated by these three functions and set pointer `ptr` to NULL. In C, `ptr` is not set to NULL when the memory it points to is deallocated. This dangling memory makes the debugging of the C program very difficult because the problem will not surface

until this deallocated memory is claimed again by other parts of the program. Other related functions such as **memcpy**() in Ch for memory manipulations are C compatible.

As described before, there are several system defined parameters such as NaN, Inf, FLT_MAX, INT_MIN, FLT_EPSILON, etc.. These parameters cannot be used as lvalues so that an accidental change of values of these parameters can be avoided. However, if really necessary, the values of these parameters can be modified by accessing their memory locations through pointers. For example, a numerical algorithm may depend on the parameters FLT_EPSILON and Inf. One can change the values of FLT_EPSILON to 10^{-4} and Inf to FLT_MAX by the following Ch code

```
float *fptr;
fptr = & FLT_EPSILON; *fptr = 1e-4;
fptr = &Inf; *fptr = FLT_MAX;
```

which may, in effect, change the underlying numerical algorithm.

7.9.2 Increment and Decrement Operators

C is well-known for the succinctness of its syntax. The increment operator ++ and decrement operator -- are unique to C. These two operators in Ch are compatible with C. The increment operator ++ adds 1 to its operand whereas the decrement operator -- subtracts 1. If ++ or -- is used as a prefix operator, the expression increments or decrements the operand before its value is used, respectively. If it is used as a postfix operator, the operation will be performed after its value has been used.

A single + is treated as an addition or unary plus operator depending on the context. Likewise, a single − can be a subtraction or unary negation operator. For example, the following is valid Ch code.

```
i = +(-9);            // unary plus and negation operators
i++;                  // i = i+1
j = ++i--;            // i = i+1; j = i; i = i-1;
j = ++i;              // i = i+1; j = i;
j = i--;              // j = i; i = i-1;
i = (*ptr++)++;       // ptr = ptr + 1; i = *ptr; *ptr = *ptr + 1;
```

By definition, `++lvalue` means `lvalue = lvalue + 1` and expression `lvalue + 1`, and `lvalue--` is equivalent to expression `lvalue - 1` and `lvalue = lvalue - 1`. The ++ and -- operators can be applied to any valid lvalues, not just integral variables, so long as the lvalue can add or subtract an integer value of 1 according to internal data conversion rules. The following is the valid Ch code.

```
int i, a[4], *aptr[5];
complex z, *zptr;       // declare complex variable and complex pointer
z = z++;                // z = z + 1; z is a complex variable
zptr = (complex *)malloc(sizeof(complex)*90);
aptr[3] = malloc(90);   // aptr[3] = (int *)malloc(90);
/* imag(z)=complex(0.0, 4.0); zptr=zptr+1; *aptr[3]=1; i=i-1 */
imag(z) = ++real(++++(zptr+++2*(int)real(++*aptr[3+i--])));
real(z)++;              // real(z) = real(z) + 1;
--imag(*zptr);          // imag(*zptr) = imag(*zptr) - 1;
a[--i] = a[2]++;        // i = i - 1; a[i] = a[2]; a[2] = a[2] + 1;
```

Details about complex numbers and functions **real**() and **imag**() in Ch are described in section 13.6. Note that the memory allocated by function **malloc**() is initialized to zero.

7.9.3 Command Substitution Operator

Command substitution operator ` returns the output from a command as a string. For example,

```
   string_t s;
s = `ls`;
```

When two command substitution operators are used together, character of formfeed, newline, carriage return, horizontal tab, and vertical tab from the output of the command is replaced by a blank space character. For example,

```
   string_t s;
s = ``ls``;
```

7.10 Member Operators

Operators `.` and `->` are called member operators. A member of class, structure, or union is referred to by these two member operators. The first operand of the `.` operator shall have a class, structure or union type, and the second operand shall name a member of that type.
The first operand of the `->` operator shall have type “pointer to class”, “pointer to structure”, or “pointer to union”, and the second operand shall name a member of the type pointed to.
For example,

```
struct tag {
    int i;
    double d;
} s, *p;
s = &p;
s.i = 10;
p->i += s.i;
```

Chapter 8

Statements and Control Flow

A *statement* specifies an action to be performed. Except as indicated, statements are executed in sequence. A *full expression* is an expression that is not part of another expression or declarator. Each of the following is a full expression: an initializer; the expression in an expression statement; the controlling expression of a selection statement (if or switch); the controlling expression of a **while** or **do** statement; each of the (optional) expressions of a **for** statement; the (optional) expression in a **return** statement. The end of a full expression is a sequence point.

8.1 Simple and Compound Statements

A *compound statement* is a block enclosed with a pair of braces. A *block* allows a set of declarations and statements to be grouped into one syntactic unit. The initializers of objects that have automatic storage duration, and the variable length array declarators of ordinary identifiers with block scope, are evaluated and the values are stored in the objects (including storing an indeterminate value in objects without an initializer) each time the declaration is reached in the order of execution, as if it were a statement, and within each declaration in the order that declarators appear. For example.

```
int i;  // simple statement
{       // compound statement
  int i;
  i =90;
  ...
}
```

8.2 Expression and Null Statements

An expression statement contains an expression only. The expression is evaluated as a void expression for its side effects. A *null statement* consisting of just a semicolon performs no operation.

If a function call is evaluated as an expression statement for its side effects only, the discarding of its value may be made explicit by converting the expression to a void expression by means of a cast as shown below:

```
int p(int);
/* ... */
(void)p(0);
```

A null statement can be used to supply an empty loop body to the iteration statement as shown in the program fragment below:

```
char *s;
/* ... */
while(*s++ != '\0')
        ;
```

A null statement may also be used to carry a label just before the closing } of a compound statement.

```
while(loop1) {
  /* ... */
  do {
    /* ... */
    if(want_out)
       goto end_loop1;
    /* ... */
  } while (loop2);
  /* ... */
  end_loop1: ;
}
```

8.3 Selection Statements

A selection statement selects among a set of statements depending on the value of a controlling expression. A selection statement is a block whose scope is a strict subset of the scope of its enclosing block. Each associated substatement is also a block whose scope is a strict subset of the scope of the selection statement.

8.3.1 If Statements

The syntax of an if-statement is as follows:

```
if(expression)
  statement
```

The controlling expression of an **if** statement shall have scalar type. The statement is executed if the expression compares unequal to 0.

Ch supports the header file **stdbool.h** added in C99. the Boolean type **bool** is defined. Macros **true** and `false` are defined to handle Boolean numbers. The macro **true** is defined as `1`, and macro **false** is defined as `0`. The code fragment below illustrates how **bool** type can be used in conditional expressions.

```
#include <stdbool.h>
bool i = true;
/* ... */
if (i) {
  i = false;
}
```

8.3.2 If-Else Statements

The syntax of an if-else statement is as follows:

```
if(expression)
  statement1
else
  statement2
```

The controlling expression of an **if** statement shall have scalar type. The first substatement is executed if the expression compares unequal to 0. The second substatement is executed if the expression compares equal to 0. If the first substatement is reached via a label, then the second substatement is not executed.

8.3.3 Else-If Statements

The syntax of the else-if statement is as follows:

```
if(expression1)
  statement1
else if (expression2)
  statement2
else if (expression3)
  statement3
else
  statement4
```

Semantically, the syntax of else-if statement is an extention of the previous if-else statement. An **else** is associated with the lexically nearest preceding **if** that is allowed by the syntax. The above statement can be rearranged as

```
if(expression1)
  statement1
else
   if (expression2)
      statement2
   else
      if (expression3)
         statement3
      else
         statement4
```

8.3.4 Switch Statements

The syntax of a switch statement is as follows:

```
switch(expression) {
  case const-expr1:
     statement1
     break;
  case const-expr2:
     statement2
```

```
        break;
      default:
        statement
        break;
   }
```

The controlling expression of a **switch** statement shall have integer or string type. The expression of each **case** label shall be an integer constant expression or string and no two of the **case** constant expressions in the same **switch** statement shall have the same value after conversion. There may be at most one **default** label in a **switch** statement. A **switch** statement causes control to jump to, into, or past the statement that is the *switch body*, depending on the value of a controlling expression, and on the presence of a **default** label and the values of any **case** labels on or in the switch body. A **case** or **default** label is accessible only within the closest enclosing **switch** statement. The number of **case** values in a switch statement is not limited.

The integer promotions are performed on the controlling expression. The constant expression in each **case** label is converted to the promoted type of the controlling expression. If a converted value matches that of the promoted controlling expression, control jumps to the statement following the matched **case** label. Otherwise, if there is a **default** label, control jumps to the labeled statement. If no converted **case** constant expression matches and there is no **default** label, no part of the switch body is executed.

In the code fragment below,

```
switch (expr) {
   int i = 10;
   f(i);
case 0:
   i = 20;
   /* falls through into default code */
default:
   printf("%d\n", i);
}
```

the object whose identifier **i** exists with automatic storage duration within the block, but is never initialized. Thus, if the controlling expression has a nonzero value, the call to the **printf** function will access an indeterminate value. Similarly, the call to the function **f** cannot be reached.

The controlling expression of a switch statement can be string, instead of integer, as shown in the example below. Accordingly, all case constant expressions for such a switch statement shall also be string.

```
string_t str;
str = ‘hostname‘;  // get host name from command ’hostname’
char *s="host2";
switch (str) {     // or switch (s)
  case "host1":
    printf("s = host1\n");
    break;
  case "host2":
    printf("s = host2\n");
    break;
  default:
    break;
}
```

8.4 Iteration Statements

An iteration statement causes a statement called the *loop body* to be executed repeatedly until the controlling expression compares equal to 0. The loop body of an iteration statement is a block.

8.4.1 While Loop

The syntax of a `while` statement is as follows:

```
while(expression)
  statement
```

The evaluation of the controlling expression takes place before each execution of the loop body.
The loop body is executed repeatedly until the controlling expression compares equal to 0.

For example, the following code fragment

```
int i =0;
while(i<5) {
  printf("%d ", i);
  i++;
}
```

will output

```
0 1 2 3 4
```

8.4.2 Do-While Loop

The syntax of a `do-while` statement is as follows:

```
do
  statement
while(expression);
```

The evaluation of the controlling expression takes place after each execution of the loop body. The loop body is executed repeatedly until the controlling expression compares equal to 0.

For example, the following code fragment

```
int i =0;
do {
  printf("i = %d\n", i);
  i++;
} while(i<5);
```

will output

```
0 1 2 3 4
```

The following code fragment

```
int i = 10;
do {
  printf("i = %d\n", i++);
} while(i<5);
```

will output

```
10
```

As shown in this example, the loop body is executed before the controlling expression is evaluated. The following code fragment with a while-loop will have no output, because the controlling expression of the while statement is evaluated first with a value of 0.

```
int i =10;
while(i<5)
  printf("%d ", i++);
```

8.4.3 For Loop

The syntax of a `for` statement is as follows:

```
for(expression1; expression2; expression3)
  statement
```

The expression *expression1* is evaluated as a void expression before the first evaluation of the controlling expression. The expression *expression2* is the controlling expression that is evaluated before each execution of the loop body. The expression *expression3* is evaluated as a void expression after each execution of the loop body. Both *expression1* and *expression3* can be omitted. An omitted *expression2* is replaced by a nonzero constant.

The for-loop is semantically equivalent to the following while-loop

```
expression1;
while(expression2) {
  statement
  expression3;
}
```

For example, the following code fragment

```
int i;
for(i=0; i<5; i++)
  printf("%d \n", i);
```

will produce the same output of

```
0 1 2 3 4
```

as in a while-loop of

```
int i =0;
while(i<5) {
  printf("%d ", i);
  i++;
}
```

More complicated expressions can be used in a for-loop statement as shown below.

```
int i, j=10;
for(i=0, j=10; i<10&&j>0; i++, j--) {
  printf("i=%d\n", i);
  printf("j=%d\n", j);
}
```

8.4.4 Foreach Loop

The syntax of a `foreach` statement is as follows:

```
foreach(token; expr1; expr2; expr3)
  statement
foreach(token; expr1; expr2)
  statement
foreach(token; expr1)
  statement
```

The foreach loop is used to handle iterations based on the condition of string type or pointer to char. The expressions *expr1, expr2,* and *expr3* shall have string type or pointer to char. The identifier *token* also shall have string type or pointer to char. In each iteration, the variable *token* takes a token from the original expression *expr1* separated by the delimiter *expr3*. The loop body is executed repeatedly until `token` is a NULL pointer or the same as `expr2`. This is achieved by comparing the controlling expression `(token==NULL || expr2!=NULL && !strcmp(token,expr2))` to 0. The omitted *expr2* and *expr3* are replaced by NULL and " ;", respectively.

As an example, the following code

```
char *token, *str="ab:12 cd ef", *cond="cd", *delimit=" :";
foreach(token; str; cond; delimit)
  printf("token= %s\n", token);
printf("after foreach token = %s\n", token);
printf("after foreach cond = %s\n", cond);
printf("after foreach delimi= %s\n", delimit);
```

gives the output of

```
token= ab
token= 12
after foreach token = cd
after foreach cond = cd
after foreach delimi=  :
```

In this example, the delimiters for token are characters of blank space and colon as shown in the value for the variable `delimit` in the program. The code below will create three directories *dir1*, *dir2*, and *dir3* in the current directory.

```
string_t token, str="dir1 dir2 dir3";
foreach(token; str) {
  mkdir $token
}
```

8.5 Jump Statements

A jump statement causes an unconditional jump to another place. To jump from one function to other function, functions **setjmp()** and **longjmp()** in header file `setjmp.h` should be used.

8.5.1 Break Statements

The `break` statement provides an early exit from `for, while, do-while, foreach` loops and `switch`. A break causes the innermost enclosing loop or switch to be exited immediately. A **break** statement shall appear only in a switch body or loop body. For example, the following code fragment

```
int i;
for(i=0; i<5; i++) {
   if(i == 3) {
       break;
   }
   printf("%d \n", i);
}
```

will produce the output of

```
0 1 2
```

8.5.2 Continue Statements

The `continue` statement causes the next iteration of the enclosing `for, while, do-while, foreach` loop to begin. A continue statement shall appear only in or as a loop body. In each of the statements

```
while (/* ... */) {
   /* ... */
   continue;
   /* ... */
   contin: ;
}
```

```
do {
   /* ... */
   continue;
   /* ... */
   contin: ;
} while (/* ... */);
```

```
for(/* ... */) {
   /* ... */
   continue;
   /* ... */
   contin: ;
}
```

```
foreach (/* ... */)
   /* ... */
   continue;
   /* ... */
   contin: ;
}
```

unless the **continue** statement shown is in an enclosed iteration statement in which case it is interpreted within that statement, it is equivalent to **goto contin;**.

For example, the following code fragment

```
int i;
for(i=0; i<5; i++) {
   if(i == 3) {
      continue;
   }
   printf("%d \n", i);
}
```

will produce the output of

```
0 1 2 4
```

8.5.3 Return Statements

A **return** statement terminates execution of the current function and returns control to its caller. A function may have any number of **return** statements. If a **return** statement with an expression is executed, the value of the expression is returned to the caller as the value of the function call expression. If the expression has a type different from the return type of the function in which it appears, the value is converted as if by assignment to an object having the return type of the function. A **return** statement with an expression shall not appear in a function whose return type is **void**. A **return** statement without an expression shall only appear in a function whose return type is **void**.

8.5.4 Goto Statements

A goto statement causes an unconditional jump to the statement prefixed by the named label in the enclosing function. A goto statement can transfer control either forward or backward within a function. For example,

```
for (/* ... */)
  for(/* ... */) {
    /* ... */
    if(emergency)
       goto hospital;
}
/* ... */
hospital:
   emergenceaction();

 void funt1(int j)
 {
   int funt2(int j)
   {
     if(j>10)
       goto label1;
     j = 10;
   }
   funct2(j)
   label1: exit(1);
 }
```

In a nested function, the flow of control can jump from an inner function to the enclosing outer function, where the label is defined. But, it cannot jump from an enclosing outer function to an inner function. For example,

```
int task() {
  int task1() {
    /* ... */
    if(student)
      goto school;
    /* ... */
  }
  int task2() {
```

```
    /* ... */
    if(tolean)
      goto school;
    /* ... */
  }
  school:
    study();
}

 void funt1(int j)
 {
   if(j>10)
       goto label1; // Error: going INTO scope of inner function
   j = 10;
   int funt2(int j)
   {
     label1:
     /* ... */
   }
   funct2(j)
 }
```

A **goto** statement shall not jump from outside the scope of an identifier having a variably modified type to inside the scope of that identifier. A jump within the scope, however, is permitted.

```
goto lab3;          // Error: going INTO scope of VLA
{
      double a[n];
      a[j] = 4.4;
lab3:
      a[j] = 3.3;
      goto lab4;    // OK, going WITHIN scope of VLA
      a[j] = 5.5;
lab4:
      a[j] = 6.6;
}
goto lab4;          // Error: going INTO scope of VLA
```

8.6 Labeled Statements

The syntax for labeled statements is as follows:

```
labeled-statement:
        identifier : statement
        case constant-integral expr : statement
        case string-expr: statement
        default : statement
```

A **case** or **default** label shall appear only in a **switch** statement. Label names shall be unique within a function. Any statement may be preceded by a prefix that declares an identifier as a label name. Labels in themselves do not alter the flow of control. Label names have function scope.

Chapter 9

Pointers

Pointer is defined as a variable which contains the address of another variable or dynamically allocated memory. If we have a pointer variable of type `pointer to int`, it might point to an int variable, or to an element of an array of int type. Pointer is essential for programming in C and Ch. It is also useful for interfacing with hardware.

Pointers in Ch are C compatible. Ch uses pointers explicitly for arrays, structures, functions, classes and simple data types. There are two basic operators for pointer. They are the indirection operator '`*`' and the address operator '`&`'. They are used in the following context.

1. To declare a pointer, add the operator '`*`' in front of its name.
2. To obtain the address of a variable, add the operator '`&`' in front of its name.
3. To obtain the value of a variable, add the operator '`*`' in front of a pointer's name.

Variables of pointer type can be declared similar to variables of other data types. For example,

```
int *p, i;
```

declares `p` as a pointer to int and `i` as an int. The expression `*p` is the type int. We can have a pointer to any variable type. Note that a pointer must be associated to a particular type. There is one exception: a "pointer to void" is used to hold any type of pointer but cannot be dereferenced itself.

The unary operator '`&`' gives the "address of a variable". The expression `&i` means the address of variable `i`. The dereference operator '`*`' gives the "contents of an object pointed to by a pointer". The expression `*p` represents the value stored in the location pointed to by variable `p`. It is different from the multiplication operator and is also different from its use in declaration of variables of pointer type.

Therefore, the programming statement

```
p = &i;
```

will set the pointer `p` to the address of `i`. After that, the equality `*p == i` holds.

9.1 Pointer Arithmetic

As mentioned above, pointers do not have to point to a simple variable of scalar type. They can also point to an element of an array. For example, we can write

```
int *p;
int a[10];
p = &a[3];
```

and we would end up with p pointing at the fourth element of the array a. Note that by default the array index starts at 0, instead of 1. The situation is illustrated below

```
a[0] a[1] a[2] a[3] a[4]  ...  a[9]
               |
               p
```

The pointer p can be used just like the one in the previous section. The expression *p gives what p points to, which in this case is the value of a[3].

Once we have a pointer pointing at an element of an array or dynamic allocated memory, we can perform pointer arithmetic. Given that *p* is a pointer to a[3], we can add 1 to *p*,

```
p + 1
```

In Ch and C, adding one gives a pointer to the next cell. The following code assigns this new pointer to another pointer variable p2.

```
int *p2;
p2 = p + 1;
```

Now the relation of pointers and array becomes

```
a[0] a[1] a[2] a[3] a[4] ... a[9]
               |    |
               p    p2
```

The programming statement

```
*p2 = 4;
```

will set a[4] to 4. We can compute a new pointer value and use it immediately as shown below.

```
*(p + 1) = 5;
```

In this example, we have changed a[4] again, setting it to 5. The parentheses are needed because the unary operator * has higher precedence than the addition operator. If we wrote *p + 1, without the parentheses, we would be fetching the value pointed to by p, and adding 1 to that value.

Besides adding one, any number can be added to or subtracted from a pointer. If p still points to a[3], then

```
*(p + 5) = 7;
```

sets a[8] to 7, and

```
*(p - 2) = 4;
```

sets a[1] to 4.

The increment operator '++' and decrement operator '--' make it easy to do two things at once. The expression like *p++ accesses what p points to, while simultaneously incrementing p so that it points to the next element. The preincrement form *++p increments p, then accesses what it points to. Note that (*p)++ increments what p points to.

Pointer to characters is commonly used. A string can be defined in Ch as shown below.

```
char * str1;
string_t str2;
```

The following example illustrates how pointers can be used to handle strings.

```
char dest[100], src[100];
char *dp = dest, *sp = src;

strcpy(src, "abcd");
/* copy src to dest */
while(*sp != '\0')
    *dp++ = *sp++;
*dp = '\0';
```

In the above example, pointers to char are used to copy a string in array `src`.

When pointer arithmetic is performed, make sure it is within the valid range. For example, if the array a has 10 elements, you can't access a[10] or a[-1], because by default the valid subscript for a 10-element array ranges from 0 to 9.

Besides through an explicit pointer, the elements of an array can be accessed through the array name itself. It is because the array's name is a pointer to the first element in the array in C and Ch. Therefore, the statement

```
p = a;
```

is equivalent to

```
p = &a[0];
```

Both of these two statements make the pointer `p` point to the first element of array `a`. Furthermore, the third element of array `a` can be accessed as follows

```
int aa2 = *(a+2);  // obtain the value of the third element
*(a+2) = 5;        // assign 5 to the third element of a
```

9.2 Dynamic Allocation of Memory

A problem using fixed-size array is that either it is too small to handle special cases, or it is too big and the resource is wasted. Without using variable length arrays, this problem can be solved by dynamically allocated memory using the standard functions **malloc()**, **calloc()**, or **realloc()** as well as the operator **new**.

The order and contiguity of storage allocated by successive calls to the functions **calloc(), malloc()**, and **realloc()** is unspecified. The pointer returned if the allocation succeeds is suitably aligned so that it may be assigned to a pointer to any type of object and then used to access such an object or an array of such objects in the space allocated (until the space is explicitly freed or reallocated). Each such allocation shall yield a pointer to an object disjoint from any other object. The pointer returned points to the start (lowest byte address) of the allocated space. If the space cannot be allocated, a null pointer is returned. If the size of the space requested is zero, the behavior is platform-dependent: either a null pointer is returned, or the behavior is as if the size were some nonzero value, except that the returned pointer shall not be used to access an object. The value of a pointer that refers to freed space is indeterminate.

As an example, we can allocate a piece of memory and copy a string into it with the function **strcpy()** as shown below.

```
char *str = "abcd", *copy;
...
```

```
/* +1 for NULL terminator */
copy = (char *)malloc(strlen(str) + 1);
strcpy(copy, str);
```

Remember that all strings have a terminating '`\0`' character which is not included by **strlen()**. The number of bytes for string `str` is `strlen(str)+1`, not `strlen(str)`.

Ch has a **sizeof** operator which computes the size, in bytes, of a variable or type. It's useful to allocate memory for variables whose sizes are unknown to the users. To allocate space for 100 ints, we could use

```
int *p = (int *)malloc(100 * sizeof(int));
```

Obviously, no computer has an infinite amount of memory available. If we call malloc(1000000000), or if we call malloc(10) 100,000,000 times, the system is probably going to run out of memory. When the function **malloc()** is unable to allocate the requested memory, it returns a **NULL** pointer. Therefore, whenever you call **malloc()**, it is important to check the returned value before using it. A call to function **malloc()** with an error check is shown below.

```
int *p = (int *)malloc(100 * sizeof(int));
if(p == NULL)
{
    printf("out of memory\n");
    exit(1);
}
```

If function **malloc()** returns **NULL**, the code should return to its caller, or exit from the program entirely after printing the error message. It cannot proceed with the code that would have used the memory pointed to by `p`. A good application example of dynamic allocation of memory is to create a linked list which will be described in Chapter 18.

Unlike automatic-duration variables, dynamically allocated memory does not automatically disappear when a function returns. Just as you can use function **malloc()** to control exactly when and how much memory you allocate, you can also control exactly when you deallocate it. In fact, many programs use memory on a transient basis. They allocate some memory, use it for a while, but then reach a point where they don't need that particular piece any more. Because memory is not inexhaustible, it's a good idea to deallocate (that is, release or free) memory you are no longer using.

Dynamically allocated memory is deallocated with the function **free()**. Dynamically allocated memory using operator **new** can be deallocated by operator **delete**, which will be described in Chapter 19. If `p` contains a pointer previously returned by function **malloc()**, you can call function

```
free(p);
```

to release the memory dynamically allocated. After calling `free(p)`, it is most likely the case that `p` still points at the same memory in C. However, `p` will be set to NULL in Ch after it is deallocated. So long as we check to see if `p` is non-NULL before using it again, we won't misuse any memory via the pointer p.

Ch supports functions which return pointers. This is useful for allocating memory within functions. Below is a simple example of a function returning a pointer to int.

```
int *fn1() {
    int *p = (int *)malloc(sizeof(int));
    .....

    *p = 5;
```

```
    ...

    return p;
}
```

The memory which is dynamically allocated by function **malloc()**, inside function `fn1()`, can be freed in the calling function.

Note that the code below is invalid.

```
int *fn2() {
    int k;
    ...

    k = 5;
    ...

    /* return address of k*/
    return &k;
}
```

The function `fn2()` tries to return the address of local variable `k`. When the function `fn2()` returns, the memory for variable `k` will be deallocated automatically.

9.3 Arrays of Pointers

Like C, Ch supports arrays of pointers since pointers are variables themselves, such as

```
int (*p1)[3], a1[2][3], a2[3][3];
p1 = a1;       // p1[i][j]<=>a1[i][j]
...
p1 = a2;       // p1[i][j]<=>a2[i][j]
int *p2[3];    // declares an array of 3 pointers to ints.
```

Arrays of pointers are very useful in some cases. Consider the following code fragment.

```
char m1[7][10] = {"Sunday", "Monday", "Tuesday", "Wednesday",
                  "Thursday", "Friday", "Saturday"};
char *m2[7] =    {"Sunday", "Monday", "Tuesday", "Wednesday",
                  "Thursday", "Friday", "Saturday"};
```

Variable `m1` is a two-dimension array of char whereas `m2` is an array of pointer to char. The memory layout for `m1` and `m2` are shown in Figures 9.1 and 9.2, respectively.

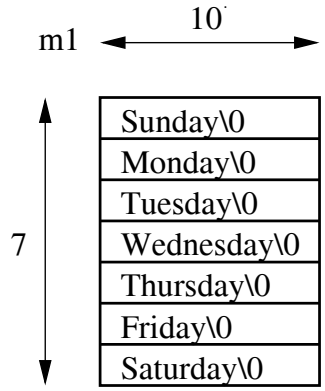


Figure 9.1: 2D array.

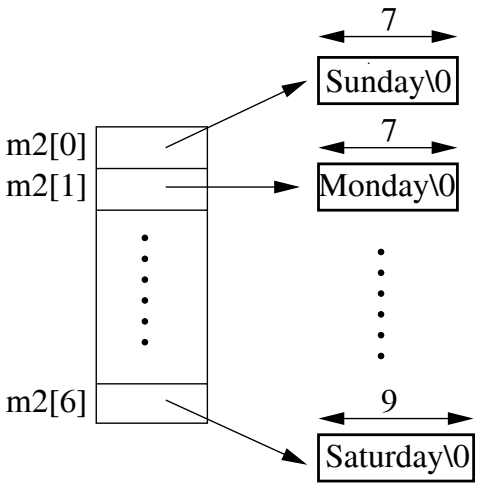


Figure 9.2: Array of pointers.

The advantage of using `m2` is that each pointer can point to arrays with different length rather than the fixed length of 10 bytes. This can be illustrated by the code below.

```
/* three text lines */
char *p[3] = {"ABC", "HIJKL", "EF"};
char *tmp;
...
tmp  = p[1];
p[1] = p[2];
p[2] = tmp;
```

This example demonstrates how an array of pointers can be used to eliminate complicated storage management and overheads of moving lines. In this example, the original strings of different lengths pointed to by pointers `p[0]`, `p[1]` and `p[2]` are shown in Figures 9.3. Without moving and copying characters in these strings, the contents pointed to by pointers `p[1]` and `p[2]` are swapped by swapping values of pointers as shown in Figures 9.4.

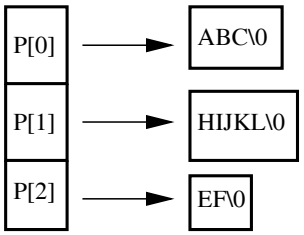


Figure 9.3: Before swapping texts.

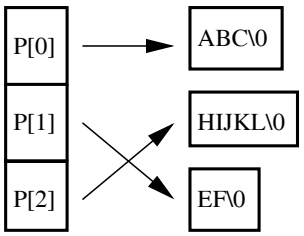


Figure 9.4: After Swapping texts.

9.4 Pointers to Pointers

Because a pointer of different type is a variable itself, Ch can handle a pointer to a pointer of any type.

Consider the following code

```
char ch;            // a character
char *p = &ch;      // a pointer to ch
char **pp = &p;     // a pointer to p
```

It is visualized in Figure 9.5. Here `**pp` refers to memory address of `*p` which refers to the memory address of the variable `ch`.

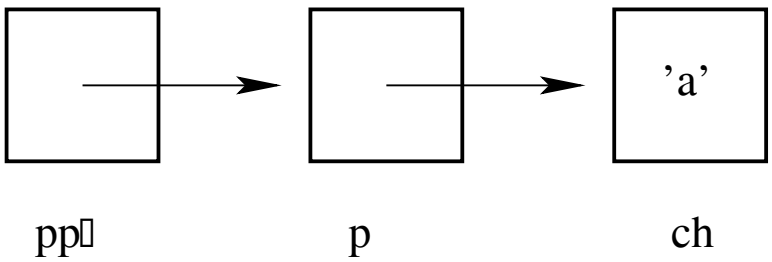


Figure 9.5: Pointer to pointer.

Because **char *** is used to refer to a NULL terminated string in Ch, one common, and convenient, notion is to declare a pointer to pointer to char. For example, the code below

```
char *p = "ab"; // a string
char **pp = &p; // a pointer to p
```

declares `p` as a pointer and `pp` as a pointer to pointer to char as is illustrated in Figure 9.6.

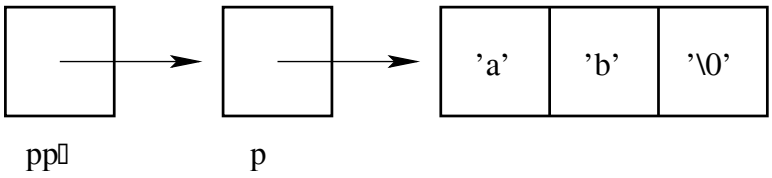


Figure 9.6: Pointer to string.

Furthermore Ch supports several strings being pointed to by a double pointer as shown in the commands below.

```
> char **pp;
> pp = (char**)malloc(3*sizeof(char*));
4006c8d0
> pp[0] = "ab";
ab
> pp[1] = "py";
py
> pp[2] = NULL;
00000000
```

The memory layout for the above code is illustrated by Figure 9.7

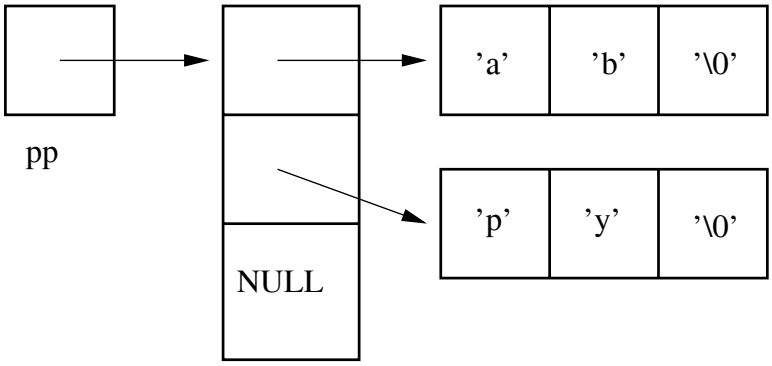


Figure 9.7: Pointer to strings.

We can refer to individual strings by `pp[0]` and `pp[1]`. Semantically this is identical to the declaration of `char *pp[]`. The double pointer is useful for command line argument handling of function **main()**.

Pointers to pointers are also useful for dynamic allocation of memory. For the program below,

```
void fn3(int **p) {
    *p = (int *)malloc(sizeof(int));
    **p = 5;
}
int main() {
    int *p;
    ....

    fn3(&p);
    ....
```

```
}
```

the memory for pointer `p` in calling function **main()** is allocated by function `fn3()`.

Interactive Ch shell is especially useful for understanding how pointer works as shown in the following interactive execution of programming statements with pointer and double pointer.

```
> int i, *p
> p = &i    // assign address of i tp p
1c4228
> *p = 90
90
> printf("i = %d\n", i);
i = 90
> int **p2
> p2 = &p
1c3c38
> printf("**p2 = %d\n", **p2)
**p2 = 90
> i**p   // i * (*p)
8100
>
```

Chapter 10

Functions

A Ch program is generally formed by a set of functions, which subsequently consist of many programming statements. Using functions, a large computing task can be broken into smaller ones; a user can develop application programs based upon what others have done instead starting from scratch. The performance and user friendly interface of functions are critical to a programming language. In Ch, it is guaranteed that all function calls to a function are governed by a prototype, that all the prototypes for the same function are compatible, and that all the prototypes match the function definition even for a program that is divided into many separate files.

All functions, including the main function `main()`, in C are at the same level; functions cannot be defined inside other functions. In other words, there are no internal procedures in C. Ch extends C with nested functions. Functions in Ch not only can be recursive, but also nested, which means that a function can call itself as well as can define other functions inside the function. With nested functions, details of one functional module can be hidden from the other modules that do not need to know about them. Each module can be studied independent of others. Software maintenance is the major cost of a program. People who were not involved in the original design often do the most program maintenance. Nested functions modularize a program, thus clarifying the whole program and easing the pain of making changes to modules written by others. Nested functions are very useful for information hiding and modular programming.

Although adding nested functions to C is a conservative enhancement to the language, addition of any new feature into the standard needs a careful examination of its potential impact on the language as a whole. The new feature should be a natural extension to C, namely, in the so-called spirit of C; it must not break all currently existing codes. With nested functions, local functions can be defined inside other functions. In the spirit of C, functions in Ch can not only be nested, but also recursive. In other words, a function can call itself either directly or indirectly. This is especially important for writing function files. Functions defined inside function files are treated as if they were the system built-in functions in a Ch language environment. This chapter, therefore, first describes how functions are handled in the C standard-conforming manner, then presents new linguistic features of nested functions as they are currently implemented in Ch in the spirit of C.

10.1 Call-by-Value versus Call-by-Reference

In general, arguments can be passed to functions in one of two models: *call-by-value* and *call-by-reference*. In the call-by-value model, the values of the actual parameters are copied into formal parameters local to the called function. When a formal parameter is used as an lvalue (the object that can occur at the left side of an assignment statement), only the local copy of the parameter will be altered. In the call-by-reference method, however, the address of an argument is copied into the formal parameter of a function. Inside the

called function, the address is used to access the actual argument used in the calling function. This means that when the formal parameter is used as an lvalue, the parameter will affect the variable used to call the function. FORTRAN uses the call-by-reference model, whereas the convention in C is call-by-value. If it is desired that the called function alter its actual parameters in the calling function in C, the addresses of the parameters shall be passed explicitly. However, in C++ and Ch, arguments can be passed by reference with reference type described in next Chapter.

10.2 Function Definitions

A function can be defined in the form of

```
return_type function_name(argument declaration)
{
  statements
}
```

Parts of the above function definition may be absent. The `return_type` can be any valid type specifier. The function definition in Ch must begin with a type specifier even for functions that return int.

The traditional function definition, known as K&R C, is also supported in Ch. Although obsolescent, in this notation, parameter identifiers in a function definition are separated by the declaration list.

Declaration statements can be mixed with executable statements. For example, in the code fragment

```
int funct(int i)
{
  i = 3;
  int j;
  return i;
}
```

the variable `j` is declared after the execution statement `i = 3`. The lexical level of parameter variables for arguments of a function is lower than that of local variables defined inside the function. When an identifier is used as both parameter variable of the function and its local variable, the variable will be treated as the argument of the function before the declaration statement that declares it as a local variable. After the declaration statement, the variable becomes the local variable within the function. Therefore, unlike in C, one may use the same identifier as both the argument of the function and its local variable as shown in the following example.

```
int funct(int i, j)
{
  printf("i = %d \n", i);// use i as the argument parameter
  int j=1, i=1;          // i and j are initialized to 1
  j = i +8 +j;           // use i and j as local variables
  return j;              // return the local variable j with 10
}
```

In Ch, variables are guaranteed to be initialized to zeros when they are declared. In the above function `funct()`, the identifier `i` that contains the value of the argument parameter is printed out by the output function **printf**(). The identifier `i` then becomes a local variable after the declaration statement `int j, i`.

The name `j` is used as both an argument and a local variable of the function. The variable `j` is declared as a local variable before it is invoked inside the function. The value passed from the calling function will never be used inside the function. In other words, the local variable hides the argument parameter of the function.

It should be pointed out that, in C, the function `funct()` in the above example has to be defined as `int funct(int i, int j)`. The type declarators for the subsequent arguments can be omitted in Ch if they have the same type as the previous one. However, if the identifier in the argument list is also a typedef name, the type declarator cannot be omitted as shown below.

```
typedef int INT;
int funct(int i, int INT) // int funct(int i, INT) is an error
{
  INT =90;
  /* ... */
}
```

The `return` statement can be used to return a value from the called function to the calling function as in

```
return expression;
```

Any expression can follow `return`, parentheses around the expression are optional. The expression will be converted to the return type of the function if necessary. But if the expression cannot be converted to the return type of the function according to the built-in data type conversion rules implicitly, it is a syntax error. For example,

```
int funct()
{
  int *p;
  return p;      // ERROR: wrong return data type
  return (int)p; // OK: C type conversion
  return int(p); // OK: functional type conversion
}
```

If the return type is not void, a return statement is necessary at the end of the function; otherwise, the default zero will be used as the return value and a warning message will be produced by the system. For example,

```
> int funct(){}
WARNING: missing return statement for function funct() and \
default zero is used
```

In other words, the function will be handled as if a return statement with a return value of zero was present before the closing right brace. Here, the value of zero is used in a general sense. For example, zero of int is 0, zero of float is 0.0, zero of a pointer is NULL, and zero of complex is complexZero. Furthermore, if the return type is not void, the expression following return is necessary; otherwise, the default zero will be used as the return value and a warning message will be produced. For example,

```
int funct()
{
  return;  // WARNING: missing return expression and use default zero
}
```

However, the calling function can freely ignore the returned value. For example,

```
int funct(int i)
{
  return i+1;      // the same as 'return (i+1);'
}
funct(5);          // ignore the return value
```

If a function is defined without returning anything, the return data type should be `void`. It is an error to call the function with the return type of `void` in a context that requires a value. For example,

```
void funct(int i){}
int k;
k = funct(3);         // ERROR: lvalue and rvalue are not compatible
```

If the return type is void, the return statement is optional. But, if there is an expression following return, it is a syntax error. For example,

```
void funct (int i)
{
  if(i == 3)
  {
    printf("i is equal to 3 \n");
    return i;           // ERROR: return int
  }
  else if(i > 3)
  {
    printf("i is not equal to 3 \n");
    return;             // OK
  }
  i = -1;
}
funct(2);
```

If the number of the arguments passed to the called function by the calling function is less than that in the function definition, it is a syntax error. For example,

```
int funct(int i, j){return i}
funct(8)          // ERROR: fewer parameters are passed to funct(),
```

On the other hand, if the number of actual arguments is more than the number of the formal definitions, it is also a syntax error. For example,

```
int funct(int i){return i}
funct(8, 9)       // ERROR: number of arguments is 2, need 1 argument
```

Both system built-in and user-defined functions can be used as arguments of functions in Ch. System built-in functions will be handled polymorphically. Furthermore, a function itself can be used as an argument of the function. For example, in Program 10.1, the arguments of the function call `funct2(abs(-6),`

`funct1(funct1(2)), funct2(1,2,3))` are the system built-in function `abs()`, user-defined function `funct1()` which uses itself as the argument, and the function `funct2()` itself.

```
#include <stdio.h>

int funct1(int j) {
    return 2*j;
}
int funct2(int j1, j2, j3) {
    return j1+j2+j3;
}
int main () {
    int i;
    i = funct2(abs(-6), funct1(funct1(2)), funct2(1,2,3));
    printf("i = %d \n", i);  // output: i = 20
}
```

Program 10.1: Using functions as arguments of a function

10.3 Function Prototypes

The type checking for the return value and arguments of functions in Ch is more rigorous and consistent than in C, which can help users detect some hidden bugs in a program. In C, when a program is divided into many source files, the compiler is not required to check that

- all calls to a function are governed by a prototype,
- all the prototypes for the same function are compatible,
- or all the prototypes match the function definition.

But, Ch can check all these even for a program that is divided into many separate files. In Ch, the data type of an actual argument of the calling function can be different from that of the formal argument of the called function so long as they are compatible. The value of an actual argument will be converted to the data type of its formal definition according to the built-in data conversion rules implicitly at the function interface stage. However, the data types for the same argument in different function prototypes for the same function must be the same even in different files. Likewise, the return types of different function prototypes for the same function must also be the same. For example,

```
int funct1(int i);        // return and argument types are int
int funct1(float f);      // ERROR: change data type of argument
int funct1(void v);       // ERROR: change data type of argument
int funct1(int &i);       // ERROR: change data type of argument
float funct1(int i);      // ERROR: change return type of function
void funct1(int i);       // ERROR: change return type of function
funct1(5);                // Ok
funct1(5.0);              // OK, 5.0 converted to 5
```

```
int funct2(int i)
{
  int funct3(int j);      // return and argument types are int
}
int funct3(int i)
{ }
int funct3(int *p);       // ERROR: change data type of argument
```

There can only be one function definition at a given lexical level in a program. For example,

```
int funct1(int i){};
int funct1(int i){};      // ERROR: redefine funct1()
```

The variable and function names cannot be the same at the same lexical level. For example,

```
int funct2;
int funct2(int i){};      // ERROR: redefine funct2
int funct3;
int funct4(int i)
{
  int funct3(int i);     // ERROR: redefine funct3
}
```

where names `funct2` and `funct3` have been defined as simple variables before they are to be defined as functions.

Parameter names must appear in function definitions, but the parameter names for arguments of function prototypes may not be included although they can help in documenting functions and improving the readability of the program. For example, the following two function prototypes are the same

```
int funct(int);
int funct(int i);
```

Functions that have more than one argument can be prototyped in the same manner. For example

```
int funct1(int, int);
int funct2(int, float);
```

Arguments of pointer types can also be prototyped without parameter names. For example,

```
void funct(int *, float *, complex **);
void funct(int *ip, float *fp, complex **zp){ }
```

If the subsequent arguments have the same basic data type, the type specifier in function prototypes can be omitted after the first argument. For example,

```
void funct1(int *ip, *jp, **ip2);
void funct1(int *, *, **);
void funct2(int, , );
void funct2(int, int, int);
void funct2(int i, j, int k);
```

Parameters with names and parameters without names can be mixed in function prototypes. For example,

```
void funct(int, *, float f)
void funct(int i, *p, float);
```

Functions requiring no argument can be prototyped with the single type specifier `void` or `void` followed by a dummy parameter name. For example,

```
void funct(void);
void funct(void){ }
```

Arguments of arrays and arrays of pointers can also be prototyped without parameter names. For example,

```
void funct(int *[], *[3], [][3], char []);
int *ap[10], *bp[3], a[10][20], ca[3];
funct(ap, bp, a, ca);
void funct(int *ap[], *bp[3], a[][3], char c[]){  }
```

Similarly, pointer to arrays and arrays of assumed-shape in function arguments can be prototyped without parameter names. For example,

```
int a[10][10];
void funct(int (*)[3], [:][:]);
void funct(int (*a)[3], b[:][:]){ }
funct(a, a);
```

References to basic data types and references to pointers can be handled in the same manner. For example,

```
int i, *p1, **p2;
void funct1(int &, &, &*, &**);
funct1(i, i, p1, p2);
void funct1(int &i, &j, &*p1, &**p2){ };
```

If no argument in a function prototype. the argument type compatibility checking will be turned off in Ch. But the return type of the function will still be checked. Both ISO C and K&R C function prototypes can be mixed in a program, but for prototypes in C style, both return type and arguments will be checked. The first C function prototype or the function definition determines the number and data types of the arguments of the function. For example, in Program 10.2, the argument of the first C function prototype in `int funct1(int i)` will be used to check other C function prototypes and function calls prior to the definition of the function `funct1()`. Note that the function `funct5()` requires no argument and its function definition in Program 10.2 is the same as `void funct5(void){ }.` It should be pointed out that the K&R C function prototypes are error-prone. The K&R C function prototype should not be used for writing new code.

Functions can occur in any order in a Ch program. If the function is called before it is defined, int is assumed to be the return type of the function. The number of arguments and their data types will be decided by the first C function prototype or the function definition. In other words, when the function `funct()` has not been defined before it is invoked, it will act as if it had been prototyped by `int funct()`. This can be illustrated by various programming examples in Program 10.3. After the function definition, the number of arguments and their data types in the definition of the function, rather than those in a function prototype, will be checked against actual arguments in subsequent function calls. The function prototypes can be used multiple times so long as they are compatible each time. But, if the return value of a function is not int, a function prototype must be used before the function can be called as shown in Program 10.4.

```
int funct1();              // return type is int, ignore arguments
int funct1(int i);         // argument is an int
int funct1(int i, j);      // ERROR: change number of argument
int funct1(int *p);        // ERROR: change data type of argument
int funct1();              // OK to repeat the same function prototype
int funct1(int i){ }       // function definition

complex funct2(int i){ }   // return type is complex,
                           // argument is an int
int funct2();              // ERROR: change return type of function
int funct2(int i);         // ERROR: change return type of function
complex funct2(int i);     // OK:

int funct3(int i)
{
  int funct4();
  int funct4(int i);       // argument is an int
  int funct4(int i, j);    // ERROR: change number of argument
}
int funct4(int i)
{ }

void funct5();
void funct5(void);
void funct5(void v);
void funct5();             // OK
void funct(int);           // ERROR: change data type of argument
void funct5(){ }           // function definition
```

Program 10.2: K&R and ISO C function prototypes.

```
funct1(3);                 // by default, funct1() return int;
void funct1(int i) { }     // ERROR: change return type of function

int i;
funct3(3);           // by default, funct3() return int;
int funct3(int i) { }// WARNING: missing return statement and default zero is returned

int i;
funct4(3);           // by default, funct4() return int;
int funct4(int i)
{ return; }          // WARNING: missing return expression, use default zero

funct5(8)            // WARNING: fewer parameters are passed to funct(),
                     //          default zeros are used for missing ones
int funct5(int i, j){return i}

funct6(8);
int funct6(int i){return 3}         // OK

funct7(8)
int funct7(int i){}  // WARNING: missing return statement, use default zero

funct8(8)
void funct8(int i){} // ERROR: change return type of function

funct9(8)
int funct9(float f){return (int)f}  // OK

funct10(8)                          // ERROR: non ptr value passed to ptr
int funct10(int *p){return 3}       // OK
```

Program 10.3: Sample programs using default prototypes.

```
void funct1(int i) { }
int funct(int i)
{
  void funct1(int i);  // redundant prototype OK
  void funct2(int i);  // must have prototype
  int  funct3(int i);  // can be omitted by default
  funct1(i);
  funct2(i);
  funct3(i);
}
void funct2(int i) { }
int funct3(int i) { }
```

Program 10.4: Examples where prototypes are optional or required.

```
int main() {       // level 1 for main
  funct(1);
}
int funct(int j)   // level 2 for funct and level 2 for j
{
  if(j <=  3)
  {
    printf ("recursively call funct() j = %d \n", j);
    j++;
    j = funct(j);
  }
  else
    printf ("exit funct() j = %d \n", j);
  return j;
}
```

Program 10.5: Direct recursive functions

10.4 Recursive Functions

Functions can be used recursively. In other words, a function can call itself directly as shown in Program 10.5. The output of Program 10.5 is as follows:
recursively call funct() j = 1
recursively call funct() j = 2
recursively call funct() j = 3
exit funct() j = 4
When a function calls itself recursively, each function call will have a new set of local variables. Inside a recursive function, conditional statements, such as `if-else`, are normally needed in order to exit the function and return the control flow of a program to the calling function. A function may also call itself indirectly as shown in Program 10.6. In Program 10.6, functions `funct1()` and `funct2()` call themselves indirectly, the function `funct2()` calls the function `funct2()` whereas `funct2()` calls `funct1()`. The output of Program 10.6 is displayed in Figure 10.1.

10.5 Nested Functions

In the spirit of C, the function definition with nested functions in Ch takes the form of

```
return_type function_name(argument declaration)
{
  statements
  function_definitions
}
```

or

```
return_type function_name(argument declaration)
```

```
int main() {
  funct1(1);
}
int funct1(int i)
{
  i = funct2(i);
  printf ("exit funct1() i = %d \n", i);
  return i;
}
int funct2(int j)
{
  if(j <=  3)
  {
    printf ("recursively call funct2() j = %d \n", j);
    j++;
    j = funct1(j);
  }
  else
  {
    printf ("exit funct2() j = %d \n", j);
    j++;
  }
  return j;
}
```

Program 10.6: Indirect recursive functions

```
  recursively call funct2() j = 1
  recursively call funct2() j = 2
  recursively call funct2() j = 3
  exit funct2() j = 4
  exit funct1() i = 5
  exit funct1() i = 5
  exit funct1() i = 5
  exit funct1() i = 5
```

Figure 10.1: Output of Program 10.6.

```
int i;                // level 1
void funct1(int i)    // level 2 for i, level 1 for funct1
{
  int i;              // level 3
  void funct2(int i) // level 4 for i, level 3 for funct2
  {
     int k;
     k = i;           // use i at level 4
     int i=6;         // level 5
     i = 30;          // use i at level 5
     printf("k = %d \n", k);
  }
  i = 10+i;           // use i at level 3
  funct2(i);          // use i at level 3
}
i = 5;                // use i at level 1
funct1(i);            // output: 10
```

Program 10.7: Lexical levels in Ch.

```
{
  function_definitions
  statements
}
```

where statements can be any valid Ch statements and local functions can be defined inside other local functions. There is no restriction on the number of function nesting in Ch. In this section, the linguistic features of nested functions will be described.

10.5.1 Scopes and Lexical Levels of Nested Functions

Variable and function names in Ch are associated with their scopes. The *scope* of a variable or function name is a part of the program within which the variable can be used. The scope of the function arguments is the body of the function. The scope of the variables defined inside a function begins right after its declaration and ends at the closing right brace for the function definition. Local variables of the same name in different functions are not related to each other. The same is true for parameters of functions. The scope of a local function is the function within which the local function is defined. The scope of a top level function in Ch is the entire program, but the function may have to be prototyped if it is invoked prior to its definition. Note that, in C, although the scope of all functions is the entire program, consistent function prototypes must be provided. The inconsistency of function prototypes for the same function in different files cannot be detected by C compilers, but it can be detected in Ch. The programs in Ch can be much less error-prone.

The *lexical level* of a variable or function name is the place where it is declared. If we treat the top level of a program as the first lexical level, the arguments of the top level function are at level 2; the local variables declared inside the function are at level 3; the argument parameters in a nested function are at level 4; the local variables defined inside the nested function are at level 5, and so forth. The lexical level i is higher than the lexical level $i + 1$. These different lexical levels of variables in nested functions are illustrated in Program 10.7. In Program 10.7, the function `funct2()` is not visible outside the function `funct1()`. The same name can be used for variables and functions at different lexical levels. The part of a program at a

```
void funct1() {    // level 1
  void funct2() {  // level 2
    void funct3()  // level 3
    { }
  }
  funct3();        // Error: funct3() is at lower level
}
```

Program 10.8: The part of the program at the higher lexical level cannot call functions at the lower level.

```
void funct1()
{
  void funct2(int i)   // level 4
  {
    int k;             // level 5
  }
  void funct22(int i); // level 4
  {
    int k;             // level 5
  }
}
```

Program 10.9: Arguments and local variables at different local functions are unrelated.

.

lower lexical level can access variables and functions at a higher lexical level in nested functions so long as the variables and functions are within their scopes. But the part of a program at a higher lexical level cannot access variables and functions at a lower lexical level. For example, the function `funct3()` defined at the lexical level 3 cannot be invoked at the lexical level 2 in Program 10.8. Arguments and local variables with the same name in different functions at the same lexical level are unrelated. For example, arguments `i` and local variables `k` of different local functions `funct2()` and `funct22()` in Program 10.9 are unrelated. The modification of the variable `k` inside the function `funct2()` will not affect the variable `k` inside the function `funct22()`.

Functions can be defined inside other local functions as shown in Program 10.10 where the function `funct3()` is defined inside the local function `funct2()`. The number of function nesting is not limited in Ch. Not only can the name of variables at different lexical levels be the same, but also the name of functions. If there are several variables with the same name at different lexical levels, the variable with the lowest lexical level will be used within the scope of all variables. It is also true for functions. For example, there are three functions with the same name at three different lexical levels in Program 10.11.

All syntax rules for regular functions, such as initialization of local variables, passing arrays of assumed-shape, and passing arguments by reference, can be applied to nested functions as well. For example, the variable `A1` in the local function `funct2()` is initialized as a 3x3 complex array in Program 10.12. The first dimension of `A1` is determined by the number of rows as the array is initialized at the declaration stage. The float array `F` is passed to the argument `A` of the function `funct2()`; the shape of array `A`, assumed from `F`, is 4x4. The second argument `z` of the function `funct2()` is passed by reference. The output of Program 10.12 printed by the last statement of the function `funct1()` is

```
void funct1(complex z1) {       // definition of the function
  int i;
  complex funct2(complex z2) {  // define local funct before it is used
    complex z;
    complex funct3(complex z3) { // double nested function
      return z3;
    }
    z = funct3(z2);
    return z;
  }
  i = funct2(complex(1,2));
}
```

Program 10.10: Double nested functions in Ch.

```
void funct() {      // level 1
  void funct() {    // level 2
    void funct()    // level 3
    { }
  }
  funct();          // invoke funct() at level 2
}
```

Program 10.11: Different functions with the same name at different lexical levels.

```
void funct1(int i)
{
  void funct2(complex A[:][:], &z)
  {
    complex A1[][3]={
                     {ComplexInf,     ComplexNaN,     Inf},
                     {-Inf,           complex(-3,-1), complex(-7,2)},
                     {complex(-4,-3), complex(6,3),   complex(2,1)}
                    };
   z += A[1][2] + A1[1][2];
  }
  float F[4][4];
  complex z = complex(-3,2);
  F[1][2] = -i;
  funct2(F, z);
  printf("z = %f \n", z);
}
funct1(10);            // output: z = complex(-20.000000,4.000000)
```

Program 10.12: Initialization, arrays of assumed-shape, and references in nested functions in Ch.

```
void funct1()           // level 1
{
  __declspec(local) float funct2(); // local function prototype
  funct2();
  float funct2()        // definition of the local function,
  {
    return 9;
  }
}
```

Program 10.13: The type qualifier `__declspec(local)` qualifies `funct2()` as a local function.

```
void funct1()           // level 1
{
  __declspec(local) float funct2(); // required 'local'
  __declspec(local) int i;          // optional 'local'
  funct2();
  __declspec(local) float funct2()  //optional 'local'
  {
    __declspec(local) int j;        // optional 'local'
    return 9;
  }
}
```

Program 10.14: Optional type qualifier in Ch.

```
z = complex(-20.000000,4.000000)
```

10.5.2 Prototypes of Nested Functions

All local functions in the program examples discussed so far have been defined before they are invoked inside nested functions. The definition of a local function, however, can be placed at any places inside a function. If a local function is invoked prior to its definition, a local function prototype must be used as shown in Program 10.13. In Program 10.13, because the function `funct2()` is used before it is defined, a function prototype is needed. Since it is a local function, the type qualifier `__declspec(local)` is used to distinguish a local function from the top level regular C functions. The type qualifier for local functions can also be placed before the type specifier for declarations of local variables and function definitions inside a function, but it is optional as shown in Program 10.14.

If a function prototype inside a function is not qualified as a local function by the type qualifier `local`, it is assumed as a top level function. This will guarantee that all existing C code will not break when nested functions are added to the language. For example, there are two functions named `funct2()` in Program 10.15. One is defined inside the main routine `main()` and the other is a top level function. Inside the function `funct1()`, the prototype `void funct2(int i)` informs the compiler that the name `funct2` is a function name with the return data type of void and one argument of int. Because there is no type qualifier preceding the function prototype, the function `funct2()` is a top level function by default. Therefore, the subsequent function call of `funct2(i)` will use the function `funct2()` at the top level. After the definition of the local function `funct2()`, the function call of `funct2(100)` inside the

```
int main()                              // level 1
{
  void funct1(int i)              // level 2
  {
    void funct2(int i);
    funct2(i);                    // use funct2() at level 1
  }
  void funct2(int i)              // level 2
  {
    printf("i = %d \n", i+1);
  }

  funct1(100);
  funct2(100);                    // use funct2() at level 2
}
void funct2(int i)                // level 1, top level
{
  printf("i = %d \n", i+5);
}
```

Program 10.15: Function prototypes with no type qualifier are at the top level.

main routine `main()` will use the local function. The output of Program 10.15 is as follows:
i = 105
i = 101

If a function is called before it is defined, it is assumed that the function is a top level function with the return type of int. The number of arguments and their data types will be decided by the first occurrence of the C function prototypes or the function definition. In other words, when the function `funct()` has not been defined at the point where it is called, it will act as if it had been prototyped by `int funct()`, which is C compatible. For example, the function `funct3()` in Program 10.16 is invoked before it is defined or prototyped. By default, the function `funct3()` is a top level function that returns a value of int type.

For deeply nested functions, if a function that is defined neither at the top level nor at the same level is to be invoked, a local function qualifier can be used to prototype the function at the beginning of the function within which the prototyped function is defined. The function prototypes for this purpose are called the *auxiliary function prototype*. In Program 10.17, the local function `funct3()` can be used by the nested functions `funct2()` and `funct4()` due to the auxiliary function prototype of `__declspec(local) void funct3()`. The scopes, lexical levels, and function prototypes in nested functions can be further demonstrated by the four code fragments given in Program 10.18.

10.5.3 Nested Recursive Functions

In Ch, whether a function is defined as a local function or top level function has no significant effect on the memory space and execution speed of a program even in recursive situations. Inside nested functions, functions can call each other recursively so long as scope and lexical rules of function calls are not violated. In Program 10.20, the function `funct1()` calls its local function `funct2()` as well as itself recursively. But, the function `funct2()` only calls itself recursively. The output of Program 10.20 is as follows:

```
void funct1(int i) {           // level 1
  void funct2(complex A[:][:]);// funct2(): default function at level 1
  complex A1[3][3];
  i = funct3()+4;   // funct3(): default function at level 1 return int
  funct2(A1);
}
void funct2(complex A[:][:]) { // level 1
  A[1][2] =70;
}
int funct3() {                  // level 1
  int i=90;
  return i;
}
```

Program 10.16: Functions invoked prior to their definitions and prototypes are at the top level by default.

```
void funct1() {        // level 1
  __declspec(local) void funct3(); // auxiliary function prototype
  void funct2() {      // level 2
    funct3();          // use funct3() at level 2
    void funct4() {
      funct3();        // use funct3() at level 2
    }
    funct3();          // use funct3() at level 2
  }
  void funct3()        // level 2
  { }
}
```

Program 10.17: Using the type qualifier `__declspec(local)` to invoke functions at different lexical levels.

```
/** EXAMPLE 1 **/
void funct1() {      // level 1
  void funct2() {    // level 2
    int i;
    i = funct3();    // use funct3() at level 1
  }
  void funct3()      // level 2
  { }
  funct3();          // use funct3() at level 2
}
int funct3()         // level 1
{ }

/** EXAMPLE 2 **/
void funct1() {      // level 1
  __declspec(local) void funct3();
  void funct2() {    // level 2
    funct3();        // use funct3() at level 2
    void funct3() {  // level 3
      __declspec(local) void funct3();
      funct3();      // use funct3() at level 4
      void funct3()  // level 4
      { }
    }
    funct3();        // use funct3() at level 3
  }
  void funct3()      // level 2
  { }
  funct3();          // use funct3() at level 2
}
```

Program 10.18: Illustrative sample programs for scopes, lexical levels, and prototypes of nested functions.

```
/** EXAMPLE 3 **/
void funct2() {      // level 1
}
void funct1() {      // level 1
  funct2();          // invoke funct2() at level 1
  void funct2() {    // level 2
    void funct3()    // level 3
    { }
  }
}

/** EXAMPLE 4 **/
void funct2()        // level 1
{ }
void funct1() {      // level 1
  __declspec(local) void funct2();
  funct2();          // invoke funct2() at level 2
  void funct2() {    // level 2
    void funct3()    // level 3
    { }
  }
}
```

Program 10.19: Illustrative sample programs for scopes, lexical levels, and prototypes of nested functions (continued).

```
void funct1(int &i)   // level 1
{
  int funct2(int j)   // level 2
  {
    if(j <=  3)
    {
      printf ("recursively call funct2() j = %d \n", j);
      j++;
      j = funct2(j);
    }
    else
    {
      printf ("exit funct2() j = %d \n", j);
      j++;
    }
    return j;
  }
  i = funct2(i);
  printf ("after call funct2() i = %d \n", i);

  if(i < 6)
    funct1(i);
}
funct1(1);
```

Program 10.20: Direct recursive functions.

```
int funct1(int i)     // level 1
{
  int funct2(int j)   // level 2
  {
    if(j <=  3)
    {
      printf ("recursively call funct2() j = %d \n", j);
      j++;
      j = funct1(j);
    }
    else
    {
      printf ("exit funct2() j = %d \n", j);
      j++;
    }
    return j;
  }
  i = funct2(i);
  printf ("after call funct2() i = %d \n", i);

  if(i < 6)
    i = funct2(i);
  return i;
}
funct1(1);
```

Program 10.21: Indirect recursive functions.

recursively call funct2() j = 1
recursively call funct2() j = 2
recursively call funct2() j = 3
exit funct2() j = 4
after call funct2() i = 5
exit funct2() j = 5
after call funct2() i = 6

In Program 10.21, the function `funct1()` calls its local function `funct2()` and the local function `funct2()` calls its upper level function `funct1()`. The output of Program 10.21 is as follows:
recursively call funct2() j = 1
recursively call funct2() j = 2
recursively call funct2() j = 3
exit funct2() j = 4
after call funct2() i = 5
exit funct2() j = 5
after call funct2() i = 6
after call funct2() i = 6

```
int funct1(int i)     // level 1
{
  int funct2(int j)   // level 2
  {
    if(j <=  3)
    {
      printf ("recursively call funct2() j = %d \n", j);
      j++;
      j = funct3(j);
    }
    else
    {
      printf ("exit funct2() j = %d \n", j);
      j++;
    }
    return j;
  }
  i = funct2(i);
  printf ("after call funct2() i = %d \n", i);
  return i;
}
funct1(1);
int funct3(int i)
{
  i = funct1(i);
  return i;
}
```

Program 10.22: Indirect recursive functions via a top level function.

after call funct2() i = 6

In Programs 10.20 and 10.21, the recursive function calls are restricted within the nested functions only. In Ch, any nested functions can call top level functions. The indirect recursive function calls with top level functions can be illustrated by Program 10.22 where the function `funct1()` calls the local function `funct2()`. The local function `funct2()` calls the top level function `funct3()` which calls the function `funct1()`. Therefore, functions `funct1()`, `funct2()`, and `funct3()` form a closed loop. One programming alternative for Program 10.22 is to handle both functions `funct1()` and `funct2()` at the same lexical level so that the function `funct3()` can be removed as shown in Program 10.23. The following output from Program 10.23 is the same as that from Program 10.22:
recursively call funct2() j = 1
recursively call funct2() j = 2
recursively call funct2() j = 3
exit funct2() j = 4
after call funct2() i = 5
after call funct2() i = 5
after call funct2() i = 5

```
int main() {
  __declspec(local) int funct2();
  int funct1(int i)
  {
    i = funct2(i);
    printf ("after call funct2() i = %d \n", i);
    return i;
  }
  funct1(1);

  int funct2(int j)
  {
    if(j <=  3)
    {
      printf ("recursively call funct2() j = %d \n", j);
      j++;
      j = funct1(j);
    }
    else
    {
      printf ("exit funct2() j = %d \n", j);
      j++;
    }
    return j;
  }
}
```

Program 10.23: Indirect recursive functions at the same lexical level

after call funct2() i = 5

10.6 Using Pointers to Pass Arguments of Function by Reference

When Ch passes arguments to functions it passes them by value. However, in many cases we may want to alter the passed argument in the function. Assume a sorting routine try to exchange two out-of-order elements `a` and `b` with the function `swap()`, the following code will not work.

```
swap(a, b);
```

where the swap function is defined as

```
void swap(int x, int y) { // doesn't work as expected
    int temp;

     temp = x;
     x = y;
     y = temp;
}
```

Because of call by value, `swap()` can't affect the arguments `a` and `b` in the calling function. It only swaps `x` and `y`, which are copies of `a` and `b` respectively, inside function `swap()`.

Pointers can be used to pass the addresses of the variables to the functions and access variables through their addresses indirectly. Using pointers explicitly, the function call in the program becomes

```
swap(&a, &b)
```

As mentioned above, the operator '`&`' gives the address of a variable, expression `&a` is a pointer to `a`. In this case, the function `swap()` should use the addresses rather than the copies of values `a` and `b`.

```
void swap(int *pa, int *pb) {
    int temp;

    temp = *pa; // contents of pointer
    *pa = *pb;
    *pb = temp;
}
```

In the function definition for `swap()`, the parameters are declared as pointers `pa` and `pb`, and the variables `a` and `b` in the calling function are accessed indirectly through pointers `pa` and `pb` .

10.7 Variable Number Arguments in Functions

Ch allows a variable number of arguments to be passed to a function. In some application, numbers of arguments passed to a function are unknown in advance and could be different for different cases. With this feature, one function could handle argument lists with different lengths for different cases. A typical function which takes a variable number of arguments is defined as follows:

Table 10.1: Macros defined in header file **stdarg.h** for handling variable argument list.

Macro	Description
VA_NOARG	Second argument for **va_start**(), if no argument is passed to function
CH_UNDEFINETYPE	not an array.
CH_CARRAYTYPE	C array.
CH_CARRAYPTRTYPE	pointer to C array.
CH_CARRAYVLATYPE	C VLA array.
CH_CHARRAYTYPE	Ch array.
CH_CHARRAYPTRTYPE	pointer to Ch array.
CH_CHARRAYVLATYPE	Ch VLA array.
va_arg	Expands to an expression that has the specified type and the value of the next argument in the calling function.
va_arraytype	Determine if the next argument is an array.
va_arraydim	Obtain the array dimension of the variable argument.
va_arrayextent	Obtain the number of elements in the array of variable argument.
va_arraynum	Obtain the number of elements in the array of variable argument.
va_copy	Makes a copy of the **va_list**.
va_count	Obtain the number of variable arguments.
va_datatype	Obtain the data type of variable argument.
va_end	Facilitates a normal return from the function.
va_start	Initializes *ap* for subsequent use by other macros.
va_tagname	Obtain the tag name of struct/class/union type of a variable argument.

```
#include <stdarg.h>
type1 funcname (arg_list, type2 paramN, ...) {
  va_list ap;
  type3 v;               // first unnamed argument
  va_start(ap, paramN);  // initialize the list
  v = va_arg(ap, type3); // get 1st unnamed argument from the list
  ...                    // get the rest of the list
  va_end(ap);            // clean up the argument list
  ...
}
```

where `arg_list` is the argument list of the named argument, `paramN` is the last named argument and `v` is the first unnamed argument of type `type3`. The data type **ChType_t** is defined in the header file **stdarg.h** also. The standard header file **stdarg.h** also contains a set of macro definitions which define how to deal with an argument list. Some of these macros for array types and functions are listed in Table 10.1.
Besides these macros, the type **va_list** is also defined in header file **stdarg.h**. It is used to declare an object that can hold information of the argument list and refer to each argument in turn. This object is referred to as `ap` according to the Ch notational convention. Macros **VA_NOARG**, **va_count**, **va_datatype**, **va_arraydim**, **va_arrayextent**, **va_arraynum**, **va_arraytype**, and **va_tagname** are usefull for implementation of polymorphic functions. Depending on the array type of its argument, function **va_arraytype**() returns a value in one of the macros **CH_UNDEFINETYPE**, **CH_CARRAYTYPE**, **CH_CARRAYPTRTYPE** , **CH_CARRAYVLATYPE** , **CH_CHARRAYTYPE**, **CH_CHARRAYPTRTYPE** , **CH_CHARRAYVLATYPE**

. Appliction these functions will be described in detail in section 19.9.

The macro **va_start** initializes `ap` to point to the first unnamed argument. It shall be called once before `ap` is used. The rightmost named parameter which plays a special role in accessing a variable argument list is designated `paramN` here. It is used by **va_start** to get started. After that, each call of **va_arg**() returns one unnamed argument and steps `ap` to the next one. The macro **va_arg** takes a type name as an argument to determine what type to return and where the next unnamed argument to get is. The data type can be a simple data type, such as **int**, pointer, or an aggregate data type, such as class, computational array. Finally, after all of the arguments have been read and before returning from the function, macro **va_end** must be called to clean up the argument list. For example, function `f1()` in Program 10.24 takes a variable number of arguments. The number of arguments, which is specified by the last named argument, `arg_num`, can range from 1 to 6. The output of Program 10.24 is shown in Program 10.25.

In C, functions which take variable-length argument lists must have at least one named parameter prior to the variable parameter list. In Ch, if there is no named argument prior to the variable parameter list, macro `VA_NOARG` is used by **va_start** to get started. For example, function `f2()` in Program 10.26 takes no named argument. In this case, `VA_NOARG` can be used by **va_start**. The number of arguments passed to the function can be obtained by macro **va_count**. The output of Program 10.26 is shown in Program 10.27. Ini conjunction with other features, this is useful for function polymorphism.

As an object of **va_list**, `ap` can be copied by macro **va_copy** or passed as arguments to functions. In Program 10.28, the object of **va_list** `ap2` is a copy of `ap`. The object `ap2` has the same state as `ap` when it is copied. It means that `ap2` points to the same argument as `ap` points to when it is copied. In this example, `ap2` starts from the second argument in the variable-length arguments list. Each invocation of **va_copy** macros shall be matched by a corresponding invocation of the **va_end** macro. Function `funct2()` takes an argument of type **va_list**. In Program 10.28, `ap` is passed to `funct2()` as an argument. The output of Program 10.28 is shown in Program 10.29.

Using variable number arguments, arrays of different data types can be passed to the same argument of a function. As an example, the source code for function **lindata()** with the function prototype

```
int lindata(double first, double last, ... /* type a[:]...[:] */);
```

defined in header file **numeric.h** is listed in Program 10.30. This function generates linearly spaced data with initial and final values specified by input arguments `first` and `last`, respectively. Function **lindata()** calls the **va_arraynum()** function to determine the number of elements of the passed array `a`. It then uses this information to generate a linearly space data set. The result is finally copied into array `a` in the third passed argument using function **arraycopy**(). The total number of data points generated is passed as the return value.

Function **arraycopy**() defined in header file **stdarg.h** has the prototype of

```
int arraycpoy(void *des, ChType_t destype,
              void *src, ChType_t srctype, int n);
```

It can be used to pass results of arrays of different data types from a called function to the calling function using a variable argument list. In Program 10.31, array `a` of int type and computational array `b` of double type in the main() function are assigned with linear-spaced values using the function `lindata()` and passed back to the calling function as the third argument. The output from Program 10.31 is shown in Figure 10.2.

```
#include<stdarg.h>
struct tag {int i; float j;};

void f1(int arg_num, ...) {
  va_list ap;
  int i;
  char *str;
  struct tag s;
  int *a;
  int a1;

  va_start(ap, arg_num);

  if (arg_num <= 1)
    return;
  if (arg_num >= 2) {
    i = va_arg(ap, int);
    printf("\nthe 2nd argument is %d\n", i);
  }
  if (arg_num >= 3) {
    str = va_arg(ap, char *);
    printf("the 3rd argument is %s\n", str);
  }
  if (arg_num >= 4) {
    s = va_arg(ap, struct tag);
    printf("the 4th argument s.i is %d, s.j is %f\n", s.i, s.j);
  }
  if (arg_num >= 5) {
    a = va_arg(ap, int *);
    printf("the 5th argument a is %d, %d, %d\n", a[0], a[1], a[2]);
  }
  if (arg_num >= 6) {
    a1 = va_arg(ap, int);
    printf("the 6th argument a1 is %d\n", a1);
  }

  va_end(ap);
  return;
}
int main(){
  struct tag s = {1, 2.0};
  int a[] = {1, 2, 3};
  int arg_num = 3;
  f1(arg_num, 3, "abc");
  arg_num = 6;
  f1(arg_num, 6, "def", s, a, a[1]);

  return 0;
}
```

Program 10.24: Variable-length argument lists.

```
the 2nd argument is 3
the 3rd argument is abc

the 2nd argument is 6
the 3rd argument is def
the 4th argument s.i is 1, s.j is 2.000000
the 5th argument a is 1, 2, 3
the 6th argument a1 is 2
```

Program 10.25: Output of Program 10.24.

```
#include<stdarg.h>
#include<stdio.h>

void f2(...) {
  va_list ap;
  int vacount;
  int i, num = 0;

  va_start(ap, VA_NOARG);
  vacount = va_count(ap);
  printf("vacount = %d\n", vacount);

  while(num++, vacount--) {
    i = va_arg(ap, int);
    printf("argument %d = %d, ", num, i);
  }
  printf("\n\n");
  va_end(ap);
  return;
}

int main(){
  f2(1);
  f2(1, 2, 3);
  f2(1, 2, 3, 4, 5);

  return 0;
}
```

Program 10.26: No named argument in argument lists.

```
vacount = 1
argument 1 = 1,

vacount = 3
argument 1 = 1, argument 2 = 2, argument 3 = 3,

vacount = 5
argument 1 = 1, argument 2 = 2, argument 3 = 3, argument 4 = 4, argument 5 = 5,
```

Program 10.27: Output of Program 10.26.

```
#include <stdarg.h>

int  funct2(int num, va_list ap) {
  int args;
  while(num--) {
    args = va_arg(ap, int);
    printf("args in funct2() is %d\n", args);
  }
}

void funct1(int arg_num, ...) {
  va_list ap, ap2;
  int args;
  int num;

  va_start(ap, arg_num);
  printf("print with ap\n");
  args= va_arg(ap, int); // ap points to the next
  printf("args in funct1 is %d\n", args);
  va_copy(ap2, ap);      // ap2 starts from the second argument

  num = arg_num - 1;
  while(num--) {
    args= va_arg(ap, int);
    printf("args in funct1 is %d\n", args);
  }
  va_end(ap);

  printf("\nprint with ap2\n");
  num = arg_num - 1;
  while(num--) {
    args= va_arg(ap2, int);
    printf("args in funct1 is %d\n", args);
  }
  va_end(ap2); // for va_copy()

  /* pass ap as argument to functions */
  printf("\npass ap to another function\n");
  va_start(ap, arg_num);  // restart
  funct2(arg_num, ap);
  va_end(ap);
}

int main(){
  int arg_num = 3;
  funct1(arg_num, 1, 2, 3);
}
```

Program 10.28: ap is copied and passed as arguments.

```
print with ap
args in funct1 is 1
args in funct1 is 2
args in funct1 is 3

print with ap2
args in funct1 is 2
args in funct1 is 3

pass ap to another function
args in funct2() is 1
args in funct2() is 2
args in funct2() is 3
```

Program 10.29: Output of Program 10.28.

```
/* File: lindata.chf */
#include <stdarg.h>
#include <stdio.h>
int lindata(double first, double last, ...){
    va_list ap;
    int i, n;
    ChType_t dtype;
    double step;
    void *vptr;

    va_start(ap, last);
    if(!va_arraytype(ap)) {
      fprintf(stderr, "Error: 3rd argument of %s() is not array\n", __func__);
      return -1;
    }

    n = va_arraynum(ap);
    double a[n];
    step = (last - first)/(n-1);
    for(i=0; i<n; i++) {
       a[i]=first+i*step;
    }

    dtype = va_datatype(ap);
    vptr = va_arg(ap, void*);
    arraycopy(vptr, dtype,  a, CH_DOUBLETYPE, n);
    // or arraycopy(vptr, dtype,  a, elementtype(double), n);
    return n;
}
```

Program 10.30: The source code for function **lindata()**.

```
#include <numeric.h>

int main () {
    int i, a[6], *p;
    array double b[6];

    lindata(2, 12, a);
    printf("a = ");
    for(i=0; i<6; i++) {
      printf("%d ", a[i]);
    }
    p = &a[0];
    lindata(20, 120, p, 6);
    printf("\na = ");
    for(i=0; i<6; i++) {
      printf("%d ", a[i]);
    }
    printf("\nb = ");
    lindata(2, 12, b);
    printf("%g", b);
}
```

Program 10.31: Use function **arraycopy**() to copy an array passed as an argument in function **lindata()**.

```
a = 2 4 6 8 10 12
a = 20 40 60 80 100 120
b = 2 4 6 8 10 12
```

Figure 10.2: Output of Program 10.31.

10.8 Pointer to Functions

In Ch, a pointer to function can be defined. Each function contains programming statements which are located in memory. A function pointer is a variable containing the address of the function. A function's address is the entry pointer of the function. So, a function pointer can be used to call a function. Furthermore, a function pointer can be assigned, placed in arrays, passed to functions, returned by functions, and so on. The declarations of function pointers are shown below.

```
void (*f1) (void);
int  (*f2) ();
int  (*f3) (float f);
typedef int (*PF)(int i);
PF f4;
```

where `f1` is declared as a pointer to function which has no return value or arguments; `f2` is declared as a pointer to function which returns an integer with or without arguments; `f3` is declared as a pointer to function which returns an integer and takes an argument of **float** type. Like other data types, pointer to function can be defined as a user-defined data type. Data type `PF` is typedefed as pointer to function which returns an integer and takes an argument of **int**. Therefore, `f4` is a variable of type `PF`, i.e. a pointer to function. Program 10.32 illustrates how a pointer to function is used. `fun()` is a regular function which

```
int fun (float f) {
  printf("f = %f\n", f);
  return 0;
}

int main() {
  int (*pf)(float f);

  fun(10);

  pf = fun; // no & before fun
  pf(20);   // call function fun by calling pf

  return 0;
}

/* execution and output
f = 10.000000
f = 20.000000
*/
```

Program 10.32: Use pointer to function.

has the same prototype as the function pointed to by `pf`. After the declaration and assignment of `pf`, the function `fun` can be called by using `pf`. The execution and output of Program 10.32 is attached at the end of the program.

Note that, like an array name, a function name stands for the address of a function, The address operator '`&`' is ignored in both C and Ch. For example, statement

```
pf = &fun;
```

is treated as

```
pf = fun;
```

Two pointers to functions can be compared like other pointers. For example, given

```
int fun (float f) {
    printf("f = %f\n", f);
    return 0;
}
int (*pf1)(float f);
int (*pf2)(float f);

pf1 = fun;
pf2 = fun;
```

the equality `pf1 == pf2` holds.

Pointers to functions can be placed in an array or struct. Array of pointers to functions is an effective way to implement a menu. In Program 10.33, array `options` has three elements of pointers to functions. It can be defined as `int (*options[])()`, an array of pointers to functions which return values of int. In program 10.33, the declaration of array `options` is simplified by a new data type `PF` which is defined as

`typedef int (*PF)()`, a pointer to function which returns a value of int. Function `getChoice()` returns an integer which is used as the subscript of array `options` to call the corresponding function. The interactive execution and output of Program 10.33 is attached at the end of the program. Pointers to functions can be passed as arguments to functions, which is commonly used to set callback functions. Program 10.34 is an example of using pointers to functions as arguments of functions. Function `f2` takes two arguments, one is a function pointer `pf` and the other is an integer. Inside function `f2`, the argument of function pointer is used to call the function which takes an argument of int. In the main function, the name of function `f1()` is passed to `f2()` as a function pointer. The execution and output of Program 10.34 is attached at the end of the program.

Like regular pointers, function pointers not only can be arguments of functions, but also be returned values of functions. Program 10.33 is rewritten in Program 10.35. Instead of the array of pointers to functions, function `processChoice()` is used to process different options. The return data type `PF` of function `processChoice()` is defined as a function pointer type. The function `processChoice()` can be prototyped either by `PF processChoice(int i);` or by `int (*processChoice(int))();` Another way to obtain a function pointer through a function is to pass the address of a function pointer (pointer to pointer to function) to the function. In Program 10.36, function `processChoice2()` takes two arguments, one is a pointer to pointer to function, and the other is an integer which is the option returned by function `getChoice()`. In function `main()`, the address of function pointer `pf` is passed into function `processChoice2()`, then the proper function pointer is assigned into the address of `pf` according to the option `i`. After calling function `processChoice2()`, through pointer to function `pf`, one of functions `opt1()`, `opt2()` and `opt3()` is called.

Pointers to nested functions are treated the same as pointers to regular functions as shown in Program 10.37, where `fp` is a pointer to the nested function `func()`. The output from executing Program 10.37 is attached at the end of the program.

Pointers to functions can be used for registering callback functions. In Ch, when a local function is registered as a callback function, it can only use local variables, arguments of the local function, or global variables, but no intermediate variable in the enclosing block of nesting function is allowed.

10.9 Communication between Functions

Because of nested functions, more options are available for communication between functions in Ch than in C. Methods for communication between functions in Ch can be summarized as follows.

Functions in Ch can communicate through return values, arguments, and variables at higher lexical levels. The input to a function can be obtained from its arguments or using the variables at higher lexical levels. The output of a function can be a return value, its arguments, and variables at higher lexical levels. In order to pass results back to the calling function from the called function, one can use pointers for pass-by-value or use references for pass-by-reference. If a function is used as an operand in expressions, the result from the function should be implemented as a return value. If a large number of variables must be shared among different functions, variables at higher lexical levels are more convenient than long argument lists. Programs written using nested functions in Ch tend to be modular. For better readability, a function shall not be defined across multiple files, hence, local variables inside a function are not visible outside the file within which the function is defined. Variables at higher lexical levels are useful for communication between local functions, especially if local functions must share some data yet neither calls the other. To avoid too many data connections between functions, a function that is self-containing should communicate with other functions with its arguments and return value.

```
#include <stdio.h>
#include <stdlib.h>

int opt0() {
    printf("to handle option 0\n");
    return 0;
}

int opt1() {
    printf("to handle option 1\n");
    return 0;
}

int opt2() {
    printf("to exit\n");
    exit(0);
}

int getChoice() {
    int i;
    printf("input the choice (0,1,2): ");
    scanf("%d", &i);
    if (i > 2 || i < 0) i = 2;
    return i;
}

typedef int (*PF)();
int main() {
   // or int (*options[])() = {
   PF options[] = {
      opt0,
      opt1,
      opt2,
   };

   do {
       options[getChoice()]();
   }
   while(1);

   return 0;
}

/***** execution and output
input the choice (0,1,2): 0
to handle option 0
input the choice (0,1,2): 1
to handle option 1
input the choice (0,1,2): 2
to exit
******/
```

Program 10.33: Implement a menu using pointers to functions.

```
#include<stdio.h>

int f1(int i) {
    printf("i = %d\n", i);
    return 0;
}

int f2(int (*pf)(), int i) {
 pf(i);
 return 0;
}

int main() {
   f2(f1, 5);
   return 0;
}

/* execution and output
i = 5
*/
```

Program 10.34: Example of passing function pointer as argument to function.

10.10 The `main()` Function and Command-Line Arguments

The main routine `main()` is a special function. Command-line arguments or parameters can be passed to a program through the arguments of the function `main()` in two formats shown below.

```
int main(int argc, char *argv[], char **environ) {
   ...
}
int main(int argc, char *argv[]) {
   ...
}
```

The function `main()` can have up to three arguments. The first argument, conventionally called `argc` for argument count, is the number of the command-line arguments; the second, called `argv` for argument vector, is a pointer to an array of character strings of variable length. Each string contains one argument of the command line. Therefore, the argument `argv` can also be considered as a pointer to pointer to char. Then, the function `main()` can be written alternatively as

```
int main(int argc, char **argv) {
   ...
}
```

The third optional argument is a pointer to the table of environmental variables. When a program is invoked, values for arguments `argc` and `argv` of the function `main()` are passed to the program by the Ch programming environment. Following the C standard, `argv[0]` is the name of the program so that `argc` is at least 1. If `argc` is 1, there are no command-line argument after the program name. In addition, the value of `argv[argc]` is a null pointer. For example, Program 10.38 will echo its command-line arguments on a single line, separated by blanks. Assume that the file name of Program 10.38 is `echo`, Program 10.38 can be executed in Ch command line mode as follows,

```
#include <stdio.h>
#include <stdlib.h>

typedef int (*PF)();

int opt0() {
    printf("to handle option 0\n");
    return 0;
}

int opt1() {
    printf("to handle option 1\n");
    return 0;
}

int opt2() {
    printf("to exit\n");
    exit(0);
}

int getChoice() {
    int i;
    printf("input the choice (0,1,2): ");
    scanf("%d", &i);
    if(i > 2 || i < 0)
       i = 2;
    return i;
}

// or int (*processChoice(int i))() {
PF processChoice(int i) {
    switch(i) {
      case 0:
        return opt0;
      case 1:
        return opt1;
      default:
        return opt2;
    }
}

int main() {
   do {
     // call function returned from processChoice()
     processChoice(getChoice())();
   }
   while(1);
   return 0;
}
```

Program 10.35: Example of returning a function pointer.

```
#include <stdio.h>
#include <stdlib.h>

int opt0() {
    printf("to handle option 0\n");
    return 0;
}

int opt1() {
    printf("to handle option 1\n");
    return 0;
}

int opt2() {
    printf("to exit\n");
    exit(0);
}

int getChoice() {
    int i;
    printf("input the choice (0,1,2): ");
    scanf("%d", &i);
    if(i > 2 || i < 0)
       i = 2;
    return i;
}

void processChoice2(int(**pf)(), int i) {
    switch(i) {
      case 0:
        *pf = opt0;
        break;
      case 1:
        *pf = opt1;
        break;
      default:
        *pf = opt2;
    }
    return;
}

int main() {
   int(*pf)();
   do {
     processChoice2(&pf, getChoice());
     pf();
   }
   while(1);
   return 0;
}
```

Program 10.36: Example of passing address of function pointer as argument to function.

```
int main() {
  int func(int i) {
    printf("i in func1() = %d\n", i);
    return 2*i;
  }
  int j;
  int (*fp)(int);

  fp =  func;
  j = fp(10);
  printf("j in main() = %d\n", j);
}
/* output
i in func1() = 10
j in main() = 20
*/
```

Program 10.37: Pointer to a nested function.

```
int main(int argc, char *argv[])
// or int main(int argc, char **argv)
{
  int i;
  for(i = 0; i < argc; i++)
    printf("%s ", argv[i]);
  /* or */
  // do{
  //   printf("%s ", argv[i]);
  // }while(argv[++i] != NULL);
  printf("\n");
}
```

Program 10.38: Command line arguments in the `main()` routine.

```
> echo testing example -a
echo testing example -a
>
```

where the command line `echo testing example -a` with four arguments is also the output of the program.

One of the common conventions of programs on Unix systems is that the argument beginning with a minus sign '`-`' indicates an option. For example, the **which.ch** program in Ch can take two valid options, `-a` and `-v`. The command `which -a` finds all commands, including environment variables and header files. The command `which -v` sends out search messages if the name is not found. These two options can be used at the same time, for example, `which -a -v` or `which -va`.

Program 10.39 is the code for handling command-line arguments, which is extracted from the program **which.ch**. Here, the variables `a_option` and `v_option` indicate that the options `-a` and `-v` are on or not. Their values are `false` by default. If there is no command-line argument, the program will print out the error message, because the program **which.ch** at least has one argument, i.e. the name to be searched for. The while-loop in this program handles all arguments which begin with the minus sign `-`. If the argument which is pointed to by the pointer `argv` begins with the minus sign, the equality

```
**argv == '-'
```

holds. The statement

```
s = argv[0]+1
```

makes `s` point to the second character of this argument. More information about pointers to pointers is available in section 9.4. If the characters 'a' and 'v' are found in these arguments, the variables `a_option` and `v_option` are set to true, respectively. If other characters are found, the error messages will be printed out. At the end of Program 10.39, options and the remaining command-line arguments are printed out. Assume that the file name of Program 10.39 is `commandline.ch`, the results from executing Program 10.39 with different options are shown below.

```
> commandline.ch -a -v arg1
option -a is on
option -v is on
arg1
> commandline.ch -av arg1
option -a is on
option -v is on
arg1
> commandline.ch -v arg1 arg2
option -v is on
arg1
arg2
```

The function function **main()** can also be used with three arguments. The third optional argument is a pointer to the table of environmental variables. The program below can be used to print out all environment variables and their corresponding values.

```
#include <stdio.h>
int main(int argc, char *argcv[], char **environ) {
    int i;
```

```
#include <stdio.h>
#include <stdbool.h>

int main(int argc, char **argv) {
  char *s;
  int a_option = false; // default, no -a option
  int v_option = false; // default, no -v option

  if(argc == 1){        // no argument
    fprintf(stderr, "Usage: which [-av] names \n");
    exit(1);
  }

  argc--; argv++;       // for every argument beginning with -
  while(argc > 0 && **argv == '-')
  {
      /* empty space is not valid option */
      for(s = argv[0]+1; *s&&*s!=' '; s++) { // for -av
        switch(*s)
        {
          case 'a':
            a_option = true;      // get all possible matches
            break;
          case 'v':
            v_option = true;      // print message
            break;
          default:
            fprintf(stderr,"Warning: invalid option %c\n", *s);
            fprintf(stderr, "Usage: which [-av] names \n");
            break;
        }
      }
      argc--; argv++;
  }

  if(a_option)
    printf("option -a is on\n");
  if(v_option)
    printf("option -v is on\n");
  while(argc > 0) { // print out the remaining arguments
      printf("%s\n",  *argv);
      argc--; argv++;
  }
  return 0;
}
```

Program 10.39: Program `commandline.ch` for handling command-line arguments.

```
    for(i=0; environ[i] != NULL; i++) {
        printf("environ[%d] = %s\n", i, environ[i]);
    }
}
```

Alternatively, using global variable `environ` defined in the header file `stdlib.h`, the following program can also print out all environment variables and their corresponding values.

```
#include <stdlib.h>
#include <stdio.h>
int main() {
   int i;
   for(i=0; environ[i] != NULL; i++) {
       printf("environ[%d] = %s\n", i, environ[i]);
   }
}
```

10.11 Function Files

A Ch program can be divided into many separate files. Each file consists of many related functions, at the top level, which are accessible to any part of a program. Each top-level function may subsequently contain many local functions in the nested form as described in the previous sections. A file that contains more than one function is usually suffixed with `.ch` to identify itself as part of a Ch program. One can create a function file in a Ch programming environment. A *function file* in Ch is a file that contains only one function definition. The name of a function file ends in `.chf`, such as `qsort.chf`. The names of the function file and function definition inside the function file must be the same. The functions defined using function files are treated as if they were the system built-in functions in a Ch programming environment. For example, if a file named `qsort.chf` contains the program shown in Program 10.40, the function `qsort()` will be treated as a system built-in function, which can be called to sort elements of a one-dimensional array in an increasing order. In Program 10.40, the function `qsort()` is called recursively to sort elements of a one-dimensional array in an increasing order. The function `swap()` is used only by the function `qsort()`, where `swap()` is defined as a local function. Therefore, the function `qsort()` can be used as a stand-alone system function, which is illustrated by Program 10.41. In Program 10.41, the function `qsort()` is called without a function prototype in the main() function so that the function prototype defined inside the function file `qsort.chf` will be invoked. Note that the return type of the function `qsort()` is void. Without function files, the default return type for functions, which are invoked before they are prototyped or defined, is int. The output of Program 10.41 is as follows

a[0] = 1 a[1] = 2 a[2] = 3 a[3] = 4 a[4] = 5 a[5] = 6

In Ch, local functions can be defined inside a function which can be called recursively as shown in Program 10.40. The function in a function file may call other function files and even recursively call itself indirectly. Like system built-in functions that can be replaced by changing keywords, the function defined in a function file can be suppressed in a Ch program. If a function is defined in a program before it is called, the user-defined function will be used in the program. Similarly, if a function is prototyped before it is called, it is a user defined function. If the function is prototyped, the user must define it somewhere, regardless of whether it has been defined in a function file or not. Although many functions can be defined in a function file, it is a good practice to contain only *one* function and many local functions in a function file. For example, if one wants to treat the function `funct()` as a top level system function, it is a bad design to include other functions in the function file `funct.chf` as shown in Program 10.42.

```
/* qsort: sort v[left] .. v[right] into increasing order */
void qsort(int v[], int left, int right) {
  int i, last;
  /* interchange v[i] and v[j] */
  void swap(int v[], int i, int j)  // local function
  {
    int temp;
    temp = v[i];  v[i] = v[j];  v[j] = temp;
  }

  if(left >= right)
    return;
  swap(v, left, (left + right)/2);
  last = left;

  for(i = left+1; i <= right; i++)
    if(v[i] < v[left])
      swap(v, ++last, i);

  swap(v, left, last);
  qsort(v, left, last-1);
  qsort(v, last+1, right);
}
```

Program 10.40: The function file `qsort.chf` for the function `qsort()`.

```
int main() {
  int i, a[] = {2, 6, 5 , 3, 4, 1};

  qsort(a, 0, 5);
  for(i=0; i<=5; i++) {
    printf("a[%d] = %d ", i, a[i]);
  }
  printf("\n");
}
```

Program 10.41: A program using the function file `qsort.chf`.

```
int funct()
{
   void localfunct1() // OK
   {  }
   void localfunct2() // OK
   {  }
}
int anotherfunct()   // bad
{  }
```

Program 10.42: More than one top level function in the function file `funct.chf`.

As described in section 6.4, functions defined in function files cannot be used as initializers for identifiers of static variables at the function or block scope.

10.12 Generic Functions

A generic function is a built-in system function. A list of generic functions in Ch is given in section 2.2. Most generic functions are polymorphic. When a generic function such as **sin()** is explicitly called, the built-in system function is used even if the user has redefined the function. In this case, the user defined function will be ignored. For example, function call of sin(x) uses the built-in system function so that argument x can be any valid data type for function sin().

However, there are no corresponding standard C functions for generic functions **alias()**, **dlrunfun()**, **elementtype()**, **polar()**, **max()**, **min()**, and **transpose()**. The user shall not redefine these generic functions. Execpt for function polar(), when one of these generic functions is redefined, a warning message will be displayed.

When a generic function name is assigned to a pointer to function, the standard C function is used. For example, in the following code fragment with symbol `sin`,

```
#include <math.h>
double func1( double (*fp)(double), double x) {
  return fp(x);
}
int main() {
  double (*fp)(double) = sin;
  fp = sin;
  double val;
  val = fp(10.0);  // same as val = sin(10.0);
  func1(fp, 10);
  func1(sin, 10);
}
```

the standard C function with the prototype of

```
double sin(double);
```

is used. The user can use a generic function name as an identifier of non-function type. For example, names of generic functions max, min, and exp are declared as scalar variables below.

```
double max;
void func2() {
    int min, exp;
    ...
}
```

Generic functions can be used in system startup files chrc, and .chrc in Unix and _chrc in Windows in the user's home directory.

Chapter 11

Reference Type

This chapter presents linguistic features of references as they are currently implemented in Ch. A program written in a procedural computer programming language is generally formed by a set of functions, which subsequently consist of many programming statements. Using functions, a large computing task can be broken into smaller ones, a user can develop application programs based on what others have done instead of starting from scratch. The performance and user-friendly interface of functions are critical to a programming language. The user may not need to know details inside functions that were developed by others. But, to use the functions effectively, the user has to understand how to interface functions through their arguments and return values. In general, arguments can be passed to functions in one of two ways: *call-by-value* and *call-by-reference*. In the call-by-value model, when a function is called, the values of the actual parameters are copied into formal parameters local to the called function. When a formal parameter is used as an lvalue (the object that can occur at the left side of an assignment statement), only the local copy of the parameter will be altered. If the user wants the called function to alter its actual parameters in the calling function, the addresses of the parameters must be passed to the called function explicitly. In the call-by-reference method, however, the address of an argument is copied into the formal parameter of a function. Inside the function, the address is used to access the actual argument used in the calling function. This means that when the formal parameter is used as an lvalue, the parameter will affect the variable used to call the function.

FORTRAN uses the call-by-reference model, whereas C uses the call-by-value. FORTRAN is one of the oldest computer programming languages and it is still the primary language for scientific supercomputing. There are numerous well-crafted FORTRAN programs. When a FORTRAN subroutine or function is ported as a function in C, the formal arguments of the subroutine are generally treated as arguments of pointer type in the function of C. All variables of arguments inside a subroutine then have to be modified accordingly, which may degrade the clarity of the original algorithm and code readability. This is also a point where beginners of C who have prior FORTRAN experience get confused. Ch is designed to be a superset of C, but it encompasses all the programming capabilities of FORTRAN 77. To bridge the gap between FORTRAN and C and to ease the pain of porting FORTRAN code to Ch, many programming features such as complex type and arrays of assumed-shape have been designed and implemented in Ch. References are added to Ch to further simplify the porting of subroutines and functions in FORTRAN to functions in Ch.

Adding references to C is not new. C++ has reference types. The primary use of references in C++ is in specifying operations for user-defined types. The references in Ch not only ease the porting of FORTRAN code to Ch and to make Ch more suitable for scientific programming and for novice users, it is also essential for passing arguments to functions in a safe Ch program where pointers are restricted. References in Ch are designed and implemented in the spirit of C, C++, and FORTRAN. We have extended the linguistic features of references in C++ and FORTRAN for scientific programming. In Ch, both variables of basic data type, and variables of pointer type can be used as references. In addition, variables of different data types can be

passed to arguments of functions by reference. Furthermore, references can be used as arguments and local variables of nested and recursively nested functions.

11.1 References in Statements

A reference in Ch is an alternative name for an object just as in C++. The declaration statement

```
int i, &j = i;
```

indicates that the variable `j` is a reference to `i` of int data type. In other words, `j` is an alias to `i`. If the variable that is declared and the variable that is referenced are the same data type, they can be considered to be references to each other. Therefore, we may also say that `i` is a reference to `j` in the above example. Both variables `i` and `j` share the same memory space inside the system. Once a linkage has been established for two variables of the same type, they can be used interchangeably. For example,

```
int i, &j = i;
i++;              // the same as 'j++'
```

In C++, only simple variables of basic data type can be treated as references. In Ch, not only can simple variables of basic data type, be declared as references, but also variables of pointer type. For example,

```
int i, *p1 = &i, **p2 = &p1;
int &*pp1 = p1, &**pp2 = p2;
```

where `pp1` is a reference to `p1` of pointer to int and `pp2` is a reference to `p2` of pointer to pointer to int.

A reference must be initialized at the declaration stage. Once the reference relation has been established, it cannot be changed. For example, the following code has syntax errors because the variables `j` and `p` of reference are not initialized.

```
int &j;                // ERROR: reference not initialized
int &*p;               // ERROR: reference not initialized
```

More than two variables can refer to the same memory location. For example,

```
int i, &j = i, &k = i, &l = k;
int &m = i;
```

where variables `i, j, k, l` and `m` are referenced to each other. The modification of one variable will affect all other variables.

To avoid the aliasing and to simplify implementation, only simple variables can be referenced to each other at its current implementation of Ch. If the rvalue initialized to a reference is not a simple variable, the reference will be treated as a simple variable and the initialization will be treated as the initialization for the simple variable. For example, all references in the following declaration are effectively treated as simple variables in the system.

```
int a[10];
float f, *fp = &f;
complex z;
int &i = 6;                    // int i = 6;
int &j =6+a[1];                // int j = 6+a[1];
```

```
int &*pp = &i+6;                // int *pp = &i+6;
float &f1 = real(complex(1,2)); // float f1 = real(complex(1,2));
float &f2 = real(z);            // float f2 = real(z);
float &f3 = a[1];               // float f3 = a[1];
float &f4 = *fp;                // float f4 = *fp;
float &*p = &f;                 // float *p = &f;
f = 5;                          // the same as *fp = 5 or *p = 5;
```

`real(z)`, `a[1]` and `*ptr` are lvalues in the above example, but they are not simple variables. Therefore, they cannot be references. Note that the pointer `p` is pointed at the address of the variable `f` in the C conforming manner.

Variables of different data types can also be considered as references so long as their data types are compatible. For example, in the following code

```
int i = 30;
double &d = i;
printf("d = %lf \n", d);   // output: d = 30.000000
```

The variable `d` of double data type is a reference to int of i. Both variables `i` and `d` refer to the same memory space of an int which occupies four bytes. The data type of the variable `d` is double, therefore, the results of **abs**(d) and `d+3` are doubles. When `d` is used in an expression, the value of int will be converted into double implicitly prior to the execution of the operation. In the same token, when `d` is used as an lvalue in an assignment statement, the result of the rvalue will be cast into an int before it is assigned to the memory which has only four bytes. Therefore, if the value is beyond the range for the integer value of `[INT_MIN, INT_MAX]`, the information may be lost because of the implicit data conversion. On the other hand, if a variable of int is a reference to a variable of double, all information, except the fractional part of the double variable, will be preserved. For example,

```
double d = 3.6;
int &i = d;
printf("i = %i \n", i);   // output: i = 3
i = 7;                    // i = 7; d = 7.000000
d = 5.2;                  // i = 5; d = 5.2
```

where both variables `i` and `d` share the same memory space of a double datum which is eight bytes. Variables of incompatible data type cannot become references. For example,

```
int i, *p = &i;
int &*ptr = i;            // data types of ptr and i are incompatible
int &j = p;               // data types of j and p are incompatible
```

The reference linkage can also be applied to variables at different lexical levels. Variables at a lower lexical level can be declared and initialized to refer to a variable defined at a higher lexical level as shown in the following sample code

```
int i = 8;
void funct()
{
  int &j = i, &k =i;         // j = 8; k = 8;
```

```
  printf("j = %d, ", j);     // output: j = 8,
  j =90;
}
funct();                     // get output: j = 8
printf("i = %d \n", i);      // output: i = 90
```

where both variables `j` and `k` share the same memory space with the variable `i`. The output of the above program is as follows:
j = 8, i = 90

11.2 Passing Arguments of Function by References

In C, when a function is called, the actual arguments of the calling function are passed to arguments of the called function by value. The values of the actual parameters are copied into formal parameters local to the called function. When a formal parameter is used as an lvalue, only the local copy of the parameter will be altered. Therefore, the function `swap()` below will not work correctly because `x` and `y` are passed by value.

```
int a = 5, b = 6;
void swap(int x, y)
{
  int temp;
  temp = x; x = y; y = temp;
}
swap(a, b);                    // fails to swap a and b
```

In C, if the user wants the called function to alter its actual parameters in the calling function, the addresses of the parameters must be passed to the called function explicitly. One correct version of the function `swap()` is to use pointers to pass the addresses of variables in the calling function to the called function as shown in the following code.

```
int a = 5, b = 6;
void swap(int *x, *y)
{
  int temp;
  temp = *x; *x = *y; *y = temp;
}
swap(&a, &b);                  // a = 6; b = 5;
```

where the indirection operations are used to change the values of variables in the calling function.

In the call-by-reference method as in FORTRAN, however, the address of an argument is copied into the formal parameter of a function. Inside the function, the address is used to access the actual argument used in the calling function. This means that when the formal parameter is used as an lvalue, the parameter will affect the variable used to call the function. When references in Ch are used as arguments of functions, the functions will be called by reference. The function `swap()` can be implemented using references in Ch as follows:

```
int i =5, *p1 = &i, **p2;
int &*pp1 = p1, &**pp2 = p2;              // pp1 = &i

p2 = malloc(5*sizeof(int));
void funct1(int& *p)
{
  p = malloc(9);
  printf("In funct2() p = %p \n", p);
}
printf("Before funct1() pp1 = %p \n", pp1);
funct1(pp1);
printf("After funct1() pp1 = %p \n", pp1);

void funct2(int& **pp)
{
  pp++;
  printf("In funct2() pp = %p \n", pp);

}
printf("Before funct2() pp2 = %p \n", pp2);
funct2(pp2);
printf("After funct2() pp2 = %p \n", pp2);
```

Program 11.1: References to pointers in Ch.

```
int a = 5, b = 6;
void swap(int &x, &y)
{
  int temp;
  temp = x; x = y; y = temp;
}
swap(a, b);                    // a = 6; b = 5;
```

where no pointer indirection is involved.

In C, if a function needs to change the value of a variable of pointer type through an argument of the function, a pointer to pointer, that is, a double pointer, has to be passed to the function. In Ch, not only simple variables, but also pointers can be passed by reference as shown in Program 11.1. In Program 11.1, the pointer variable `pp1` points at the memory location for the variable `i` before the function `funct1(pp1)` is called. The pointer `pp1` points at the newly allocated memory space of 9 bytes through the function call of `funct1(pp1)`, which is achieved by the formal argument `p` of the function. Similarly, the variable of double pointer `pp2` is passed to the formal argument `pp` of the function `funct2()`. It is incremented by 4 bytes, the space for an int, by the address arithmetic inside the function. The output of Program 11.1 is as follows:

Before funct1() pp1 = 11b578
In funct2() p = 11ea38
After funct1() pp1 = 11ea38

```
void deallocate(void &* ptr)
{
  free(ptr);
  ptr = NULL;
}
void *p;
p = malloc(10);
deallocate(p);  // free memory and reset p to NULL
```

Program 11.2: Function `deallocate()` free the memory and reset the pointer to NULL.

```
int i;
void funct(int &r1, &r2, r3)
{
  r1 = 3; r2++; r3++;
  printf("r1 = %d\n", r1);  // output: r1 = 4
}
funct(i, i, i);
printf("i = %d\n", i);      // output: i = 4
```

Program 11.3: A same variable passed to different references

Before funct2() pp2 = 11e930
In funct2() pp = 11e934
After funct2() pp2 = 11e934

In Ch, the function `free(ptr)` will deallocate the memory pointed at by the pointer `ptr` and reset the pointer `ptr` to NULL. In C, `ptr` is not set to NULL when the memory it points to is deallocated. This dangling memory makes the debugging of a C program very difficult because the problem will not surface until this deallocated memory is claimed again by other parts of the program. Because the function **free**() is implemented as an external function in C, there is no way to set the pointer `ptr` to NULL when it is freed by the function call of `free(ptr)`. But, if references were added to the C, we could implement the function `deallocate(ptr)` which would free the memory, pointed to by the pointer `ptr`, and reset `ptr` to NULL as shown in Program 11.2 in Ch. In Program 11.2, we assume that the function **free**() is a C function which does not set its argument to NULL upon the completion of the function call.

In Ch, the same memory space of a variable can be passed to different references in the arguments of a function. For example, in Program 11.3, both arguments `r1` and `r2` in the function `funct()` use the same memory space of the variable `i` whereas `r3` has its own local memory when the function is called by `funct(i, i, i)`.

In FORTRAN, when an argument of a function is used as an lvalue inside a subroutine, the actual argument in the calling function must be a variable. Unlike in FORTRAN, a reference variable in Ch can be used as an lvalue inside a function even if the actual argument is not an lvalue. If the actual argument of a function, corresponding to a reference in the formal definition, is not a simple variable, the argument will be passed by value. In Program 11.4, references `r1` and `r2` are used as lvalues in the function `funct()`. The function call of `funct(i+8,6)` passes expressions `i+8` and 6 to references `r1` and `r2`, respectively. Note that the reference `k`, instead of the variable `j`, is passed to the reference `r2` in the function call of `funct(i, k)`. In the function call of `funct(abs(-3), funct(1,2))`, the user-defined function

```
int i =50, j=0, &k = j;
int funct(int &r1, &r2)
{
  r1 += 100;
  r2 += r1+2;
  printf("r1 = %d, r2 = %d\n", r1, r2);
  return r1+r2;
}
funct(i+1,3);               // output: r1 = 151, r2 = 156
funct(i,k);                 // output: r1 = 150, r2 = 152
funct(abs(-3), funct(1,2)); // output: r1 = 101, r2 = 105
                            // output: r1 = 103, r2 = 311
printf(i, " ", j, "\n");    // output: 150 152
```

Program 11.4: Using references as lvalues when actual arguments are expressions.

```
int i=5;
void funct(int j)
{
  int &r = j;
  printf("r = %d ", r);
  r++;
  printf("j = %d ", j);
}
funct(i);
printf("i = %d\n", i);
```

Program 11.5: Local variable is a reference to the argument of the function.

uses the system built-in function `abs()` and itself as arguments of the function.

Local variables inside functions can be references to the arguments of functions. For example, in Program 11.5, the local variable `r` is a reference to the integer argument `j` of the function. The output of Program 11.5 is as follows:
r = 5 j = 6 i = 5

11.3 Passing Variables of Different Data Types to the Same Reference

Like initializing a reference with a variable of different data type in a declaration statement, variables of different data types can also be passed to references in the arguments of functions. The interface rules in this case are similar to those described in section 11.1. For example, in Program 11.6, the variables `r1` and `r2` inside the function `funct()` share the same memory spaces of the variables `f` and `i` in the function call of `funct(f, i)`, respectively. The interface of arguments is treated as if the values of variables `f` and `i` were converted to int and complex types first and then copied to variables `r1` and `r2`, respectively. When the flow of program execution exits the function, the results of `r1` and `r2` were then converted to the values of variables `f` and `i`, respectively. The ability to interface different data types in function arguments

```
float f = 90;
int i = -4;
void funct(int &r1, complex & r2)
{
   printf("r1 = %d ", r1++);
   printf("sqrt(r2) = %.3f \n", sqrt(r2--));
}
funct(f, i);  // output: r1 = 90 sqrt(r2) = complex(0.000,2.000)
printf(f," ", i,"\n");  // output: 91.000 -5
```

Program 11.6: Passing actual arguments to references with different data types.

```
float f = 90;
int i = -4;
void funct(int *r1, complex * r2)
{
   printf("r1 = %d ", (*r1)++);
   printf("sqrt(r2) = %.3f \n", sqrt((*r2)--));
}
funct(&f, &i); // output: r1 = 1119092736 sqrt(r2) = complex(NaN,NaN)
printf(f," ", i,"\n");  // output: 90.000 -4
```

Program 11.7: Passing pointers of different type to arguments of pointer type in functions.

is a significant enhancement as there is no way to pass variables whose data types are different from their definitions and get the correct results back in C. Note that the square root of an int datum returns a float in Ch, therefore, **sqrt**(-4) is NaN. Program 11.6 is different from Program 11.7. In Program 11.7, when the address of the variable `f` is passed to the argument `r1` in the function call of `funct(&f,&i)`, nothing but the address is passed. Inside the function, the memory map of a float is used as a memory map for an int, which may not be what the user intended to do. Similarly, the memory location for the variable `i` of int is passed to the pointer to complex of the variable `r2` in the function call.

If the actual argument of a reference of a function call is not a simple variable, the reference inside the function will be treated as a simple variable. If the actual argument of a reference of a function call is not a simple variable and its data type is different from its definition, the result will be converted to the data type of the definition before it is assigned to the variable of the reference. For example,

```
void funct(float &r)
{
   printf("r = %.3f \n", r);
}
funct(90.0);               // output: r = 90.000
funct(90);                 // output: r = 90.000
funct(complex(90,0));      // output: r = 90.000
funct(complex(1,2));       // output: r = NaN
```

Note that the real number converted from a complex number whose imaginary part is not identically zero is NaN.

When a simple variable whose data type is different from the formal definition of a reference argument of a function is passed to the argument of the function, the Ch program will reconcile the value when it is used as an lvalue or operand in expressions. However, if the argument of a function is a reference to pointer data type, the system will treat the object passed into the function as the pointer type declared for the reference. In other words, only the memory for the object is used and its original pointer type in the calling function will be ignored inside the function. For example, in Program 11.8, variables `p1` and `p2` are pointers to int and float, respectively. They have been passed to the function `funct(int &*, float &*)` by the function calls of `funct(p1, p2)` and `funct(p2, p1)`. When the reference of pointer type is passed with different data type, the indirection operation will not be reconciled to deliver the correct result in the function call of `funct(p2, p1)`. The output of Program 11.8 is as follows:
*before: *iptr = 4 *fptr = 5.000000*
*after: *iptr = 90 *fptr = 90.000000*
**p1 = 90 *p2 = 90.000000*
*before: *iptr = 1084227584 *fptr = 0.000000*
*after: *iptr = 90 *fptr = 90.000000*
**p1 = 1119092736 *p2 = 0.000000*
However, a reference to a pointer can be used as a regular pointer inside the function when no indirection operation is involved. For example, pointer `p1` is allocated its memory by the function `getmem(p1, sizeof(int))` and is freed of its memory and reset to NULL by the function call `deallocate(p1)`. Note that the variable `p1` is a pointer to int, but the data type of the corresponding argument of functions `getmem()` and `deallocate()` are pointer to void.

```
void deallocate(void &* ptr)
{
  free(ptr);
  ptr = NULL;
}
void getmem(void &* ptr, int i)
{
  ptr = malloc(i);
}
void funct(int &*iptr, float &*fptr)
{
  printf("before: *iptr = %d *fptr = %f \n", *iptr, *fptr);
  *iptr = 90;
  *fptr = 90;
  printf("after: *iptr = %d *fptr = %f \n", *iptr, *fptr);
}
int *p1;
float *p2;
getmem(p1, sizeof(int)); // p1 = malloc(sizeof(int))
p2 = malloc(sizeof(float));
*p1 = 4; *p2 = 5;
funct(p1, p2);
printf("*p1 = %d *p2 = %f \n", *p1, *p2);
*p1 = 4; *p2 = 5;
funct(p2, p1);
printf("*p1 = %d *p2 = %f \n", *p1, *p2);
deallocate(p1);  // free memory and reset p1 to NULL
deallocate(p2);  // free memory and reset p2 to NULL
```

Program 11.8: Passing pointers of different type to arguments of reference to pointer in functions.

Chapter 12

Scientific Computing Using Generic Mathematical Functions

Ch is a language designed for both scientific and system programming. In this chapter, the scientific computing aspect of the Ch language will be addressed. The ANSI/IEEE 754 standard for binary floating-point arithmetic [11] is a significant milestone on the road to consistent floating-point arithmetic with respect to real numbers. To make the power of the IEEE 754 standard easily available to the programmer, the floating-point numbers of Inf, $-$Inf, and NaN, referred to as *metanumbers*, are introduced in Ch. These metanumbers are transparent to the programmer. Signed zeros $+0.0$ and -0.0 in Ch behave like correctly signed infinitesimal quantities 0_+ and 0_-, whereas symbols Inf and $-$Inf correspond to mathematical infinities ∞ and $-\infty$, respectively. Although the application of symbols such as Inf and NaN can be found in some software packages, their handling of these special numbers has deficiency. For example, one can find ComplexInfinity in the software package Mathematica, and Inf and NaN in MATLAB. In Mathematica, there is no distinction between complex infinity and real infinities, nor between -0.0 and 0.0; therefore, many operations defined in this chapter cannot be achieved in Mathematica. In MATLAB, there is no complex infinity.

A computer language with no mathematical functions is not suitable for scientific computing and many other applications. The C language is a small language; it does not provide mathematical functions internally. The mathematical functions are provided in a standard library of mathematical functions. Because C does not provide mathematical functions internally, like arithmetic operations in K&R C, the returned value from a standard mathematical function is a double floating-point number regardless of the data types of the input arguments. In some of C implementations, if the input arguments are not doubles, the mathematical functions may return erroneous results without warning. Numerically oriented programmers have little tolerance with respect to the implicit conversion of the data type from float to double for arithmetic operations of a computer language. However, they generally accept the strongly typed implementation of mathematical functions. If a different return data type is desired for a mathematical function, a new function with a different name will be needed. For example, the function call of **sin**(1) appears right in C. Indeed, most C programs will execute this operation calmly, but maybe with an erroneous result because the input data type of integer is not what **sin**() function expected. As another example, the function **abs**() in C returns an absolute int number whereas **fabs**() will result in a double number. To get a float absolute value, a new function has to be created. As a result, one has to remember many arcane names for different functions. Ch uses generic functions to resolve this problem.

The external functions of Ch can be created in the same manner as in C. The commonly used mathematical functions are built internally into Ch. The mathematical functions in Ch can handle different data types of the arguments gracefully. The output data type of a function depends on the data types of the input arguments, which is called *polymorphism*. Like arithmetic operators, the commonly used generic mathematical

functions in Ch are polymorphic. For example, for the polymorphic function **abs**(), if the data type of the input argument is int, it will return an int as the absolute value. If the input argument of **abs**() is a float or double, the output will return the same data type of float or double, respectively. For a complex number input, the result of **abs**() is a float with the value of the modulus of the input complex number. Similarly, if the argument data type is lower than or equal to float, **sin**() will return a float result correctly. Function **sin**() can also return double and complex results for double and complex input arguments, respectively. Because I/O functions are also built into Ch itself, different data types are reconciled inside Ch. For example, `printf("%f", x)` in C can print x if x is a float. However, if x is changed to int in a program, the printing statement must also be changed accordingly as `printf("%d", x)`. Therefore, the change of data type declaration of a variable will have to accompany the change of many other parts of the program. The commands **printf**(x) and **printf**(sin(x)) in Ch are flexible and can handle different data types of x; x can be char, int, float, double, or complex.

For portability, all mathematical functions included in the C header `math.h` have been implemented polymorphically in Ch. The returned data type of a function depends on the data types of the input arguments. This will simplify scientific numerical computing significantly. The names of these generic mathematical functions of Ch described in this chapter are based upon the C header file `math.h`. These mathematical functions are C compatible. If the arguments of these functions have the data types of the corresponding C mathematical functions, there is no difference between the C and Ch functions from a user's point of view.

12.1 Generic Mathematical Functions in the Entire Domain

In this section, the generic mathematical functions of Ch will be discussed. The input and output of the functions involving the metanumbers will be highlighted. The results of the mathematical functions involving metanumbers are given in Tables 12.1 to 12.4. In Tables 12.1 to 12.4, unless indicated otherwise, x, x_1, x_2 are real numbers with $0 < x, x_1, x_2 < \infty$; and k is an integral value. The value of `pi` is the finite representation of the irrational number π in floating-point numbers. The returned data of a function is float or double depending on the data type of the input arguments. In Table 12.1, if the order of the data type `x` is less than or equal to float, the returned data type is float. The returned data type is double if `x` is of double type. If the argument `x` of a function in Table 12.1 is NaN, the function will return NaN. In Tables 12.2 to 12.4, the returned data type will be the same as the higher order data type of two input arguments if any of two arguments is float or double. Otherwise, the float is the default returned data type.

Functions defined in this section will return float or double, except for functions **abs**() and **pow**(). If the argument of the function **abs**() is an integral value, the returned data type is int. If the argument of the function **fabs**() is a simple data type including int and float, the returned data type is double. If the arguments of the function **pow**() are integral values, the returned data type is double. For example, **pow**(2,16) will return the value 65536 of double type.

The absolute function **abs**(x) will compute the absolute value of an integer or a floating-point number. The absolute value of a negative infinity $-\infty$ is a positive infinity ∞.

The **sqrt**(x) function computes the nonnegative square root of `x`. If `x` is negative, the result is NaN, except that **sqrt**$(-0.0) = -0.0$ according to the IEEE 754 standard. The square root of infinity **sqrt**(∞) is infinity.

The **exp**(x) function computes the exponential function of `x`. The following results hold: $e^{-\infty} = 0.0; e^{\infty} = \infty; e^{\pm 0.0} = 1.0$.

The **log**(x) function computes the natural logarithm of `x`. If `x` is negative, the result is NaN. The value of -0.0 is considered equal to 0.0 in this case. The following results hold: $\mathbf{log}(\pm 0.0) = -\infty; \mathbf{log}(\infty) = \infty$. The `log10`(x) function computes the base-ten logarithm of `x`. If `x` is negative, the result is a NaN. Like the

Table 12.1: Results of real functions for ± 0.0, $\pm\infty$, and NaN

function	x value and results						
	$-$Inf	$-$x1	-0.0	0.0	x2	Inf	NaN
abs(x)	Inf	x_1	0.0	0.0	x_2	Inf	NaN
fabs(x)	Inf	x_1	0.0	0.0	x_2	Inf	NaN
sqrt(x)	NaN	NaN	-0.0	0.0	sqrt(x)	Inf	NaN
exp(x)	0.0	e^{-x_1}	1.0	1.0	e^{x_2}	Inf	NaN
log(x)	NaN	NaN	$-$Inf	$-$Inf	log(x_2)	Inf	NaN
log10(x)	NaN	NaN	$-$Inf	$-$Inf	$\log_{10}(x_2)$	Inf	NaN
sin(x)	NaN	$-$sin(x_1)	-0.0	0.0	sin(x_2)	NaN	NaN
cos(x)	NaN	cos(x_1)	1.0	1.0	cos(x_2)	NaN	NaN
tan(x)	NaN	$-$tan(x_1)	-0.0	0.0	tan(x_2)	NaN	NaN
	Note: tan($\pm\pi/2 + 2 * k * \pi$) = $\pm$Inf						
asin(x)	NaN	$-$asin(x_1)	-0.0	0.0	asin(x_2)	NaN	NaN
	Note: asin(x) = NaN, for $\lvert x\rvert > 1.0$						
acos(x)	NaN	acos(x_1)	pi/2	pi/2	acos(x_2)	NaN	NaN
	Note: acos(x) = NaN, for $\lvert x\rvert > 1.0$						
atan(x)	$-$pi/2	$-$atan(x_1)	-0.0	0.0	atan(x_2)	pi/2	NaN
sinh(x)	$-$Inf	$-$sinh(x_1)	-0.0	0.0	sinh(x_2)	Inf	NaN
cosh(x)	Inf	cosh(x_1)	1.0	1.0	cosh(x_2)	Inf	NaN
tanh(x)	-1.0	$-$tanh(x_1)	-0.0	0.0	tanh(x_2)	1.0	NaN
asinh(x)	$-$Inf	$-$asinh(x_1)	-0.0	0.0	asinh(x_2)	Inf	NaN
acosh(x)	NaN	NaN	NaN	NaN	acosh(x_2)	Inf	NaN
	Note: acosh(x) = NaN, for $x < 1.0$; acosh(1.0) = 0.0						
atanh(x)	NaN	$-$atanh(x_1)	-0.0	0.0	atanh(x_2)	NaN	NaN
	Note: atanh(x) = NaN, for $\lvert x\rvert > 1.0$; atanh($\pm$1.0) = $\pm$Inf						
ceil(x)	$-$Inf	ceil($-x_1$)	-0.0	0.0	ceil(x_2)	Inf	NaN
floor(x)	$-$Inf	floor($-x_1$)	-0.0	0.0	floor(x_2)	Inf	NaN
ldexp(x, k)	$-$Inf	ldexp($-x_1, k$)	-0.0	0.0	ldexp(x_2, k)	Inf	NaN
modf(x, &y)	-0.0	modf($-x_1$, &y)	-0.0	0.0	modf(x_2, &y)	0.0	NaN
y	$-$Inf	y	-0.0	0.0	y	Inf	NaN
frexp(x, &k)	$-$Inf	frexp($-x_1$, &k)	-0.0	0.0	frexp(x_2, &k)	Inf	NaN
k	0	k	0	0	k	0	0

Table 12.2: Results of the function pow(y, x) for $\pm 0.0, \pm\infty$, and NaN

pow(y, x)											
y value	x value										
	−Inf	−x1	$-2k-1$	−2k	−0.0	0.0	2k	$2k+1$	x2	Inf	NaN
Inf	0.0	0.0	0.0	0.0	1.0	1.0	Inf	Inf	Inf	Inf	NaN
y2 > 1	0.0	$y_2^{-x_1}$	y_2^{-2k-1}	y_2^{-2k}	1.0	1.0	y_2^{2k}	y_2^{2k+1}	$y_2^{x_2}$	Inf	NaN
1.0	NaN	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	NaN	NaN
0 < y2 < 1	Inf	$y_2^{-x_1}$	y_2^{-2k-1}	y_2^{-2k}	1.0	1.0	y_2^{2k}	y_2^{2k+1}	$y_2^{x_2}$	0.0	NaN
0.0	Inf	Inf	Inf	Inf	1.0	1.0	0.0	0.0	0.0	0.0	NaN
−0.0	Inf	Inf	−Inf	Inf	1.0	1.0	0.0	−0.0	0.0	0.0	NaN
−y1	NaN	NaN	$-y_1^{-2k-1}$	y_1^{-2k}	1.0	1.0	y_1^{2k}	$-y_1^{2k+1}$	NaN	NaN	NaN
−Inf	NaN	NaN	−0.0	0.0	1.0	1.0	Inf	−Inf	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Table 12.3: Results of the function atan2(y, x) for $\pm 0.0, \pm\infty$, and NaN

atan2(y, x)							
y value	x value						
	−Inf	−x1	−0.0	0.0	x2	Inf	NaN
Inf	3∗pi/4	pi/2	pi/2	pi/2	pi/2	pi/4	NaN
y2	pi	atan2($y_2, -x_1$)	pi/2	pi/2	atan2(y_2, x_2)	0.0	NaN
0.0	pi	pi	pi	0.0	0.0	0.0	NaN
−0.0	−pi	−pi	−3∗pi/4	−pi/2	−0.0	−0.0	NaN
−y1	−pi	atan2($-y_1, -x_1$)	−pi/2	−pi/2	atan2($-y_1, x_2$)	−0.0	NaN
−Inf	−3∗pi/4	−pi/2	−pi/2	−pi/2	−pi/2	−pi/4	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Table 12.4: Results of the function fmod(y, x) for $\pm 0.0, \pm\infty$, and NaN

fmod(y, x)							
y value	x value						
	−Inf	−x1	−0.0	0.0	x2	Inf	NaN
Inf	NaN	NaN	NaN	NaN	NaN	NaN	NaN
y2	y_2	fmod($y_2, -x_1$)	NaN	NaN	fmod(y_2, x_2)	y_2	NaN
0.0	0.0	0.0	NaN	NaN	0.0	0.0	NaN
−0.0	−0.0	−0.0	NaN	NaN	−0.0	−0.0	NaN
−y1	y_1	fmod($-y_1, -x_1$)	NaN	NaN	fmod($-y_1, x_2$)	$-y_1$	NaN
−Inf	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

function **log**(), the value of -0.0 is considered equal to 0.0. The following results hold: $\mathbf{log10}(\pm 0.0) = -\infty; \mathbf{log10}(\infty) = \infty$.

The trigonometric functions **sin**(x), **cos**(x), and **tan**(x) compute sine, cosine, and tangent of x measured in radians, respectively. The sine and tangent are odd functions so that $\mathbf{sin}(\pm 0.0) = \pm 0.0$ and $\mathbf{tan}(\pm 0.0) = \pm 0.0$. The cosine is an even function so that $\mathbf{cos}(\pm 0.0) = 1.0$. When the value of the argument is positive or negative infinity, all these functions return NaNs. Theoretically, it is true that $\mathbf{tan}(\pm\pi/2 + 2 * k * \pi) = \pm\infty$. But, in practice, because the irrational number π cannot be represented exactly in float or double data, the **tan**(x) function will never return infinities of $\pm\infty$. The function **tan**() is not continuous at $\pi/2$, $\mathbf{tan}(\pi/2 - \varepsilon) = \infty$, and $\mathbf{tan}(\pi/2 + \varepsilon) = -\infty$, where ε is a very small number. Due to the finite precision and round-off errors of floating-point numbers, one may get a wrong result near the value of $\pi/2$.

The properties of odd functions of sine and tangent are reflected in their inverse functions **asin**(x) and **atan**(x). The **asin**(x) function computes the principal value of the arc sine of x. When the value of x is in the range of $[-1.0, 1.0]$, the **asin**(x) function returns the value in the range of $[-\pi/2, \pi/2]$ radians. When x is outside the range of $[-1.0, 1.0]$, the arc sine is undefined and **asin**(x) returns NaN. The range of the input value for the even function **acos**(x) of arc cosine is the same as that of **asin**(x). The **acos**(x) function computes the principal value of the arc cosine of x. The range of the principal value of the arc cosine is $[0.0, \pi]$ radians. The **atan**(x) function computes the principal value of the arc tangent of x. The **atan**(x) function returns the value in the range of $[-\pi/2, \pi/2]$ radians. The following results hold: $\mathbf{atan}(\pm\infty) = \pm\pi/2$.

Like trigonometric functions **sin**(x) and **tan**(x), the hyperbolic functions **sinh**(x) and **tanh**(x) are odd functions. The **sinh**(x) and **tanh**(x) functions compute the hyperbolic sine and tangent of x, respectively. The even function **cosh**(x) computes the hyperbolic cosine of x. The following results hold: $\mathbf{sinh}(\pm 0.0) = \pm 0.0; \mathbf{cosh}(\pm 0.0) = 1.0; \mathbf{tanh}(\pm 0.0) = \pm 0.0; \mathbf{sinh}(\pm\infty) = \pm\infty; \mathbf{cosh}(\pm\infty) = \infty; \mathbf{tanh}(\pm\infty) = \pm 1.0;$

The inverse hyperbolic functions are not defined by the C standard. In Ch, the inverse hyperbolic sine, cosine, and tangent are defined as **asinh**(x), **acosh**(x), and **atanh**(x), respectively. For the **acosh**(x) function, if the argument is less than 1.0, it is undefined and **acosh**(x) returns NaN. **acosh**(1.0) returns a positive zero. The valid domain for function **atanh**(x) is $[-1.0, 1.0]$. The following results hold: $\mathbf{asinh}(\pm 0.0) = \pm 0.0; \mathbf{asinh}(\pm\infty) = \pm\infty; \mathbf{acosh}(\infty) = \infty; \mathbf{atanh}(\pm 0.0) = \pm 0.0; \mathbf{atanh}(\pm 1.0) = \pm\infty$.

The **ceil**(x) function computes the smallest integral value not less than the value of x. The counterpart of **ceil**(x) is the function **floor**(x) which computes the largest integral value not greater than the value of x. The following results hold: $\mathbf{ceil}(\pm 0.0) = \pm 0.0; \mathbf{floor}(\pm 0.0) = \pm 0.0; \mathbf{ceil}(\pm\infty) = \pm\infty; \mathbf{floor}(\pm\infty) = \pm\infty$

The **ldexp**(x, k) function multiplies the value of the floating-point number x with the value of 2 raised to the power of k. The returned value of $x * 2^k$ keeps the sign of x.

The functions **modf**(x, xptr) and **frexp**(x, iptr) have two arguments. The first argument is the input data and the second argument is a pointer which will store the resulted integral part of the function call. The **modf**(x, xptr) function breaks the argument x into integral and fractional parts, each of which has the same sign as the argument. The **modf**() function returns the fractional part and the integral part is stored to the memory pointed to by the second argument. The basic data types of two arguments must be the same. For example, if the first argument x is float, the second argument `xptr` must be a pointer to float. If the first argument is a metanumber, the integral part will equal the metanumber whereas the fractional part becomes zero with the sign of the first argument except for NaN. The **frexp**(x, iptr) function breaks a floating-point number into a normalized fraction and an integral power of 2 in the form of $x * 2^k$. The **frexp**(x, iptr) function returns the normalized fraction and the integral part is stored to the memory pointed to by the second argument, which is a pointer to int. If the first argument is a metanumber, the fractional part will equal the metanumber whereas the integral part becomes zero.

The mathematical functions **pow**(y, x), **atan2**(y,x), and **fmod**(y,x) have two input arguments. The results of these three functions are given in Tables 12.2 to 12.4. The **pow**(y, x) function computes y raised to the

power of x, which is y^x or $e^{x\log(y)}$. If x is negative, y^x becomes $1/y^{|x|}$ with the defined division operation given in Table 7.6. If y is less than zero and x is not an integral value including zero, the function is undefined. The value of -0.0 is considered equal to 0.0 in the evaluation of $\log(-0.0)$ when the value of x is not an integral number. When x is an odd integer number and y is negative, the result is negative. For a positive value of y, the result depends on the value of y when x is infinity. If y is less than 1, y^∞ is 0.0; 1.0^∞ is indeterminate; if y is greater than 1, y^∞ is infinity. If y is infinity and x is zero, $(\pm\infty)^{\pm 0.0}$ are 1.0.

The **atan2**(y, x) function computes the principal value of the arc tangent of `y/x` using the signs of both arguments to determine the returned value in the range of $[-\pi, \pi]$ radians. Given the (x, y) coordinates of a point in the X-Y plane, the **atan2**(y, x) function computes the angle of the radius from the origin to the point. Any positive number that overflows is represented by Inf. The negative overflow is $-$Inf. The following results hold: $\mathbf{atan2}(\pm\mathrm{Inf}, -\mathrm{Inf}) = \pm 3\pi/4$; $\mathbf{atan2}(\pm\mathrm{Inf}, \mathrm{Inf}) = \pm\pi/4$; $\mathbf{atan2}(\pm\mathrm{Inf}, \mathrm{x}) = \pm\pi/2$; $\mathbf{atan2}(\pm y, \mathrm{Inf}) = \pm 0.0$; and $\mathbf{atan2}(\pm y, -\mathrm{Inf}) = \pm\pi$. When both values of y and x are zeros, the function **atan2**(y, x) will return the results consistent with the manipulation of metanumbers discussed so far. The value of -0.0 is considered as a negative number less than zero. Therefore, the following results are defined for these special operations: $\mathbf{atan2}(0.0, -0.0) = \pi$; $\mathbf{atan2}(0.0, 0.0) = 0.0$; $\mathbf{atan2}(-0.0, -0.0) = -3\pi/4$; and $\mathbf{atan2}(-0.0, 0) = -\pi/2$, which is consistent with the treatment of the metanumbers of $\pm$Inf in **atan2**($-$Inf, $-$Inf) = $-3pi/4$. In Ch, $\mathbf{atan2}(0.0, 0.0)$ is a specially defined value. These results are different from those by the SUN's C compiler, which is in conformance with 4.3 Berkeley Software Delivery (SUN, 1990a). According to 4.3BSD, the results for these special cases are $\mathbf{atan2}(\pm 0.0, -0.0) = \pm 0.0$ and $\mathbf{atan2}(\pm 0.0, 0.0) = \pm\pi$, which implies that the values of ± 0.0 on the x-axis are different from those on the y-axis.

The **fmod**(y,x) function computes the floating-point remainder of `y/x`. The **fmod**(y,x) function returns the value of $y - i * x$ for some integer i. The magnitude of the returned value with the same sign of x is less than the magnitude of x. If x is zero, the function is undefined and returns NaN. When y is infinity, the result is also undefined. If x is infinity and y is a finite number, the result is the same as y.

12.2 Programming Examples

12.2.1 Computation of Extreme Values of Floating-Point Numbers

Due to different machine architectures for representation of floating-point numbers, the extreme values such as the maximum representable floating-point value are different. For two machines with the same representation of floating-point values, the same operations such as adding two values on each machine may get different results, depending on the schemes for rounding a number that cannot be represented exactly. To aid serious numerically oriented programmers in writing their programs, the C standard added the header `float.h` as a companion to the existing header `limits.h` to deal with the machine-dependent integer values only. In this section, we will show how parameters defined in the C standard library `float.h` can be computed in Ch without knowing the intricate architecture of the computer. A program can depend less on these parameters if a language can support metanumbers Inf and NaN. The use of metanumbers such as Inf and NaN instead of parameters is recommended for Ch programming.

Minimum Floating-Point Numbers FLT_MIN and FLT_MINIMUM

The parameter FLT_MIN is defined in the C standard library header `float.h` as a minimum normalized positive floating-point float number. If a number is less than FLT_MIN, it is called an *underflow*. Because the IEEE 754 standard provides a *gradual underflow*, the minimum denormalized positive floating-point float number is defined as FLT_MINIMUM in Ch. Because of gradual underflow, the Ch expression `x - y == 0` is TRUE iff x = y, which is not true for systems that lack gradual underflow. This parameter is

very useful from a programming point of view. As an example, assume that values of FLT_MINIMUM and FLT_MIN are 1.401298e-45 and 1.175494e-38, respectively. The following Ch code will illustrate subtleties of these two parameters.

```
float f, *flt_minimum;
int minimum, i;
minimum = 1;                          // memory location becomes 00000001
flt_minimum = &minimum;               // *flt_minimum becomes FLT_MINIMUM
i = *flt_minimum > 0.0;               // i becomes 1
i = FLT_MIN > *flt_minimum;           // i becomes 1
i = fabs(*flt_minimum) > 0.0;         // i becomes 1
f = (*flt_minimum)/(*flt_minimum); // f becomes 1.0; note 0.0/0.0 = NaN
f = f/1.e-46                      // f becomes Inf: 1.e-46 < FLT_MINIMUM
```

Applications of these two numbers in the handling of branch cuts of multiple-valued complex functions are described in Chapter 13.

Machine Epsilon FLT_EPSILON

The machine epsilon `FLT_EPSILON` is the difference between 1 and the least value greater than 1 that is representable in float. This parameter, defined in the C header `float.h`, is a system constant in Ch. This parameter is very useful for scientific computing. For example, due to the finite precision of the floating-point representation and alignment of addition operation, when a significantly small value and a large number are added together, the small number may not have contribution to the summation. Using FLT_EPSILON, adding a small positive number x to a large positive number y can capture at least three decimal digits of significance of y that can be tested by

```
if(x < y * FLT_EPSILON * 1000)
```

The following Ch code can calculate and print out the machine epsilon on the screen

```
float epsilon;
epsilon = 1.0;
while(epsilon+1 > 1)
  epsilon /= 2;
epsilon *= 2;
printf("The machine epsilon FLT_EPSILON is %e", epsilon);
```

For SUN SPARCStations, the output from the execution of the above code is as follows:
The machine epsilon FLT_EPSILON is 1.192093e-07
which matches the value of the parameter `FLT_EPSILON` defined in the C header `float.h`. Although the above computation of the parameter `FLT_EPSILON` is simple in Ch which uses the default rounding mode of round toward nearest, it may be vulnerable to other rounding modes. A more robust method (Plauger, 1992) to obtain this parameter is to manipulate the bit pattern of the memory of a float variable as shown in Section 12.2.1.

Maximum Floating-Point Number FLT_MAX

The parameter `FLT_MAX` defined in the C header `float.h` is the maximum representable finite floating-point number. Any value that is larger than `FLT_MAX` will be represented as Inf and any value less than

$-$FLT_MAX is represented by $-$Inf. If the value of FLT_MAX is represented as $fltmax * 10^e$, then the following two equations will be satisfied

$$(fltmax + FLT_EPSILON) * 10^e = Inf$$

$$(fltmax + FLT_EPSILON/2) * 10^e = FLT_MAX$$

where the machine epsilon FLT_EPSILON was defined in Section 12.2.1 and the exponential value e is to be calculated. The following Ch program will calculate FLT_MAX as well as FLT_MAX_10_EXP and FLT_MAX_EXP of the machine and print them on the screen. The value of FLT_MAX_10_EXP is the maximum integer such that 10 raised to its power is in the range of the representable finite floating-point numbers. The value of FLT_MAX_EXP is the maximum integer such that 2 raised to its power minus 1 is a representable finite floating-point number. For the illustrative purpose, only the while-loop control structure is used in this example.

```
float b, f, flt_max;
int e, i, flt_max_exp, flt_max_10_exp;
b = 10; e = 0; f = b;
/* calculate exponential number e, 38 in the example */
while(f != Inf)
{
  e++; f*=b;
}
flt_max_10_exp = e;
/* calculate leading non-zero number, 3 in the example */
i = 0; f = 0.0;
while(f != Inf)
  f = ++i * pow(b, e);
/* calculate numbers after decimal point, 40282347... in the example */
flt_max = i;
while(e != 0)
{
  flt_max = --flt_max * b;
  e--; i = 0; f = 0.0;
  while( f != Inf && i < 10)
  {
    f = ++flt_max * pow(b, e);
    i++;
  }
}
f = frexp(flt_max, &flt_max_exp);       // calculate FLT_MAX_EXP
printf("FLT_MAX = %.8e \n", flt_max);
printf("FLT_MAX (in binary format) = %b \n", flt_max);
printf("FLT_MAX_10_EXP = %d \n", flt_max_10_exp);
printf("FLT_MAX_EXP = %d \n", flt_max_exp);
```

The output of the above code on SUN SPARCStations is as follows:
FLT_MAX = 3.40282347e+38
FLT_MAX (in binary format) = 01111111011111111111111111111111

FLT_MAX_10_EXP = 38
FLT_MAX_EXP = 128
The above values for `FLT_MAX, FLT_MAX_10_EXP`, and `FLT_MAX_EXP` are the same as the parameters defined in the C header `float.h`. By just changing the declaration of the first statement from float to double, the corresponding extreme values `DBL_MAX, DBL_MAX_10_EXP`, and `DBL_MAX_EXP` for double can be obtained. In this case, the polymorphic arithmetic operators and mathematical functions **pow**() and **frexp**() will return double data.

In the above calculation of the extreme floating-point values, the user does not need to know the intricate machine representation of floating-point numbers. If one knows the machine representation of a floating-point number, the calculation of the extreme values can be much simpler. For example, according to Table 6.1, the value of `FLT_MAX` is represented in a hexadecimal form as $(7\text{F7FFFFF})_{16}$. The following Ch program can be used to obtain the maximum representable finite floating-point number `FLT_MAX`.

```
int i; float *flt_max;
flt_max = &i;       // flt_max points to the memory location of i
i = 0X7F7FFFFF;     // *flt_max becomes FLT_MAX
```

The maximum float number `FLT_MAX` can also be readily obtained by the I/O function **scanf**() with the binary input format `"%32b"`. For interested readers, can you think of any other method for computing the maximum representable finite floating-point number `FLT_MAX` by a C or Fortran program without knowing the machine architecture? The major difficulty is that, due to the internal alignment for calculation of the floating-point numbers, the significantly small number will be ignored when it is added to or subtracted from a large number. For example, the execution of the command `f = FLT_MAX + 3.0e30` will give the variable `f` the value of FLT_MAX although the value of $3.0 * 10^{30}$ is not a small number, but it is significantly smaller than `FLT_MAX` and ignored in the above addition operation. The following two Ch expressions will further demonstrate the difference between FLT_MAX and Inf.

$$1/\text{Inf} * \text{FLT_MAX} = 0.0$$

,

$$1/\text{FLT_MAX} * \text{FLT_MAX} = 1.0$$

.

12.2.2 Programming with Metanumbers

The Ch language distinguishes -0.0 from 0.0 for real numbers. The metanumbers $0.0, -0.0$, Inf, $-$Inf, and NaN are very useful for scientific computing. For example, the function $f(x) = e^{\frac{1}{x}}$ is not continuous at the origin as is shown in Figure 12.1, which was generated by Program 23.12 on page 433 described in Chpater 23. This discontinuity can be handled gracefully in Ch. The evaluation of the Ch expression **exp**(1/0.0) will return Inf and **exp**$(1/(-0.0))$ gives 0.0, which corresponds to mathematical expressions $e^{\frac{1}{0_+}}$ and $e^{\frac{1}{0_-}}$ or $\lim_{x\to 0_+} e^{\frac{1}{x}}$ and $\lim_{x\to 0_-} e^{\frac{1}{x}}$, respectively. In addition, the evaluation of expressions **exp**(1.0/Inf) and **exp**(1.0/($-$Inf)) will get the value of 1.0. As another example, the function `finite(x)` recommended by the IEEE 754 standard is equivalent to the Ch expression `-Inf < x && x < Inf`, where x can be a float/double variable or expression. If x is a float, `-Inf < x && x < Inf` is equivalent to `-FLT_MAX <= x && x <= FLT_MAX`; If x is a double, `-Inf < x && x < Inf` is equivalent to `-DBL_MAX <= x && x <= DBL_MAX`. The mathematical statement "$if\ -\infty < value <= \infty, then\ y\ becomes\ \infty$" can be easily programmed in Ch as follows

```
if(-Inf < value && value <= Inf) y = Inf;
```

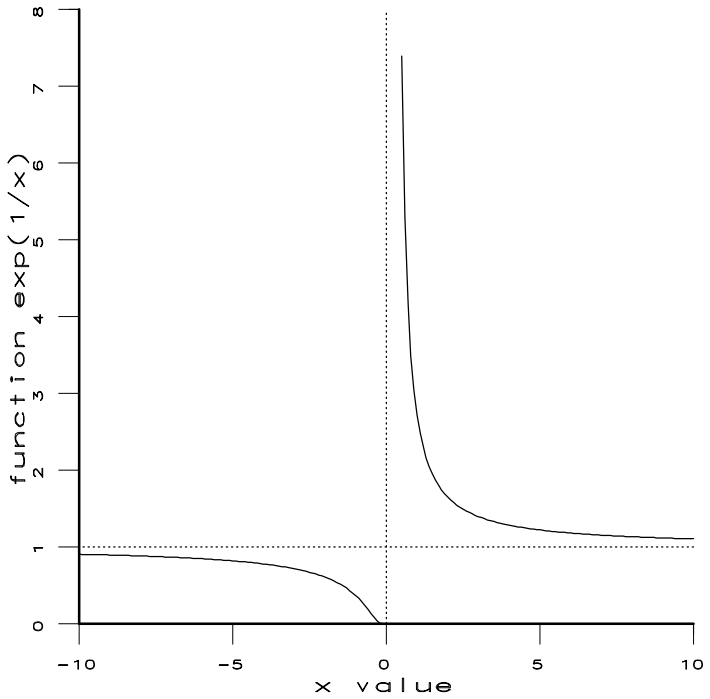


Figure 12.1: Function $f(x) = e^{\frac{1}{x}}$.

However, a computer can only evaluate an expression step by step. Although the metanumbers are limits of the floating-point numbers, they cannot replace mathematical analysis. For example, the natural number e equal to 2.718281828... is defined as the limit value of the expression

$$\lim_{x \to \infty} \left(1 + \frac{1}{x}\right)^x = e.$$

However, the value of the expression **pow**(1.0 + 1.0/Inf, Inf) in Ch is NaN. The evaluation of this expression is carried out as follows:

$$\left(1.0 + \frac{1.0}{Inf}\right)^{Inf} = (1.0 + 0.0)^{Inf} = 1.0^{Inf} = NaN$$

If the value FLT_MAX instead of Inf is used in the above expression, the result is obtained by

$$\left(1.0 + \frac{1.0}{FLT_MAX}\right)^{FLT_MAX} = (1.0 + 0.0)^{FLT_MAX} = 1.0^{FLT_MAX} = 1.0$$

Because metanumber NaN is unordered, a program involving relational operations should be handled cautiously. For example, the expression `x > y` is not equivalent to `!(x <= y)` if either x or y is a NaN. As another example, the following Ch code fragment

```
if(x > 0.0)  function1();
else function2();
```

is different from the code fragment

```
if(x <= 0.0) function2();
else function1();
```

The second `if`-statement should be written as `if(x <= 0.0 || isnan(x))` in order for these two code fragments to have the same functionality.

Chapter 13

Programming with Complex Numbers

The complex number, an extension of real number, has wide applications in science and engineering. Owing to its importance in scientific programming, numerically oriented programming languages and software packages usually provide complex number support in one way or another. For example, Fortran a language mainly for scientific computing, has provided complex data type since its earliest days. The early version of C does not have complex as a basic data type because numerically oriented scientific computing was not its original design goal. Complex data types have been added in C99. Ch supports all features mandated by C99 with extensions. Generic mathematical functions are overloaded for handling complex numbers with optional arguments for different branch cuts. Ch provides real metanumbers of Inf, $-$Inf, and NaN and signed zeros 0.0 and -0.0, which makes the power of the IEEE 754 standard for binary floating-point arithmetic easily available to the programmer. Ch extends the idea of metanumbers to complex numbers not only for arithmetic, but also for commonly used mathematical functions in the spirit of the IEEE 754 standard. Ch treats floating-point real numbers with signed zeros and complex numbers with unsigned zeros as well as Not-a-Number and infinities in an integrated consistent manner.

13.1 Complex Numbers

13.1.1 Complex Constants and Complex Variables

Complex numbers $z \in \boldsymbol{\mathcal{C}} = \{(x, y) \mid x, y \in \boldsymbol{\mathcal{R}}\}$ can be defined as ordered pairs

$$z = (x, y) \tag{13.1}$$

with specific addition and multiplication rules [10][17]. The real numbers x and y are called the *real* and *imaginary parts* of z. If we identify the pair of $(x, 0.0)$ as the real numbers, the real number $\boldsymbol{\mathcal{R}}$ is a subset of $\boldsymbol{\mathcal{C}}$; that is, $\boldsymbol{\mathcal{R}} = \{(x, y) \mid x \in \boldsymbol{\mathcal{R}}, y = 0.0\}$ and $\boldsymbol{\mathcal{R}} \subset \boldsymbol{\mathcal{C}}$. If a real number is considered either as x or $(x, 0.0)$ and let i denote the *pure imaginary number* $(0, 1)$ with $i * i = -1$, complex numbers can be mathematically represented as

$$z = x + iy \tag{13.2}$$

Both Equations (13.1) and (13.2) can be implemented for complex numbers in a computer language. General-purpose computer programming languages such as Fortran, Ada, and Common Lisp tend to use Equation (13.1) whereas some mathematical software packages incline to Equation (13.2).

Following the lead of FORTRAN in scientific programming, a complex number can be created in Ch by the complex constructor **complex**(x, y) with $x, y \in \boldsymbol{\mathcal{R}}$. For example, a complex number with its real part of 3.0 and imaginary part of 4.0 can be constructed by **complex**(3.0, 4.0). The new type qualifier **complex** is a keyword in Ch. Internally, a complex number consists of two floats at the current implementation. Therefore,

if arguments of a complex constructor are not floats, they will be cast to floats internally. All floating-point constants in Ch are double by default. The float constants can be obtained by suffixing a floating-point constant with F or f. The complex constructor returns complex or double complex polymorphically, depending on the data types of the input arguments. For example, **complex**(3, 4.0), **complex**(3.0f, 4.0), and **complex**(3.0, 4.0F) return a double complex number of **complex**(3.0, 4.0).

One can declare not only a *simple complex variable*, but also *pointer to complex*, *array of complex*, and *array of pointer to complex*, etc. Declarations of these complex variables are similar to the declarations of any other data types in C. The array and pointer of complex in Ch are manipulated in the same manner as the floating-point float and double. The following code segment will illustrate how complex is declared and manipulated in Ch:

```
double complex z;      // declare z as double complex variable
float complex z1;      // declare z1 as float complex variable
complex *zptr1;        // declare zptr1 as pointer to complex variable
complex z2[2], z3[2,3];// declare z2 and z3 as arrays of complex
complex *zptr2[2][4];  // declare zptr2 as array of pointer to complex
zpt1r = &z1;           // zptr1 point to the address of z1
*zptr1 = complex(1,2); // z1 becomes 1+i2
```

Complex numbers are supported in C99 and C++. In order to be compatible with both C99 and C++, Ch defined one micro, two types, and some functions prototypes in both header files **complex.h** and **complex**. The macro **I** is defined as `complex(0.0, 1.0)` to represent an imaginary number with the unit length.

13.2 Complex Planes and Complex Metanumbers

Mathematically, complex numbers can be represented in the extended complex plane shown in Figure 13.1 [10][17]. In Figure 13.1, there is a one-to-one correspondence between the points on the Riemann sphere Γ and the points on the extended complex plane $\mathcal{C}$. The point p on the surface of the sphere is determined by the intersection of the line through the point z and the north pole N of the sphere. There is only *one* complex infinity in the extended complex plane. The north pole N corresponds to the point at infinity. Because of the finite representation of floating-point numbers, *the extended finite complex plane* shown in Figure 13.2 is introduced in this chapter. Any complex values inside the ranges of $|x| < \text{FLT_MAX}$ and $|y| < \text{FLT_MAX}$ are representable in finite floating-point numbers. Variable x is used to represent the real part of a complex number and y the imaginary part; FLT_MAX, a predefined system constant, is the maximum representable finite floating-point number in the float data type. Outside this rectangular area, a complex number is treated as a Complex-Infinity represented as ComplexInf or complex(Inf,Inf) in Ch. The one-to-one correspondence between points on the Riemann sphere Γ and the extended complex plane is no longer valid for the unit sphere Λ and the extended finite complex plane. All points on the surface of the upper part Λ_1 of the unit sphere correspond to the complex infinity. Points on the lower part Λ_2 of the sphere and points in the extended finite complex plane are in one-to-one correspondence. The boundary between surfaces Λ_1 and Λ_2 corresponds to the threshold of overflow. For example, points p_1 and p_2 on the unit sphere Λ correspond to points $z_1 =$ complex(FLT_MAX, 0.0) and $z_2 =$ complex(FLT_MAX, FLT_MAX), respectively, in the extended finite complex plane shown in Figure 13.2. The origin of the extended finite complex plane is complex(0.0, 0.0), which stands for complex zero. In Ch, an undefined or mathematically indeterminate complex number is denoted as complex(NaN, NaN) or ComplexNaN, which stands for Complex-Not-a-Number. The special complex numbers of ComplexInf and and ComplexNaN are referred to as *complex metanumbers*.

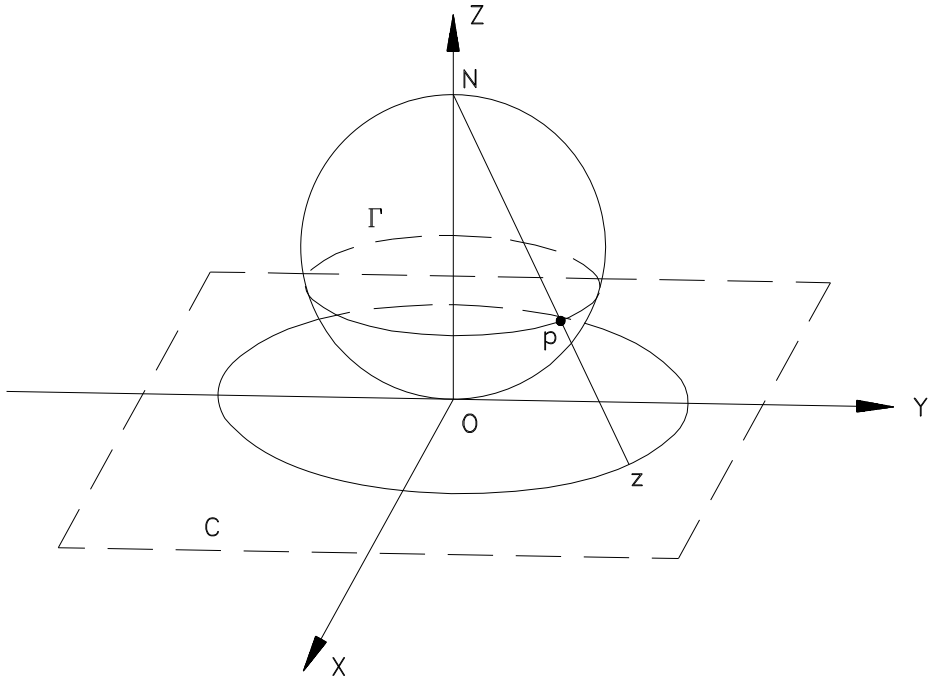


Figure 13.1: The Riemann sphere Γ and extended complex plane.

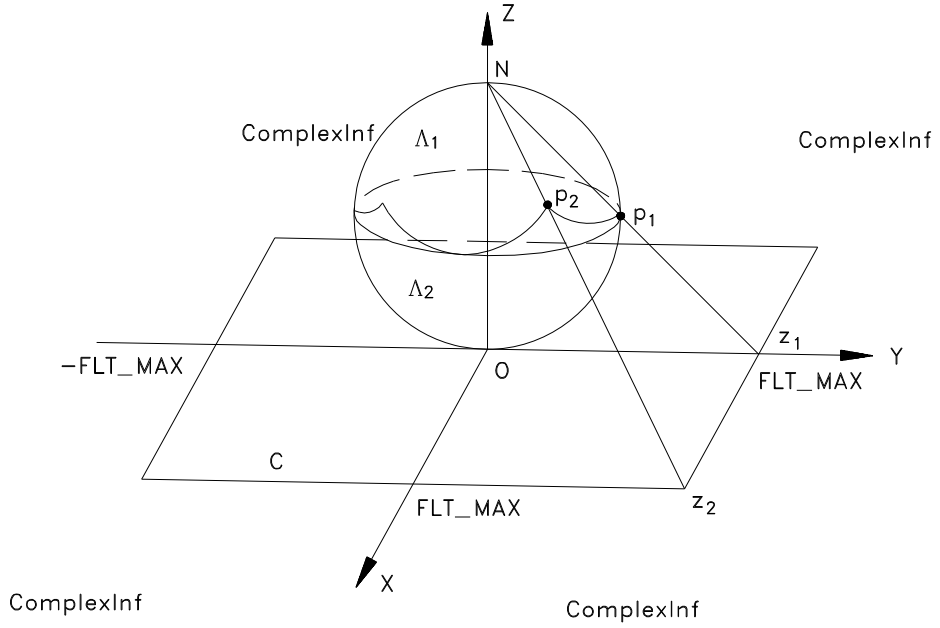


Figure 13.2: The unit sphere Λ and extended finite complex plane.

Because of the mathematical infinities of $\pm\infty$, it becomes necessary to distinguish a positive zero 0.0 from a negative zero -0.0 for real numbers. Unlike the real line, along which real numbers can approach the origin through the positive or negative numbers, the origin of the complex plane can be reached in any directions in terms of the limit value of $\lim_{r\to 0} re^{i\theta}$ where r is the modulus and θ is the phase of a complex number. Therefore, complex operations and complex functions in Ch do not distinguish 0.0 from -0.0 for real and imaginary parts of complex numbers. Because of these differences, some operations and functions need to be handled differently for real and complex numbers, especially for real metanumbers and complex metanumbers. For example, following the IEEE 754 standard, the addition of two real positive infinities is a value of infinity in Ch. The addition of two complex infinities is indeterminate according to complex analysis, although the value of ComplexInf is represented internally as two positive infinities of Inf. As another example, following the C standard, the mathematical function **atan2**(y, x) in Ch returns a value in the range of $[-\pi, \pi]$. The value of the expression **atan2**$(-0.0, -1)$ is $-\pi$. Using this result as the phase angle for complex number $-1.0 - i0.0$, the square root of $-1.0 - i0.0$, expressed in Ch as **sqrt**(complex$(-1.0, -0.0)$), becomes complex$(0.0, -1.0)$, which is obtained by $\cos(-\pi/2) + i\sin(-\pi/2) = 0.0 - i$. In our definition, this is the second branch of the square root function for the complex number of complex$(-1.0, -0.0)$ obtained by the expression **sqrt**(complex$(-1.0, -0.0), 1)$ where the second argument of the function **sqrt**() indicates the branch number with the default value of 0. As illustrated in this example, the mathematical functions in Ch are polymorphic with a variable number of arguments so that the function **sqrt**() cannot only be used to compute the square root of a real number, but also to calculate the different branches of the square root of a complex number. Due to polymorphism and variable number of arguments for mathematical functions, scientific computing with complex numbers in Ch is much simpler in comparison to Fortran and other languages.

13.2.1 Data Conversion Rules

Ch is a loosely typed language. All arguments of calling functions will be checked for compatibility with the data types of the called functions. The data types of operands for an operation will also be checked for compatibility. If data types do not match, the system will signal an error and print out some informative messages for the convenience of program debugging. However, unlike languages such as Pascal which prohibits automatic type conversion, some data type conversion rules have been built into Ch so that they can be invoked whenever necessary. This will save many explicit type conversion commands for a program. The order of the data type in Ch is arranged as

data type	order
double complex	↑ high
complex	
double	
float	
int	↓ low
char	

with char being the lowest data type and double complex the highest data type. The default conversion rules will be briefly discussed in this section as follows:

1. Char, int, float, and double can be converted according to ISO C data conversion rules. The ASCII value of a character will be used in conversion for a char data type. Demotion of data may cause loss of the information.

2. Char, int, float, and double can be converted to complex with its imaginary part being zero. When casting a real number into a complex number, the values of Inf and $-$Inf become ComplexInf; and the value of NaN becomes ComplexNaN. Conversion from double to complex may lose the information. A real number can be cast into a complex explicitly by the complex construction function **complex**(x,y), which will be discussed in details in section 13.5.

3. When a complex is converted to char, int, float, and double, only its real part is used and the imaginary part will be discarded if the imaginary part is zero. If the imaginary part is not equal to zero, the converted real number becomes NaN. The real and imaginary components of a complex number can be obtained explicitly by the functions **real**(z) and **imag**(z), which will be discussed in detail in Section 13.5. When a complex number is converted to a real number either implicitly by assignment statement such as f = z or explicitly by **real**(z), **imag**(z), **float**(z), **double**(z), (**float**)z, and (**double**)z; the sign of a zero will not be carried over. Converting a complex number to an integral value such as char and int is equivalent to conversion of **real**(z) to an integral value if the imaginary part is not identically zero. For example, `i = ComplexInf` will make i equal to INT_MAX. However, if **real**() or **imag**() is used as an lvalue, the sign of zeros from rvalue will be preserved, which will allow experimentation with signed zeros in computations of complex numbers. An *lvalue* is any object that occurs on the left hand side of an assignment statement. The lvalue refers to a memory such as a variable or pointer, not a function or constant. On the other hand, the *rvalue* refers to the value of the expression on the right hand side of an assignment statement. Details about the lvalue will be discussed in Section 13.6.

4. In binary operations such as addition, subtraction, multiplication, and division with mixed data types, the result of the operation will carry the higher data type of two operands. For example, the result of addition of an int and a double will result in a double. When one of the two binary operands is complex and the data type of other operand is a real number, the real number will be cast into a complex before the operation is carried out. This conversion rule is also valid for an assignment statement when data types of the lvalue and rvalue are different.

5. In a pointer assignment statement, the pointer types of lvalue and rvalue can be different. They will be reconciled internally. To comply with the ISO C standard, the data type of the rvalue can also be explicitly cast into that of the lvalue in an assignment in Ch. For example, the statement $fp = (float*)intptr$ will cast the integer pointer $intptr$ to float pointer before its address is assigned to float pointer fp. However, the contents pointed to by $intptr$ will not be changed by this data type casting operation. For example, if $*intptr$ is 90, the value of $*fp$ will not be equal to 90 because of the difference in their internal representations for int and float. The memory of a complex variable can be accessed by pointers. If the real or imaginary part of a complex variable is obtained by a float pointer, the sign of a zero will be carried over, which will be discussed in Section 13.6.

The following code segment will illustrate how different data types are automatically converted in Ch.

```
char c;
int i;
float f;
double d;
complex z, *zptr;
c = 'a';              // c is  'a'
i = c;                // i is 97, ASCII number of  'a'
f = i;                // f is 97.0
d = i;                // d is 97.0
```

```
z = complex(c+1, f);  // z is 98.0 + i 97.0
z = complex(Inf, Inf);// z is ComplexInf
z = Inf;              // z is ComplexInf
z = -Inf;             // z is ComplexInf
f = z;                // f is NaN, since real(ComplexInf) is NaN
d = z;                // d is NaN, since real(ComplexInf) is NaN
i = Inf;              // i is 2147483647 = INT_MAX,
i = z;                /* i is 2147483647, int of NaN is 2147483647
                         plus warning message */
z = complex(d+1, 3);  // z is 98.0 + i 3.0
c = z;                // c is the delete character, ASCII number is 127
i = z;                // i is 2147483647, int of NaN
f = z;                // f is NaN
d = z;                // d is NaN
z = NaN;              // z is ComplexNaN
zptr = &z;            // zptr point to address of z
zptr++;               // zptr point to memory z plus 8 bytes
```

13.3 I/O for Complex Numbers

Since complex is a basic data type in Ch, it is desired that the I/O for this data type is also handled in the same manner as real numbers. Similar to Fortran, the real and imaginary parts of a complex number can be treated as two individual floats by the functions **real**(z) and **imag**(z) as will be discussed in Sections 13.4 and 13.5. Then, all standard I/O functions such as **printf**() and **scanf**() for real numbers can be readily used. In this section, how a complex number is treated as a single object by the standard I/O function will be discussed. Due to the space limit, only the enhancement related to the function **printf**() will be explained in the following discussions. However, the underlining principle can be applied to other I/O functions as well. The format of function **printf**() in Ch is as follows

```
int printf(char *format, arg1, arg2, ...)
```

The function **printf**() prints output to the standard output device under the control of the string pointed to by `format` and returns the number of characters printed. If the format string contains two types of objects — ordinary characters and conversion specifications beginning with a character of % and ending with a conversion character — the ISO C rules for **printf**() will be used. If the format string in **printf**() contains only ordinary characters, the subsequent numerical constants or variables will be printed according to preset default formats. For function **printf**(), a single conversion specification for a float will be used for both real and imaginary parts of a complex number. The default format for complex is %.2f, which will be applied to both real and imaginary parts of a complex number. The metanumbers ComplexInf and ComplexNaN are treated as regular complex numbers in I/O functions. For debugging purposes, the default output for ComplexInf and ComplexNaN are complex(Inf, Inf) and complex(NaN, NaN), respectively. The default output for complex zero is complex(0.00,0.00). The format for real and imaginary parts can be controlled by a format specifier. The following Ch program will illustrate how complex numbers are handled by the I/O functions **printf**() and **scanf**().

```
complex z1;
double complex z2, *zptr;
zptr = &z2;         /* zptr points to z2's memory location  */
printf("Please type in real and imaginary of two complex numbers \n");
scanf(&z1, zptr);
```

Table 13.1: The complex operations

Definition	Ch Syntax	Ch Semantics
negation	$-$z	$-x - iy$
addition	z1 $+$ z2	$(x_1 + x_2) + i(y_1 + y_2)$
subtraction	z1 $-$ z2	$(x_1 - x_2) + i(y_1 - y_2)$
multiplication	z1 $*$ z2	$(x_1 * x_2 - y_1 * y_2) + i(y_1 * x_2 + x_1 * y_2)$
division	z1 / z2	$\frac{x_1 * x_2 + y_1 * y_2}{x_2^2 + y_2^2} + i\frac{y_1 * x_2 - x_1 * y_2}{x_2^2 + y_2^2}$
equal	z1 == z2	$x_1 == x_2$ and $y_1 == y_2$
not equal	z1 != z2	x_1!=x_2 or y_1 != y_2

```
printf("The first complex is ", z1, "\n");
printf("The second complex is ", z2, "\n");
printf("The second complex is  %f \n", z2);
```

The result of the interactive execution of the above program is shown as follows

```
Please type in real and imaginary of two complex numbers
```

1 2.0 3.0 4

```
The first complex is complex(1.0000,2.0000)
The second complex is complex(3.0000,4.0000)
The second complex is complex(3.000000,4.000000)
```

where the second line in italic is the input and the rest are the output of the program.

13.4 Complex Operations

The arithmetic and relational operations for complex numbers are treated in the same manner as those for real numbers in Ch. This section will discuss how these operations are defined and handled by Ch.

13.4.1 Complex Operations with Regular Complex Numbers

The negation of a complex number, and arithmetic and comparison operations for two complex numbers are defined in Table 13.1, where the complex numbers z, z_1, and z_2 are defined as $x + iy, x_1 + iy_1$, and $x_2 + iy_2$, respectively.

The negation of a complex number will change the sign of both real and imaginary parts of the complex number. The addition of two complex numbers will add the real and imaginary components of two complex numbers, separately. The subtraction of two complex numbers will subtract the real and imaginary parts of the second complex number from the real and imaginary of the first complex number, respectively. Treating the imaginary number i as a complex number of complex(0, 1), the multiplication and division for two complex numbers are defined in Table 13.1. For binary operations with real and complex operands, the regular real operand will be cast into a complex before the operation. Complex numbers are not ordered; one cannot compare to see whether one complex number is larger or smaller than the other. But, two complex numbers can be tested whether they are equal or not. Two complex numbers are equal to each other if and

Table 13.2: Complex negation results

Negation −				
operand	complex(0.0, 0.0)	z	ComplexInf	ComplexNaN
result	complex(0.0, 0.0)	−z	ComplexInf	ComplexNaN

Table 13.3: Complex addition and subtraction results

Addition and Subtraction ±				
left operand	right operand			
	complex(0.0, 0.0)	z2	ComplexInf	ComplexNaN
complex(0.0, 0.0)	complex(0.0, 0.0)	±z2	ComplexInf	ComplexNaN
z1	z1	z1 ± z2	ComplexInf	ComplexNaN
ComplexInf	ComplexInf	ComplexInf	ComplexNaN	ComplexNaN
ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN

only if both the real and imaginary parts of two complex numbers are equal to each other, separately. If the real or imaginary parts of two complex numbers are not equal to each other, then the two complex numbers are not equal.

13.4.2 Complex Operations with Complex Metanumbers

In the above definitions of complex operations, we assume that all operands are regular complex numbers. The real and imaginary parts of a complex number are then treated as two regular floating-point floats. If the values of operands involve complex metanumbers, the definitions defined in Table 13.1 may not be valid. For example, ComplexInf is represented internally as complex(Inf, Inf). According to the complex addition definition defined in Table 13.1 and addition rule for real numbers in Ch, the result of addition of two ComplexInfs would be complex(Inf, Inf). But, addition of two complex infinities is mathematically indeterminate. Therefore, the results for arithmetic and relational operations with both regular complex numbers and complex metanumbers are defined in Tables 13.2 to 13.7.

From a programmer's point of view, values of complex($\pm 0.0, \pm 0.0$) are the same as complex(0.0, 0.0) when they are used as operands or arguments in Ch. In the following discussions, the positive zero 0.0 and the negative zero -0.0 for real and imaginary components of a complex number are considered the same. Therefore, although the negation of complex(0.0, 0.0) returns complex($-0.0, -0.0$), the result listed in Table 13.2 is complex(0.0, 0.0). Negation of a complex infinity is still a complex infinity. And of course, negation of a complex not-a-number is ComplexNaN.

For binary operations in Tables 13.3 to 13.5, if any one of the operands is ComplexNaN, the result is ComplexNaN. If one of two operands is ComplexInf and other is a finite complex number, the result of addition and subtraction is ComplexInf. Unlike real numbers, addition and subtraction of two ComplexInfs are ComplexNaNs. Multiplication of ComplexInf with complex(0.0, 0.0) is ComplexNaN; multiplication of ComplexInf with a finite nonzero number is ComplexInf; multiplication of two ComplexInfs becomes ComplexInf. Like real numbers, divisions of complex(0.0, 0.0) by complex(0.0, 0.0) and ComplexInf by ComplexInf are ComplexNaNs. A finite number or infinity divided by complex(0.0, 0.0) becomes Complex-Inf. The division of ComplexInf by a finite number gives ComplexInf. Theoretically, two complex infinities cannot be compared with each other because they may or may not be equal to each other. In Ch, however,

Table 13.4: Complex multiplication results

Multiplication ∗				
left operand	right operand			
	complex(0.0, 0.0)	z2	ComplexInf	ComplexNaN
complex(0.0, 0.0)	complex(0.0, 0.0)	complex(0.0, 0.0)	ComplexNaN	ComplexNaN
z1	complex(0.0, 0.0)	z1∗z2	ComplexInf	ComplexNaN
ComplexInf	ComplexNaN	ComplexInf	ComplexInf	ComplexNaN
ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN

Table 13.5: Complex division results

Division /				
left operand	right operand			
	complex(0.0, 0.0)	z2	ComplexInf	ComplexNaN
complex(0.0, 0.0)	ComplexNaN	complex(0.0, 0.0)	complex(0.0, 0.0)	ComplexNaN
z1	ComplexInf	z1/z2	complex(0.0, 0.0)	ComplexNaN
ComplexInf	ComplexInf	ComplexInf	ComplexNaN	ComplexNaN
ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN

Table 13.6: Complex equal comparison results

Equal comparison ==				
left operand	right operand			
	complex(0.0, 0.0)	z2	ComplexInf	ComplexNaN
complex(0.0, 0.0)	1	0	0	0
z1	0	z1 == z2	0	0
ComplexInf	0	0	1	0
ComplexNaN	0	0	0	0

Table 13.7: Complex not equal comparison results

Not equal comparison !=				
left operand	right operand			
	complex(0.0, 0.0)	z2	ComplexInf	ComplexNaN
complex(0.0, 0.0)	0	1	1	0
z1	1	z1 != z2	1	0
ComplexInf	1	1	0	0
ComplexNaN	0	0	0	0

two ComplexInfs are considered the same from the programming point of view as shown in Table 13.6. Like NaN in real number, the comparison of two ComplexNaNs will get a logic false. This design consideration is also reflected in the `not equal` relational operation shown in Table 13.7.

13.5 Complex Functions

Besides the polymorphism, the mathematical functions implemented in Ch can have a variable number of arguments, which is very convenient for calculations of complex mathematical functions with multiple branches. If a mathematical function, as a real function, has only one real argument, the additional second argument will render the function to a complex function unless explained otherwise. The integral value of the second argument will indicate the branch of the complex function. When this second argument is present, the first argument will be cast into a complex number according to the previously discussed data type conversion rules when the order of its data type is lower than complex. For a mathematical function with two arguments as a real function, if either one of two input arguments is a complex, the mathematical function becomes a complex function. If an additional third argument as a branch indicator is provided, the function becomes a complex function if data types of the first two arguments are lower than or equal to complex. If their data types are lower than complex, they will be cast into complex numbers.

13.5.1 Results of Complex Functions with Regular Complex Numbers

The built-in functions related to the complex numbers are listed in Table 13.8 along with their definitions. The input arguments of these functions can be complex numbers, variables, or expressions. For presentation purposes, the complex numbers z, z_1, and z_2 are defined as $x + iy, x_1 + iy_1$, and $x_2 + iy_2$, respectively. The integer values of k, k_1, and k_2 are the branch numbers of complex functions. If arguments for these branch numbers of the calling function are not integers, they will be cast into integers internally. For mathematical expressions in the second column in Table 13.8, if the arguments of mathematical functions are regular real numbers, the mathematical functions are real mathematical functions. The results of complex functions involving complex metanumbers will be discussed in the next section. In Table 13.8, the principal value Θ of the argument of a complex number is in the range of $-\pi < \Theta \leq \pi$. The definition of the principal value Θ for various complex numbers is given in Table 13.9. Note that the trigonometric function atan2(y,x) is in the range of $-\pi \leq \text{atan2}(y, x) \leq \pi$. Normally, through complex arithmetic and complex functions, one shall not get a complex number with its real or imaginary part being the value of $-$Inf, Inf, or NaN whereas the other part is a regular real number. This kind of result can be obtained only explicitly by functions **real**(z) and **imag**(z), and float pointer variables through lvalues, which will be discussed in Section 13.6.

The first four functions in Table 13.8 return real numbers. The **sizeof**() function returns, in bytes, an integer of the variable, type specifier, or expression that it precedes. the returned data type is of type unsigned int. If the argument is a complex, it will return the value of 8, which is the number of bytes required for storing two floats of real and imaginary parts of a complex. The **abs**(z) function computes the modulus of a complex number. Its returned data type is float if the input is float complex. Its returned data type is double if the input is double complex. When the input type is complex type, the function **fabs**(z) behaves the same as the function **abs**(z). The functions **real**(z) and **imag**(z) return the real and imaginary parts of a complex number, respectively. The results of **real**(z) and **imag**(z) are always floats. If the data type of the argument for **real**() is lower or equal to double, the input data will be cast into a float. If the data type of the argument for **imag**() is lower than or equal to double, the value of zero will be returned. The sign of a zero will be ignored in **real**(z) and **imag**(z) functions. For example, **real**(complex(-0.0,0.0)) will return 0.0.

A complex number can be created from two real numbers by the complex construction function **com-**

Table 13.8: The syntax and semantics of built-in complex functions

Ch Syntax	Ch Semantics
sizeof(z)	8
abs(z)	sqrt(x^2+y^2)
fabs(z)	sqrt(x^2+y^2)
real(z)	x
imag(z)	y
complex(x,y)	$x+iy$
conj(z)	$x-iy$
carg(z)	$\Theta;\ \Theta=\text{atan2}(y,x)$
polar(z)	$\text{sqrt}(x^2+y^2)+i\Theta;\ \Theta=\text{atan2}(y,x)$
polar($r,theta$)	$r\cos(theta)+ir\sin(theta)$
sqrt(z)	$\text{sqrt}(\text{sqrt}(x^2+y^2))(\cos\frac{\Theta}{2}+i\sin\frac{\Theta}{2});\ \Theta=\text{atan2}(y,x)$
sqrt(z,k)	$\text{sqrt}(\text{sqrt}(x^2+y^2))(\cos\frac{\Theta+2k\pi}{2}+i\sin\frac{\Theta+2k\pi}{2});\ \Theta=\text{atan2}(y,x)$
exp(z)	$e^x(\cos y+i\sin y)$
log(z)	$\log(\sqrt{x^2+y^2})+i\Theta;\ \Theta=\text{atan2}(y,x)$
log(z,k)	$\log(\sqrt{x^2+y^2})+i(\Theta+2k\pi);\ \Theta=\text{atan2}(y,x)$
log10(z)	$\frac{\log(z)}{\log(10)}$
log10(z,k)	$\frac{\log(z,k)}{\log(10)}$
pow(z_1,z_2)	${z_1}^{z_2}=e^{z_2\ln z_1}=\exp(z_2*\log(z_1))$
pow(z_1,z_2,k)	${z_1}^{z_2}=e^{z_2\ln z_1}=\exp(z_2*\log(z_1,k))$
ceil(z)	$\text{ceil}(x)+i\ \text{ceil}(y)$
floor(z)	$\text{floor}(x)+i\ \text{floor}(y)$
fmod($z1,z2$)	$z;\quad \frac{z_1}{z_2}=k+\frac{z}{z_2},\ k\geq 0$
modf($z1,\&z2$)	$\text{modf}(x_1,\&x_2)+i\ \text{modf}(y_1,\&y_2)$
frexp($z1,\&z2$)	$\text{frexp}(x_1,\&x_2)+i\ \text{frexp}(y_1,\&y_2)$
ldexp($z1,z2$)	$\text{ldexp}(x_1,x_2)+i\ \text{ldexp}(y_1,y_2)$
sin(z)	$\sin x\cosh y+i\cos x\sinh y$
cos(z)	$\cos x\cosh y-i\sin x\sinh y$
tan(z)	$\frac{\sin z}{\cos z}$
asin(z)	$-i\log(iz+\text{sqrt}(1-z^2))$
asin(z,k)	$-i\log(iz+\text{sqrt}(1-z^2,k))$
asin($z,k1,k2$)	$-i\log(iz+\text{sqrt}(1-z^2,k_1),k_2)$
acos(z)	$-i\log(z+i\text{sqrt}(1-z^2))$
acos(z,k)	$-i\log(z+i\text{sqrt}(1-z^2,k))$
acos($z,k1,k2$)	$-i\log(z+i\text{sqrt}(1-z^2,k_1),k_2)$
atan(z)	$\frac{1}{2i}\log(\frac{1+iz}{1-iz})$
atan(z,k)	$\frac{1}{2i}\log(\frac{1+iz}{1-iz},k)$
atan2($z1,z2$)	$\frac{1}{2i}\log(\frac{1+iz_1/z_2}{1-iz1_/z_2})$
atan2($z1,z2,k$)	$\frac{1}{2i}\log(\frac{1+iz_1/z_2}{1-iz1_/z_2},k)$
sinh(z)	$\sinh x\cos y+i\cosh x\sin y$
cosh(z)	$\cosh x\cos y+i\sinh x\sin y$
tanh(z)	$\frac{\sinh x\cos y+i\cosh x\sin y}{\cosh x\cos y+i\sinh x\sin y}$

continued on next page

Table 13.8: continued

Ch Syntax	Ch Semantics
$\mathrm{asinh}(z)$	$\log(z+\mathrm{sqrt}(z^2+1))$
$\mathrm{asinh}(z,k)$	$\log(z+\mathrm{sqrt}(z^2+1,k))$
$\mathrm{asinh}(z,k1,k2)$	$\log(z+\mathrm{sqrt}(z^2+1,k_1),k_2)$
$\mathrm{acosh}(z)$	$\log(z+\mathrm{sqrt}(z+1)\mathrm{sqrt}(z-1))$
$\mathrm{acosh}(z,k)$	$\log(z+\mathrm{sqrt}(z+1,k)\mathrm{sqrt}(z-1,k))$
$\mathrm{acosh}(z,k1,k2)$	$\log(z+\mathrm{sqrt}(z+1,k_1)\mathrm{sqrt}(z-1,k_1),k_2)$
$\mathrm{atanh}(z)$	$\frac{1}{2}\log(\frac{1+z}{1-z})$
$\mathrm{atanh}(z,k)$	$\frac{1}{2}\log(\frac{1+z}{1-z},k)$

Table 13.9: The principal value Θ $(-\pi < \Theta \leq \pi)$ of the argument for complex(x,y)

Θ						
y value	x value					
	$-$x1	-0.0	0.0	x2	Inf	NaN
y2	atan2$(y_2,-x_1)$	pi/2	pi/2	atan2(y_2,x_2)		
0.0	pi	0.0	0.0	0.0		
-0.0	pi	0.0	0.0	0.0		
$-$y1	atan2$(-y_1,-x_1)$	$-$pi/2	$-$pi/2	atan2$(-y_1,x_2)$		
Inf					Inf	
NaN						NaN

plex(x,y). If the input arguments are not floats, they will be cast into floats according to the internal data conversion rules. The sign of a zero for x or y will be carried over to the complex number.

The **conj**(z) function returns the complex conjugate $\overline{z}$ of z. The complex number $\overline{z}$ represented by the point $(x, -y)$ is the reflection in the real axis of the point (x, y) representing z.

The function **polar**() is implemented mainly for the convenience of transformation between Cartesian and polar representations of a complex number. If there is only one input argument, then a complex number with its real and imaginary parts being the modulus and argument, respectively, of the input complex number will be returned. If there are two input arguments, the complex number z in the polar form will be returned. The first and second input arguments are the modulus and argument of z, respectively. According to the definition $re^{i\theta}$ for the polar function, negative values for r are valid.

For the square root function **sqrt**(), whenever there are two arguments, the first argument is treated as a complex number. In case it is not a complex number and cannot be cast into a complex number, a syntax error message will be reported by the system. If the second argument is not an integer, it will be cast into an integral value according to internal data conversion rules. For the complex square root, there are only two distinct branches because of the periodic natures of the sine and cosine functions. In general, for taking the nth root, there are n distinct branches. If the function **sqrt**() is invoked with a single complex argument, the default branch value of 0 will be used.

The **exp**(z) function will calculate the exponential function of the complex number z.

Like the square root function, the natural logarithmic function **log**() has multiple branches. The branch number is provided by the second argument of the function. For convenience, the function **log10**() will calculate the base-ten logarithmic function of a complex value.

The exponential function with a complex base can be calculated by the function **pow**(), which is accomplished by the exponential function and logarithmic function as is shown in Table 13.8. The branch of the logarithmic function determines the branch of the function **pow**(). Unlike its corresponding real function, the complex function **pow**() is always well defined. If any one of two arguments of **pow**(z1, z2) is complex, the result is complex, which is obtained by the principal branch of the expression **exp**(z2$*$**log**(z1)). The result of the expression y^x equals the real part of the expression **pow**(complex(y,0.0), complex(x,0.0)) with its imaginary part being zero. For the function **pow**(z1, z2, k), z1 and z2 can be any data type lower than or equal to complex, and k is an integer. Whenever there are three arguments for the function **pow**(), the first and second arguments are treated as complex numbers. If z2 is an integer, all branches will have the same result; thus, the solution is unique.

For functions **ceil**(z), **floor**(z), and **ldexp**(z1, z2), the real and imaginary parts are treated as if they were two separate real functions. The functions **modf**(z1, &z2) and **frexp**(z1, &z2) are handled in the same manner. For these two functions, when the data type of the first arguments is complex, the data type of the second argument must be a pointer to complex. The **fmod**(z1,z2) function computes the complex remainder of z1/z2.

The complex trigonometric functions **sin**(z), **cos**(z), and **tan**(z) and complex hyperbolic functions **sinh**(z), **cosh**(z), and **tanh**(z) have unique values. However, the complex inverse trigonometric functions **asin**(z), **acos**(z), and **atan**(z) and complex inverse hyperbolic functions **asinh**(z), **acosh**(z), and **atanh**(z) have multiple branches for a given input complex value. The second argument of these inverse functions indicates the branch number. For functions **asin**(), **acos**(), **asinh**(), and **acosh**(), the second and third arguments specify the branches of the related square root and logarithmic functions, respectively. The function **atan2**() is implemented similar to the function **atan**().

13.5.2 Results of Complex Functions With Complex Metanumbers

Like complex arithmetic operations, the definition for regular complex functions may not be valid when the input arguments are complex metanumbers. The results of the built-in complex functions with complex

metanumbers as their input arguments are given in Table 13.10.

In Table 13.10, complex($\pm 0.0, \pm 0.0$) in Ch is treated as complex(0.0, 0.0). When the input argument of a function is ComplexNaN, the returned result is always ComplexNaN except for the function **sizeof**(). As shown in Figure 13.2, a complex infinity is different from the real infinities of $\pm\infty$. When either the real or imaginary part of a complex value is outside the range of the representable floating-point number, it becomes ComplexInf. Therefore, the absolute value of ComplexInf is a real number of Inf. The real and imaginary parts of ComplexInf are NaN. However, the conjugate of ComplexInf is still a complex infinity. The result of **polar**(complex(0.0,0.0)) is defined as complex(0.0,0.0) because the principal value Θ for complex(0.0, 0.0) equals 0.0 as defined in Table 13.9. The result of **polar**(ComplexInf) is defined as complex(Inf, Inf). Therefore, if z equals complex(0.0,0.0) or ComplexInf, the equality of z = **polar**(real(polar(z)), imag(polar(z))) will still be satisfied. Like a real function, the square root of ComplexInf is ComplexInf.

As a real function, **exp**(Inf) = Inf whereas **exp**($-$Inf) = 0.0. However, both values of $\pm$Inf become ComplexInf if they are cast into complex numbers. Therefore, the complex exponential function **exp**(z) is ComplexNaN when the input argument is ComplexInf. The complex logarithmic function **log**(z) with the input argument of complex(0.0,0.0) or ComplexInf will return ComplexInf. With complex metanumbers as their input arguments, the real and imaginary parts of functions **ceil**(z), **floor**(z), and **ldexp**(z1, z2) are handled equivalent to two individual real functions. Like real functions, the complex trigonometric functions **sin**(z), **cos**(z), and **tan**(z) are undefined when the input arguments are ComplexInfs. The irrational number π is not representable in a computer program. If we had the value of π, the expression of $\mathbf{tan}(k\pi + \pi/2)$ would return ComplexInf. Unlike real functions, the complex inverse trigonometric functions **asin**(z) and **acos**(z) return ComplexInfs when the input arguments are ComplexInfs. As an inverse function of **tan**(z), the function **atan**(z,k) has different branches when the first input value is ComplexInf. According to the definition, **atan**($\pm i$) equals ComplexInf. The results of the complex hyperbolic functions **sinh**(z), **cosh**(z), and **tanh**(z), and complex inverse hyperbolic functions **asinh**(z), **acosh**(z), and **atanh**(z) are implemented similar to those of complex trigonometric functions and complex inverse trigonometric functions.

The results of the complex construction function **complex**(x,y) are given in Table 13.11. For constructing a complex number, if either its real or imaginary part is NaN, the result is a complex Not-a-Number. Likewise, if either one is a value of $\pm\infty$, the result is ComplexInf. For the function **polar**(r, theta) shown in Table 13.12, when the modulus is infinitely large, the resultant complex number is ComplexInf even if the provided argument of a complex number is infinity, which is compatible with the result of **polar**(ComplexInf) = complex(Inf, Inf). This also follows the rule that, through complex arithmetic and complex functions, one shall not get a complex number, what is $-$Inf, Inf, or NaN for either the real or imaginary part while the other part is a regular real number. Like the exponential function **exp**(z), the function **pow**(z1,z2) is undefined when the second argument is ComplexInf as shown in Table 13.13.

When the imaginary part y2 of z2 is a finite value, the results of the function depend on the value of its real part x2 when the value of z1 is complex(0.0, 0.0) or ComplexInf. Like the real function, the following expressions **pow**(complex(0.0,0.0), complex(0.0,0.0)), **pow**(complex(0.0,0.0), complex(0.0,y2)), **pow**(ComplexInf, complex(0.0,0.0)), and **pow**(ComplexInf, complex(0.0,y2)) are ComplexNaN. Because **pow**(0.0, Inf) = 0.0 and **pow**(0.0, $-$Inf) = Inf, and both Inf and $-$Inf are considered as ComplexInf, **pow**(complex(0.0,0.0),ComplexInf) is defined as ComplexNaN. The results of function **fmod**(z1,z2) for complex metanumbers are given in Table 13.14.

13.6 Lvalues Related to Complex Numbers

As defined before, an lvalue is any object that occurs on the left hand side of an assignment statement. The valid lvalues related to complex numbers are listed in Table 13.15. The assignment operations `+=, -=, *=, /=`, as well as increment operation `++` and decrement operation `--` can be applied to all

Table 13.10: Results of complex functions for complex$(0.0, 0.0)$, ComplexInf, and ComplexNaN

function	z value and results		
	complex$(0.0, 0.0)$	ComplexInf	ComplexNaN
sizeof(z)	8	8	8
abs(z)	0.0	Inf	NaN
real(z)	0.0	NaN	NaN
imag(z)	0.0	NaN	NaN
conj(z)	complex$(0.0, 0.0)$	ComplexInf	ComplexNaN
polar(z)	complex$(0.0, 0.0)$	ComplexInf	ComplexNaN
sqrt(z)	complex$(0.0, 0.0)$	ComplexInf	ComplexNaN
exp(z)	complex$(1.0, 0.0)$	ComplexNaN	ComplexNaN
log(z)	ComplexInf	ComplexInf	ComplexNaN
log10(z)	ComplexInf	ComplexInf	ComplexNaN
ceil(z)	complex$(0.0, 0.0)$	ComplexInf	ComplexNaN
floor(z)	complex$(0.0, 0.0)$	ComplexInf	ComplexNaN
modf(z, &z2)	complex$(0.0, 0.0)$	complex$(0.0, 0.0)$	ComplexNaN
z2	complex$(0.0, 0.0)$	ComplexInf	ComplexNaN
frexp(z, &z2)	complex$(0.0, 0.0)$	ComplexInf	ComplexNaN
z2	complex$(0.0, 0.0)$	complex$(0.0, 0.0)$	ComplexNaN
ldexp(z, z2)	complex$(0.0, 0.0)$	ComplexInf	ComplexNaN
sin(z)	complex$(0.0, 0.0)$	ComplexNaN	ComplexNaN
cos(z)	complex$(1.0, 0.0)$	ComplexNaN	ComplexNaN
tan(z)	complex$(0.0, 0.0)$	ComplexNaN	ComplexNaN
	Note: tan(complex($\pi/2 + k * \pi, 0.0$)) = ComplexInf		
asin(z)	complex$(0.0, 0.0)$	ComplexInf	ComplexNaN
acos(z)	complex(pi/2, 0.0)	ComplexInf	ComplexNaN
atan(z)	complex$(0.0, 0.0)$	complex(pi/2, 0.0)	ComplexNaN
	Note: atan(complex($0.0, \pm 1.0$)) = ComplexInf;		
	atan(ComplexInf, k) = complex(pi/2 + k$*$pi, 0.0)		
sinh(z)	complex$(0.0, 0.0)$	ComplexNaN	ComplexNaN
cosh(z)	complex$(1.0, 0.0)$	ComplexNaN	ComplexNaN
tanh(z)	complex$(0.0, 0.0)$	ComplexNaN	ComplexNaN
	Note: tanh(complex($0.0, \pi/2 + k * \pi$)) = ComplexInf		
asinh(z)	complex$(0.0, 0.0)$	ComplexInf	ComplexNaN
acosh(z)	complex(0.0, pi/2)	ComplexInf	ComplexNaN
atanh(z)	complex$(0.0, 0.0)$	complex$(0.0$, pi/2)	ComplexNaN
	Note: atanh(complex($\pm 1.0, 0.0$)) = ComplexInf;		
	atanh(ComplexInf, k) = complex(0.0, pi/2 + k$*$pi)		

Table 13.11: Results of the function complex(x, y) for $0.0, \pm\infty$, and NaN

complex(x, y)						
x value	y value					
	−Inf	−y1	0.0	y2	Inf	NaN
Inf	ComplexInf	ComplexInf	ComplexInf	ComplexInf	ComplexInf	ComplexNaN
x2	ComplexInf	complex(x2, −y1)	complex(x2, 0.0)	complex(x2, y2)	ComplexInf	ComplexNaN
0.0	ComplexInf	complex(0.0, −y1)	complex(0.0, 0.0)	complex(0.0, y2)	ComplexInf	ComplexNaN
−x1	ComplexInf	complex(−x1, −y1)	complex(−x1, 0.0)	complex(−x1, y2)	ComplexInf	ComplexNaN
−Inf	ComplexInf	ComplexInf	ComplexInf	ComplexInf	ComplexInf	ComplexNaN
NaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN

Table 13.12: Results of the function polar(r, theta) for $0.0, \pm\infty$, and NaN

polar(r, theta)						
r value	theta value					
	−Inf	−theta1	0.0	theta2	Inf	NaN
Inf	ComplexInf	ComplexInf	ComplexInf	ComplexInf	ComplexInf	ComplexNaN
r2	ComplexNaN	polar(r2, −theta1)	complex(r2, 0.0)	polar(r2, theta2)	ComplexNaN	ComplexNaN
0.0	ComplexNaN	complex(0.0, 0.0)	complex(0.0, 0.0)	complex(0.0, 0.0)	ComplexNaN	ComplexNaN
−r1	ComplexNaN	polar(−r1, −theta1)	complex(−r1, 0.0)	polar(−r1, theta2)	ComplexNaN	ComplexNaN
−Inf	ComplexInf	ComplexInf	ComplexInf	ComplexInf	ComplexInf	ComplexNaN
NaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN

Table 13.13: Results of the function pow(z1, z2) for complex$(0.0, 0.0)$, ComplexInf, and ComplexNaN

pow(z1, z2)						
z1 value	z2 value					
	complex(0.0, 0.0)	z2; ($\lvert y2\rvert < \infty$)			ComplexInf	ComplexNaN
		$-\infty < x2 < 0.0$	x2 = 0.0	$0 < x2 < \infty$		
complex(0.0, 0.0)	ComplexNaN	ComplexInf	ComplexNaN	complex(0.0, 0.0)	ComplexNaN	ComplexNaN
z1	complex(1.0, 0.0)	$z_1^{z_2}$	$z_1^{z_2}$	$z_1^{z_2}$	ComplexNaN	ComplexNaN
ComplexInf	ComplexNaN	complex(0.0, 0.0)	ComplexNaN	ComplexInf	ComplexNaN	ComplexNaN
ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN

Table 13.14: Results of the function fmod(z1, z2) for complex$(0.0, 0.0)$, ComplexInf, and ComplexNaN

fmod(z1, z2)				
z1 value	z2 value			
	complex$(0.0, 0.0)$	z2	ComplexInf	ComplexNaN
complex$(0.0, 0.0)$	ComplexNaN	complex$(0.0, 0.0)$	complex$(0.0, 0.0)$	ComplexNaN
z1	ComplexNaN	fmod(z1,z2)	z1	ComplexNaN
ComplexInf	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN
ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN	ComplexNaN

Table 13.15: The valid lvalues related to complex numbers

Case	Meaning of lvalue	Example
1	simple variable	`z = complex(1.0, 2);`
2	an element of a complex array	`zarray[i] = complex(1.0,2)+ComplexInf;`
3	complex pointer variable	`zptr = malloc(sizeof(complex) * 3;` `zptr = &z;`
4	address pointed to by a complex variable	`*zptr = complex(1.0, 2) + z;`
5	an element of a complex pointer array	`zarrayptr[i] = malloc(sizeof(complex)*3;` `zarrayptr[i] = &z;`
6	address pointed to by an element of a complex pointer array	`*zarrayptr[i] = complex(1.0, 2);`
7	real part of a complex variable	`real(z) = 3.4;`
	real part of a complex variable	`real(*zptr) = 3.4;`
	real part of a complex variable	`real(*(zptr+1)) = 3.4;`
	real part of a complex variable	`real(*zarrayptr[i]) = 3.4;`
8	imaginary part of a complex variable	`imag(z) = complex(1.0, 2);`
	imaginary part of a complex variable	`imag(*zptr) = 3.4;`
	imaginary part of a complex variable	`imag(*(zptr+1)) = 3.4;`
	imaginary part of a complex variable	`imag(*zarrayptr[i]) = 3.4;`
9	float pointer variable	`fptr = &z;` `fptr = zptr;`
	pointer to real part of a complex variable	`*fptr = 1.0;`
	pointer to imaginary part of a complex variable	`*(fptr+1) = 2.0;`

these lvalues. Besides the simple variable in case 1, an element of a complex array can be an lvalue, which is case 2 in Table 13.15. In case 3, pointer to complex is used as an lvalue to get the memory or to point to a memory of a complex object. In case 4, the memory pointed to by the pointer $zptr$ is assigned the value of the expression on the right hand side of an assignment statement. In addition to a single pointer variable, one can have an array of complex pointers. Cases 5 and 6 show how an element of a complex pointer array is used to access its memory. The function **real**() can not only be used as an rvalue or an operand, but it can also be used as an lvalue to access the memory of its argument. In case 7, the argument of **real**() must be a complex variable, or an address pointed to by a complex pointer or pointer expression. A constant complex number or expression can be used as an input argument of the function **real**() only when it is an rvalue or an operand. In case 8, the imaginary part of a complex is accessed by the function **imag**() in the same manner as the function **real**(). Because a complex number occupies two floats internally, this memory storage can be accessed not only by the functions **real**() and **imag**(), but also by a pointer to float as is shown in case 9 where the variable $fptr$ is a pointer to float. For cases 7-9, a real number, including $\pm 0.0, \pm \text{Inf}$, and NaN, on the right hand side will be assigned to an lvalue formally without filtering. Therefore, abnormal complex numbers such as complex(Inf, NaN), complex(Inf,0.0), etc. may be created. For example, two Ch commands **real**(z) = NaN and **imag**(z) = Inf make z equal to complex(NaN,Inf); and **real**(z) = -0.0 and **imag**(z) = NZero gives z the value of complex($-0.0, -0.0$).

13.7 Creation of User's Complex Functions

User's complex functions in Ch can be created in the spirit of ISO C, which will be demonstrated by the computation of the following complex function $f(z_1, z_2)$.

$$\mathbf{f}(z_1, z_2) = \frac{(4z_1 + 3 + i5) * \sin(z_1 * z_2) * e^{i2.5}}{z_1(z_2 - 2 - i2)} \tag{13.3}$$

The complex function $f(z_1, z_2)$ can be easily programmed in Ch as follows.

```
complex f(complex z1, complex z2) {
  complex z;
  z = (4*z1+3+complex(0,5))*sin(z1*z2)*polar(1, 2.5))/
      (z1*(z2-complex(2,2)));
  return z;
}
```

Using the above-programmed external gamma function, the commands

```
printf("f(0, 0) = %f \n", f(0, 0));
printf("f(0, 1) = %f \n", f(0, 1));
printf("f(1, 1) = %f \n", f(1, 1));
printf("f(0, complex(2, 2)) = %f \n", f(0, complex(2,2)));
printf("f(1, complex(2, 2)) = %f \n", f(1, complex(2,2)));
```

will produce the following output.

f(0, 0) = CcomplexNaN
f(0, 1) = ComplexNaN
f(1, 1) = complex(1.385598,-2.925680)
f(0, complex(2, 2)) = ComplexInf
f(1, complex(2, 2)) = ComplexInf

Note that the function $f(z_1, z_2)$ gets ComplexInf at the singular point $z_2 = 2 + i2$ and $f(0, z_2)$ becomes division of complex zero by complex zero.

Chapter 14

Pointers and Arrays

Arrays are commonly used programming features. An *array* consists of elements that extend in one or more dimensions to represent columns, planes, cubes, etc. The number of dimensions in an array is referred to as the *rank* of the array, the number of elements in a dimension is called the *extent* of the array in that dimension. The *shape* of an array is a vector where each element of the vector is the extent in the corresponding dimension of the array. The *size* of an array is the number of bytes used to store the total number of elements of the array.

This chapter will first illustrate how pointers can be used to access elements of arrays. Next, we will describe how to allocate memory for one- and two-dimensional arrays. From a mathematical point of view, these two kinds of arrays are very useful to represent vectors and matrices. Then mechanisms for passing arrays to functions in C90 are described. It shows that passing multi-dimensional arrays of variable length to functions in the C90 standard conforming manner is cumbersome and error-prone. Variable length arrays described in the next chapter are recommended for applications.

14.1 Accessing Array Elements Through Pointers

Arrays and pointers are intimately tired. Not only a pointer can be used to access an array, but also the variable name of an array itself can be treated as a pointer. Assume that `A1` is a one-dimensional array of int type with length of 10 and `p` is a pointer to int, elements of `A1` can be accessed by three methods illustrated in the interactive execution in a Ch shell as follows.

```
> int i
> int A1[10], *p
> A1[3]=3      // method 1
> for(i=0; i<10; i++) printf("%d ", A1[i])
0 0 0 3 0 0 0 0 0 0
> *(A1+4)=4    // <==> A1[4]=4, method 2
> for(i=0; i<10; i++) printf("%d ", A1[i])
0 0 0 3 4 0 0 0 0 0
> p = A1
> *(p+5)=5     // <==> A1[5]=5, method 3
> for(i=0; i<10; i++) printf("%d ", A1[i])
0 0 0 3 4 5 0 0 0 0
>
```

According to the pointer arithmetic described in Chapter 9, `p+5` points to the sixth element of `A1`. The variable name `A1` in statement

```
*(A1+4)=4
```

is actually treated as a pointer to int.

For two- or multiple-dimensional arrays, the variable name of an array is treated as a pointer to array. For example, if `A2` is a two-dimensional array of int type with size (3×4) and `p` is a pointer to int, methods of accessing elements of `A2` are shown below.

```
> int i, j
> int A2[3][4], *p;
> A2[1][1]=3      // method 1
> for(i=0; i<3; i++) for(j=0; j<4; j++) printf("%d ", A2[i][j])
0 0 0 0 0 3 0 0 0 0 0 0
> p = A2
> *(p+1*4+2)=4    // <==>A2[1][2]=4, method 2
> for(i=0; i<3; i++) for(j=0; j<4; j++) printf("%d ", A2[i][j])
0 0 0 0 0 3 4 0 0 0 0 0
> *(*(A2+1)+3)=5  // <==>A2[1][3]=5, method 3
> for(i=0; i<3; i++) for(j=0; j<4; j++) printf("%d ", A2[i][j])
0 0 0 0 0 3 4 5 0 0 0 0
>
```

The value of `p+1*4+2` points to the address of the seventh element of array `A2` at `A2[1][2]`. Variable `A2` is a pointer to array of 4 elements. The point expression `(A2+1)` gives the address at the fifth element of array `A2`. Therefore, `*(*(A2+1)+3)` is equivalent to the array element `A2[1][3]`.

14.2 Dynamic Allocation of Arrays

In many applications, especially in engineering and science, the size of an array or a matrix is known only at the program execution. It will be more efficient to use dynamic allocation of arrays instead of declaring arrays of large fixed size. In this section, we will describe how to implement dynamic allocation of one- and two-dimensional arrays. The standard functions **malloc()**, **calloc()**, and **realloc()** described in Chapter 9 can be used to dynamically allocate memory for arrays. Function **free()** can be called to deallocate the dynamically allocated memory when they are no longer needed.

14.2.1 Dynamic Allocation of One-Dimensional Arrays

A one-dimensional array is typically used to represent a vector. For example, a row vector is a $(1 \times n)$ matrix, and a column vector is a $(n \times 1)$ matrix, where n is a positive integral value. Both of them can be written in the form of one-dimensional array.

As an example, assume vector $\mathbf{A}1$ is of double type and its length `vectLen` is determined at runtime. The following code fragment shows how to allocate memory dynamically for `A1` with function **malloc**().

```
double *A1;
/* ... source code to obtain vectLen at runtime */
A1 = (double *)malloc(vectLen*sizeof(double));
if(A1 == NULL) {
  fprintf(stderr, "ERROR: %s(): no enough memory\n", __func__);
```

```
  exit(1);
}
/* ... source code to handle vector A1 */
free(A1);
/* ... source code no longer use A1 */
```

Variable `A1` is declared as a pointer to double. After obtaining the value for `vectLen` at run time, the memory with `vectLen * sizeof(double)` bytes is allocated dynamically. `A1` can be treated as a one-dimensional array, and its elements can be accessed through its name `A1`.

```
> int i
> double *A1
> A1 = (double *)malloc(10*sizeof(double))
40070280
> A1[5] = 10     // method 1
> for(i=0; i<10; i++) printf("%1.1f ", A1[i])
0.0 0.0 0.0 0.0 0.0 10.0 0.0 0.0 0.0 0.0
> *(A1+6) = 20;  // <==> A1[6]=20, method 2
0.0 0.0 0.0 0.0 0.0 10.0 20.0 0.0 0.0 0.0
> p = A1
40070280
```

Two different methods to access an element of the array are presented in the above example. For array `A1` with `vectLen` number of elements, the subscript `i` in the form of `A1[i]` or `*(A1+i)` can go from 0 to `vectLen-1`. An attempt to access element `A1[vectLen]` is illegal, because it points to the memory outside the boundary for array `A1`.

A one-dimensional array can also be used to represent a two-dimensional matrix. For example, pointer `p` can allocate a memory for a two-dimensional array of size $(n \times m)$. The element `(i, j)` of matrix $(n \times m)$ can be accessed by pointer indirection operation `*(p+i*m+j)`. In the example below, `n` is 3 and `m` is 4.

```
> int i
> double *A1, *p
> A1 = (double *)malloc(3*4*sizeof(double))
40070280
> p = A1
40070280
> *(p+1*4+2) = 30;   // assigned 30 to element (1,2)
>
```

14.2.2 Dynamic Allocation of Two-Dimensional Arrays

From a mathematical point of view, a matrix or an array of $(m \times n)$ is a set of numbers arranged in a rectangular block of `m` horizontal rows and `n` vertical columns. From a programming point of view, it is a block of memory with each row of the matrix lying in a contiguous block of memory. Two methods of dynamical allocation of two-dimensional arrays will be presented in this section. In the first method, the memory for the matrix allocated is in a single contiguous block, whereas in the second method, each row of the matrix lies in a contiguous block of memory, but the whole matrix is not in a sequential memory block.

Assume `A2` is a two-dimensional array of size $(m \times n)$. The values of `m` and `n` are determined at run time. The following code fragment shows how to allocate a single contiguous memory dynamically for array `A2[m][n]`.

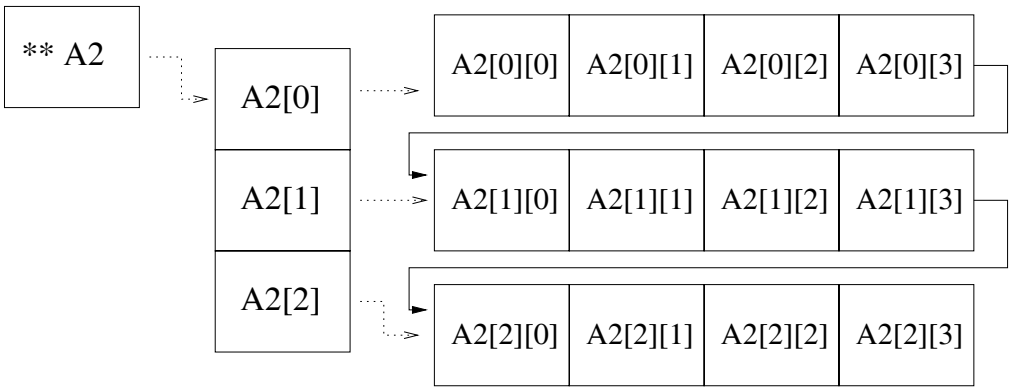


Figure 14.1: Dynamic allocation of two-dimensional arrays (method 1).

```
int i;
double **A2;
/* ... source code to obtain m and n at runtime */
A2 = (double **)malloc(m * sizeof(double*));
if(A2 == NULL) {
  fprintf(stderr, "ERROR: %s(): no enough memory\n", __func__);
  exit(1);
}
A2[0] = (double *)malloc(m * n * sizeof(double));
if(A2[0] == NULL) {
  fprintf(stderr, "ERROR: %s(): no enough memory\n", __func__);
  exit(1);
}
for(i = 1; i < m; i++) {
  A2[i] = A2[0] + i * n;
}
/* ... source code to handle vector A2 */
free(A2[0]);
free(A2);
/* ... source code no longer use A2 */
```

Frist, `A2` is declared as a pointer to pointer to double. After obtaining the values of `m` and `n`, the memory for `m` pointers to double is allocated for `A2`. So, `A2` can be considered as a one-dimensional array of pointers with `m` elements shown in Figure 14.1. Then a single contiguous memory with `m * n * sizeof(double)` bytes for `m * n` elements is allocated. Each pointer `A[i]` points to a segment in this block. Therefore, `A2` becomes a two-dimensional array. In the example below, elements of array of size (3×4) are accessed by two different methods.

```
> int i, j
> double **A2
> A2 = (double **)malloc(3 * sizeof(double*)); // with size (3 X 4)
4006cdc0
> for(i=0; i<3; i++) printf("%p ", A2[i])
00000000 00000000 00000000
> A2[0] = (double *)malloc(3 * 4 * sizeof(double));
```

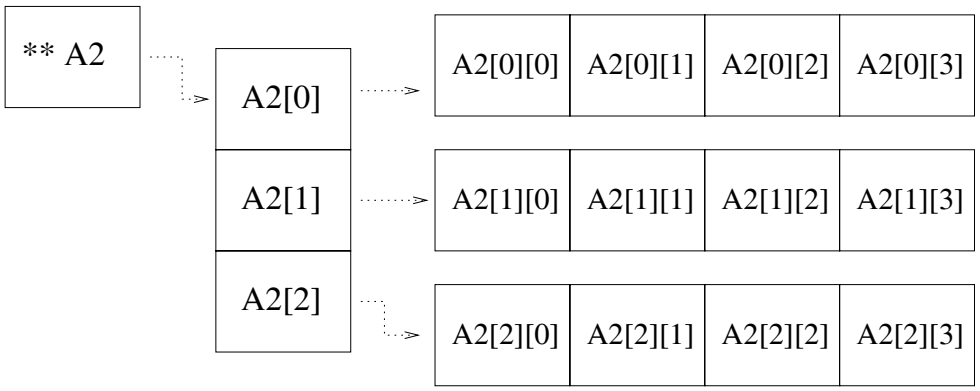


Figure 14.2: Dynamic allocation of two-dimensional arrays (method 2).

```
> A2[1] = A2[0] + 4                          // A2[i] = A2[0] + i * n
> A2[2] = A2[0] + 2*4                        // A2[i] = A2[0] + i * n
> for(i=0; i<3; i++) printf("%p ", A2[i])   // 1-dimension of pointer
4007bee8 4007bf08 4007bf28
> for(i=0; i<3; i++) for(j=0; j<4; j++) printf("%1.1f ", A2[i][j])
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
> A2[1][1]=3           // method 1
> *(*(A2+1)+2)=4       // <==> A2[1][2]=4, method 2
> for(i=0; i<3; i++) for(j=0; j<4; j++) printf("%1.1f ", A2[i][j])
0.0 0.0 0.0 0.0 0.0 3.0 4.0 0.0 0.0 0.0 0.0 0.0
> pp = A2
4006cdc0
```

Once a memory block with `m * n * sizeof(double)` bytes is allocated for `A2`, the subscripts `i` and `j` in the form of `A2[i][j]` or `*(*(A2+i)+j)` can go from 0 to `m-1` and 0 to `n-1`, respectively. Any attempt of access to `A2[m][n]` is illegal.

The following code fragment shows the second method for dynamic allocation of two-dimensional array `A2`. Memory for each row of the matrix is allocated separately. Therefore, the memory block for different rows may not be continguous. The memory layout for array `A2` is shown in Figure 14.2.

```
int i;
double **A2;
/* ... source code to obtain m and n at runtime */
A2 = (double **)malloc(m * sizeof(double*));
if(A2 == NULL) {
  fprintf(stderr, "ERROR: %s(): no enough memory\n", __func__);
  exit(1);
}
for(i = 0; i < m; i++) {
  A2[i] = (double *)malloc(n * sizeof(double));
  if(A2[i] == NULL) {
    fprintf(stderr, "ERROR: %s(): no enough memory\n", __func__);
    exit(1);
  }
}
```

```
int main() {
  double d1[10], d2[10];
  double d3[5], d4[5];
  void oneDadd(double dd1[], double *dd2, int n);
  oneDadd(d1, d2, 10);
  oneDadd(d3, d4, 5);
}

void oneDadd(double dd1[], double *dd2, int n) {
  int i;
  for(i=0; i<=n-1; i++)
    dd1[i] += dd2[i]; /* the same as *(dd1+i) += *(dd2+i) */
}
```

Program 14.1: Passing one-dimensional arrays of variable length to a function.

```
  /* ... source code to handle vector A2 */
  for(i = 0; i < m; i++)
    free(A2[i]);
  free(A2);
  /* ... source code no longer use A2 */
```

The major difference between the two methods for dynamical allocation of two-dimensional arrays described in this section is that function **malloc**() is called for each row of the matrix in the latter.

14.3 Passing One and Multi-Dimensional Arrays of Fixed Length

14.3.1 One-Dimensional Arrays

When an array is passed to a function, what is actually passed is only the address of the first element of the array. In the called function, this argument is a local variable of pointer type. Program 14.1 adds one-dimensional arrays `d1` and `d2`, each 10 elements of double data type, and stores the result in array `d1` element-wise by the function `oneDadd()`. In the function definition of `void oneDadd(double dd1[], *dd2, int n)` in Program 14.1, `dd1` is defined as a variable of double array type whereas `dd2` is a variable of pointer to double. One-dimensional arrays of variable length can be passed to this function as shown in the program. The extent of both arrays `d1` and `d2` is 10 whereas the extent of arrays `d3` and `d4` is 5. In general, the size of an array is not available to the called function, the sizes of arrays in the function `oneDadd()` are passed in by the parameter `n` in the program. However, a string is a special case. Strings are zero-terminated so that their sizes can be computed easily by the function **strlen**(). The expressions of `dd1[-2]` and `dd2[20]` are equivalent to `*(dd1-2)` and `*(dd2+20)`, respectively. If they were used inside the function `oneDadd()` in Program 14.1, they would refer to objects outside the passed array bounds. Because no extents are specified in the declaration of formal arrays, the statement of `dd1[-2] += dd2[20+n]` would be syntactically legal if they were in the function `oneDadd()`, even if they may be problematic. It is not possible to generate any warning or error message for this kind of statement. It is the programmer's responsibility to make sure that each element of the formal array in the called function is within the array bounds of actual arrays in the calling function.

However, if the extents of arrays to be passed in the calling functions are known, the formal array arguments in the function can be declared with specified extents. For example, in Program 14.2, both `dd1`

```
int main() {
  double d1[10], d2[10];
  double d3[5], d4[5];
  void oneDadd(double  dd1[10], double dd2[10], int n);
  oneDadd(d1, d2, 10);   /* OK */
  oneDadd(d3, d4, 5);    /* WARNING: incompatible dimensions */
}

void oneDadd(double  dd1[10], double dd2[10], int n) {
  int i;
  for(i = 0; i <=n-1; i++)
    dd1[i] += dd2[i];
}
```

Program 14.2: Passing one-dimensional arrays of fixed length to a function.

and `dd2` are defined as variables of array of 10 elements. When it is invoked by the function call of `oneDadd(d3, d4, 5)`, two warning messages may be produced by the system because the extents of passed arrays `d3` and `d4` do not match with those of the formal definitions for `dd1` and `dd2`. Furthermore, because the extent has been specified in the formal argument, error messages would be generated in Ch at the runtime due to the array boundary error if the statement of `dd1[-2] += dd2[20]` were used in the function `oneDadd()`. To avoid a likely crash of the system, if an array index is smaller than zero, the lower bound of zero will be used as the array index in Ch. Similarly, if an index is greater than the upper bound of the formal array, the upper bound of the formal array will be used as the index. The assignment statement `dd1[-2] += dd2[20]` would be, therefore, handled as the statement `dd1[0] += dd2[9]`. Although `dd1[10]` and `dd2[10]` in the function definition `oneDadd(double dd1[10], dd2[10], int n)` are declared with specific extents, what is passed to the called function are still only pointers.

14.3.2 Multi-Dimensional Arrays of Fixed Length

One-dimensional arrays can be passed to functions conveniently in C as described in the previous section. In this section, we will describe how to pass multi-dimensional arrays of fixed shape to functions.

Let us examine the example of passing two-dimensional arrays to a function in Program 14.3 when the function `twoDadd()` is used to add two two-dimensional arrays. If a two-dimensional array of fixed length is to be passed to a function, the parameters of the array definition in the arguments of the function should include the number of columns, the second dimension of the array. We illustrate three different formats for formal array arguments of a function by using the following declaration in Program 14.3.

```
void twoDadd(double (*dd1)[5], double dd2[][5], double dd3[4][5],
             int n, int m)
```

Although the declaration formats for `dd1, dd2` and `dd3` are different, all of them are defined as the *pointer to array* of 5 elements of double data type. Unlike one-dimensional arrays, the internal data structures for variables `dd1` and `dd2` are identical, they both ignore the number of rows of the actual arrays. However, if the number of rows of the argument in the calling function corresponding to the variable `dd3` is not 4, the system may generate a warning message. In general, in passing arrays to functions, the rank of actual arrays shall match with the rank of the formal array argument in the function definition and function prototypes. Otherwise, the system may produce a warning message. If the extent for a dimension of an

```
int main() {
  double d1[4][5], d2[4][5], d3[4][5];
  double d4[3][5], d5[3][5], d6[3][5];
  double d7[4][6];
  void twoDadd(double (*dd1)[5], double dd2[][5], double dd3[4][5],
               int n, int m);
  twoDadd(d1, d2, d3, 4, 5); /* OK */
  twoDadd(d4, d5, d6, 3, 5); /* WARNING: incompatible first dimension
                                d6[3][5] != dd3[4][5] */
  twoDadd(d7, d2, d3, 4, 5); /* WARNING: incompatible second dimension
                                d7[4][6] != (*dd1)[5] */
}

void twoDadd(double (*dd1)[5], double dd2[][5], double dd3[4][5],
             int n, int m) {
  /* dd1[n][m] = dd2[n][m] + dd3[n][m] */
  int i, j;

  for(i=0; i<=n-1; i++)
    for(j=0; j<=m-1; j++)
      dd1[i][j] = dd2[i][j]+dd3[i][j];
}
```

Program 14.3: Passing two-dimensional arrays of fixed length to a function.

```
int main() {
  double d1[3][5][7], d2[3][5][7], d3[3][5][7];
  void threeDadd(double (*dd1)[5][7], double dd2[][5][7],
                 double dd3[3][5][7], int n, int m, int r);
  threeDadd(d1, d2, d3, 3, 5, 7);   /* d1 = d2 + d3 */
}

void threeDadd(double (*dd1)[5][7], double dd2[][5][7],
               double dd3[3][5][7], int n, int m, int r) {
  /* dd1[n][m][r] = dd2[n][m][r] + dd3[n][m][r] */
  int i, j, k;

  for(i=0; i<=n-1; i++)
    for(j=0; j<=m-1; j++)
      for(k=0; k<=r-1; k++)
        dd1[i][j][k] = dd2[i][j][k]+dd3[i][j][k];
}
```

Program 14.4: Passing three-dimensional arrays of fixed length to a function.

array is given in the function definition or function prototypes, the extent of the actual array shall match with the corresponding extent of the array in the formal definition. Otherwise, the system may also produce a warning message. The shapes of the array in the function definition and its function prototypes shall be the same. Otherwise, it is a syntax error. The exception is declaring a one-dimensional array of variable length. For example, the following two function prototypes

```
void funct(double *d);
void funct(double d[]);
```

are considered to be compatible. The function call of `twoDadd(d4, d5, d6, 3, 5)` in Program 14.3 may get a warning message for array `d6[3][5]` which has a different extent for the first row dimension from the formal argument `dd3`. A warning message may also be generated for array `d7[4][6]` because the number of columns is different from the formal definition. Arrays `d1, d2, d4` and `d5` with different numbers of rows can be passed to the function `twoDadd()` correctly without any warning message.

Multi-dimensional arrays higher than two-dimension can be handled in the same manner. In general, only the size of the first dimension of an array can vary; all others shall be specified in the function definition and function prototypes. Program 14.4 demonstrates how to pass three-dimensional arrays to a function in three different syntactical forms for array arguments. In Program 14.4, the function call of `threeDadd(d1, d2, d3, 4, 5, 6)` stores the sum of each elements of arrays `d2` and `d3` to the corresponding element of array `dd1`.

Because an array parameter in a function is handled as a pointer to array, a pointer to array can also be used as an actual argument of a function. This can be illustrated by Program 14.5. In Program 14.5, the variable `dp` is a pointer to array of 7 elements and `d[1]` is an array of 5x7 elements. The assignment statement `dp = d[1]` points `dp` at the starting address of the 5x7 array so that expressions `dp[i][j]` and `d[1][i][j]` will refer to the same object. In Program 14.5, statement `dp = d[1]` can be substituted by one of the following assignment statements

```
void funct(double dd[][7]){ }
int main() {
  double d[3][5][7], (*dp)[7], (*dp2)[7];
  dp = d[1];              /* dp = &d[1][0][0]; dp = d+1; dp = *(d+1); */
  funct(dp);              /* funct(&d[1][0][0]); funct(d[1]); */
                          /* funct(d+1); funct(*(d+1)); */
  dp[2][3] = dp[3][5]+6;  /* treat dp as an array */
  dp2 = malloc(sizeof(double)*5*7);
  funct(dp2);
}
```

Program 14.5: Using pointer and pointer to array as actual arguments in passing arrays to a function.

```
dp = &d[1][0][0];
dp = d+1;
dp = *(d+1);
```

where `&d[1][0][0]` is the address of the first element `d[1][0][0]` of array `d`, `d+1` is a pointer to array of 5x7 elements, and `*(d+1)` is an array of 5x7 elements from `d[1][0][0]` to `d[1][4][6]`. Similarly, if the statement `funct(dp)` is replaced by any one of the following programming statements

```
funct(&d[1][0][0]);
funct(d[1]);
funct(d+1);
funct(*(d+1));
```

the result will be the same as long as the element `dd[i][j]` referenced inside function `funct()` is within the valid range of array `d` in the main routine.

An array consists of a continuous segment of memory in C. The memory pointed at by a pointer to array can be allocated dynamically by memory allocation functions **malloc**(), **calloc**(), and **realloc**(). Using the indirection of pointers, a pointer to array of n dimensions can be treated as an array of (n+1) dimensions. This can be illustrated by variable `dp2` in Program 14.5. The variable `dp2` is a pointer to array of 7 elements of double data type. The memory for `dp2` is allocated dynamically and it is passed to the function `funct()` in the same manner as `dp`.

Similarly, arrays of different data type can be passed to functions. For example, the following code fragment shows how arrays of pointers of different data type are passed to the function `funct()`.

```
char *c[]={"strings", "with different", "length", ""};;
int **i[2][4];
float ***f[3][5][7];
int funct(char *cc[], int **ii[2][4], float ***ff[3][5][7]);
funct(c, i, f);
```

where `c` is an array of pointer to char, `i` is an array of double pointer to 2x4 int elements, and `f` is an array of triple pointer to 3x5x7 float elements. Note that arrays of pointer to char are useful for system programming because it can store strings of variable length.

Chapter 15

Variable Length Arrays

Arrays in C are intimately tied with pointers. Treating array variables as pointers in C is very elegant for system programming; it is one of C's major strengths. Because C was not designed for numerical computing, handling multi-dimensional arrays in C is cumbersome in many situations. For example, in contrary to the fame of C for its conciseness and clarity, passing arrays of variable length to functions in C90 is neither intuitive nor easy to understand; it is very complicated. Arrays of variable length were available in FORTRAN since its earlist days [1]. Scientific programmers with prior FORTRAN experiences are often disappointed at C's inability to handle arrays of variable length. Adding variable length arrays (VLA) to C is a critical step in evolving C as a leading programming language for applications in science and engineering. Arrays of variable length whose size is known only at program execution time are added in C99 and Ch. Details about variable length arrays are described in this chapter.

Four different types of variable length arrays called deferred-shape arrays, assumed-shape arrays, pointer to assumed-shape arrays, and array of reference in Ch can be illustrated by the following code fragment.

```
int funct(int a[&][&], int (*b)[:], int c[:][:], int n, int m) {
  int (*d)[:] = a;
  int e[n][m];
}
```

where `a` is an array of reference; `b` and `c` are assumed-shape array; `d` is a pointer to assumed-shape array; and `e` is a deferred-shape array. Ch extends C with arrays of explicit lower and upper bounds for numerical computing. Arrays of explicit lower and upper bounds can be illustrated by the following code fragment.

```
int funct(int a[1:5], int b[1:][1:], int n, int m) {
  int c[3][1:3];
  int d[n:m];
}
```

where `a` is a fixed-length array with the subscript range from 1 to 5, `b` is a pointer to two-dimensional assumed-shape array with unit-offset. The subscript range of the first dimension of two-dimensional array `c` with fixed subscript range is from 0 to 2 while the subscript range of the second dimension of array `c` is from 1 to 3. The variable `d` is an array with deferred subscript range and it is also a deferred-shape array. The first element of an array in C starts with index zero such as `a[0]`. In contrast to C, the first element of an array in FORTRAN 77 starts with index one such as `a[1]`. An array with explicit lower and upper bounds allows an array element to start with any index number. Arrays with explicit lower and upper bounds have many applications. For example, the coefficients a_i of a polynomial $a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$ can be appropriately handled using a zero-offset array `a[0:n]` whereas a vector of N data points x_i for $i = 1$ to N calls for a unit-offset array `x[1:N]`.

15.1 Storage Duration and Declaration of Arrays

15.1.1 Storage Duration of Objects

Storage duration determines the lifetime of an object. An array declared with external or internal linkage, or with the storage-class specifier `static` has *static storage duration*. For such an array, its storage is reserved and its stored value for each element is initialized once only. Each element of the array exists and retains its last-stored value throughout the execution of the entire program. The shape of the array with static storage duration has to be resolved before the execution of the function `main()`. Therefore, each extent of an array definition with static storage duration shall be an integral constant expression with a value greater than zero as shown in the following sample program.

```
int n = 5;
int a[4][5], aa[3] = {1,2,3};
extern int b[6][7], c[8], d[][9], e[]; /* d and e are incomplete */
/* complete shape for d and e in the following definition */
int  b[6][7], c[8], d[4][9], e[10], ee[2][3] = {1,2,3,4,5,6};
int main(){
  static int s[4], ss[2+3] ={1,2,3,4,5};
  extern int a[4][5];
  extern int b[6][7], c[8], d[][9], e[];
}
```

An array declared with no linkage and without the storage class specifier `static` within a function or nested function has *automatic storage duration*. Storage is guaranteed to be reserved for a new instance of such an array on each normal entry into the block with which it is associated. If an initialization is specified for the array with automatic storage duration, it is performed on each normal entry. Storage for the array is no longer guaranteed to be reserved when execution of the enlosing block ends in any way including by means of `goto, continue, break,` and `return` statements. For example, arrays in the following code fragment have automatic storage duration.

```
int n = 5;
void funct1(){
  int m = 6;
  int a[4][5], aa[3] = {1,n,m};     /* n==5, m==6 */
  void funct2(){
    int s[4], ss[2+3] ={1,2,3,n,m}; /* n==5, m==6 */
  }
}
```

In this sample code, the shape of an array is completely specified by constant integral expressions for each extent. Since memory space for an array of automatic storage duration is allocated at execution time upon the entry to the block within which it is declared, it is desirable that the length of the array can be different when the block or function is invoked each time. An array whose size is determined at program execution time is called a *variable length array* (VLA).

15.1.2 Declaration of Arrays

Variables can be declared in the form of

```
T D1
```
(15.1)

where T contains the declaration specifiers that specify a type T such as `int` and D1 is a declarator that contains an identifier *ident*. The delimiters [and] may delimit an expression or : for declaration of arrays.

If D1 has the form

$$\mathrm{D}[expression_{opt}] \tag{15.2}$$

and the type specified for identifier in the declaration "T D1" is "*derived-declarator-type-list T*," then the type specified for *ident* is "*derived-declarator-type-list array of T*." If the size *expression* is not present, the array type is an incomplete type. If delimiters [and] delimit an expression that specifies the size of an array, it shall be an integral type. If it is a constant integral expression, it shall have a value greater than zero. If the size expression of an array is not a constant expression, it is evaluated at program execution time and shall evaluate to a value greater than zero. The array type is a *deferred-shape array* type. If the size expression is an integral constant expression and the element type has a fixed size, the array type is a *fixed-length array* type.

If D1 has the form

$$\mathrm{D}[:] \tag{15.3}$$

and the type specified for identifier in the declaration "T D1" is "*derived-declarator-type-list T*," then the type specified for *ident* is "*derived-declarator-type-list assumed-shape array of T*." The array is called *assumed-shape array* type. The shape of the array is assumed at program execution time.

An array with specified lower bounds shall be declared in one of the following two forms:

```
T D[lower:upper]
```
(15.4)

```
T D[lower:]
```
(15.5)

(15.6)

where `T` contains the declaration specifiers that specify a type such as `int`, `D` is a declarator that contains an identifier *ident*, `lower` is the lower bound of the array, and `upper` is the upper bound. The expressions `lower` and `upper` shall be of integral type.

A pointer to array of fixed-length shall be declared as

```
T (*D)[expr]
```
(15.7)

where `T` contains the declaration specifiers that specify a type and `D` is a declarator that contains an identifier *ident*. The expression `expr` shall be constant integral type. A pointer to array of assumed-shape shall be declared in one of the following two forms:

```
T (*D)[:]
```
(15.8)

```
T (*D)[lower:]
```
(15.9)

The expression `lower` for the lower bound of the array shall be constant integral type.

Array of reference shall be declared in the form of

```
T D[&]
```
(15.10)

Array of reference can be used to pass arrays of different data types to functions. It shall be declared at the function parameter scope or in a typedef declaration only.

The *variable length array* type includes assumed-shape array, pointer to array of assumed-shape, deferred-shape array, and array of reference. The following example will clarify the concepts of these various array definitions.

```
void funct(int a[:][:], (*b)[:], c[&], d[], e[1:], n, m){
/* a: assumed-shape array */
/* b: pointer to array of assumed-shape */
/* c: array of reference */
/* d: incomplete array completed by function call */
/* e: assumed-shape array with explicit lower bound */
/* n, m: int */
  int f[4][5];    // f: fixed-length array
  int g[n][m];    // g: deferred-shape array
  int (*h)[4];    // h: pointer to array of 4 elements.
  int (*i)[:];    // i: pointer to array of assumed-shape
  extern int j[]; // j: incomplete array completed by external linkage
  int k[] = {1,2};// k: incomplete array completed by initialization
}
```

For two array types to be compatible, both shall have compatible element types. In addition, if both size specifiers are present and they are integral constant expressions, then both size specifiers shall have the same constant value. If either size specifier is variable, the two sizes shall evaluate to the same value at program execution time. If the two array types are used in a context which requires them to be compatible, the behavior is undefined if the two size specifiers evaluate to unequal values at execution time.

15.2 Deferred-Shape Arrays

15.2.1 Constraints and Semantics

The size of a deferred-shape array type is obtained at program execution time and the value of the size shall be greater than zero. The size of a deferred-shape array type shall not change until the execution of the block containing the declaration has ended. Therefore, at least one of the size expressions is a non-constant integral expression for deferred-shape arrays. The variables used in the size expression must be declared beforehand. For example, arrays a, b, c, d, and e in the following code fragment are valid declaration of deferred-shape arrays whereas arrays f, g, and h are not.

```
int N1;
extern int N;
void funct1(int n, m){
  int i = 8*n;
  int j = 0, k = -9;
  int a[i][4];        /* OK */
  int b[3][m];        /* OK: mix fixed-extent with deferred-extent */
  int c[n*m][n];      /* OK */
  int d[funct2(n)][3*funct2(i)]; /* OK */
  int e[N][N1*n];               /* OK */
  int f[M];          /* ERROR: M has not been defined yet */
  int g[j], gg[0];   /* ERROR: zero size */
  int h[k], hh[-9];  /* ERROR: negative size */
}
int funct2(int i)
{ return i*i;}
int N, M;            /* define N and M */
```

As an application example, a plot with different number of elements for deferred-shape arrays can be generated as follows:

```
int n;
scanf("%d", &n);
double x[n], y[n];
...
plotxy(x, y, n, "title", "x", "y");
```

where the variable `n` contains the number of elements for the deferred-shale arrays `x` and `y`. The value for the variable `n` is obtained from the user input.

Deferred-shape arrays shall be declared in block scope such as variables inside functions and nested functions. Arrays declared with the `static` storage class specifier in block scope shall not be declared as deferred-shape arrays. The behavior for declarations of deferred-shape arrays with file or program scope is undefined. For example,

```
#include <stdio.h>
void funct1(int n, m){
  int funct2(int n, i){
    int a[n][i];          /* OK */
    int b[n];             /* OK */
    return n+m;
  }
  int b[funct2(n,m)][printf("%d\n",n)];  /* OK */
}
extern int n;
int a[n][n];              /* UNDEFINED: not block scope */
static int b[n][n];       /* UNDEFINED: not block scope */
extern int c[n][n];       /* UNDEFINED: not block scope */
int d[2+3][90];           /* OK */
void funct3(int i){
  extern int a[n][n];     /* UNDEFINED: a has linkage */
  static int b[n][n];     /* ERROR: b is static identifier */
  int c[i+3][abs(i)];     /* OK */
}
```

The initializers of objects that have static storage duration are evaluated and the results are stored to objects at compilation time. But, the initializers of objects with automatic storage duration and size expression of deferred-shape arrays are evaluated and values are stored in the object at program execution time. For example,

```
#include <stdio.h>
int n = 4;               /* compile time n==4 */
int main(){
  int m = 5;             /* runtime m == 5 */
  int a[n++][n++];       /* order of evaluation is undefined */
  int b[n++], c[n++];    /* order of evaluation is undefined */
  int d[n++]; int e[n++];/* order of evaluation is defined */
  printf("%d %d %d", n--, b[n--], c[n--]); /* order of evaluation
                                              is undefined */
}
```

Since the size of a deferred-shape array is unknown until the execution time. The size of the deferred-shape array often time is different at each invocation. Therefore, the deferred-shape array shall not be initialized. For example,

```
void funct1(int n){
  int a[3] = {1,2,3};              /* OK */
  int b[ ] = {1,2,3};              /* OK */
  int c[2][3] = {{1,2,3},{4,5,6}}; /* OK */
  int d[ ][3] = {{1,2,3},{4,5,6}}; /* OK */
  int e[n] = {1,2};                /* ERROR: initialization */
  int f[n][n] = {1,2,4,5};         /* ERROR: initialization */
}
```

Pointers to deferred-shape array shall not be declared. For example,

```
void funct(int n){
  int (*p1)[3];      /* OK: pointer to fixed-length array */
  int (*p2)[n];      /* ERROR: pointer to deferred-shape array */
  int (*p3)[n][3];   /* ERROR: pointer to deferred-shape array */
  int (*p3)[3][n];   /* ERROR: pointer to deferred-shape array */
}
```

Deferred-shape arrays can be declared at the function prototype scope if its array index is also declared at the same function prototype scope beforehand as an integral type. In function prototype, the array index can be subsitituted by symbol `*`. For example,

```
void funct1(int n, a[n]);        // OK
void funct1(int n, a[*]);        // OK
void funct1(int n, a[n]){ }      // OK
void funct2(double n, int a[n]); // ERROR: n is not integral
void funct3(int a[n], n);        // ERROR: n in a[n] not declared
```

Deferred-shape shall not mix with the incomplete array type. For example,

```
int n;
int a[][n] = {{1,2},{3,4}}; /* ERROR: initialization */
void funct(int n, b[][n]);  /* ERROR: function prototype scope */
extern c[][n];              /* ERROR: static storage duration */
```

For two array types to be compatible, both shall have compatible element types and the same shape. For example,

```
void funct1(int (*p)[4]])
{int i = sizeof(p);}        /* i == 4 */
void funct2(int p[3][4])
{int i = sizeof(p);}        /* i == 4 */
void funct3(int p[ ][4])
{int i = sizeof(p);}        /* i == 4 */
void funct4(int n)
{
  int i = 3, j = 4;
```

```
  int (*p)[4];
  int a[i][j];
  int b[j][j];
  int c[i][i];
  p = a; funct1(a); funct2(a); funct3(a); /* compatible */
  p = b; funct1(b); funct2(b); funct3(b); /* compatible */
  p = c; funct1(c); funct2(c); funct3(c); /* incompatible */
}
```

15.2.2 Deferred-Shape Arrays Related to Switch Statement

The controlling expression for a `switch` statement shall not cause a block to be entered by a jump from outside the block to a statement that follows a `case` or `default` label in the block if it contains the declaration of a deferred-shape array. Otherwise, the memory for the deferred-shape array within the block will not be allocated. For example,

```
int i;
int main(){
  int n = 10;
  switch (n){
    int a[n]; /* ERROR: bypass declaration of a[n] */
    case 10:
      a[0] = 1;
      break;
    case 20:
      a[1] = 2;
      break;
    case 30:
      {
        int b[n]; /* OK */
        b[1] = 90;
      }
      break;
  }
}
```

15.2.3 Deferred-Shape Arrays Related to Goto Statement

Similarly, the identifier in a `goto` statement shall name a label located somewhere in the enclosing block or its calling function. A `goto` statement shall not cause a block to be entered by a jump from outside the block to a labeled statement in the block if it contains the declaration of a deferred-shape array. For example,

```
void funct(int n){
  int i;
label1:
  if(n>10)
    goto label2;      /* ERROR: bypass declaration of a[n] */
  {
    int a[n];
```

```
    a[i] = 8;
label2:
    a[i] = 9;
    goto label1;      /* OK */
label3:
    a[i] = 10;
    goto label2:      /* OK */
  }

void funct1(int m){
  void funct2(int r){
    if(r)
      goto label4;    /* OK */
    else
      goto label5;    /* ERROR: bypass declaration of b[m] */
  }
label4:
  {
    int b[m];
label5:
    a[0] = 9;
    goto label5;     /* OK */
  }
}
```

When a `goto` statement transfers the program execution flow from a nested function to its parent function, it shall terminate execution of the active function invocation. All dynamically allocated memory including those for deferred-shape arrays shall be deallocated and the previous calling environment shall be restored. The function that called the function containing the `goto` statement once again becomes the active function. If the label named in the `goto` statement is not in the now-active function, the deactivation of the current function and activation of its parent function continue. Eventually, the function containing the label of the `goto` statement will be active, and control flow will be transferred to the statement with the proper label. For example,

```
void funct1(int n){
  local void funct2(int n);
  local void funct3(int n);
  int a[n];
label:
  funct2(n);
  void funct2(int n){
    int b[n];
    funct3(n);
  }
  void funct3(int n){
    int c[n];
    goto label; /* b[n] and c[n] will be deallocated*/
  }
}
```

In this example, memory allocated for deferred-shape arrays `b[n]` and `c[n]` will be deallocated when the control flow is transferred from function `funct3()` to function `funct1()` through function `funct2()`. If *label* and *goto label* statements in the above example were replaced by functions `setjmp(buf)` and `longjmp(buf)`, respectively, the memory of deferred-shape arrays may not be deallocated. With nested functions, functions `setjmp(buf)` and `longjmp(buf)` may become good candidates for obsolete features.

15.2.4 Deferred-Shape Arrays as Members of Structures and Unions

Not only ordinary identifiers, but also members of structures and unions, can be declared as deferred-shape arrays. But, structures and unions with members of deferred-shape arrays shall be declared with automatic storage duration. The behavior for declaring structures and unions with members of deferred-shape array at file or program scope is undefined. Structures declared with the `static` storage specifier in block scope shall not be declared with members of deferred-shape arrays.

Like `sizeof`, `offsetof` is also a built-in operator. If a structure has no member of deferred-shape array, the operation `offsetof`(*type, member-designator)* evaluates to an integral constant value that has type `size_t`, the value of which is the offset in bytes, to the structure member (designated by *member-designator*), from the beginning of the structure (designated by *type*). If the structure contains a member of deferred-shape array, the result is not a constant expression and is computed at program execution time. Because of the variable length of deferred-shape arrays, given

`static` *type* t;

the expression `&(t.`*member-designator)* will not evaluate to an address constant if the structure contains a deferred-shape array.

Structures and unions shall not be defined at the function prototype scope. Structures and unions with members of deferred-shape array can be declared at the function prototype scope of nested functions. For example,

```
int n;
struct tag{
  int m;
  int a[n];           /* UNDEFINED: not block scope for tag1 */
  int b[m];           /* UNDEFINED: not block scope for tag1 */
};
void funct1(int m){
  int l;
  static struct tag{
    int m;
    int a[n];         /* ERROR: static block scope for tag1 */
    int b[m];         /* ERROR: static block scope for tag1 */
  };
  struct tag1{        /* structure shared by
                         funct1(), funct2(), and funct3() */
    int r=2*m;        /* initialization of member r */
    int a[n][m][l];               /* OK */
    int q=2+l, q2;    /* initialization of member q */
    int b[r][q];                  /* OK */
  };
  void funct2(){
```

```
     struct tag1 s1;               /* OK */
     int i;
     i = offsetof(struct tag1, r); /* OK: runtime offsetof() */
     i = offsetof(struct tag1, a); /* OK: runtime offsetof() */
     i = offsetof(struct tag1, b); /* OK: runtime offsetof() */
  }
  /* structure with deferred-shape array as function arg */
  void funct3(struct tag1 s){
    int i, j;
    struct tag1 s1;               /* OK */
    for(i=0; i<s.r; i++)
      for(j=0; j<s.q; j++)
        s1.b[i][j] = s.b[i][j];
  }
  struct tag2{
    int a[2][3];
    int b[4][5];
  };
  l = offsetof(struct tag1, r);    /* OK: runtime offsetof() */
  l = offsetof(struct tag1, a);    /* OK: runtime offsetof() */
  l = offsetof(struct tag1, b);    /* OK: runtime offsetof() */
  l = offsetof(struct tag2, a);    /* OK: compile time offsetof() */
  l = offsetof(struct tag2, b);    /* OK: compile time offsetof() */
}
```

15.2.5 Sizeof

When the built-in `sizeof` operator is applied to an operand that has array type, the result is the total number of bytes allocated for storing the elements of the array. For deferred-shape arrays, the result is not a constant expression and is computed at program execution time. For example,

```
int funct(int n, m){
  int i;
  int a[3][4];
  int b[n][m];
  int c[sizeof(a)];        /* c is fixed-length array */
  int d[sizeof(b)];        /* d is deferred-shape array */
  i = sizeof(a);           /* compile time sizeof(a) is 48 */
  j = sizeof(b);           /* runtime sizeof(b) is nxmx4 */
  return j;
}
```

When the `sizeof` operand is applied to an operand that has structure or union type, the result is the total number of bytes in such an object, including internal and trailing padding. If any member of a structure or union is a deferred-shape array, the result is not a constant expression and it is computed at program execution time. For example,

```
int n;
int funct1(int m){
```

```
  int l;
  struct tag1{
    int a[2][3];
  };
  struct tag2{
    int r;
    int a[4][5];
    int b[n][m][l][r];
  };
  int i;
  struct tag1 s1;
  struct tag2 s2;
  void funct2(struct tag1 s1, struct tag2 s2){
    int i;
    i = sizeof(s1);        /* compile time sizeof() */
    i = sizeof(s1.a);      /* compile time sizeof() */
    i = sizeof(s2.a);      /* compile time sizeof() */
    i = sizeof(s2);        /* runtime sizeof() */
    i = sizeof(s2.b);      /* runtime sizeof() */
  }
  i = sizeof(struct tag1); /* compile time sizeof() */
  i = sizeof(s1);          /* compile time sizeof() */
  i = sizeof(s1.a);        /* compile time sizeof() */
  i = sizeof(s2.a);        /* compile time sizeof() */
  i = sizeof(struct tag2); /* runtime sizeof() */
  i = sizeof(s2);          /* runtime sizeof() */
  i = sizeof(s2.b);        /* runtime sizeof() */
}
```

15.2.6 Typedef

Typedef declarations that specify an aggregate type with a deferred-shape array shall have block scope. The behavior for typedef declaration with deferred-shape arrays in file or program scope is undefined. The deferred-shape of the array shall be evaluated whenever it is used as a type specifier in an actual declarator, not when the type definition is declared. For example,

```
int n = 5;
typedef int A[n];                  /* UNDEFINED: not block scope */
typedef struct tag{int aa[n]} TAG1;/* UNDEFINED: not block scope */
int main(){
  int n;
  typedef int B[n];                /* OK */
  B bb;                            /* OK: int bb[n] */
  B *cc;                           /* ERROR: int (*cc)[n] */
}
void funct(int m){
  typedef int A[m];                /* m is not stored in A */
  typedef struct tag {
```

```
    int b[m];                    /* store m in b[m] */
    struct tag *prev;
    struct tag *next;
} TAG1;
A d;                             /* store m in d */
m++;                             /* increment m */
{
   A a;            /* a[] has one more element than d[] */
   TAG1 s;         /* s.b[] has one more element than d[] */
   int c[m];       /* c[] has one more element than d[] */
}
}
funct(6);          /* ==> d[6], a[7], s.b[7], c[7] */
```

15.2.7 Other Data Types and Pointer Arithmetic

Deferred-shape arrays of different data type can be declared in the same manner as fixed-length arrays. For example,

```
void funct(int n){
  char c[n], *cp[n];
  int *ip[n][n];
  float f[n], **fp[n][n];
  double d[n], *dp[n][n];
  complex z[n], *zp[n][n];
}
```

The pointer arithmetic related to fixed-length arrays is still valid for deferred-shape arrays. For example,

```
void funct(int n, m){
  int i=0, j=0;
  int a[n][m];
  a[i][j] = 90;
  *(a[i]+j) = 90;         /* a[i][j] = 90 */
  *(*(a+i)+j) = 90;       /* a[i][j] = 90 */
  *(&a[0][0]+i*m+j) = 90; /* a[i][j] = 90 */
  *((int *)a[i]+j) = 90;  /* a[i][j] = 90 */
  *((int *)(a+i)+j) = 90; /* a[i][j] = 90 */
  *((int *)a+i*m+j) = 90; /* a[i][j] = 90 */
  i = a[n-1] - a[n-2];    /* i == m */
}
```

15.3 Assumed-Shape Arrays

15.3.1 Constraints and Semantics

Assumed-shape arrays shall be declared at the function prototype scope or in a typedef declaration. The assumed-shape array is a formal argument that takes the shape of the actual argument passed to it. That is, the arrays for actual and formal arguments have the same rank and the same extent in each dimension. The

```
void bxc(double aa[:][:], double bb[:][:], double cc[:][:], int n, int m, int r);
int main() {
  int i, n = 2, m = 4, r =6;
  double a[2][6], b[2][4], c[4][6]; // double a[n][r], b[n][m], c[m][r]
  /* ... */
  bxc(a,b,c,n,m,r);
}

void bxc(double aa[:][:], double bb[:][:], double cc[:][:], int n, int m, int r) {
  /* array multiplication a = b[n][m]*c[m][r] */
  int i, j, k;

  for(i=0; i<=n-1; i++)
    for(j=0; j<=r-1; j++) {
      aa[i][j] = 0;
      for(k=0; k<=m-1; k++)
        aa[i][j] += bb[i][k]*cc[k][j];
    }
}
```

Program 15.1: Passing two-dimensional arrays to a function using assumed-shape arrays.

shape of assumed-shape arrays cannot be determined until execution time. The rank of an assumed-shape array is equal to the number of colons in the assumed-shape specification. For example,

```
void funct(int [:], [:][:])      // OK
void funct(int dummy1[:], dummy2[:][:]) // OK
void funct(int a[:], b[:][:])    // OK
int A[:];                        // ERROR: not function prototype scope
static int B[:][:];              // ERROR: not function prototype scope
extern C[:][:];                  // ERROR: not function prototype scope
void funct(int a[:], b[:][:]){   // OK
  int c[:][:];                   // ERROR: not function prototype scope
  extern int A[:];               // ERROR: not function prototype scope
  void funct2(int a[:], b[:][:]){// OK
     int c[:][:];                // ERROR: not function prototype scope
  }
  funct2(a, b);                  // OK
}
```

Application of assumed-shape arrays can be illustrated by Program 15.1. In this program, the function `bxc()` will multiply two two-dimensional arrays `b` and `c`. The product is passed back to the calling function by argument `a` The dimensions of arrays in the calling function are passed to the function `bxc()` by three parameters `n, m` and `r`.

Assumed-shape arrays may also appear in a typedef declaration. For example,

```
typedef int A[:];
A a;                /* ERROR: not function prototype scope */
void funct(A a);    /* OK */
```

Only variables of fixed-length, deferred-shape, or assumed-shape array type can be used as an actual argument of a formal argument of assumed-shape array type in function parameters. A pointer or pointer

to array, which does not have the complete shape information, shall not be used as an actual argument of a formal argument of assumed-shape array type. For example,

```
funct1(int a[:][:]){
  int n=a[1][1], m = a[1][2];
  int b[3][4];
  int c[n][m];
  int *p1, (*p2)[4], (*p3)[:];
  void funct3(int a[:][:]);
  void funct2(int a[:][:])
  { }
  funct2(a);  funct3(a); // OK a is assumed-shape array
  funct2(b);  funct3(b); // OK b is fixed-length array
  funct2(c);  funct3(c); // OK c is deferred-shape array
  funct2(p1); funct3(p1);// ERROR: p1 is pointer
  funct2(p2); funct3(p2);// ERROR: p2 is pointer to fixed-length array
  funct2(p3); funct3(p3);// ERROR: p3 is pointer to assumed-shape array
}
void funct3(int a[:][:])
{ }
```

Although complete arrays can be extracted from a pointer to array, they shall not be used as actual arguments of an assumed-shape array. For example,

```
void funct1(int a[3]);
void funct2(int a[5][7]);
void funct11(int a[:]);
void funct22(int a[:][:]);
void funct3(int p2[][5][7]){
  int a[5][3];
  int (*p1)[3];
  p1 = a;
  funct1(p1[0]);   // OK: passed a[0][0], ..., a[0][2]
  funct1(p1[1]);   // OK: passed a[1][0], ..., a[1][2]
  funct1(*(p1+1)); // OK: passed a[1][0], ..., a[1][2]
  funct1(a[4]);    // OK: passed a[4][0], ..., a[4][2]
  funct1(p1+1);    // OK: p1+1 is a pointer to array of 3 ints
  funct2(p2[1]);   // OK: passed p2[1][0][0], ..., p2[1][4][6]

  funct11(p1[0]);  // ERROR: passing array a[0][0], ..., a[0][2]
  funct11(p1[1]);  // ERROR: passing array a[1][0], ..., a[1][2]
  funct11(*(p1+1));// ERROR: passing array a[1][0], ..., a[1][2]
  funct11(a[4]);   // ERROR: passing array a[4][0], ..., a[4][2]
  funct11(p1+1);   // ERROR: p1+1 is a pointer to array of 3 floats
  funct22(p2[1]);  // ERROR: passing array p2[1][0][0],...,p2[1][4][6]
}
```

where `p1[0], p1[1], *(p1+1)`, and `a[4]` are arrays with size of 12 bytes, `p2[1]` is an array of 140 bytes, and `p+1` is a pointer to array with size of 4 bytes.

Assumed-shape array shall not mix with fixed-length or incomplete array type. For example,

```
void funct(int a[:][3]);// ERROR: mix assumed-shape with fixed-length
void funct(int a[3][:]);// ERROR: mix assumed-shape with fixed-length
void funct(int a[n][:]);// ERROR: mix assumed-shape with deferred-shape
void funct(int a[:][n]);// ERROR: mix assumed-shape with deferred-shape
void funct(int a[ ][:]);// ERROR: mix assumed-shape with incomplete
```

If the operand of a polymorphic operation or function is an element of an assumed-shape array, the data type of the result and operation depend on the data type of the formal argument. However, if the formal and actual data types of an argument are different but compatible, the operand will be cast to an operand with data type of the formal argument before operation takes place. If an element is used as an lvalue, the rvalue is cast to the data type of the actual argument if they are different. In other words, elements of the actual array are coerced to the data type of the assumed-shape array at program execution time when they are fetched whereas they are coerced to data type of the actual argument when they are stored. For example,

```
float A[3] = {1, 2};
complex Z[3] = {complex(1,0), complex(2,0)};
void funct(float a[:], complex z[:]){
  a[2] = a[0] + a[1];     /* addition of floats */
  z[2] = z[0] + z[1];     /* addition of complexs */
}
funct(Z, A);              /* A[2]==3.0, Z[2]=3.0+i0.0 */
```

If the formal argument is an assumed shape, the actual argument can also be an assumed-shape array. For example,

```
void funct2(complex aa[:], b[:][:], (*c)[6], d[][6], e[4][6]){
  aa[1] = b[1][2];
}
void funct1(complex a[:], b[:][:]){
  if(real(a[1]) == 0)
  funct2(a,b,b,b,b);      /* a and b are assumed-shape arrays */
}
int main(){
  complex  A[2], B[4][6];
  funct1(A,B);            /* A and B are fixed-length arrays */
}
```

When the function `funct2()` is invoked by the function call of `funct2(a,b,b,b,b)`, the memory allocated for array `A` in the main routine is used by the assumed-shape array `a` in the function `funct1()`, and subsequently it is passed to the assumed-shape array `aa` in the function `funct2()`. An assumed-shape array can also be used as the actual argument of a pointer to fixed-length array in a function. In the above example, the memory allocated for array `B` in the main routine is used as `b` in the function `funct1()` and as `b, c, d, e` in the function `funct2()`. Different identifiers `a` and `aa` are used for the same array object allocated at the declaration of array `A`. But, the same identifier `b` has been used in both functions `funct1()` and `funct2()` for the array object `B`. This shows that the names of identifiers are irrelevant to argument association of functions.

15.3.2 Sizeof

When the operand of **sizeof**() operation is an assumed-shape array type, the result is the total number of bytes used to store elements of the array computed at program execution time. Furthermore, since arrays

of different data types can be passed to assumed-shape arrays, the size of an element of an assumed-shape array will also be computed at program execution time. For example,

```
int funct(complex z[:]){
  int i, numOfElement;
  numOfElement = sizeof(z)/sizeof(z[0]);
  return numOfElement; /* sizeof(z)=80, sizeof(z[0])=4 */
}
int main(){
  int num;
  float a[20];
  num = funct(a);      /* num == 20 */
}
```

15.3.3 Other Data Types and Pointer Arithmetic

Assumed-shape arrays of other data types are handled in the same manner as assumed-shape arrays of ints. For example, the following statement declares that variables `a, b`, and `c` are assumed-shape complex arrays of rank one, two, and three, respectively.

```
int funct(complex a[:], b[:][:], c[:][:][:]);
```

Assumed-shape arrays of different data types can be handled in the same manner. For example, in the following code fragment

```
char *cc[10]; float **ff[2][4]; double ***dd[3][5][7];
int funct(char *c[:]; float **f[:][:], double ***d[:][:][:]);
funct(cc, ff, dd);
```

the function prototype

```
int funct(char *c[:], float **f[:][:], double ***d[:][:][:]);
```

defines variables `c, f`, and `d` as the rank-one assumed-shape array of pointer to char, rank-two assumed-shape array of double pointer to float, and rank-three assumed-shape array of triple pointer to double, respectively. Arrays `cc, ff`, and `dd` are passed to the formal assumed-shape arrays `c, f`, and `d` in the function `funct()`, respectively. The assumed-shape arrays in a function are handled in the same manner as fixed-length arrays. For example,

```
void funct(complex z1[:], z2[:][:]){
  complex z, *zp, **zp2;
  zptr = z1;             /* the address of the array */
  zptr = &z1[2];         /* the address of the third element */
  /* z2[2][1] -= 1; z1[1] = z2[2][1] + z1[2]; z1[2] += 1; */
  z1[1] = --z2[2][1]+z1[2]++;
  z = *z1;               /* z = z1[0] */
  z = *(z1+5);           /* z = z1[5] */
  z = **z2;              /* z = z2[0][0] */
  zp = z2[2];            /* zp = &z2[2][0] */
  zp2 = (complex **)z2;
```

```
    /* z2[1][1] = z2[1][3] + z2[2][3] - z2[2][4]; */
    zp2[1][1] = z2[1][3]+ *(*(z2+2)+3) - *(4+*(z2+2));
    /* z2[2][3] = z2[1][3] + z2[2][3] */
    *(*(z2+2)+3) = z2[1][3]+  *(3+*(z2+2));
}
```

15.4 Pointers to Array of Assumed-Shape

15.4.1 Declaration

A *pointer type* may be derived from a function type, and object type, or an incomplete type, called the *reference type*. A pointer type describes an object whose value provides a reference to an entity of the reference type. A pointer type derived from the reference type *T* is sometimes called "pointer to *T*." The construction of a pointer type from a referenced type is called "pointer type construction."

If, in the declaration "T D1" described in section 15.1.2, D1 has the form

* *type-qualifier-list*$_{opt}$ D

and the type specified for *ident* in the declaration "T D" is "*derived-declarator-type-list T*," then the type specified for *ident* is "*derived-declarator-type-list* pointer to *T*." For each type qualifier in the list, *ident* is a so-qualified pointer.

Pointers to array of fixed-length is declared as

T (*D)[*assignment-expression*]

where T contains the declaration specifiers that specify a type and the assignment expression is an integral constant expression. For example,

```
int (*p1)[3];     /* p1 is pointer to array of 3 ints */
int *(*p2)[3];    /* p2 is pointer to array of 3 pointer to int */
int (*p3)[3][4];  /* p3 is pointer to 3x4 array of ints */
int *(*p4)[3][4]; /* p4 is pointer to 3x4 array of pointer to int */
```

Pointers to array of assumed-shape are declared as

T (*D)[:]

where T contains the declaration specifiers that specify a type. For example,

```
int (*p1)[:];     /* OK */
int (*p2)[:][:];  /* OK */
int *(*p3)[:][:]; /* OK */
int n = 8;
int (*p4)[3][:];  /* ERROR: mix fixed-length with assumed-shape */
int (*p5)[:][3];  /* ERROR: mix fixed-length with assumed-shape */
int (*p6)[n][:];  /* ERROR: mix deferred-shape with assumed-shape */
int (*p7)[:][n];  /* ERROR: mix deferred-shape with assumed-shape */
int (*p8)[ ][:];  /* ERROR: mix deferred-shape with incomplete type */
```

where `p1` is pointer to array of assumed-shape of rank 1 with int type, `p2` is pointer to array of assumed-shape of rank 2 with int type, and `p3` is pointer to array of assumed-shape of rank 2 with pointer to int type.

The shape of the array pointed to by a pointer to assumed-shape array is determined at program execution time. A pointer to assumed-shape array is sometimes called a *fat pointer* since it can store more information than a pointer to object of scalar type or a pointer to array of fixed-length at program execution time.

15.4.2 Constraints and Semantics

Except for pointer to assumed-shape array type, a pointer to `void` may be converted to or from a pointer to any incomplete or object type. A pointer to any incomplete or object type, except pointer to assumed-shape array type, may be converted to a pointer to `void` and back again; the result shall compare equal to the original pointer.

For any qualifier *q*, a pointer to non-*q*-qualified type may be converted to a pointer to the q-qualified version of the type; the values stored in the original and converted pointers shall compare equal.

An integral constant expression with the value 0, or such an expression cast to type `void *`, is called a *null pointer*. If a null pointer constant is assigned to or compared for equality to a pointer, the constant is converted to a pointer of that type. Such a pointer, called a *null pointer*, is guaranteed to compare unequal to a pointer to any object or function.

Two null pointers, converted through possibly different sequences of casts to pointer types, shall compare equal.

When a null pointer is converted to a pointer to array of assumed-shape, a null pointer is installed at the base pointer of the assumed-shape array and the bounds of the assumed-shape array are undefined.

An array, including fixed-length array, deferred-shape array, and assumed-shape array, may be converted to a pointer to assumed-shape array. A pointer to array of fixed-length or pointer to array of assumed-shape may also be converted to a pointer to assumed-shape array. The base pointer to array and all bounds are stored in the pointer to assumed-shape array. All other pointer types that do not have the array shape information shall not be converted to a pointer to array of assumed-shape. For example,

```
void funct(int a[:][:], p1[2][4], (*p2)[4], p3[][4], n, m){
  int *p;
  int b[3][4];
  int c[n][m];
  int (*p4)[4];
  int (*p5)[:];
  int (*p6)[:];
  p6 = NULL;
  p6 = a;        /* OK: a is an assumed-shape array */
  p6 = b;        /* OK: b is a fixed-length array */
  p6 = c;        /* OK: c is a deferred-shape array */
  p6 = p1;       /* OK: p1 is a pointer to array of fixed-length */
  p6 = p2;       /* OK: p2 is a pointer to array of fixed-length */
  p6 = p3;       /* OK: p3 is a pointer to array of fixed-length */
  p6 = p4;       /* OK: p4 is a pointer to array of fixed-length */
  p6 = p5;       /* OK: p5 is a pointer to array of assumed-shape */
  p4 = p;        /* WARNING: array bounds do not match */
  p6 = p;        /* ERROR: p is not array type */
}
```

For two pointer types to be compatible, both shall be identically qualified and both shall be pointers to compatible types. For two pointers to fixed-length array to be compatible, both shapes of array pointed to by the pointer shall be the same. For two pointers to assumed-shape array to be compatible, both ranks of the array pointed to by the pointer shall be the same and the shapes shall evaluate to the same value at program execution time. For example,

```
void funct(int a[:], b[:][:][:], (*p1)[4][5], p2[3], n, m){
```

```
  int c[3][4][5];
  int d[n][m][m];
  int (*p3)[4][5];
  int (*p4)[:];
  p4 = a;        /* ERROR: incompatible, wrong rank  */
  p4 = b;        /* ERROR: incompatible, wrong rank  */
  p4 = p1;       /* ERROR: incompatible, wrong rank  */
  p4 = p2;       /* ERROR: incompatible, wrong rank  */
  p4 = c;        /* ERROR: incompatible, wrong rank  */
  p4 = d;        /* ERROR: incompatible, wrong rank  */
  p3 = p4;       /* WARNING: incompatible, wrong rank */
}
```

When a pointer to array of assumed-shape is converted to any other pointer to object or to a scalar value, only the base pointer to assumed-shape array is used.

```
char c, *cp;
int i, *ip;
float f, *fp;
int (*ap)[4];
int (*p)[:];
c  = (char) p;       /* OK */
cp = (char *) p;     /* OK */
i  = (int) p;        /* OK */
ip = (int*) p;       /* OK */
ip = p;              /* OK */
f  = (float) p;      /* OK */
fp = (float*) p;     /* OK */
ap = p;              /* OK */
```

15.4.3 Function Prototype Scope

A pointer to assumed-shape array can be used as an argument parameter of a function to pass arrays of different size to the function. For example,

```
void funct(int (*)[:]);
void funct(int (*dummy)[:]);
void funct(int (*p)[:]);
int a[3][4], b[4][3];
int (*p1)[:];
funct(a,3,4);       /* passing fixed-length array a[3][4] */
funct(b,3,4);       /* passing fixed-length array b[4][3] */
p1 = a;
funct(p1,3,4);      /* passing fixed-length array a[3][4] */
funct(NULL,0,0);    /* passing NULL */
void funct(int (*p)[:], n, m){
  int i, j;
  int a[n][m];
  if(p == NULL)
```

```
    return;
  for(i=0; i<n; i++)
    for(i=0; i<m; i++)
      a[i][j] = p[i][j];
}
```

Arrays of deferred-shape and assumed-shape can also be passed to a pointer to array of assumed-shape. For example,

```
void funct1(int a[:][:], n, m){
  int b[n][m];
  void funct2(int (*p)[:])
  {
    int i, j;
    int c[n][m];
    if(p == NULL)
      return;
    for(i=0; i<n; i++)
      for(i=0; i<m; i++)
        c[i][j] = p[i][j];
  }
  funct2(a);  /* a is an assumed-shape array */
  funct2(b);  /* b is a deferred-shape array */
}
```

15.4.4 Typedef

The assumed-shape array and pointer to assumed-shape array are handled in the same manner as the fixed-length array and pointer to fixed-length array in typedef declarations. For example,

```
typedef int A[5];
typedef int B[:];
A a;            // OK: int a[5]
A *ap;          // OK: int (*ap)[5]
B b;            // ERROR: not function prototype scope for 'int b[:]'
B *bp;          // OK: int (*bp)[:]
/* void funct(int a[5], (*ap)[5], int b[:], (*bp)[:]); */
void funct(A a, *ap, B b, *bp); // OK
```

where `a` is an assumed-shape array and `ap` is a pointer to assumed-shape array.

15.4.5 Arrays Allocated by Dynamic Memory Allocation Functions

Arrays can be dynamically allocated as shown in the following example.

```
funct(int n, int m){
  double a[n][m];
  double (*p1)[:] = a;   // OK p[i][j] = a[i][j]
  double (*p2)[:] = (double [n][m])malloc(sizeof(double)*n*m);// OK
  double (*p3)[:] = (double [ ][m])malloc(sizeof(double)*n*m);// OK
```

```
  double (*p4)[:] = (double [][m])malloc(sizeof(double)*n*m); // OK
  /* ERROR: pointer to deferred-shape array is not allowed */
  double (*p5)[:] = (double(*)[m])malloc(sizeof(double)*n*m);// ERROR
}
```

where `a` is a deferred-shape array, and `p1, p2`, and `p3` are pointers to assumed-shape array of double data type. All memories pointed to by `p1, p2, p3`, and `p4` are dynamically allocated. But, memories for `p2, p3,` and `p4` are obtained explicitly by the memory allocation function `malloc()`.

15.4.6 Similarities between Pointers to Fixed-Length Array and Pointers to Assumed-Shape Array

A pointer to array of assumed-shape behaves very much like a pointer to array of fixed-length. For example, as a pointer, it should be pointed to an object before its elements can be referenced. Some other points that may be not so straightforward at the first sight will be clarified in this section.

Static and Automatic Storage Duration

Unlike deferred-shape arrays, there is no restriction on the scope where a pointer to array of assumed-shape can be declared. It can be declared with either static storage duration or automatic storage duration. For example,

```
int (*p1)[:];
extern int (*p2)[:];
static int (*p3)[:];
int main(){
  int (*p4)[:];
  static (*p5)[:];
  extern int (*p1)[:];
}
```

Initialization

A pointer to array of assumed-shape can be initialized at both compilation and program execution time. For example,

```
int a[3][4];
int (*p1)[:] = NULL;        /* runtime initialization */
extern int (*p2)[:];
static int (*p3)[:] = NULL; /* runtime initialization */
int main(){
  int b[3][4];
  int (*p4)[:] = NULL;      /* compile time initialization */
  int (*p5)[:] = b;         /* compile time initialization */
  int (*p6)[:] = p1;        /* compile time initialization */
  static (*p7)[:] = a;      /* runtime initialization */
  static (*p8)[:] = p1;     /* runtime initialization */
  static (*p8)[:] = b;      /* ERROR: b is variable of auto class */
}
```

Members of Structures and Unions

Not only ordinary identifiers, but also members of classes, structures and unions, can be declared as pointer to array of assumed-shape. For example,

```
struct tag1{
  int (*p1)[3];    /* pointer to fixed-length array */
  int (*p2)[:];    /* pointer to assumed-shape array */
};
int main(){
  struct tag2{
    int (*p1)[3];  /* pointer to fixed-length array */
    int (*p2)[:];  /* pointer to assumed-shape array */
  } s;
}
```

where the structure `tag1` has static storage duration and structure `tag2` has automatic storage duration. In the interactive commands executed in a Ch shell below, member s.a first shares the same memory as array a1, then shares the memory of array a2.

```
> struct tag{ int (*a)[:];} s
> int a1[2][3] = {1,2, 3, 4, 5, 6}, a2[3][4]
> s.a = a1;  // s.a and a1 share the memory
> a1[1][1]
5
> s.a[1][1]
5
> s.a = a2;  // s.a and a2 share the memory
s.a[1][1] = 10
> a2[1][1]
10
> a1[1][1]
5
```

Sizeof

The size of a pointer to array of assumed-shape is the same as the size of a pointer to array of fixed-length. The size of a pointer to array of assumed-shape is the same as the size of the pointer to the data type of the array, which is evaluated at compile time. For example,

```
int (*a)[5];
int (*p1)[:];
int main(){
  int (*b)[5];
  int (*p2)[:];
  void funct(int (*p3)[:], (*p4)[:][:])
  {
    int i;
    i = sizeof(a);  /* i == 4 */
    i = sizeof(b);  /* i == 4 */
```

```
    i = sizeof(p1); /* i == 4 */
    i = sizeof(p2); /* i == 4 */
    i = sizeof(p3); /* i == 4 */
    i = sizeof(p4); /* i == 4 */
  }
}
```

Other Data Types and Pointer Arithmetic

Pointers to array of assumed-shape of different data type can be declared in the same manner as pointers to array of assumed-shape of int. For example,

```
void funct(int n){
  char    (*cp1)[:], *(*cp2)[:], **(*cp3)[:];
  int     (*ip1)[:], *(*ip2)[:], **(*ip3)[:];
  float   (*fp1)[:], *(*fp2)[:], **(*fp3)[:];
  double  (*dp1)[:], *(*dp2)[:], **(*dp3)[:];
  complex (*zp1)[:], *(*zp2)[:], **(*zp3)[:];
}
```

The pointer arithmetic related to pointers to array of fixed-length is still valid for pointers to array of assumed-shape. For example,

```
int main(){
  int i=2, j=3;
  int n=4, m=5;
  int a[4][5];
  int (*p)[:]
  p = a;
  p[i][j] = 90;           /* a[i][j] = 90 */
  *(p[i]+j) = 90;         /* a[i][j] = 90 */
  *(*(p+i)+j) = 90;       /* a[i][j] = 90 */
  *(&p[0][0]+i*m+j) = 90; /* a[i][j] = 90 */
  *((int *)p[i]+j) = 90;  /* a[i][j] = 90 */
  *((int *)(p+i)+j) = 90; /* a[i][j] = 90 */
  *((int *)p+i*m+j) = 90; /* a[i][j] = 90 */
  i = p[n-1] - p[n-2];    /* i == m */
}
```

15.5 Arrays with Explicit Lower and Upper Bounds

As one can see from the previous sections, it is painful to handle arrays of variable subscript range in C, especially for high dimensional arrays. For arrays of different data type or different dimension, different memory allocation and deallocation functions equivalent to `mallocMatrix()` and `freeMatrix()` have to be used. Evidently, in order to evolve C as a major player in the numerical computing world, simple mechanisms must be designed to handle variable length arrays with variable subscript ranges. In this section, such simple mechanisms for handling variable length arrays with explicit lower and upper bounds as they are currently implemented in the Ch programming language will be described. It should be emphasized that new features presented here will not break the C standard and existing C code.

15.5.1 Arrays of Fixed Subscript Range

An array with specified lower bounds shall be declared in one of the following forms:

```
T D[lower:upper]          (15.11)
   T D[expr]              (15.12)
  T D[lower:]             (15.13)
                          (15.14)
```

where `T` contains the declaration specifiers that specify a type such as `int`, `D` is a declarator that contains an identifier *ident*, `lower` is the lower bound of the array, `upper` is the upper bound, and `expr` is the number of the elements of the array. The expressions `lower`, `upper` and `expr` shall be of integral type. For example,

```
int a[1:3], b[0:2][1:5], *c[1:3][1:4][0:5];
```

where the lower and upper bounds of array `a` are 1 and 3, respectively. Elements `a[0]` and `a[4]` are out of the array boundary.

If the lower bound is not present in declaration (15.12), zero is used as the default value for the lower bound of the array. The upper bound is the value inside delimiters [and] minus 1, which is `expr`-1. For example,

```
int b[0:2][5];          /* equivalent to int b[0:2][0:4] */
int a[3];               /* equivalent to int a[0:2] */
int *c[1:3][4][0:5];    /* equivalent to int *c[1:3][0:3][0:5]; */
```

where the lower and upper bounds of array `a` are 0 and 2, respectively. Elements `a[-1]` and `a[3]` are out of the array boundary.

Both lower and upper bounds may be negative integral values. For example,

```
int a[-5:5], b[-5:0], c[-10:-5];
```

For arrays of fixed subscript range, both lower and upper expressions are constant integral values. The upper bound shall evaluate to a value greater than the lower bound. For example,

```
#define N 0
float ff = 5;
int a[5.0];           /* ERROR: expression double type*/
int b[ff];            /* ERROR: expression float type */
int c[0];             /* ERROR: lower and upper bounds are equal */
int d[N];             /* ERROR: lower and upper bounds are equal */
int e[5:5];           /* ERROR: lower and upper bounds are equal */
int f[5:0];           /* ERROR: upper is not greater than lower */
int g[5:-5];          /* ERROR: upper is not greater than lower */
```

When the upper bound is not present such as in declarations (15.13) the array type is an incomplete type. If there is no expression inside delimiters [and], the lower bound is the default value 0. For example,

```
/* incomplete array completed by external linkage
   same as extern int a[0:], b[0:][5]; */
```

```
extern int a[], b[][5];
extern int c[1:], b[1:][1:5]; /* completed by external linkage */
void funct1(int e[]);         /* completed by function call */
void funct2(int f[][5]);      /* completed by function call */
void funct3(int g[1:]);       /* completed by function call */
void funct4(int h[1:][1:5]);  /* completed by function call */
void funct5(int i[1:][5]);    /* completed by function call */
int j[] = {1,2,3};            /* completed by initialization */
int k[][2] = {{1,2}, {3,4}};  /* completed by initialization */
int l[1:] = {1,2,3};          /* completed by initialization */
int m[1:][2] = {{1,2}, {3,4}};/* completed by initialization */
int a[3], b[4][5];            /* external linkage */
int c[1:3], b[1:4][1:5];      /* external linkage */
```

Details about passing arrays with specified lower bounds in functions `funct3(), funct4()` and `funct5()` will be described in the next section.

Arrays shall not be declared with an upper bound alone without a lower bound. For example,

```
int a[:5];             /* ERROR: without lower bound */
int funct(int b[:5]);  /* ERROR: without lower bound */
```

There is a strong relation between pointers and arrays in C. The variable name of an array in an expression is also a pointer to the memory for the first element of the array. This strong tie between pointer and array is retained. If the lower bound of an array is zero, all semantics about the array name as a pointer remain the same. For example, a subscript is equivalent to an offset from a pointer.

```
int a[5], b[0:4], *p;
p = &b[0];             /* p = b */
*(a+0) = *(b+0);       /* a[0] = b[0] */
*(a+4) = *(b+4);       /* a[4] = b[4] */
*(p+1) = p[1]*2;       /* b[1] = b[1]*2 */
```

But, when the lower bound of an array is not zero, there is a difference between the array subscripting and pointer arithmetic. A subscript is equivalent to an offset from a pointer minus the lower bound of the array. For example,

```
#define i 1
int b[i:5], *p, j=3;
p = &b[i];             /* p = b */
*(b+j) = b[j];         /* b[j+i] = *(b+j-i) */
*(p+j) = b[j];         /* p[j]   = *(b+j-i) */
```

The same principle can be applied to multi-dimensional arrays, For example,

```
#define n 1
#define m 2
int a[n:8][m:9], i=3, j=4;
a[i][j] = 90;
*(&a[i][j]) = 90;                    /* a[i][j] = 90 */
```

```
*(a[i]+j-m) = 90;                    /* a[i][j] = 90 */
*(*(a+i-n)+j-m) = 90;                /* a[i][j] = 90 */
*((int *)a[i]+j-m) = 90;             /* a[i][j] = 90 */
*((int *)(a+i-n)+j-m) = 90;          /* a[i][j] = 90 */
*((int *)a+(i-n)*(9-m+1)+j-m) = 90;  /* a[i][j] = 90 */
*(&a[n][m]+(i-n)*(9-m+1)+j-m) = 90;  /* a[i][j] = 90 */
i = a[i+1] - a[i];                   /* i = 9-m+1  is 8 */
```

Pointers to array with explicit lower and upper bounds can be handled in the same manner. For example,

```
int a[3][1:5], b[0:5][1:5];
int (*p)[1:5];
p = a;           /* p[i][j] = a[i][j] */
p = b;           /* p[i][j] = b[i][j] */
```

where `p` is a pointer to array of 10 elements with lower bound 1. In the next section, we will describe how to use a pointer to assumed-shape array so that elements `p[i][j]` and `a[i][j]` refer to the same object when the pointer `p` points to array `a`. And elements `p[i][j]` and `b[i][j]` also refer to the same object when the same pointer `p` points to array `b`.

Arrays with explicit lower and upper bounds can be used in casting operations. For example,

```
int a[3][1:5], b[1:5][2:6];
int (*p)[1:5];
p = (int (*)[1:5])a;         /* p[i][j] == a[i][j] */
p = (int (*)[1:5])b;         /* p[i][j] == b[i][j+1] */
p = (int (*)[1:5])malloc(3*5*sizeof(int)); free(p);
p = (int [][1:5])malloc(3*5*sizeof(int));  free(p);
p = (int [0:][1:5])malloc(3*5*sizeof(int));
```

Arrays with explicit lower and upper bounds can be used in typedef declaration. For example,

```
typedef int A[1:5];
A a;                /* int a[1:5] */
```

For two array types to be compatible, both shall have compatible element types and the same shape. Only if both lower and upper bounds of the subscript for each dimension of two arrays are the same, the shapes of these two arrays are said to be the same. For example,

```
extern int a[3], c[0:2], b[1:5];
int a[0:2], c[3];        // OK
int b[5];                // ERROR
int funct(int aa[1:3]);
int funct(int aa[0:2]);  // ERROR: change array bounds
int e[3][1:5], f[10][1:5], g[3][5], h[1:3][1:5], i[3][0:5];
int (*p)[1:5];
p = e; // OK: compatible
p = f; // OK: compatible
p = g; /* incompatible second dimension p[i][j+1] == g[i][j],
```

```
            no warning or error message */
p = h; /* incompatible first dimension p[i][j] == h[i+1][j],
          no warning or error message */
p = i; // WARNING: incompatible second dimension p[i][j+1] != i[i][j]
```

Note that elements `p[i][j+1]` and `g[i][j]` refer to the same object because the extent of value 5 for the second dimension of the arrays is the same. But, elements `p[i][j+1]` and `i[i][j]` do not refer to the same object.

15.5.2 Arrays of Variable Subscript Range

Arrays of variable length whose size is known only at program execution time have been presented in the previous sections. The variable length array type includes deferred-shape array, assumed-shape array, and pointer to assumed-shape array. This variable length array type will be extended with explicit lower and upper bounds in this section. All syntax and semantics of deferred-shape arrays and pointers to assumed-shape arrays described in the previous sections are still valid. The semantics of assumed-shape arrays remain the same whereas its syntax has been modified, which will be described in the next section.

Arrays of Deferred Subscript Range

If the lower or upper bound of the array subscript is a nonconstant integral expression, it is evaluated at program execution time and the array type is *array of deferred subscript range*. For example,

```
int funct(int n, int m) {
  int i = n;
  int a[n:m], b[i:m], c[-n:2*m][i:n+m];
  int d[1:n], e[n:10], f[1:5][0:n];
}
```

where `a, b, c, d, e,` and `f` are arrays of deferred subscript range. The upper bound of an array of deferred subscript range shall evaluate to a value greater than the value of lower bound at runtime. For example,

```
int funct(int n, int m) {
  int a[n:m];
}
funct(1,5);               // OK: int a[1:5]
funct(5,1);               // ERROR: int a[5:1]
funct(5,5);               // ERROR: int a[5:5]
```

Because arrays of deferred subscript range are also arrays of deferred-shape, all constraints and semantics about deferred-shape arrays described in the previous sections can be applied to arrays of deferred subscript range. For example, pointers to arrays of deferred subscript range shall not be declared.

```
/* ERROR: pointer to deferred-shape array */
int funct(int n, int m, int a[n:m], int (*b)[n:m]) {
  int (*p1)[n:m];         // ERROR: pointer to deferred-shape array
  int (*p2)[1:m];         // ERROR: pointer to deferred-shape array
  int (*p3)[n][1:m];      // ERROR: pointer to deferred-shape array
}
```

Arrays of deferred subscript range shall not mix with incomplete array type. For example,

```
int n=4, m=5;
int a[][n:m]={{1,2}, {3,4}};        // ERROR: initialization
int b[1:][n:m]={{1,2}, {3,4}};      // ERROR: initialization
int funct(int n, int m, c[][n:m]);  // ERROR: func parameter scope
int funct(int n, int m, d[1:][n:m]);// ERROR: func parameter scope
int funct(int n, int m, e[n:][n:m]);// ERROR: func parameter scope
extern int f[][n:m];             // ERROR: static storage duration
extern int g[1:][n:m];           // ERROR: static storage duration
```

Pointers to Assumed-Shape Array

A pointer to array of fixed-length shall be declared as

```
T (*D)[expr]
```
(15.15)

where `T` contains the declaration specifiers that specify a type and `D` is a declarator that contains an identifier *ident*. The expression `expr` shall be constant integral type. A pointer to array of assumed-shape shall be declared in one of the following two forms:

```
T (*D)[:]
```
(15.16)

```
T (*D)[lower:]
```
(15.17)

The expression `lower` for the lower bound of the array shall be constant integral type. For example,

```
int n=10;
int (*p1)[:];        /* OK */
int (*p2)[:][:];     /* OK */
int *(*p3)[:][:];    /* OK */
int (*p4)[3][:];     /* ERROR: mix with fixed-length */
int (*p5)[n][:];     /* ERROR: mix with deferred-shape */
int (*p5)[n:];       /* ERROR: mix with deferred-shape */
int (*p6)[ ][:];     /* ERROR: mix with incomplete */
```

When the shape of an array is assumed by a pointer to assumed-shape array, both lower and upper bounds of the subscript of the assumed array will be assumed. For example,

```
int n=3, m=4;
int a[3][4], b[1:n][1:m], c[3][1:4];
int (*p)[:];
p = a;          /* p[i][j] == a[i][j] */
p = b;          /* p[i][j] == b[i][j] */
p = c;          /* p[i][j] == c[i][j] */
```

The declaration (15.17) with a specified lower bound shall be used only at the function parameter scope, and nowhere else. For example,

```
int a[1:3][1:4];
int (*p1)[1:4];     /* OK: pointer to fixed-length array */
int (*p2)[1:];      /* ERROR: pointer to incomplete array
                              not at function parameter scope */
p1 = a;             /* OK */
p2 = a;             /* ERROR */
```

In this example, variable `p1` is a pointer to array of 3 elements of int type with unit-offset. Declaration of variable `p2` is invalid. Since `p2` were invalid, no consistent grammar can be composed for assignment statement `p2 = a` for the lower bounds of the array. The problem is that, for a variable of pointer such as `p2`, the lower bound for the subscript through an indirection operation of the pointer cannot be provided explicitly in declaration of the variable according to its declaration specification. Therefore, for consistency, no lower bound of the subscript shall be specified in a pointer to assumed-shape array except when a pointer to assumed-shape array is declared at the function parameter scope, which will be described in the next section. All other constraints and semantics about a pointer to assumed-shape array described in the previous sections are still valid. For example, pointers to assumed-shape array can be used to access arrays allocated dynamically.

```
int funct(int n, int m) {
  double a[1:n][1:m];
  /* OK */
  double (*p1)[:] = a;
  double (*p2)[:] = (double [1:n][1:m])a;
  double (*p3)[:] = (double [1:n][1:m])malloc(n*m*sizeof(double));
  double (*p4)[:] = (double [1:][1:m])malloc(n*m*sizeof(double));
  double (*p5)[:] = (double [ ][1:m])malloc(n*m*sizeof(double));
  /* ERROR */
  double (*p6)[:] = (double (*)[1:m])malloc(n*m*sizeof(double));
}
```

In this example, the casting operation `(double [ ][1:m])` is the same as `(double [0][1:m])` or `(double [0:][1:m])`. A pointer to deferred-shape array is erroneously used in the last programming statement of `funct()`.

15.6 Passing Arrays with Explicit Lower and Upper Bounds to Functions

In this section, we will describe the linguistic features of passing arrays with explicit lower and upper bounds to functions. All syntax and semantics presented in this section will not break the C standard and existing C code.

15.6.1 Passing Arrays of Fixed Subscript Range

When passing arrays of fixed subscript range to a function, the actual passed array argument in the called function shall be compatible with the array argument declared at the function parameter scope. Both shall have compatible element types and the same shape. For example,

```
int a[1:3][1:5], b[0:3][1:5], c[3][1:5], d[1:3][1:6], e[1:3];
float f[1:3][1:5];
```

```
int funct(int aa[1:3][1:5]);
funct(a);   /* OK */
funct(b);   /* WARNING: incompatible first dimension */
funct(c);   /* incompatible first dimension c[i][j] == aa[i+1][j]
               no warning or error message  */
funct(d);   /* WARNING: incompatible second dimension */
funct(e);   /* WARNING: incompatible shape */
funct(f);   /* WARNING: incompatible data type */
```

When the lower bound of the subscript of an array is not present, the default value is 0. Although the first dimension of the array is incompatible in the function call `func(c)`, no warning message will be produced because a meaningful relation between arrays in the calling function and called function can be established if the extents of the associated arrays are the same. If the extents are different, a warning message will be generated for incompatibility. For example,

```
int a[3][1:5], b[0:2][1:5], c[1:3][1:5];
int funct1(int aa[3][1:5]);
int funct2(int (*bb)[1:5]);
funct1(a);  /* OK */
funct1(b);  /* OK */
funct1(c);  /* incompatible first dimension c[i+1][j] == aa[i][j]
               no warning or error message  */
funct2(a);  /* OK */
funct2(b);  /* OK */
funct2(c);  /* incompatible first dimension c[i+1][j] == bb[i][j]
                no warning or error message  */
```

An array name in the declaration of a function parameter is treated as a pointer to the first element of the array. However, an array name can be used to specify the lower bound of an array in the function parameter. The incomplete array type can be used at the function parameter scope. The incomplete array will be completed at the time of function call. For example,

```
int a[1:5], b[1:10], c[0:5], d[3];
int funct(int aa[1:]);
funct(a);   /* OK */
funct(b);   /* OK */
funct(c);   /* Ok */
funct(d);   /* OK */
```

It will be discussed in the next section that an incomplete one-dimensional array in the function parameter scope is treated as a pointer to assumed-shape array. Therefore, it is compatible to pass arrays of different subscript range to an incomplete one-dimensional array. The incomplete array type can be used for multi-dimensional arrays as well. The extents of the first dimension of the associated incomplete multi-dimensional arrays will not be checked for compatibility. For example,

```
int a[1:3][1:5], b[1:2][1:5], c[0:3][1:5], d[1:3][5], e[1:3][0:5];
int funct(int aa[1:][1:5]);
funct(a);   /* OK */
funct(b);   /* OK */
```

```
funct(c);   /* incompatible first dimension c[i][j] == aa[i+1][j]
               no warning or error message  */
funct(d);   /* incompatible second dimension d[i][j] == aa[i][j+1]
               no warning or error message  */
funct(e);   /* WARNING: incompatible second dimension */
```

When arrays of variable length are passed to arrays of fixed subscript range, the compatibility about shape could be checked at runtime. For example,

```
int n = 3, m = 4;
int a[1:n][1:m], b[n][1:m], c[0:n][1:m], d[1:n][0:m], e[1:n][1:m+1];
int funct(int aa[1:3][1:4]);
funct(a);   /* OK */
funct(b);   /* incompatible first dimension b[i][j] == aa[i+1][j]
               no warning or error message  */
funct(c);   /* WARNING: incompatible first dimension */
funct(d);   /* WARNING: incompatible second dimension */
funct(e);   /* WARNING: incompatible second dimension */
```

At the current implementation, the runtime checking is disabled. Therefore, the warning messages shown in the above program will not be produced. Because the shape of a pointer to assumed-shape array is assumed at execution time, the compatibility could also be checked at runtime. For the same reason, the warning messages are suppressed in the following sample code.

```
int n = 3, m = 4;
int a[1:3][1:4], b[3][1:4], c[0:3][1:4], d[1:n][1:m], e[1:n][0:m];
int (*p)[:];
int funct(int aa[1:3][1:4]);
p = a;
funct(p);   /* OK */
p = b;
funct(p);   /* incompatible first dimension p[i][j] == aa[i+1][j]
               no warning or error message  */
p = b;
funct(p);   /* WARNING: incompatible first dimension */
p = d;
funct(p);   /* OK */
p = e;
funct(p);   /* WARNING: incompatible second dimension */
```

15.6.2 Passing Arrays of Variable Subscript Range Using Pointers to Assumed-Shape Array

In the previous section, array shapes passed to a function are fixed except for the upper bound of the first dimension of the array passed to an incomplete array type. In this section, linkages for passing variable length arrays with explicit lower and upper bounds will be described.

To pass variable length arrays with variable subscript range to a function, a pointer to assumed-shape array can be used. At the function parameter scope, the following declaration, for a pointer to assumed-shape

array can be used:

```
T (*D)[lower:]      (15.18)
T (*D)[:]           (15.19)
T D[lower:]         (15.20)
T D[:]              (15.21)
```

where `T` contains the declaration specifiers that specify a type, `D` is a declarator that contains an identifier *ident*, and `lower` of constant integral type is the lower bound of the array. Declaration (15.20) allows specification of the lower bound of the first dimension of the array parameter in the function argument. If the lower bound is not present such as in declarations (15.19) and (15.21), the default value 0 is used. That is, `T (*D)[:]` is equivalent to `T (*D)[0:]` and `T D[:]` is equivalent to `T D[0:]`. All linguistic features about pointers to assumed-shape array described in the previous sections can be applied to pointers to assumed-shape array with explicit array bounds as if the lower bound were zero. Therefore, we only highlight new features related to explicit array bounds in the following presentation. An array name in the declaration of a function parameter is treated as a pointer to the first element of the array. In declaration (15.20), the lower bound of an array parameter of a function argument is specified. For example,

```
int funct1(int a[1:]);        // OK: pointer to assume-shape (pass)
int funct2(int a[1:][1:]);    // OK:
int funct3(int a[1:][:]);     // OK: pass a[1:][0:]
int funct4(int a[:][1:]);     // OK: pass a[0:][1:]
int funct5(int a[:][:]);      // OK: pass a[0:][0:]
int funct6(int a[][:]);       // OK: pass a[0:][0:]
int funct7(int a[][1:]);      // OK: pass a[0:][1:]
int funct8(int (*a)[1:]);     // OK: pass a[0:][1:]
int funct9(int (*a)[:]);      // OK: pass a[0:][0:]
int funct11(int a[0:]);       // OK:
int funct12(int a[]);         // OK: incomplete array type as pass
                              // the same as int funct11(int a[0:]);
int funct13(int a[:][5]);     // OK: incomplete array type a[0:][5]
int funct14(int a[1:][5]);    // OK: incomplete array type
int funct15(int a[1:][1:5]);  // OK: incomplete array type
int funct16(int a[1:][1:5]);  // OK: incomplete array type
/* ERROR: fixed-length array no upper bound */
int funct17(int a[1:5][1:]);
int funct18(int a[5][1:]);
int funct19(int a[1:][1:][5]);
int funct20(int a[:5]);       // ERROR: upper bound only
int funct21(int n, int m, int a[n:m]);// ERROR: deferred-shape array
int a[:], b[:][:];      // ERROR: not in function prototype scope
```

A one-dimensional incomplete array in a function parameter is treated as a pointer to assumed-shape array internally as shown in `funct12()` in the above example. But, one-dimensional incomplete arrays in external linkage and initialization are treated as fixed-length arrays.

Passing arrays of different lower bounds to a pointer to assumed-shape array is not considered to be incompatible. The upper bound of a pointer to assumed-shape array inside a called function will be adjusted at function call. The upper bound is the sum of the extent of the passed array and the lower bound of

the declared pointer to assumed-shape array at the function parameter. For example, in the following code fragment,

```
#define low 1
int n = 3, m = 5;
int a[n:m], b[n:2*m];
int funct(int aa[low:]);
funct(a); /* OK */
funct(b); /* OK */
```

the lower bound of array `aa` inside function `funct()` is 1 and the upper bound is 4, equal to `low+m-n+1` for the function call of `funct(a)`. For the function call of `funct(b)`, the lower bound of array `aa` inside function is still 1, but the upper bound becomes 9, equal to `low+2*m-n+1`.

The dynamic adjustment of the upper bound allows arrays of different subscript range to be passed to a function, which is not feasible using arrays of fixed subscript range described in the previous section. Using a pointer to assumed-shape array, only the upper bounds need to be explicitly passed to a function as additional parameters. This dynamic feature is useful for numerical computing. For example, when a FORTRAN function with arrays of unit-offset parameters are ported, the function can be called by passing both traditional C arrays with zero-offset and FORTRAN-style arrays with unit-offset. For example,

```
int n=3, m=4;
int a[n][m], b[1:2*n][1:2*m];
int funct(int aa[1:][1:], int n, int m) {
  int i, j;
  for(i=1; i<=n; i++)
    for(j=1; j<=m; j++)
      aa[i][j] += 2;
}
funct(a, n, m);           /* passing a[0:2][0:3] */
funct(b, 2*n, 2*m);       /* passing b[1:6][1:8] */
```

Similarly, a function with parameters of zero-offset array can be called with arguments of unit-offset array. For example,

```
int n=3, m=4;
int a[n][m], b[1:2*n][1:2*m];
int (*p)[:] = a;
int funct(int aa[:][:], int n, int m);
int funct(int   [:][:], int,   int  );   /* OK */
int funct(int bb[:][:], int l, int r);   /* OK */
int funct(int aa[:][:], int n, int m) {
  int i, j;
  for(i=0; i<=n-1; i++)
    for(j=0; j<=m-1; j++)
      aa[i][j] += 2;
}
funct(a, n, m);           /* passing a[0:2][0:3] */
funct(p, n, m);           /* passing a[0:2][0:3] */
funct(b, 2*n, 2*m);       /* passing b[1:6][1:8] */
```

```
#include <stdio.h>
int main() {
  int oldrlow = 0, oldrup = 3, oldclow = 0, oldcup = 5;
  int newrlow = 1, newrup = 4, newclow = 1, newcup = 6, i, j;
  double  a[oldrlow:oldrup][oldclow:oldcup], (*pa)[:];
  void funct(double aa[1:][1:], int rup, int cup);

  pa = (double [newrlow:newrup][newclow:newcup])a;
  for(i=oldrlow; i<=oldrup; i++)
    for(j=oldclow; j<=oldcup; j++)
      a[i][j] = 2;
  funct(pa,newrup,newcup);
  for(i=newrlow; i<=newrup; i++)
    for(j=newclow; j<=newcup; j++)
      printf("pa[i][j] = %f \n", pa[i][j]);
}

void funct(double aa[1:][1:], int rup, int cup) {
  int i, j;

  for(i=1; i<=rup; i++)
    for(j=1; j<=cup; j++)
      aa[i][j] += 2;
}
```

Program 15.2: Changing the array subscript ranges.

The above program also shows that variable length arrays such as deferred-shape arrays `a` and `b` and pointer to assumed-shape array `p` can be passed to pointer to assumed-shape array `aa` in the function argument. Different syntactic forms for function prototypes are used in the above example.

One common programming style in FORTRAN is to pass a segment of an array to a function by calling the function with an element of the array as an actual argument through call by reference. This type of FORTRAN code can be ported as shown in the following example.

```
int n=10;
double X[1:n];
void funct(double A[1:], int n);
funct(&X[5], n);
```

Elements `pa[i+1][j+1]` and `a[i][j]` refer to the same object in Program 15.2. The function call of

```
funct(pa,newrup,newcup);
```

in Program 15.2 can be replaced by either

```
funct(a,newrup,newcup);
```

or

```
funct((int [newrlow:newrup][newclow:newcup])a,newrup,newcup);
```

Chapter 16

Computational Arrays and Matrix Computations

Arrays in C are intimately tied with pointers. For the comparison purpose, these arrays are called *C arrays*. For numerical computing and data analysis, *computational arrays* which are first-class objects with more information are introduced in Ch. Many operators including arithmetic operators are overloaded to handle computational arrays.

If $\mathbf{A}_1$ and $\mathbf{A}_2$ wto two arrays, in general, array expression $\mathbf{A}_1/\mathbf{A}_2$ is undefined mathematically in linear algebra. However, $\mathbf{A}_1/\mathbf{A}_2$ is defined as an element-wise division in Fortran 90 whereas in MATLAB it is defined as the product of $\mathbf{A}_1$ and the inverse matrix of $\mathbf{A}_2$, that is, $\mathbf{A}_1/\mathbf{A}_2$ is the same as $\mathbf{A}_1\mathbf{A}_2^{-1}$. This kind of operator overloading for division is quite confusing for learners of linear algebra. This may lead leaners to use the expression $\mathbf{x} = \mathbf{b}/\mathbf{A}$ as a solution to the system of linear equations $\mathbf{A}\mathbf{x} = \mathbf{b}$. To avoid such a mistake, one of the guiding principles in designing Ch is to follow the mathematical conventions. For example, the element-wise division of two matrices $\mathbf{A}_1$ and $\mathbf{A}_2$ with the same rank is programmed in Ch as `A1./A2` and the product of $\mathbf{A}_1$ and the inverse of matrix $\mathbf{A}_2$ is written as `A1*inverse(A2)`. The expression $s = \mathbf{v}^T\mathbf{A}\mathbf{v}$ is translated into `transpose(v)*A*v` in Ch.

The notations used in this chapter are listed in Table 16.1. A digital number may follow a symbol for multiple variables. For example, symbols **V**, **V1** and **V2** are used for vectors; symbols **A**, **A1** and **A2** stand for vectors, matrices, or high-dimension arrays.

16.1 Declaration and Initialization of Computational Arrays

The extent and range of subscripts for each dimension are fully specified for a fully-specified-shape array. The computational arrays are declared with type qualifier **array**, which is defined as a macro in the header file **array.h**. The program using computational arrays should include this header file. In the command mode, the type qualifier **array** is defined as an alias by default. The computational arrays below are fully specified.

```
array int a1[10];             // a1[0], ..., a1[9]
array int a2[0:9];            // a2[0], ..., a2[9]
array int a3[1:10];           // a3[1], ..., a3[10]
array double  a4[10][10];     // a4[0:9][0:9]
array complex a5[1:10][1:10]; // a5[1:10][1:10]
```

where symbol '`:`' is used to specify the range of the subscripts of arrays. By default, it is from 0 to `n-1`, where `n` is the number in the operator **[]** to specify the size of the array.

Table 16.1: Shape and data type notations.

Symbol	Meaning
	Shape
A	vector, matrix, or high-dimension arrays of char, int, float, double, complex, or double complex
B	vector, matrix, or high-dimension arrays of char, int, float, double
I	vector, matrix, or high-dimension arrays with integral data types of char, int
M	two-dimension matrix of char, int, float, double, complex, or double complex
V	ond-dimension vector of char, int, float, double, complex, or double complex
a	scalar of char, int, float, double, complex, or double complex
	Data type
b	bool
c	char
s	short
i	int
f	float
d	double
z	complex
p	higher order data type of operands in operations or arguments in functions
k	the same data type of the original operand or argument
m	the same data type of the original operand or argument, double if the data type of the original operand or argument is char or int
	Data type modifier
u	unsigned
l	long

If two computational arrays have the same number of elements in each dimension, the assignment operator '=' can be used to assign arrays element-wise as shown in the execution of the commands below.

```
> array double a[0:3]
> array int b[4] = { 0, 1, 2, 3}
> a = b
0.00 1.00 2.00 3.00
>
```

Computational arrays can be initialized when they are declared in the same manner as C arrays. By default, computational arrays are initialized to zeros. For example,

```
array int a1[3] = {1, 2, 3};
array int a2[3]= { 2.3e3d, 2.2F, 3.D }; // a2 = {1,2,3}, data cast
array int a3[] = {0.0, -0.0, -0.0};     // a3 = {0.0, -0.0, -0.0}
array double a4[][3] = {{1, 2, 3}, {1, 2, 3}};
array double a5[3][3] = {1, 2, 3, 1, 2, 3};
```

16.2 Array Reference

16.2.1 Whole Arrays

The name of a computational array can be used to access a whole array. For example, the following code fragment

```
array int a[20], b[20];
b = a+b;
```

adds each element of `a` to the corresponding element of `b`. Arrays `a` and `b` are treated as vectors, just like in linear algebra. This feature makes programs much simpler compared to programs using normal C arrays. As an example, Programs 16.1 and 16.2 perform the same task of adding array `a` element-wise and multiply it by 3, and print out the results. Program 16.1 uses computational arrays whereas Program 16.2 doesn't. Clearly, Program 16.2 contains less lines of code and is more readable and easier to maintain. Note that the **array** qualifier is defined as a macro in header file **array.h**. In order to use the computational array, the program should include this header file. The output for these programs are the same and given below.

```
b =
2 4 6
8 10 12
b =
3 6 9
12 15 18
```

16.2.2 Array Elements

Similar to C arrays, the operator `[n]` can be used to access elements of computational arrays, where `n` is a valid subscript. For example, the following code fragment

```
array int a[20], b[20];
b[1] = a[2]+b[2];
```

```
/* File: declare.ch */
#include <stdio.h>
#include <array.h>
#define N 2
#define M 3

int main() {
    array int a[N][M] = {1,2,3,
                         4,5,6};
    array int b[N][M];

    b = a+a;
    printf("b = \n%d", b);
    b = 3*a;
    printf("b = \n%d", b);
    return 0;
}
```

Program 16.1: Declaring and using computational arrays.

```
/* File: declare.c */
#include <stdio.h>
#define N 2
#define M 3

int main() {
  int a[N][M] = {1,2,3,
                 4,5,6};
  int b[N][M];
  int i, j;

  printf("b = \n",);
  for(i=0; i<N; i++) {
    for(j=0; j<M; j++) {
       b[i][j] = a[i][j]+a[i][j];
       printf("%d ", b[i][j]);
    }
    printf("\n",);
  }
  printf("b = \n",);
  for(i=0; i<N; i++) {
    for(j=0; j<M; j++) {
       b[i][j] = 3*a[i][j];
       printf("%d ", b[i][j]);
    }
    printf("\n",);
  }

  return 0;
}
```

Program 16.2: Implementing program declare.ch in C.

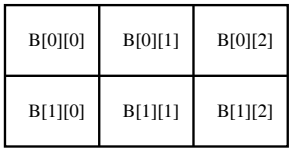


Figure 16.1: Computational array B.

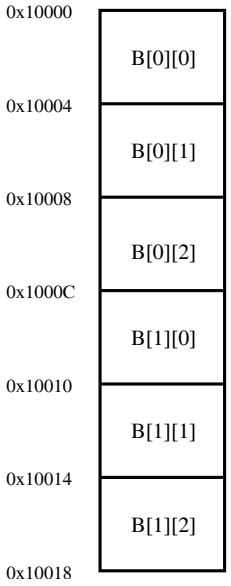


Figure 16.2: Memory layout of two-dimension computational array B.

adds the third element of a to the third of b, and saves the result to the second element of b.

Like C arrays, computational arrays are also row-wise. For example, for computational array B declared below,

```
array int B[2][3];
```

assume the address of computational array B of dimension 2x3 shown in Figure 16.1 is 0x10000, the internal memory layout of array B is shown in Figure 16.2.

16.3 Formatted Input and Output for Computational Arrays

Like C arrays, the input of computational arrays can be handled by the function **scanf()** element by element. For example,

```
> array int  a[2]
> scanf("%d", &a[0])
10
> a
10 0
```

The computational array can also be conveniently handled by the function **scanf()** for the entire array. If the data type of the array is not char or unsigned char type, input numbers can be separated by one or multiple of white space, chararacters ' ' ',' ';' ':' or a newline character. For example,

```
> array int  b[6]
> scanf("%d", &b)
10 11, 12
13; 14: 15
> b
10 11 12 13 14 15
```

The family of output functions **fprintf()**, **sprintf()**, **printf()**, etc. can be used to print out all elements of a computational array once. The format specifier will be applied to each element of the array. For example,

```
> array int a[3] = {1,2,3}
> array int b[2][3] = {1,2,3,4,5,6}
> printf("a = %d", a);
a = 1 2 3
> printf("b = \n%d", b);
b =
1 2 3
4 5 6
```

For computational arrays with large extents, 74 characters including elements of arrays and delimiting spaces at each line will be printed out. For example, each element of array `a` below has the same value of 90. The output is wrapped in the subsequent line beyond 74 characters.

```
> array int a[2][50] = 90
> a
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
>
```

A multi-dimensional array will be printed out in multiple two-dimensional arrays with the rows and columns of the last two extents of the array as shown below.

```
> array int a[2][2][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
> a
1 2 3
4 5 6

7 8 9
10 11 12
```

By default, a one-dimensional array is a column vector in Ch. For a one-dimensional array of a column or row vector, the output will be printed out as a row vector even if it is a column vector. For example,

```
> array int a[3] = {1,2,3}
> a                 // column vector
1 2 3
> transpose(a)      // row vector
1 2 3
```

The vector `a` is a column vector, the transpose of `a`, `transpose(a)` is a row vector. When they are printed out, both are displayed as row vectors.

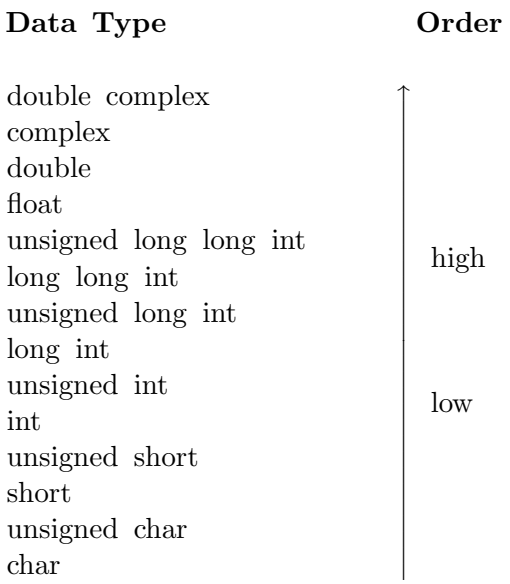


Figure 16.3: Data type hierarchy.

16.4 Implicit Data Type Conversion for Computational Arrays

In computational array operations, the data types of operands will be checked for compatibility. If data types do not match, Ch will signal an error and print out some informative messages for the convenience of program debugging. However, some data type conversion rules have been built into Ch so that they can be invoked whenever necessary. This will save many explicit type conversion commands for a program. The order of the data type for computational array is arranged as shown in Figure 16.3 with char being the lowest data type and double complex the highest data type. The default conversion rules are summerized as follows.

1. Arrays of char, int, float, and double can be converted according to data conversion rules of the corresponding scalar types.

2. Arrays of char, int, float, and double can be converted to arrays of complex with the imaginary part of each element being zero. When casting an array of real number to an array of complex number, the values of elements of Inf and $-$Inf become ComplexInf, and the values of elements of NaN become ComplexNaN. Conversion from array of double to array of complex may lose information.

3. In binary operations such as addition, subtraction, multiplication, and division with arrays of mixed data types, the result of the operation will carry the higher data type of two operands. For example, the result of addition of an array of int and an array of double will result in an array of double.

The following code segment will illustrate how arrays with different data types are automatically converted.

```
> array int i[2] = {1, 2}
> array float f[2]
> array double d [2]
> f = i            // float = int
1.00 2.00
```

```
> d = f + i        // double = float + int
2.0000 4.0000
>
```

For operation `d = f + i`, elements of arrays `f` and `i` of float and int types, respectively, are added with the result of a computational array of float type. The resultant computational array of float type is then cast to a computational array of double type and assigned to variable `d` of computational array of double type. Data type conversion for various array operations are discussed in detail in Section 16.5.

16.5 Array Operations

16.5.1 Arithmetic Operations

The arithmetic operations for computational arrays are listed in Table 16.2. The symbol $\mathbf{A}/\mathrm{k}$ in the third column of Table 16.2 indicates that the results are arrays with the same shape and data type of the operand. For the symbol $\mathbf{A}/\mathrm{p}$, the result is the same shape and higher order of data type of two operands. These symbols are described in Table 16.1. The arithmetic operations include unary plus operator '`+`', unary minus operator '`-`', addition operator '`+`', subtraction operator '`-`', multiplication operator '`*`', division operator '`/`', array multiplication operator '`.*`' and array division operator '`./`'. The operator '`*`' is for multiplication of two arrays of one-dimensional vectors or two-dimensional matrices. The multiplication of two arrays follows the rule of linear algebra. For element-wise array multiplication operator '`.*`' and array division operator '`./`', the operation is performed on each corresponding element of two array operands, which shall be of the same shape (dimension and extent).

The data type of the result of the operation of unary plus operator or unary minus operator is the same as that of the operand. The resulting data types of other operations in Table 16.2 will have the higher order data type of the operands in operations. If one of the operands of addition or subtraction operator is a scalar and the other is a computational array, the scalar will be promoted to a computational array for the corresponding array operation. If the numerator of the array division operator '`./`' is a scalar, it will be promoted to a computational array.

Applications of these operations are illustrated in the commands below. For example,

```
> array int a1[2][2] = {1, 0, 2, 3}
> array int a2[2][2] = {0, 5, 2, 2}
> float s = 2.0
> a1 * a2
0 5
6 16
> a1 .* a2
0 0
4 6
> a1/s
0.50 0.00
1.00 1.50
> a1 +2
3 2
4 5
```

For multiplication of two arrays, the dimensions of the arrays have to follow the rule of linear algebra as shown below.

Table 16.2: Array arithmetic operation.

Definition	Operation	Result
unary plus	$+\mathbf{A}$	$\mathbf{A}$/k
unary minus	$-\mathbf{A}$	$\mathbf{A}$/k
addition	$\mathbf{A1} + \mathbf{A2}$	$\mathbf{A}$/p
addition	$\mathbf{A} + [s]$	$\mathbf{A}$/p
addition	$[s] + \mathbf{A}$	$\mathbf{A}$/p
subtraction	$\mathbf{A1} - \mathbf{A2}$	$\mathbf{A}$/p
subtraction	$\mathbf{A} - [s]$	$\mathbf{A}$/p
subtraction	$[s] - \mathbf{A}$	$\mathbf{A}$/p
multiplication	$\mathbf{A1} * \mathbf{A2}$	$\mathbf{A}$/p or a/p
multiplication	$\mathbf{A} * s$	$\mathbf{A}$/p
multiplication	$s * \mathbf{A}$	$\mathbf{A}$/p
division	$\mathbf{A}/s$	$\mathbf{A}$/p
array multiplication	$\mathbf{A1}. * \mathbf{A2}$	$\mathbf{A}$/p
array division	$\mathbf{A1}./\mathbf{A2}$	$\mathbf{A}$/p
array division	$[s]./\mathbf{A2}$	$\mathbf{A}$/p

```
> array int a1[2][3] = {1, 2, 3, 4, 5, 6}
> array int a2[3][2] = {1, 2, 3, 4, 5, 6}
> array int b[3] = {1, 2, 3}
> a1*a2
22 28
49 64
> a1*b
14 32
> a1*a1
ERROR: array dimensions do not match for matrix operations
```

As a special case, the result from multiplication of two arrays is a scalar instead of an array, if the shapes of **A1** and **A2** are $(1 \times n)$ and $(n \times 1)$, where n is 1, 2, 3, ... For example,

```
> int i
> array int a[1] = {10}
> array int b[1] = {20}
> array int c[2] = {1, 2}
> i = a * b  // (1x1) * (1x1), the result is a scalar
200
> b = a + b  // it's an array
30
> transpose(c) * c // (1x2) * (2x1), the result is a scalar
5
> c * transpose(c) // (2x1) * (1x2), the result is an array
1 2
2 4
```

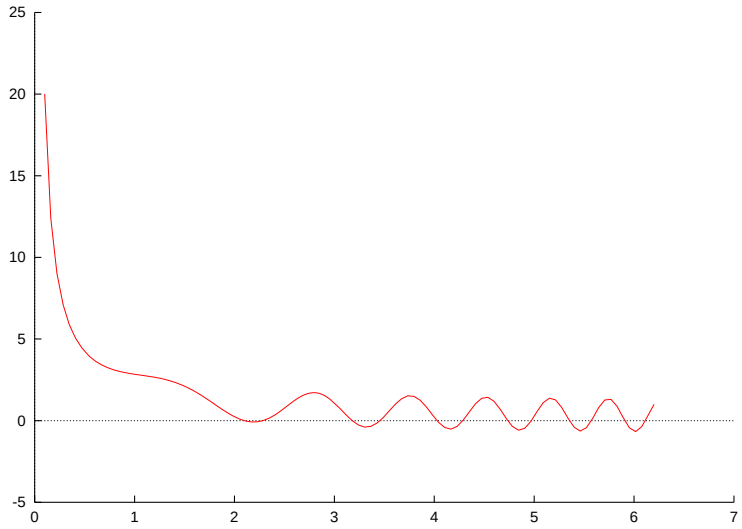


Figure 16.4: Function $y(x) = 2/x + \sin(x^2)$.

The result of `a * b` is an integer, so is `transpose(c) * c`. The one-dimensional array by default is a column vector with the shape of $(n \times 1)$ at declaration and calculation. For example, `c` has the shape of (2×1) instead of (1×2).

Array multiplication operator '`.*`' and array division operator '`./`' are useful to handle formulas without loops such as for-loop and while-loop. For example, the plot of function $y(x) = 2/x + \sin(x^2)$ in the range of $0.1 \leq x \leq 6.2$ with 100 points can be created as follows.

```
> array double x[100], y[100]
> lindata(0.1, 6.2, x)
> y =  2.0./x +sin(x.*x)
> plotxy(x, y)
```

The output of a plot is displayed in Figure 16.4. Function call of **lindata(0.1, 6.2, x)** assigns linearly spaced values starting with 0.1 and ending with 6.2 for elements of array *x*. Details about function **lindata()** and generic mathematical function **sin()** for handling arguments of array type will be described later. Note that for computational array `x`, expression `2./x` is interpreted as `2.0/x`, not array operation `2 ./ x`. Therefore, `2./x` is invalid because of unmatched array dimensions.

16.5.2 Assignment Operations

The assignment operations for computational arrays are listed in Table 16.3. They include simple assignment '`=`' and compound assignments which include assign sum operator '`+=`', assign difference operator '`-=`', assign product operator '`*=`', and assign quotient operator '`/=`'. The data types of the results of operations of these operators are the same as those of the left operands.

Applications of these operations are illustrated in the commands below.

```
> array int a1[4] = {1, 0, 2, 3}
> array int a2[4] = {0, 5, 2, 2}
> a1 += a2
1 5 4 5
```

Table 16.3: Array assignment operation.

Definition	Operation	Result
assignment	**A1=A2**	**A**/k
assign	**A**=$[s]$	**A**/k
assign sum	**A1+=A2**	**A**/k
assign difference	**A1-=A2**	**A**/k
assign product	**A1*=A2**	**A**/k
assign product	**A1***=s	**A**/k
assign quotient	**A1/**=s	**A**/k

Table 16.4: Array increment and decrement operation.

Definition	Operation	Result
plus	**A++**	**A**/k
plus	**++A**	**A**/k
minus	**A--**	**A**/k
minus	**--A**	**A**/k

16.5.3 Increment and Decrement Operations

The increment and decrement operations for computational arrays are listed in Table 16.4. They include increment operator '++' and decrement operator '--', which add 1 to and subtract 1 from each element of the array, respectively. The resulting data type of these operations are the same as those of the original operands.

Applications of these operations are illustrated in the commands below.

```
> array int a1[4] = {1, 0, 2, 3}
> array int a2[4] = {0, 5, 2, 2}
> a1++
1 0 2 3
> a1
2 1 3 4
> --a2
-1 4 1 1
```

In most cases, a computational array has a rank of 1 or higher. In some situations, a computational array can have value of NULL. Before it is allocated memory, a pointer to computational array has a value of NULL. A value of NULL can also be passed to an argument of array of reference type in a function. A computational array with value of NULL can be used as an operand of equal operator '==' or not equal operator '!='. They cannot be used as an operand for other operations. If one of two operands for equal operator '==' or not equal operator '!=' is pointer to computational array or array of reference, the other operand can be NULL. The result of the operation in this case is a boolean type of either **true** or **false**. This can be used to test if NULL has been passed to array of reference or if a pointer to computational array points to a valid object. Details about pointer to computational array and computational array of reference will described later.

In Programs 16.11, NULL is passed to the argument `a` of array of reference in function `func()`. In Program 16.12, variable `a` of pointer to computational array has a default value of NULL before it is pointed to an array. The output of these two programs are the same as shown below.

```
a==NULL is true
a!=NULL is false
```

16.5.4 Relational Operations

The relational operations for computational arrays are listed in Table 16.5. They include the less than operator '<', less than equal operator '<=', equal operator '==', greater than equal operator '>=', greater than operator '>', and not equal operator '!='. Using these operators results in an array of int type, with values of either 0 or 1, depending on how each element of the array compares. For these binary operators, if one of operands is a computational array and the other is a scalar, the scalar will be promoted to a computational array with the shape of the array operand. Applications of these operations are illustrated in the commands below.

```
> array int a1[4] = {1, 0, 2, 3}
> array int a2[4] = {0, 5, 2, 2}
> a1 < a2
0 1 0 0
> a1 >=  a2
1 0 1 1
```

In most cases, a computational array has a rank of 1 or higher. In some situations, a computational array can have value of NULL. Before it is allocated memory, a pointer to computational array has a value of NULL. A value of NULL can also be passed to an argument of array of reference type in a function. A computational array with value of NULL can be used as an operand of equal operator '==' or not equal operator '!='. They cannot be used as an operand for other operations. If one of two operands for equal operator '==' or not equal operator '!=' is pointer to computational array or array of reference, the other operand can be NULL. The result of the operation in this case is a boolean type of either **true** or **false**, which can be used as a controlling expression of **if**-statement to test if NULL has been passed to array of reference or if a pointer to computational array points to a valid object. Details about pointer to computational array and array of reference will be described later.

16.5.5 Logic Operations

The logic operations for computational arrays are listed in Table 16.6. They include the AND operator'&&', XOR operator '^^', OR operator '||', and NOT operator '!'. The results of evaluating with these operators are arrays of int type, with values of either 0 or 1. For these binary operators, if one of operands is a computational array and the other is a scalar, the scalar will be promoted to a computational array with the shape of the array operand.

Applications of these operations are illustrated in the commands below.

```
> array int a1[4] = {1, 0, 2, 3}
> array int a2[4] = {0, 5, 2, 2}
> a1 && a2
0 0 1 1
> a1 ||  a2
1 1 1 1
```

Table 16.5: Array relational operation.

Definition	Operation	Result
less than	$\mathbf{B1} < \mathbf{B2}$	**I**/i
less than	$\mathbf{B1} < [s]$	**I**/i
less than	$[s] < \mathbf{B2}$	**I**/i
less equal	$\mathbf{B1} <= \mathbf{B2}$	**I**/i
less equal	$\mathbf{B1} <= [s]$	**I**/i
less equal	$[s] <= \mathbf{B2}$	**I**/i
equal	$\mathbf{A1} == \mathbf{A2}$	**I**/i
equal	$\mathbf{A1} == [s]$	**I**/i
equal	$[s] == \mathbf{A2}$	**I**/i
equal	$NULL == \mathbf{A1}$	b
equal	$\mathbf{A1} == NULL$	b
greater equal	$\mathbf{B1} >= \mathbf{B2}$	**I**/i
greater equal	$\mathbf{B1} >= [s]$	**I**/i
greater equal	$[s] >= \mathbf{B2}$	**I**/i
greater than	$\mathbf{B1} > \mathbf{B2}$	**I**/i
greater than	$\mathbf{B1} > [s]$	**I**/i
greater than	$[s] > \mathbf{B2}$	**I**/i
not equal	$\mathbf{A1}$!= $\mathbf{A2}$	**I**/i
not equal	$\mathbf{A1}$!= $[s]$	**I**/i
not equal	$[s]$!= $\mathbf{A2}$	**I**/i
not equal	$\mathbf{A1}$!= $NULL$	b
not equal	$NULL$!= $\mathbf{A1}$	b

Table 16.6: Array logic operation.

Definition	Operation	Result
AND	**A1** && **A2**	**I**/i
AND	**A1** && $[s]$	**I**/i
AND	$[s]$&& **A2**	**I**/i
XOR	**A1** ^^ **A2**	**I**/i
XOR	**A1** ^^ $[s]$	**I**/i
XOR	$[s]$^^ **A2**	**I**/i
OR	**A1** || **A2**	**I**/i
OR	**A1** || $[s]$	**I**/i
OR	$[s]$|| **A2**	**I**/i
NOT	!**A**	**I**/i

16.5.6 Conditional Operation

The conditional operator '`?:`' can be applied to computational arrays in Ch. If this is the case, the first operand of a conditional expression shall have scalar type, and the other two operands are computational array of the same shape. The result is a computational array with the higher order type of these two operands.

Applications of the conditional operation are illustrated in the commands below.

```
> array int a[2][3] = 1, b[2][3]=2
> array float f[2][3] = 3.0
> 1 ? a:b // operands of array
1 1 1
1 1 1
> 0 ? f:b  // operands of array
3.00 3.00 3.00
3.00 3.00 3.00
```

In these two examples, both the second and third operands have the same shape, which are (2×3) and (2×2), respectively. The result of the latter example is a computational array of type float, since the type of the second operand is float, which has higher order than int type of the third operand.

16.5.7 Address Operations

The address operator '`&`' can also be used to get the address of a computational array or the address of an element of a computational array. The commands below illustrate how the address operator works.

```
array int a[0:9], b[2][3];
int *ptr;
ptr = &a;        // the address of a
ptr = &a[2]      // the address of third element of a
ptr = &b;        // the address of b
ptr = &b[1][2];  // the address of an element of b
```

The address operator '`&`' applied to a computational array gives the address of the first element of the array. For the sample commands below, `&a` gives the address of `a[0][0]` and `&b` gives the address of `b[0][0]`. So, if the memory for a pointer to computational array, which will be described later, has not been allocated, the address operation gives NULL. Furthermore, the address operator '`&`' before an element of a computational array gives the address of this element. For example, `&b[1][0][0]` gives the address of `b[1][0][0]` as shown below.

```
> array int a[2][2] = {1, 2, 3, 4}
> array int b[2][2][2] = {1, 2, 3, 4, 5, 6, 7, 8}
> &a
4005e3e0
> &a[0][0]            // same as &a
4005e3e0
> &b
4005e4e0
> &b[0][0][0]         // same as &b
4005e4e0
> &b[1][0][0]
4005e4f0
```

16.5.8 Cast Operations

Because Ch allows array operations of computational arrays with different types or even operations with computational arrays and C arrays, the cast operation is sometimes important to prevent confusion.

Below are some examples of cast operations for computational arrays.

```
array double a[3][1], b[3], c[4][3];
array int d[3][1];
a = (array double [3][1])b;      // cast [3] to [3][1]
b = (array double [3])a;         // cast [3][1] to [3]
b = (array double [3])&c[1][0];  // cast 2nd row of c to vector b
b = (array double [3])&c[2][0];  // cast 3rd row of c to vector b
c = (array double [4][3])4;      // cast scalar to array
d = (array int [3][1])a;         // cast double to int
```

Through cast operations, the assignment operations can be performed for two computational arrays. For example, the extent of the last dimension of array `c` is the same as array `b`. Although `a` has the different extent in the last dimension, it also has the same amount of memory of array `b`. Note that scalars may be cast as computational arrays. The statement `c = (array double [4][3])4` above will set all the elements of array `c` to 4. It is also possible to cast computational arrays of one data type to another, such as in the last operation above.

If the number of array elements the cast operation is smaller than the number of array elements of the operand, the extra elements of the operand are ignored. If the number of array elements the cast operation is larger than the number of array elements of the operand, the remaining elements of the resulting array are filled with 0's. For example,

```
> array double a[3] = {1,2,3}
> (array int [2])a
1 2
> (array int [4])a
1 2 3 0
```

The casting operator preceding an array can give the address or value of the first element of the array. If the type is a pointer, it gives the address of the first element of the array. Otherwise, it gives the value of the first element. For example

```
> array int a[2][2] = {1, 2, 3, 4}
> (int *)a
4005ef10
> &a
4005ef10
> (int)a
1
```

16.6 Promotion of Scalars to Computational Arrays in Operations

A scalar value can be cast to a computational array explicitly. But, the scalar operand will be promoted to a computational array implicitly for addition, subtraction, array division, assignment, logic and relational operations, if the other operand is a computational array. An array promotion is used for operations with two arrays operands, i.e. the operand of a scalar in the operation internally is treated as an array in which

Table 16.7: Array promotions.

Definition	Operation	Promotion	Result
assignment	$\mathbf{A} = s$	$\mathbf{A} = [s]$	**A**/k
addition	$\mathbf{A} + s$	$\mathbf{A} + [s]$	**A**/p
addition	$s + \mathbf{A}$	$[s] + \mathbf{A}$	**A**/p
subtraction	$\mathbf{A} - s$	$\mathbf{A} - [s]$	**A**/p
subtraction	$s - \mathbf{A}$	$[s] - \mathbf{A}$	**A**/p
division	s ./ **A**	$[s]$ ./ **A**	**A**/p
less than	$\mathbf{B} < s$	$\mathbf{B} < [s]$	**I**/i
less than	$s < \mathbf{B}$	$[s] < \mathbf{B}$	**I**/i
less equal	$\mathbf{B} <= s$	$\mathbf{B} <= [s]$	**I**/i
less equal	$s <= \mathbf{B}$	$[s] <= \mathbf{B}$	**I**/i
equal	$\mathbf{A} == s$	$\mathbf{A} == [s]$	**I**/i
equal	$s == \mathbf{A}$	$[s] == \mathbf{A}$	**I**/i
greater equal	$\mathbf{B} >= s$	$\mathbf{B} >= [s]$	**I**/i
greater equal	$s >= \mathbf{B}$	$[s] >= \mathbf{B}$	**I**/i
greater than	$\mathbf{B} > s$	$\mathbf{B} > [s]$	**I**/i
greater than	$s > \mathbf{B}$	$[s] > \mathbf{B}$	**I**/i
not equal	**A** != s	**A** != $[s]$	**I**/i
not equal	s != **A**	$[s]$!= **A**	**I**/i
XOR	**B** ^^ s	**B** ^^ $[s]$	**I**/i
XOR	s ^^ **B**	$[s]$ ^^ **B**	**I**/i
OR	**B** || s	**B** || $[s]$	**I**/i
OR	s || **B**	$[s]$ || **B**	**I**/i
AND	**B** && s	**B** && $[s]$	**I**/i
AND	s && **B**	$[s]$ && **B**	**I**/i

the value of each element equals this scalar value. In some cases, array promotions make the programming easier. Consider the following statements, where 2 is added to each element of computational array a with a single statement. If a were a regular C array, the process would require some sort of loop.

```
> array int a[2][2] = {1, 0, 2, 3}
> a1 + 2                  // 2 is promoted to array
3 2
4 5
```

Table 16.7 provides a list of operations with implicit array promotion.

16.7 Passing Computational Arrays to Functions

There are four different methods to passing computational arrays to functions. These methods are listed in Table 16.8 along with brief descriptions about their characteristics. A fixed dimension means that only arrays of a specified dimension may be passed into a function. For the sample codes shown in the table, only two dimensional array arguments are allowed unless the fourth method, *array of reference*, is used. Using this method, arrays of any dimensions may be passed to an argument of a function. The extent of an

Table 16.8: Methods for passing computational arrays to functions.

Method	Sample Code	Dimension	Extent	Data Type
Fully-specified arrays	`array double a[2][3]`	fixed	fixed	fixed
Assumed-shape arrays	`array double a[:][:]`	fixed	variable	fixed
Variable number argument	type1 func(type2 a, ...)	variable	variable	variable
Array of reference of fixed dimension	`array double a[&][&]`	fixed	variable	variable
Array of reference	`array double &a`	variable	variable	variable

array argument refers to the number of elements in a dimension. Aside from the fully-specified arrays, all other array argument types have variable extents. Thus the number of elements may vary for these types of arguments. For fully-specified and assumed-shape arrays, the data types have to be fixed when passing computational arrays to functions, whereas the other two does not.

16.7.1 Fully-Specified-Shape Arrays

Program 16.3 demonstrates how fully-specified-shape arrays are used as arguments of a function. In Program 16.3, the function `sum1()` with arguments of fully-specified-shape arrays is called to calculate the matrix expression of dimension 2x3

$$\mathbf{b} = \mathbf{a} + \mathbf{2} * \mathbf{a}, \tag{16.1}$$

and returns the sum of values for each element of array `a`. If the argument of a function is defined as a fully-specified-shape array, addresses of arrays are passed to this function. The output of Program 16.3 is displayed in Figure 16.5

16.7.2 Assumed-Shape Arrays

The arguments of the function `sum1()` in the previous example are declared as fully-specified-shape arrays. This is not flexible to handle arrays with different extents in each dimension. Ch provides assumed-shape arrays to deal with arrays of variable length. If arguments are declared as assumed-shape arrays, they can take arrays which have the same dimension but different number of elements in each dimension.

Assumed-shape arrays declared with a colon as array subscripts are shown below.

```
int funct1(array int a[:][:], b[:]);
int func2(array double c[:]);
```

We can rewrite Program 16.3 to use functions with arguments of assumed-shape arrays. In Program 16.4, the function `sum2()` which takes two arguments of assumed-shape array is called to calculate the same matrix expression

$$\mathbf{b} = \mathbf{a} + \mathbf{2} * \mathbf{a}, \tag{16.2}$$

and also returns the sum of value for each element of array `a`. The output of Program 16.4 is displayed in Figure 16.6

If the argument of a function is defined as as an assumed-shape array, not only addresses but also boundaries of arrays are passed to this function. So, arrays with different numbers of elements in each dimension can be passed to the same function. For example, in Program 16.4, arrays `a1` and `a2` have the same dimension, but the extents are different. They can be passed to the same argument of function `sum2()`. Similarly, arrays `b1` and `b2` of different extents are also passed to the same argument. The output

```
/* File: sum1.ch */
#include <array.h>
#define N 2
#define M 3

double sum1(array double a[N][M], array double b[N][M]){
    double sum = 0;
    int i, j;

    b = a + 2 * a;  // b = 3*a
    for(i=0; i<N; i++)
      for(j=0; j<M; j++)
        sum += a[i][j];
    return sum;
}

double main() {
    double sum;
    array double b1[N][M], a1[N][M] = {1, 2, 3,
                                       4, 5, 6};

    sum = sum1(a1, b1);
    printf("b1 = \n%g", b1);
    printf("sum = %g\n",  sum);
    return 0;
}
```

Program 16.3: Passing computational arrays of fixed shape and data type.

```
b1 =
3 6 9
12 15 18
sum = 21
```

Figure 16.5: Output of Program 16.3.

```
/* File: sum2.ch */
#include <array.h>

double sum2(array double a[:][:], array double b[:][:]){
    int n = shape(a, 0), m = shape(a, 1);
    /* or array int dim[2] = shape(a);
          int n = dim[0], m = dim[1]; */
    double sum = 0;
    int i, j;

    printf("n = %d, m = %d\n", n, m);
    b = a + 2 * a;  // b = 3*a
    for(i=0; i<n; i++)
       for(j=0; j<m; j++)
          sum += a[i][j];
    return sum;
}

double main() {
    double sum;
    array double b1[2][3], a1[2][3] = {1, 2, 3,
                                       4, 5, 6};
    array double b2[3][4], a2[3][4] = {1, 2, 3, 4,
                                       5, 6, 7, 8,
                                       9, 10, 11, 12};

    sum = sum2(a1, b1);
    printf("b1 = \n%g", b1);
    printf("sum = %g\n\n",  sum);
    sum = sum2(a2, b2);
    printf("b2 = \n%g", b2);
    printf("sum = %g\n",  sum);
    return 0;
}
```

Program 16.4: Passing computational arrays of different shapes and fixed data type.

```
n = 2, m = 3
b1 =
3 6 9
12 15 18
sum = 21

n = 3, m = 4
b2 =
3 6 9 12
15 18 21 24
27 30 33 36
sum = 78
```

Figure 16.6: Output of Program 16.4.

of Program 16.4 is displayed in Figure 16.6. The generic function **shape()** can be used to obtain the extent of each dimension of the assumed-shape array. If a single argument of function **shape()** is of array type, it returns its shape as a computational array of int type as if the function was prototype as

```
array int shape(array type [:]...[:])[:];
```

where `type` can be any valid type for computational array. If the argument of function **shape()** is a one-dimensional array, the return value is a computational array of size 1x1. Thus the return value may be cast to a scalar. Function **shape()** can also also be used to obtain the extent of a specified dimension for an array. In this case, it acts as if the function was prototyped as

```
int shape(array type [:]...[:], int index);
```

For example,

```
> array int a[3][4], b[5]
> shape(a)
3 4
> shape(a, 0)
3
> shape(a, 1)
4
> shape(b)
5
> (int)shape(b)    // cast 1x1 array to scalar
5
> (int)shape(shape(a))
2
```

Function call `shape(b)` in the above function returns a computational array of size 1x1. It can be cast to a scalar by expression `(int)shape(b)`. Similarly, a scalar value can be obtained from the expression `(int) shape(shape(a))`.

16.7.3 Deferred-Shape Arrays

Ch supports deferred-shape computational arrays, which is another way to handle arrays with different numbers of elements in each dimension at run time. For a deferred-shape array, the array subscript of integral expression is evaluated at run time. Examples of declaration of deferred-shape arrays are shown below,

```
array int A[n][m], B[m];
array double C[m];
```

where `n` and `m` are variables of **int** type.

Program 16.5 demonstrates how to use a deferred-shape array within a function. Array `b` in function `defshape()` is deferred. The shape of array `b` is derived from the shape of array `a`. The output of Program 16.5 is the same as that of Program 16.3 shown in Figure 16.5.

```
/* File: defshape.ch */
#include <array.h>
#define N 2
#define M 3

double defshape(array double a[:][:], int n, int m) {
    array double b[n][m];   // b is deferred-shape array
    double sum = 0;
    int i, j;

    b = a + 2 * a;  // b = 3*a
    for(i=0; i<n; i++)
      for(j=0; j<m; j++)
        sum += a[i][j];
    printf("b = \n%g", b);
    return sum;
}

double main() {
    double sum;
    array double a1[N][M] = {1, 2, 3,
                             4, 5, 6};

    sum = defshape(a1, N, M);
    printf("sum = %g\n",  sum);
    return 0;
}
```

Program 16.5: Using computational arrays of deferred-shape.

16.7.4 Arrays in Variable Number Arguments

Arrays of different shapes and types can be passed to a function using variable number arguments and macros defined in header file **stdarg**. In section10.7 of Chapter 10, we illustrate how to change the arrays of different types in a calling function illustrated by function **lindata()** in Program 10.30.

Program 16.6 with output in Figure 16.7 illustrates how arrays `a` and b of of different shapes and types are passed to function `func()` through variable number arguments. The contents of these arrays are copied into temporary arrays `a` and `b` inside function `func()` using function **arraycopy()**. The memory fo passed arrays in a calling function can also be used inside the called function can also be used directly using pointer to array. More information about handling of polymorphic functions using variable number arguments and pointer to array can be found in section 19.9.3 in Chapter 19.

16.7.5 Arrays of Reference

It is recommended to pass arrays of different shapes and types using variable number arguments described in section10.7 in Chapter 10 instead of using arrays of reference. Arrays of reference is obsolete and will be phased out in the future.

We have described how assumed-shape arrays can be used to handle arrays of variable length. Arrays of reference are introduced in Ch to deal with arrays of not only different length, but also different data type. It can be used effectively for function overloading. Arrays of reference are declared with ampersand signs, '`&`,' as array subscripts. Furthermore, an array of reference without the subscript can be used to handle arrays of different dimension, different length, and different data type. For arrays of reference `a`, `b` and `c` declared below,

```
int fun(array int a[&], array int b[&][&], array int &c);
```

`a` and `b` are arrays of reference with fixed dimension whereas `c` is an array of reference without constraint of dimension. For arguments with reference type, a function shall be defined or prototyped with arguments first before it is called.

If the argument of a function is defined as an array of reference, not only addresses and boundaries, but also data types of arrays are passed to this function. So, arrays with different data type can be handled by the same function. To use arrays of reference, an array with data type of the largest memory requirement and highest order shall be declared in the function argument list. For example, to handle arrays of double, float, and integral type, an array of reference with double type shall be declared. The values of the passed array will be typically assigned to a temporary computational array of double type. This temporary array will be used for computations inside the function. To pass the result back to the calling function, the temporary array shall be assigned to the array variable declared in the function argument list.

Program 16.7 illustrates how an array of reference can be used to handle arrays of different data type. In Program 16.7, the function `sum3()`, which takes two arguments of arrays of reference, is called to calculate the same matrix expression

$$\mathbf{b} = \mathbf{a} + \mathbf{2} * \mathbf{a}, \tag{16.3}$$

and returns the sum of value for each element of array `a`. To handle arrays of double, float, and integral type, arrays `a` and `b` of reference type are declared as type double in Program 16.7. Arrays `a1` and `a2` in function `main()` have the same dimensions, but the extents and data types are different. They can be passed to the same argument of function `sum3()`. Similarly, arrays `b1` and `b2` of different extents and data types are also passed to the same argument. Array `a` in function `sum3()` is assigned to array `aa` first, so that internally the addition of each element of the passed array is performed in **double** data type. The output of Program 16.7 is the same as that of Program 16.4 shown in Figure 16.6.

```
#include <stdarg.h>
#include <array.h>

void func(int k, ...) {
    int i, m, n, vacount, num;
    ChType_t dtype;
    void *vptr;
    va_list ap;

    va_start(ap, k);
    vacount = va_count(ap);
    printf("va_count(ap) = %d\n", vacount);
    for(i = 0; i<vacount; i++) {
      if(va_arraytype(ap)==CH_CARRAYTYPE ||
         va_arraytype(ap)==CH_CHARRAYTYPE) {
        printf("va_arraydim(ap)= %d\n", va_arraydim(ap));
        num = va_arraynum(ap);
        printf("va_arraynum(ap)= %d\n", num);
        m = va_arrayextent(ap, 0);
        printf("va_arrayextent(ap, 0)= %d\n", m);
        if(va_arraydim(ap) > 1) {
          n = va_arrayextent(ap, 1);
          printf("va_arrayextent(ap, 1)= %d\n", n);
        }
        if(va_datatype(ap) == CH_INTTYPE) {
          int a[num], *p;
          dtype = va_datatype(ap);
          vptr = va_arg(ap, void *);
          printf("array element is int\n");
          p = vptr;
          printf("p[0] = %d\n", p[0]);
          arraycopy(a, CH_INTTYPE, vptr, dtype, num);
          printf("a[0] = %d\n", a[0]);
        }
        else if(va_datatype(ap) == CH_DOUBLETYPE) {
          array double b[m][n];
          dtype = va_datatype(ap);
          vptr = va_arg(ap, void *);
          printf("array element is double\n");
          arraycopy(&b[0][0], CH_DOUBLETYPE, vptr, dtype, num);
          printf("b = \n%f", b);
        }
      }
      else  if(va_datatype(ap) == CH_INTPTRTYPE)
        printf("data type is pointer to int\n");
    }
    va_end(ap);
}

int main() {
   int i, a[4]={10, 20, 30}, *p;
   array double b[2][3]={1, 2, 3, 4, 5, 6};

   p = &i;
   func(i, a);
   func(i, b,a);
   func(i, p);
}
```

Program 16.6: Pass arrays of different shapes and types to a function.

```
va_count(ap) = 1
va_arraydim(ap)= 1
va_arraynum(ap)= 4
va_arrayextent(ap, 0)= 4
array element is int
p[0] = 10
a[0] = 10
va_count(ap) = 2
va_arraydim(ap)= 2
va_arraynum(ap)= 6
va_arrayextent(ap, 0)= 2
va_arrayextent(ap, 1)= 3
array element is double
b =
1.000000 2.000000 3.000000
4.000000 5.000000 6.000000
va_arraydim(ap)= 1
va_arraynum(ap)= 4
va_arrayextent(ap, 0)= 4
array element is int
p[0] = 10
a[0] = 10
va_count(ap) = 1
data type is pointer to int
```

Figure 16.7: Output of Program 16.6.

Program 16.8 illustrates how to handle arrays of different dimensions and data types using arrays of reference. The function `sum4()` in Program 16.8 takes two arguments of arrays of reference and one argument of **int** type for the number of elements of array `a`.

Arrays `a1` and `a2` of different dimension and type are passed to the same argument. Similarly, array `b1` and `b2` of different dimension and type are used to pass back the result of a matrix expression calculated inside function `sum4()`. The output of Program 16.8 is displayed in Figure 16.8.

Elements of array of reference without subscripts can not be accessed directly with subscripts. For example, elements of array of reference of `a` and `b` in function `sum4()` of Program 16.8 can not be followed by subscripts such as `a[2]` or `a[1][2]`.

In Program 16.8, the third argument of function `sum4()` contains the number of elements of the array passed to the array `a` of the function. The number of dimensions, extends of each dimension, and total number of elements of of the array passed can be obtained by expression `n = (int)shape(shape(a)), dim = shape(a), totnum *= dim[i]`, respectively, inside the function as shown in Program 16.9. The output of Program 16.9 is shown in in Figure 16.9. Should the printing statements for dimensions and total number of elements in Program 16.9 be commented out, the ouput of Program 16.9 shall be the same as that of Program 16.8.

The generic function **elementtype()** can be used to obtain the data type of its argument. The argument of the function **elementtype()** can be a type declarator, C array, computational array, or an array of reference. For example, given

```
array double a[3][4];
int b[3][4];
```

the following two equations hold.

```
elementtype(double) == elementtype(a);
```

```
#include <array.h>

double sum3(array double a[&][&], array double b[&][&]){
    int n = shape(a, 0), m = shape(a, 1);
    double sum = 0;
    int i, j;
    array double aa[n][m];

    printf("n = %d, m = %d\n", n, m);
    b = a + 2 * a;  // b = 3*a
    aa = a;
    for(i=0; i<n; i++)
      for(j=0; j<m; j++)
        sum += aa[i][j];
    return sum;
}

int main() {
    double sum;
    array double b1[2][3], a1[2][3] = {1, 2, 3,
                                       4, 5, 6};
    array float b2[3][4], a2[3][4] = {1, 2, 3, 4,
                                      5, 6, 7, 8,
                                      9, 10, 11, 12};

    sum = sum3(a1, b1);
    printf("b1 = \n%g", b1);
    printf("sum = %g\n\n",  sum);
    sum = sum3(a2, b2);
    printf("b2 = \n%g", b2);
    printf("sum = %g\n",  sum);
    return 0;
}
```

Program 16.7: Passing computational arrays of different shapes and data types.

```
/* File: sum4.ch */
#include <array.h>
#define N 2
#define M 3

double sum4(array double &a, array double &b, int total_num){
    int i;
    double sum;
    array double aa[total_num];

    b = a + 2 * a;  // b = 3*a
    aa = a;
    for(i=0; i<total_num; i++)
      sum += aa[i];

    return sum;
}

int main() {
    double sum;
    array double b1[N][M], a1[N][M] = {1, 2, 3,
                                       4, 5, 6};
    array float b2[M], a2[M] = {10, 20, 30};


    sum = sum4(a1, b1, N*M);
    printf("b1 = \n%g", b1);
    printf("sum = %g\n\n",  sum);
    sum = sum4(a2, b2, M);
    printf("b2 = \n%g", b2);
    printf("sum = %g\n",  sum);
    return 0;
}
```

Program 16.8: Passing computational arrays of different ranks and data types.

```
b1 =
3 6 9
12 15 18
sum = 21

b2 =
30 60 90
sum = 60
```

Figure 16.8: Output of Program 16.8.

```
/* File: sum5.ch */
#include <array.h>
#define N 2
#define M 3

double sum5(array double &a, array double &b){
    int n, i, total_num;
    double sum;

    b = a + 2 * a;  // b = 3*a
    total_num = 1;
    n = (int)shape(shape(a));  // number of dimensions
    array int dim[n];

    dim = shape(a);            // extent of each dimension
    printf("n = %d\n", n);
    for(i = 0; i < n; i++) {
       printf("dim[%d] = %d\n", i, dim[i]);
       total_num *= dim[i];    // total number of elements
    }
    printf("total_num = %d\n", total_num);
    array double aa[total_num];
    aa = a;
    for(i=0; i<total_num; i++)
      sum += aa[i];

    return sum;
}

int main() {
    double sum;
    array double b1[N][M], a1[N][M] = {1, 2, 3,
                                       4, 5, 6};
    array float b2[3], a2[3] = {10, 20, 30};


    sum = sum5(a1, b1);
    printf("b1 = \n%g", b1);
    printf("sum = %g\n\n",  sum);
    sum = sum5(a2, b2);
    printf("b2 = \n%g", b2);
    printf("sum = %g\n",  sum);
    return 0;
}
```

Program 16.9: Passing computational arrays of different ranks and data types and using function **shape()**.

```
n = 2
dim[0] = 2
dim[1] = 3
total_num = 6
b1 =
3 6 9
12 15 18
sum = 21

n = 1
dim[0] = 3
total_num = 3
b2 =
30 60 90
sum = 60
```

Figure 16.9: Output of Program 16.9.

```
elementtype(int) == elementtype(b);
```

In most cases, mathematical algorithms for arrays of complex and arrays of real number are different. In Program 16.10, function `arrayfunc()` can handle both arrays of complex and arrays of real number using function **elementtype()**. Depending on the data type of array argument `a`, the real function `realfunc()` or complex function `complexfunc()` will be called inside function `arrayfunc()` to calculate the array expression $\mathbf{a} + 2\sin(\mathbf{a})$. The output of Program 16.10 is displayed in Figure 16.10.

If the pointer NULL is passed into a function as an argument of array of reference, the argument is also equal to NULL inside the function, and the function **shape()** returns an array of zero dimension. For example, if functions `func1()` and `func2()` are defined below.

```
int func1(array double a[&]) {
  if(((int)shape(a)) == 0) {
    printf("shape is zero dimension\n");
  }
  if(a == NULL) {
     printf("a is NULL \n");
  }
  return 0;
}

int func2(array double &a) {
  if((int)shape(shape(a)) == 0) {
    printf("shape is zero dimension\n");
  }
  if(a == NULL) {
     printf("a is NULL \n");
  }
  return 0;
}
```

both function calls of `func1(NULL)` and `func2(NULL)` will print out

```
shape is zero dimension
```

```
#include <array.h>

void complexfunc(array double complex a[:][:], array double complex b[:][:]){
    b = a + 2 * sin(a);
}
void realfunc(array double a[:][:], array double b[:][:]){
    b = a + 2 * sin(a);
}
void arrayfunc(array double complex a[&][&], array double complex b[&][&]){
  int n = shape(a, 0), m = shape(a, 1);
  // or array int dim[2] = shape(a);
  // int n = dim[0], m = dim[1];

  if(elementtype(a) == elementtype(complex) ||
     elementtype(a) == elementtype(double complex)) {
    array double complex aa[n][m], bb[n][m];
    aa = (array double complex [n][m])a;
    complexfunc(aa, bb);
    b = bb;
  }
  else {
    array double aa[n][m], bb[n][m];
    aa = (array double [n][m])a;
    realfunc(aa, bb);
    b = bb;
  }
}

int main() {
  array double complex b1[3][4], a1[3][4] = {1, complex(1,2), 2, 5,
                                             7, complex(3,4), 9, 3,
                                             5, 7, 3, 2};
  array double  b2[2][3], a2[2][3] = {1, 5, 3,
                                      5, 6, 7};
  arrayfunc(a1, b1);
  printf("b1 = \n%.1f", b1);
  arrayfunc(a2, b2);
  printf("\nb2 = \n%.1f", b2);
  return 0;
}
```

Program 16.10: Passing arrays of different data type to a function.

```
b1 =
complex(2.7,0.0) complex(7.3,5.9) complex(3.8,0.0) complex(3.1,0.0)
complex(8.3,0.0) complex(10.7,-50.0) complex(9.8,0.0) complex(3.3,0.0)

complex(3.1,0.0) complex(8.3,0.0) complex(3.3,0.0) complex(3.8,0.0)

b2 =
2.7 3.1 3.3
3.1 5.4 8.3
```

Figure 16.10: Output of Program 16.10.

```
/* File: arrayrefnull.ch */
#include <array.h>

void func(array double &a) {
    if(a==NULL) {
        printf("a==NULL is true\n");
    }
    else {
        printf("a==NULL is false\n");
    }
    if(a!=NULL) {
        printf("a!=NULL is true \n");
    }
    else {
       printf("a!=NULL is false\n");
    }
}

int main() {
   func(NULL);
   return 0;
}
```

Program 16.11: Passing NULL to computational array of reference.

```
a is NULL
```

16.8 Computational Arrays with Value NULL

In most cases, a computational array has a rank of 1 or higher. In some situations, a computational array can have value of NULL. Before it is allocated memory, a pointer to computational array has a value of NULL. A value of NULL can also be passed to an argument of array of reference type in a function. A computational array with value of NULL can be used as an operand of equal operator '==' or not equal operator '!=' as well as a controlling expression of **if**-statement and loops. They cannot be used as an operand for other operations.

If one of two operands for equal operator '==' or not equal operator '!=' is pointer to computational array or array of reference, the other operand can be NULL. The result of the operation in this case is a boolean type of either **true** or **false**. This can be used to test if NULL has been passed to array of reference or if a pointer to computational array points to a valid object.

A computational array can be used as a controlling expression for **if**-statement, **while**-loop, **do-while**-loop, or **for**-statement. When an array of reference or a pointer to computational array with a value of NULL is used as a controlling expression, it evaluates to **false**. Otherwise, the controlling expression evaluates to true, even if all elements of the array are zero.

In Programs 16.11, NULL is passed to the argument `a` of array of reference in function `func()`. In Program 16.12, variable `a` of pointer to computational array has a default value of NULL before it is pointed to an array. The output of these two programs are the same as shown below.

```
a==NULL is true
a!=NULL is false
```

```
/* File: arrayptrnull.ch */
#include <array.h>

int main() {
    array double *a;

    if(a==NULL) {
        printf("a==NULL is true\n");
    }
    else {
        printf("a==NULL is false\n");
    }
    if(a!=NULL) {
        printf("a!=NULL is true \n");
    }
    else {
       printf("a!=NULL is false\n");
    }
    return 0;
}
```

Program 16.12: Pointer to computational array with value NULL.

16.9 Functions Return Computational Arrays

A function can return computational arrays as first-class objects. For a function that returns a computational array, the rank of the returned array in the function definition and that of an array expression following a return statement inside the function must be the same.

16.9.1 Functions Return Computational Arrays of Fixed Length

The prototype of functions returning computational arrays of fixed length is as follows.

```
array datatype funcname(argument_list) [n1]...[nm];
```

where `n1` and `nm` are constant integers, such as 2 and 3, for the lengths of the corresponding dimensions. The number of symbol `[]` following the closing parenthesis of the function argument list indicates the rank of the returned computational array.

Program 16.13 is an example to demonstrate how a function returns a computational array to the calling function. Function `funct()` of this program returns the result of matrix expression of dimension 2x3.

$$\mathbf{b} = \mathbf{2} * \mathbf{a}, \tag{16.4}$$

which is shown in Figure 16.11.

16.9.2 Functions Return Computational Arrays of Variable Length

The prototype of functions returning computational arrays of variable length is as follows.

```
array datatype funcname(argument_list) [:]...[:]
```

The number of symbol `[:]` following the closing parenthesis of the function argument list indicates the rank of the returned computational array.

```
/* File: retfix.ch */
#include <array.h>

int main() {
    array int a[2][3] = {1, 2, 3, 4, 5, 6};
    array int funct(array int a[2][3])[2][3];

    a = funct(a);
    printf("a[1][2] = %d\n", a[1][2]);
    printf("a = \n%d", a);
    return 0;
}

array int funct(array int a[2][3])[2][3] {
    array int b[2][3];

    b = 2*a;
    return b;
}
```

Program 16.13: Function returning computational array of fixed length.

```
a[1][2] = 12
a =
2 4 6
8 10 12
```

Figure 16.11: Output of Program 16.13.

```
/* File: retvla.ch */
#include<array.h>

array int func2(array int a[:])[:] {
    int n = (int)shape(a);
    array int x[n];

    printf("n = %d\n", n);
    x = 2*a;
    return x;
}

int main() {
    array int a[2] = {1, 2};
    array int b[5] = {10, 20, 30, 40, 50};

    a = func2(a);
    printf("a = %d\n", a);
    b = func2(b);
    printf("b = %d", b);
    return 0;
}
```

Program 16.14: Function returning computational array of variable length.

```
n = 2
a = 2 4

n = 5
b = 20 40 60 80 100
```

Figure 16.12: Output of Program 16.14.

Program 16.14 provides an example of a function that returns a computational array of variable length. The dimensions of the returned arrays in function calls of `func2(a)` and `func2(b)` are different. The output is shown in Figure 16.12.

16.10 Type Generic Array Functions

Function **shape()** presented in section 16.7.2 is a generic function related to arrays. In addition, commonly used generic mathematical functions are overloaded to handle computational arrays. They are overloaded to handle arguments of different dimensions, lengths, and data types.

For an argument of computational array type, function **abs**() returns an array with the absolute value for each element. For an argument of complex type, each element contains the magnitude of the corresponding complex number. The function is handled as if it was prototyped as

```
array int abs(array int a[:]...[:])[:]...[:]
array float abs(array float a[:]...[:])[:]...[:]
array float abs(array float complex a[:]...[:])[:]...[:]
array double abs(array double a[:]...[:])[:]...[:]
```

```
array double abs(array double complex a[:]...[:])[:]...[:]
```

For example,

```
> array int a[2][3] = {-1, 2, 3, -4, -5, 6}
> abs(a)
1 2 3
4 5 6
> array complex b[3] = {complex(3, 4), 4, -5}
> abs(b)
5.0000 4.0000 5.0000
```

Mathematical functions **acos, acosh, asin, asinh, atan, atanh, ceil, cos, cosh, exp, floor, log, log10, sin, sinh, sqrt, tan, tanh** have one argument only. They are overloaded to handle arguments of different dimensions, lengths, and data types. If the data type of the input argument is an integral type, it will be promoted to float for computation. For array arguments, they behave as if they were prototyped as

```
array float func(array float a[:]...[:])[:]...[:]
array double func(array double a[:]...[:])[:]...[:]
array float complex func(array float complex a[:]...[:])[:]...[:]
array double complex func(array double complex a[:]...[:])[:]...[:]
```

where `func` is one of the above mathematical functions. If the data type of the input argument is integral type, it will be promoted to float for computation. For example,

```
> array int a[2][3] = {-1, 2, 3, -4, -5, 6}
> sin(a)
-0.84 0.91 0.14
0.76 0.96 -0.28
> array complex b[3] = {complex(3, 4), 4, -5}
> sin(b)
complex(3.8537,-27.0168) complex(-0.7568,-0.0000) complex(0.9589,0.0000)
```

For array arguments, function **atan2()** acts as if it was prototyped as

```
array float          atan2(array float y[:]...[:],
                           array float x[:]...[:])[:]...[:]
array double         atan2(array double y[:]...[:],
                           array double x[:]...[:])[:]...[:]
array float complex  atan2(array float complex y[:]...[:],
                           array float complex x[:]...[:])[:]...[:]
array double complex atan2(array double complex y[:]...[:],
                           array double complex x[:]...[:])[:]...[:]
```

Function **atan2()** has two arguments. The data type of both computational arrays shall be the same. If data types of both arguments are integral type, they will be promoted to float for computation. For example,

```
> array int   y[4]={1,-2, 3,  -4}
> array float x[4]={5, 6, -7, -8}
> atan2(y, x)
0.20 -0.32 2.74 -2.68
```

If the first argument of function **pow(a, x)** is a computational array with shape of NxN and the second is an integral type, it will return a computational array of the same type and dimension as the first argument as if the function was prototyped as

```
array int pow(array int a[:][:], int x)[:][:]
array float pow(array float a[:][:], int x)[:][:]
array double pow(array double a[:][:], int x)[:][:]
array float complex pow(array float complex a[:][:], int x)[:][:]
array double complex pow(array double complex a[:][:], int x)[:][:]
```

In this case, array function **pow(a, x)** behaves like matrix multiplication. For example,

```
> array int a[2][2] = {-1, 2, 3, -4}
> pow(a, 2)
7 -10
-15 22
> a*a
7 -10
-15 22
```

If both arguments of array function **pow()** are computational array type, it will return an array with the value of each element calculated by scalar function **pow()** with corresponding elements of the two input arrays. In this case, data types of two input arrays shall be the same as if the function was prototyped as

```
array int            pow(array int y[:]...[:],
                         array int x[:]...[:])[:]...[:]
array float          pow(array float y[:]...[:],
                         array float x[:]...[:])[:]...[:]
array double         pow(array double y[:]...[:],
                         array double x[:]...[:])[:]...[:]
array float complex  pow(array float complex y[:]...[:],
                         array float complex x[:]...[:])[:]...[:]
array double complex pow(array double complex y[:]...[:],
                         array double complex x[:]...[:])[:]...[:]
```

For example,

```
> array int a[3] = {-1, 2, 3}
> pow(a, a)
-1 4 27
```

Functions **real()** and and **imag()** will give the real and imaginary parts of the input argument. For array arguments, They behave as if they were prototyped as

```
array float func(array float a[:]...[:])[:]...[:]
array double func(array double a[:]...[:])[:]...[:]
array float func(array float complex a[:]...[:])[:]...[:]
array double func(array double complex a[:]...[:])[:]...[:]
```

where `func` is either **real** or **imag**. If the data type of the input argument is integral type, it will be promoted to float for computation. For example,

```
> array int a[3] = {-1, 2, 3}
> real(a)
-1.00 2.00 3.00
> array complex z[3] = {complex(1,2), complex(-3, -4), complex(0, -6)}
> real(z)
1.0000 -3.0000 0.0000
> imag(z)
2.0000 -4.0000 -6.0000
```

Array function **transpose()** returns a transpose of the input array of one or two dimensions. If the input array is of size NxM, the size of the returned array is MxN. By default, a one-dimensional array is a column vector. If the input array is a column vector of size Nx1, the return array is a row vector of 1xN, and vice versa. The data type of the returned array is the same as the input array as if the function was prototyped as

```
array type transpose(array data_type a[:])[:]
array type transpose(array data_type a[:][:])[:][:]
```

where `data_type` can be any valid type for computational array. For example,

```
> array float a[2][3]={1,2,3,4,5,6}
1.00 2.00 3.00
4.00 5.00 6.00
> transpose(a)
1.00 4.00
2.00 5.00
3.00 6.00
> array int b[3] = {1, 2, 3}
> a*b
14.00 32.00
> transpose(b)*b
14
> b*transpose(b)
1 2 3
2 4 6
3 6 9
```

16.11 Some Commonly Used Array Functions

Many advanced numerical functions are available in Ch. These functions are prototyped in header file **numeric.h**. Some commonly used numerical functions are presented in this section.

Function **lindata()** introduced in section 10.7 is prototyped in header file **numeric.h** as follows.

```
int lindata(double first, double last, ... /* type a[:]...[:] */);
```

The **lindata()** function generates linearly spaced data with initial and final values specified by input arguments `first` and `last`, respectively. The result is passed back to the calling function in the third argument of array type with different data types.

Given a square matrix $\mathbf{A}$ and its inverse $\mathbf{A}^{-1}$, then $\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$ and $\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$ where $\mathbf{I}$ is an identity matrix. Function **inverse()** calculates the inverse of a square matrix, provided that it is not singular. Function **inverse()** is prototyped in header file **numeric.h** as follows.

```
array double inverse(array double x[&][&], ... /* [int *status */)[:][:];
```

The dimension of the returned matrix of function **inverse()** is the same as the input argument of matrix. This function can be used to solve a linear system of equations. For example, a linear system of equations below,

$$\begin{aligned} 3x_1 + 6x_3 &= 2 \\ 2x_2 + x_3 &= 12 \\ x_1 + x_3 &= 25 \end{aligned}$$

or in the matrix form

$$\begin{bmatrix} 3 & 0 & 6 \\ 0 & 2 & 1 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 12 \\ 25 \end{bmatrix}$$

can be written in the form of

$$\mathbf{Ax} = \mathbf{b}.$$

The solution $\mathbf{x} = \mathbf{A^{-1}b}$ can be written in Ch as `x = inverse(A)*b`. The solutions for x_1, x_2, and x_3 can be determined by the following statements.

```
> array double a[3][3]={3, 0, 6, 0, 2, 1, 1, 0, 1}
> array double ai[3][3], x[3], b[3]= {2, 13, 25}
> ai = inverse(a)
-0.3333 -0.0000 2.0000
-0.1667 0.5000 0.5000
0.3333 0.0000 -1.0000
> x = ai*b
49.3333 18.6667 -24.3333
```

As another example, consider the following two matrix equations

$$(\mathbf{A} + \mathbf{5B^{-1}})\mathbf{x} + \mathbf{2a} = (\mathbf{ab^T})\mathbf{b}, \tag{16.5}$$

$$(\mathbf{5AB})\mathbf{x} + \mathbf{ABy} = \mathbf{Bb}, \tag{16.6}$$

where

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 2 \\ 4 & 4 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \mathbf{B} = \begin{bmatrix} 7 & 8 & 9 \\ 1 & 2 & 2 \\ 4 & 4 & 6 \end{bmatrix}, \mathbf{a} = \begin{bmatrix} 1 \\ 4 \\ 7 \end{bmatrix}, \text{and } \mathbf{b} = \begin{bmatrix} 5 \\ 6 \\ 8 \end{bmatrix},$$

The two unknown vectors $\mathbf{x}$ and $\mathbf{y}$ can be computed by the following equations

$$\mathbf{x} = (\mathbf{A} + \mathbf{5B^{-1}})^{-1}(\mathbf{ab^T b} - \mathbf{2a}), \tag{16.7}$$

$$\mathbf{y} = (\mathbf{AB})^{-1}(\mathbf{Bb} - \mathbf{5ABx}), \tag{16.8}$$

The vectors $\mathbf{x}$ and $\mathbf{y}$ can be calculated using Program 16.15. The output of Program 16.15 is as follows.

```
x = 51.048 15.170 37.020
y = -544.617 -70.723 13.777
```

```
/* File: matrixeq.ch */
#include <stdio.h>
#include <array.h>    // for array qualifier
#include <numeric.h>  // for inverse()

int main() {
    array double A[3][3] = {{1,2,2},{4,4,6},{7,8,9}};
    array double B[3][3] = {{7,8,9},{1,2,2},{4,4,6}};
    array double a[3] = {1,4,7}, b[3] = {5,6,8}, x[3], y[3];

    x = inverse(A+5*inverse(B))*(a*transpose(b)*b - 2*a);
    y = inverse(A*B)*(B*b - 5*A*B*x);
    printf("  x = %.3f  y = %.3f \n", x, y);
    return 0;
}
```

Program 16.15: A program solving a system of linear equations.

Function **sum()** defined in header file **numeric.h** calculates the sum of all the elements in an array. Its prototype is as follow.

```
double sum(array double &a, ... /* [array double v[:]] */);
```

If the array is a two-dimensional matrix, the function can calculate the sum of each row with the result stored in the optional second argument as a one-dimensional array. For example,

```
> double a[3] = {10, 2, 3}
> sum(a)
15.0000
> array double v[3], b[3][2] = {1, 2, 3, 4, 5, 6}
> sum(b, v)
21.0000
> v
3.0000 7.0000 11.0000
```

Some useful array functions can be implemented using function **sum()**. For example, the array functions `all()`, `any()`, and `count()` in Program 16.16 are implemented using function **sum()** with array promotions. For function `all()`, if all elements of the array argument are zero, it returns 1. Otherwise, it returns 0. If all elements of the array argument `a` in function `all()` are zero, the resultant array of array expression `a!=0` will be of values of zeros for all its elements. The summary of all elements of this array from the function `sum()` will be zero. If there is any element of zero value in the array argument of of function `any()`, it returns 1. Otherwise, it returns 0. Function `count()` calculates the number of zero in an array. The output of Program 16.16 is shown in Figure 16.13.

16.12 Pointer to Computational Arrays

16.12.1 Pointer to Computational Arrays of Fixed Length

In some applications, using pointers to computational arrays is more convenient than using multi-dimensional arrays. The same variable of pointer to computational arrays can be used to access different computational arrays. Pointer to computational arrays can be declared similar to pointer to C arrays as shown below.

```
/* promotion.ch */
#include<array.h>
#include<numeric.h>

array int a[2][3] = {1, 2, 3, 4, 5, 6};
array double b[2][3] = {1, 2, 0, 4, 5, 0};
array int c[2][3];

int all(array double &a) {
    return (int)sum(a != 0) == 0;
}

int any(array double &a) {
    return (int)sum(a == 0) > 0;
}

int count(array double &a) {
    return (int)sum(a == 0);
}
int main () {
    printf("%freturn value of all() is %d\n\n", b, all(b));
    printf("%dreturn value of all() is %d\n\n", c, all(c));
    printf("%dreturn value of any() is %d\n\n", a, any(a));
    printf("%freturn value of any() is %d\n\n", b, any(b));
    printf("%dreturn value of count() is %d\n\n", a, count(a));
    printf("%freturn value of count() is %d\n", b, count(b));
    return 0;
}
```

Program 16.16: Functions with array promotions.

```
1.000000 2.000000 0.000000
4.000000 5.000000 0.000000
return value of all() is 0

0 0 0
0 0 0
return value of all() is 1

1 2 3
4 5 6
return value of any() is 0

1.000000 2.000000 0.000000
4.000000 5.000000 0.000000
return value of any() is 1

1 2 3
4 5 6
return value of count() is 0

1.000000 2.000000 0.000000
4.000000 5.000000 0.000000
return value of count() is 2
```

Figure 16.13: Output of Program 16.16.

```
array double (*p)[10];
```

which declares p as a pointer to two-dimensional computational arrays with 10 columns of double data type. The following code fragment shows how to use pointer to computational arrays to handle multiple dimension computational arrays.

```
> array int (*p)[3], b1[2][3] ={1, 2, 3, 4, 5, 6}
> int b2[3][3] = {7, 8, 9, 1, 2, 3, 4, 5, 6}, *t
> p == NULL
1
> p = (array int [:][:])(array int [2][3])malloc(2*3*sizeof(int))
0 0 0
0 0 0
> p == NULL
0
> p[1][1] = 40
40
> p
0 0 0
0 40 0
> p = b1                    // array assignment
1 2 3
4 5 6
> delete p                   // free memory
> p
NULL

> p = (array int [:][:])b1 // p and b1 share the same memory
1 2 3
4 5 6
> b1[0][1] = 30;
30
> b1
1 30 3
4 5 6
> p
1 30 3
4 5 6

> p = (array int [:][:])b2 // p and b2 share the same memory
7 8 9
1 2 3
4 5 6

> t = &b1[0][0]
> p = (array int [:][:])(array int [2][3])t
1 2 3
4 5 6
```

Before a pointer to computational arrays is used, it has to be allocated memory, or error messages will be displayed. One it is allocated memory, it will be treated as a regular computational array. It can be used as an operand in array operations and as an argument of functions. Using casting operator `(array` *`data_type`* `[:]...[:])` or `(array` *`data_type`* `(*)[:]...[:])` the following two methods can be used to allocate memory for a pointer to computational arrays, where *data_type* is any valid type of computational array.

1. Casting operator `(array data_type [:]...[:])(array data_type [n1]...[ni])` followed by a pointer to memory. Or casting operator `(array data_type [:]...[:])new data_type [n1]...[ni];`
2. Casting operator `(array data_type [:]...[:])` followed by the name of a C array.

For the first method, if the memory is allocated by function **malloc()**, **calloc()**, or **realloc()**, the memory allocated for the pointer to computational array can be released by the function **free()** or **delete** later. If the memory is allocated by operator **new**, it shall be released by operator **delete**. In the previous example, the statements

```
array int (*p)[3], b1[2][3] ={1, 2, 3, 4, 5, 6}
p = (array int [:][:])(array int [2][3])malloc(2*3*sizeof(int))
```

allocate memory for `p` using function **malloc()**. The memory can also be allocated by operator **new** and deleted by **delete** as follows.

```
p = (array int [:][:])new int [2][3];
...
delete p;
```

Details about operators **new** and **delete** are described in chapter 19. Statement

```
t = &b1[0][0]
p = (array int [:][:])(array int [2][3])t
```

or

```
p = (array int [:][:])(array int [2][3])&b1[0][0]
```

or

```
p = (array int [:][:])(int [2][3])&b1[0][0]
```

enables computational array `p` to share the memory of array `b1`.

For the second method, both pointer to computational array and original array share the same memory. In the previous example, the statement

```
p = (array int [:][:])b1
```

makes `p` point to computational array `b1`. Later, the statement

```
p = (array int [:][:])b2
```

will point `p` at the C array `b2`. Note that the extents of the second dimension for arrays `p`, `b1` and `b2` shall be the same.

If `p` has been allocated memory, the assignment operation is allowed. For example, the statement

```
p = (array int [2][3])b1
```

assigns the value of each element of array `b1` to the corresponding element of array `b`. The memory for array `p` and `b1` can be of different.

Pointer to one-dimension computational array is declared without array index as shown in the commands below.

```
> array int *p, a[3] = {1, 2, 3}
> p = (array int [:])a   // p and a share the memory
1 2 3
> p = (array int [:])(array int [4])malloc(4*sizeof(int))
0 0 0 0
>p = (array int [4])10
10 10 10 10
> delete (p)
```

The interactive execution of the code below illustrates how to using a pointer to one-dimensional computational array to access rows of two-dimensional arrays.

```
> array int *p, b[2][3] = {1, 2, 3, 4, 5, 6}
> p = (array int [:])(int[3])&b[0][0]
1 2 3
> &p                  // same as &b and &b[0][0]
1e8650
> p = (array int [:])(int[3])(int*)b
1 2 3
> &p                  // same as &a and &a[0][0][0]
1e8650
> p = 10
10 10 10
> b
10 10 10
4 5 6
> p = (array int [:])(int[3])&b[1][0]
4 5 6
> &p                  // same as &b[1][0]
1e865c
```

As mentioned in section 16.5.8, the casting operator with a pointer type gives the address of the first element of an array. Thus, the statement below

```
p = (array int [:])(int[3])(int*)b
```

is equivalent to

```
p = (array int [:])(int[3])&b[0][0]
```

The command above may be used to have `p` refer to the first row of a two-dimensional array `b`, whereas command

```
p = (array int [:])(int [3])&b[1][0]
```

will have p point to the second row of the array b.

Note that pointers to computational arrays might be used incorrectly as illustrated below. Given the following declarations,

```
array short h[2][3];
array int (*p)[3], a[2][3], b[3][2], c[6];
array float f[2][3];
array double d[2][3];
int e[2][3], g[3][2];
int *ptr;
```

the statement

```
p = a;
```

is incorrect because p has not been allocated memory yet. A pointer to computational array should be allocated memory before it is used. The statement

```
p = (array int [:][:])(a+a);  // bad
```

is also incorrect because p will point to some intermediate memory instead of a. Similarly, for the statement below.

```
p = (array int [:][:])(array int [2][3])b;  // bad
```

p will point to some intermediate memory instead of the array b. The statement below will point p to the memory of array b.

```
p = (array int [:][:])(array int [2][3])&b[0][0];  // ok
```

A pointer to computational array can also point to regular C arrays as shown below.

```
p = (array int [:][:])e;  // ok
p = (array int [:][:])(array int [2][3])g;  // ok
```

The statement

```
p = (array int [:][3])a;  // bad
```

is incorrect. It should use the casting operator `(array int [:][:])` or `(array int (*)[:])` to make p share the memory with a. The statement

```
p = (int (*)[3])a;        // bad
```

is missing the keyword **array**. Pointers of scalar types and pointers to computational arrays are incompatible. The assignment operations below with incompatible lvalue and rvalue are not allowed.

```
p = ptr;                  // bad
ptr = p;                  // bad
p = (void *)malloc(100);  // bad
```

In the statement

```
p = (array int [:][:]) c; // bad
```

the dimensions of p and c do not match. So, it will get an error message. In the statement

```
p = (array int [:][:])h;  // bad
```

computational array h of short type does not have enough memory to share with p because p is of int type. The statement

```
p = (array int [:][:])f;
```

is correct from the memory space point of view, because f is of float type and has enough memory to share with p.

After the pointer is allocated memory, or shares the memory with an array, the address operator gives the address of the memory, or the first element of the array. In the example below, the commands

```
> p = (array int [:][:])a
> &p
```

give the address of the first element of a.

A pointer to computational array can be used to obtain a subarray or 'slice' of a multi-dimensional array as shown below.

```
> array int a[2][2][2] = {1, 2, 3, 4, 5, 6, 7, 8}
> array int (*p)[2]
4005e3e0
> p = (array int [:][:])(int[2][2])&a[0][0][0]
1 2
3 4
> &p                  // same as &a and &a[0][0][0]
4005e4e0
> p = (array int [:][:])(int[2][2])(int*)a
1 2
3 4
> &p                  // same as &a and &a[0][0][0]
4005e4e0
> p = (array int [:][:])(int[2][2])&a[1][0][0]
5 6
7 8
> &p                  // same as &a[1][0][0]
4005e4f0
```

Similar to the one-dimensional array, the casting operator with a pointer type gives the address of the first element of an array. Thus, the statement below

```
p = (array int [:][:])(int[2][2])(int*)a
```

is equivalent to

```
p = (array int [:][:])(int[2][2])&a[0][0][0]
```

The command above may be used to have p refer to one portion of the three dimensional array a, whereas command

```
p = (array int [:][:])(int [2][2])&a[1][0][0]
```

will have p point to the other portion.

16.12.2 Pointer to Computational Arrays of Assumed Shape

Besides pointer to computational arrays, Ch also supports pointer to computational arrays of assumed shape. Unlike pointer to computational arrays, pointer to computational array of assumed shape can be used to point to arrays of variable length. So, users do not need to worry about the extents of the arrays to be pointed. Pointers to computational array of assumed shape are declared with colon ':' as the array's subscripts. Before a pointer to computational arrays of assumed-shape is used, it also has to be allocated memory in the same manner as a pointer to computational arrays of fixed length described in the previous section.

For example, statement below

```
array float (*fp)[:];
```

declares `fp` as a pointer to computational arrays of assumed shape with float type. It can be pointed to two-dimensional arrays of variable length.

The following commands show how to use pointer to computational arrays of assumed shape to handle multiple dimension arrays of different length.

```
> array int (*p)[:]
> array int b1[2][3] ={1, 2, 3, 4, 5, 6}
> array int b2[2][2] ={5, 6, 7, 8}
> p = (array int [:][:])b1
1 2 3
4 5 6
> p = (array int [:][:])b2
5 6
7 8
```

In the above commands, arrays `b1` and `b2` are of the same dimension. But the extents of the second dimension for `b1` and `b2` are different. Unlike pointer to computational array of fixed length, `p`, a pointer to computational array of assumed-shape, can be used to point at either `b1` or `b2`.

Furthermore, a pointer to computational array can also be used to represent a subspace (or subarray) of a multiple dimension array. For example,

```
> array int a[2][2][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
> array int b[3][2][2] = {12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1}
> array int (*p)[:]
> a
1 2 3
4 5 6

7 8 9
10 11 12
> p = (array int [:][:])(int [2][3])&a[0][0][0]
1 2 3
4 5 6
> p = (array int [:][:])(int [2][3])&a[1][0][0]
7 8 9
10 11 12
> p = (array int [:][:])(int [2][2])&b[0][0][0]
12 11
10 9
```

where `p` is used to represent the first and second "slices" of the array `a`, as well the first "slice" of the array `b` which has different dimensions. The address operator '`&`' gets the addresses of these array elements.

Not only ordinary identifiers, but also members of classes, structures, and unions, can be declared as pointer to computational array of assumed-shape. This means a member of class/struct/union can be a computational array of variable length. In the interactive command execution below, member `s.a` first shares the same memory with array `a1`, and then shares the memory with array `a2`.

```
> struct tag{ array int (*a)[:];} s
> array int a1[2][3] = {1, 2, 3, 4, 5, 6}, a2[3][4]
> s.a = (array int [:][:])a1;  // s.a and a1 share the memory
> s.a
1 2 3
4 5 6
> s.a = (array int [:][:])a2;  // s.a and a2 share the memory
s.a[1][1] = 10
> a2[1][1]
10
> a1[1][1]
5
```

16.12.3 Using Pointer to Computational Arrays to Pass Arrays to Functions

Pointer to one-dimension computational array can be used to handle arrays of variable length. Like assumed-shape arrays, pointers to computational arrays can also be used as arguments of functions. For example,

```
> void func1(array int *p) {printf("%d", p);}
> int array a[2] = {1,2}
> int array b[3] = {3,4,5}
> func1(a)
1 2
> func1(b)
3 4 5
```

The function `func1()` takes an argument of pointer to computational array. It can handle arrays, such as `a` and `b`, with different length. The function definition in the above example is equivalent to the function definition below which takes an argument of assumed-shape array.

```
void func1(array int p[:]) { printf("%d", p); }
```

Like multi-dimensional assumed-shape arrays, pointers to arrays of assumed shape can also be used as arguments of functions. For example,

```
> void func2(array int (*p2)[:]) { printf("%d", p2); }
> int array a2[2][2] = {1,2,3,4}
> int array b2[3][3] = {1,2,3,4,5,6,7,8,9}
> func2(a2)
1 2
3 4
> func2(b2)
1 2 3
4 5 6
7 8 9
```

The function `func2()` takes an argument of pointer to two-dimensional assumed-shape array. So, it can handle two-dimensional computational arrays with different extents, such as `a2` and `b2`. The definition of function `func2()` in the above example is equivalent to the definition below, which takes an argument of two-dimensional assumed-shape array.

```
void func2(array int p2[:][:]) { printf("%d", p2); }
```

16.13 Relationship between Computational Arrays and C Arrays

A C array is only an address or a pointer whereas a computational array in Ch is a first-class object that contains more information. As mentioned before, a computational array is declared with the type qualifier **array**. Computational arrays can support many operators while C arrays cannot. Given the same extents, the value of each element of a C array can be assigned to a computational array. For example,

```
int a[3][4];   // C array, 'a' represents an address or a pointer
int b[4][3];   // C array, 'b' represents an address or a pointer
array int A[3][4]; // Ch computational array
array int (*p)[4]; // point to computational array
array int (*p2)[:];// point to computational array of assumed shape
A = (array int[3][4])a; // OK
p = (array int[:][:])a; // OK
p = (array int[:][:])(array int [3][4])b; // OK
p = (array int[:][:])(int [3][4])b;       // OK
p2 = (array int[:][:])a;// OK
p2 = (array int[:][:])(array int [3][4])b;// OK
p2 = (array int[:][:])(int [3][4])b;      // OK
A = a;                  // OK
a = A;                  // Error
```

In the above example, the memory of array `b` with dimension 4x3 is accessed by pointer to computational arrays `p` and `p2` as array of dimension 3x4. Because `p2` is a pointer to computational array of assumed-shape, it can point to array of different extent for its second dimension. For example, p2 can also point to array `b` of dimension 4x3 as follows.

```
p2 = (array int[:][:])b;      // OK
```

C arrays can be passed to functions that take computational arrays as arguments, and vice versa. For example,

```
int f1(array int A[3][4]); // argument is computational array
int f2(int a[3][4]);       // argument is C array
f1(a); // OK
f1(A); // OK
f2(a); // OK
f2(A); // OK
```

If the variable of a C array is used as an address of the memory for the array, the address of the computational array shall be used for the equivalent code. Fro example,

```
int f3(int *a);            // argument is a pointer
f3(a);        // OK
f3(&A[0][0]); // OK
```

Chapter 17

Characters and Strings

Types **char** and **wchar_t** are used to define variables of characters and wide characters in Ch as shown below.

```
char ch = 'a';
char ch2 = '\0';  /* null character */
wchar_t wch = L'a';
```

The value of a variable of type **char** is a single character or escape sequence which is enclosed in single-quotes, as in `'x'`. A character constant has type **int** in C. Like C++, a character constant has type **char** in Ch.

The value of wide characters is the same, except prefixed by the letter L, such as `L'x'`. A wide character has the type **wchar_t**, an integer type defined in the stddef.h or stdlib.h header file. The value of a wide character containing a single multibyte character that maps to a member of the extended execution character set is the *wide character* (code) corresponding to that multibyte character, as defined by the **mbtowc** function, with a platform-dependent current locale.

In Ch, a string is a sequence of multibyte characters enclosed in double-quotes, as in `"xyz"`. Like a C compiler, Ch automatically supplies an extra null character after all strings. A wide string literal is the same, except prefixed by the letter L.

Ch and C use array of character (or wide-character) to define a string (or wide-character string) variable. For example,

```
char *str = "this is a string.";
char str2[] = "this is also a string.";
char str3[6] = "abcde"; /* the last one is '\0' */
wchar_t *wstr = L"this is a wide string.";
```

17.1 Using Functions in string.h Header File

The header **string.h** declares one type **size_t** and several functions useful for manipulating arrays of character type and other objects treated as arrays of character type. Various methods can be used to determine the lengths of the arrays, but in all cases a char * or void * argument points to the initial (lowest addressed) character of the array. If an array is accessed beyond the end of an object, the value of the last element is used. Commonly used functions declared in header file **string.h** are categorized in this section.

17.1.1 Copying Functions

Function name	Description
memcpy()	copies characters from one object to another.
memmove()	moves characters form one object to another.
strcpy()	copies one string to another.
strncpy()	copies specified number of characters of one string to another.

For example, in the code below

```
> char str1[80] = "abcdefghijk"
> strcpy(str1, "efghij")
efghij
> str1
efghij
> strncpy(str1, "klmnopqrs", 3)
klmhij
> str1
klmhij
> strncpy(str1, "tuv", 5)
tuv
> str1
tuv
>
```

the function call

```
strcpy(str1, "efghij")
```

copies the string "`efghij`" (including the terminating null character) into the array pointed to by `str1`. This function does not allocate any storage. The caller must insure that the buffer pointed to by `str1` is long enough to hold string s2 and its terminating null character. Similarly, the function call

```
strncpy(str1, "klmnopqrs", 3)
```

copies up to 3 characters from the string "`klmnopqrs`" into the buffer pointed to by `str1`. Once **strncpy()** has copied 3 characters to `str1`, it does not append a terminating null character. So, the result is "`klmhij`", rather than "`klm`". The function call

```
strncpy(str1, "tuv", 5)
```

copies up to 5 characters, including the terminating null character, from the string "`tuv`" into the buffer pointed to by `str1`. Because the length of the string "`tuv`" is less than 5, the terminating null is added. The function **strncpy()** does not allocate any storage either. The caller must insure that the buffer pointed to by `str1` is long enough to hold the characters copied to it.

17.1.2 Concatenation Functions

Function name	Description
strcat()	appends a copy of a string to the end of another.
strncat()	appends specified number of characters of a string to the end of another.

For example, in the code below

```
> char str1[80] = "abcd"
> strcat(str1, "efg")
abcdefg
> str1
abcdefg
>
```

the function **strcat()** appends a copy of the string "`efg`" (including the terminating null character) to the end of the string pointed to by `str1`, The initial character '`e`' of the second argument overwrites the null character at the end of `str1`. This function does not allocate any storage. The caller must insure that the buffer pointed to by `str1` is long enough for appending the second string and its terminating null character.

17.1.3 Comparison Functions

Function name	Description
memcmp()	compares n characters of the object to another.
strcmp()	compares two strings.
strcoll()	compares a string to another.
strncmp()	compares a specified number of characters of a string to another.
strxfrm()	transforms a string to another.

For example, in the code below

```
> char str1[80] = "abcd"
> strcmp(str1, "aacd")
1
> strcmp(str1, "abcd")
0
> strcmp(str1, "efg")
-1
>
```

the function **strcmp()** compares the string pointed to by `str1` to the strings "`aacd`", "`abcd`" and "`efg`", respectively. It returns 1, when the string `str1` is lexically greater than string "`aacd`"; zero, when the strings `str1` and "`abcd`" are identical; and -1, when the string `str1` is lexically less than "`efg`".

17.1.4 Search Functions

Function name	Description
memchr()	locates the first occurrence of a character in the object.
strchr()	locates the first occurrence of a character in a string.
strcspn()	computes the length of the maximum initial segment of a string.
strpbrk()	locates a string in another.
strrchr()	locates the last occurrence of a character in a string.
strspn()	computes the length of the maximum initial segment of a string.
strstr()	locates the first occurrence of a string in another.
strtok()	breaks the string into a sequence of tokens.

For example, in the code below

```
> char str1[80] = "abcdefgdef"
> strchr(str1, 'd')
defgdef
> strchr(str1, 'w')
00000000
> strstr(str1, "def")
defgdef
> strstr(str1, "dev")
00000000
> strtok(str1, "efg")
> char *str2 = "abcd;1234 ABCD"
> char *delimiter=" ;", *token
> token = strtok(str2, delimiter)
abcd
> token = strtok(NULL, delimiter)
1234
> token = strtok(NULL, delimiter)
ABCD
> token = strtok(NULL, delimiter)
(null)
>
```

the function call

```
strchr(str1, 'd')
```

locates the first occurrence of character 'd' in the string pointed to by `str1`, and returns a pointer to the location. As the character 'w' does not occur in the string `str1`, the function call

```
strchr(str1, 'w')
```

returns a null pointer. Similarly, the function call

```
strstr(str1, "def")
```

finds the first occurrence of substring "`def`" within string `str1`, exclusive of the terminating null character, and returns a pointer to this substring. Since the substring "`def`" cannot be found in the string `str1`, the function call

```
strstr(str1, "dev")
```

returns a null pointer. The function **strtok()** gets the next token from a string. The tokens are strings separated by characters specified by the second argument. To get the first token from the string `str2`, the function call

```
token = strtok(str2, delimiter)
```

use `str2` as its first parameter. The subsequent function calls

```
token = strtok(NULL, delimiter)
```

with null pointers for the first parameters return all other tokens from `str1` one after another. The second argument is the string of delimiters which can differ from call to call. The section 17.3 introduces the **foreach** loop to obtain tokens from a string.

17.1.5 Miscellaneous Functions

Function name	Description
memset()	copies a value into each of the first specified number of characters of an object.
strerror()	maps the number in the errnum to a message string.
strlen()	computes the length of a string.

For example,

```
> strlen("abcde")
5
>
```

where the function **strlen()** returns the length of the string "`abcde`". The terminating null of the string is not counted by the function **strlen()**, so that the result is 5, instead of 6, in this case. If this function is used to calculate the size of the dynamically allocated memory for a string, the return value should be added 1.

17.1.6 String Functions Supported by Ch, but not in C Standard Library

Function name	Description
strcasecmp()	compare two strings, ignoring case.
strconcat()	concatenates strings.
strjoin()	combines strings to a string separated by the specified delimiter string.
strncasecmp()	compare part of two strings, ignoring case.

For example, in the code below,

```
> char *buffer
> char test[90] = "abcd"
> buffer = strconcat(test, "efgh", "ijk")
abcdefghijk
> free(buffer)
> buffer = strjoin("+", test, "efgh", "ijk")
abcd+efgh+ijk
> free(buffer)
>
```

assume the character array `test` has the value of string `abcd`, the function call

```
buffer = strconcat(test, "efgh", "ijk")
```

concatenates these three strings, and puts the result into the returned string with dynamically allocated memory. The dynamically allocated memory need to be freed later by the user. The function call

```
buffer = strjoin("+", test, "efgh", "ijk")
```

also combines these three strings to the returned string with dynamically allocated memory. But, the returned string is separated by a delimiter string "`+`" which is specified by the first argument of the function **strjoin()**.

Table 17.1: Functions for type **string_t**.

Function Name	Description
str2ascii()	get the ASCII value of a string.
str2mat()	change strings to a matrix.
stradd()	add the second string to the first one.
strgetc()	get a character from a string.
strputc()	put a character into a string.
strrep()	replace a string within a string by another string.

17.2 String Type string_t

C has no string data type. As mentioned above, arrays of characters are handled as strings in C. In Ch, a string data type **string_t** is added. It is seamlessly merged with pointer to char. All functions defined in the standard C library header string.h are valid for both pointers to char and strings. String through the string type **string_t** is a first-class object in Ch. For example, the following code fragment

```
string_t s, a[3];
s = "great string"
s = stradd("greater ", s)
strcpy(a[0], s);
printf("a[0] = %s\n", a[0]);
```

will display `greater great string`. **string_t** is a keyword in Ch and the function stradd() is a built-in generic function. Format specifier `"%s"` can be used to obtain input to a variable of string type as shown in the commands below.

```
> string_t s
> scanf("%s", &s)
123abc
> printf("%s", s)
123abc
>
```

For string functions **strcpy()**, **strncpy()**, **strcat()**, and **strncat()**, the memory will be automatically handled if the first argument is of the type **string_t**. For example,

```
> string_t s
> strcpy(s, "abcd")
abcd
> strcat(s, "ABCD")
abcdABCD
> s
abcdABCD
>
```

In Ch, the header file **string.h** delares some additional functions, which are listed in Table 17.1, for the type **string_t** specifically. They mainly include **str2ascii()**, **str2mat()**, **strgetc()**, **strputc()**, and **strrep()**. For example, in the code below

```
> str2ascii("a")
97
> str2ascii("b")
98
> str2ascii("ab")
195
> array char mat[3][10]
> str2mat(mat, "abcd", "0123456789")
0
> mat
a b c d
0 1 2 3 4 5 6 7 8 9
> str2mat(mat, "ABCD", "EFGH", "ab23456789", "too many strings")
-1
> mat
A B C D
E F G H
a b 2 3 4 5 6 7 8 9
> string_t s1 = "abcd"
> stradd(s1, "efg")         // add "efg" to s1
abcdefg
> strgetc(s1, 0)            // get the first character of s1
a
> strgetc(s1, 2)            // get the third character of s1
c
> strputc(s1, 2, 'z')       // change the third character to 'z'
0
>
s1
> abzdefg
> strrep(s1, "def", "xyz")  // replace "def" with "xyz" in s1
abzxyzg
>
```

the function call

```
str2ascii("ab")
```

calculates the ASCII value of the string "`ab`" by adding up the ASCII values of characters '`a`' and '`b`'. The function call

```
str2mat(mat, "abcd", "0123456789")
```

assigns two strings "`abcd`" and "`0123456789`" to the first two rows of array `mat`, and return `0` upon successful completion. The rest rows of `mat` retain null. If strings in the argument list are more than the rows in the array `mat`, such as

```
str2mat(mat, "ABCD", "EFGH", "ab23456789", "too many strings")
```

the function will return `-1`, and ignore the string "`too many strings`". The function **stradd()** is a generic function for adding a string to another. Ch will handle the memory for users. Assume the string `s1` with value of "`abcd`" has type of **string_t**, the function call

```
stradd(s1, "efg")
```

adds the string "`efg`" to the end of `s1`, and then returns `s1`. The function calls

```
strgetc(s1, 0)
strgetc(s1, 2)
```

return the first and the third characters, i.e. '`a`' and '`c`', of the string, respectively. Functions **strgetc()** and **strputc()** are particularly useful for manipulating characters inside a string. The function call

```
strputc(s1, 2, 'z')
```

changes the third character in the string `s1` to '`z`'. The function call

```
strrep(s1, "def", "xyz")
```

replaces the string "`def`" in `s1` with the string "`xyz`, and return `s1`.

As it is mentioned above, one of the advantages of type **string_t** is that Ch can handle the memory for variables of type **string_t** automatically. For every operations on these variables, Ch will figure out the size of the memory required, and then allocate enough memory for the variables. At the end of the lifetimes of these variables, Ch will free their memory automatically. For example, in the following program, the memory of the variable `s1` in the function `fun()` is freed upon exit of the function, and the memory of variable `s2` is freed at the return of function, or at the assignment of the variable `s` in the **main()** function. On the other hand, the memory for `s` is allocated automatically at its assignment.

```
string_t fun() {
  string_t s1;
  string_t s2;
  ...
  return s2;
}
int main() {
  string_t s;
  fun();
  s = fun();
  ...
}
```

17.3 Handling String Tokens Using foreach Loop

Besides the **while** loop, **do-while** loop and **for** loop, the **foreach** loop presented in section 8.4.4 is especially convenient for handling of strings. It causes one piece of text to be used repeatedly, each time with a different substitution performed on it. This gives an easy way to handle strings or to iterate over arrays.

For example, the function **strtok()** or **strtok_r()** can be used to retrieve tokens in a null-terminated string. The following code fragment

```
char *s = "abcd;1234 ABCD;56;xyz";
char *delimiter=" ;", *token;
token = strtok(s, delimiter);
while(token) {
    printf("token = %s\n", token);
    token = strtok(NULL, delimiter);
}
```

will output

```
abcd
1234
ABCD
56
xyz
```

We can rewrite this example with a **foreach** loop as follows.

```
char *s = "abcd;1234 ABCD;56;xyz";
char *delimiter=" ;", *token;
foreach(token; s; NULL; delimiter)
    printf("token = %s\n", token);
```

If we replace **NULL** in the above code with the string "ABCD" as a value for `cond` of **foreach** loop, the code fragment becomes

```
char *s = "abcd;1234 ABCD;56;xyz";
char *delimiter=" ;", *token;
foreach(token; s; "ABCD"; delimiter)
    printf("token = %s\n", token);
```

The output of the above code is

```
abcd
1234
```

17.4 Wide Characters

A wide character constant has type **wchar_t**, an integer type defined in the stddef.h header. It is a sequence of one or more multibyte characters enclosed in single-quotes, as in `'x'` or `'ab'` prefixed by the letter L. The value of a wide character constant containing a single multibyte character that maps to a member of the extended execution character set is the *wide character* (code) corresponding to that multibyte character, as defined by the **mbtowc** function, with platform-dependent current locale. The value of a wide character constant containing more than one multibyte character, or containing a multibyte character or escape sequence not represented in the extended execution character set, is platform-dependent.

For example, the definition of a wide character variable `wc` is shown below.

```
wchar_t wc = L'a';
```

To effectively use wide characters and strings in Ch command shell for multi-byte languages such as Simplified Chinese, Russian, Japanese, etc. To use functions in the header files **wchar.h** and **wctype.h**, add the statements

```
#include <locale.h>
#pragma exec setlocale(LC_ALL, "Chinese-Simplified");
```

for a specific Unicode, or

```
#include <locale.h>
#pragma exec setlocale(LC_ALL, "");
```

for the default Unicode of the system at the beginning of a program. Or add the statement

```
setlocale(0, "Chinese-Simplified");
```

for a specific Unicode, or

```
_setlocale = 1;
```

or

```
setlocale(LC_ALL, "");
```

for the default Unicode of the system in the individual user's startup file **_chrc** in the user's home directory or in the system startup file **CHHOME/config/chrc** so that the setup will be effective for all programs.

17.5 Wide Strings

A wide-character string constant is a sequence of zero or more multibyte characters enclosed in double-quotes prefixed by the letter L.

The following code

```
wchar_t *wstr = L"abcd";
```

defines a wide-character string `wstr` in Ch.

The function **mbstowcs()** declared in the file **stdlib.h** can convert a multibyte string to a wide-character string and the function **wcstombs()** does it contrarily.

The header file **wchar.h** declares some data types, tags, macros, and functions for wide characters.

Chapter 18

Structures, Unions, Bit Fields, and Enumerations

18.1 Structures

The structure types in Ch are similar to those in C++. They are collections of members that can have different types. For example, the type of complex in Ch is equivalent to the following definition of structure.

```
struct Complex{
     float r;
     float m;
};
```

where two members `r` and `m` are used to hold the real and imaginary parts of a complex number. `Complex` is called the tag of the structure.

The following code fragment can create an object of the type `Complex`.

```
Complex z;
z.r = 10;
z.m = 5;
```

where the selection operator "`.`" is used to access members of the structure. The member `r` is set to 10 and `m` to 5. If the variable is defined as a pointer to a struct, the operator "`->`" is used to access its members. For example,

```
Complex *pz = &z;
pz->r = 10;
pz->m = 5;
```

There are two namespaces for structures in C, one for structures' tags and one for variables. But there are one and an half namespaces for structures in C++, one for tags and a half for variables. Ch handles structures the same way as the latter. Tags and variables share the same namespace. Once a tag name is used as a variable explicitly, it will not be treated as a typed name implicitly in Ch. For example,

```
struct tag1_t {
    struct tag2_t;
    ....
};
```

```
tag1_t s;          // ok
int tag1_t;        // ok
struct tag1_t s2;  // ok
tag1_t s3;         // error, tag1_t is a variable of int
struct tag2_t s4;  // Not valid in Ch and C++, OK in C
```

18.2 Unions

A union type describes an overlapping non-empty set of member objects, each of which has an optionally specified name and possibly distinct type. Like structures, unions can have some members. Unlike structures, a union can only hold one of its members at a time. The members are conceptually overlaid in the same memory. Each member of a union is located at the beginning of the union.

For example, below is a union with three members.

```
union U1{
     double d;
     char c[12];
     int i;
} obj, *P = &obj;
```

then the following equalities hold.

```
(union U1*)&(P->d) == (union U1*)(P->c) == (union U1*)&(P->i) == P
```

The size of an instance of a union is the amount of memory necessary to represent the largest member, plus the padding that raises the length up to an appropriate alignment boundary. In the previous example, the following equalities hold.

```
sizeof U1 == 16
```

although the largest member `c` occupies only 12 bytes memory.

Because a union only holds one member at a time, if two or more members are used without casting, the result could be strange. For example, the following code fragment

```
obj.i = 10;
printf("obj.d = %f\n", obj.d);
```

will print out zero or a tiny value instead of 10, because of the differences in representations of int variables and float variables.

There are one and half namespaces for union in Ch and C++, one for struct tags and half for variables. Like C++, a union tag, such as `U1`, is put into typedefed namespace by default in Ch.

18.3 Bit-fields

Like C, Ch offers bit-fields which have the capability of defining and accessing within a word directly.

In the following code fragment,

```
struct Bf1 {
    unsigned int a;
    unsigned int b;
    unsigned int c;
}  bf1 = {1, 1, 1};

struct Bf2 {
    unsigned int a : 4;
    unsigned int b : 4;
    unsigned int c : 4;
}  bf2 = {1, 2, 3};

bf2.c = 4;
printf("sizeof Bf1 is %d\n", sizeof(struct Bf1));
printf("sizeof Bf2 is %d\n", sizeof(struct Bf2));
```

the size of `Bf1` is 12 because there are three integers inside the structure. But, the size of `Bf2` is 4 because three members only take 12 bits memory, plus padding.

Consider the bit field below.

```
struct eeh_type {
  uint16 u1: 10;   /* 10 bits */
  uint16 u2:  6;   /* 6 bits */
};
```

This might actually be implemented as

```
<10-bits><6-bits>
```

or as

```
<6-bits><10-bits>
```

depending on the endian type of the machine and operating system.

The selection operator "`.`" can be used to access the members of a bit-field. For example, the following equalities hold.

```
bf2.a == 1;
bf2.b == 2;
bf2.c == 4;
```

18.4 Enumerations

An enumerated type is a set of integer values represented by enumeration constants.

For example, the declaration

```
enum datatypes {
    inttype,     // 0
    floattype,   // 1
    doubletype,  // 2
} d1, d2;
```

creates a new enumerated type, `enum datatypes`, whose values are `inttype`, `floattype` and `doubletype`. It also declares two variables of the enumerated type `d1` and `d2`, which can be assigned enumeration constants with the following assignment statement.

```
d1 = inttype;
d2 = doubletype;
```

The first enumeration constant receives the value 0 by default and the subsequent enumeration constants receive an integer value one greater than the previous enumeration constant. The values of `d1` and `d2` will be 0 and 2, respectively.

An explicit integer value can be associated with an enumeration constant in the definition. For example, given the declaration,

```
enum datatypes {
    inttype,          // 0
    floattype = 10,   // 10
    doubletype        // 11
};
```

the value of `inttype`, `floattype`, `doubletype` will be 0, 10 and 11, respectively.

Enumerated type can be used to replace the **#define** directive in some applications. The following code fragment uses a variable of enum type in a **switch** statement.

```
enum datatypes {
    inttype,
    floattype,
    doubletype
};
enum datatype dt1;

...

switch(dt1) {
    case inttype:
        ...
        break;

    case floattype:
        ...
        break;

    case doubletype:
        ...
        break;
}
...
```

Chapter 19

Classes and Object-Based Programming

19.1 Class Definition and Objects

The class in C++ and Ch is a natural evolution of the structure. Class can be used to create user-defined types. Functions can be members of a class but not members of a structure in C. Like C++, both class and struct in Ch can have member functions. By default, members of a class are private whereas members of a struct are public.

The following is an example of the definition of a class.

```
class Student {
    int id;
    char *name;
};
```

The class `Student` has two members. Assume `id` holds the ID number of a student and `name` is the name of the student. After defining a class, it can be used in the program as shown below.

```
int main() {
    class Student s1;
    ....
}
```

where `s1` is called an *object* or an *instance* of class `Student`.

19.2 Member Functions of Class

As it is mentioned above, function can be a member of a class. We can redefine the class of `Student` in a header file `student.h` as follows.

```
/* Filename: student.h */
#ifndef STUDENT_H
#define STUDENT_H

class Student {
     int id;
     char *name;
   public:
```

```
    void setID(int i);
    void setName(const char *n);
    int getID();
};

#pragma importf <Student.cpp>

#endif
```

The member functions are defined in a separate file `Student.cpp` shown below. This file located in a directory specified by the system variable **_fpath** are loaded once by the statement.

```
#pragma importf <Student.cpp>

 /* Filename: Student.chf */
 #include <string.h> /* for strdup() */
 #include "student.h"

 void Student::setID(int i) {
     id = i;
 }

 void Student::setName(const char *n) {
     if(n)
       name = strdup(n);
 }

 int Student::getID() {
     return id;
 }
```

In the definition of a member function, a function name is preceded by the class name and the scope resolution operator '`::`', which will be explained later. The function `setID()` takes the ID number of a student as the argument and then sets the class member `id` to it. The function `setName()` sets the member `name` to a new name. The function `getID()` gets the ID of the student. The members `setID()`, `setName()`, and `getID()` are called *member functions* or *methods* in Ch and C++. One can invoke a member function by using the member operator '`.`' which is just like accessing a member of a structure as shown in program `prog.cpp`.

```
 /* Filename: prog.cpp */
 #include <iostream.h>  /* for cout */
 #include "student.h"

 using namespace std;   /* for cout */
 int main() {
     class Student s;
     s.setID(1);
     s.setName("Jason");
     cout << "id is " << s.getID() << endl;
```

```
    return 0;
}
```

Private members of a class such as `name` and `id` are normally not accessible outside the class, which is called information hiding. One of the main roles of member functions of a class is to provide a means to access private members of the class.

19.3 Public and Private Members of Class

As it is pointed out before, the members `id` and `name` of class `Student` are not accessible outside the class. The outside code has access to them only through some member functions of the class. This is because they are private members and all of the member functions defined in the class `Student` are public members. Ch has two member access specifiers **public:** and **private:**. They can appear multiple times and in any order in a class definition. By default, members in class are private whereas members in struct are public. We can write the definition of the class `Student` like

```
class Student {
  public:
    void setID(int i);
    void setName(const char *n);
  private:
    int id;
    char *name;
};
```

Normally the data members of a class are defined as private members and member functions are defined as public members. The set of public member functions of a class is called the `interface`. But in certain cases, we may need to define a public data member or a private member function. A public data member is accessible outside the class by the member operator '`.`' just like public member functions. On the other hand, the private member functions can only be called by other member functions of the class.

19.4 Constructors and Destructors in Class

Data members of a class cannot be initialized in the class definition in Ch. Initializations can be done in a constructor. The constructor and destructor are member functions, which have no return value specified. The constructor has the same name as the class name. It is invoked automatically each time an object is instantiated and performs some initializations. A constructor can take arguments for initializing its data members while a destructor can not take any argument. For example, we can add constructor and destructor for class `Student` as follows.

```
class Student {
  public:
    Student(int, const char *);  // constructor
    ~Student();            // destructor
    void setID(int i);
    void setName(const char *n);
  private:
    int id;
```

```
    char *name;
};

Student::Student(int i, const char *n) {
    id = i;            /* initialize id */
    name = strdup(n);  /* initialize name */
}

Student::~Student() {
    /* release the memory allocated in constructor */
    free(name);
}
```

where the constructor sets the data member `id` and `name` when the new object is created, and the destructor frees the memory allocated in the constructor.

The declaration with initialization in the `main` function is shown below.

```
int main() {
    class Student s1 = Student(1, "Jason");
    class Student s2 = Student(2, "Bob");
    ....
}
```

After the constructors are called during the declarations, the data members of `s1` and `s2` have been set. When the function **main()** terminates, the destructor will be called.

19.5 The new and delete Operators

In C, the dynamic memory allocation and deallocation are normally performed by the functions **malloc()** and **free()**. In Ch and C++, the operator pair **new** and **delete** can do the same thing as **malloc()** and **free()** and also provide other benifits.

The operator **new** can calculate automatically the proper size of the memory to be allocated while the function **malloc()** must take an argument as the size of the memory. The operator **new** can return a pointer of the correct type while the function **malloc()** only returns a pointer to void. The most important thing is that the operator **new** can invoke the contructor of a class automatically and perform the initialization if necessary while the function **malloc()** does not provide any initialization of the memory allocated. The corresponding destructor will be called by the operator **delete**.

The following code fragment shows how the operators of **new** and **delete** are used.

```
int main() {
    class Student *s1 = new Student (1, "Jason");
    class Student *s2 = new Student (2, "Bob");
    ...

    s1->setID((5); // change ID of s1 to 5
    ...

    delete s1;
    delete s2;
}
```

To users, this example does the same thing as the previous one including initializations, but it is more flexible and useful in some cases.

If the attempt to allocate memory is successful, the operator `new` returns a pointer to the allocated memory. Otherwise, it calls the handler function pointed to by **_new_handler** if **_new_handler** is not NULL, and then returns a NULL pointer. A program may install different handler functions for **new** operator during execution, by supplying a pointer to a function defined in the program or the library as an argument to the function **set_new_handler()**. The function **set_new_handler()** is defined in header file **new.h** as follows.

```
void (*set_new_handler (void(*)()))();
```

It establishes the function designated by the argument for the current **_new_handler**, and returns NULL on the first call, or the previous **_new_handler** on subsequent calls. Program 19.1 sets the function `newhandle()` as the handler function for **new** operator, and then allocates memory for variables `p` and `sp`, respectively. The system has enough memory for `p`, but not for `sp`. The function `newhandler()` is called when the **new** operator fails to allocate memory for `sp`. The output from executing Program 19.1 is appended at the end of the file.

For a variable of pointer to class, the operator '`->`' shall be used to access a member of the class.

The **new** and **delete** operators can handle not only a single value, but also an array. For example, the following code

```
class Employee {
    char *name;
};
int main() {
    class Employee *e = new Employee[10];
    ....

    delete [10] e;
}
```

instantiates 10 new objects of the class `Employee`. At the end of the program, all of these 10 objects will be deleted.

19.6 Static Member of Class

Typically each object of a class has its own copy of the data members in memory. But in certain cases, different objects of a class need to use some "class-wide" information. That means they have to share the same copy of a variable. A static class variable can provide this mechanism. The values of a static member in all objects of a class are the same. The change of its value affects all objects. Even if no object of a class exists, the static member is still there and can be manipulated. The declaration of a static member begins with the keyword **static**. For example, a static member `count` can be added to the definition of class `Student` as shown below,

```
class Student {
    // number of objects instantiated
    static int count;
    int id;
    char *name;
  public:
```

```
#include <new.h>
#include <stdio.h>

struct tag{ int i; int j[900000];} s;
void newhandler(void);

int main() {
  set_new_handler(newhandler);

  int *p = new int[20];
  if(p==NULL)
    printf("not enough memory for p\n\n");
  else
    printf("enough memory for p\n\n");


  tag *sp = new tag[90];
  if(sp==NULL) {
    printf("not enough memory for sp\n");
    printf("sp = %p\n", sp);
  }
  else {
    printf("enough memory for sp\n");
    printf("sp = %p\n", sp);
  }
}

void newhandler(void) {
   printf("message from newhandler\n");
}

/**** result of the program
newhandler.ch
enough memory for p

message from newhandler
not enough memory for sp
sp = 00000000
****/
```

Program 19.1: Setting handler function for **new** operator.

```
    Student(int, char *);
    ~Student();
    void setID(int i);
    void setName(const char *n);
};
```

where the member `count` maintains the count of objects of class `Student`. The static data member can be initialized with the following statement along with definitions of other member functions. A static data member must be initialized once at file scope. For example,

```
int Student::count = 0;
```

The member `count` can be referenced through any member function of `Student` object. In this example, the constructor will add 1 to `count` and the destructor will subtract 1 from it. We can rewrite the constructor and destructor as follows,

```
Student::Student(int i, char *n) {
    id = i;            /* initialize id */
    name = strdup(n);  /* initialize name */
    count++;
}

Student::~Student() {
    /* release the memory allocated in constructor */
    free(name);
    count--;
}
```

Like C++, Ch not only has static members of simple data types but also has static member functions such as the member function `getCount()` defined below.

```
class Student {
    // number of objects instantiated
    static int count;
    int id;
    char *name;
  public:
    Student(int, char *);
    ~Student();
    void setID(int i);
    void setName(const char *n);
    static int getCount();
};

int Student::getCount() {
    return count;
}
```

The function can be used to get the count of objects currently instantiated as shown below.

```
int main() {
    class Student s1 = Student(1, "Jason");
    ....

    cout << "Number of student is "
         << s1.getCount() << endl;
    ....
}
```

The static member function can be called even though there is no object instantiated. In other words, getCount() can be called using the following statement before `s1` is instantiated. For example,

```
int main() {
    ....
    cout << "Number of student is "
         << Student::getCount() << endl;
    ....
}
```

19.7 Scope Resolution Operator ::

Ch and C++ provides a unary scope resolution operator '`::`' to access a global variable when a local variable of the same name is in the scope. For example,

```
#include <stdlib.h>
    int num;
    int main() {
      int num;
      num = 10;        // use local num
      ::num = ::num+2  // use global num

      ...

      ::exit(0);       // use C function exit()
    }
```

Furthermore, this operator is used very often with classes. We have already used the scope resolution operator '`::`' in the previous examples. It is mainly used in the following occasions.
1. Member function definition. When a member function is defined after the class definition, the function name is preceded by the class name and the scope resolution operator '`::`'. Because different classes can have members with the same name, the scope resolution can prevent the confusion. For example, the member function getCount() in the previous examples is defined with the code fragment below.

```
... /* definition of the class Student */

int Student::getCount() {
    return count;
}
...
```

2. Accessing the static member. As mentioned above, if no object of a class exists, the static members of that class are still accessible by adding the class name and the scope resolution operator '`::`' in front of the static data member name. For example, the code

```
int main() {
    ....
    cout << "Number of student is "
         << Student::getCount() << endl;
    ....
}
```

can print out the number of the object of the class `Student` even when no object has been declared.

19.8 The Implicit this Pointer

In Ch and C++, every object has an implicit pointer called `this` to point to its own address. Although the pointer `this` is not regarded as a part of the object, i.e., it is not reflected in the **sizeof()** operation, it is actually implicitly used to reference the data members and member functions of an object. The following definitions of member functions

```
void setID(int i) {
    id = i;
}

void setName(const char *n) {
    if(n)
      name = strdup(n);
}
```

are equivalent to

```
void Student::setID(int i) {
    this->id = i;
}

void Student::setName(const char *n) {
    if(n)
      this->name = strdup(n);
}
```

where `this` pointers are used explicitly. A static member function has no `this` pointer because it exists independent of any object of a class.

19.9 Polymorphism

Although Ch does not support operator overloading at the user's level. Commonly used mathematical operators are overloaded internally to handle operands of different data types. For example, operator '+' can be used for addition of integral numbers, floating-point numbers, complex numbers, and computational arrays of different data types.

Ch supports polymorphism — the ability for normal functions and member functions of classes to respond differently to the same function calls, but with different argument numbers and types. Ch does not support C++ function overloading at the user's level, in which a function can be defined multiple times with different data types and arguments. This is achieved by mangling function names internally inside a C++ compiler. This name mangling is not suitable for interpretive implementation with a single pass because of its overhead. The polymorphism in Ch is implemented mainly by the function reloading. How polymorphic functions are handled in Ch will be summarized in this section.

19.9.1 Polymorphic Generic Mathematical Functions

Commonly used generic mathematical functions such as $\sin()$ are polymorphic. They can handle arguments of integral values, floating-point values, complex values, and computational arrays of different data types and sizes. Generic mathematical functions for real numbers, complex numbers, and computational arrays of different data types and sizes have been described in sections 12.1, 13.5, and 16.10, respectively. Below is an example of calling generic mathematical function **sin**() with different arguments in a Ch shell.

```
> float f = 1.0
> sin(f)       // call with a float
0.84
> double d = 1.0
> sin(d)       // call with a double
0.8415
> complex float zf = 1
> sin(zf)      // call with float complex
complex(0.84,0.00)
> complex double zd = 1
> sin(zd)      // call with double complex
complex(0.8415,0.0000)
> array float af1[2] = {1.0, 2.0}
> sin(af1)     // call with a one-dimensional array
0.84 0.91
> array float af2[2][3] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0}
> sin(af2)     // call with a two-dimensional array
0.84 0.91 0.14
-0.76 -0.96 -0.28
> array double ad1[2] = {1.0, 2.0}
> sin(ad1)     // call with a double array
0.8415 0.9093
> array complex double az1[2] = {1.0, 2.0}
> sin(az1)     // call with a complex array
complex(0.8415,0.0000) complex(0.9093,-0.0000)
>
```

19.9.2 Functions with Parameter Type of Array of Reference

A function with parameters of array of reference can be used to handle array arguments of different dimensions and data types. For example, the function `func`() with prototype

```
int func(double a[&][&], array double &b);
```

can take arguments of different dimensions and types as follows.

```
int func(double a[&][&], array double &b);
array double A1[2][3], B1[10];
array float  A2[4][5], B2[3][4];
int func(A1, B1);
int func(A2, B2);
```

Details about using functions with parameter type of array of reference to pass arguments of different dimensions and data types are described in section 16.7.5.

19.9.3 Polymorphic Functions

Polymorphic generic mathematical functions are implemented as built-in functions in Ch. Using facilities in the standard library defined in header file **stdarg.h** described in section 10.7, a user can write polymorphic functions. Header file **stdarg.h** contains function prototypes and macros listed in Table 10.1 for handling arguments of variable length. Through some sample code, this section will illustrate how to use macros **VA_NOARG**, **va_count**, **va_datatype**, **va_arraydim**, **va_arrayextent**, **va_arraynum**, and **va_arraytype** to implement polymorphic functions in a user's program.

Function `func`() in Program 19.2 will print out the values of the variable number arguments of different data types. Inside function, the macro **va_count**() returns the number of the remaining arguments in the argument list. Thus the while-loop

```
while(va_count(ap)) {
  ...
}
```

can retrieve all of the arguments one by one. The output of Program 19.2 is shown in Figure 19.1.

```
#include<stdarg.h>

void func(...) {
  va_list ap;
  int arg_num = 0;
  int i;
  float f;
  double d;

  va_start(ap, VA_NOARG);
  while(va_count(ap)) {
    if(va_datatype(ap)  == elementtype(int)) {
      i = va_arg(ap, int);
      printf("the %d argument is int %d\n", ++arg_num, i);
    }
    else if(va_datatype(ap)  == elementtype(float)) {
      f = va_arg(ap, float);
      printf("the %d argument is float %f\n", ++arg_num, f);
    }
    else if(va_datatype(ap)  == elementtype(double)) {
      d = va_arg(ap, double);
      printf("the %d argument is double %f\n", ++arg_num, d);
    }
  }
  va_end(ap);
  return;
}

int main(){
  int i = 10;
  float f = 2.0;
  double d = 3.0;

  func(i, f);
  func(f, i, d);  // different types and different order
  return 0;
}
```

Program 19.2: A polymorphic function handling variable length arguments.

```
the 1 argument is int 10
the 2 argument is float 2.000000
the 1 argument is float 2.000000
the 2 argument is int 10
the 3 argument is double 3.000000
```

Figure 19.1: Output of Program 19.2.

In addition to regular data types, arrays of different data types can be passed as pointers in a variable-length argument list. The macros **va_arraytype**(), **va_datatype**(), **va_arraydim**(), **va_arrayextent**() and **va_arraynum**() can be used to get the data type, dimension, extent, and number of elements of a computational array or C array. These macros must be called before the macro **va_arg** is called. For example, in

function `fun()` of Program 19.3, the statements

```
if(va_arraytype(ap) == CH_UNDEFINETYPE) { // check if it is not an array
  printf("the argument is not an array\n");
  return -1;
}
```

give an error message if the argument is not an array. The statements

```
if(va_datatype(ap) == elementtype(int)) {
/* if(va_datatype(ap) == CH_INTTYPE) {
  printf("elementtype = int\n");
}
else {
  printf("elementtype = other types\n");
}
```

determine if the type of the argument is **int** by calling macro **va_datatype**() and generic function **elementtype()**, and then print out the corresponding message. The generic function **elementtype()** gives the data type defined in the header file **stdarg.h**. Because both **char*** and **string_t** represent strings in Ch, the following statement could be used in some occasions to determine the string type.

```
if(va_datatype(ap)==elementtype(char*)
   || va_datatype(ap)==elementtype(string_t)) {
  printf("element type is string\n");
}
else {
  printf("element type is not string\n");
}
```

In this example, statements

```
dim = va_arraydim(ap);
num = va_arraynum(ap);
```

gets the dimension and number of elements of the array argument. The statement

```
m = va_arrayextent(ap, 0);
```

gets the number of elements in the first dimension of the array. If the array has one dimension, the statements

```
array int *p;
ptr = va_arg(ap, int*);
p = (array int [:])(int [m])ptr;
```

make `p` share the memory with one-dimension arrays, such as `A3` in the `main()` function. If the array has two dimensions, the statements

```
array int (*p)[:];
n = va_arrayextent(ap, 1);
ptr = va_arg(ap, int*);
p =  (array int [:][:])(int [m][n])ptr;
```

make `p` share the memory with two-dimension arrays, such as `A1` and `A2` in the `main()` function. `A1` is a fully-specified-shape computational array and `A2` is an assumed-shape computational array. Like `A1`, they are passed into function `fun()` as pointers to int. More information about computational arrays can be found in Chapter 16. The output of Program 19.3 is shown in Figure 19.2.

```
#include <array.h>
#include <stdarg.h>
int fun(...) {
   int *ptr, count, dim,num, m, n;
   va_list ap;

   va_start(ap, noarg);
   count = va_count(ap);
   printf("count = %d\n", count);
   if(count >= 1) {
     if(va_arraytype(ap) == CH_UNDEFINETYPE) { // check if it is not an array
       printf("the argument is not an array\n");
       return -1;
     }
     dim = va_arraydim(ap);     // get the dimension
     printf("dim = %d\n",dim);
     num = va_arraynum(ap);     // get the number of element
     printf("num= %d\n",num);
     if(va_datatype(ap)  == elementtype(int)) // get the type
     // or if(va_datatype(ap)  == CH_INTTYPE)
       printf("elementtype = int\n");
     else
       printf("elementtype = other types\n");
     m = va_arrayextent(ap, 0);  // get the extent
     if(dim == 1) {
       printf("extent1 = %d\n",m);
       array int *p;
       ptr = va_arg(ap, int*);
       p =  (array int [:])(int [m])ptr;
       printf("p =\n%d\n", p);
     }
     if(dim == 2) {
       array int (*p)[:];
       n = va_arrayextent(ap, 1);
       printf("extent1 = %d, extent2 = %d\n", m, n);
       ptr = va_arg(ap, int*);
       p =  (array int [:][:])(int [m][n])ptr;
       printf("p =\n%d\n", p);
     }
   }
   va_end(ap);
   return 0;
}

int main() {
     array int A1[2][3];
     array int (*A2)[:];
     array int A3[3] = {1, 2, 3};

     A1 = (array int [2][3])50;
     fun(A1);

     A1 = (array int [2][3])80;
     A2 = (array int [:][:])A1;
     fun(A2);
     fun(A3);
}
```

Program 19.3: Pass computational arrays to functions.

```
count = 1
dim = 2
num= 6
elementtype = int
extent1 = 2, extent2 = 3
p =
50 50 50
50 50 50

count = 1
dim = 2
num= 6
elementtype = int
extent1 = 2, extent2 = 3
p =
80 80 80
80 80 80

count = 1
dim = 1
num= 3
elementtype = int
extent1 = 3
p =
1 2 3
```

Figure 19.2: Output of Program 19.3.

One restriction of polymorphic functions in Ch is that functions cannot return values of different data types. In case results of different data types are to be obtained from a function, it can be implemented by passing a pointer as an argument to retrieve value of different types. For example, function `func`() in Program 19.4 can retrieve values of int or float as the returned values through the first argument which is a pointer. The output of Program 19.4 is shown in Figure 19.3.

19.9.4 Polymorphic Member Functions of Class

Not only functions, but also constructors and member functions of classes in Ch can be polymorphic using facilities in header file **stdarg.h** for handling variable length arguments.

In Program 19.5, both constructor `C1`() and member function `memfunc()` of class `C1` can take variable number of arguments of int type. Objects `c1` and `c2` are instantiated by using the constructor with one and two arguments, respectively. The output of Program 19.5 is shown in Figure 19.4.

```
#include<stdarg.h>

void func(...) {
  va_list ap;
  int *pi, flag;
  float *pf;

  va_start(ap, VA_NOARG);
  if(va_count(ap) != 2) {
    printf("need 2 arguments\n");
    return;
  }

  if(va_datatype(ap)  == elementtype(int *)) { // get the 1st argument
    pi = va_arg(ap, int*);
    flag = 1;
  }
  else if(va_datatype(ap)  == elementtype(float *)) {
    pf = va_arg(ap, float*);
    flag = 2;
  }

  if(va_datatype(ap)  == elementtype(int)) {// get the 2nd argument
    if(flag == 1)
      *pi = va_arg(ap, int);
  }
  else if(va_datatype(ap)  == elementtype(float)) {
    if(flag == 2)
      *pf = va_arg(ap, float);
  }
  va_end(ap);
  return;
}

int main(){
  int ret_i, i = 10;
  float ret_f, f = 1.0;

  func(&ret_i, i);
  printf("ret_i = %d\n", ret_i);
  func(&ret_f, f);
  printf("ret_f = %f\n", ret_f);
  return 0;
}
```

Program 19.4: Polymophism of functions returning different data type.

```
ret_i = 10
ret_f = 1.000000
```

Figure 19.3: Output of Program 19.4.

```
#include<stdarg.h>
#include<stdio.h>

class C1 {
    double m_d;
  public:
    C1(...);      // constructor taking variable length arguments
    void memfunc(...); // member function taking variable length arguments
};

C1::C1(...) {
  va_list ap;
  int vacount;

  va_start(ap, VA_NOARG);
  vacount = va_count(ap);
  m_d = 0;
  if(vacount == 1 || vacount == 2) { /* integral value for 1st arg */
    if(va_datatype(ap)  <= elementtype(int)) {
      m_d += va_arg(ap, int);
    }
    else {
      printf("Error: wrong data type\n");
    }
  }
  else {
      printf("Error: wrong number of arguments\n");
  }

  if(vacount == 2) { /* floating-point number for 2nd arg */
    if(va_datatype(ap)  == elementtype(float)) {
      m_d += va_arg(ap, float);
    }
    else if(va_datatype(ap)  == elementtype(double)) {
      m_d += va_arg(ap, double);
    }
    else {
      printf("Error: wrong data type\n");
    }
  }
  va_end(ap);
}
```

Program 19.5: Member functions with variable-length argument lists.

```
void C1::memfunc(...) {
  va_list ap;
  int vacount;
  int i, num = 0;

  printf("m_d = %f\n",  m_d);
  va_start(ap, VA_NOARG);
  vacount = va_count(ap);
  printf("vacount = %d\n", vacount);
  while(num++, vacount--) {
    i = va_arg(ap, int);
    printf("argument %d = %d, ", num, i);
  }
  printf("\n\n");
  va_end(ap);
  return;
}

int main() {
  class C1 c1 = C1(3);
  class C1 c2 = C1(3, 6.5);
  c1.memfunc(1);
  c2.memfunc(1, 2, 3);

  return 0;
}
```

Program 19.5: Member functions with variable-length argument lists (Contd.).

```
m_d = 3.000000
vacount = 1
argument 1 = 1,

m_d = 9.500000
vacount = 3
argument 1 = 1, argument 2 = 2, argument 3 = 3,
```

Figure 19.4: Output of Program 19.5.

19.10 Nested Classes

Nested classes are classes that are defined within the scope of another class. Classes in which the nested classes are defined are called *surrounding classes* or *enclosing classes*. Ch supports nested classes.

A class can be nested in every part of the surrounding class. A nested class is actually considered a member of the enclosing class. So, the normal access and visibility rules in classes apply to nested classes.

If a class is nested in the public section of a class, it is visible outside the surrounding class. If it is nested in the private section, it is only visible for the members of the surrounding class.

Although a nested class is considered a member of the enclosing class, its members are not members of the enclosing class. So, member functions of the surrounding class have no special access to members of a

nested class. On the other hand, member functions of a nested class follow regular access rules too and have no special access privileges to members of their enclosing classes.

The Program 19.6 shows how to define nested classes in the enclosing class `Encl`.

In this example, access to the members is defined as follows.

1. The public nested class `nestPub` is visible both outside and inside the enclosing class `Encl`.
2. The public member function `getVar()` of the class `nestPub` are also globally visible.
3. The private data member `variable` of the class `nestPub` is only accessible for the members of the class `nestPub`.
4. The private class `nestPrv` is visible only inside the surrounding class `Encl`.
5. The public members of the class `nestPrv` can be used by the members of the public nested class `nestPub`.
6. The public member function `getVar()` of the class `nestPrv` can only be accessed by the members of the enclosing class `Encl` and the members of its nested classes.
7. The private data member `variable` of the class `nestPrv()` is only visible for the members of the class `nestPrv`.

Besides the definition of the nested class, their member functions are also defined in Program 19.6.

The definitions of member functions of nested classes are similar to the definitions of the member functions of normal classes. The function name is preceded by both the surrounding class name and the nested class name. Two scope resolution operators '`::`' are used because both `nestPub` and `nestPrv` have member functions named `getVar()`. The scope resolution can prevent the confusion.

19.11 Classes inside Member Function

As an extension to C++, Ch provides classes inside member function. The classes which are defined in member functions of other classes are called classes inside member functions in Ch.

The Program 19.7 shows how to define the class `C2` in the member function `C1::func()`.

In this example, access to the members is defined as follows.

1. A class inside a member function is only visible inside the member function in which it is defined, regardless of whether the member function is public or private. In the example, declaration of a variable of the type class `C2` outside the member function `C1::func()` is a syntax error in Ch.
2. The member function `C1::func()` in which the class `C2` is nested has no special access privileges to members of `C2`.
3. The public members of `C2` is accessible within `C1::func()`.
4. The private members of `C2` can only be accessed by its own members.

19.12 Passing Member Functions to Arguments of Functions

Passing member functions to functions as arguments of pointer to function is another feature which is supported by Ch but not supported by C++.

In Program 19.8, member functions `C1::f5()` and `C2::f()`, and regular function `func()` take an argument which is a pointer to function.

```
/* Nested classes */

#include <iostream.h>

class Encl {
    public:
        Encl(int); /* constructor */
        int getVar();

        class nestPub {
            public:
                int getVar();
            private:
                int variable;
        };

    private:
        class nestPrv{
            public:
                int getVar();
            private:
                int variable;
        }nPr;
        int variable;
};

Encl::Encl(int var) {
    variable = var;
}

int Encl::getVar() {
    return variable;
}

int Encl::nestPub::getVar() {
    return variable;
}

int Encl::nestPrv::getVar() {
    return variable;
}

int main() {
   Encl e1 = Encl(5);
   cout << "variable = " << e1.getVar() << endl;

   return 0;
}
```

Program 19.6: Nested class.

```
/* Classes inside member functions */

#include <iostream.h>

int main () {
  int t;

  class C1 {
      int v1;
  public:
      int func();
  };

  int C1::func() {
      class C2 {
          int v2;
      public:
          int func2();
          int v3;
      };
      int C2::func2() {
          class C1 c;
          class C2 c2;
          v2 = 10;
          c.v1 = 20;
          c2.v2 = 30;
          return 10;
      }
      C2 c2;
      /* c2.v2 = 30; is wrong */
      c2.func2();
      c2.v3 = 50;

      v1 = 30;
      return v1;
  }

  C1 s;
  /* C2 s2; is wrong */
  cout << s.func() << endl;

  return 0;
}
```

Program 19.7: Classes inside member functions.

```
#include <stdio.h>
/* pass member function to a function */
/* normal function with argument of pointer to function */
int func(void (*fp)()) {
   printf("func() called\n");
   fp();
   return 0;
}

class C1 {
   int i;
   void f1(); // private member function access i
public:
   C1();
   void f2(); // access member i
   void f3(); // does not access any member
   void f4(); // call func()
   /* function with argument of pointer to function */
   void f5(void (*fp)());
};
C1::C1() {
   i = 5;
}
void C1::f1() { // private member function
   printf("C1::f1() called, i = %d\n", i);
}
void C1::f2() {
   printf("C1::f2() called, i = %d\n", i);
}
void C1::f3() {
   printf("C1::f3() called\n");
}
/* member function with argument of pointer to function */
void C1::f4() {
   func(f1); /* pass private function, ok in Ch and bad in C++ */
   func(f2); /* pass public function,  ok in Ch and bad in C++ */
}
/* member function with argument of pointer to function */
void C1::f5(void (*fp)()) {
   printf("C1::f5() called \n");
   fp(); /* function as argument */
}

class C2 {
   int d;
public:
   C2();
   /* function with argument of pointer to function */
   void f(void (*fp)());
};
C2::C2() {
   d = 10;
}
/* member function with argument of pointer to function */
void C2::f(void (*fp)()) {
   fp(); /* function as argument */
}
```

Program 19.8: Passing member functions to functions as arguments.

```
int main() {
   class C1 s;
   class C2 s2;

   printf("(1) passed member func to regular func\n");
   func(s.f2);  // OK in Ch, bad in C++
   func(s.f3);  // OK in Ch, bad in C++
   printf("(2) passed member func to regular func inside member func\n");
   s.f4();
   printf("(3) passed member func to member func of the same class \n");
   s.f5(s.f2);  // OK in Ch, bad in C++
   s.f5(s.f3);  // OK in Ch, bad in C++
   printf("(4) passed member func, without accessing member field,\n");
   printf("    to member func of a diff class.\n");
   s2.f(s.f3);  // Ok in Ch, bad in C++

   printf("\n(5) Error: passed member func, with accessing member field,\n");
   printf("    to member func of a diff class.\n");
   s2.f(s.f2);  // bad in Ch and C++
   return 0;
}
```

Program 19.8: Passing member functions to functions as arguments (Contd.).

The output from executing Program 19.8 is as follows.

```
(1) passed member func to regular func
func() called
C1::f2() called, i = 5
func() called
C1::f3() called
(2) passed member func to regular func inside member func
func() called
C1::f1() called, i = 5
func() called
C1::f2() called, i = 5
(3) passed member func to member func of the same class
C1::f5() called
C1::f2() called, i = 5
C1::f5() called
C1::f3() called
(4) passed member func, without accessing member field,
    to member func of a diff class.
C1::f3() called

(5) Error: passed member func, with accessing member field,
    to member func of a diff class.
C1::f2() called, i = 10
```

Passing member functions to arguments of functions in Ch follows the normal rules of accessing class members for private and public members. For example, the private member function can only be used as an argument of a function by a member of the class. However, there are some additional constraints on passing

member function as an argument of pointer to function as follow.

1. A member function can be passed as an argument to pointer to function of a regular function as shown in the following statements in Program 19.8.

   ```
   func(s.f2);  // OK in Ch, bad in C++
   func(s.f3);  // OK in Ch, bad in C++
   ```

   In this case, the passed member function, such as C1::f2(), can access its members.

2. A member function can be passed as an argument to pointer to function of a regular function inside a member function. In this case, the passed member function can access its members as shown by function call `s.f4()` in Program 19.8.

3. A member function can be passed as an argument to pointer to function of a member function of the same instance of a class. In this case, the passed member function can access its members as shown in the following statements in Program 19.8.

   ```
   s.f5(s.f2);  // OK in Ch, bad in C++
   s.f5(s.f3);  // OK in Ch, bad in C++
   ```

4. A member function can be passed as an argument to pointer to function of a member function of a different class. In this case, the passed member function cannot access its members as shown in the following statement in Program 19.8.

   ```
   s2.f(s.f3); // Ok in Ch, bad in C++
   ```

   The `s` and `s2` are instances of different classes. However, since member function `C1::f3()` does not access a member field, it can be passed as an argument to a member function of different class.

5. A member function, which access a member, cannot be passed as an argument to pointer to function of a member function of a different class as shown in the following statement in Program 19.8.

   ```
   s2.f(s.f2); // bad in Ch, bad in C++
   ```

   The `s` and `s2` are instances of different classes. Because member function `C1::f2()` access a member field, it cannot be passed as an argument to a member function of different class. The memory for `s` is bounded to the memory for `s2` in this case.

19.13 Predefined Identifiers __class__ and __class_func__

Like predefined identifier **__func__**, the predefined identifiers **__class__** and **__class_func__** can be used to obtain the class name and both class and function names within a member function. The identifiers **__class__** and **__class_func__** are implicitly declared by as if, immediately following the opening brace of each member function definition, the declaration

```
static const char __class__[] = "class-name";
static const char __class_func_[] = "class-name:function-name";
```

```
/* Filename: classname.ch */
#include <iostream>
class tag{
    public:
      tag(int i1); /* only one argument */
      ~tag();
      void func(void);
    private:
      int m_i;
};

tag::tag(int i) {
    cout << "__func__ in tag::tag() = " << __func__ << endl;
    cout << "__class_ in tag::tag() = " << __class__ << endl;
    cout << "__class_func__ in tag::tag() = " << __class_func__ << endl;
    m_i = i;
}

tag::~tag() {
    cout << "__func__ in tag::~tag() = " << __func__ << endl;
    cout << "__class_ in tag::~tag() = " << __class__ << endl;
    cout << "__class_func__ in tag::~tag() = " << __class_func__ << endl;
}

void tag::func(void) {
    cout << "__func__ in tag::func() = " << __func__ << endl;
    cout << "__class_ in tag::func() = " << __class__ << endl;
    cout << "__class_func__ in tag::func() = " << __class_func__ << endl;
}

int main() {
    struct tag s = tag(10);
    s.func();
    return 0;
}
```

Program 19.9: Using __class__ and __class_func__ to get class and function names.

appeared, where class and function-name is the class name and name of the lexically enclosing member function of a class. For example, Program 19.9 uses these predefined identifiers to print out the names of class and member function with the output as follows.

```
__func__ in tag::tag() = tag
__class_ in tag::tag() = tag
__class_func__ in tag::tag() = tag::tag
__func__ in tag::func() = func
__class_ in tag::func() = tag
__class_func__ in tag::func() = tag::func
__func__ in tag::~tag() = ~tag
__class_ in tag::~tag() = tag
__class_func__ in tag::~tag() = tag::~tag
```

Chapter 20

Input and Output

20.1 Streams

Input and output in Ch, whether to or from physical devices such as terminals and tape drives, or whether to or from files supported on structured storage devices, are mapped into logical data *streams*, whose properties are more uniform than their various inputs and outputs. Two forms of mapping are supported, mapping for *text streams* and for *binary streams*. A text stream is an ordered sequence of characters composed into *lines*, each line consisting of zero or more characters plus a terminating new-line character. A binary stream is an ordered sequence of characters that can transparently record internal data.

Each stream has an *orientation*. After a stream is associated with an external file, but before any operations are performed on it, the stream is without orientation. Once a wide-character input/output function has been applied to a stream without orientation, the stream becomes a *wide-oriented stream*. Similarly, once a byte input/output function has been applied to a stream without orientation, the stream becomes a *byte-oriented stream*. Only a call to function **freopen()** or function **fwide()** can otherwise alter the orientation of a stream. A successful call to **freopen()** removes any orientation. Byte input/output functions shall not be applied to a wide-oriented stream, and wide-character input/output functions shall not be applied to a byte-oriented stream. Each wide-oriented stream has an associated **mbstate_t** object that stores the current parse state of the stream. A successful call to **fgetpos()** stores a representation of the value of this **mbstate_t** object as part of the value of the **fpos_t** object. A later successful call to **fsetpos()** using the same stored **fpos_t** value restores the value of the associated **mbstate_t** object as well as the position within the controlled stream.

When a Ch program begins execution, there are three text streams predefined and opened. The stream **stdin** stands for standard input, **stdout** stands for standard output, and **stderr** stands for standard error. They are defined in header file **stdio.h**

20.2 Buffered and Unbuffered I/O

A stream can be *buffered* or *unbuffered* in Ch. When a stream is unbuffered, characters are intended to appear from the source or at the destination as soon as possible. Otherwise characters may be accumulated and transmitted to or from the host environment as a block.

Furthermore, a buffered stream can be *fully buffered* or *line buffered*. When a stream is fully buffered, characters are intended to be transmitted to or from the host environment as a block when a buffer is filled. When a stream is line buffered, characters are intended to be transmitted to or from the host environment as a block when a new-line character is encountered.

By default, output to a terminal is line buffered and all other input/output is fully buffered in Ch, but the stream **stderr** is unbuffered.

Functions **setbuf()** and **setvbuf()** can be used to assign buffering to a stream file. The prototypes of these two functions are shown below.

```
void setbuf(FILE *stream, char *buf);
int setvbuf(FILE *stream, char *buf, int type, size_t size);
```

Function **setbuf()** can be used after a stream has been opened but before it is read or written. It causes the buffer pointed to by `buf` to be used instead of an automatically allocated buffer. If `buf` is not **NULL**, input/output will be fully buffered. Otherwise it will be completely unbuffered. The size of the buffer is specified by the constant **BUFSIZ** which is defined in the **stdio.h** header file. In Ch, this value is platform dependent. For example,

```
char buf[BUFSIZ];
FILE *ptf = fopen("example", "a+");
if(ptf)
  setbuf(ptf, buf);  // set user-defined buffer
```

Function **setvbuf()** provides more flexibility to assign buffering to a stream file. Like function **setbuf()**, it may be used only after the stream pointed to by `stream` has been associated with an open file and before any other operation is performed on the stream. Argument `type` determines how `stream` will be buffered. Legal values for type defined in **stdio.h** are described in Table 20.1:

Table 20.1: Buffering type for function **setvbuf()**.

Type	Description
_IOFBF	causes input/output to be fully buffered
_IOLBF	causes input/output to be line buffered
_IONBF	causes input/output to be unbuffered

The size of buffer pointed to by `buf` is specified by `size` instead of **BUFSIZ**. The **setvbuf()** function returns zero on success, or nonzero if an invalid value is given for argument `type` or if the request cannot be satisfied. The expression

```
setbuf(stream, buf);
```

is equivalent to the conditional expression

```
((buf == NULL) ?
 (void) setvbuf(stream, NULL, _IONBF, 0) :
 (void) setvbuf(stream, buf, _IOFBF, BUFSIZ))
```

Function **fflush()** can also be used to explicitly flush a buffer of a file stream. A file may be disassociated from a controlling stream by closing the file. Output streams are flushed before the stream is disassociated from the file. The value of a pointer to a file object becomes **NULL** after the associated file is closed.

20.3 I/O Formats

In C, the input of integers and floating-point numbers is obtained through the standard input functions **scanf**(), **fscanf**(), etc.; the output is accomplished using the output functions **fprintf**(), **printf**(), etc. These functions are also available in Ch and are in full compliance with the C standard. However, implementation of some of these functions in Ch is different from traditional compiler-based C. In this section, the differences of these functions between Ch and C, and enhancements of these functions in Ch will be highlighted.

The major difference of these functions between Ch and C is that some of these functions are built-in internal functions in Ch whereas they are external functions in C. Therefore, they can be reconciled inside Ch so that they are more flexible and powerful. From an application point of view, a programmer will not notice differences in these functions between Ch and C.

20.3.1 Output Format for fprintf Family of Output Function

In this section, the format for the **fprintf()** family of output functions will be described in detail. We will highlight how Ch extends C in output format for **fprintf()**. The underlining principle can be applied to other output functions as well. The format of function **fprintf**() in Ch is as follows

```
int fprintf(FILE *stream, char *format, arg1, arg2, ...);
```

The function **fprintf**() prints output to the stream pointed to by argument `stream` under the control of the string pointed to by `format`, and returns the number of characters printed. If the format string contains two types of objects, ordinary characters and conversion specifications beginning with the character '%' and ending with a conversion character, the following C rules for family of function **fprintf**() will be used. After the %, the following appear in sequence:

– Zero or more *flags* (in any order) that modify the meaning of the conversion specification.

– An optional minimum *field width*. If the converted value has fewer characters than the field width, it is padded with spaces (by default) on the left (or right, if the left adjustment flag, described later, has been given) to the field width. The field width takes the form of an asterisk (described later) or a decimal integer.

– An optional precision that gives the minimum number of digits to appear for the **d, i, o, u, x**, and **X** conversions, the number of digits to appear after the decimal-point character for **a, A, e, E, f**, and **F** conversions, the maximum number of significant digits for the **g** and **G** conversions, or the maximum number of characters to be written from a string in **s** conversions. The precision takes the form of a period (.) followed either by an asterisk (described later) or by an optional decimal integer; if only the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined.

– An optional length modifier that specifies the size of the argument.

– A conversion specifier character that specifies the type of conversion to be applied.

As noted above, a field width and/or precision, may be indicated by an asterisk. In this case, an **int** argument supplies the field width or precision. The arguments specifying field width and/or precision, shall appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a - flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted.

The flag characters and their meanings are:

- The result of the conversion is left-justified within the field. (It is right-justified if this flag is not specified).

+ The result of a signed conversion always begins with a plus or minus sign. (It begins with a sign only when a negative value is converted if this flag is not specified).

space If the first character of a signed conversion is not a sign. Or, if a signed conversion results in no characters, a space is prefixed to the result. If the *space* and + flags both appear, the *space* flag is ignored.

The result is converted to an "alternative form". For **o** conversion, it increases the precision, if and only if necessary, to force the first digit of the result to be a zero (if the value and precision are both 0, a single 0 is printed). For **x** (or **X**) conversion, a nonzero result has **0x** (or **0X**) prefixed to it. For **a, A, e, E, f, F, g**, and **G** conversions, the result of converting a floating-point number always contains a decimal-point character, even if no digits follow it. (Normally, a decimal-point character appears in the result of these conversions only if a digit follows it.) For g and G conversions, trailing zeros are not removed from the result. For other conversions, the behavior is undefined.

0 For **d, i, o, u, x, X, a, A, e, E, f, F, g**, and **G** conversions, leading zeros (following any indication of sign or base) are used to pad the field width rather than perform space padding, except when converting an infinity or NaN. If the **0** and - flags both appear, the **0** flag is ignored. For **d, i, o, u**, **x**, and **X** conversions, if a precision is specified, the **0** flag is ignored. For other conversions, the behavior is undefined.

The length modifiers and their meanings are:

hh Specifies that a following **d, i, o, u, x**, or **X** conversion specifier applies to a **signed char** or **unsigned char** argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to **signed char** or **unsigned char** before printing), or that a following **n** conversion specifier applies to a pointer to a **signed char** argument.

h Specifies that a following **d, i, o, u, x**, or **X** conversion specifier applies to a **short int** or **unsigned short int** argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to **short int** or **unsigned short int** before printing), or that a following **n** conversion specifier applies to a pointer to a **short int** argument.

l Specifies that a following **d, i, o, u, x**, or **X** conversion specifier applies to a **long int** or **unsigned long int** argument; that a following **n** conversion specifier applies to a pointer to a **long int** argument; that a following **c** conversion specifier applies to a **wint_t** argument; that a following **s** conversion specifier applies to a pointer to a **wchar_t** argument; or has no effect on a following **a, A, e, E, f, F, g**, or **G** conversion specifier.

ll Specifies that a following **d, i, o, u, x**, or **X** conversion specifier applies to a **long long int** or **unsigned long long int** argument; or that a following **n** conversion specifier applies to a pointer to a **long long int** argument.

j Specifies that a following **d, i, o, u, x**, or **X** conversion specifier applies to an **intmax_t** or **uintmax_t** argument; or that a following **n** conversion specifier applies to a pointer to an **intmax_t** argument.

z Specifies that a following **d, i, o, u, x**, or **X** conversion specifier applies to a **size_t** or the corresponding signed integer type argument; or that a following n conversion specifier applies to a pointer to a signed integer type corresponding to **size_t** argument.

t Specifies that a following **d, i, o, u, x**, or **X** conversion specifier applies to a **ptrdiff_t** or the corresponding unsigned integer type argument; or that a following **n** conversion specifier applies to a pointer to a **ptrdiff_t** argument.

L Specifies that a following **a, A, e, E, f, F, g**, or **G** conversion specifier applies to a **long double** argument.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

The conversion specifiers and their meanings are:

d,i The **int** argument is converted to signed decimal in the style *[-]dddd*. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

o,u,x,X The **unsigned int** argument is converted to unsigned octal (**o**), unsigned decimal (**u**), or unsigned hexadecimal notation (**x** or **X)** in the style *dddd*; the letters **abcdef** are used for **x** conversion and the letters **ABCDEF** for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

f,F A **double** argument representing a floating-point number is converted to decimal notation in the style *[-]ddd.ddd*, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is zero and the # flag is not specified, no decimal-point character appears. If a decimal-point character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.
A **double** argument representing an infinity is converted in one of the styles *[-]***inf** or *[-]***infinity** – which style is implementation-defined. A **double** argument representing a NaN is converted in one of the styles *[-]***nan** or *[-]***nan** *(n-char-sequence)* – which style, and the meaning of any n-char- sequence, is implementation-defined. The **F** conversion specifier produces **INF, INFINITY**, or **NAN** instead of **inf, infinity**, or **nan**, respectively.

e,E A double argument representing a floating-point number is converted in the style *[-]d.ddd* **e**±*dd*, where there is one digit (which is nonzero if the argument is nonzero) before the decimal-point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero and the # flag is not specified, no decimal-point character appears. The value is rounded to the appropriate number of digits. The **E** conversion specifier produces a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits, and only as many more digits as necessary to represent the exponent. If the value is zero, the exponent is zero.
A **double** argument representing an infinity or NaN is converted in the style of an **f** or **F** conversion specifier.

g,G A **double** argument representing a floating-point number is converted in style **f** or **e** (or in style **F** or **E** in the case of a **G** conversion specifier), with the precision specifying the number of significant digits. If the precision is zero, it is taken as 1. The style used depends on the value converted; style **e**

(or **E**) is used only if the exponent resulting from such a conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion of the result unless the # flag is specified; a decimal-point character appears only if it is followed by a digit.
A double argument representing an infinity or NaN is converted in the style of an **f** or **F** conversion specifier.

a,A A **double** argument representing a floating-point number is converted in the style *[-]***0x***h.hhhh***p**$\pm d$, where there is one hexadecimal digit (which is nonzero if the argument is a normalized floating-point number and is otherwise unspecified) before the decimal-point character and the number of hexadecimal digits after it is equal to the precision; if the precision is missing and **FLT_RADIX** is a power of 2, then the precision is sufficient for an exact representation of the value; if the precision is missing and **FLT_RADIX** is not a power of 2, then the precision is sufficient to distinguish values of type **double**, except that trailing zeros may be omitted; if the precision is zero and the # flag is not specified, no decimal- point character appears. The letters **abcdef** are used for **a** conversion and the letters **ABCDEF** for **A** conversion. The **A** conversion specifier produces a number with **X** and **P** instead of **x** and **p**. The exponent always contains at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2. If the value is zero, the exponent is zero.
A **double** argument representing an infinity or NaN is converted in the style of an **f** or **F** conversion specifier.

c If no **l** length modifier is present, the **int** argument is converted to an **unsigned char**, and the resulting character is written.
If an **l** length modifier is present, the **wint_t** argument is converted as if by an ls conversion specification with no precision and an argument that points to the initial element of a two-element array of **wchar_t**, the first element containing the **wint_t** argument to the **lc** conversion specification and the second a null wide character.

s If no **l** length modifier is present, the argument shall be a pointer to the initial element of an array of character type. Characters from the array are written up to (but not including) the terminating null character. If the precision is specified, no more than that many characters are written. If the precision is not specified or is greater than the size of the array, the array shall contain a null character.
If an **l** length modifier is present, the argument shall be a pointer to the initial element of an array of **wchar_t** type. Wide characters from the array are converted to multibyte characters (each as if by a call to the **wcrtomb** function, with the conversion state described by an **mbstate_t** object initialized to zero before the first wide character is converted) up to and including a terminating null wide character. The resulting multibyte characters are written up to (but not including) the terminating null character (byte). If no precision is specified, the array shall contain a null wide character. If a precision is specified, no more than that many characters (bytes) are written (including shift sequences, if any). Also, if the function needs to access a wide character one past the end of the array, the array shall contain a null wide character to equal the multibyte character sequence length given by the precision. In no case is a partial multibyte character written.

p The argument shall be a pointer to **void**. The value of the pointer is converted to a sequence of printing characters, in an implementation-defined manner.

n The argument shall be a pointer to signed integer into which is *written* the number of characters written to the output stream so far by this call to **printf**. No argument is converted, but one is consumed. If the conversion specification includes any flags, a field width, or a precision, the behavior is undefined.

% A % character is written. No argument is converted. The complete conversion specification shall be %%.

If a conversion specification is invalid, the behavior is undefined. If any argument is not the correct type for the corresponding conversion specification, the behavior is undefined. In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result. For **a** and **A** conversions, if **FLT_RADIX** is a power of 2, the value is correctly rounded to a hexadecimal floating number with the given precision. If **FLT_RADIX** is not a power of 2, the result should be one of the two adjacent numbers in hexadecimal floating style with the given precision, with the extra stipulation that the error should have a correct sign for the current rounding direction. For **e, E, f, F, g**, and **G** conversions, if the number of significant decimal digits is at most **DECIMAL_DIG**, then the result should be correctly rounded. If the number of significant decimal digits is more than **DECIMAL_DIG** but the source value is exactly representable with **DECIMAL_DIG** digits, then the result should be an exact representation with trailing zeros. Otherwise, the source value is bounded by two adjacent decimal strings $L < U$, both having **DECIMAL_DIG** significant digits; the value of the resultant decimal string D should satisfy $L \leq D \leq U$, with the extra stipulation that the error should have a correct sign for the current rounding direction.

Following commands are examples using different formats for function **printf**.

```
> double d = 123.45678
> float f = 123.45678
> char *str = "123456789"
> printf("d = %1.3f", d)
d = 123.457
> printf("f = %5.10f", f)
f = 123.4567794800
> printf("%-15s", str)
123456789
> printf("%15s", str)
      123456789
```

Besides the control characters specified by the C standard, Ch has one more conversion character 'b' that is used to print real numbers in binary format. An integer number between the symbol % and the character 'b' specifies how many bits starting with bit 0 will be printed. If without an integer number between the symbol % and the character 'b', the default format will print int data without leading zeros, float data in 32 bits, and double data in 64 bits. This binary format is very convenient to examine the bit patterns of metanumbers. For example,

```
> int i = 5
> float f = 1.234
> printf("binary of i = %b, f = %b", i, f)
binary of i = 101, f = 00111111100111011111001110110110
```

In Ch, if the format string in **printf**() is absent or contains only ordinary characters, the subsequent numerical constants or variables will be printed according to preset default formats. The default format for int, float, and double are %d, %.2f, and %.4lf, respectively. For example,

```
> float f = 1.234
> printf(f)
1.23
```

Default formats for the family of the **fprintf()** function will be described in detail in Section 20.4.

20.3.2 Input Format for fscanf Family of Input Function

In this section, the format for the **fscanf()** family of input functions **scanf(), fscanf(), sscanf()** will be described in detail. The extension to C for input format for **fscanf()** in Ch will be highlighted. The format of function **fscanf**() in Ch is as follows

```
int fscanf(FILE *stream, char *format, arg1, arg2, ...);
```

The function **fscanf**() reads input from the stream pointed to by argument `stream` under the control of the string pointed to by `format` and returns the number of the input items on success. The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: one or more white- space characters, an ordinary multibyte character (neither % nor a white-space character), or a conversion specification. Each conversion specification is introduced by the character %.
After the %, the following appear in sequence:

– An optional assignment-suppressing character *****.

– An optional nonzero decimal integer that specifies the maximum field width (in characters).

– An optional *length modifier* that specifies the size of the receiving object.

– A *conversion specifier* character that specifies the type of conversion to be applied.

The **fscanf** function executes each directive of the format in turn. If a directive fails, as detailed below, the function returns. Failures are described as input failures (due to the occurrence of an encoding error or the unavailability of input characters), or matching failures (due to inappropriate input).

A directive composed of white-space character(s) is executed by reading input up to the first non-white-space character (which remains unread), or until no more characters can be read.

A directive that is an ordinary multibyte character is executed by reading the next characters of the stream. If any of those characters differ from the ones composing the directive, the directive fails and the differing and subsequent characters remain unread.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each specifier. A conversion specification is executed in the following steps:

Input white-space characters (as specified by the **isspace** function) are skipped, unless the specification includes a **[, c**, or **n** specifier.

An input item is read from the stream, unless the specification includes an **n** specifier. An input item is defined as the longest sequence of input characters which does not exceed any specified field width and which is, or is a prefix of, a matching input sequence. The first character, if any, after the input item remains unread. If the length of the input item is zero, the execution of the directive fails; this condition is a matching failure unless end-of-file, an encoding error, or a read error prevented input from the stream, in which case it is an input failure.

Except in the case of a % specifier, the input item (or, in the case of a %**n** directive, the count of input characters) is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails: this condition is a matching failure. Unless assignment suppression was indicated by a *****, the result of the conversion is placed in the object pointed to by the first argument following the *format* argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the object, the behavior is undefined.

The length modifiers and their meanings are:

hh Specifies that a following **d, i, o, u, x, X**, or **n** conversion specifier applies to an argument with type pointer to **signed char** or **unsigned char**.

h Specifies that a following **d, i, o, u, x, X**, or **n** conversion specifier applies to an argument with type pointer to **short int** or **unsigned short int**.

l Specifies that a following **d, i, o, u, x, X**, or **n** conversion specifier applies to an argument with type pointer to **long int** or **unsigned long** int; that a following **a, A, e, E, f, F, g**, or **G** conversion specifier applies to an argument with type pointer to **double**; or that a following **c, s**, or **[** conversion specifier applies to an argument with type pointer to **wchar_t**.

ll Specifies that a following **d, i, o, u, x, X**, or **n** conversion specifier applies to an argument with type pointer to **long long int** or **unsigned long long int**.

j Specifies that a following **d, i, o, u, x, X**, or **n** conversion specifier applies to an argument with type pointer to **intmax_t** or **uintmax_t**.

z Specifies that a following **d, i, o, u, x, X**, or **n** conversion specifier applies to an argument with type pointer to **size_t** or the corresponding signed integer type.

t Specifies that a following **d, i, o, u, x, X**, or **n** conversion specifier applies to an argument with type pointer to **ptrdiff_t** or the corresponding unsigned integer type.

L Specifies that a following **a, A, e, E, f, F, g**, or **G** converson specifier applies to an argument with type pointer to **long double**.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

The conversion specifiers and their meanings are:

d Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **strtol** function with the value 10 for the **base** argument. The corresponding argument shall be a pointer to signed integer.

i Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the **strtol** function with the value 0 for the **base** argument. The corresponding argument shall be a pointer to signed integer.

o Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the **strtoul** function with the value 8 for the **base** argument. The corresponding argument shall be a pointer to unsigned integer.

u Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **strtoul** function with the value 10 for the **base** argument. The corresponding argument shall be a pointer to unsigned integer.

x Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the **strtoul** function with the value 16 for the **base** argument. The corresponding argument shall be a pointer to unsigned integer.

a,e,f,g Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected for the subject sequence of the **strtod** function. The corresponding argument shall be a pointer to floating.

c Matches a sequence of characters of exactly the number specified by the field width (1 if no field width is present in the directive).
If no **l** length modifier is present, the corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence. No null character is added.
If an **l** length modifier is present, the input shall be a sequence of multibyte characters that begins in the initial shift state. Each multibyte character in the sequence is converted to a wide character as if by a call to the **mbrtowc** function, with the conversion state described by an **mbstate_t** object initialized to zero before the first multibyte character is converted. The corresponding argument shall be a pointer to the initial element of an array of **wchar_t** large enough to accept the resulting sequence of wide characters. No null wide character is added.

s Matches a sequence of non-white-space characters.
If no **l** length modifier is present, the corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically.
If an **l** length modifier is present, the input shall be a sequence of multibyte characters that begins in the initial shift state. Each multibyte character is converted to a wide character as if by a call to the **mbrtowc** function, with the conversion state described by an **mbstate_t** object initialized to zero before the first multibyte character is converted. The corresponding argument shall be a pointer to the initial element of an array of **wchar_t** large enough to accept the sequence and the terminating null wide character, which will be added automatically.

[Matches a nonempty sequence of characters from a set of expected characters (the scanset).
If no **l** length modifier is present, the corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically.
If an **l** length modifier is present, the input shall be a sequence of multibyte characters that begins in the initial shift state. Each multibyte character is converted to a wide character as if by a call to the **mbrtowc** function, with the conversion state described by an **mbstate_t** object initialized to zero before the first multibyte character is converted. The corresponding argument shall be a pointer to the initial element of an array of **wchar_t** large enough to accept the sequence and the terminating null wide character, which will be added automatically.
The conversion specifier includes all subsequent characters in the **format** string, up to and including the matching right bracket (**]**). The characters between the brackets (the *scanlist*) compose the scanset,

unless the character after the left bracket is a circumflex (^), in which case the scanset contains all characters that do not appear in the scanlist between the circumflex and the right bracket. If the conversion specifier begins with **[]** or **[^]**, the right bracket character is in the scanlist and the next following right bracket character is the matching right bracket that ends the specification; otherwise, the first following right bracket character is the one that ends the specification. If a - character is in the scanlist and is not the first, nor the second where the first character is a ^, nor the last character, the behavior is implementation-defined.

p Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the **%p** conversion of the **fprintf** function. The corresponding argument shall be a pointer to a pointer to **void**. The input item is converted to a pointer value in an implementation-defined manner. If the input item is a value converted earlier during the same program execution, the pointer that results shall be equal to that value; otherwise, the behavior of the **%p** conversion is undefined.

n No input is consumed. The corresponding argument shall be a pointer to signed integer into which is to be written the number of characters read from the input stream so far by this call to the **fscanf** function. Execution of a **%n** directive does not increment the assignment count returned at the completion of executing of the **fscanf** function. No argument is converted, but one is consumed. If the conversion specification includes an assignment- suppressing character or a field width, the behavior is undefined.

% Matches a single % character; no conversion or assignment occurs. The complete conversion specification shall be %%.

If a conversion specification is invalid, the behavior is undefined. The conversion specifiers **A, E, F, G**, and **X** are also valid and behave the same as, respectively, **a, e, f, g**, and **x**. If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any characters matching the current directive have been read (other than leading white space, where permitted), execution of the current directive terminates with an input failure; otherwise, unless execution of the current directive is terminated with a matching failure, execution of the following directive (other than **%n**, if any) is terminated with an input failure. Trailing white space (including new-line characters) is left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable other than via the **%n** directive. If conversion terminates on a conflicting input character, the offending input character is left unread in the input stream. The following commands are examples of using different formats for function **scanf()**.

```
> int i
> float x
> char name[50]
> scanf("%2d%f%*d %[01234567890]", &i, &x, name)
56789 0123 34a72
> i
56
> x
789.00
> name
34
```

In Ch, If the the format string in **fscanf**() function family is absent, preset default formats will be used, such as %d for int, %f for float, and %lf for double. For example,

```
> float f;
> scanf(&f); // <==> scanf("%f", &f);
10
> f
10.00
```

Default formats for **fscanf()** will be discussed in detail in Section 20.4.

20.4 Default I/O Formats

20.4.1 Default Format for fprintf Family of Output Functions

In Ch, when the format for an output function is missing, the default format will be used. The default output format for different data types for functions **printf()**, **fprintf()**, **vprintf()**, **vfprintf()**, and **vsprintf()** are listed in Table 20.2.

Table 20.2: Default format for **fprintf** family of output functions.

Data Type	Format
char	"%c"
unsigned char	"%c"
short	"%hd"
unsigned short	"%hu"
int	"%d"
unsigned int	"%u"
long	"%ld"
unsigned long	"%lu"
long long	"%lld"
unsigned long long	"%llu"
float	"%.2f"
double	"%.4lf"
char *	"%s"
unsigned char *	"%s"
string_t	"%s"
pointer type	"%p"

For example,

```
> int i = 8
> float f = 8.0
> double d = 8.0
> printf(i) /* <==> printf("%d", i); */
8
> printf(f) /* <==> printf("%.2f", f); */
8.00
> printf(d) /* <==> printf("%.4f", d); */
8.0000
```

```
> f
8.00
> d
8.0000
```

The default format for pointer to char and pointer to unsigend char is `%s`, which displays the output as a string. To display the address to which such a pointer points, the value can be cast to a pointer to void first before it is printed out as illustrated below.

```
> char *p = "abc"
> p
abc
> (void*)p
004a3418
```

20.4.2 Default Format for fscanf Family of Input Functions

Similarly, there are default formats for input functions when the format for an input argument is missing. The default input format for different data types for functions **scanf()**, **fscanf()**, and **sscanf()** are listed in Table 20.3.

Table 20.3: Default format for **fscanf** family of input functions.

Data Type	Format
char	`"%c"`
unsigned char	`"%c"`
short	`"%hd"`
unsigned short	`"%hu"`
int	`"%d"`
unsigned int	`"%u"`
long	`"%ld"`
unsigned long	`"%lu"`
long long	`"%lld"`
unsigned long long	`"%llu"`
float	`"%f"`
double	`"%lf"`
char *	`"%s"`
unsigned char *	`"%s"`
string_t	`"%s"`
pointer type	`"%p"`

For example,

```
> int i;
> float f;
> scanf(&i); /* <==> scanf("%d", &i); */
10
> i
```

```
10
> scanf(&f); /* <==> scanf("%f", &f); */
10
> f
10.00
```

20.4.3 I/O Using cout, cin, cerr, and endl

Ch provides three C++ style streams for input and output of programs. They are the standard input stream **cin**, which is normally connected to the keyboard; the standard output stream **cout**, which is normally connected to the computer screen; and the standard error stream **cerr**, which is normally connected to the screen. All these three streams can be assigned to other devices rather than the default devices.

Furthermore, Ch provides stream insertion operator "<<" which performs output to stream **cout** , stream extraction operator ">>" which performs input from stream **cin**, and stream manipulator **endl** which issues a newline character and flushes the output buffer. Operators "<<" and ">>" are associate from left to right, and can be used in a cascaded form. For example,

```
> int i, j
> cin >> i          // <==> scanf(&i)
10
> cout << i         // <==> printf(i)
10
> cerr << i
10
> cin >> i >> j     // <==> scanf(&i, &j)
20 30
> cout << i << j    // <==> printf(i, j)
2030
```

Note that streams **cout**, **cin** and **cerr** are objects of classes **istream** and **ostream** in C++. In Ch, they are shortcuts for functions **printf()**, **scanf()** and **fprintf(stderr,)**, respectively. The default input/output formats described in section 20.4 are used. They are defined as aliases in the system-wide startup file **chrc**, and defined as macros in header file **iostream.h**. Therefore, to use streams **cin**, **cout**, and **cerr** in a Ch program, it is necessary to include header file **iostream.h** as shown below.

```
#include <iostream.h>  // for cout/cin/cerr/endl
int main() {
  int i;

  cout << "Type a number :" << endl;
  cin >> i;
  cout << "The input number is " << i << endl;
}
```

The stream manipulator **endl** is valid in both program and command modes.

To be compatible with the new C++ standard, the **using** directive for namespace in I/O stream is supported in the following format.

```
using std::cout;
using std::cin;
```

```
using std::cerr;
using std::endl;
using std::ends;
```

or

```
using namespace std;
```

For example, the above program can be modified to conform to the C++ standard as follows.

```
#include <iostream.h>  // for cout/cin/cerr/endl
using std::cout;
using std::cin;
using std::endl;
/* or using namespace std; */
int main() {
  int i;

  cout << "Type a number :" << endl;
  cin >> i;
  cout << "The input number is " << i << endl;
}
```

20.5 I/O for Metanumbers

The metanumbers Inf and NaN are treated as regular numbers in I/O functions. The default data types for these numbers are float. The following Ch program illustrates how b-format and metanumbers are handled by the I/O functions **printf**() and **scanf**().

```
float fInf, fNaN;
double dInf, dNaN;
printf("Please type 'Inf NaN Inf NaN' \n");
scanf(&fInf, &fNaN, &dInf, &dNaN);
printf("The float   Inf = %f\n", fInf);
printf("The float  -Inf = ", -fInf, "\n");
printf("The float   NaN = %f\n", fNaN);
printf("The float   Inf = %b \n", fInf);
printf("The float  -Inf = %b \n", -fInf);
printf("The float   NaN = %b \n", fNaN);
printf("The double  Inf = %lf\n", dInf);
printf("The double -Inf = ", -dInf, "\n");
printf("The double  NaN = %lf\n", dNaN);
printf("The double  Inf = %b \n", dInf);
printf("The double -Inf = %b \n", -dInf);
printf("The double  NaN = %b \n", dNaN);
printf("The int     2   = %b \n",  2);
printf("The int     2   = %32b \n",  2);
printf("The int    -2   = %b \n", -2);
printf("The float   0.0 = %b \n", 0.0);
```

```
printf("The float  -0.0 = %b \n", -0.0);
printf("The float   1.0 = %b \n",  1.0);
printf("The float  -1.0 = %b \n", -1.0);
printf("The float   2.0 = %b \n",  2.0);
printf("The float  -2.0 = %b \n", -2.0);
```

The first two lines of the program declare two float variables fInf and fNaN, and two double variables dInf and dNaN. The function **scanf**() will get Inf and NaN for the declared variables from the standard input device, which is the terminal keyboard in this example. These metanumbers will be printed in default formats %.2f for float and %0.4lf for double. These numbers are also printed using the binary format %b. For comparison, the memory storage for integers of ± 2, and floats of $\pm 0.0, \pm 1.0, \pm 2.0$ are printed. The result of the interactive execution of the above program is shown as follows

```
Please type 'Inf NaN Inf NaN'
```

Inf NaN Inf NaN

```
The float   Inf = Inf
The float  -Inf = -Inf
The float   NaN = NaN
The float   Inf = 01111111100000000000000000000000
The float  -Inf = 11111111100000000000000000000000
The float   NaN = 01111111111111111111111111111111
The double  Inf = Inf
The double -Inf = -Inf
The double  NaN = NaN
The double  Inf = 01111111111100000000000000000000\
                  00000000000000000000000000000000
The double -Inf = 11111111111100000000000000000000\
                  00000000000000000000000000000000
The double  NaN = 01111111111111111111111111111111\
                  11111111111111111111111111111111
The int     2   = 10
The int     2   = 00000000000000000000000000000010
The int    -2   = 11111111111111111111111111111110
The float   0.0 = 00000000000000000000000000000000
The float  -0.0 = 10000000000000000000000000000000
The float   1.0 = 00111111100000000000000000000000
The float  -1.0 = 10111111100000000000000000000000
The float   2.0 = 01000000000000000000000000000000
The float  -2.0 = 11000000000000000000000000000000
```

where the second line in italic is the input and the rest are the output of the program. As one can see that, for metanumbers Inf, $-$Inf, and NaN, there is no difference between float and double types from the user's point of view.

It can be easily verified that the bit-mappings of all these numbers in memory match with data representations discussed in Section 6.1.

20.6 I/O Formats for Aggregate Data Types

The input of aggregate data types such as complex numbers, computational arrays, structures, classes, and unions shall be handled element by element in Ch. For example,

```
> array int  A[2]
> scanf("%d", &A[0])
10
> A
10 0
>
```

Ch can use the family of output functions **fprintf()**, **printf()**, etc. to print out all elements of variables and constants of aggregate date types once. For complex numbers and computational arrays, the output format specifier will be applied to each element. For structures, classes, and unions, the default output format is used for each member. For example,

```
> printf("%.2f", complex(1.0, 2.0))
complex(1.00,2.00)
> array int  A[2][3], B[2][2]  // array
> A[0][0] = 1; B[1][1] = 6
1
> printf("A = \n%d\nB = \n%d\n", A, B);
A =
1 0 0
0 0 0

B =
0 0
0 6

> struct tag1{int i; float f;} s  //struct
> s.i = 10
10
> printf(s)
.i = 10
.f = 0.00
>
```

20.7 Verbatim Output Blocks Using fprintf

A block of the verbatim output can be achieved using the feature of function **fprintf**. The syntax for a block of verbatim output is

`fprintf` *stream* << *TERMINATOR*
`...`
TERMINATOR

or

`fprintf` *stream* << "*TERMINATOR*"
`...`

TERMINATOR

where *stream* is a valid file stream and terminator *TERMINATOR* is a valid identifier that have not been used as a keyword or variable name in the program. Macro names, such as "END", can be used as the terminator, since they are processed verbatim without macro expansion in this case. It is recommended that an identifier of all capital letters is used. The verbatim block output using **fprintf** has the following constraints.

- White spaces and comments can follow the first terminator.
- White spaces can precede the second terminator.
- The second terminator shall be terminated with a new line character. No character, even a white space, is allowed to appear after the second terminator.
- All characters, including white characters and comments, between the first and last lines are processed verbatim.
- The first terminator can be enclosed in double-quotes, whereas the second shall not. If the first terminator is enclosed in double-quotes, the dollar sign '$' within the enclosing block will be treated verbatim. Otherwise, the single dollar sign '$' is used for variable or expression substitution. Two syntaxes of

  $*var* and ${*var*}

  can be used for variable substitution. The variable name or symbol to be expanded may be enclosed in braces, which are optional but serve to protect the variable to be expanded from characters immediately following it which could be interpreted as part of the name.

  The variable in a variable substitution could be a predefined identifier; a user-defined variable of string, pointer to char, integral, floating-point, or complex data type; an environment variable; or undefined symbol. For a variable substitution, the Ch shell will first search the Ch name space for the variable name according to its scope rule. If the variable is not defined, then it searches the environment variables of the current process. If no variable with the specified name is found either in Ch space or environment space, no substitution will take place and the variable is ignored.
- Expression substitution in the form of

  ```
  $(expression)
  ```

  can be used to substitute the valid Ch expression with its result. *The expression* shall be an expression of string, pointer to char, integral, floating-point, or complex data type.
- The variable or expression substitution can be prevented by preceding the '$' with a '\'. A '$' is passed unchanged if followed by a blank, tab, or end-of-line.
- A value through variable substitution or expression substitution will be printed out using a default format control string for its data type.

For example, if the program `verbat.ch` consists of the following programming statements,

```
#include <stdio.h>
int sum = 2
fprintf stdout << END /* This is a comment */
  /* this is verbatim output */
```

```
  sum = \$$sum
  sum + 1 = \$$(sum+1)
END
```

The result from executing `verb.ch` is shown as follows.

```
> verbat.ch
    /* this is verbatim output */
    sum = $2
    sum + 1 = $3
>
```

In command

```
sum = \$$sum
```

the escape character '\' is used to print out as a single dollar sign, and the symbol `$sum` is substituted with the value of `sum`, i.e. 2. In the next command

```
sum + 1 = \$$(sum+1)
```

the symbol `$(sum+1)` indicates an expression substitution. It is replaced by the result of the expression `sum+1`, i.e. 3. The comment following the first terminator `END` and the white spaces preceding the second `END` are ignored. But, the comment inside the block is printed out verbatim.

By default, a variable of double type is printed out with four digits after the decimal point whereas a variable of float type is printed out with two digits after the decimal point. To print out a variable of double, one may cast it to float before printing it out if the value is within the representable range of float type. For example, $((float)d) can be printed out with two digits after decimal point, $((int)d) with integral part only.

Often time, a block of HTML code needs to be sent as a standard output stream in a CGI program. For example, Program 20.1 will generate the code below,

```
Content-Type: text/html

<HTML>
<HEAD>
<Title> Hello, world </Title>
</Head><BODY>
<h4> Hello, world </h4>
</BODY>
</HTML>
```

which displays the text `Hello, world` in a web browser. According to the HTTP protocal, the line

```
Content-Type: text/html
```

must start without any white space, and there must be only an empty line without white space following it. Using the verbatim output feature, the above Ch CGI program can be simplied as Program 20.2. Note that the value of `hello` is retrieved by using the dollar sign $ inside the verbatim output block,

```
/* File: genereatehtml.c */
#include <stdio.h>
int main() {
    char hello[] = "Hello, world";

    printf("Content-Type: text/html\n\n");
    printf("<HTML>\n");
    printf("<HEAD>\n");
    printf("<Title> Hello, world </Title>\n");
    printf("</Head>");
    printf("<BODY>\n");
    printf("<h4> %s </h4>\n", hello);
    printf("</BODY>\n");
    printf("</HTML>\n");
    return 0;
}
```

Program 20.1: Generating an html file.

```
#!/bin/ch
/* File: genereatehtml.ch */
#include <stdio.h>
int main() {
    char hello[] = "Hello, world";

    printf("Content-Type: text/html\n\n");
    fprintf stdout << ENDPRINT
        <HTML>
        <HEAD>
        <Title> Hello, world </Title>
        </Head>
        <BODY>
        <h4> $hello </h4>
        </BODY>
        </HTML>
    ENDPRINT
    return 0;
}
```

Program 20.2: Using fprintf for a block output.

As another example, the function `sendApplet()` below generates a C program.

```
void sendApplet(char *x, char *y, char *expr) {
    fprintf(stdout, "#include<stdio.h>\n");
    fprintf(stdout, "int main() {\n");
    fprintf(stdout, "  double x = %s;\n", x);
    fprintf(stdout, "  double y = %s;\n", y);
    fprintf(stdout, "  printf(\"x = %%f, \", x);\n");
    fprintf(stdout, "  printf(\"y = %%f \\n\", y);\n");
    fprintf(stdout, "  printf(\"%s = %%f\\n\", %s);\n", expr, expr);
    fprintf(stdout, "}\n");
}
```

This function `sendApplet()` can be rewritten in Ch as follows.

```
void sendApplet(char *x, char *y, char *expr) {
    fprintf stdout << ENDFILE
      #include<stdio.h>
      int main() {
        double x = $x;
        double y = $y;
        printf("x = %f", x);
        printf("y = %f\n", y);
        printf("$expr = %f\n", $expr);
      }
    ENDFILE
}
```

where the values of variables x, y and $expr$ are obtained using operator $.

20.8 File Manipulation

20.8.1 Opening and Closing a File

File is the most common I/O facility which can be used as a stream in Ch. Data type **FILE** which is defined in header file **stdio.h** maintains information about the stream. An object of type **FILE ***, created by calling some functions such as **fopen()**, is used to access the file by other file manipulation functions such as **fscanf()**. Function **fopen()** is a common function to open a file. Its prototype is

```
FILE *fopen(const char *filename, const char *mode);
```

It returns a pointer to the object controlling the stream on success. If the open operation fails, **NULL** is returned. The name of the file which will be opened and associated with a stream is specified by the first argument **filename**. Another argument `mode` specifies the meaning of opening a file. Its valid values are described in Table 20.4.

Table 20.4: Opening modes for function **fopen()**.

Mode	Meaning
r	open text file for reading
w	truncate to zero length or create text file for writing
a	append; open or create text file for writing at end-of-file
rb	open binary file for reading
wb	truncate to zero length or create binary file for writing
ab	append; open or create binary file for writing at end-of-file
r+	open text file for update (reading and writing)
w+	truncate to zero length or create text file for update
a+	append; open or create text file for update, writing at-end of-file
r+b *or* **rb+**	open binary file for update (reading and writing)
w+b *or* **wb+**	truncate to zero length or create binary file for updating
a+b *or* **ab+**	append; open or create binary file for update, writing at end-of-file

Opening a file with read mode '**r**' as the first character in the mode argument causes the file to be opened only for read operations. Opening a file with write mode '**w**' as the first character in the mode argument causes the file to be opened only for write operations. Opening a file with append mode '**a**' as the first character in the *mode* argument causes all subsequent writes to the file to be forced to the then current end-of-file.

When a file is opened with update mode '**+**' as the second or third character in the above list of *mode* argument values, both input and output may be performed on the associated stream. However, output shall not be directly followed by input without an intervening call to the function **fflush()** or to a file positioning function (**fseek**, **fsetpos**, or **rewind**), and input shall not be directly followed by output without an intervening call to a file positioning function, unless the input operation encounters end-of-file. Opening (or creating) a text file with update mode may instead open (or create) a binary stream in some platforms.

When opened, a stream is fully buffered if and only if it can be determined not to refer to an interactive device. The error and end-of-file indicators for the stream are cleared.

All files which are opened and associated with streams shall be closed before programs exit. The common used function to close a file is **fclose()**. Its prototype is

```
int fclose(FILE *stream);
```

The **fclose()** function returns zero if the stream was successfully closed, or the value of macro **EOF** if any errors were detected. The **fclose** function causes the stream pointed to by *stream* to be flushed and the associated file to be closed. Any unwritten buffered data for the stream is delivered to the host environment to be written to the file; unread buffered data is discarded. The stream is disassociated from the file. If the associated buffer was automatically allocated, it is deallocated.

The following code fragment illustrates how functions **fopen()** and **fclose()** are used.

```
FILE *fpt1, *fpt2;
/* create file named "testfile1" or append to it if exists */
if((fpt1 = fopen("testfile1","a+")) == NULL) {
   printf("Cannot create or open the file\n");
   exit(1);
}
/* create file named "testfile2 for both reading and writing,
   starting at the beginning. */
```

```
if((fpt2 = fopen("testfile2","r+")) == NULL) {
   printf("Cannot open the file\n");
   exit(1);
}
...
fclose(fpt1);
fclose(fpt2);
```

20.8.2 Reading and Writing a File

After a file is opened and associated with a stream, it can perform read or write operations according to the opening mode. The commonly used functions to read a file include functions **getc()** and **fgetc()** which read the next character from the input stream; functions **gets()** and **fgets()** which reads the specified number of characters from the input stream; function **fscanf()** which reads input from the stream under control of certain format, which is discussed in Section 20.3; function **fread()** which is effective to read data block such as some aggregate data type with the specified size. Because of a potential security flaw caused by buffer overflow, function **gets()** is obsolete in C and shall not be used. The application of these input functions are shown in the program below.

```
#include <stdio.h>

FILE *fpt;
char c;
char s[100];

if((fpt = fopen("testfile","r")) == NULL) {
   printf("Cannot create or open the file\n");
   exit(1);
}

/* read a character from file testfile */
if((c = fgetc(fpt)) != EOF)
   printf("c = %c", c);

/* read up to 99 characters from file testfile
   to string s which ends with \0 */
if(fgets(s, 100, fpt)
   printf("s = %s", s);

fclose(fpt);
```

In Ch, files can be manipulated interactively in command mode as follows.

```
> FILE *fp
> fp = fopen("testfile", "w")
> fprintf(fp, "This is output to testfile\n");
> fclose(fp)
> more testfile
This is output to testfile
>
```

where command **more** can be used to display files on screen.

Each input function above used to read files has a corresponding output function used to write files. The commonly used functions to write a file include functions **putc()** and **fputc()** which write a character to the output stream; functions **puts()** and **fputs()** which write a string to the output stream; function **fprintf()** which writes output to the stream under control of certain format, which is discussed in Section 20.3; function **fwrite()** which is effective to write data block such as some aggregate data type with the specified size. For example,

```
#include <stdio.h>

FILE *fpt;

if((fpt = fopen("testfile","w")) == NULL) {
   printf("Cannot create or open the file\n");
   exit(1);
}

/* write a character 'a' to file testfile */
fputc('a', fpt);
/* write a character 'b' to file testfile */
putc('b', fpt);

/* write string "this is a test" to file testfile */
fputs("this is a test", fpt);

fclose(fpt);
```

The following program illustrates how functions **fread()** and **fwrite()** are used to read and write binary files.

```
#include <stdio.h>

FILE *fpt;
struct tag {int i; float f;} s[2];
char buf[20*sizeof(tag)];

if((fpt = fopen("testfile","rb+")) == NULL) {
   printf("Cannot create or open the file\n");
   exit(1);
}

/* read 20 elements of struct tag to buf */
if(fread(buf, sizeof(tag), 20, fpt) != 10) {
   if(feof(fpt))
      printf("End of file.");
   else
      printf("File read error.");
}
```

```
/* write 2 elements of struct tag to file testfile */
s[0].i = 10; s[0].f = 1.2;
s[1].i = 20; s[1].f = 3.4;
fwrite(&s, sizeof(tag), 2, fpt);

fclose(fpt);
```

20.8.3 Random Access

Some files support random access, such as files on the hard drive, but some don't, such as stdout and stdin which are connected to the console. If a file supports random access, a *file position indicator* can be used to determine the position to read or write the next item. By default, the file position indicator points to the beginning of a file when it is opened. The reading or writing functions mentioned in the preceding section read or write items from the position pointed by the file position indicator, and then increment the indicator properly so that it points the next position to read or write. For example, if the item read or written is a character, the indicator is incremented by 1.

Furthermore, for files supporting random access, the indicator can be set by function **fseek()**. Its prototype is

```
int fseek(FILE *stream, long int offset, int whence);
```

The function returns nonzero only for a request that cannot be satisfied. It sets the file position indicator for the stream pointed to by argument `stream`. For a binary stream, the new position, measured in characters from the beginning of the file, is obtained by adding `offset` to the position specified by `whence`. The specified position is the beginning of the file if `whence` is **SEEK_SET**, the current value of the file position indicator if **SEEK_CUR**, or end-of-file if **SEEK_END**.

For a text stream, `offset` shall either be zero or a value returned by an earlier successful call to function **ftell()** on a stream associated with the same file, and `whence` shall be **SEEK_SET**.

After a successful **fseek** call, the next operation on an update stream may be either input or output.

For example, the following codes reads the 6th element of struct `S1` in file `testfile`.

```
struct S1 {
  int i;
  float f;
} s;

FILE *ftp;
int num = 6; /* the 6th element */

if((fpt = fopen("testfile","rb")) == NULL) {
   printf("Cannot create or open the file\n");
   exit(1);
}

/* set the indicator to the 6th element */
fseek(fpt, (num-1) * sizeof(S1), SEEK_SET);

/* read 6th element of struct S1 from testfile */
if(fread(&s, sizeof(S1), 1, fpt) != 1) {
```

```
   if(feof(fpt))
      printf("End of file.");
   else
      printf("File read error.");
}

fclose(fpt);
```

Besides function **fseek()**, file positioning functions also include function **fgetpos()** which stores current values of file position indicators; **fsetpos()** which sets the file position indicators according to the value of an object of type **fpos_t**; function **ftell()** which obtains the current value of the file position indicator for the stream; and function **rewind()** which sets the file position indicator to the beginning of the file.

20.9 Directory Manipulation

Different operating systems have different file systems, the internal handling of a directory is system-dependent. Ch provides a system-independent method to open, close, and read a directory of different operating systems by functions **opendir()**, **closedir()**, **readdir()** and **rewinddir()** defined in the POSIX standard. Prototypes of these functions shown below are defined in the header file **dirent.h**.

```
DIR           *opendir (const char *dirname);
struct dirent *readdir (DIR *dirp);
void          rewinddir(DIR *dirp);
int           closedir (DIR *dirp);
```

Besides these functions, two structs, directory entry struct **dirent** and directory stream struct **DIR**, are also defined in this header file. Struct **dirent** is used for storing the information of a directory entry, such as a file or a subdirectory in the specified directory. The commonly used member in the struct **dirent** is `d_name`, which represents the name of the directory entry. The member `d_name` is of type char array with size `_MAXNAMLEN + 1` including the terminating null character. Macro `_MAXNAMLEN`, which is a system-dependent value, is defined in **dirent.h**. The struct **DIR** represents a directory stream, which is an ordered sequence of all the directory entries in a particular directory.

20.9.1 Opening and Closing a Directory

The function call *dirp* = **opendir(** *dirname* **)** opens a directory stream corresponding to the directory named by the argument of string *dirname*. It returns a directory stream *dirp*, which is a pointer to an object of type **DIR** for directory stream. The directory stream is positioned at the first entry. Variable *dirp* can be used by other functions, such as **readdir()**, to manipulate the directory. The function call **closedir(** *dirp* **)** closes the directory stream referred to by *dirp* and returns zero if successful. The typical structure of a program using functions **opendir()** and **closedir()** is as follows.

```
#include <stdio.h>
#include <dirent.h>
int main(void) {
  DIR *dirp;

  dirp = opendir("."); // open the current directory
  ...   // manipulating current directory by dirp
```

```
  closedir(dirp);
  return 0;
}
```

Function **stat()** can be used to verify that the argument *dirname* of function **opendir()** is a directory name before opening it. The first argument of function **stat()** is a file name and the second argument will pass back all information about that file in an object of struct **stat**. Upon successful completion, a value of zero shall be returned. Otherwise, a value of `-1` shall be returned and **errno** shall be set to indicate the error. The prototype of function **stat()** is defined in header file **sys/stat.h** as follows.

```
int stat(const char * name, struct stat *stbuf);
```

The structure describing the value passed in the second argument of function **stat()** is typically defined as follow.

```
struct stat {
    dev_t   st_dev;    /* block device inode is on */
    ino_t   st_ino;    /* inode number */
    mode_t  st_mode;   /* protection and file type */
    nlink_t st_nlink;  /* hard link count */
    uid_t   st_uid;    /* user id */
    gid_t   st_gid;    /* group id */
    dev_t   st_rdev;   /* the device number for a special file */
    off_t   st_size;   /* number of bytes in a file */
    time_t  st_atime;  /* time of last access */
    time_t  st_mtime;  /* time of last modify */
    time_t  st_ctime;  /* time of last status change */
}
```

The meanings of these members of structure are explained by the comment fields. All typedefed types, such as **dev_t** and **mode_t**, are defined in **sys/types.h**. With the macros shown below, the member **st_mode** is used to test whether a file is of the specified type.

```
S_ISDIR(m)    // Test macro for a directory file.
S_ISCHR(m)    // Test macro for a character special file.
S_ISBLK(m)    // Test macro for a block special file.
S_ISREG(m)    // Test macro for a regular file.
S_ISFIFO(m)   // Test macro for a pipe or a FIFO special file.
```

The value *m* supplied to the macro is the value of **st_mode** from the **stat** structure *stbuf*. The macro evaluates to a nonzero value if the test is true, zero if the test is false. For example, the code

```
#include <dirent.h>
#include <sys/stat.h>
...

struct stat stbuf;
DIR *dirp;
char *dirname = "/home/myname";
...
```

```
#include <dirent.h>
#include <sys/stat.h>

int main() {
    struct stat stbuf;
    DIR *dirp;
    struct dirent *direntp;
    char *dirname = ".";  /* current directory */

    if(!stat(dirname, &stbuf) && S_ISDIR(stbuf.st_mode)) {
      dirp = opendir(dirname);
    }

    printf("List of files in directory %s :\n", dirname);
    while(direntp = readdir(dirp)) {
      printf("%s\n", direntp->d_name);
    }

    closedir(dirp);
    return 0;
}
```

Program 20.3: Read all entries in the current directory.

```
if(!stat(dirname, &stbuf) && S_ISDIR(stbuf.st_mode)) {
  dirp = opendir(dirname); // open the directory
}
...   // manipulating the directory by dirp

closedir(dirp);
...
```

checks if `/home/myname` is a directory name before opening it using function **opendir()**.

Note that a successful call to any of the familys of function **exec**() will close any directory streams that are open in the calling process.

20.9.2 Reading a Directory

A currently open directory, which is referred to by *dirp*, can be read by function call *direntp*= **readdir(***dirp***)**. The function **readdir()** takes an argument of pointer to struct **DIR**, *dirp*, which is returned by the function **opendir()**. The return value of function **readdir()**, *direntp*, is a pointer to the structure **dirent** which represents the directory entry at the current position in the directory stream to which *dirp* refers. The function positions the directory stream at the next entry. The name of the current directory entry can be expressed by `direntp->d_name`. Upon reaching the end of the directory stream, it returns a **NULL**. For example, Program 20.3 opens the current directory, reads all entries and prints out their names.

As another example, Program 20.4 searches through directories in the system variable **_path** and prints a list of the executable files to stdout. Whether a file is executable or not is checked by function call of `access(file, X_OK)` described in section 4.16.

Based on Program 20.3, Program 20.5 not only reads the current directory, but also goes into its subdirectories recursively. The function `dirwalk` is a recursive function which takes a directory name as the argument and returns no value. Names and sizes of all entries in the directory, which is specified by the argument, will be printed out. If an entry is a directory, except for "`.`" and "`..`", the function `dirwalk()`

```
#!/bin/ch
/*----------- printexec --------------------
This program searches through _path and
prints all the names of the executable files.
-----------------------------------------*/
#include<unistd.h>
#include<sys/stat.h>
#include<dirent.h>
string_t s, filename;
struct stat sbuf;
struct dirent *direntp;
DIR * dirp;

foreach(s; _path) { //or foreach(s; _path; NULL; ";")
  dirp = opendir(s);
  if(dirp != NULL) {
    while(direntp = readdir(dirp)) {
      /* or filename = stradd(s, "/", direntp->d_name); */
      sprintf(filename, "%s/%s", s, direntp->d_name);
      stat(filename, &sbuf);
      if(S_ISREG(sbuf.st_mode) && access(filename, X_OK) == 0)
        printf("%s\n", filename);
    }
    closedir(dirp);
  }
}
```

Program 20.4: Print out commands in the search paths.

will call itself and pass the name of the entry as the argument. So, this program can loop through all subdirectories in the current directory. Each directory contains entries for itself, "`.`", and its parents, "`..`". These must be skipped, or the program will get into an infinite loop. As it is described in the previous section, the member **st_size** of struct **stat** is used to indicate the number of bytes of a file.

The **rewinddir()** function resets the position of the directory stream so that *dirp* refers to the beginning of the directory. It also causes the directory stream to refer to the current state of the corresponding directory, as a call to **opendir()** would have done. It does not return a value. For example, Program 20.6 opens the current directory, reads the first four entries, and prints out their names first. Then, the **rewinddir()** function is called to read the directory again from the beginning.

```
/* File: rec.ch */
#include <dirent.h>
#include <sys/stat.h>

void dirwalk(char *dirname);

int main() {
    char *dirname = ".";

    dirwalk(dirname);
    return 0;
}

/* open, read a directory, and go into its subdirectories recursively */
void dirwalk(char *dirname) {
    struct stat stbuf;
    DIR *dirp;
    struct dirent *direntp;
    char filename[1024];

    /* open the directory */
    if(!stat(dirname, &stbuf) && S_ISDIR(stbuf.st_mode)) {
      dirp = opendir(dirname);
      if (dirp == NULL) return;
    }

    /* read the directory and go into its subdirectories recursively */
    while(direntp = readdir(dirp)) {
      sprintf(filename, "%s/%s", dirname, direntp->d_name);
      stat(filename, &stbuf);
      printf("size of %s is %d\n", filename, stbuf.st_size);

      /* if the file is a directory, except for "." and ".." */
      if((strcmp(".", direntp->d_name) != 0) \
          && (strcmp("..", direntp->d_name) != 0) \
          && S_ISDIR(stbuf.st_mode))
      {
        dirwalk(filename);  /* recursive calling this function */
      }
    }

    closedir(dirp);
}
```

Program 20.5: Go through the directories recursively.

```
#include <dirent.h>
#include <sys/stat.h>

int main() {
    int i;
    struct stat stbuf;
    DIR *dirp;
    struct dirent *direntp;
    char *dirname = ".";

    if(!stat(dirname, &stbuf) && S_ISDIR(stbuf.st_mode)) {
      dirp = opendir(dirname);
    }

    printf("The first four entries in directory %s are :\n", dirname);
    for(i = 0; i < 4; i++) {
      direntp = readdir( dirp );
      printf( "%s\n", direntp->d_name );
    }

    /* reset the position of the directory stream to which
       dirp refers to the beginning of the directory. */
    rewinddir(dirp);

    printf("\nAfter calling function rewinddir(), files in "
           "directory %s are :\n", dirname);
    while(direntp = readdir(dirp)) {
      printf("%s\n", direntp->d_name);
    }

    closedir(dirp);
    return 0;
}
```

Program 20.6: Rewind the directory.

Chapter 21

Safe Ch

Ch was designed with the ease of use and security in mind. The pointer and memory allocation/deallocation make C/C++ powerful, but they are not easy to handle for an inexperienced programmer. The inappropriate handling of the pointer and memory can lead to buffer overflow. We have noticed that a very high percentage of programs that crash suffer from the mishandling of the string.

Ch recognizes this shortcoming and has a built-in string type with automatic memory management to resolve this problem. It can work seamlessly with the type char* and char []. Users are encouraged to use this feature for rapid application development without concerns for memory handling and pointers. Furthermore, Ch checks array bounds automatically to avoid memory corruption.

21.1 Safe Ch Shell

Safe Ch is introduced to address the security concerns for C-based applets to run across the Internet or as a restricted shell. Safe Ch disables the use of C pointer and reduces the potential security risk while taking advantage of pointers in other applications such as real-time control of machinery and data acquisition. Safe Ch has a sandbox and limits a malicious applet from gaining privilege to take full control of the computer.

21.1.1 Startup in Windows

Once you have downloaded and installed the Ch software, safe Ch can be started by typing command `chs` in Unix. In Windows, use the menu sequence **Start->Run**, then type `chs` or `chs.exe`. Safe Ch shell can be invoked by command with option as `ch -S` in both Unix and Windows.

21.2 Features Disabled in a Sandbox

Program chs is safe Ch shell. If the `-S` flag is present when the Ch language environment is invoked as `ch -S` , the Ch shell is invoked as a safe shell also. The macro `_SCH_` is predefined with value 1 for a safe shell. The execution environment of a safe shell is more controlled than that of the regular shell. The actions of `ch -S` are identical to those of ch, except that the following features are disabled:

- Changing directory by cd and chdir.
- Specifying path or command names containing character '`/`' in input command, command statement, and command substitution.
- Specifying path or command names containing character '`/`' or '`\`' in a dot command statement.

- Specifying path with first character '/' or '\' in `file` of `#include<file>`.
- `#include ''file"` is treated as `#include<file>`
- Redirecting input and output (`<, <<, >, >|, and >>`)
- Using system variables `_path, _fpath, _lpath, _ppath, _ipath, _user, _home, _cwd, _cwdn, _shell, _host` as lvalues (The values of these system variables after execution of .chsrc are kept for internal use. To the safe shell user, these system variables have a NULL value. To test setup, you may print the values of these system variables inside file .chsrc in Unix and _chsrc in Windows).
- Using shell commands `chparse` and `chrun` in interactive mode.
- Specifying path or command names containing ; (will be Ok in future)
- Specifying path or command names containing `|` (will be Ok in future).
- Declaration of pointer type is permitted. But, an lvalue cannot be of pointer type. For example,

```
char *p, **p2;   // ok
p = malloc(90);  // bad
p2 = &p;         // bad
```

  But, array type can be used as an lvalue. For example,

```
int a[2]={1,2};
a[1] =90;
```

- Casting a non-pointer value to a pointer. For example,

```
char *p;
p = (char *)16; // bad
```

- Pointer arithmetic. For example,

```
char *p;
int a[15];
p = p+128;    // bad
*(a+10)=12;   // bad
```

- Generic functions _execv(), _execvp(), _fopen(), _fork(), _fstat(), _lstat(), _pipe(), _popen(), _remove(), _rename(), _socket(), _socketpair(), _stat(), _utime(), _system(), access(), getenv(). open(), putenv(), setrlimit(), umask().
- Functions or function files with return type qualified with type qualifier **restrict** are called restricted functions. The following restricted functions cannot be called by safe Ch programs:

  accept(), chdir(), chown(), chroot(), creat(), execl(), execv(), execle(), execve(), execlp(), execvp(), fchdir(), fchown(), fchroot(), fdopen(), fopen(), fstat(), gethostname(), kill(), lchown(), link(), lstat(), mkdir(), pipe(), popen(), remove(), rename(), rmdir(), socket(), socketpair(), stat(), system(), unlink().

- Macro offsetof().
- No command and file name completion in a command shell.

The restrictions above are enforced after .chsrc in Unix and _chsrc in Windows in the home directory is interpreted. For maximum security, a system administrator may take the ownership of .chsrc in Unix and _chsrc in Windows and change the mode of the file to readable only. The CPU resource in safe shell for each process can also be restricted by modifying file .chsrc. If Ch is invoked with option -S, option -f will be ignored. These additional restrictions are relaxed for function files located in a client. Therefore, an important safety guideline is to not use arguments from a function file as input to restricted functions directly. Function files can call restricted functions. The memory outside the array boundary is guaranteed to not be contaminated if an array is used as a pointer to void or pointer to char in the following functions: fgets(), fread(), gets(), memcpy(), memmove(), memset(), read(), recv(), sprintf() strcat(), strcpy(), strncat(), strncpy(). The contaminated memory by functions fscanf(), scanf(), and sscanf() are set to null character to close a possible security hole. If a program tries to write to the memory outside the array boundary, an error message will be generated. Since variables of pointer type cannot be invoked in safe shell, these functions will be safe to use.

When a command to be executed is found to be a Ch program, the safe shell invokes ch -S to execute it. If a Ch program is invoked with shell identification #!/bin/ch without option -S, the safe shell invokes ch to execute it. Thus, it is possible to provide to the end-user Ch programs that have access to the full power of the regular shell, while imposing a limited number of commands; this scheme assumes that the end-user does not have write and execute permissions in a directory containing commands. Therefore, the writer of the .chsrc has complete control over user actions, by performing guaranteed setup actions and leaving the user in an appropriate directory (probably not the login directory). A default directory CHHOME/sbin has been setup for putting binary and Ch commands, that can be safely invoked by the safe Ch shell.

In Winwodw, a safe Ch shell can be started from the Windows Explorer or **Start->Run** menu. Commands `chs` and `ch -S` do not work in Windows NT/2000/XP when they are executed in command prompt windows or Ch windows.

21.3 Restricted Functions

Functions declared with type modifier `restrict` for return type are called restricted functions. They cannot be called by safe Ch programs as shown below.

```
#!/bin/ch -S
restrict void funct(void) {
   printf("This function cannot be called by Safe Ch program.\n");
}
funct(); // Error: call restricted functions
```

21.4 Safe Ch Programs

Programs in directories `CHHOME/sbin` and `CHHOME/toolkit/sbin` are accessible by regular and safe Ch shells. For example, binary executable program **gnuplot** for plotting and Ch program **license.ch** for license information in `CHHOME/sbin` are accessible to safe Ch shell and safe Ch scripts.

21.5 Applets and Network Computing

Cross-platform network computing in Ch is accomplished using a safe shell. A data stream, transferred through the WWW with a mime type extension .chs, is treated as a Ch applet and executed in safe Ch shell. To perform network computing, both Web server and Web browser shall be setup to generate and recognize Ch applets, respectively.

On-line documentation and demonstrations of cross-platform network computing in Ch for applications in design and manufacturing are available on the WWW at `http://www.softintegration.com`.

Chapter 22

Library, Toolkit, and Package

Section 3.4.5 described how to run a program with multiple files. This chapter will give details on how to create library and software packages to run in Ch.

22.1 Library

Functions are building blocks for Ch programs. Functions could be either written by users or included in libraries provided by some software packages. Many commonly used functions such as operations on characters and strings, input and output operations, mathematical functions, date and time conversions, and dynamic memory allocation have been included in *standard libraries*. Users don't need to rewrite these functions, what you need to do is to just call these functions from a standard library. To use functions in a library, the program shall have preprocessing directive `#include` to include proper header files of the library. Header files are interfaces between the application programs and libraries. All the function prototypes, data types and macros defined in header files provide the users with more facilities to construct their application programs efficiently. For example, including the header file `math.h` in the C standard library, a program can use all the mathematical functions such as **sin()**, and **cos()** declared in this header file,

C includes a large member of standard libraries that are independent of the host operating system. It supports operations on the floating-point environment which is defined in `float.h`, mathematics in `math.h`, input and output in `stdio.h`, string handling in `string.h`, data and time in `time.h`, extended multibyte and wide-character utilities in `wchar.h`, etc. In addition to standard libraries in C, IEEE Portable Operating System Interface (POSIX) Standard also provides a large member of libraries to promote portability of application programs across Unix system environments, such as operations on file system directory entries defined in `dirent.h`, interprocess communication in `fcntl.h`, semaphore mechanism in `semaphore.h`, etc.

Most functions defined in C and POSIX standards are supported in Ch. In addition, Ch supports many high-level functions such as 2D and 3D plotting functions defined in `chplot.h` and advanced numerical analysis functions defined in `numeric.h`, etc. The summary of the libraries supported by Ch is shown in Table 22.1. In Windows, the libraries **windows.h** and **windows/winsock.h** are also supported by Ch. All the header files for libraries supported by Ch can be found in the directory `CHHOME/include` and its subdirectories.

Ch allows the users to add their own libraries. The components, such as function files, header files, dynamically loaded library, and commands of a library could be placed anywhere in the file system, so that it is important to make sure that Ch knows where these components are. As it is mentioned before, Ch searchs the directories specified in system variables **_fpath**, **_ipath**, **_lpath**, and **_path** for function files, header files, dynamically loaded library, and commands, respectively. The descriptions and default values of

Table 22.1: Library summary.

Header file	Description	C	POSIX	Ch
aio.h	Asynchronous input and output		X	X
arpa/inet.h	Internet operations			X
array.h	Computational arrays			X
assert.h	Diagnostics	X	X	X
chplot.h	2D and 3D plotting			X
chshell.h	Ch Shell functions			X
complex.h	Complex numbers	X		X
cpio.h	Cpio archive values			X
crypt.h	Encryption functions			X
ctype.h	Character handling	X	X	X
dirent.h	Format of directory entries		X	X
dlfcn.h	Dynamically loaded functions			X
errno.h	Error numbers	X	X	X
fcntl.h	Interprocess communication functions		X	X
fenv.h	Floating-point environment	X		X
float.h	platform-dependent floating-point limits	X	X	X
glob.h	Pathname pattern-matching types			X
grp.h	Group structure		X	X
inttypes.h	Fixed size integral types	X		X
iostream.h	input and output stream in C++ style			X
iso646.h	Alternative spellings	X		X
libintl.h	Message catalogs for internationalization			X
limits.h	platform-dependent integral limits	X	X	X
locale.h	Locale functions	X	X	X
malloc.h	Dynamic memory management functions			X
math.h	Mathematical functions	X	X	X
mqueue.h	Message queues		X	X
netconfig.h	Network configuration database			X
netdb.h	Network database operations			X
netdir.h	Name-to-address mapping for transport protocols			X
netinet/in.h	Internet Protocol family			X
new.h	Memory allocation error handling in C++ style			X
numeric.h	Numerical analysis			X
poll.h	Definitions for the poll() function			X
pthread.h	Threads		X	
pwd.h	Password structure		X	X
re_comp.h	regular-expression-matching functions for re_comp()			X
readline.h	Readline function			X
regex.h	regular-expression-matching types			X
sched.h	execution scheduling		X	X
semaphore.h	Semaphore functions		X	X
setjmp.h	Non-local jumps	X	X	X

Table 22.1: Library summary (continued).

Header file	Description	C	POSIX	Ch
signal.h	Signal handling	X	X	X
stdarg.h	Variable argument lists	X	X	X
stdbool.h	Boolean numbers	X		X
stddef.h	Miscellaneous functions and macros	X	X	X
stdint.h	Integer types	X	X	X
stdio.h	Input and output	X	X	X
stdlib.h	Utility functions	X	X	X
string.h	String functions	X	X	X
stropts.h	streams interface			X
sys/acct.h	Process accounting			X
sys/fcntl.h	Control file			X
sys/file.h	Accessing the file struct array			X
sys/ioctl.h	Control device			X
sys/ipc.h	Interprocess communication access structure			X
sys/lock.h	Locking processes			X
sys/mman.h	Memory management declarations		X	X
sys/msg.h	Message queue structures			X
sys/procset.h	Set processes			X
sys/resource.h	XSI resource operations			X
sys/sem.h	Semaphore facility			X
sys/shm.h	Shared memory facility			X
sys/socket.h	Internet Protocol family			X
sys/stat.h	File structure function		X	X
sys/time.h	Time types			X
sys/times.h	File access and modification times structure		X	X
sys/types.h	Data types		X	X
sys/uio.h	Vector I/O operations			X
sys/un.h	Unix-domain sockets			X
sys/utsname.h	System name structure		X	X
sys/wait.h	Evaluating exit statuses		X	X
syslog.h	System error logging			X
tar.h	Extended tar definitions			X
termios.h	Define values for termios		X	X
tgmath.h	Type-generic mathematical functions	X		X
time.h	Time and date functions	X	X	X
tiuser.h	Transport layer interface			X
unistd.h	System and process functions		X	X
utime.h	Access and modification times structure		X	X
wait.h	Wait for child process to stop or terminate			X
wchar.h	Multibyte I/O and string functions	X		X
wctype.h	Multibyte character class tests	X		X

Note: the symbol ‘X’ indicates that the library is supported by the standard.

system variables **_path**, **_ipath**, **_fpath** and **_lpath** can be found in section 2.3.1. To check the current values of these system variables, the user can type variable names in the command mode directly. For example, to check the current value of **_ipath**, the user can type the command

```
> _ipath
/usr/ch/include;/usr/ch/toolkit/include;
>
```

Thus, Ch searchs the directories `/usr/ch/include` first for a header file, if can't find it, then searchs the directory `/usr/ch/toolkit/include`.

Assume a Ch program `pack1.ch`, which is in a software package, includes statements below

```
#include <stdio.h>
#include "pack1.h"
int main() {
  /* ... */
  myfunc1(10);
  /* ... */
  return 0;
}
```

To run this program, Ch needs to know where to find the header files **stdio.h**, `pack1.h`, and function file `myfunc1.chf`. By default, Ch searches directories specified by system variable **_ipath** for header file **stdio.h**; searches the current directory and then directories specified by system variable **_ipath** for header file `pack1.h`; searches directories specified in system variable **_fpath** for function file `myfunc1.chf`.

To ensure that Ch finds the right files in the right directories, the user can either add the directories in which the relevant files are located into the values of the corresponding system variables, or copy these files to the directories which are already included in the corresponding system variables. But, in most cases, users may have no write permission of the directories which are included in the default values of these system variables. Generic function **stradd()** can be used to add paths into the value of the corresponding system variable. Assume the function file `myfunc1.chf` is located in the directory `/home/mydir/pack1/lib` and the header file `pack1.h` in the directory `/home/mydir/pack1/include`. The commands below add these two directories to the end of the system variables **_fpath** and **_ipath**, respectively.

```
> _fpath = stradd(_fpath, "/home/mydir/pack1/lib;")
/usr/ch/lib/libc;/usr/ch/lib/libch;/usr/ch/lib/libopt;
/usr/ch/lib/libch/numeric;/home/mydir/pack1/lib;
> _ipath = stradd(_ipath, "/home/mydir/pack1/include;")
/usr/ch/include;/usr/ch/toolkit/include;/home/mydir/pack1/include;
>
```

If the user wants these two paths to be automatically added every time when Ch is started, the commands below

```
_fpath = stradd(_fpath, "/home/mydir/pack1/lib;");
_ipath = stradd(_ipath, "/home/mydir/pack1/include;");
```

should be added in the startup file, such as **.chrc** in user's home directory in Unix. More information about how to customize the startup files can be found in section 3.2. After that, when the program `pack1.ch` is executed, Ch will search for header file `pack1.h` in directories `/usr/ch/include`, `/usr/ch/toolkit/include`, and `/home/mydir/pack1/include`, one after another. Similarly, after searching the default paths for function files, Ch will search for function file `myfunc1.chf` in the directory `/home/mydir/pack1/lib`.

22.2 Toolkit

Unlike C and POSIX standard libraries which provide general utilities, some self-contained systems and software standards, such as Windows, X-Windows, Motif, OpenGL, ODBC, only provide functions for special purposes or in some special fields. For example, X-Windows provides a low-level programming interface called Xlib to the X Windows, a network-based graphics window system. Motif provides a high-level programming interface to the X Windows. OpenGL provides a programming interface for 3D graphics. ODBC provides a programming interface for database access. These software standards are treated as toolkits in Ch and located in CHHOME/toolkit.

To organize these toolkits, all the header files are placed in the directory `CHHOME/toolkit/include` and its subdirectories, the dynamic loaded files are in the directory `CHHOME/toolkit/dl`. By default, these two system directories are already included in the system variables **_ipath** and **_lpath**. Nevertheless, the directories for Ch function files are too many to be added to the system variable **_fpath** in advance. The solution in Ch is to add preprocessing directives `#pragma _fpath` into the corresponding header files. As it is described in Table 5.2, the pragma directive

```
#pragma _fpath <fpathname>
```

adds the directory `CHHOME/toolkit/lib/fpathname` into the system variable **_fpath** of the subshell in which the Ch program is running. For example, if the header file `gtk/gtk.h` for the GTK toolkit contains

```
#ifndef __GTK_H__
#define __GTK_H__
#pragma _fpath <GTK/gtk>
...

#endif /* __GTK_H__ */
```

with the directive `#pragma _fpath <GTK/gtk>` in the header file `gtk/gtk.h`, the Ch program using GTK will search the directory `CHHOME/toolkit/lib/GTK/gtk` for function files. The other form of the pragma directive,

```
#pragma _fpath /dir1/dir2/fpathname
```

can add the absolute path name `/dir1/dir2/fpathname` into the system variable **_fpath** of the subshell. Similarly, directives

```
#pragma _ipath /dir1/dir2/ipathname
#pragma _lpath /dir1/dir2/lpathname
#pragma _path  /dir1/dir2/pathname
```

add these absolute path names into the system variables **_ipath**, **_lpath**, and **_path**, respectively.

The pragma directive

```
#pragma exec expr
```

will evaluate an expression when the expression is parsed. This directive can be used to add a path to the system path variables. For example, assume the home directory obtained by the function call `getenv("HOME")` is `/home/myname`, the directive

```
#pragma exec _fpath=stradd(_fpath, getenv("HOME"), "/chfunc;");
```

adds the directory `/home/myname/chfunc` in the system variable **_fpath** for function files at runtime.

22.3 Package

Section 3.4.5 described how to run programs with mulitple files using preprocessing directive `pragma` along with `import` and `importf`. In this section, handling of packages in Ch will be presented.

Besides libraries and toolkits which come with Ch, the user developed software packages can also be supported in Ch. Since a software package may consist of many files located in different directories, and these files may be relevant to other components or files inside or outside the package, such as header files, function files, dynamically loaded libraries, imported files, and even commands, Ch has to know where these relevant files can be found in order to execute programs in a software package properly. According to Ch conventions, header files for a software package are located in the `include` subdirectory, function files in the `lib` subdirectory, dynamically loaded libraries in the `dl` subdirectory, and command files in the `bin` subdirectory.

Assume a Ch program `pack1.ch`, which is in a software package, includes statements below

```
#include <stdio.h>
#include "pack1.h"
int main() {
  /* ... */
  myfunc1(10);
  /* ... */
  return 0;
}
#pragma importf "module2.ch"
#pragma import  <module3.ch>
```

To run this program, Ch needs to search some paths for header files **stdio.h** and `pack1.h`, function file `myfunc1.chf`, and imported files `module2.ch` and `module3.ch` in the parse phase. By default, Ch searches directories specified by system variable **_ipath** for header file **stdio.h**; searches the current directory and then directories specified by system variable **_ipath** for header file `pack1.h`; searches directories specified in system variable **_fpath** for function file `myfunc1.chf`; searches the current directory and then directories specified by system variable **_fpath** for the file `module2.ch`; searches directories specified in system variable **_path** for the file `module3.ch`; searches directories specified in system variable **_lpath** for the dynamic loaded files if they are used in function files. For many software packages, if all the directories for these software packages are placed into these system variables, the sizes of the values of these system variables will be too large to be maintained or searched efficiently. Ch introduces the directive `#pragma package` to help the users to organize their software packages.

Like the preprocessing directives `#pragma _fpath` described in the previous section, which adds a single user specified directory to system variable `_fpath`, the preprocessing directive `#pragma package` adds multiple user specified directories automatically to corresponding system variables, including `_fpath`, `_ipath`, `_lpath`, and `_path`. All these system variables are updated in subshells in which Ch programs are executed. This directive doesn't affect the system variables in its parent shell. As described in Table 5.2, the directive `#pragma package` has three formats. The first one

```
#pragma package <packagename>
```

adds the subdirectories `bin`, `lib`, `include` and `dl` in the specified directory `_ppath/packagename` into the system variables **_path**, **_fpath**, **_ipath** and **_lpath**, respectively. The description and default values of the system variable **_ppath** can be found in section 2.3.1. Assume the value of the system variable **_ppath** is `/usr/ch/package`, the directive `#pragma package <pack1>` adds path names `/usr/ch/package/pack1/bin`, `/usr/ch/package/pack1/lib`,

`/usr/ch/package/pack1/include`, and `/usr/ch/package/pack1/dl` into the corresponding system variables. The other two formats

```
#pragma package "/home/mydir/packagename"
```

and

```
#pragma package </home/mydir/packagename>
```

add the subdirectories `bin`, `lib`, `include` and `dl` in the specified directory `/home/mydir/packagename` into system variables **_path**, **_fpath**, **_ipath** and **_lpath**, respectively. The absolute path names are used in these two cases.

An example of a Ch software package below illustrates how the directive `#pragma package` works. Assume files of this package is located in the directory `/home/mydir/pack1` as follows.

```
pack1.ch     --  /home/mydir/pack1/bin
pack1.h      --  /home/mydir/pack1/include
module1.ch   --  /home/mydir/pack1/bin
myfunc1.chf  --  /home/mydir/pack1/lib
```

List 1 - File `/home/mydir/pack1/bin/pack1.ch`.

```
#!/bin/ch
#pragma package </home/mydir/pack1>
#pragma import <module1.ch>

#include <stdio.h>
#include "pack1.h"

int main() {
  printf("Output from the main function\n");
  myfunc1(10);
  return 0;
}
```

When this Ch program is executed, the directive `#pragma package </home/mydir/pack1>` adds the subdirectories `bin`, `lib`, `include` and `dl` in directory `/home/mydir/pack1` to the end of the system variables **_path**, **_fpath**, **_ipath** and **_lpath**, respectively, so that Ch can find all relevant components in the parse phase.

List 2 - File `/home/mydir/pack1/include/pack1.h`.

```
#ifndef PACK1_H
#define PACK1_H
int myfunc1(int i);
#endif // PACK1_H
```

The prototype of the user defined functions `myfunc1()` is in this header file.

List 3 - The file `/home/mydir/pack1/bin/module1.ch` includes one statement

```
printf("Output from module1.ch in the subdirectory bin\n");
```

List 4 - The file `/home/mydir/pack1/lib/myfunc1.chf`.

```
int myfunc1(int i) {
  printf("Output from the function file myfunc1.ch in\
          subdirectory lib, i = %d\n", i);
  return 0;
}
```

The function `myfunc1()` is defined in this function file.

List 5 - The result for execution of the Ch program `pack1.ch` is shown below.

```
> cd /home/mydir/pack1/bin
> ./pack1.ch
Output from module1.ch in the subdirectory bin
Output from the main function
Output from the function file myfunc1.ch in subdirectory lib, i = 10
>
```

If the path name `/home/mydir` has been added to the system variable `_ppath` or the package is moved to `CHHOME/package/pack1` where `CHHOME` is the Ch home directory, the directive `#pragma package </home/mydir/pack1>` can be changed to `#pragma package <pack1>` in the program `pack1.ch`, and the result of the execution is the same. Also program `pack1.ch` can be moved to any directory which is included in search paths for commands.

Sometimes, it is desirable to run C programs in Ch without any modification of the original C code. This can be achieved by modifying a header file and startup file. For example, if the package `pack1` is installed in `CHHOME/package/pack1` and the header file pack1.h contains the code as shown below.

```
#ifndef PACK1_H
#define PACK1_H
#pragma package pack1
int myfunc1(int i);
#endif // PACK1_H
```

A new path for the system variable **_ipath** for header files can be added by the following statement in a system startup file `chrc` or individual user's startup file `.chrc` in Unix or `_chrc` in Winodws.

```
_ipath = stradd(_ipath, "<CHHOME>/package/pack1/include;");
```

where `<CHHOME>` shall be substituted by the Ch home directory. The program `pack1.ch` or `pack1.c` can then be executed without modification.

A Ch package can be packaged using command `crteatepkg.ch` in a Ch shell. For example, assume the directory `sample` contains a package with separate subdirectories for header files, function files, dynamically loaded library, and demos. The command below

```
> ch createpkg.ch sample 1.0
```

will create a package file called `sample-1.0.tar.gz`, which, after unpacking, can be installed by command **installpkg.ch** in Ch.

Part II

The Library for Scientific Computing

Chapter 23

Two and Three-Dimensional Plotting

Two and three dimensional plottings can be easily accomplished in Ch or in C++ using SoftIntegration Graphical Library. A program using SoftIntegration Graphical Library can run portably either in Ch intepretively or compiled using a C++ compiler. Plots can be generated from data arrays or files, and can be displayed on a screen, saved as an image file in different file formats, or output as a stdout stream in png or gif file format for display in a Web browser through a Web server. This chapter describes how to write programs to generate plots in two and three dimensional spaces.

Note that for plotting in Mac OS X using Aquaterm available at http://aquaterm.soureforge.net, you need to add

```
putenv("GNUTERM=aqua");
```

in the system startup file `CHHOME/config/chrc` or the individual user's startup file `.chrc` in the user's home directory.

23.1 A Class for Plotting

The plotting class **CPlot** enables high-level creation and manipulation of plots. Member functions of class **CPlot** are listed in Table 23.1. Detailed description of each function can be found in the chapter about plotting in *The Ch Language Environment — Reference Guide*. In the following subsections, features applicable to both 2D and 3D plotting will be presented.

23.1.1 Data for Plotting

A data set is necessary for creating a plot. The data for a plot can be stored in the memory of the program or in a file. The simplest form of data used for a two-dimensional plot has two arrays, one for x-axis and the other for y-axis as shown in Program 23.1. Figure 23.1 displays the plot produced by Program 23.1. There are two member functions used in Program 23.1. Function **CPlot**::**data2D**() adds data for plotting. At the point where function **CPlot**::**plotting**() is called, a plot is generated. If function $x\sin(x)$ is plotted, the statement `y = x*sin(x)` shall be changed to `y = x.*sin(x)` with element-wise multiplication operator for array in Program 23.1.

The first parameter `x` of member function

```
int CPlot::data2D(array double  x[&], array double &y);
```

is a one-dimensional array of reference type. The second parameter `y` in function **CPlot**::**data2D**(`x, y`) is an array of reference type. It can be a one-dimensional array of size n or a two-dimensional array of size m

Table 23.1: Member functions of class **CPlot**.

Function	Description
CPlot()	Class constructor. Creates and initializes a new instance of the class.
~**CPlot**()	Class destructor. Frees memory associated with a instance of the class.
arrow()	Add an arrow to a plot.
autoScale()	Enable or disable autoscaling of plot axes.
axis()	Enable or disable drawing of x-y axis on 2D plots.
axisRange()	Set the range for a plot axis.
axes()	Specify the axes for a data set.
barSize()	Set the size of error bars.
border()	Enable or disable drawing of a border around the plot.
borderOffsets()	Set plot offsets of the plot border.
boxBorder()	Enable or disable drawing of a border for a plot of box type.
boxWidth()	Set the width for a box.
changeViewAngle()	Change the view angles for a 3D plot.
circle()	Add a circle to a 2D plot.
colorBox()	Enable or disable the drawing of a color box for 3D surface plots.
contourLabel()	Enable or disable contour labels for 3D surface plots.
contourLevels()	Set contour levels for 3D plot to be displayed at specific locations.
contourMode()	Set the contour display mode for 3D surface plots.
coordSystem()	Set the coordinate system for a 3D plot.
data()	Add 2D, 3D, or multi-dimensional data to an instance of the **CPlot** class.
data2D()	Add one or more 2D data sets to an instance of the **CPlot** class.
data2DCurve()	Add a set of data for 2D curve to an instance of the **CPlot** class.
data3D()	Add one or more 3D data sets to an instance of the **CPlot** class.
data3DCurve()	Add a set of data for 3D curve to an instance of the **CPlot** class.
data3DSurface()	Add a set of data for 3D surface to an instance of the **CPlot** class.
dataFile()	Add a data file to an instance of the **CPlot** class.
dataSetNum()	Obtain the current data set number in an instance of the **CPlot** class.
deleteData()	Remove data from a previously used instance of the **CPlot** class.
deletePlots()	Remove any data from a previously used instance of the **CPlot** class and reinitialize option to default values.
dimension()	Set plot dimension to 2D or 3D.
displayTime()	Display the current time and date on the plot.
enhanceText()	Use enhanced text for special symbols.
fillStyle()	Fill a box or filled curve with a solid color or pattern.
func2D()	Add a set of 2D data using a function to an instance of the **CPlot** class.
func3D()	Add a set of 3D data using a function to an instance of the **CPlot** class.
funcp2D()	Add a set of 2D data using a function with a paramenter to an instance of the **CPlot** class.
funcp3D()	Add a set of 3D data using a function with a parameter to an instance of the **CPlot** class.
getLabel()	Get the label of an axis.
getOutputType()	Get the output type of a plot.
getSubplot()	Get a pointer to an element of a subplot.
getTitle()	Get the title of the plot.
grid()	Enable or disable display of a grid.
isUsed()	Test if an instance of the **CPlot** class has been used.
label()	Set axis labels.

Table 23.1: Member functions of class **CPlot**(continued).

Function	Description
legend()	Add a legend for a data set.
legendLocation()	Specify the plot legend location.
legendOption()	Set options for legends of a plot.
line()	Add a line to a plot.
lineType()	Set the line type, width, and color for lines, impulses, steps, etc.
margins()	Set plot margins.
origin()	Set the location of the origin for the bounding box of the plot.
outputType()	Set the plot output type.
plotType()	Set the plot type.
plotting()	Produce a plot from an instance of the **CPlot** class.
point()	Add a point to a plot.
pointType()	Set the point type, size, and color.
polarPlot()	Set a 2D plot to use the polar coordinate system.
polygon()	Add a polygon to a plot.
rectangle()	Add a rectangle to a 2D plot.
removeHiddenLine()	Enable or disable hidden line removal for 3D plots.
scaleType()	Set the axis scale type for a plot.
showMesh()	Enable or disable display of mesh of a 3D plot.
size()	Change the size of a plot.
size3D()	Change the size of a 3D plot.
sizeRatio()	Change the aspect ratio of a plot.
smooth()	Smooth plotting curves by interpolation and approximation of data.
subplot()	Create a group of subplots.
text()	Add a text string to a plot.
tics()	Enable or disable display of axis tics.
ticsDay()	Set axis tic-mark labels to days of the week.
ticsDirection()	Set the direction in which axis tic-marks are drawn.
ticsFormat()	Set the number format for tic labels.
ticsLabel()	Set location and text label for arbitrary axis labels.
ticsLevel()	Set the z-axis offset for drawing of tics in 3D plots.
ticsLocation()	Specify the location of axis tic marks to be on the border or the axis.
ticsMirror()	Enable or disable display of axis tics on the opposite axis.
ticsMonth()	Set axis tic-mark labels to months.
ticsPosition()	Add tic-marks at the specified positions to an axis.
ticsRange()	Specify the range for a series of tics on an axis.
title()	Set the plot title.

```
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;

    lindata(-M_PI, M_PI, x);
    y = sin(x);
    plot.data2D(x, y);
    plot.plotting();
}
```

Program 23.1: A simple program using **CPlot** class.

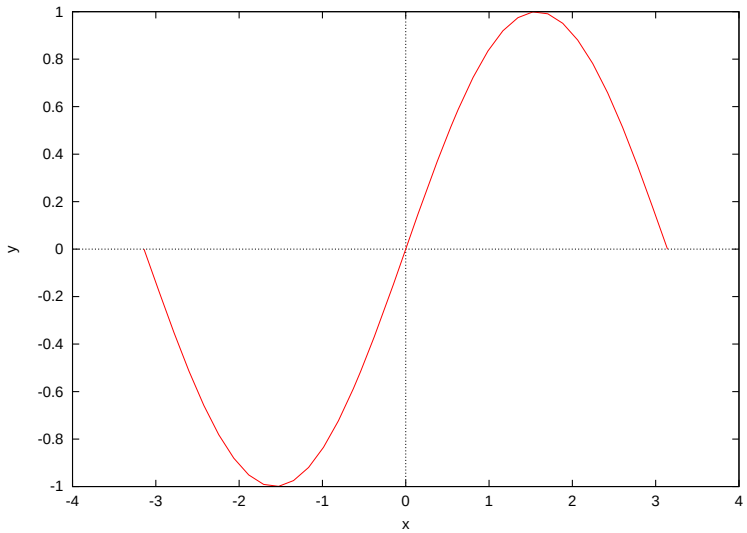


Figure 23.1: A very simple plot.

```
#include <math.h>
#include <chplot.h>

int main() {
    array double x[360], y[360], z[360];
    class CPlot plot;

    lindata(0, 360, x);
    y = sin(x*M_PI/180);
    z = cos(x*M_PI/180);
    plot.data3D(x, y, z);
    plot.plotting();
}
```

Program 23.2: A plotting program for a 3D curve.

x n, if the size of array x is n. In case y is a matrix of m x n, each of the m rows of y is plotted against x. Each of the rows of y is a separate data set. The arrays x and y are of real type. Conversion of the data to double type is performed internally as if function **CPlot**::**data2D**(x, y) were polymorphic. For example, if the declaration for computational arrays x and y in Program 23.1 are changed from double to float data type as follows,

```
array float x[numpoints], y[numpoints];
```

the displayed plot will remain the same. The data for arguments x and y of member function **CPlot**::**data2D**() can also be an expression of computational arrays as shown below.

```
array double complex z[numpoints];
/* ... */
plot.data2D(real(z), imag(z));
```

Data points for array y of value NaN are internally removed before plotting occurs. The "holes" in a data set can be constructed by manually setting elements of y to this value.

The data for plotting of 2D curve can also be added to an instance of **CPlot** class by the member function

```
int CPlot::data2DCurve(double  x[], double y[], int n);
```

Both one-dimensional arrays x and y have the same number of elements of size n.

Similarly, data for 3-dimensional plot can be added to an instance of **CPlot** class by member function

```
int CPlot::data3D(array double x[&], array double y[&],
                  array double &z);
```

For Cartesian data, *x* is a one-dimensional array of size n_x and *y* is a one-dimensional array of size n_y. The array *z* can be of two different sizes depending on what type of data is to be plotted. If the data is for a 3D curve, *z* is a one-dimensional array of size n_z or a two-dimensional array of size m x n_z, with $n_x = n_y = n_z$. Program 23.2 with corresponding plot in Figure 23.2 illustrates how a spatial curve can be generated. If the data is for a 3D surface or grid, *z* is m x n_z, with $n_z = n_x \cdot n_y$. For cylindrical or spherical data *x* is a one dimensional array of size n_x (representing θ), *y* is a one dimensional array of size n_y (representing z or ϕ), and *z* is m x n_z(representing r) where $n_x = n_y = n_z$. Each of the m rows of *z* are plotted against x and y, and correspond to a separate data set. In all cases these data arrays can be of any supported data type for real numbers. Like function **CPlot**::**data2D**(), conversion of the data to type double is performed internally. For

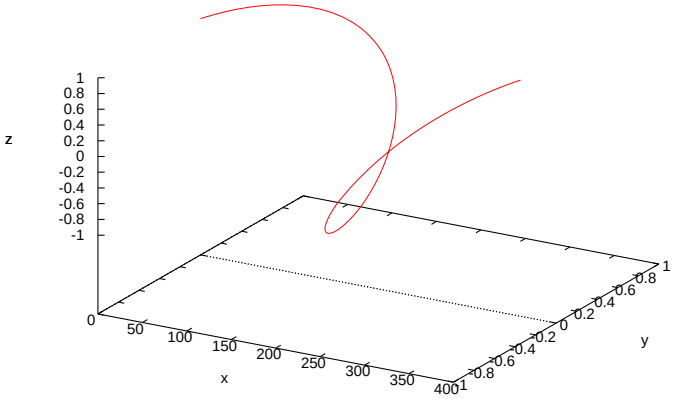


Figure 23.2: A plot with a 3D curve.

a 3D grid, the ordering of the z data is important. For calculation of the z values, the *x* value is held constant while *y* is cycled through its range of values. The *x* value is then incremented and *y* is cycled again. This is repeated until all the data is calculated. So, for a 10x20 grid the data shall be ordered as follows:

x1 y1 z1
x1 y2 z2
.
.
.
x1 y19 z19
x1 y20 z20
x2 y1 z21
x2 y2 z22
.
.
.
x2 y19 z29
x2 y20 z30
x3 y1 z31
x3 y2 z32
.
.
.
x10 y18 z198
x10 y19 z199
x10 y20 z200

A 3D-plot in Figure 23.3 is produced by Program 23.3. Unlike Program 23.2, the number of elements (600) for array `z` in Program 23.3 is the product of the number of elements (20) for array `x` and that (30) for array `y`. The color box with the gradient of the smooth color between the maximum and minimum values of the color palette for a 3D plot can be removed by the member function **CPlot**::**colorBox**().

```
#include <chplot.h>
#include <math.h>

#define NUMX 20
#define NUMY 30
int main() {
    double x[NUMX], y[NUMY], z[NUMX*NUMY];
    double r;
    int i, j;
    class CPlot plot;

    lindata(-10, 10, x);
    lindata(-10, 10, y);
    for(i=0; i<NUMX; i++) {
        for(j=0; j<NUMY; j++) {
            r = sqrt(x[i]*x[i]+y[j]*y[j]);
            z[30*i+j] = sin(r)/r;
        }
    }
    plot.data3D(x, y, z);
    plot.plotting();
}
```

Program 23.3: A plotting program for a 3D grid.

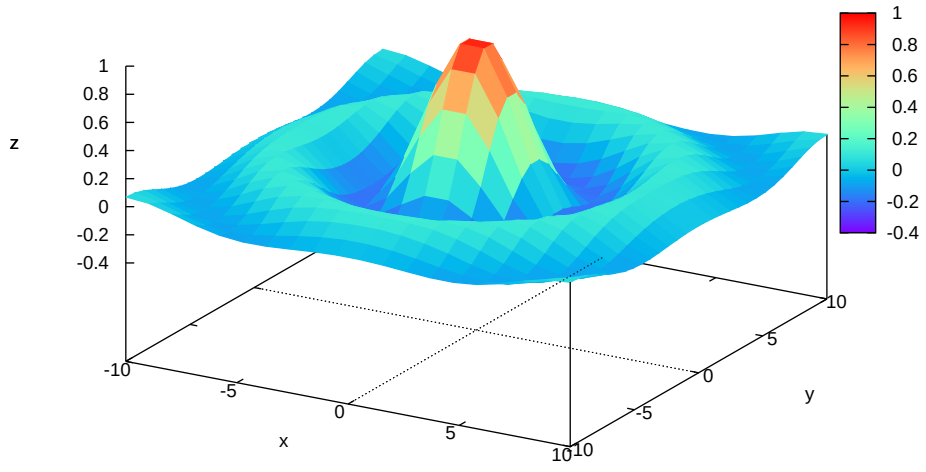


Figure 23.3: A plot with a 3D grid.

```
#include <stdio.h>
#include <chplot.h>
#include <math.h>

int main() {
    string_t filename;
    int i;
    class CPlot plot;
    FILE *out;

    filename = tmpnam(NULL);       //Create a temporary file.
    out=fopen (filename,"w");      //Write data to the file.
    for (i=-180;i<=180;i++)
       fprintf(out,"%i %f \n",i,sin(i*M_PI/180));
    fclose(out);
    plot.dataFile(filename);
    plot.plotting();
    remove(filename);
}
```

Program 23.4: A plotting program using data from a file.

The data for plotting of 3D curve can also be added to an instance of **CPlot** class by the member function

```
int CPlot::data3DCurve(double  x[], double y[], double z[], int n);
```

One-dimensional arrays `x`, `y`, and `z` have the same number of elements of size `n`. A set of data for 3D surface plotting can be added to an instance of **CPlot** class by the member function

```
int CPlot::data3DSurface(double  x[], double y[], double z[],
                         int n, int m);
```

If one-dimensional array `x` has the number of elements of size n, and *y* has size m, *z* shall be a one-dimensional array of size $n_z = n \cdot m$. In a Cartesian coordinate system, arrays *x*, *y*, and *z* represent values in X-Y-Z coordinates, respectively. In a cylindrical coordinate system, arrays *x*, *y*, and *z* represent θ, z, and r coordinates, respectively. In a spherical coordinate system, arrays *x*, *y*, and *z* represent θ), ϕ, and r coordinates, respectively.

The data for plotting can also be stored in a file first and then obtained by function

```
int CPlot::dataFile(string_t filename, ... /* string_t option */);
```

Each data file corresponds to a single data set. The data file should be formatted with each data point on a separate line. 2D data is specified by two values per point. An empty line in a 2D data file causes a break of the curve in the plot. Multiple curves can be plotted in this manner, however the plot style will be the same for all curves. The symbol # will comment out a subsequent text terminated at the end of a line in a data file. For example, Program 23.4 will generate a plot shown in Figure 23.1.

3D data is specified by three values per data point. For 3D grid or surface data, each row is separated in the data file by a blank line. For example, a 3 x 3 grid would be represented as follows:

Table 23.2: The macros for axes.

PLOT_AXIS_X	Select the x axis only.
PLOT_AXIS_X2	Select the x2 axis only.
PLOT_AXIS_Y	Select the y axis only.
PLOT_AXIS_Y2	Select the y2 axis only.
PLOT_AXIS_Z	Select the z axis only.
PLOT_AXIS_XY	Select the x and y axes.
PLOT_AXIS_XYZ	Select the x, y, and z axes.

```
# This is a comment line
x1   y1   z1
x1   y2   z2
x1   y3   z3

x2   y1   z4
x2   y2   z5
x2   y3   z6

x3   y1   z7
x3   y2   z8
x3   y3   z9
```

Two empty lines in the data file will cause a break in the plot. Multiple curves or surfaces can be plotted in this manner, however, the plot style will be the same for all curves or surfaces. Member function **CPlot**::**dimension**() with the value of 3 as the argument must be called before a 3D data file can be added.

23.1.2 Annotations

A plot can be annotated with a title and labels on axes using corresponding member functions

```
void CPlot::title(string_t title);
```

and

```
void CPlot::label(int axis, string_t label);
```

respectively. The argument *axis* of member function **CPlot**::**label**() is the axis to be set. The valid macros for *axis* are listed in Table 23.2. Figure 23.4 displays the plot produced by Program 23.5 using member functions **CPlot**::**title**() and **CPlot**::**label**(). By default, no title is displayed and the coordinate axes are labeled with symbols x, y, and z.

Program 23.6 demonstrates how arrow, text, axis limits, grid, border, and axis are handled using member functions of **CPlot** class. Figure 23.5 displays the plot produced by Program 23.6. In Program 23.6, the axis limits are set by member function **CPlot**::**axisRange**(),

```
void CPlot::axisRange(int axis, double minimum, double maximum);
```

The valid macros for *axis* are listed in Table 23.2. Each of four borders x (bottom), x2 (top), y (left), and y2 (right) can be used as an independent axis. The minimum and maximum values for an axis are given in second and third arguments, respectively. The tic marks on an axis can be set by the function

```
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;
    string_t title="Sine Wave",
             xlabel="degree",
             ylabel="amplitude";

    lindata(0, 360, x);
    y = sin(x*M_PI/180);
    plot.data2D(x, y);
    plot.title("Sine Wave");
    plot.label(PLOT_AXIS_X, xlabel);
    plot.label(PLOT_AXIS_Y, ylabel);
    plot.plotting();
}
```

Program 23.5: A plotting program with annotation.

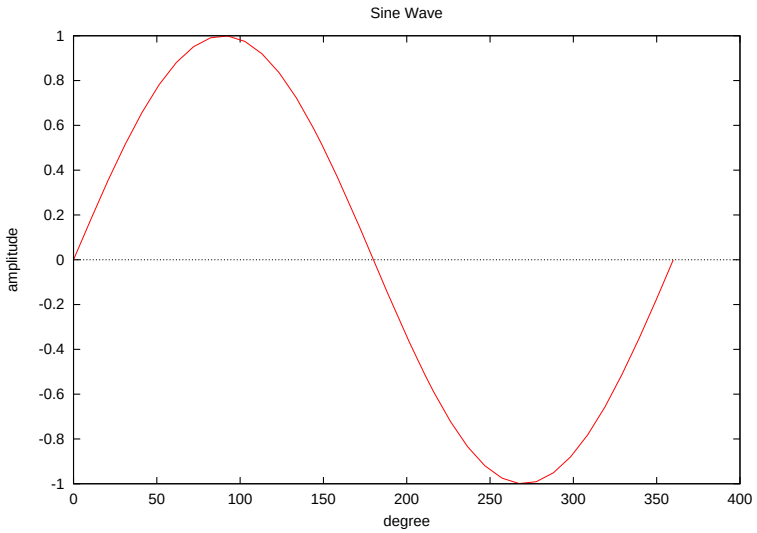


Figure 23.4: A plot with annotation.

Table 23.3: The macros for border locations.

PLOT_BORDER_BOTTOM	The bottom of the plot.
PLOT_BORDER_LEFT	The left side of the plot.
PLOT_BORDER_TOP	The top of the plot.
PLOT_BORDER_RIGHT	The right side of the plot.
PLOT_BORDER_ALL	All sides of the plot.

```
void CPlot::ticsRange(int axis, double incr, ...
                      /* [double start], [double end] */);
```

The increment between tic marks is given in *incr*. By default, this value is calculated internally. The start and end positions for tic marks are optional arguments. For example, function calls

```
plot.axisRange(PLOT_AXIS_X, 0, 360);
plot.ticsRange(PLOT_AXIS_X, 30, 0, 360);
```

set the range of the `x`-axis from 0 to 360 degrees with tic marks at every 30 degrees.

Member function **CPlot**::**axes**(),

```
void CPlot::axes(int num, char *axes);
```

lets you choose which pair of axes a given set of data specified in `num` is plotted against. There are four possible sets of axes available. The argument `axes` is used to select the axes for which a particular line should be scaled. The string `"x1y1"` refers to the axes on the bottom and left; `"x2y2"` to those on the top and right; `"x1y2"` to those on the bottom and right; and `"x2y1"` to those on the top and left.

Drawing the `x` and `y` axes on a 2D plot can be enabled or disabled using member function

```
void CPlot::axis(int axis, int flag);
```

The valid macros for *axis* are the same as those for other member functions. The *flag* can be set to **PLOT_ON** to enable the drawing of the specified axis, or **PLOT_OFF** to disable the drawing of the specified axis. In Program 23.5, the drawing of `x` and `y` axes is disabled at the same time by using function call `plot.axis(PLOT_AXIS_XY, PLOT_OFF)`. Member function

```
void CPlot::border(int location, int flag);
```

turns a border display around the plot on or off. By default, the border is drawn on the left and bottom sides for 2D plots, and on all sides on the x-y plane for 3D plots. The valid *location* for function **CPlot**::**border**() is given in Table 23.3. Figure 23.5 is generated with borders on four sides by function call `CPlot::border(PLOT_BORDER_ALL, PLOT_ON).` The display of a grid on the x-y plane can be enabled or disabled by member function

```
void CPlot::grid(int flag, ... /* char *option */);
```

The *flag* can be set to **PLOT_ON** to enable or **PLOT_OFF** to disable the display of the grid. For a polar plot, a polar grid will be drawn. Otherwise, the grid is rectangular. By default, the grid is not displayed. A plot can be annotated with arrows by function

```
void CPlot::arrow(double x_head, y_head, z_head, x_tail, y_tail,
                  z_tail, ... /* char *option */);
```

Table 23.4: The macros for text locations.

PLOT_TEXT_LEFT	The left side of the text string.
PLOT_TEXT_RIGHT	The right side of the text string.
PLOT_TEXT_CENTER	The center of the text string.

where (*x_head, y_head, z_head*) and (*x_tail, y_tail, z_tail*) are the coordinates of the head and tail of an arrow, respectively. The arrow points from (*x_tail*, *y_tail*, *z_tail*) to (*x_head*, *y_head*, *z_head*). These coordinates are specified using the same coordinate system as the curves of the plot. An optional argument can be used to specify other attributes of an arrow. The annotation of text on a plot is achieved by member function

```
void CPlot::text(string_t string, int just,
                 double x, double y, double z);
```

where text *string* is placed at location (*x*,*y*) for 2D plots or (*x*,*y*,*z*) for 3D plots. The location of the text is measured in the plot coordinate system. The position of the text is adjusted by the argument *just*. The valid macros for argument *just* are given in Table 23.4. In Figure 23.5, the tail of the arrow is the location for the text `testing text` left adjusted using functions **CPlot**::**arrow**() and **CPlot**::**text**().

Additional features such as different tic marks and scales for data can be found in the reference for **CPlot** class.

23.1.3 Multiple Data Sets and Legends

A plot with multiple sets of data can be produced as shown in Figure 23.7. Figure 23.7 with legends can be generated by either Program 23.7 or 23.8. Program 23.7 is semantically the same as Program 23.8. In Program 23.8, array `y` is two-dimensional for two sets of data and member function **CPlot**::**data2D**() is only called once for adding the data to the plot. A string of *legend* can be added to the plot by member function

```
void CPlot::legend(string_t legend, int num);
```

The number of data set to which the legend is added is indicated by the second argument *num*. Numbering of the data sets starts with zero. New legends will replace previously specified legends. This member function shall be called after plotting data have been added by member functions **CPlot**::**data2D**(), **CPlot**::**data2DCurve**(), **CPlot**::**data3D**(), **CPlot**::**data3DCurve**(), **CPlot**::**data3DSurface**(), or **CPlot**::**dataFile**(). The member function

```
void CPlot::legendLocation(double x, double y, ... /* [double z] */);
```

specifies the position of the plot legend using plot coordinates *(x, y, z)*. The position specified is the location of the top right of the box for the markers and labels of the legend as shown in Figure 23.6. By default, the location of the legend is near the upper-right corner of the plot.

A 3D plot with multiple sets of data can be produced similarly. The 3D plot in Figure 23.8 can be generated by either Program 23.9 or 23.10. Program 23.9 is semantically the same as Program 23.10. In Program 23.9, array `z` is two-dimensional for two sets of data, so that member function **CPlot**::**data3D**() for adding the data to the plot is only called once instead of twice in Program 23.10. Program 23.11 demonstrates how to superimpose a curve on a surface with output shown in Figure 23.9. Because no hidden lines can be removed from non grid data, the hidden line removal is turned off by member function **CPlot**::**removeHiddenLine**().

```
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 36;
    array double x[numpoints], y[numpoints];
    class CPlot plot;
    string_t title="Sine Wave",
             xlabel="degree",
             ylabel="amplitude";
    double x1=180, y1=0.0,  z1=0;
    double x2=225, y2=0.1,  z2=0;

    lindata(0, 360, x);
    y = sin(x*M_PI/180);
    plot.data2D(x, y);
    plot.title("Sine Wave");
    plot.label(PLOT_AXIS_X, xlabel);
    plot.label(PLOT_AXIS_Y, ylabel);
    plot.axisRange(PLOT_AXIS_X, 0, 360);
    plot.ticsRange(PLOT_AXIS_X, 30, 0, 360);
    plot.axisRange(PLOT_AXIS_Y, -1, 1);
    plot.ticsRange(PLOT_AXIS_Y, .25, -1, 1);
    plot.axis(PLOT_AXIS_XY, PLOT_OFF);
    plot.border(PLOT_BORDER_ALL, PLOT_ON);
    plot.grid(PLOT_ON);
    plot.arrow(x1, y1, z1, x2, y2, z2);
    plot.text("inflection point", PLOT_TEXT_LEFT, x2, y2, z2);
    plot.plotting();
}
```

Program 23.6: A plotting program with many features.

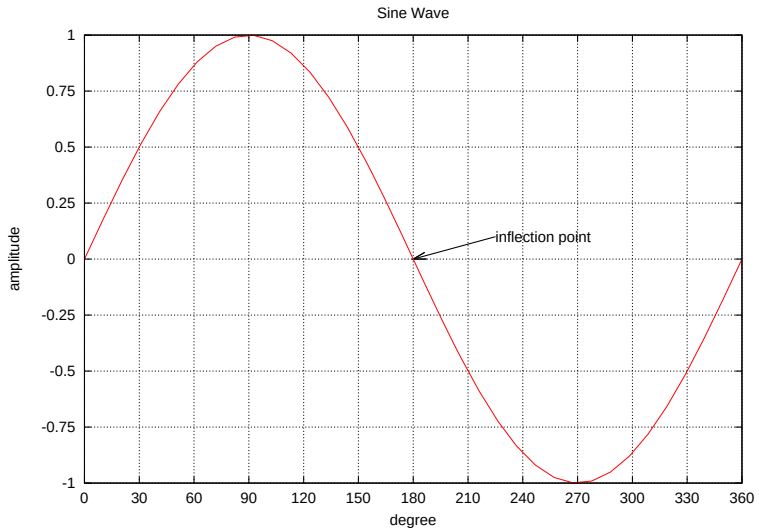


Figure 23.5: A plot with many features.

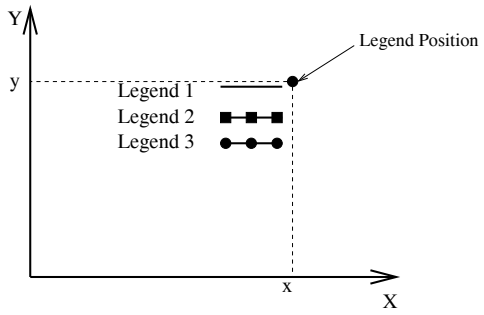


Figure 23.6: The position for legend.

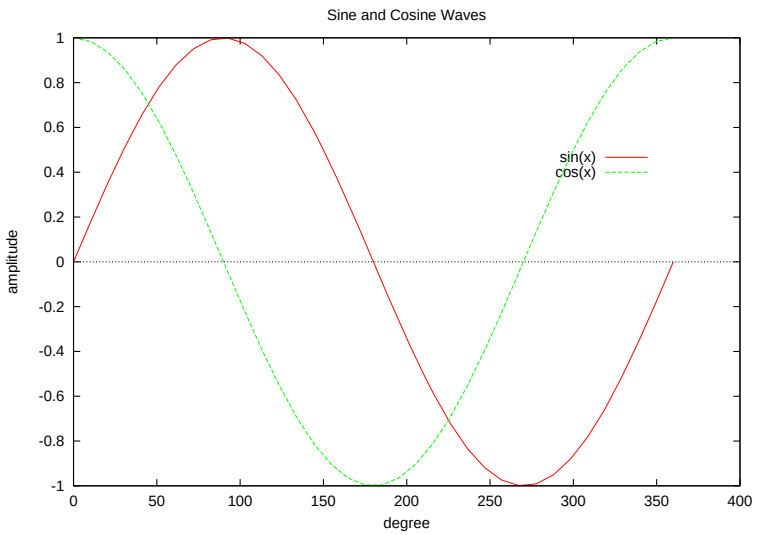


Figure 23.7: A plot with two sets of data, title, labels, and legends.

```
#include<math.h>
#include<chplot.h>

int main() {
  int numpoints = 36;
  array double x1[numpoints], y1[numpoints];
  array double x2[numpoints], y2[numpoints];
  string_t title="Sine and Cosine Waves",
         xlabel="degree",
         ylabel="amplitude";
  class CPlot plot;

  lindata(0, 360, x1);
  lindata(0, 360, x2);
  y1 = sin(x1*M_PI/180);
  y2 = cos(x2*M_PI/180);
  plot.data2D(x1, y1);
  plot.data2D(x2, y2);
  plot.legend("sin(x)", 0);
  plot.legend("cos(x)", 1);
  plot.legendLocation(350, 0.5);
  plot.title(title);
  plot.label(PLOT_AXIS_X, xlabel);
  plot.label(PLOT_AXIS_Y, ylabel);
  plot.plotting();
}
```

Program 23.7: A program for plotting two functions on the same plot with title, labels, and legends.

```
#include<math.h>
#include<chplot.h>

int main() {
  int i, numdataset = 2, numpoints = 36;
  array double x[numpoints], y[numdataset][numpoints];
  string_t title="Sine and Cosine Waves",
         xlabel="degree",
         ylabel="amplitude";
  class CPlot plot;

  lindata(0, 360, x);
  for(i = 0; i < numpoints; i++) {
    y[0][i] = sin(x[i]*M_PI/180);
    y[1][i] = cos(x[i]*M_PI/180);
  }
  plot.data2D(x, y);
  plot.legend("sin(x)", 0);
  plot.legend("cos(x)", 1);
  plot.legendLocation(350, 0.5);
  plot.title(title);
  plot.label(PLOT_AXIS_X, xlabel);
  plot.label(PLOT_AXIS_Y, ylabel);
  plot.plotting();
}
```

Program 23.8: Another program for plotting two functions on the same plot with title, labels, and legends.

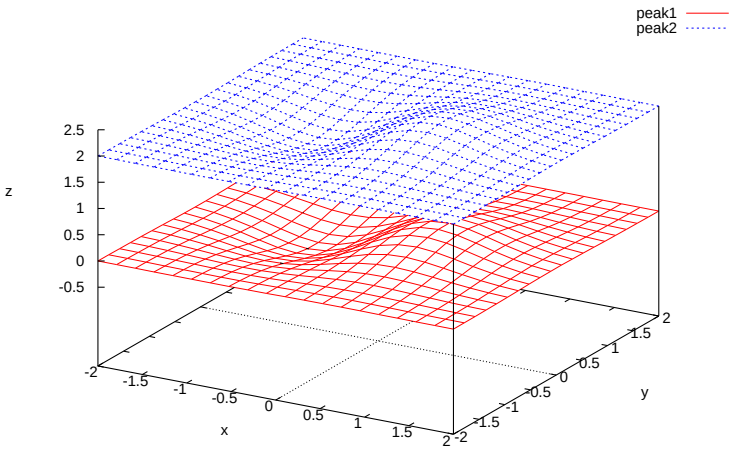


Figure 23.8: A 3D plot with two sets of data.

```
#include <chplot.h>
#include <math.h>

#define NUMX 20
#define NUMY 20
#define NUMCURVE 2
int main() {
    array double x[NUMX], y[NUMY], z[NUMCURVE][NUMX*NUMY];
    int datasetnum =0, i, j;
    class CPlot plot;

    lindata(-2, 2, x);
    lindata(-2, 2, y);
    for (i=0; i<NUMX; i++) {
        for(j=0; j<NUMY; j++) {
            z[0][i*NUMX+j] = x[i]*exp(-x[i]*x[i]-y[j]*y[j]);
            z[1][i*NUMX+j] = z[0][i*NUMX+j] +2;
        }
    }
    plot.data3D(x, y, z);
    plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum++);
    plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum);
    plot.legend("peak1", 0);
    plot.legend("peak2", 1);
    plot.colorBox(PLOT_OFF);
    plot.plotting();
}
```

Program 23.9: A program for plotting two functions on the same 3D plot.

```
#include <chplot.h>
#include <math.h>

#define NUMX 20
#define NUMY 20
#define NUMCURVE 2
int main() {
    array double x[NUMX], y[NUMY], z1[NUMX*NUMY], z2[NUMX*NUMY];
    int datasetnum =0, i, j;
    class CPlot plot;

    lindata(-2, 2, x);
    lindata(-2, 2, y);
    for (i=0; i<NUMX; i++) {
        for(j=0; j<NUMY; j++) {
            z1[i*NUMX+j] = x[i]*exp(-x[i]*x[i]-y[j]*y[j]);
            z2[i*NUMX+j] = z1[i*NUMX+j] +2;
        }
    }
    plot.data3D(x, y, z1);
    plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum++);
    plot.data3D(x, y, z2);
    plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum);
    plot.legend("peak1", 0);
    plot.legend("peak2", 1);
    plot.colorBox(PLOT_OFF);
    plot.plotting();
}
```

Program 23.10: Another program for plotting two functions on the same 3D plot.

```
#include <chplot.h>
#include <math.h>

#define NUMX 20
#define NUMY 20
#define NUMCURVE 2
#define NUM  20
int main() {
    array double x[NUMX], y[NUMY], z[NUMCURVE][NUMX*NUMY];
    array double x0[NUM], y0[NUM], z0[NUM];
    int datasetnum=0, i, j, linetype, linewidth;
    class CPlot plot;

    lindata(-2, 2, x);
    lindata(-2, 2, y);
    lindata(-2, 2, x0);
    y0 = (array double [NUM])-1;
    for (i=0; i<NUMX; i++) {
        for(j=0; j<NUMY; j++) {
            z[0][i*NUMY+j] = x[i]*exp(-x[i]*x[i]-y[j]*y[j]);
            z[1][i*NUMY+j] = z[0][i*NUMY+j] +2;
        }
    }
    for (i=0; i<NUM; i++)
        z0[i] = x0[i]*exp(-x0[i]*x0[i]-y0[i]*y0[i]);
    plot.data3D(x, y, z);
    plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum++);
    plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum++);
    plot.data3D(x0, y0, z0);
    plot.legend("peak1", 0);
    plot.legend("peak2", 1);
    linetype = 5;
    linewidth = 2;
    plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum);
    plot.lineType(datasetnum, linetype, linewidth);
    plot.legend("curve", datasetnum);
    plot.removeHiddenLine(PLOT_OFF);
    plot.colorBox(PLOT_OFF);
    plot.plotting();
}
```

Program 23.11: A program superimposing a curve on a surface.

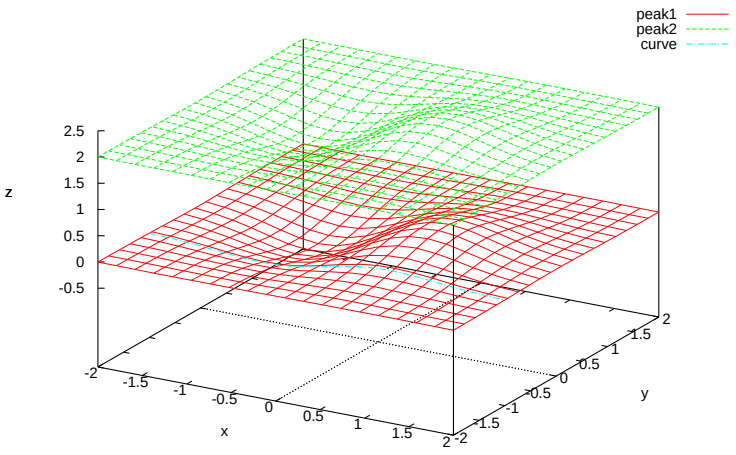


Figure 23.9: A 3D plot with a curve superimposed on a surface.

23.1.4 Using Predefined Geometric Primitives

For the user's convenience, several geometric primitives such as line, circle, rectangle, polygon, are predefined as member functions of the plotting class.

A line can be drawn using function

```
int CPlot::line(double x1, double y1, double z1, double x2,
                double y2, double z2);
```

For 2D rectangular and 3D cartesian plots, (*x1*, *y1*, *z1*) and (*x2*, *y2*, *z2*) are the coordinates of the endpoints of the line, specified in units of the x, y, and z axes. However, for 2D plots, *z1* and *z2* are ignored. For 2D polar and 3D cylindrical plots, the endpoints are specified in polar coordinates where *x* is θ, *y* is r, and *z* is unchanged. Again, for 2D plots, *z1* and *z2* are ignored. For 3D plots with spherical coordinates *x* is θ, *y* is ϕ and *z* is r. Function

```
int CPlot::circle(double x, double y, double r);
```

adds a circle to a 2D plot. For rectangular plots, *x* and *y* are the coordinates of the center of the circle and *r* is the radius of the circle, all specified in units of the x and y axes. For polar plots, the location of the center of the circle is specified in polar coordinates where *x* is θ and *y* is r. Function

```
int CPlot::rectangle(double x, double y,
                     double width, double height);
```

adds a rectangle to a 2D plot. For rectangular plots, *x* and *y* are the coordinates of the lower-left corner of the rectangle. For polar plots, the coordinates of the lower-left corner are given in polar coordinates where *x* is theta and *y* is r. In both cases the *width* and *height* are the dimensions of the rectangle in rectangular coordinates. Function

```
int CPlot::polygon(double x[:], double y[:], double z[:]);
```

adds a polygon to a plot. For 2D rectangular plots and 3D cartesian plots, *x*, *y*, and *z* contain the polygon vertices specified in units of the x, y, and z axes. However, for 2D plots, *z* is ignored. For 2D polar and 3D

```
/* expx.ch */
#include <chplot.h>

int main() {
    array double x1[100], f1[100];
    array double x2[100], f2[100];
    CPlot plot;

    lindata(0.5, 10, x1);
    f1=exp((array double [100])1.0./x1); // f1=exp(1.0./x1);
    lindata(x2, -10, -0.5);
    f2=exp((array double [100])1 ./x2); // f1=exp(1 ./x1);
    plot.label(PLOT_AXIS_X, "x");
    plot.label(PLOT_AXIS_Y, "exp(1/x)");
    plot.data2D(x1, f1);
    plot.data2D(x2, f2);
    plot.line(-10, 1, 0, 10, 1, 0);
    plot.plotType(PLOT_PLOTTYPE_LINES, 2, 3, 0);
    plot.line(0, 0, 0, 0, 8, 0);
    plot.plotType(PLOT_PLOTTYPE_LINES, 3, 3, 0);
    plot.plotting();
    return 0;
}
```

Program 23.12: Plot function $e^{1/x}$.

cylindrical plots, the locations of the vertices are specified in polar coordinates where *x* is θ, *y* is r, and *z* is unchanged. Again, for 2D plots, *z* is ignored. For 3D plots with spherical coordinates *x* is θ, *y* is ϕ and *z* is r. Each of the points is connected to the next in a closed chain.

A geometric primitive added by these member functions is counted as a data set for later calls to **CPlot**::**legend**() and **CPlot**::**plotType**(). As an example, the function $f(x) = e^{\frac{1}{x}}$ described in section 12.2.2 is not continuous at the origin as shown in Figure 12.1 on page 219, which is generated by Program 23.12. The program draws two curves for function $f(x) = e^{\frac{1}{x}}$. The data sets of these two curves are calculated using type generic mathematical function **exp**() with input argument of computational array type. Two lines, one horizontal and one vertical at the point (0, 1) are drawn using geometric primitive member function **CPlot**::**line**(). The line type is specified by member function **CPlot**::**plotType**().

23.1.5 Subplots

Multiple plots can be displayed in the same figure and printed on the same piece of paper using function

```
int CPlot::subplot(int row, int col);
```

Function **CPlot**::**subplot**() breaks the figure into an `m`-by-`n` matrix of small subplots. The subplots are numbered as if there were a 2-dimensional matrix with the numbers of rows and columns specified in its arguments. Each index starts with 0. A pointer to **CPlot** class as a handle for subplot at location `(i,j)` can be obtained by function

```
class CPlot* CPlot::getSubplot(int row, int col);
```

where *row* and *col* are the row and column numbers of the desired subplot element, respectively. Numbering starts with zero. For example, Program 23.13 breaks a plot with four subplots in a 2-by-2 matrix. Each subplot can be annotated with title, label, etc. as if it were a separate plot. Figure 23.10 displays the plot

produced by Program 23.13. In Program 23.13, the plotting data for function $\sin(x)/(x)$ in subplot located at (1, 1) is generated without using iterative loops such as for-loop or while-loop. The array operator `./` for element-wise division of two arrays is applied in the program. To avoid division by zero, a small floating-point value `DBL_EPSILON` is used.

23.1.6 Export and Zoom Plots

A plot not only can be displayed in a terminal screen, but also be exported in a variety of formats for various applications. Different output types can be achieved by member function

```
void CPlot::outputType(int outputtype, ...,
            /* [string_t terminal, string_t filename] */);
```

The argument *outputtype* can be one of the following macros **PLOT_OUTPUTTYPE_DISPLAY**, **PLOT_OUTPUTTYPE_STREAM**, and **PLOT_OUTPUTTYPE_FILE**. The output type **PLOT_OUTPUTTYPE_DISPLAY** displays the plot on the screen. The plot is displayed in its own separate window. A plot window can be closed by pressing the 'q' key in Unix. By default, the output type is **PLOT_OUTPUTTYPE_DISPLAY**. For the output type **PLOT_OUTPUTTYPE_STREAM**, the output from the plot engine is a standard output stream. This output type is useful when a Ch program is used as a CGI script in a Web server to generate a plot dynamically as a standard output stream in a png or gif file format in Web browser displays. For **PLOT_OUTPUTTYPE_FILE**, a plot can be saved in files in a variety of different formats that can be controlled by two optional arguments *terminal* and *filename*. The supported terminal types are listed in Table 23.5. Some terminals can have additional parameters such as size and color of a plot as part of the string for the argument *terminal*. Details for each terminal are given in the chapter about plotting in *The Ch Language Environment — Reference Guide*. The last optional argument *filename* is a string containing a file name to which the plot is saved. On machines that support pipes, the output can also be piped to another program by placing the '`|`' character in front of the command name and using it as the *filename*. For example, on Unix systems, setting *terminal* to "`postscript`" and *filename* to "`|lp`" could be used to send a plot directly to a postscript printer.

Program 23.14 shows how to export a plot in the formats of encapsulated postscript, latex, pbm, gif, and png formats. The 2D plotting function **plotxy**() in Program 23.14 will be explained in detail in section 23.2.3.

In Windows, a plot displayed on screen can be copied to a clipboard through a menu on the upper left corner of the plot. It then can be pasted in other Windows application programs such as Word, a word processor from Microsoft. To zoom in a portion of the plot, right click the mouse to select the upper left corner, drag the mouse to a lower right corner and then right click to select the area to zoom in. Afer zoom in, press the key 'p' to zoom out and 'u' to restore the original plot.

23.1.7 Print Plots

Printing Plots in Windows

One of the following two methods can be used to print a plot in Windows.
Method 1
Step 1. Run a Ch program with plotting, click the upper left corner of the window with plot.
Step 2. Select "print" from the options menu, configure, and print accordingly.

Method 2
Step 1. Run a Ch program with plotting, click the upper left corner of the window with plot.

```
#include <float.h>
#include <math.h>
#include <chplot.h>

#define NUM1 36
#define NUM2 101
#define NUMX 20
#define NUMY 30
int main() {
    array double x[NUM1], y[NUM1];
    double x3[NUMX], y3[NUMY], z3[NUMX*NUMY], r;
    array double x4[NUM2], y4[NUM2];
    int i, j;
    class CPlot subplot, *plot;

    lindata(-M_PI, M_PI, x);
    y = sin(x);
    subplot.subplot(2, 2);
    plot = subplot.getSubplot(0, 0);
    plot->data2D(x, y);

    plot = subplot.getSubplot(0, 1);
    plot->data2D(x, y);
    plot->axisRange(PLOT_AXIS_Y, -1, 1);
    plot->ticsRange(PLOT_AXIS_Y, 0.25, -1, 1);
    plot->grid(PLOT_ON);

    lindata(-20, 20, x4);
    x4 = x4+(x4==0)*DBL_EPSILON; /* if x4==0, x4 becomes epsilon */
    y4 = sin(x4)./(x4);
    plot = subplot.getSubplot(1, 0);
    plot->data2D(x4, y4);
    plot->label(PLOT_AXIS_Y, "sin(x)/x");

    lindata(-10, 10, x3);
    lindata(-10, 10, y3);
    for(i=0; i<NUMX; i++) {
        for(j=0; j<NUMY; j++) {
            r = sqrt(x3[i]*x3[i]+y3[j]*y3[j]);
            z3[NUMY*i+j] = sin(r)/r;
        }
    }
    plot = subplot.getSubplot(1, 1);
    plot->data3D(x3, y3, z3);
    plot->colorBox(PLOT_OFF);

    subplot.plotting();
}
```

Program 23.13: A plotting program with subplots.

Table 23.5: The terminal types of plot output type.

Terminal	Description
aifm	Adobe Illustrator 3.0.
corel	EPS format for CorelDRAW.
dxf	AutoCAD DXF.
dxy800a	Roland DXY800A plotter.
eepic	Extended LaTeXpicture.
emtex	LaTeXpicture with emTeX specials.
epson-180dpi	Epson LQ-style 24-pin printer with 180dpi.
epson-60dpi	Epson LQ-style 24-pin printers with 60dpi.
epson-lx800	Epson LX-800, Star NL-10 and NX-100.
excl	Talaris printers.
fig	Xfig 3.1.
gif	GIF file format.
gpic	gpic/groff package.
hp2648	Hewlett Packard HP2647 an HP2648.
hp500c	Hewlett Packard DeskJet 500c.
hpdj	Hewlett Packard DeskJet 500.
hpgl	HPGL output.
hpljii	HP LaserJet II.
hppj	HP PaintJet and HP3630 printers.
latex	LaTeXpicture.
mf	MetaFont.
mif	Frame Maker MIF 3.00.
nec-cp6	NEC CP6 and Epson LQ-800.
okidata	9-pin OKIDATA 320/321 printers.
pcl5	Hewlett Packard LaserJet III.
pbm	Portable BitMap.
png	Portable Network Graphics.
postscript	Postscript.
pslatex	LaTeXpicture with postscript specials.
pstricks	LaTeXpicture with PSTricks macros.
starc	Star Color Printer.
tandy-60dpi	Tandy DMP-130 series printers.
texdraw	LaTeXtexdraw format.
tgif	TGIF X-Window drawing format.
tpic	LaTeXpicture with tpic specials.

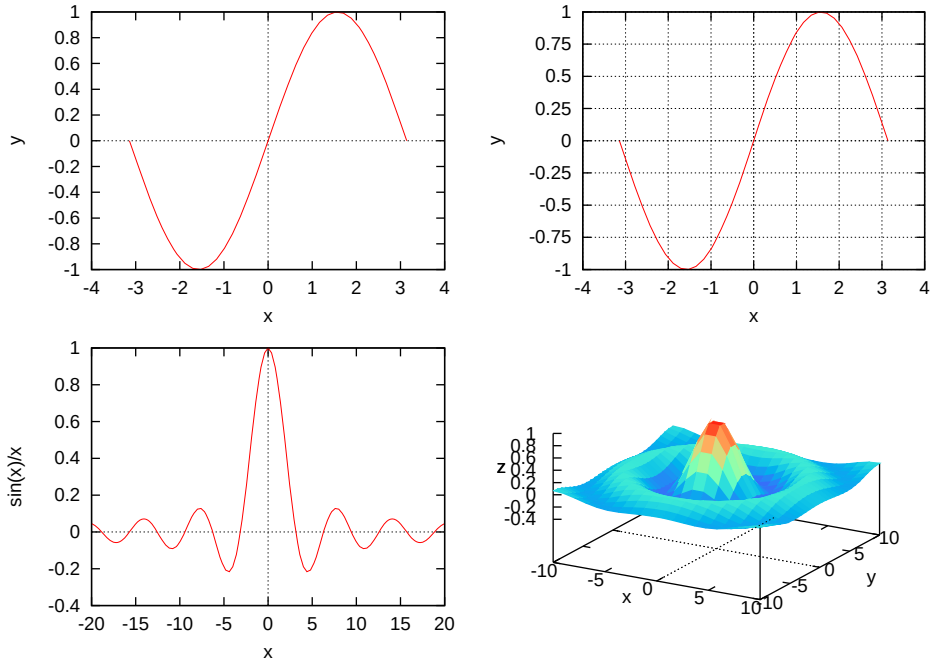


Figure 23.10: A plot with subplots.

Step 2. Select "copy to clipboard" from the options menu.
Step 3. Open Painbrush in **Start**− >**Accessories**.
Step 4. Paste the plot by clicking "paste" from the edit menu or using the key combination `<Ctrl><V>`.
Step 5. Save the plot as a bmp file.
Step 6. Print the plot.

Printing Plots in Unix

In Unix, first, one can save a plot in a file according to the terminal type described in the previous section. Then, print it out. For example, for a postscript printer, a plot can first be saved as a color encapsulated postscript file named **filename.eps** by function `plot.outputType(PLOT_OUTPUTTYPE_FILE, "postscript eps color", "filename.eps").` The postscript file **filename.eps** can then be printed out by command **lp**. Alternatively, the plot can be printed out by setting the output type of the plot using function call `plot.outputType(PLOT_OUTPUTTYPE_FILE, "postscript eps color", "| lp").`

23.2 2D Plotting

The features presented in the previous sections can be applied to both 2D and 3D plotting. This section will describe features specific to 2D plotting only.

23.2.1 Plot Types, Line Styles, and Markers

Different plot types can be selected by function

```
#include<math.h>
#include<chplot.h>

int main() {
  int numpoints = 36;
  array double x[numpoints], y[numpoints];
  string_t title="Sine Wave",                      // Define labels.
         xlabel="degree",
         ylabel="amplitude";
  class CPlot plot;

  lindata(0, 360, x);
  y = sin(x*M_PI/180);
  plotxy(x,y,title,xlabel,ylabel,&plot);

  /* create a postscript file */
  plot.outputType(PLOT_OUTPUTTYPE_FILE, "postscript eps color", "demo.eps");
  plot.plotting();

  /* create a latex file */
  plot.outputType(PLOT_OUTPUTTYPE_FILE, "latex roman 11", "demo.tex");
  plot.plotting();

  /* create a pbm file */
  plot.outputType(PLOT_OUTPUTTYPE_FILE, "pbm", "demo.pbm");
  plot.plotting();

  /* create a gif file */
  plot.outputType(PLOT_OUTPUTTYPE_FILE, "gif", "demo.gif");
  plot.plotting();

  /* create a png file */
  plot.outputType(PLOT_OUTPUTTYPE_FILE, "png", "demo.png");
  plot.plotting();
}
```

Program 23.14: A program for exporting plot.

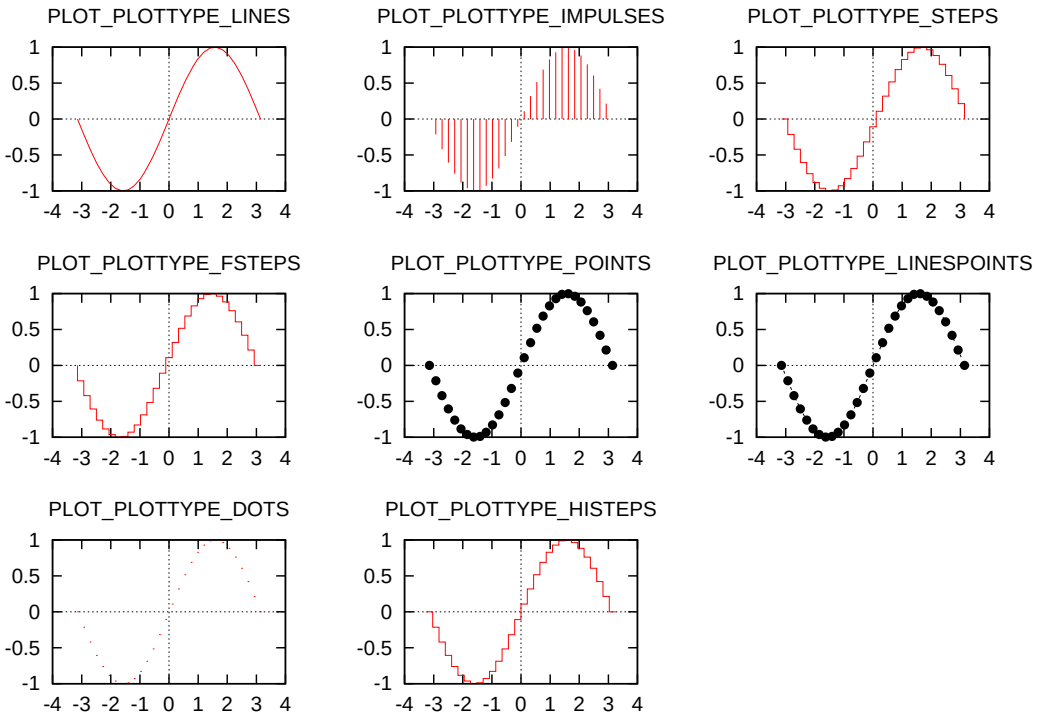


Figure 23.11: Two-dimensional plot types.

```
void CPlot::plotType(int plot_type, int num, ...
    /* [string_t option]*/);
```

Function **CPlot**::**plotType**() sets the desired plot type for a data set to be plotted. The valid macros for argument *plot_type* are given in Table 23.6 with some corresponding plots shown in Figure 23.11. By default, a 2D plot uses plot type **PLOT_PLOTTYPE_LINES**. Data sets in the same plot can have different plot types. The argument *num* indicates the data set to which the plot type is applied. Numbering of the data sets starts with zero. New plot types replace previously specified types.

The line type, width, and color for lines, impulses, steps, etc. can be set by function

```
void lineType(int num, int line_type, int line_width, ...
              /* [char *line_color] */);
```

Function **CPlot**::**lineType**() sets the desired line style for a data set to be plotted. The line style and/or marker type for the plot are selected automatically. The *line_type* specifies an index for the line type used for drawing the line. The line type varies depending on the terminal type used (see **CPlot**::**outputType**). Typically, changing the line type will change the color of the line when the plot is display, Changing the line type makes it dashed, dotted, or other shape when the plot is saved as a postscript file. All terminals support at least six different line types. By default, the line type is 1. The *line_width* specifies the line width. The line width is *line_width* multiplied by the default width. Typically the default width is one pixel. An optional fourth argument can specify the color of a line by a color name or RGB value, such as `"blue"` or `"#0000ff"` for color blue.

Program 23.15 illustrates how different line types are used. The plot displayed in Windows is shown in Figure 23.12. Line type is typically associated with a color. Program 23.16 illustrate how to specify colors of lines inside a program. Figure 23.13 shows the generated plot in the postscript file format.

Table 23.6: The macros for 2D plot types.

PLOT_PLOTTYPE_BOXERRORBARS	It is a combination of the **PLOT_PLOTTYPE_BOXES** and **PLOT_PLOTTYPE_YERRORBARS** plot types.
PLOT_PLOTTYPE_BOXES	Draw a box centered about the given x coordinate.
PLOT_PLOTTYPE_BOXXYERRORBARS	A combination of **PLOT_PLOTTYPE_BOXES** and **PLOT_PLOTTYPE_XYERRORBARS** plot types.
PLOT_PLOTTYPE_CANDLESTICKS	Display box-and-whisker plots of financial or statistical data.
PLOT_PLOTTYPE_DOTS	Use dots to mark each data point.
PLOT_PLOTTYPE_FILLEDCURVES	Fill an area bounded by a curve with a solid color or pattern.
PLOT_PLOTTYPE_FINANCEBARS	Display financial data.
PLOT_PLOTTYPE_FSTEPS	Adjacent points are connected with two line segments, one from (x1,y1) to (x1,y2), and a second from (x1,y2) to (x2,y2).
PLOT_PLOTTYPE_HISTEPS	The point x1 is represented by a horizontal line from ((x0+x1)/2,y1) to ((x1+x2)/2,y1). Adjacent lines are connected with a vertical line from ((x1+x2)/2,y1) to ((x1+x2)/2,y2).
PLOT_PLOTTYPE_IMPULSES	Display vertical lines from the x-axis (for 2D plots) or the x-y plane (for 3D plots) to the data points.
PLOT_PLOTTYPE_LINES	Data points are connected with a line.
PLOT_PLOTTYPE_LINESPOINTS	Markers are displayed at each data point and connected with a line.
PLOT_PLOTTYPE_POINTS	Markers are displayed at each data point.
PLOT_PLOTTYPE_STEPS	Adjacent points are connected with two line segments, one from (x1,y1) to (x2,y1), and a second from (x2,y1) to (x2,y2).
PLOT_PLOTTYPE_VECTORS	Display vectors.
PLOT_PLOTTYPE_XERRORBARS	Display dots with horizontal error bars.
PLOT_PLOTTYPE_XERRORLINES	Display linepoints with horizontal error lines.
PLOT_PLOTTYPE_XYERRORBARS	Display dots with horizontal and vertical error bars.
PLOT_PLOTTYPE_XYERRORLINES	Display linepoints with horizontal and vertical error lines.
PLOT_PLOTTYPE_YERRORBARS	Display points with vertical error bars.
PLOT_PLOTTYPE_YERRORLINES	Display linepoints with vertical error lines.

```
#include <chplot.h>

int main() {
    double x, y, xx[2], yy[2];
    string_t text;
    int line_type = -1, line_width = 2, datasetnum = 0;
    class CPlot plot;

    plot.axisRange(PLOT_AXIS_X, 0, 5);
    plot.axisRange(PLOT_AXIS_Y, 0, 4);
    plot.ticsRange(PLOT_AXIS_Y, 1, 0, 4);
    plot.title("Line Types in Ch Plot");
    for (y = 3; y >= 1; y--) {
        for (x = 1; x <= 4; x++) {
            sprintf(text, "%d", line_type);
            lindata(x, x+.5, xx);
            lindata(y, y, yy);
            plot.data2D(xx, yy);
            plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum);
            plot.lineType(datasetnum, line_type, line_width);
            plot.text(text, PLOT_TEXT_RIGHT, x-.125, y, 0);
            datasetnum++;
            line_type++;
         }
    }
    plot.plotting();
}
```

Program 23.15: A plotting program for different line types.

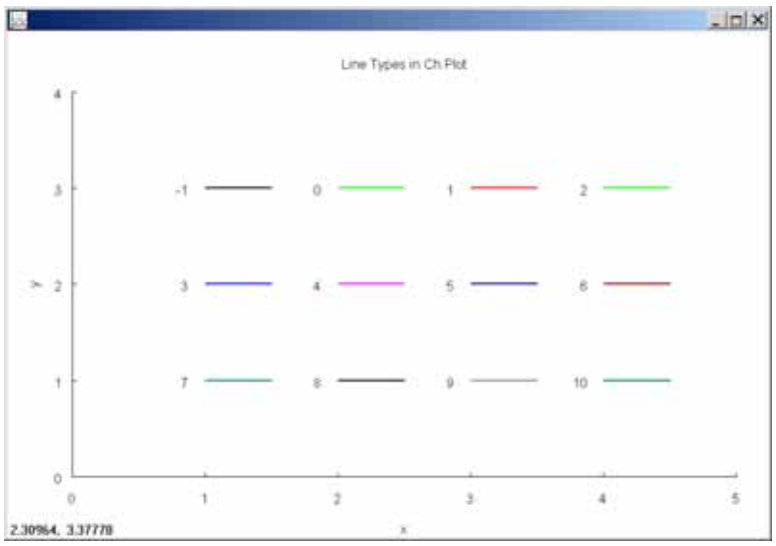


Figure 23.12: A plot with different line types displayed in Windows.

```
/* File: color1.ch */
#include <chplot.h>

/* colors of lines for displayed plot */
char *color[] = {
  "black",
  "white",
  "grey",
  "grey40",
  "grey60",
  "red",
  "yellow",
  "green",
  "blue",
  "navy",
  "cyan",
  "magenta",
  "orange",
  "gold",
  "brown",
  "purple",
};


int main() {
    double x[2], y[2];
    int i, line_type= 1, line_width = 1, datasetnum = 0, n;
    CPlot plot;

    plot.title("Line Colors in Ch Plot");
    n = sizeof(color)/sizeof(color[0]);
    y[0] = 0; y[1] = 1;
    for (i = 0; i < n; i++) {
       x[0] = i+1; x[1] = i+1;
       plot.data2D(x, y);
       plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum);
       plot.lineType(datasetnum, line_type, line_width, color[i]);
       datasetnum++;
    }
    /* color of the horizontal line added below is green */
    x[0] = 1; x[1] = 15;
    y[0] = 0.5; y[1] = .5;
    plot.data2D(x, y);
    plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum);
    plot.lineType(datasetnum, line_type, line_width, "green");
    plot.plotting();
}
```

Program 23.16: Specify colors of curves inside a program.

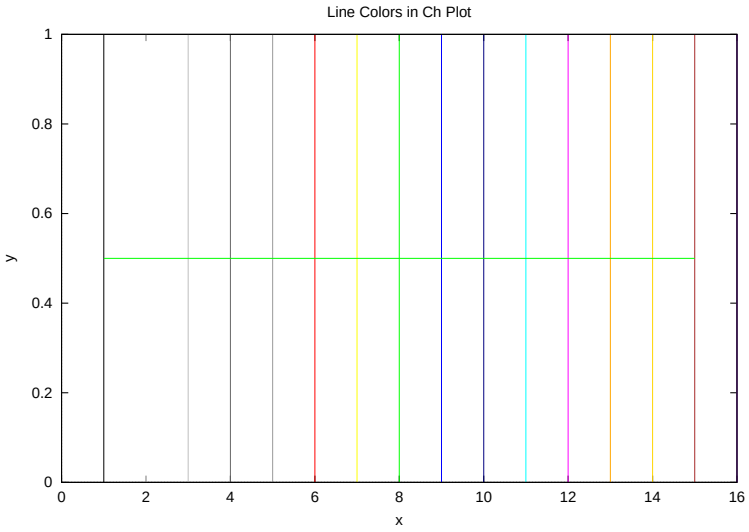


Figure 23.13: A plot with colors specified inside a program saved as a postscript file.

Program 23.17 illustrates how to generate multiple plots using the same instance of a class. When Program 23.17 is executed, a plot with a red curve is first displayed. Next, a plot with blue curve is displayed. Then, the color of the curve is changed by overlaying another curve with red color. The last color dominates. The last color can be determined dynamically by a program at execution time. Finally, a new curve with color of magenta is added to the plot. Four separate plots generated by Program 23.17 are displayed in Figure 23.14.

The point type, size , and color points can be set by function

```
void pointType(int num, int point_type, int point_size, ...
               /* [char *point_color] */);
```

Function **CPlot**::**pointType**() sets the desired point style for a data set to be plotted. The *point_type* specifies an index for the point type used for drawing the point. The point type varies depending on the terminal type used (see **CPlot**::**outputType**). The value *point_type* is used to change the appearance (color and/or marker type) of a point. It is specified with an integer representing the index of the desired point type. All terminals support at least six different point types. *point_size* is an optional argument used to change the size of the point. The point size is *point_size* multiplied by the default size. If *point_type* and *point_size* are set to zero or a negative number, a default value is used. An optional fourth argument can specify the color of a point by a color name or RGB value, such as `"blue"` or `"#0000ff"` for color blue.

Program 23.18 illustrates how different point types are used. The plot displayed in Windows displayed is shown in Figure 23.15. In Figure 23.16, two sets of data, one in line type and the other in point type, are displayed in the same plot. The source code generating this figure is listed in Program 23.19.

23.2.2 Polar Plot

A 2D plot in a polar coordinate system can be specified using member function

```
void CPlot::polarPlot(int angle_unit);
```

The argument *angle_unit* specifies the unit for measurement of angular positions. It can be one of the following macros **PLOT_ANGLE_DEG** for angles measured in degree and **PLOT_ANGLE_RAD** in ra-

```
#include <math.h>
#include <chplot.h>

int main() {
    array double x[36], y[36];
    int line_type = 1, line_width = 1, datasetnum = 0;
    CPlot plot;

    lindata(-M_PI, M_PI, x);
    y = sin(x);

    plot.data2D(x, y);
    plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum);
    plot.lineType(datasetnum, line_type, line_width, "red");
    plot.legend("red line", datasetnum);
    plot.plotting();

    /* change the color of the curve from the same data set to blue  */
    plot.lineType(datasetnum, line_type, line_width, "blue");
    plot.legend("blue line", datasetnum);
    plot.plotting();

    /* overlaying blue curve with red curve .*/
    plot.data2D(x, y);
    datasetnum++;
    plot.lineType(datasetnum, line_type, line_width, "red");
    plot.legend("red line", datasetnum);
    plot.plotting();

    /* add a new curve with color of magenta to the plot */
    y = sin(x)+0.5;
    plot.data2D(x, y);
    datasetnum++;
    plot.lineType(datasetnum, line_type, line_width, "magenta");
    plot.legend("magenta line", datasetnum);
    plot.plotting();
}
```

Program 23.17: Change the color of curve by overlaying a new curve with a different color.

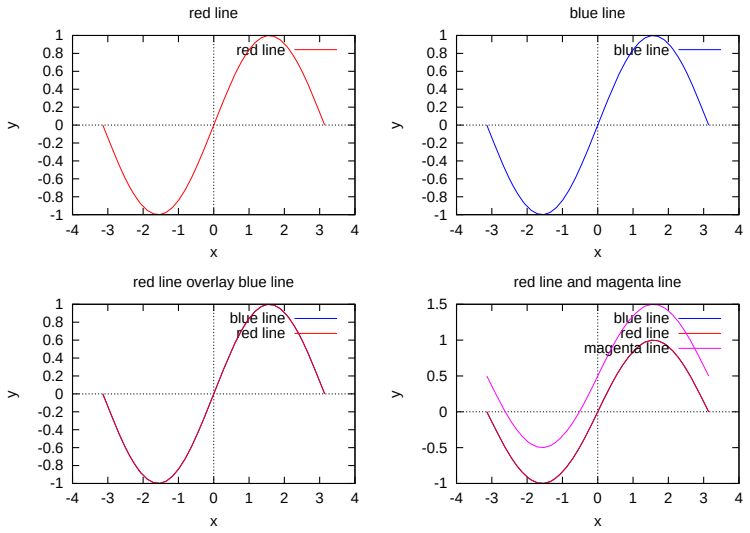


Figure 23.14: A plot with the color of curve changed by overlaying a new curve.

```
#include <chplot.h>

int main() {
    double x, y;
    string_t text;
    int datasetnum=0, point_type = 1, point_size = 5;
    class CPlot plot;

    plot.axisRange(PLOT_AXIS_X, 0, 7);
    plot.axisRange(PLOT_AXIS_Y, 0, 5);
    plot.title("Point Types in Ch Plot");
    for (y = 4; y >= 1; y--) {
        for (x = 1; x <= 6; x++) {
            sprintf(text, "%d", point_type);
            plot.point(x, y, 0);
            plot.plotType(PLOT_PLOTTYPE_POINTS, datasetnum);
            plot.pointType(datasetnum,  point_type, point_size);
            plot.text(text, PLOT_TEXT_RIGHT, x-.25, y, 0);
            datasetnum++; point_type++;
        }
    }
    plot.plotting();
}
```

Program 23.18: A plotting program for different point types.

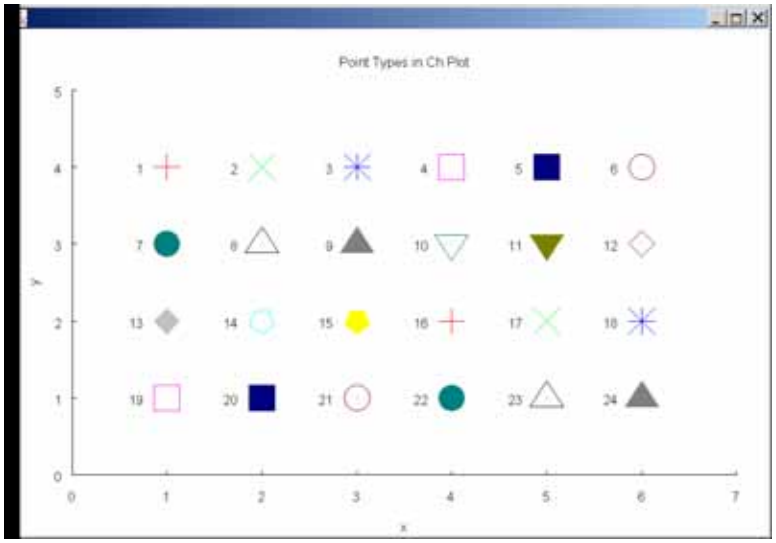


Figure 23.15: A plot with different point types dispayed in Windows.

```
#include <chplot.h>
#include <math.h>

int main() {
    array float x1[75], y1[75];
    array float x2[300], y2[300];
    class CPlot plot;
    int numdataset=0, pointtype =1, pointsize=1,
                      linetype =3,  linesize=1;

    lindata(-2*M_PI, 2*M_PI, x);
    lindata(-2*M_PI, 2*M_PI, x2);
    y1 = x1.*x1+5*sin(10*x1);
    y2 = x2.*x2+5*sin(10*x2);
    plot.data2D(x1, y1);
    plot.data2D(x2, y2);
    plot.plotType(PLOT_PLOTTYPE_POINTS, numdataset);
    plot.pointType(numdataset, pointtype, pointsize);
    numdataset++;
    plot.plotType(PLOT_PLOTTYPE_LINES, numdataset);
    plot.lineType(numdataset, linetype, linesize);
    plot.plotting();
}
```

Program 23.19: A plotting program with two different plot types.

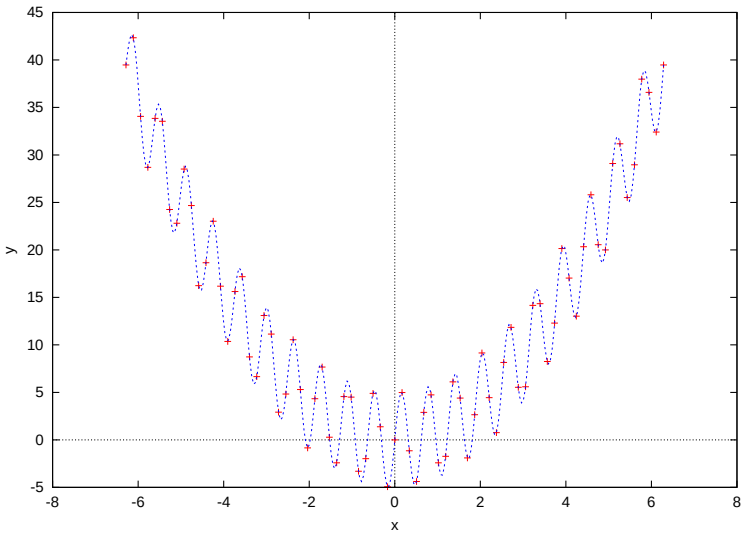


Figure 23.16: A plot with plot types of point and line.

dian. As shown in Program 23.20, in a polar coordinate system (θ, r), the first and second array arguments in member function call of `plot.data2D(theta, r)` are the phase angle and magnitude of points to be plotted, respectively. The polar grid displayed in Figure 23.17 is achieved by a function call of `plot.grid(PLOT_ON)`. The aspect ratio of a plot can be set by member function

```
void CPlot::sizeRatio(float ratio);
```

The meaning of *ratio* changes depending on its value. A positive *ratio* is the ratio of the length of the y-axis to the length of the x-axis. So, if *ratio* is 2, the y-axis will be twice as long as the x-axis. If *ratio* is zero, the default aspect ratio for the terminal type is used. A negative *ratio* is the ratio of the y-axis units to the x-axis units. So, if *ratio* is -2, one unit on the y-axis will be twice as long as one unit on the x-axis. In case of a polar plot, the aspect ratio should be set to 1 as shown in Program23.20.

23.2.3 2D Plotting Functions

The high-level plotting functions **fplotxy**(), **plotxy**(), and **plotxyf**() are easy to use, and can be conveniently used for 2D plotting. These functions can be used in conjunction with the **CPlot** member functions to create more sophisticated plots.

The plotting function **plotxy**() is prototyped as,

```
int plotxy(double x[&], array double &y, ...
 /*[int n] [string_t title, xlabel, ylabel], [class CPlot *plot] */);
```

The arrays `x` and `y` are of real type. Conversion of the data to data type **double** is performed internally. To be compatible with the SIGL graphical library, the third argument for the number of elments of array `x` is optional. The subsequent optional arguments of function **plotxy**() give the title, label in x-axis, and label in y-axis. A pointer to a plot **CPlot** class can also be passed to this function. If argument *plot* is not NULL, an instance of class, pointed to by the argument *plot*, will be initialized with parameters passed to function **plotxy**(). The plot can then be displayed using the **CPlot**::**plotting**() member function. If a previously initialized **CPlot** variable is passed, it will be re-initialized with the function parameters. If no pointer or a NULL pointer is passed internally, an instance of **CPlot** class will be used, and a plot will be displayed without calling the **CPlot**::**plotting**() member function. The following code segment

```
#include <math.h>
#include <chplot.h>

int main() {
    int numpoints = 360;
    array double theta[numpoints], r[numpoints];
    class CPlot plot;

    lindata(0, M_PI, theta);
    r = sin(5*theta);
    plot.polarPlot(PLOT_ANGLE_RAD);
    plot.data2D(theta, r);
    plot.sizeRatio(1);
    plot.grid(PLOT_ON);
    plot.plotting();
}
```

Program 23.20: A plotting program using a polar coordinate system.

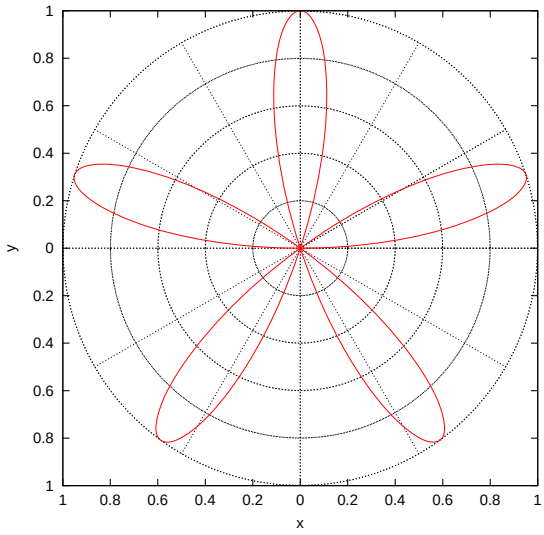


Figure 23.17: A plot in a polar coordinate system.

```
/* plot a sine wave */
#include<math.h>
#include<chplot.h>
int main() {
  int numpoints = 36;
  array double x[numpoints], y[numpoints];

  lindata(0, 360, x);
  y = sin(x*M_PI/180);
  plotxy(x,y);
}
```

Program 23.21: A simple program using plotting function **plotxy**().

```
class CPlot plot;
plotxy(x, y, title, xlabel, ylabel, &plot);
```

is equivalent to

```
class CPlot plot;
plot.data2D(x, y);
plot.title(title);
plot.label(PLOT_AXIS_X, xlabel);
plot.label(PLOT_AXIS_Y, ylabel);
```

The code segment

```
class CPlot plot;
plotxy(x, y, n);
```

is equivalent to

```
class CPlot plot;
plot.data2DCurve(x, y, n);
```

In the simplest form, function **plotxy**() takes two arguments of arrays of scalar types as shown in Program 23.21 with plot displayed in Figure 23.1. If the plotting function call `plotxy(x, y)`, in Program 23.21, is changed to `plotxy(x, y, "Sine Wave", "degree", "amplitude")`, the title and labels of the plot will be created as shown in Figure 23.4. Function **plotxy**() can also be used along with member functions of class **CPlot** to plot multiple sets of data as shown in Program 23.22. Figure 23.7 with legends can be generated by Program 23.22. In Program 23.14, the same plot instance is used to export a plot with different file formats.

Instead of using data from arrays, plotting function **plotxyf**() uses data from a file. The plotting function **plotxyf**() is prototyped as,

```
int plotxyf(string_t filename, ...
 /* [string_t title, xlabel, ylabel], [class CPlot *plot] */);
```

The data format for `filename` is the same as that for a file used in the argument of member function **CPlot**::**dataFile**().

Function **fplotxy**() uses data generated from a function as shown in Program 23.23 with output in Figure 23.18. The plotting function **fplotxy**() is prototyped as,

```
/* two plots in a same figure */
#include<math.h>                                    // M_PI defined
#include<chplot.h>
int main() {
  int numpoints = 36;
  array double x1[numpoints], y1[numpoints];
  array double x2[numpoints], y2[numpoints];
  string_t title="Sine and Cosine Waves",
         xlabel="degree",
         ylabel="amplitude";
  class CPlot plot;

  lindata(0, 360, x1);
  lindata(0, 360, x2);
  y1 = sin(x1*M_PI/180);
  y2 = cos(x2*M_PI/180);
  plotxy(x1,y1,title,xlabel,ylabel,&plot);
  plot.legend("sin(x)", 0);                         // add legend for 1st plot
  plot.data2D(x2, y2);                              // Add data for 2nd plot
  plot.legend("cos(x)", 1);                         // add legend for 2nd plot
  plot.plotting();                                  // do plotting
}
```

Program 23.22: A program for plotting two functions on the same plot with title, labels, and legends.

```
#include <math.h>
#include <chplot.h>
double func(double x) {
  double y;
  y =  sin(x)/(x);
  return y;
}
int main() {
  double x0 = -20, xf = 20;
  fplotxy(func, x0, xf);
}
```

Program 23.23: A plotting program using function **fplotxy**().

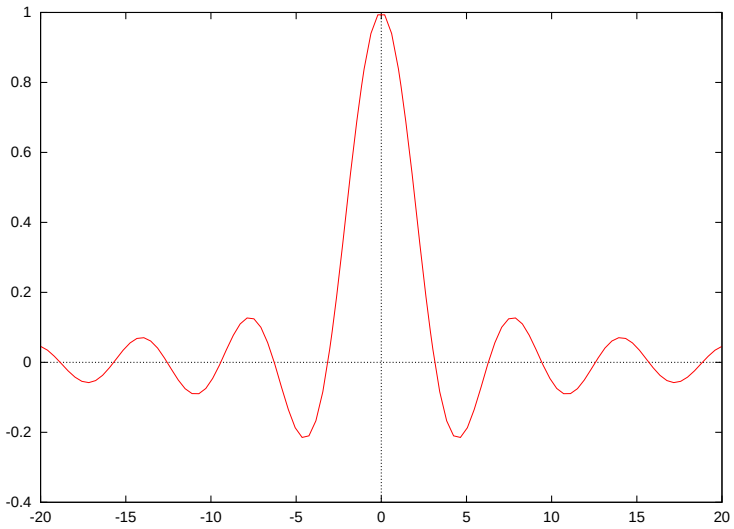


Figure 23.18: A plot generated by function **fplotxy**().

Table 23.7: The macros for 3D plot types.

PLOT_PLOTTYPE_LINES	Data points are connected with a line.
PLOT_PLOTTYPE_IMPULSES	Display vertical lines from the xy plane to the data points.
PLOT_PLOTTYPE_POINTS	Markers are displayed at each data point.
PLOT_PLOTTYPE_LINESPOINTS	Markers are displayed at each data point and connected by a line.
PLOT_PLOTTYPE_SURFACES	Data points are connected and meshed in a smooth surface.
PLOT_PLOTTYPE_VECTORS	Display vectors.

```
int fplotxy( double (*func)(double x), double x0, double xf, ...
/* [num, [string_t title, xlabel, ylabel], [class CPlot *plot]] */);
```

Function **fplotxy**() plots a function of x in the range $x0 \leq x \leq xf$. The function to be plotted, *func*, is specified as a pointer to a function that takes a *double* as an argument and returns a *double*. The arguments *x0* and *xf* are the end-points of the range to be plotted. The optional argument *num* specifies how many points in the range are to be plotted. The number of points plotted are evenly spaced in the range. By default, 100 points are plotted. Like functions **plotxy**() and **plotxyf**(), the optional arguments *title*, *xlabel*, *ylabel*, and *plot* for the plot can also be specified.

23.3 3D Plotting

This section describes features applicable to 3D plotting only.

23.3.1 Plot Types

Like in 2D, the plot type in 3D can also be specified by member function **CPlot**::**plotType**(). The valid macros for plot type are listed in Table 23.7 with corresponding plot types displayed in Figure 23.19. By default, the plot type **PLOT_PLOTTYPE_LINES** is used in 3D plotting.

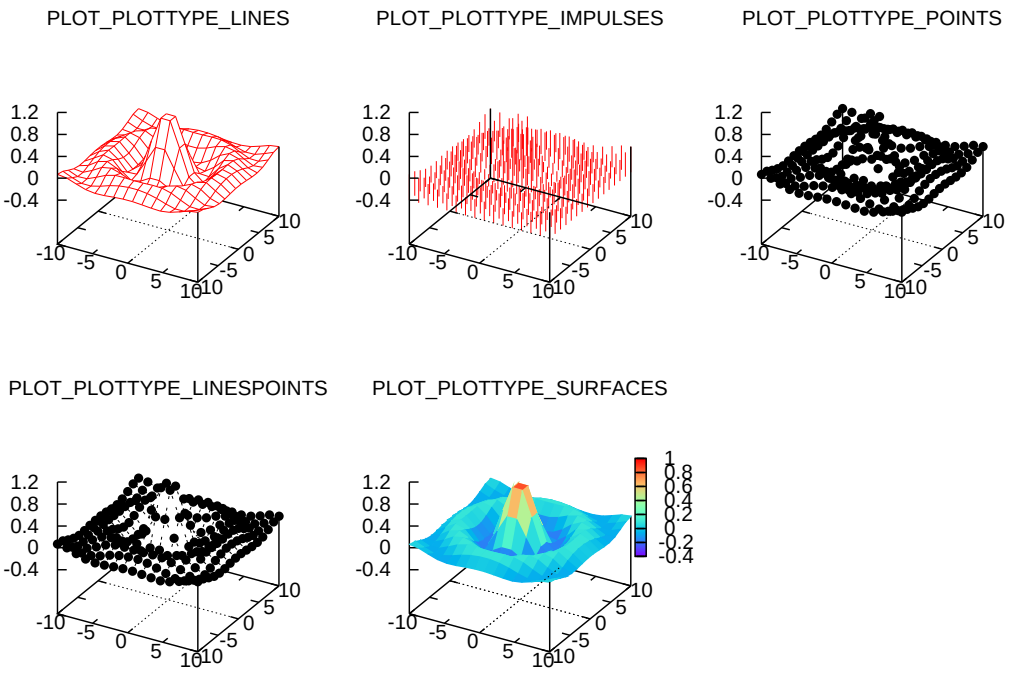


Figure 23.19: Three-dimensional plot types.

23.3.2 Plotting in Different Coordinate Systems

In a two-dimensional case, a data set can be plotted in either a Cartesian or a polar coordinate system. In a three-dimensional case, a data set can be plotted in either the Cartesian, spherical, or cylindrical coordinate system. The coordinate system can be specified by member function

```
void CPlot::coordSystem(int coord_system, .../* int angle_unit */);
```

The argument *coord_system* for the coordinate system can be set to one of three macros **PLOT_COORD_CARTESIAN**, **PLOT_COORD_SPHERICAL**, and **PLOT_COORD_CYLINDRICAL** which stand for Cartesian, spherical, and cylindrical coordinate systems, respectively. By default, a 3D plot uses the Cartesian coordinate system. A point in each coordinate system consists of three values. They are (x,y,z), (θ,ϕ,r), and (θ,z,r) for Cartesian, spherical, and cylindrical coordinate systems, respectively.

Points in a spherical coordinate system are mapped to the Cartesian space by the following formulas:

$$\begin{aligned} x &= r\cos(\theta)\cos(\phi) \\ y &= r\sin(\theta)\cos(\phi) \\ z &= r\sin(\phi) \end{aligned}$$

Program 23.24 generates a plot in a spherical coordinate system shown in Figure 23.20.

For a cylindrical coordinate system, points are mapped to the Cartesian space by formulas:

$$\begin{aligned} x &= r\cos(\theta) \\ y &= r\sin(\theta) \\ z &= z \end{aligned}$$

```
#include <chplot.h>
#include <math.h>

#define NUMT 37
#define NUMP 19
int main() {
    array double theta[NUMT], phi[NUMP], r[NUMT*NUMP];
    class CPlot plot;

    lindata(0, 2*M_PI, theta);
    lindata(-M_PI/2, M_PI/2, phi);
    r = (array double [NUMT*NUMP])1;
    plot.data3D(theta, phi, r);
    plot.coordSystem(PLOT_COORD_SPHERICAL);
    plot.axisRange(PLOT_AXIS_XY, -2.5, 2.5);
    plot.plotting();
}
```

Program 23.24: A plotting program using a spherical coordinate system.

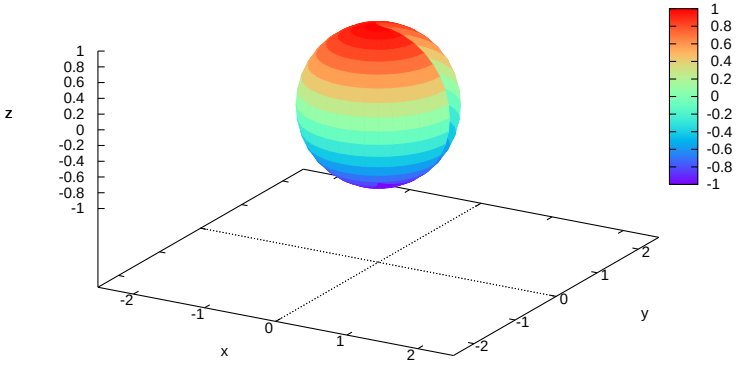


Figure 23.20: A plot in a spherical coordinate system.

```
#include <math.h>
#include <chplot.h>

#define NUMT 36
#define NUMZ 20
int main() {
    int numpoints = 36;
    array double theta[NUMT], z[NUMZ], r[NUMT*NUMZ];
    int i, j;
    class CPlot plot;

    lindata(0, 360, theta);
    lindata(0, 2*M_PI, z);
    for(i=0; i<NUMT; i++) {
        for(j=0; j<NUMZ; j++) {
            r[i*20+j] = 2+cos(z[j]);
        }
    }
    plot.data3D(theta, z, r);
    plot.coordSystem(PLOT_COORD_CYLINDRICAL, PLOT_ANGLE_DEG);
    plot.axisRange(PLOT_AXIS_XY, -4, 4);
    plot.plotting();
}
```

Program 23.25: A plotting program using a cylindrical coordinate system.

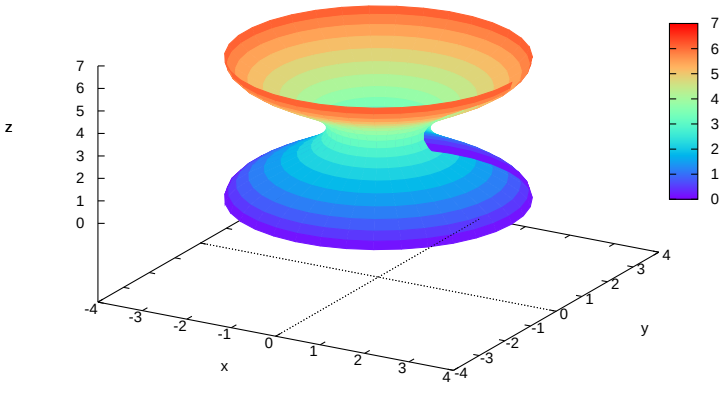


Figure 23.21: A plot in a cylindrical coordinate system.

```
#include <chplot.h>
#include <math.h>

#define NUMX 20
#define NUMY 30
int main() {
    double x[NUMX], y[NUMY], z[NUMX*NUMY];
    double r;
    int i, j;

    lindata(-10, 10, x);
    lindata(-10, 10, y);
    for(i=0; i<NUMX; i++) {
        for(j=0; j<NUMY; j++) {
            r = sqrt(x[i]*x[i]+y[j]*y[j]);
            z[30*i+j] = sin(r)/r;
        }
    }
    plotxyz(x, y, z);
}
```

Program 23.26: A plotting program using function **plotxyz**().

Program 23.25 generates a plot in a cylindrical coordinate system shown in Figure 23.21.

An optional argument *angle_unit* in member function **CPlot**::**coordSystem**() specifies the unit for measurement of angular positions in spherical and cylindrical coordinate systems. The valid macros for optional argument *angle_unit* are **PLOT_ANGLE_DEG** for measurement of angles in degrees and **PLOT_ANGLE_RAD** in radians. In a spherical or cylindrical coordinate system, by default, the angular position is measured in radian.

23.3.3 3D Plotting Functions

Like functions **fplotxy**(), **plotxy**(), and **plotxyf**() in 2D plotting, high-level 3-dimensional plotting functions **fplotxyz**(), **plotxyz**(), and **plotxyzf**() are designed for easy creation of 3D plots. These functions can be used in conjunction with the **CPlot** member functions to create more sophisticated plots.

The plotting function **plotxyz**() is prototyped as,

```
int plotxyz(double x[&], array double y[&], array double &z, ...
/* [int n] [int nx, int ny] [string_t title, xlabel, ylabel, zlabel],
   [class CPlot *plot] */);
```

The arrays *x*, *y*, and *z* are of real type. Conversion of the data to type **double** is performed internally. The title and labels can be specified by optional arguments. A pointer to a plot **CPlot** class can also be passed to this function to obtain the values passed to function **plotxyz**(). The following code segment

```
class CPlot plot;
plotxyz(x, y, z, title, xlabel, ylabel, zlabel, &plot);
```

is equivalent to

```
class CPlot plot;
plot.data3D(x, y, z);
plot.title(title);
```

```
plot.label(PLOT_AXIS_X, xlabel);
plot.label(PLOT_AXIS_Y, ylabel);
plot.label(PLOT_AXIS_Z, zlabel);
```

The following code segment for plotting a 3D curve

```
class CPlot plot;
plotxyz(x, y, z, n);
```

is equivalent to

```
class CPlot plot;
plot.data3DCurve(x, y, z, n);
```

The following code segment for plotting a 3D surface

```
class CPlot plot;
plotxyz(x, y, z, nx, ny);
```

is equivalent to

```
class CPlot plot;
plot.data3DSurface(x, y, z, nx, ny);
```

In the simplest form, function **plotxyz**() takes three arguments of arrays of scalar types as shown in Program 23.26, with the plot displayed in Figure 23.3.

Instead of using data from arrays, plotting function **plotxyzf**() uses data from a file. The plotting function **plotxyzf**() is prototyped as,

```
int plotxyzf(string_t filename, ...
 /* [string_t title, xlabel, ylabel, zlabel], [class CPlot *plot] */);
```

The data format for `filename` is the same as that for a file used in the argument of member function **CPlot**::**dataFile**().

Function **fplotxyz**() uses data generated from a function as shown in Program 23.23 with output in Figure 23.22. The plotting function **fplotxyz**() is prototyped as,

```
int fplotxyz( double (*func)(double x, double y), double x0,
 double xf, double y0, double yf, ..., /* [x_num,  y_num,
 [string_t title, xlabel, ylabel, zlabel], [class CPlot *plot]] */);
```

Function **fplotxy**() plots a function with variables *x* and *y* in the range $x0 \leq x \leq xf$ and $y0 \leq y \leq yf$. The function to be plotted, *func*, is specified as a pointer to a function that takes two arguments for *x* and *y*, and returns a value of double type. The arguments *x0* and *xf* are the end-points for *x* and arguments *y0* and *yf* are for *y*. The optional arguments *x_num* and *y_num* specify how many points in the x and y coordinates are to be plotted, respectively. The number of points plotted are evenly spaced in the range. By default, 25 points are plotted in both the *x* and *y* coordinates. Like functions **plotxyz**() and **plotxyzf**(), the arguments *title*, *xlabel*, *ylabel*, *zlabel*, *xlabel*, *ylabel*, and *plot* for the plot can also be optionally specified.

```
#include <math.h>
#include <chplot.h>

int main() {
    string_t title="fplotxyz()",
             xlabel="X-axis",
             ylabel="Y-axis",
             zlabel="Z-axis";
    double x0 = -3, xf = 3, y0 = -4, yf = 4;
    int x_num = 20, y_num = 50;

    double func(double x, double y) {   // function to be plotted

        return 3*(1-x)*(1-x)*exp(-(x*x) - (y+1)*(y+1) )
               - 10*(x/5 - x*x*x - pow(y,5))*exp(-x*x-y*y)
               - 1/3*exp(-(x+1)*(x+1) - y*y);
    }
    fplotxyz(func, x0, xf, y0, yf, x_num, y_num, title, xlabel, ylabel, zlabel);
}
```

Program 23.27: A plotting program using function **fplotxyz**().

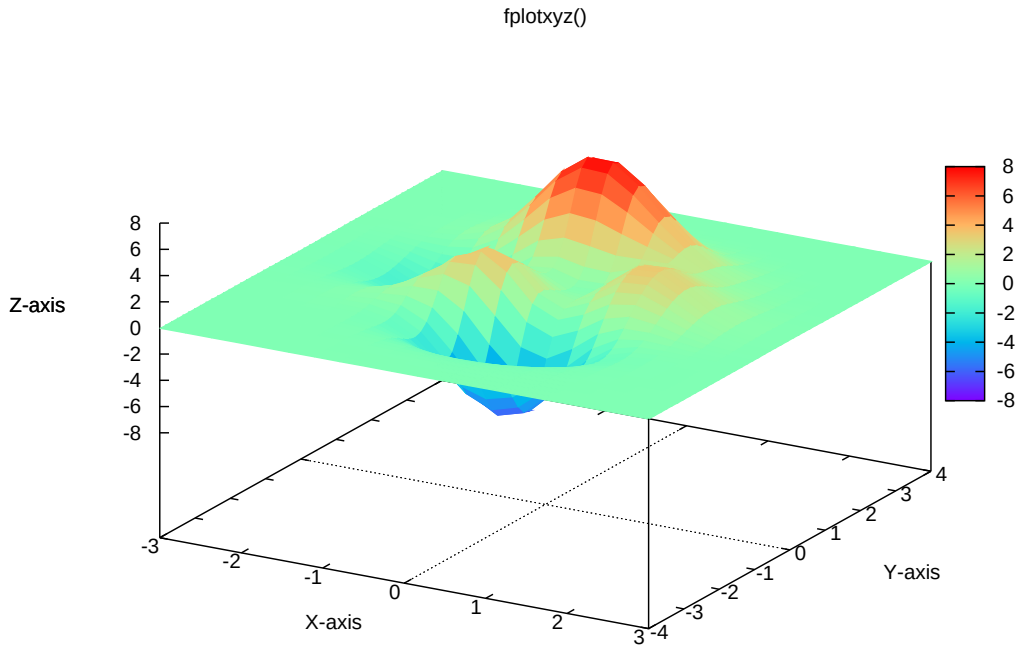


Figure 23.22: A plot generated by function **fplotxyz**().

23.4 Dynamic Web Plotting

Plotting through CGI programs is very useful for many Web-based applications. With Ch Professional Edition and CGI toolkit, plots can be very easily generated dynamically on-line. How to generate a dynamic plot will be presented in this section. We will also describe how data is encoded and decoded for transferring among the browser, Web server, and CGI programs.

In a Web-based plotting, the parameters for plotting are submitted from a Web browser, shown in Figure 23.23, with its corresponding HTML file in Program 23.28 and encoded by the browser. The parameters as name-value pairs are decoded by member function **CRequest**::**getFormNameValue**() in first CGI program **webplot1.ch** shown in Program 23.29. They are then passed as query strings to the second CGI program **webplot2.ch** shown in Program 23.30. These parameters are obtained again using member function **CRequest**::**getFormNameValue**(). The plot generated as a PNG file and displayed through a Web browser is shown in Figure 23.24 .

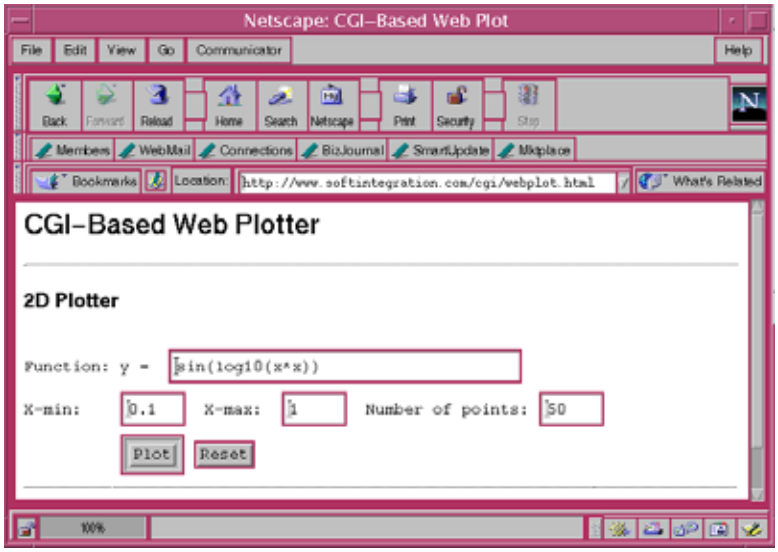


Figure 23.23: A Web-plotter based on the fill-out form.

```
<HTML>
<HEAD>
<TITLE>
CGI-Based Web Plot
</TITLE>
</HEAD>
<BODY bgcolor="#FFFFFF" text="#000000" vlink="#FF0000">
<H1>
CGI-Based Web Plotter
</H1>

<HR>
<H2>2D Plotter</H2>
<PRE>
<FORM method="post" action="/cgi-bin/chcgi/toolkit/demos/sample/webplot1.ch">
Function: y =  <INPUT name="expression" value="sin(log10(x*x))" size=35>
X-min:    <INPUT name="xMin" value="0.1" size=5>  X-max:  <INPUT name="xMax"
value="1" size=5>  Number of points: <INPUT name="numpoints" value="50" size=5>
          <INPUT type="submit" value="Plot"> <INPUT type="reset" value="Reset">
<HR>
</BODY>
</HTML>
```

Program 23.28: HTML file for submitting plotting parameters.

```
#!/bin/ch
#include <cgi.h>

int main() {
   int i, num;
   chstrarray name, value;
   class CResponse Response;
   class CRequest Request;
   class CServer Server;

   num = Request.getFormNameValue(name, value);
   Response.setContentType("text/html");
   Response.begin();
   Response.title("Web Plot");
   printf("<center>\n");
   printf("<img src=\"/cgi-bin/chcgi/toolkit/demos/sample/webplot2.ch");
   for (i=0; i<num; i++){
      putc(i == 0 ? '?' : '&', stdout);
      fputs(Server.URLEncode(name[i]),stdout);
      putc('=', stdout);
      fputs(Server.URLEncode(value[i]),stdout);
   }
   printf("\">\n");
   printf("</center>\n");
   Response.end();
}
```

Program 23.29: CGI program webplot1.ch

```
#!/bin/ch
#include <cgi.h>
#include <chplot.h>

int main() {
   double MinX, MaxX, Step, x, y;
   int pointsX, pointsY, i;
   chstrarray name, value;
   class CResponse Response;
   class CRequest Request;
   class CPlot plot;

   Request.getFormNameValue(name, value);
   MinX = atof(value[1]);
   MaxX = atof(value[2]);
   pointsX = atoi(value[3]);
   double x1[pointsX], y1[pointsX];

   Step = (MaxX - MinX)/(pointsX-1);
   for(i=0;i<pointsX;i++) {
     x = MinX + (i*Step);
     y = streval(value[0]);
     x1[i] = x;
     y1[i] = y;
   }

   Response.setContentType("image/png");
   Response.begin();
   plotxy(x1, y1, value[0], "X", "Y", &plot);
   /* output plot in color png file format */
   plot.outputType(PLOT_OUTPUTTYPE_STREAM, "png");
   plot.plotting();
   Response.end();
}
```

Program 23.30: CGI program webplot2.ch

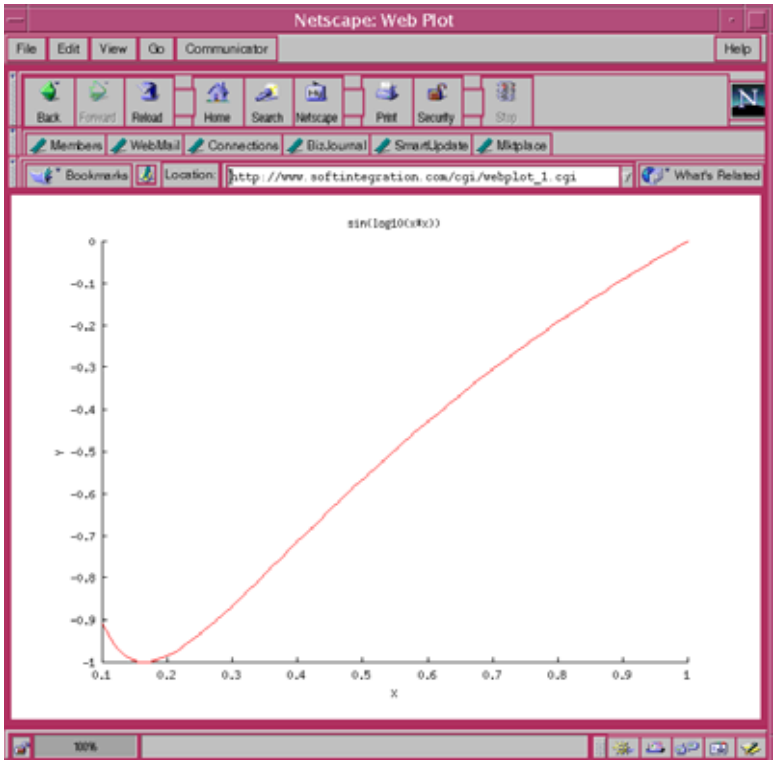


Figure 23.24: Plot generated through the Web plotting.

Chapter 24

Numerical Analysis

Numerical analysis in Ch is the simplist possible extension in the spirit of C. Complicated problems in numerical analysis can often be solved with just one function call in Ch. The advanced features for numerical analysis in Ch are very useful for applications in engineering and science.

The category of numerical analysis functions in Ch is listed in Table 24.1. Functions for numerical analysis prototyped in header file **numeric.h** are listed in Table 24.2. File **numeric.h** includes the following header files **math.h, stdarg.h, array.h** and **dlfcn.h**. It should be pointed out that functions **maxloc(), mean(), median(), minloc(), product(), sort(), std(), sum()** use array of reference function arguments. These functions can handle arrays of different dimensions and data types. The optional second argument shall be array of double type for functions **mean(), median(), product(), sort(), std(),** and **sum().** The optional second argument shall be array of double complex type for functions **cproduct()** and **csum().** The optional second arguments of vectors for these functions contain the results calculated for each row of the first arguments of two-dimensional arrays. Similarly, functions minv() and maxv() return the maximum and minimum values for each row of the input two-dimensional array argument, respectively.

In this chapter, numerical analysis functions for applications in engineering and science will be presented. The detailed description of each function can be found in the chapter about numerical analysis in *The Ch Language Environment — Reference Guide*.

Table 24.1: Category of numerical analysis functions in Ch.

Data Analysis and Statistics
Data Interpolation and Curve Fitting
Minimization or Maximization of Functions
Polynomials
Nonlinear Equations
Ordinary Differential Equations
Derivatives
Integration
Matrix Analysis Functions
Matrix Decomposition
Special Matrices
Linear Equations
Eigenvalues and Eigenvectors
Fast Fourier Transforms
Convolution and Filtering
Cross Correlation
Special Mathematical Functions

24.1 Mathematical Functions

Many elementary mathematical functions are defined in the standard C header file **math.h**. This section presents some commonly used mathematical functions not defined in standard C libraries.

24.1.1 Cross Product

Function **cross**() with the prototype of

```
array double cross(array double a[&], array double b[&])[3];
```

calculates the cross product of two vectors with three elements. The function returns the cross product of vectors *a* and *b* of real type.

For example, the following commands can be used to evaluate the cross product of two vectors

```
> array double a[3] = {1, 2, 3}
> array float b[3] = {2, 3, 4}
> cross(a, a)
0.0000 0.0000 0.0000
> cross(a, b)
-1.0000 2.0000 -1.0000
```

24.1.2 Dot Product

Function **dot**() with the prototype of

```
double dot(array double a[&], array double b[&]);
```

Table 24.2: Functions for numerical analysis.

Function	Description
balance()	Balances a general real matrix to improve accuracy of computed eigenvalues.
ccompanionmatrix()	Calculate companion matrix with complex number.
cdeterminant()	Calculate the determinant of a complex matrix.
cdiagonal()	Get a vector with diagonals of a complex matrix.
cdiagonalmatrix()	Create diagonal matrix from a complex vector.
cfevalarray()	Complex array function evaluation.
cfunm()	Evaluate general complex matrix function.
charpolycoef()	Calculate the coefficients of characteristic polynomial of a matrix.
choldecomp()	Computes the Cholesky factorization of a symmetric positive definite matrix A.
cinverse()	Calculate the inverse of a complex square matrix.
clinsolve()	Solve a system of complex linear equations by LU decomposition.
cmean()	Calculate the mean value of all the elements and mean values of each row of a complex array.
combination()	Calculate the number of combination of *n* things taken *k* at a time.
companionmatrix()	Calculate a companion matrix.
complexsolve()	Solve a complex equation.
condnum()	Calculate condition number of a matrix.
conv()	One-dimensional Discrete Fourier Transform (DFT) based convolution.
conv2()	Two-dimensional Discrete Fourier Transform (DFT) based convolution.
corrcoef()	Correlation coefficients calculation.
correlation2()	Obtain a two-dimensional correlation coefficient.
cpolyeval()	Calculate the value of a complex polynomial at a complex point.
cproduct()	Calculate product of all elements or products of the elements of a complex array of any dimension and the products of of each row of a two-dimensional complex array.
covariance()	Covariance matrix.
cross()	Calculate vector cross product.
csum()	Calculate the sum of all elements of a complex array of any dimension or sums of elements in each row of a two-dimensional complex array.
ctrace()	Calculate the sum of diagonal elements of a complex matrix.
ctriangularmatrix()	Get the upper or lower triangular part of a complex matrix.
cumprod()	Calculate cumulative product of elements in an array
cumsum()	Calculate cumulative sum of elements in an array
curvefit()	Fit a set of data points *x* and *y* to a linear combination of specified base functions.
deconv()	One-dimensional Discrete Fourier Transform (DFT) based deconvolution.
derivative()	Calculate numerical derivative of a function at a given point.
derivatives()	Calculate numerical derivatives of a function at multiple points.
determinant()	Calculate the determinant of a matrix.
diagonal()	Get a vector with diagonals of a matrix.
diagonalmatrix()	Create diagonal matrix of a vector.
difference()	Calculate differences between adjacent elements of array.
dot()	Calculate vector dot product.

Table 24.2: Functions for numerical analysis (continued).

Function	Description
eigen()	Find eigenvalues and eigenvectors.
expm()	Computes a matrix exponential.
factorial()	Factorial function.
fevalarray()	Array function evaluation.
fft()	N-dimensional Fast Fourier Transform (FFT) calculation.
filter()	Filters the data.
filter2()	Two-dimensional Discrete Fourier Transform based FIR filter.
findvalue()	Obtain indices of nonzero elements of an array.
fliplr()	Flip matrix in left/right direction.
flipud()	Flip matrix in up/down direction.
fminimum()	Find the minimum value of a one-dimensional function and its corresponding position for the minimum value.
fminimums()	Find the minimum position and value of an n-dimensional function.
fsolve()	Find a zero position of a nonlinear system of equations.
funm()	Evaluate general real matrix function.
fzero()	Find a zero position of a nonlinear function with one variable.
gcd()	Obtain the greatest common divisor of the corresponding value of two arrays of integral type.
getnum()	Get a number with default value from the console.
hessdecomp()	Reduces a real general matrix to upper Hessenberg form by an orthogonal/unitary matrix similarity transformation.
histogram()	Calculate and plot histograms.
householdermatrix()	Get the Householder matrix.
identitymatrix()	Create an identity matrix.
ifft()	N-dimensional inverse Fast Fourier Transform (FFT) calculation.
integral1()	Numerical integration of a one-dimensional function.
integral2()	Numerical integration of a two-dimensional function.
integral3()	Numerical integration of a three-dimensional function.
integration2()	Numerical integration of a two-dimensional function.
integration3()	Numerical integration of a three-dimensional function.
interp1()	One-dimensional interpolation.
interp2()	Two-dimensional interpolation.
inverse()	Calculate the inverse of a square matrix.
lcm()	Obtain the least common multiple of the corresponding value of two arrays of integral type.
lindata()	Generate linearly spaced data.
linsolve()	Solve a system of linear equations by LU decomposition.
linspace()	Generate a linearly spaced array.
llsqcovsolve()	Solve linear system of equations based on linear least-squares with known covariance.
llsqnonnegsolve()	Solve linear system of equations with non-negative values based on linear least-squares method.
llsqsolve()	Least squares solution of sets of linear equations.
logm()	Computes matrix natural logarithm.
logdata()	Generate logarithmically spaced data.
logspace()	Generate a logarithmically spaced array.
ludecomp()	LU decomposition of general m-by-n matrix.

Table 24.2: Functions for numerical analysis (continued).

Function	Description
maxloc()	Find the location of maximum value of an array.
maxv()	Find the maximum values of the elments of each row of a matrix.
mean()	Calculate the mean value of all the elements and mean values of each row in an array.
median()	Find the median value of all the elements and median values of elements in each row of an array.
minloc()	Find the location of the minimum value of an array.
minv()	Find the minimum values of each row in a two-dimensional array.
norm()	Calculate vector and matrix norms.
nullspace()	Calculate null space of a matrix.
oderk()	Solve ordinary differential equation using a Runge-Kutta method.
orthonormalbase()	Find orthonormal bases of a matrix.
pinverse()	Calculate Moore-Penrose pseudoinverse of a matrix.
polyder()	Take derivative of a polynomial.
polyder2()	Calculate the derivative of product or quotient of two polynomials.
polyeval()	Calculate the value of a polynomial and its derivatives.
polyevalarray()	Polynomial evaluation for a sequence of points.
polyevalm()	Calculate the value of a matrix polynomial.
polyfit()	Fit a set of data points to a polynomial function.
product()	Calculate product of all elements of an array of any dimension or products of the elements of each row of a two-dimensional array.
qrdecomp()	Compute the orthogonal-triangular QR decomposition of a matrix.
rank()	Find the rank of a matrix.
residue()	Partial-fraction expansion or residue computation.
rcondnum()	Matrix reciprocal condition number estimate.
roots()	Find the roots of a polynomial.
rot90()	Rotate matrix 90 degrees.
specialmatrix()	Generate a special matrix.
schurdecomp()	Compute Schur decomposition.
sign()	Sign function.
sort()	Sorting and ranking elements in ascending order.
sqrtm()	Computes the square root of a matrix.
std()	Calculate the standard deviation of a data set.
sum()	Calculate the sum of all elements or sums of elements in each row of an array.
svd()	Singular value decomposition.
trace()	Calculate the sum of diagonal elements.
triangularmatrix()	Get the upper or lower triangular part of a matrix.
unwrap()	Unwrap radian phase of each element of input array by changing its absolute jump greater than π to its $2 * \pi$ complement.
urand()	Uniform random-number generator.
xcorr()	Obtain a one-dimensional cross-correlation vector.

calculates the dot product of two vectors. The function returns the dot product of vectors *a* and *b* of real type. The number of elements of these two vectors shall be the same. Otherwise, function **dot**() returns NaN.

For example, the following commands can be used to evaluate the dot product of two vectors

```
> array double a[3] = {1, 2, 3}
> array double b[] = {1, 2, 3, 4, 5}
> dot(a, a)
14.0000
> dot(b, b)
55.0000
```

24.1.3 Uniform Random Numbers

Standard C functions **srand**() and **rand**() in header file **stdlib.h** can be used to obtain random numbers. Function **urand**() with the prototype of

```
double urand(array double &x);
```

uses a random-number generator with period 2^{32} to obtain successive pseudo-random numbers in the range from 0 to 1. If the parameter for array argument *x* is not NULL, random numbers are stored in argument *x* and the function returns the value of the first element of the array. If the argument of function **urand**() is NULL, only an uniform random number is returned.

For example,

```
> urand(NULL)
0.494766
> array double x[2], y[2][3]
> urand(x)
> x
0.513871 0.175726
> urand(y)
> y
0.3086 0.5345 0.9476
0.1717 0.7022 0.2264
```

24.1.4 Sign Function

Function **sign**() with the prototype of

```
int sign(double x);
```

determines the sign of argument *x*. The function returns the 1, -1 and 0 for $x > 0$, $x < 0$ and $x = 0$, respectively.

For example,

```
> sign(-10)
-1
> sign(10)
1
```

24.1.5 Greatest Common Divisor

Function **gcd**() with the prototype of

```
int gcd(array int &u,  array int &v, array double &g, ...
    /* [array int c[&], array int d[&]] */);
```

obtains the greatest common divisor of the corresponding elements of two arrays *u* and *v* of integer type. The arrays *u* and *v* shall be the same size and contain non-negative integer data. The output array *g* is a positive integer array of the same size as *u*. The optional output arrays *c* and *d* are of the same size as *u* and satisfy the equation $u. * c + v. * d = g$.
Function **gcd**() returns 0 on success and -1 on failure.

For example,

```
> array int u[2][3] = {1,  2, 7,\
                       15, 3, 4}
> array int v[2][3] = {2,  4, 8,\
                       3,  8, 3}
> array int g[2][3],c[2][3],d[2][3]
> gcd(u,v,g,c,d)
> g
1 2 1
3 1 1
> u.*c+v.*d
1 2 1
3 1 1
```

24.1.6 Least Common Multiple

Function **lcm**() with the prototype of

```
int lcm(array int &g,  array double &u, array double &v);
```

obtains the least common multiple of the corresponding elements of two arrays of integer type. The arrays *u* and *v* shall be the same size and contain non-negative integral data. The output array *g* is a positive integral array of the same size as *u*. Function **lcm**() returns 0 on success and -1 on failure.

For example,

```
> array int u[2][3] = {1,2,7,15,3,4}
> array int v[2][3] = {2,4,8,3,8,3}
> array int g[2][3]
> lcm(g,u,v)
> g
2 4 56
15 24 12
```

24.1.7 Complex Equation

A complex equation can be expressed in a general polar form of

$$R_1 e^{i\phi_1} + R_2 e^{i\phi_2} = z_3 \tag{24.1}$$

where z_3 can be expressed in either Cartesian coordinates $x_3 + iy_3$ as `complex(x3, y3)`, or polar coordinates $R_3 e^{i\phi_3}$ as `polar(R3, phi3)`. Because it is a complex equation that can be partitioned into real and imaginary parts, two unknowns out of four parameters R_1, ϕ_1, R_2, and ϕ_2 can be solved in this equation. The parameters R_1, ϕ_1, R_2, and ϕ_2 are in positions 1, 2, 3, and 4, respectively.

Function **complexsolve**() with the prototype of

```
int complexsolve(int n1, int n2,
     double phi_or_r1, double phi_or_r2, double complex z3,
     double &x1, double &x2, double &x3, double &x4);
```

can solve equation (24.1) for two unknowns. The arguments *n1* and *n2* are the positions of the first and second unknowns on the left hand side of equation (24.1), respectively. The valid values for position are 1, 2, 3, or 4. Arguments *phi_or_r1* and *phi_or_r2* are the values of the remaining two known variables on the left hand side of the equation. Argument *z3* is a complex number on the right hand side. Arguments *x1* and `x2` contain the results for the first and second unknowns, respectively. If there are more than one set of solutions, arguments *x3* and `x4` contain the results for the the first and second unknowns for the second set of solution, respectively. Function **complexsolve**() returns the number of solutions with values of 0, 1, or 2.

For example, unknowns in equation $3.5e^{i4.5} + R_2 e^{i\phi_2} = 1 + i2$ can be solved by the following commands.

```
> int n1 = 3, n2 = 4
> double phi_or_r1= 3.5, phi_or_r2 = 4.5
> double R2, phi2, x3, x4
> complex z3 = complex(1, 2)
> complexsolve(n1, n2, phi_or_r1, phi_or_r2, z3, R2, phi2, x3, x4)
1
> R2
5.6931
> phi2
1.2606
```

For example, unknowns ϕ_1 and ϕ_2 in equation $3.5e^{i\phi_1} + 4.5e^{i\phi_2} = e^i + 3e^{i4}$ can be solved by the following commands.

```
> int n1 = 2, n2 = 4
> double phi_or_r1= 3.5, phi_or_r2 = 4.5
> double phi1, phi2, phi1_2, phi2_2
> complex z3 = polar(1, 1)-polar(3, 4)
> complexsolve(n1, n2, phi_or_r1, phi_or_r2, z3, \
               phi1, phi2, phi1_2, phi2_2)
2
> phi1
-0.3890
> phi2
1.7354
> phi1_2
2.1765
> phi2_2
0.0521
```

24.2 Data Analysis and Statistics

24.2.1 Get Numbers from Console

Function **getnum**() with the prototype of

```
double getnum(string_t msg, double d);
```

obtains a real number from the console through the standard input stream or a default value in the second argument. This function returns a default number when the carriage RETURN is entered as input, or new number from *stdin* as a double-precision floating-point number. However, if an invalid number is entered, a different number is requested. The message in string *msg* will be printed out.

For example,

```
> double d
> d = getnum("Please enter a number[10.0]: ", 10);
Please enter a number[10.0]:
90
> d
90.0000
```

24.2.2 Assign Data to Arrays

In addition to initialization and assignment of arrays, values can be assigned to arrays by functions **lindata**() and **logdata**(). Function **lindata**() with the prototype of

```
int lindata(double first, double last, ... /*[array] type a[:]...[:]*/);
```

assigns linearly spaced values starting with *first* and ending with *last* for elements of array *a* passed in the third argument. The number of points is taken internally from the array *a*. Function **lindata**() returns the number of elements in array *a* or 0 on failure. Function **logdata**() is similar to **lindata**(), except that the values are logarithmically spaced for elements of the array in the third argument.

For example,

```
> array double a[6];
> lindata(0, 2, a);
> a
0.0000 0.4000 0.8000 1.2000 1.6000 2.0000
```

Function **linspace**() with the prototype of

```
int linspace(array double &a, double first, double last);
```

assigns linearly spaced values starting with *first* and ending with *last* for elements of array *a*. The number of points is taken internally from the ar ray *a*. Function **linspace**() returns the number of elements in array *a* or 0 on failure. Function **logspace**() is similar to **linspace**(), except that the values are logarithmically spaced for elements of the array in the first argument. For example,

```
> array double a[6];
> linspace(a, 0, 2);
> a
0.0000 0.4000 0.8000 1.2000 1.6000 2.0000
```

Functions **linspace**() and **logspace**() are obsolete and will be phased out. Users are encouraged to use functions **lindata**() and **logdata**(). Function calls

```
linspace(a, 0, 2);
logpace(a, 0, 2);
```

can be replaced by

```
lindata(0, 2, a);
logdata(0, 2, a);
```

24.2.3 Minimum and Maximum

The minimum value in a list of expressions of scalar and array types can be obtained using the generic function **min**(). If all arguments are integer types, **min**() returns a value int type. Otherwise, it returns a value of double type. Function **minloc**() with the prototype of

```
int minloc(array double &a);
```

returns the index for the element with the minimum value in array *a*. Function **minv**() with the prototype of

```
array double minv(array double a[&][&])[:];
```

returns an array of minimum values of each row in the argument of a two-dimensional array. To calculate an array of minimum values of each column in the argument of a two-dimensional array, function **transpose()** can be used to obtain the transpose of the array first.

Generic function **max**() is similar to **min**(). Instead of a minimum value, function **max**() returns the maximum value from the list of expressions in its argument. Function **maxloc**() with the prototype of

```
int maxloc(array double &a);
```

returns the index for the element with the maximum value in array *x*. Function **maxv**() with the prototype of

```
array double maxv(array double a[&][&])[:];
```

returns an array of maximum values of each row in the argument of a two-dimensional array.

For example,

```
> array double a[3] = {10, 2, 3}
> float f= 2.5;
> min(a, f, 2.4)
2.0000
> minloc(a)
1
> array int i[3][2] = {1, 2, 3, 4, 5, 6}
> minv(i)
1.0000 3.0000 5.0000
> minv(transpose(i))
1.0000 2.0000
> max(4, 5, i, 10)
10
```

24.2.4 Sum

Function **sum**() with the prototype of

```
double sum(array double &a, ... /* [array double v[:]] */);
```

calculates the sum of all the elements in an array. If the array is a two-dimensional matrix, the function can calculate the sum of each row with the result stored in the optional second argument of the one-dimensional array. For example,

```
> array double a[3] = {10, 2, 3}
> sum(a)
15.0000
> array double b[3][2] = {1, 2, 3, 4, 5, 6}
> array double v[3]
> sum(b, v)
> v
3.0000 7.0000 11.0000
```

For an array of complex numbers, function **csum**() with the prototype of

```
double complex csum(array double complex &a, ...
                    /* [array double complex v[:]] */);
```

shall be used to calculate the sum.

For a vector $x = [x_1, x_2, \cdots, x_n]$, cumulative sum $y = [y_1, y_2, \cdots, y_n]$ is defined by

$$y_i = x_1 + x_2 + \cdots + x_i \qquad i = 1, 2, \cdots, n$$

Function **cumsum**() with the prototype of

```
int cumsum(array double complex &y, array double complex &x);
```

computes the cumulative sum of the input array *x*. If the input *x* is a vector, it calculates the cumulative sum of the elements of *x*. If the input *x* is a two-dimensional matrix, it calculates the cumulative sum over each row. If the input *x* is a three-dimensional array, it calculates the cumulative sum over the first dimension. It is invalid for calculation of cumulative sum for arrays with more than three dimensions.

For example,

```
> array double x[6]={1,2,3,4,5,6}, y[6]
> cumsum(y, x)
> y
1.0000 3.0000 6.0000 10.0000 15.0000 21.0000
> array double complex zx[3][2]={complex(1,1),2,3,complex(2,2),5,6}
> array double complex zy[3][2]
> cumsum(zy, zx)
complex(1.0000,1.0000) complex(3.0000,1.0000)
complex(3.0000,0.0000) complex(5.0000,2.0000)
complex(5.0000,0.0000) complex(11.0000,0.0000)
```

24.2.5 Product

Function **product**() with the prototype of

```
double product(array double &a, ... /* [array double v[:]] */);
```

calculates the product of all the elements in an array. If the array is a two-dimensional matrix, the function can calculate the product of each row with the result stored in the optional second argument of the one-dimensional array. For example,

```
> array double a[3] = {10, 2, 3}
> product(a)
60.0000
> array double b[3][2] = {1, 2, 3, 4, 5, 6}
> array double v[3]
> product(b, v)
> v
2.0000 12.0000 30.0000
```

For an array of complex numbers, function **cproduct**() with the prototype of

```
double complex cproduct(array double complex &a, ...
                    /* [array double complex v[:]] */);
```

shall be used to calculate the product.

For a vector $x = [x_1, x_2, \cdots, x_n]$, cumulative product $y = [y_1, y_2, \cdots, y_n]$ is defined by

$$y_i = x_1 * x_2 * \cdots * x_i \qquad i = 1, 2, \cdots, n$$

Function **cumprod**() with the prototype of

```
int cumprod(array double complex &y, array double complex &x);
```

computes the cumulative product of the input array *x*. If the input *x* is a vector, it calculates the cumulative product of the elements of *x*. If the input *x* is a two-dimensional matrix, it calculates the cumulative product over each row. If the input *x* is a three-dimensional array, it calculates the cumulative product over the first dimension. It is invalid for calculation of cumulative product for arrays with more than three dimensions.

For example,

```
> array double x[6]={1,2,3,4,5,6}, y[6]
> cumprod(y, x)
> y
1.0000 2.0000 6.0000 24.0000 120.0000 720.0000
> array double complex zx[3][2]={complex(1,1),2,3,complex(2,2),5,6}
> array double complex zy[3][2]
> cumprod(zy, zx)
complex(1.0000,1.0000) complex(2.0000,2.0000)
complex(3.0000,0.0000) complex(6.0000,6.0000)
complex(5.0000,0.0000) complex(30.0000,0.0000)
```

24.2.6 Mean

Function **mean**() with the prototype of

```
double mean(array double &a, ... /* [array double v[:]] */);
```

can be used to calculate the mean value of all the elements in an array of any dimension and real type. If the array is a two-dimensional matrix, the function can calculate mean values of each row. The mean values of each row are passed out by the second argument *v* of the one-dimensional array. For complex arrays, function **cmean**() with the prototype of

```
double complex cmean(array double complex &a, ...
                     /* [array double complex v[:]] */);
```

shall be used for the calculation of mean values.

For example, the following commands can be used to evaluate the mean values.

```
> double a[2][3] = {1, 2, 3, 6, 5, 4}
> double m
> m = mean(a)
3.5000
> array double v[2]
> mean(a, v)
> v
2.0000 5.0000
```

24.2.7 Median

Function **median**() with the prototype of

```
double median(array double &a, ... /* [array double v[:]] */);
```

can be used to calculate the median value of all elements in an array of any dimension and real type. If the array is a two-dimensional matrix, the function can calculate median values of each row. The median values of each row are passed out by the second argument *v* of the one-dimensional array.

For example, the following commands can be used to evaluate a median value.

```
> double a[2][3] = {1, 2, 3, 6, 5, 4}
> double m
> m = median(a)
3.5000
```

24.2.8 Standard Deviation

The standard deviation of the data set σ_i is defined as

$$\sigma = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2}$$

where N is the number of observations of data set *x*, and $\overline{x}$ is the mean of data set *x*. Function **std**() with the prototype of

```
double std(array double &a, ... /* [array double v[:]] */);
```

can be used to calculate the standard deviation of all elements in an array of any dimension and real type. If the array is a two-dimensional matrix, the function can calculate standard deviations of each row. The standard deviations of each row are passed out by the second argument *v* of the one-dimensional array.

For example, the following commands can be used to evaluate a standard deviation.

```
> double a[2][3] = {1, 2, 3, 6, 5, 4}
> double m
> m = std(a)
1.8708
```

24.2.9 Covariance and Correlation Coefficients

For array **X** of n-by-m dimension, with n variables, which has a total of m observations, the element (i,j) of the covariance matrix **C** of array **X** is defined as

$$\mathbf{C}[i][j] = E[(x_i - \mu_i)(x_j - \mu_j)] \qquad i, j = 1, 2, \cdots, N;$$

where E is the mathematical expectation and $\mu_i = Ex_i$. Define element (i, j) matrix **u** as

$$u_{ij} = x_{ij} - \mu_i; \qquad i = 0, 1, \cdots, N; j = 0, 1, \cdots, M$$

where

$$\mu_i = \frac{1}{M}\sum_{j=1}^{M} x_{ij}; \qquad i = 0, 1, \cdots, N$$

Then, the covariance matrix can be calculated by

$$\mathbf{C} = \frac{1}{N-1}\mathbf{u} * \mathbf{u^T}$$

Function **covariance**() with the prototype of

```
int covariance(array double &c, array double &x, ...
               /* [array double &y] */);
```

calculates the covariance of array *x*. The input **array** *x* can be of any supported arithmetic data type of vector or matrix of any size $n \times m$ (if *x* is a vector, regard the size as $1 \times m$). Each column of *x* is an observation and each row is a variable. Optional input *y* can only be of **double** data type and the same number of columns as *x*. That is, the size of *y* is $n_1 \times m$, or $1 \times m$ if *y* is a vector. Attaching *y* to a row of *x* as a new expanding matrix [**X**] is performed internally. Function call **covariance**(*c*, *x*, *y*) is equivalent to **covariance**(*c*,[**X**]) where **X** is defined as $\mathbf{X} = \begin{pmatrix} x \\ y \end{pmatrix}$. The size of [**X**] is $(n+n_1) \times (m)$. Conversion of the data to double is performed internally. The result *c* is a variance or a covariance matrix depending on *x* and *y*. If *x* is a vector and no optional input *y*, the result *c* is a variance. Otherwise it is a covariance. Diagonal of *c* is a vector of variances for each row. The square root of the diagonal of array *c* is a vector of standard deviations. In function **covariance**(), the mean from each column is removed before calculating the result. Function **covariance**() returns 0 on success and -1 on failure.

Element (i, j) of the correlation coefficient matrix is defined as

$$\texttt{corrcoef}(i, j) = \frac{C_{ij}}{\sqrt{C_{ii} * C_{jj}}}$$

where C_{ij} is the element of the covariance matrix. Function **corrcoef**() with the prototype of

```
int corrcoef(array double &c, array double &x, ...
           /* [array double &y] */);
```

calculates the correlation coefficient matrix. The input array *x* can be of any supported arithmetic data type of vector or two-dimensional array of any size $n \times m$ (if *x* is a vector, regard the size as $1 \times m$). Each column of *x* is an observation and each row is a variable. Optional input *y* can only be of double data type and the same number of columns as *x*. That is, the size of *y* is $n_1 \times m$ or regard as $1 \times m$ if *y* is a vector. Attachment of *y* to a row of *x* as a new expanding matrix [**X**] is performed internally. Function call **corrcoef**(*c*, *x*, *y*) is equivalent to **corrcoef**(*c*,[**X**]) where **X** is defined as $\mathbf{X} = \begin{pmatrix} x \\ y \end{pmatrix}$. The size of [**X**] is $(n + n_1) \times m$. Conversion of the data to double is performed internally. The result *c* is a matrix of correlation coefficients.

For example, the covariance and correlation coefficients of a matrix can be calculated by the following commands.

```
> #define N 3
> #define M 4
> array double  x[N][M]={1,2,3,4, \
                         0,0,0,1, \
                         2,3,4,5}
> array double c[N][N]
> covariance(c, x)
> c
1.6667 0.5000 1.6667
0.5000 0.2500 0.500000
1.6667 0.5000 1.6667
> corrcoef(c, x)
1.0000 0.7746 1.0000
0.7746 1.0000 0.7746
1.0000 0.7746 1.0000
```

Function **correlation**2() with the prototype of

```
int correlation2(array double x[&][&], array double y[&][&])
```

calculates the correlation coefficient of two square matrices *x* and *y*. The correlation coefficient is defined as

$$c = \frac{\sum_{i=1}^{n}\sum_{j=1}^{n}(xx_{ij} * yy_{ij})}{\sqrt{(\sum_{i=1}^{n}\sum_{j=1}^{n} xx_{ij}^2) * (\sum_{i=1}^{n}\sum_{j=1}^{n} yy_{ij}^2)}}$$

where

$$xx_{ij} = x_{ij} - \mu_x$$
$$yy_{ij} = y_{ij} - \mu_y$$

μ_x and μ_y are the mean values of the matrices *x* and *y*.

$$\mu_x = \frac{1}{n * n}\sum_{i=1}^{n}\sum_{j=1}^{n} x_{ij}$$
$$\mu_y = \frac{1}{n * n}\sum_{i=1}^{n}\sum_{j=1}^{n} y_{ij}$$

Table 24.3: Definition of Norms.

Mode	Norm Type	Algorithm
	Norm for Vectors	
"1"	1-norm	$\|\|a\|\|_1 = \|a_1\| + \|a_1\| + ... + \|a_n\|$
"2"	2-norm	$\|\|a\|\|_2 = (\|a_1\|^2 + \|a_2\|^2 + ... + \|a_n\|^2)^{1/2}$
"p"	p-norm	$\|\|a\|\|_p = (\|a_1\|^p + \|a_2\|^p + ... + \|a_n\|^p)^{1/p}$ "p" is a floating-point number.
"i"	infinity norm	$\|\|a\|\|_\infty = \max_i \|a_i\|$
"-i"	negative infinity norm	$\|\|a\|\|_{-\infty} = \min_i \|a_i\|$
	Norm for m-by-n Matrices	
"1"	1-norm	$\|\|a\|\|_1 = \max_j \sum_{i=1}^{m} \|a_{ij}\|$
"2"	2-norm	$\|\|a\|\|_2$ = maximum singular value of a
"i"	infinity norm	$\|\|a\|\|_\infty = \max_i \sum_{j=1}^{m} \|a_{ij}\|$
"f"	Frobenius norm	$\|\|a\|\|_F^2 = \sum_{i=1}^{m} \sum_{j=1}^{n} \|a_{ij}\|^2$
"m"	norm	$\|\|a\|\| = \max(\mathbf{abs}(A[i][j]))$

For example,

```
> array double x[3][3]={1,2,3, 3,4,5, 6,7,8}
> array double y[3][3]={3,2,2, 3,8,5, 6,2,5}
> correlation2(x, y)
0.3266
```

24.2.10 Norm

The norm of a vector or matrix is a scalar that gives some measure of the magnitude of the elements of the vector or matrix. Function **norm**() with the prototype of

```
double norm(array double complex &a, char *char);
```

calculates norms of different types for a vector or matrix *a* according to the argument *mode*. The mode and algorithm of various norms for both vectors and matrices are defined in Table 24.3.

For example,

```
> array double a[6] = {1, 2, 3, 6, 5, 4}
> array double b[2][3] = {1, 2, 3, 6, 5, 4}
> norm(a, "1")
21.0000
> norm(b, "1")
7.0000
```

24.2.11 Factorial

The factorial is defined as

$$f = n!$$

Function **factorial**() with the prototype of

```
unsigned long long factorial(unsigned int n);
```

can be used to calculate a factorial.

For example, factorial 3! can be evaluated by the following command.

```
> factorial(3)
6
```

24.2.12 Combination

Function **combination**() with the prototype of

```
unsigned long long combination(unsigned int n, unsigned int k);
```

can be used to calculate the number of combination of *n* different things taken *k* at a time without repetitions. The number of combination is the number of sets that can be made up from *n* things, each set containing *k* different things and no two sets containing exactly the same *k* things. It is defined as

$$C_k^n = \frac{n!}{(n-k)!k!}$$

For example, combination C_5^2 can be evaluated by the following command.

```
> combination(5, 2)
10
```

24.2.13 Sort Data

Section 24.2.3 described how to find the minimum or maximum value, as well as its location, of arrays or a list of variables. Functions **minloc()** and **maxloc()** find locations of the mimimum and maximum values with offset 0. Data can be sorted by standard C function **qsort**() defined in header file **stdlib.h**. In this section, sorting data stored in arrays will be described.

Function **findvalue**() with the prototype of

```
int findvalue(array int y[&], array double complex x[&]);
```

finds indices of nonzero elements of an array. Vector *y* is integer data type with the total number of elements of *x*. It contains the indices of nonzero elements of array *x*. The remaining elements of *y* contains the value of -1. If *x* is an array of complex data type, a zero element is defined as a value with zero in both real and imaginary parts. Function **findvalue**() returns the number of nonzero elements of array *x*.

For example,

```
> array double x[5]={0,43.4,-7,-2.478,7}
> array int y[5]
> findvalue(y,x)
4
> y
1 2 3 4 -1
> findvalue(y,x<0)
2
> y
2 3 -1 -1 -1
```

Function **sort**() with the prototype of

```
int sort(array double complex &y, array double complex &x, ...
             /* [string_t method], [array int &ind] */);
```

sorts and ranks elements in ascending order. The original data in array *x* can be of any supported arithmetic data type and dimension. Array *y* is the same data type and size as *x* and it contains sorted data. If *x* is a complex data type, it is sorted by the magnitude of each element. If *x* includes NaN or ComplexNaN elements, **sort**() places these at the end. The index in array *ind* contains the ranking index, starting with 0, corresponding to array *x* and the optional argument *method* specifies the sorting method. When *x* is a two-dimensional array, this parameter specifies the sorting method defined as follows: "array" - sorted by total elements; "row" - sorted by row for two-dimensional array; "column" - sorted by column for two-dimensional array. By default, two-dimensional arrays are sorted by total elements. If *x* is not a two-dimensional array, the array is sorted by total elements. Function **sort**() returns 0 on success and -1 on failure.

For example,

```
> array double x[4] = {0.1, NaN, -0.1, 3}, y[4]
> sort(y, x)
> y
-0.1000 0.1000 3.0000 NaN
> array double x2[2][3] = {5.0, NaN, -3.0, -6.0, 4.0, 3.0}, y2[2][3]
> array int ind[2][3];
> sort(y2, x2, "array", ind)
> y2
-6.0000 -3.0000 3.0000
4.0000 5.0000 NaN
> ind
3 2 5
4 0 1
> sort(y2, x2, "row", ind)
> y2
-3.0000 5.0000 NaN
-6.0000 3.0000 4.0000
> ind
2 0 1
0 2 1
```

24.2.14 Unwrap

Function **unwrap**() with the prototype of

```
int unwrap(array double &y, array double &x, ...
          /* [double cutoff] */);
```

unwraps the radian phase of each element of input array *x* by changing its absolute jump greater than π to its 2π complement. The input array *x* can be of a vector or a two-dimensional array. If it is a two-dimensional array, the function unwraps it through every row of the array.

Array argument *y* is the same dimension and size as *x*. It contains the unwrapped data. Optional argument

```
#include <math.h>
#include <complex.h>
#include <chplot.h>

int main(){
    double r[1:4],theta1,theta31;
    int n1=2,n2=4, i;
    double complex z,p,rb;
    double  x1,x2,x3,x4;
    array double theta2[36],theta4[36],theta41[36];
    class CPlot subplot, *plot;

    /* four-bar linkage*/
    r[1]=5; r[2]=1.5; r[3]=3.5; r[4]=4;
    theta1=30*M_PI/180;
    linspace(theta2,0,2*M_PI);
    for (i=0;i<36;i++) {
       z=polar(r[1],theta1)-polar(r[2],theta2[i]);
       complexsolve(n1,n2,r[3],-r[4],z,x1,x2,x3,x4);
       theta4[i] = x2;
    }
    unwrap(theta41, theta4);
    subplot.subplot(2,1);
    plot = subplot.getSubplot(0,0);
    plot->data2D(theta2, theta4);
    plot->title("Wraped");
    plot->label(PLOT_AXIS_X,"Crank input: radians");
    plot->label(PLOT_AXIS_Y,"Rocker output: radians");

    plot = subplot.getSubplot(1,0);
    plot->data2D(theta2, theta41);
    plot->title("Unwraped");
    plot->label(PLOT_AXIS_X,"Crank input: radians");
    plot->label(PLOT_AXIS_Y,"Rocker output: radians");
    subplot.plotting();
}
```

Program 24.1: A program using **unwrap**().

cutoff specifies the jump value. If the user does not specify this input, *cutoff* has a value of π by default. Function **unwrap** returns 0 on success and -1 on failure.

For example, in motion analysis of a crank-rocker mechanism using Program 24.1, the output range of the rocker is within $0 \sim 2\pi$. For this mechanism, the output may be as shown on the top part in Figure 24.1. There is a jump when θ_4 is π, because $\theta_4 = \pi$ and $\theta_4 = -\pi$ are the same point for the crank-rocker mechanism. If the **unwrap**() function is used, a smooth curve for output angle θ_4 can be obtained as shown on the lower part in Figure 24.1.

24.2.15 Functions Applied to Elements of Arrays

Function **fevalarray**() with the prototype of

```
int fevalarray(array double &y, double (*func)(double),
    array double &x, ... /* [array int &mask, double value] */);
```

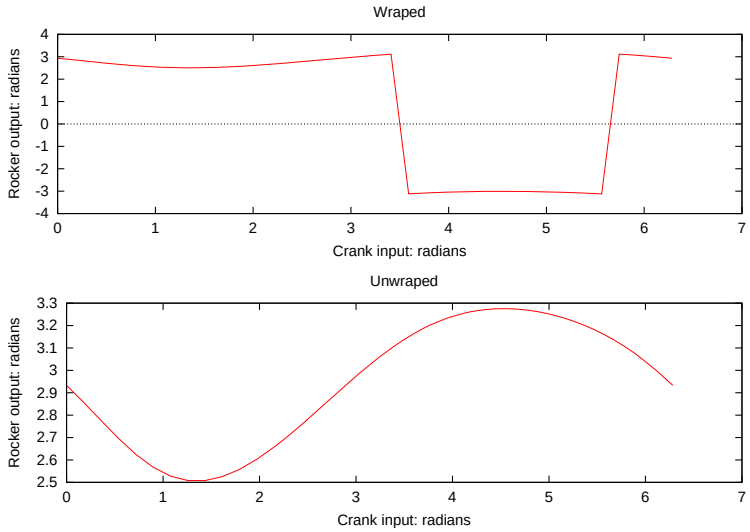


Figure 24.1: Comparison of results with and without using **unwrap**() function.

evaluates a user function applied to each element of array *x*. The input array can be of any arithmetic data type and dimension.

Array argument *y* contains the result of function evaluation. Argument *func* is a pointer to function given by the user. If the value of an element of array *mask* is 1, the function evaluation will be applied to that element. If the value of an element of array *mask* is 0, the value from the optional fifth argument *value* will be used for that element. Function **fevalarray**() returns 0 on success and -1 on failure.

For example,

```
> double func(double x) {return x*x;}
> array double x[4] = {1, 2, 3, 4}, y[4]
> array int mask[4] = {1, 0, 1, 0}
> fevalarray(y, func, x);
> y
1.0000 4.0000 9.0000 16.0000
> fevalarray(y, func, x, mask, 5);
> y
1.0000 5.0000 9.0000 5.0000
```

For arrays of complex numbers, function **cfevalarray**() with the prototype of

```
int cfevalarray(array double complex &y,
    double complex (*func)(double complex), array double complex &x,
    ... /* [array int &mask, double complex value] */);
```

shall be used.

24.2.16 Histogram

Function **histogram**() with the prototype of

```
int histogram(array double &y, array double x[&], ...
              /* [array double hist[:]] */);
```

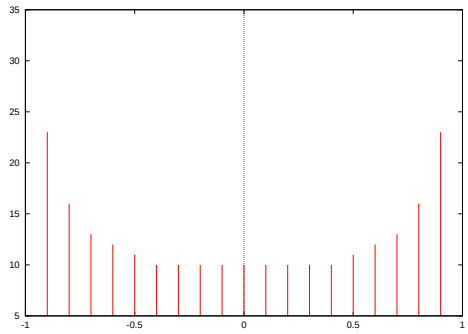


Figure 24.2: The histogram for sin(x) in the range of $-\pi <= x <= \pi$.

can be used to calculate the histogram of data set *y* with the bins defined in array *x*. The array of data set *y* can be of any dimensions and real type. If the function is called without the optional argument *hist*, the histogram will be plotted. Otherwise the function only generates the histogram data saved in array *hist* without plotting.

For example, the following commands can be typed

```
> array double yy[300], xx[300], x[21]
> lindata(-1, 1, x)
> lindata(-3.14, 3.14, xx)
> yy = sin(xx)
> histogram(yy, x)
```

The output from the above commands is displayed in Figure 24.2

24.3 Data Interpolation and Curve Fitting

24.3.1 One-Dimensional Interpolation

Function **interp1**() with the prototype of

```
int interp1(double y[&], double x[&], double xa[&], double ya[&],
            char *method);
```

finds the values of the function, which is expressed in terms of two arrays *xa* and *ya*, at points expressed in array *x* by linear or cubic spline interpolation. The interpolation method can be chosen by input argument *method* of a string with `"linear"` or `"spline"`. The function returns 0 on success and -1 on failure.

For example, the following commands are valid.

```
> array double x[1]=0.5, y[1], xa[180],ya[180]
> lindata(-3.14, 3.14, xa)
> ya = sin(xa)
> interp1(y, x, xa, ya, "spline")
> y
0.4794
```

Figure 24.3, generated by Program 24.2, displays the original data and interpolated points using function **interp1**().

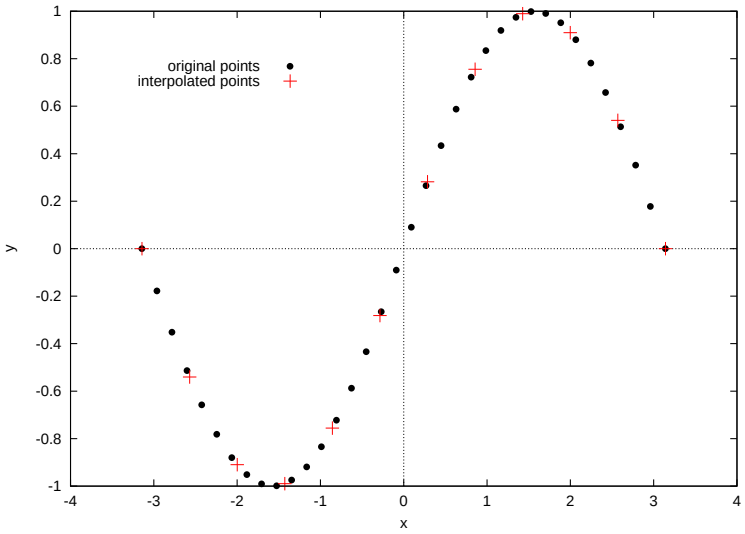


Figure 24.3: The original data and interpolated points using function **interp1**().

```
#include <chplot.h>
#include <math.h>

int main() {
    array double xa[36], ya[36];
    array double x[12], y[12];
    class CPlot plot;
    int numdataset=0, pointtype=7, pointsize=1;

    linspace(xa, -M_PI, M_PI);
    linspace(x,  -M_PI, M_PI);
    ya = sin(xa);
    interp1(y, x, xa, ya, "spline");

    plot.data2D(xa, ya);
    plot.plotType(PLOT_PLOTTYPE_POINTS, numdataset, pointtype, pointsize);
    plot.legend("original points", 0);
    numdataset=1, pointtype=1, pointsize=2;
    plot.data2D(x, y);
    plot.plotType(PLOT_PLOTTYPE_POINTS, numdataset, pointtype, pointsize);
    plot.legend("interpolated points", 1);
    plot.legendLocation(-1, 0.8);
    plot.plotting();
}
```

Program 24.2: A program using function **interp1**().

24.3.2 Two-Dimensional Interpolation

Function **interp2**() with the prototype of

```
int interp2(double z[&][&], double x[&], double y[&],
    double xa[&], double ya[&], double za[&][&], char *method);
```

finds the values of a two-dimensional function at points indicated by two one-dimensional arrays *x* and *y*, by two-dimensional linear or cubic spline interpolation. The function is expressed in terms of tabulated values, in two arrays *xa* of dimension *m* and *ya* of dimension *n* , and a matrix of function value *za* of dimension $m \times n$ tabulated at the grid points defined by *xa* and *ya*. The dimensions for arrays *xa*, *ya*, *x* and *y* can be different. The interpolation method is chosen by input argument *method* of string with `"linear"` or `"spline"`. The function returns 0 on success and -1 on failure.

Program 24.3 interpolates a two-dimensional function using cubic splines, $z(x,y) = 3(1-x)^2 e^{-x^2-y^2+1} - 10(\frac{x}{5} - x^3 - y^5)e^{-x^2-y^2} - \frac{1}{3}e^{-(x+1)^2-y^2}$ The two-dimensional arrays `z` and za are cast to one-dimensional arrays `z1` and za1 so that they can be used by plotting member function **CPlot**::**data3D**(). The output from Program 24.3 is displayed in Figure 24.4

24.3.3 General Curve Fitting

The general form of a fitting formula is

$$y(x) \quad = \quad \sum_{k=1}^{M} a_k f_k(x)$$

where $f_1(x), ..., f_M(x)$ are arbitrary fixed functions of *x*, called base functions. $f_k(x)$ can be a nonlinear function of *x*. The coefficients a_k are determined by minimizing chi-square which is defined as

$$\chi^2 = \sum_{i=1}^{N} \left(\frac{y_i - \sum_{k=1}^{M} a_k f_k(x_i)}{\sigma_i} \right)^2$$

where σ_i is the standard deviation of the ith data point. If the standard deviations are unknown, they can be set to the constant value $\sigma = 1$. By defining a $n \times m$ matrix α with element (k, j) as

$$\alpha_{kj} = \sum_{i=1}^{N} \frac{f_j(x_i) f_k(x_i)}{\sigma_i^2}$$

The element (k, j) of the covariance matrix can be expressed as

$$cover_{kj} = [\alpha]_{kj}^{-1}$$

Function **curvefit**() with the prototype of

```
int curvefit(double a[&], double x[&], double y[&],
    void (*funcs)(double, double []), ...
  /* double sig[], int ia[], double covar[:][:], double *chisq] */);
```

uses χ^2 minimization to fit for some or all of the coefficients in array *a* of a function that depends linearly on $y = a_i * funcs_i(x)$, given a set of data points in arrays *x* and *y* with the same dimension and individual standard deviations in array *sig*. The program can also pass χ^2 and covariance matrix *covar*. If the values

```
#include <chplot.h>
#include <math.h>
#include <numeric.h>
#define M 20
#define N 30
#define NUM_X 40
#define NUM_Y 50

int main() {
    int datasetnum = 0, i,j;
    array double za1[M*N],za[M][N],xa[M],ya[N];
    array double z1[NUM_X*NUM_Y],z[NUM_X][NUM_Y],x[NUM_X],y[NUM_Y];
    class CPlot plot;

    /* Construct data set of the peaks function */
    linspace(xa, -3, 3);
    linspace(ya, -4, 4);
    for(i=0; i<M; i++) {
        for(j=0; j<N; j++) {
            za[i][j] = 3*(1-xa[i])*(1-xa[i])*
                       exp(-(xa[i]*xa[i])-(ya[j]+1)*(ya[j]+1))
                       - 10*(xa[i]/5 - xa[i]*xa[i]*xa[i]-
                       pow(ya[j],5))*exp(-xa[i]*xa[i]-ya[j]*ya[j])
                       - 1/3*exp(-(xa[i]+1)*(xa[i]+1)-ya[j]*ya[j]);
        }
    }
    linspace(x, -3, 3);
    linspace(y, -4, 4);
    interp2(z,x,y,xa,ya,za,"spline");

    za1 = (array double[M*N])za;
    /* add offset for display */
    z1 = (array double[NUM_X*NUM_Y])z + (array double[NUM_X*NUM_Y])50;
    plot.data3D(xa, ya, za1);
    plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum++);
    plot.data3D(x, y, z1);
    plot.plotType(PLOT_PLOTTYPE_LINES, datasetnum++);
    plot.ticsLevel(0);
    plot.text("spline", PLOT_TEXT_RIGHT,4,4.5,55);
    plot.text("original", PLOT_TEXT_RIGHT,4,4.5,5);
    plot.colorBox(PLOT_OFF);
    plot.plotting();
}
```

Program 24.3: A program using **interp2**().

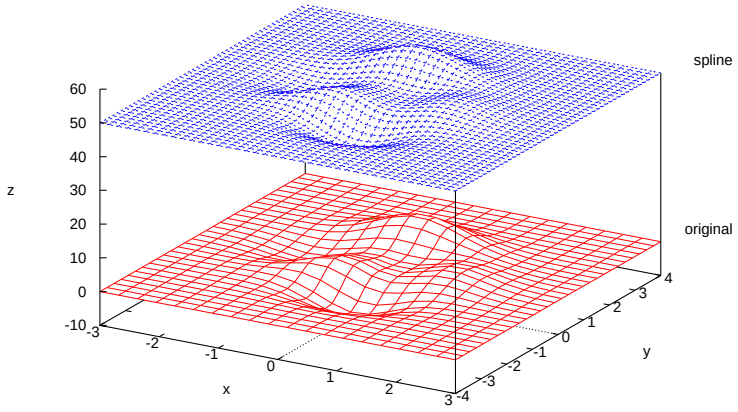


Figure 24.4: The result from two-dimensional interpolation function **interp2**().

```
#include <stdio.h>
#include <math.h>
#include <numeric.h>
#define NPT 100
#define SPREAD 0.1
#define NTERM 4

void funcs(double x, double func[]) {
    func[0]=cos(x);
    func[1]=sin(x);
    func[2]=exp(x);
    func[3]=1.0;
}
int main(void) {
    int i,j;
    array double a[NTERM],x[NPT],y[NPT],sig[NPT],u[NPT];

    linspace(x, 0, 10);
    urand(u);
    y = 4*cos(x) + 3*sin(x) +2*exp(x) +(array double [NPT])1.0 + SPREAD*u;
    curvefit(a,x,y,funcs);
    printf("    %s\n","parameters");
    for (i=0;i<NTERM;i++)
       printf("    a[%d] = %f\n", i,a[i]);
}
```

Program 24.4: A program using **curvefit**().

for array argument *ia* are constants, the corresponding components of the covariance matrix will be zero. The user supplies a routine *funcs*(*x*,*func*) that passes the function values evaluated at *x* by the array *func*. The standard deviations in array *sig* can be set to 1 if they are unknown. The function returns 0 on success and -1 on failure.

Program 24.4 fits data points generated by $f(x) = 4\cos(x) + 3\sin(x) + 2e^x + 1$ with a uniform random deviation to the base functions $\cos(x)$, $\sin(x)$, e^x, and 1. The curve fitting function **curvefit**(*a*, *x*, *y*, *funcs*) is called in this example. The output from this program is given as follows:

```
parameters
a[0] = 3.990255
a[1] = 2.994660
a[2] = 2.000000
a[3] = 1.051525
```

24.3.4 Curve Fitting Using Polynomial Functions

The algorithm of the function **polyfit**() is based on that of the function **curvefit**(). For polynomial fitting, the base functions for **curvefit**() are set to the terms of polynomials internally. The general form of a fitting formula is

$$y(x) \quad = \quad \sum_{k=1}^{M} a_k x^k$$

Function **polyfit**() with the prototype of

```
int polyfit(double a[&], double x[&], double y[&],
  /* double sig[], int ia[], double covar[:][:], double *chisq] */);
```

uses χ^2 minimization to fit for polynomial coefficients in array *a*, given a set of data points in arrays *x* and *y* with the same dimension and individual standard deviations in array *sig*. The program can also pass χ^2 and covariance matrix *covar*. If the values for array argument *ia* are constants, the corresponding components of covariance matrix will be zero. The standard deviations in array *sig* can be set to 1 if they are unknown. The function returns 0 on success and -1 on failure.

Program 24.5 fits data points generated by the polynomial

$$8x^4 + 5x^3 + 3x^2 + 6x + 7$$

with a uniform random deviation. The polynomial curve fitting function **polyfit**(*a*, *x*, *y*) is used to obtain the coefficients in array *a* of a fourth order polynomial. The output from this program is given below.

```
Coefficients
a[0] =  8.000048
a[1] =  4.999537
a[2] =  2.998743
a[3] =  6.015915
a[4] =  7.033636
y = 199.057500 at x = 2.000000
```

24.4 Minimization or Maximization of Functions

In this section, minimization of functions will be described. The maximum value of a function $f(x)$ can be obtained by finding the minimum value of function $-f(x)$.

```
#include <stdio.h>
#include <numeric.h>
#define NPT 100                 /* Number of data points */
#define NTERM 5                 /* Number of terms */

int main() {
    int i,j,status;
    array double  u[NPT],x[NPT],y[NPT],a[NTERM];

    /* Create a data set of NTERM order polynomial with uniform random deviation*/
    linspace(x,0.1,0.1*NPT);
    y = 8*x.*x.*x.*x + 5*x.*x.*x + 3*x.*x + 6*x  + (array double[NPT])(7);
    urand(u);
    y += 0.1*u;

    status=polyfit(a,x,y);
    if(status) printf("Abnormal fit");
    printf("    %s\n","Coefficients");
    for (i=0;i<NTERM;i++)
        printf("    a[%1d] =  %8.6f \n",i,a[i]);
    printf("    y = %f at x = %f\n", polyeval(a, 2.0), 2.0);
}
```

Program 24.5: A program using **polyfit**().

24.4.1 Minimization of Function with One Variable

Function **fminimum**() with the prototype of

```
int fminimum(double *fminval, double *xmin, double (*func)(double),
double x0, double xf, ...  /* [double rel_tol], [double abs_tol]*/);
```

finds the minimum value of a function with one variable and its corresponding position for the minimum value.

The function determines a point between *x0* and *xf* where the real function *func* assumes a minimum value. The function *func* given by the user has an argument for *x* value input. The argument *fminval* passes the calculated minimum value of the function. The argument *xmin* contains the position where the minimum value of the function is found. The tolerance is defined as a function of x: $|x|rel_tol + abs_tol$, where *rel_tol* is the relative precision and *abs_tol* is the absolute precision (which should not be zero). The default value for *rel_tol* and *abs_tol* is 10^{-6}. Function **fminimum**() returns 0 on success and -1 on failure.

In the interval where the function has a minimum value, it is assumed that for some point *u* either (a) *func* is strictly monotonically decreasing on [a,u) and strictly monotonically increasing on [u,b] or (b) these two intervals may be replaced by [a,u] and (u,b], respectively.

For example, the commands below can find the minimum of the function $f(x) = -\frac{1}{(x-0.3)^2+0.01} - \frac{1}{(x-0.9)^2+0.04} + 6$ shown in Figure 24.5 at intervals [-1,0], [0,0.8], [0.4,1] and [0,1] with the default value for tolerance. For interval [0,1] this function can only find the local minimum at $x = 0.892716$.

```
> double xmin, fminval
> double func(double x) { return -1/((x-0.3)*(x-0.3)+0.01) -\
  1/((x-0.9)*(x-0.9)+0.04) + 6;}
> fminimum(&fminval, &xmin, func, -1, 0)
```

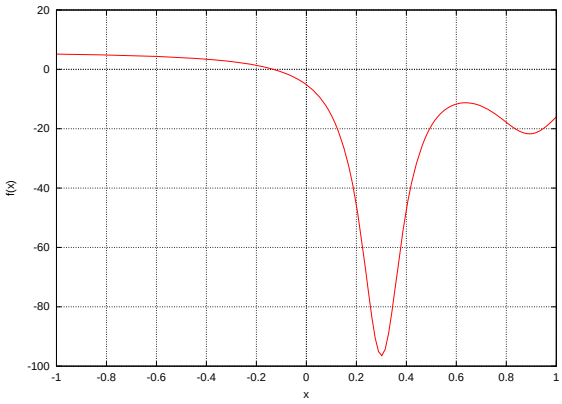


Figure 24.5: Function $f(x) = -\frac{1}{(x-0.3)^2+0.01} - \frac{1}{(x-0.9)^2+0.04} + 6.$

```
> xmin
0.0000
> fminval
-5.1765
> fminimum(&fminval, &xmin, func, 0, 0.8)
> xmin
0.3003
> fminval
-96.5014
> fminimum(&fminval, &xmin, func, 0.4, 1)
> xmin
0.4000
> fminval
-47.448l
> fminimum(&fminval, &xmin, func, 0, 1)
> xmin
0.8927
> fminval
-21.7346
```

24.4.2 Minimization of Function with Multiple Variables

Function **fminimums**() with the prototype of

```
int fminimums(double *fminval, double xmin[&],
      double (*func)(double[]), double x0[&], ...
      /* [double rel_tol], [double abs_tol], [int numfuneval] */);
```

finds the minimum value of a function with multiple variables and its corresponding position for the minimum value. Given an n-dimensional function and an array which contains an initial estimate by the user as input, this function calculates the array of the position at which the function has a minimum value. The number of dimensions is taken from input array *x* internally. The function *func* should deliver the value of the function to be minimized, at the points given by *x*. The tolerance is defined as a function of *x* as $|x|rel_tol + abs_tol$, where *rel_tol* is the relative precision and *abs_tol* is the absolute precision (which

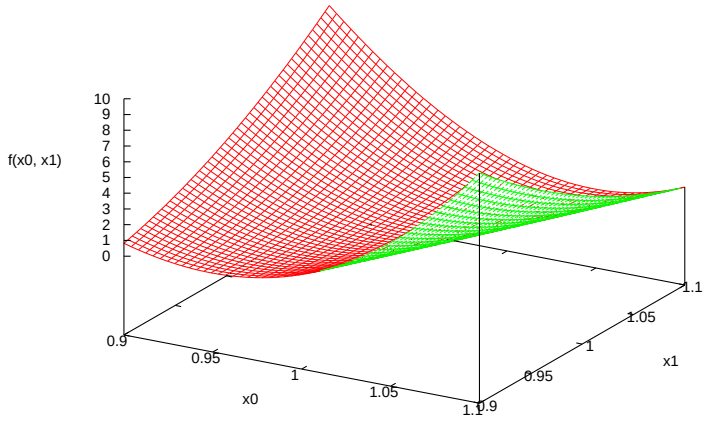


Figure 24.6: Function $f(x_0, x_1) = 100(x_1 - x_0^2)^2 + (1.0 - x_0)^2$.

should not be zero). The argument *numfunceval* is the maximum number of function evaluations allowed. The default values for *rel_tol, abs_tol* and *numfuneval* are $10^{-6}, 10^{-6}$, and 250, respectively. Function **fminimums**() returns 0 on success and a negative value on failure.

For example, function $f(x_0, x_1) = 100(x_1 - x_0^2)^2 + (1.0 - x_0)^2$ shown in Figure 24.6 has a minimum value of 0 at $\mathbf{x} = (1, 1)$. This minimum value and its position can be found by the following commands using function **fminimums**() with the initial guess of $\mathbf{x_0} = (-1.2, 1.0)$.

```
> double fminval;
> array double xmin[2], x0[2]= {-1.2, 1.0};
> double func(double x[]) { return 100*(x[1]-x[0]*x[0])* \
  (x[1]-x[0]*x[0]) + (1.0-x[0])*(1.0-x[0]);}
> fminimums(&fminval, xmin, func, x0);
> xmin
1.0000 1.0000
> fminval
0.0000
```

24.5 Polynomials

Functions related to polynomials are described in this section. In Ch, a polynomial is represented by an array for its coefficients in the order of descending powers. For example, a polynomial of

$$p_0 x^n + p_1 x^{n-1} + ... + p_{n-1} x + p_n$$

can be represented by an array with $(n+1)$ elements of $[p_0, p_1, ..., p_{n-1}, p_n]$ Fitting a curve using a polynomial by function **polyfit**() has been presented in section 24.3.4. Evaluation of matrix polynomials of matrix using **polyevalm**() is described in section 24.10.4. Section 24.15 will discuss how to use functions **conv**() and **deconv**() for multiplication and division of two polynomials, respectively.

24.5.1 Evaluation of Polynomials

Function **polyeval**() with the prototype of

```
double polyeval(array double p[&], double x, ...
                /* array double dp[&] */);
```

evaluates a polynomial and its derivatives at *x* . The polynomial is represented by array *p* using its coefficients. Function **polyeval**() returns the value of the polynomial at *x*. If the optional array argument *dp* of double type and dimension m is passed, it will contain the values of 1st to mth order derivatives of the polynomial.

For example, the value of polynomial $P(x) = x^5 - 5x^4 + 10x^3 - 10x^2 + 5x - 1$ at point $x = 0.1$ and its first, second and third order derivatives can be obtained by the following commands.

```
> array double p[6] = {1.0, -5.0, 10.0, -10.0, 5.0, -1.0}
> array double dp[3];
> polyeval(p, 0.1, dp);
-0.5905
> dp
3.2805 -14.5800 48.6000
```

If the coefficients or variable of a polynomial are complex numbers, function **cpolyeval**() with the prototype of

```
double complex cpolyeval(array double complex p[&], double complex x);
```

shall be used for evaluation of the polynomial.

Function **polyevalarray**() with the prototype of

```
int polyevalarray(array double complex &y, array double complex p[&],
                  array double complex &x);
```

evaluates a polynomial at a sequence of points. The vector *p* with the coefficients of a polynomial can be of any supported arithmetic data type and size. The value of *x* can be any arithmetic type. Conversion of the data to double complex is performed internally. Array *val*, the same dimension and size as *x*, contains the values of the polynomial at points represented by array *x*. If both *c* and *x* are real type, *val* is real type. Otherwise, it shall be complex type. Function **polyevalarray**() returns 0 on success and -1 on failure.

For example, the polynomial $P(x) = x^5 - 5x^4 + 10x^3 - 10x^2 + 5x - 1$ at points $\mathbf{x} = \{0, 0.5, 1.0, 1.5, 2.0\}$ can be obtained by the following commands.

```
> array double p[6] = {1.0, -5.0, 10.0, -10.0, 5.0, -1.0}
> array double val[5], x[5]= {0, 0.5, 1, 1.5, 2}
> polyevalarray(val, p, x)
> val
-1.0000 -0.0313 0.0000 0.0313 1.0000
```

If the variable x in a polynomial is a square matrix, the evaluation of such a matrix polynomial can be performed by function **polyevalm**(), which will be described in section 24.10.4.

24.5.2 Derivative of Polynomials

For a polynomial $P(t)$ with coefficients of x_i for $i = 0, 1, \cdots, N$ as

$$P(t) = x_0 t^n + x_1 t^{n-1} + x_2 t^{n-2} + \cdots + x_{n-1} t + x_n,$$

the derivative of this polynomial is

$$\begin{aligned} P'(t) &= n * x_0 t^{n-1} + (n-1) * x_1 t^{n-2} + (n-2) * x_2 t^{n-3} + \cdots + x_{n-1} \\ &= y_0 t^{n-1} + y_1 t^{n-1} + y_2 t^{n-3} + \cdots + y_{n-1} \end{aligned}$$

Function **polyder**() with the prototype of

```
int polyder(array double complex y[&], array double complex x[&]);
```

obtains the coefficients of the derivative $P'(x)$ of polynomial $P(x)$. The vector of the coefficients for polynomial *x* can be of any supported arithmetic type and size. Conversion of the data to double complex type is performed internally. If vector *x* is real type with size n, the vector *y* for derivative is real type with size $(n-1)$. If vector *x* is complex type, the vector *y* for derivative is complex type.

For example, the derivative of polynomial $P(x) = x^5 - 5x^4 + 10x^3 - 10x^2 + 5x - 1$ is $P'(x) = 5x^4 - 20x^3 + 30x^2 - 20x + 5$. The coefficients of $P'(x)$ can be obtained by the following commands.

```
> array double x[6] = {1.0, -5.0, 10.0, -10.0, 5.0, -1.0}
> array double y[5]
> polyder(y, x)
> y
5.0000 -20.0000 30.0000 -20.0000 5.0000
```

Given two polynomials $U(x)$ and $V(x)$

$$\begin{aligned} U(x) &= u_0 x^n + u_1 x^{n-1} + u_2 x^{n-2} + \cdots + u_{n-1} x + u_n \\ V(x) &= v_0 x^m + v_1 x^{m-1} + v_2 x^{m-2} + \cdots + v_{m-1} x + v_m \end{aligned}$$

The derivative of the product of two polynomials $U(x) * V(x)$ becomes

$$Q(x) = (U(x)V(x))' = U'(x)V(x) + V'(x)U(x)$$

The derivative of the quotient of two polynomials $U(x)/V(x)$ becomes

$$\left(\frac{U(x)}{V(x)}\right)' = \frac{Q(x)}{R(x)} = \frac{U'(x)V(x) - V'(x)U(x)}{V^2(x)}$$

where $Q(x)$ and $R(x)$ are the numerator and denominator of the derivative, respectively. The polynomials for $Q(x)$, applicable to the derivative of either the product or quotient of two polynomials, and $R(x)$ can be represented as

$$\begin{aligned} Q(x) &= q_0 x^{n+m-1} + q_1 x^{n+m-2} + \cdots + q_{n+m-2} x + q_{n+m-1} \\ R(x) &= r_0 x^{2m-1} + r_1 x^{2m-2} + \cdots + r_{2m-2} x + r_{2m-1} \end{aligned}$$

Function **polyder**2() with the prototype of

```
int polyder2(array double complex q[&], array double complex r[&],
             array double complex u[&], array double complex v[&]);
```

can be used to calculate the coefficients of $Q(x)$ and $R(x)$ from the derivative of product or quotient of two polynomials $U(x)$ and $V(x)$. The algorithm inside function **polyder**2() for derivative of product or quotient of two polynomials *u* and *v* depends on argument *r*. If NULL is passed to argument *r*, function **polyder**2() calculates the derivative of the product $(u * v)'$ of two polynomials *u* and *v*. Otherwise, it calculates the derivative of the quotient $(u/v)'$. The coefficient vector of polynomials *u* and *v* can be of any supported arithmetic data type with sizes n and m, respectively. Conversion of the data to double complex type is performed internally. If both vectors *u* and *v* are real type, vector *q* with size $n + m - 2$ and *r* with size $2 * m - 1$ are real type. If either vector *u* or *v* is complex type, vectors *q* and *r* are complex type.

For example, given two polynomials

$$\begin{aligned} U(x) &= x + 2 \\ V(x) &= x^2 + 2x \end{aligned}$$

The coefficients of the derivatives of product and quotient of these two polynomial are

$$\begin{aligned} (U(x)V(x))' &= 3x^2 + 8x + 4 \\ \left(\frac{U(x)}{V(x)}\right)' &= \frac{-x^2 - 4x - 4}{x^4 + 4x^3 + 4x^2} \end{aligned}$$

These coefficients of polynomials can be obtained by the following commands.

```
> int n=2, m = 3
> array double u[2] = {1.0, 2}, v[3] = {1, 2, 0}
> array double q[2+3-2], r[2*3-1]
> polyder2(q, NULL, u, v)
> q
3.0000 8.0000 4.0000
> polyder2(q, r, u, v)
> q
-1.0000 -4.0000 -4.0000
> r
1.0000 4.0000 4.0000 0.0000 -0.0000
```

24.5.3 Find Roots of Polynomials

Function **roots**() with the prototype of

```
int roots(array double complex x[&], array double complex p[&]);
```

finds the roots of a polynomial $p_0x^n + p_1x^{n-1} + ... + p_{n-1}x + p_n$. The function can handle polynomials with coefficients of real type or complex type. The arguments *p* and *x* contain the coefficients and roots of a polynomial, respectively. The function returns 0 on success and -1 on failure.

For example, the roots of polynomial $p = x^2 - 2x + 1$ can be determined by the following command-line executions.

```
> array double x[2]
> array double p[3] = {1, -2, 1}        /* p = x^2-2x+1 */
> roots(x, p)
```

```
> x
1.0000 1.0000
>
```

The **roots**() function can also calculate the complex roots of a polynomial. For example, the polynomial $x^4 - 12x^3 + 25x + 116$ has two roots of complex numbers and two roots of real numbers. The coefficients of the polynomial $(3+i4)x^4 + (4+i2)x^3 + (5+i3)x^2 + (2+i4)x + (1+i5)$ are complex numbers. The roots of these polynomials can be obtained by the following commands.

```
> array double x1[4], p1[5]  = {1, -12, 0, 25, 116}
> array double complex z1[4]
> roots(x1, p1)
> x1
11.7473 2.7028 NaN NaN
> roots(z1, p1)
> z1
complex(11.7473,0.0000) complex(2.7028,0.0000) \
complex(-1.2251,1.4672) complex(-1.2251,-1.4672)
> array double complex z2[4], p2[5] = {complex(3,4), complex(4,2),\
  complex(5,3), complex(2,4), complex(1,5)}
> roots(z2, p2)
> z2
complex(0.2263,1.2815) complex(0.4311,-0.7280) \
complex(-0.7541,-0.7078) complex(-0.7034,0.5543)
```

24.5.4 Find Coefficients of Polynomials

When roots to a polynomial equation are known, the polynomial can be expressed as the products of expressions $(x - x_i)$ as follows.

$$\begin{aligned} P(x) &= (x - x_0) * (x - x_1) * ... * (x - x_{n-1}) \\ &= p_0 x^n + ... + p_{n-1}x + p_n. \end{aligned}$$

When the roots x_i's of a polynomial are given, the coefficients p_i of the polynomial $P(x)$ can be obtained as by function **polycoef**(). Function **polycoef**() is prototyped as

```
int polycoef(array double complex p[&], array double complex x[&]);
```

Array arguments *x* are the given roots of a polynomial. Array argument *p* contains the calculated coefficients of the polynomial. Function **roots**() that finds the roots of a polynomial is complementary to function **polycoef**().

For example, the coefficients of polynomial

$$\begin{aligned} P(x) &= (x-1)(x-2)(x-3)(x-4) \\ &= x^4 - 10x^3 + 35x^2 - 50x + 24 \end{aligned}$$

can be obtained by the following commands.

```
> array double x[4]= {1, 2, 3, 4}
> array double p[5]
```

```
> polycoef(p, x)
> p
1.0000 -10.0000 35.0000 -50.0000 24.0000
> roots(x, p)
> x
3.0000 4.0000 2.0000 1.0000
```

24.5.5 Residues for Factorization of Polynomials

Given the ratio of two polynomials

$$\frac{U(s)}{V(s)} = \frac{u_0 s^m + u_1 s^{m-1} + \cdots + u_{m-1} s + u_m}{s^n + v_1 s^{n-1} + \cdots + v_{n-1} s + v_n}$$

If there are no multiple roots, the expansion becomes

$$\frac{U(s)}{V(s)} = \frac{r_0}{s - p_0} + \frac{r_1}{s - p_1} + \cdots + \frac{r_{n-1}}{s - p_{n-1}} + K(s)$$

If p_i is a pole of multiplicity l, then the expansion becomes

$$\frac{U(s)}{V(s)} = \frac{r_0}{s - p_0} + \frac{r_1}{s - p_1} + \cdots + \frac{r_i}{(s - p_i)^l} + \cdots + \frac{r_{i+1}}{(s - p_i)^2} + \frac{r_{i+l}}{s - p_i} + \cdots + \frac{r_{n-1}}{s - p_{n-1}} + K(s)$$

where $K(x)$ is the direct term. If $m > n$, $K(s)$ is

$$K(s) = k_0 s^{m-n} + k_1 s^{m-n-1} + \cdots + k_{m-n} s + k_{m-n+1}$$

Otherwise, $K(s)$ is empty.

Function **residue**() finds the residues, poles and direct term for a partial fraction expansion of the ratio of two polynomials $V(s)$ and $U(s)$. Function **residue**() is prototyped as

```
int residue(array double u[&], array double v[&],
            array double complex r[&], array double complex p[&],
            array double k[&]);
```

Vectors *u* of size m and *v* of size n specify the coefficients of the polynomials in descending powers of s. They can be any real type. Conversion of the data to double type is performed internally. The residues *r* of vector size $(n-1)$ and poles *p* of vector size $(n-1)$ are any compatible data type according to the residue computation. If the argument of real type is passed and the result is complex type, the value of NaN will be passed out. The direct term *k* with size of $(m-n+1)$ is always real type. If $m \leq n$, then *k* contains NULL.

For example, the following partial-fraction expansion with single roots of real numbers and without direct term

$$\frac{10s + 6}{2s^3 + 12s^2 + 22s + 12} = \frac{-1}{s+1} + \frac{7}{s+2} + \frac{-6}{s+3}$$

can be obtained by the commands below.

```
> int M =2, N =4
> array double u[2] = {10, 6}
> array double v[4] = {2, 12, 22, 12}
> array double r[N-1], p[N-1]
```

```
> residue(u, v, r, p, NULL)
> r
-1.0000 7.0000 -6.0000
> p
-1.0000 -2.0000 -3.0000
```

For the partial-fraction expansion with single roots of complex numbers and without direct term,

$$\frac{x+3}{x^2+2x+5} = \frac{0.5+i0.5}{s+(1+i2)} + \frac{0.5-i0.5}{s+(1-i2)}$$

the following commands can be used.

```
> int M =2, N =3
> array double u[2] = {1, 3}
> array double v[3] = {1, 2, 5}
> array double r[N-1], p[N-1]
> array double complex zr[N-1], zp[N-1]
> residue(u, v, r, p, NULL)
> r
NaN NaN
> p
NaN NaN NaN
> residue(u, v, zr, zp, NULL)
> zr
complex(0.5000,0.5000) complex(0.5000,-0.5000)
> zp
complex(-1.0000,-2.0000) complex(-1.0000,2.0000)
```

A partial-fraction expansion with single roots and has a direct term.

$$\frac{2s^3+12s^2+22s+12}{10s+6} = \frac{0.2688}{s+0.6} + 0.2s^2 + 1.08s + 1.552$$

```
> int M =4, N =2
> array double u[4] = {2, 12, 22, 12}
> array double v[2] = {10, 6}
> array double r[N-1], p[N-1], k[M-N+1]
> residue(u, v, r, p, k)
> r
0.2688
> p
-0.6000
> k
0.2000 1.0800 1.5520
```

A partial-fraction expansion with multiplicity roots and without direct term.

$$\frac{s^2+2s+3}{s^5+5s^4+9s^3+7s^2+2s} = \frac{3}{2(s+2)} - \frac{3}{(s+1)^3} - \frac{2}{s+1} + \frac{3}{2s}$$

Note that the numerator for term $(s+1)^2$ is zero.

```
> int M =3, N =6
> array double u[3] = {1, 2, 3}
> array double v[6] = {1, 5, 9, 7, 2}
> array double r[N-1], p[N-1]
> residue(u, v, r, p, NULL)
> r
1.5000 -3.0000 0.0000 -2.0000 1.5000
> p
-2.0000 -1.0000 -1.0000 -1.0000 0.0000
```

24.5.6 Characteristic Polynomials of Matrices

The characteristic polynomial

$$p_0 x^n + ... + p_{n-1} x + p_n$$

of matrix **A** is defined as the determinant of matrix $(x\mathbf{I} - \mathbf{A})$. The roots of the characteristic polynomial of matrix **A** are the eigenvalues of the matrix. Function **charpolycoef**() with the prototype of

```
int charpolycoef(array double complex p[&],
                 array double complex a[&][&]);
```

calculates the coefficients of the characteristic polynomial of a matrix passed as array *a*. Array argument *p* contains the calculated coefficients of the characteristic polynomial of the matrix. This function returns 0 on success and -1 on failure.

For example, the coefficients of the characteristic polynomial

$$x^3 - 2.1x^2 + 1.4x - 0.3$$

and eigenvalues (1, 0.5, 0.6) for matrix **A** below

$$\mathbf{A} = \begin{bmatrix} 0.8 & 0.2 & 0.1 \\ 0.1 & 0.7 & 0.3 \\ 0.1 & 0.1 & 0.6 \end{bmatrix},$$

can be obtained by the following commands.

```
> array double a[3][3] = {0.8,0.2,0.1, 0.1,0.7,0.3, 0.1,0.1,0.6}
> array double p[4], x[3]
> charpolycoef(p, a)
> p
1.0000 -2.1000 1.4000 -0.3000
> roots(x, p)
> x
1.0000 0.5000 0.6000
> polycoef(p, x)
> p
1.0000 -2.1000 1.4000 -0.3000
```

24.6 Nonlinear Equations

24.6.1 Solve a Nonlinear Equation

Function **fzero**() with the prototype of

```
int fzero(double *x, double (*func)(double), ...
          /* [double x0] | [double x02[2]*/);
```

finds a zero position of a nonlinear function with one variable. The argument *func* is a pointer to the function given by the user. The position where the function is zero is passed by argument *x*. Argument *x0* contains the initial guess for the zero position. Argument *x02* is a vector of length 2 and double type. The function shall be bracketed in the interval of [*x02*[0], *x02*[1]] so that the sign of *func*(*x02*[0]) differs from the sign of *func*(*x02*[1]). Otherwise, an error occurs. Function **fzero**() returns 0 on success and -1 on failure.

For example, the zero position 1.414213 of function

$$f(x) = x^2 - 2$$

can be obtained with an initial guess $x_0 = 2.0$ by the following commands.

```
> double x, func(double x) { return x*x-2.0;}
> fzero(&x, func, 2.0);
> x
1.4142
> double x02[2]={-2, 0}
> fzero(&x, func, x02);
> x
-1.4142
```

24.6.2 Solve System of Nonlinear Equations

A system of nonlinear equations can be found by function **fsolve**(). Function **fsolve**() has the prototype of

```
int fsolve(double x[:], void (*func)(double[], double []),
           double x0[:]);
```

Array arguments *x* and *x0* contain the calculated zero position and its initial guess, respectively. The user function has two arguments, first one for input and the second one for the values of the functions. The input argument is an n-dimensional array, and the function values calculated will be delivered by the second array argument of the same dimension. The number of dimensions is taken from the function given by the user internally. Function **fsolve**() returns 0 on success and -1 on failure.

For example, the following nonlinear system of two equations

$$\begin{aligned} f_0 &= -(x_0^2 + x_1^2 - 2.0) = 0 \\ f_1 &= e^{x_0 - 1.0} + x_1^3 - 2.0 = 0 \end{aligned}$$

has a zero point at (1, 1). It can be solved by function **fsolve**() with the initial guesses of zero point at $x_0 = 2.0$, and $x_1 = 0.5$ using the following commands.

```
> void func(double x[], double f[]){f[0]=-(x[0]*x[0]+x[1]*x[1]-2.0);\
  f[1]=exp(x[0]-1.0)+x[1]*x[1]*x[1]-2.0;}
> array double x[2], x0[2] = {2.0, 0.5};
> fsolve(x, func, x0);
> x
1.0000 1.0000
```

24.7 Derivatives and Ordinary Differential Equations

24.7.1 Difference

Function **difference**() with the prototype of

```
array double difference(array double a[&])[:];
```

calculates differences between the adjacent elements of an array. The one-dimensional input array is of real type.

For example,

```
> array double a[6] = {1, 2, 10, 4, 5, 6}
> difference(a)
1.0000 8.0000 -6.0000 1.0000 1.0000
```

24.7.2 Derivatives

Function **derivative**() with the prototype of

```
double derivative(double (*func)(double), double x, ...
        /* [double &err], [double h]*/);
```

calculates numerically the derivative of a function pointed to by *func* at a given point *x*. The returned value is the derivative of the function at point *x*. The optional argument *err* contains the estimate of the error in the calculation of the derivative. It helps the user to estimate the result. If the optional argument *h* is given, calculation of the derivative uses the initial step size *h*. The value for *h* does not have to be small, but it should be an increment in *x* over which *func* changes substantially. If the argument for *h* is not passed, the value of $0.02 * x$ or 0.0001 (if $x < 0.0001$) as the initial step size is used by default.

For example, the derivative of function $x\sin(x)$ at point $x = 2.5$ can be calculated by

```
> double func(double x) {return x*sin(x);}
> derivative(func, 2.5);
-1.404387
```

Function **derivatives**() with the prototype of

```
array double derivatives(double (*func)(double), double x[&], ...
        /* [double &err], [double h]*/)[:];
```

calculates derivatives of a function numerically at multiple points. This function returns an **array** of derivative values of function *func* at points specified in array *a*. The values of *x* can be of any real type. The other arguments are the same as in function **derivative**().

For example, the derivatives of function $\sin(x)$ at 36 points evenly spaced in the range of $-\pi \leq x \leq \pi$ can be calculated by the following commands.

```
> array double x[36], y[36];
> double func(double x) {return sin(x);}
> lindata(-3.14, 3.14, x);
> y = derivatives(func, x);
> plotxy(x, y);
```

Note that, unlike in C, the generic functions such as `sin()` cannot be passed to a pointer to function in Ch. Therefore, the generic function `sin()` in the above code is wrapped in function `func()`. The output from the above code is displayed in Figure 24.7.

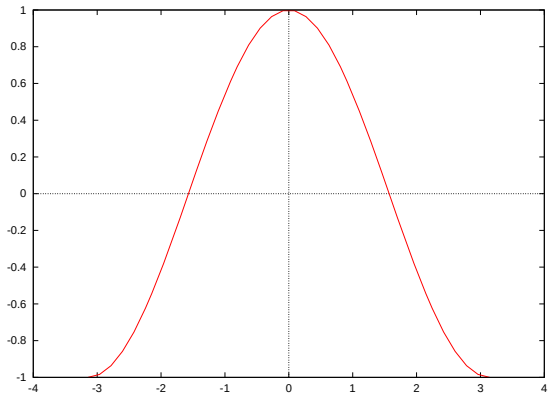


Figure 24.7: The derivatives for sin(x) in the range of $-\pi <= x <= \pi$.

24.8 Solve Ordinary Differential Equations

Function **oderk**() with the prototype of

```
int oderk( void (*func)(double x, double y[], double dydx[], void *param),
       double t0, double tf, double y0[:], void *param,
       double t[:], double *y, ... /* double tol */);
```

numerically solves an ordinary differential equation (ODE)

$$\frac{dy}{dt} = \text{func}(t, y, p)$$

or a system of ordinary differential equations.

$$\frac{d\mathbf{y}}{dt} = \mathbf{func}(t, \mathbf{y}, p)$$

using a Runge-Kutta method. The function can be called using one of the following forms. The argument `param` is used to pass information from the calling function to the ODE function. The argument *func* is specified as a pointer to function for a first-order differential equation. The initial and final values for `t` are specified as arguments *t0* and *tf*. The argument `y0` of array type contains the initial values of differential equations. Its dimension equals the number of dependent variables. If successful, this function returns the number of points calculated in the interval between *t0* and *tf*. Otherwise, it returns -1. Vector *t* contains the values between *t0* and *tf* in which the ODE function is solved and the results are stored in the memory pointed to by the variable `y`. The user shall pass the address of a one-dimensional array for an ordinary differential equation or the address of a two-dimensional array for a system of ordinary differential equations to the argument `y` of the function **oderk**(). If the optional argument *eps* is passed, the algorithm uses the user specified tolerance to decide the iteration stop. Otherwise, the value of 10^{-8} is used by default. In general, the user should guess how many points it would produce and define the size of array *t* and *y* accordingly. Function **oderk**() will automatically pad the leftover space in *t* and *y* with results at *tf*.

For example, the ordinary differential equation

$$\frac{dy}{dt} = \sin(t)$$

with the initial condition $t_0 = -\pi$ and $y_0 = 0$ can be solved with more than 50 points in the interval $t_0 = -\pi$ to $t_f = \pi$ by the following commands.

```
> double t[50], y[50], y00[1]={0}
> void func(double t, double y[], double dydt[], void *param) {dydt[0]=sin(t);}
> oderk(func, -3.14, 3.14, y00, NULL, t, y)
34
> plotxy(t, y)
```

As displayed in the above code, although 50 elements have been allocated for arrays `t` and `y`, only the first 34 elements have the values from the ODE solution. The remaining elements are filled with the values at the end point `tf`. The output from the above code is the same as the one displayed in Figure 24.7.

Function **oderk**() can be used to solve the Van der Pol equation

$$\frac{d^2u}{dt^2} - \mu(1-u^2)\frac{du}{dt} + u = 0$$

in the range of $1 \leq t \leq 30$ with $\mu = 2$, and initial condition $t_0 = 1, u(t_0) = 1$ and $u'(t_0) = 0$. The Van der Pol equation can be reformulated as a set of first-order differential equations with two dependent variables first. Let

$$\begin{aligned} y_0 &= u \\ y_1 &= \frac{du}{dt} \end{aligned}$$

then

$$\begin{aligned} \frac{dy_0}{dt} &= \frac{du}{dt} = y_1 \\ \frac{dy_1}{dt} &= \frac{d^2u}{dt^2} = \mu(1-u^2)\frac{du}{dt} - u = \mu(1-y_0^2)y_1 - y_0 \end{aligned}$$

with the initial condition $t_0 = 1, y_0(t_0) = 1$ and $y_1(t_0) = 0$. Program 24.6 solves the Van der Pol equation with the above initial condition in the range of $1 \leq t \leq 30$. The parameter μ is passed from the main() function to the ODE function through the argument `param`. The output from Program 24.6 is displayed in Figure 24.8.

As another example, a dynamical system with two degrees of freedom might be modeled in a system of differential equations as follows.

$$\begin{aligned} (1.5+q_2)\ddot{q_1} - \dot{q_1}\dot{q_2} - q_1 &= 0 \\ (1+q_1)\ddot{q_2} - \dot{q_1} - q_2 &= 0 \end{aligned}$$

The above equations can be reformulated in the following standard form readily for numerical implementation using function **oderungekutta**() with $y_0 = q_1, y_1 = \dot{q_1}, y_2 = q_2, y_3 = \dot{q_2}$

$$\begin{aligned} \frac{dy_0}{dt} &= y_1 \\ \frac{dy_1}{dt} &= \frac{y_0 + y_1 y_3}{1.5 + y_2} \\ \frac{dy_2}{dt} &= y_3 \\ \frac{dy_3}{dt} &= \frac{y_1 + y_2}{1 + y_0} \end{aligned}$$

```
#include <chplot.h>
#include <numeric.h>

#define NVAR 2
#define POINTS 256
void func(double t, double y[], double dydt[], void *param) {
    double mu;
    mu = *(double*)param;
    dydt[0] = y[1];
    dydt[1]=mu*(1-y[0]*y[0])*y[1] - y[0];
}

int main() {
    double t0=1, tf=30, y0[NVAR] = {1, 0};
    double t[POINTS], y[NVAR][POINTS];
    double mu = 2;

    oderk(func, t0, tf, y0, &mu, t, y);
    plotxy(t, y, "The solution for the van der Pol equation", "t (seconds)", "y1 and y2");
}
```

Program 24.6: A program solves the Van der Pol equation.

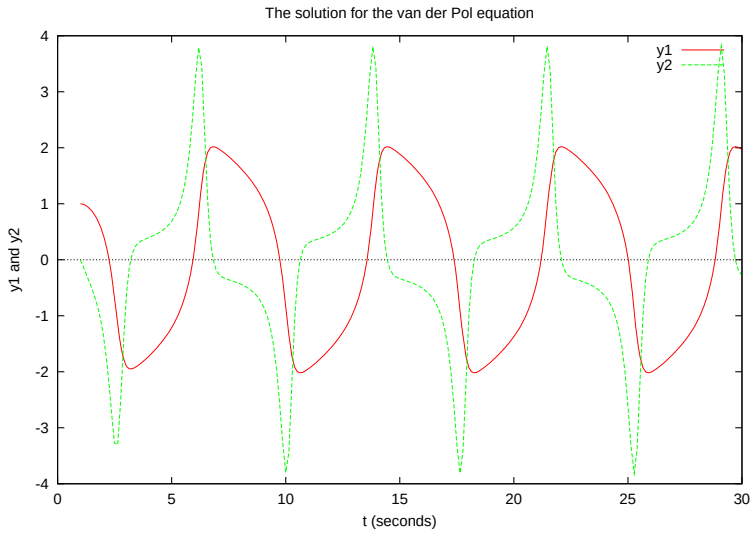


Figure 24.8: The result from oderk().

24.9 Numerical Integration

24.9.1 One-Dimensional Integration

Function **integral1**() with the prototype of

```
double integral1(double (*func)(double x),
                 double x1, double x2, ... /* [double tol] */);
```

performs numerical integration for integral

$$\int_{x_1}^{x_2} f(x)dx$$

The argument *func*, a pointer to function, is the function to be integrated. Arguments *x1* and *x2* are the end-points of the interval. The function returns the value of the integral. If the optional argument *tol* is passed, it is used to decide the iteration stop. Otherwise, the default value of $10 *$ FLT_EPSILON is used.

For example, the integral

$$\int_0^{\frac{\pi}{2}} x^2(x^2-2)\sin(x)dx$$

can be calculated by the following commands.

```
> double func(double x) { return x*x*(x*x-2.0)*sin(x);}
> integral1(func, 0, 3.1416/2);
-0.4792
```

24.9.2 Two-Dimensional Integration

For two-dimensional integration

$$I = \int_{x_1}^{x_2} dx \int_{y_1}^{y_2} dy f(x,y),$$

function **integral2**() with the prototype of

```
double integral2(double (*func)(double x, double y),
                 double x1, double x2, double y1, double y2);
```

can be used. The argument *func*, a pointer to function, is the function to be integrated. Arguments *x1* and *x2* are the end-points in x, and arguments *y1* and *y2* are the end-points in y, respectively. For example, the integral

$$I = \int_0^{\pi} \int_{-\pi}^{\pi} (\sin(x)\cos(y)+1)dxdy$$

can be calculated by the following commands.

```
> double func(double x,double y) { return sin(x)*cos(y)+1;}
> integral2(func,0, 3.14, -3.14, 3.14)
19.7392
```

If the lower limit $y_1(x)$ and upper limit $y_2(x)$ are functions of variable x, function **integration2**() with the prototype of

```
#include <stdio.h>
#include <math.h>
#include <numeric.h>

double func(double x,double y) {
   return x*x+y*y;
}
double y1(double x) {
   return -sqrt(4-x*x);
}
double y2(double x) {
   return sqrt(4-x*x);
}

int main() {
  double x1=-2, x2=2;
  double s;
  s=integration2(func,x1,x2,y1,y2);
  printf("integration2() = %.3f\n", s);
}
```

Program 24.7: A program using **integration2**().

```
double integration2(double (*func)(double x, double y), double x1,
       double x2, double (*y1)(double x), double (*y2)(double x));
```

can be used. Unlike in function **integral2**(), arguments `y1` and `y2` in function **integration2**() are pointer to functions. For example, integration of r^2 over a circular area with a radius of $r = 2$

$$I = \int_{-r}^{r} \int_{-\sqrt{r^2-x^2}}^{\sqrt{r^2-x^2}} (x^2 + y^2) dxdy$$

can be performed by Program 24.7. The output of `integration2() = 25.156` will be printed out from execution of Program 24.7.

24.9.3 Three-Dimensional Integration

Similarly, a three-dimensional integration

$$I = \int_{x_1}^{x_2} dx \int_{y_1}^{y_2} dy \int_{z_1}^{z_2} dz f(x, y, z),$$

can be performed using function **integral3**() with the prototype of

```
double integral3(double (*func)(double x, double y, double y),
   double x1, double x2, double y1, double y2, double z1, double z2);
```

The argument *func*, a pointer to function, is the function to be integrated. Arguments *x1* and *x2*, *y1* and *y2*, *z1* and *z2* are the end-points in x, y, z, respectively. For example, the integral

$$I = \int_{0}^{\pi} \int_{-\pi}^{\pi} \int_{0}^{\pi} (\sin(x)\cos(y)\sin(z) + 1) dxdydz$$

can be calculated by the following commands.

```
#include <stdio.h>
#include <math.h>
#include <numeric.h>

double func(double x,double y,double z) {
   return x*x+y*y+z*z;
}
double y1(double x) {
   return -sqrt(4-x*x);
}
double y2(double x) {
   return sqrt(4-x*x);
}
double z1(double x,double y) {
   return -sqrt(4-x*x-y*y);
}
double z2(double x,double y) {
   return sqrt(4-x*x-y*y);
}

int main() {
   double x1=-2, x2 =2, s;
   s=integration3(func,x1,x2, y1,y2, z1,z2);
   printf("integration3() = %.3f\n", s);
}
```

Program 24.8: A program using **integration3**().

```
> double func(double x,double y, double z) \
      { return sin(x)*cos(y)*sin(z)+1;}
> integral3(func,0, 3.14, -3.14, 3.14, 0, 3.14)
62.0126
```

If the lower limit $y_1(x)$ and upper limit $y_2(x)$ are functions of variable x, or the lower limit $z_1(x, y)$ and upper limit $z_2(x, y)$ are functions of variables x and y, function **integration3**() with the prototype of

```
double integration3(double (*func)(double x, double y), double x1,
   double x2, double (*y1)(double x), double (*y2)(double x));
   double (*z1)(double x, double y), double (*z2)(double x, double z));
```

can be used. Unlike in function **integral3**(), arguments `y1`, `y2`, `z1`, and `z2` in function **integration3**() are of type of pointer to function. For example, integration of r^2 over a spherical volume with a radius of $r = 2$

$$I = \int_{-r}^{r} \int_{-\sqrt{r^2-x^2}}^{\sqrt{r^2-x^2}} \int_{-\sqrt{r^2-x^2-y^2}}^{\sqrt{r^2-x^2-y^2}} (x^2 + y^2 + z^2) dx dy dz$$

can be performed by Program 24.8. The output of `integration3() = 80.487` will be printed out by executing of Program 24.8.

24.10 Matrix Functions

Elementary functions applicable to only n-by-n square matrices are described in this section.

24.10.1 Characteristics of Matrices

Calculation of the coefficients for the characteristic polynomial of a matrix using function **charpolycoef**() has been presented in section 24.5.6. Functions for obtaining other features of matrices are presented in this section. Some functions in section 24.2 for various properties of multi-dimensional arrays are also applicable to n-by-n square matrices.

Determinant

Function **determinant**() with the prototype of

```
double determinant(array double complex a[&][&]);
```

returns the determinant of matrix *a*. If matrix *a* is not a square matrix, the determinant is NaN. For a matrix of complex numbers, function **cdeterminant**() with the prototype of

```
double complex cdeterminant(array double complex a[&][&]);
```

shall be used to calculate its determinant.

For example,

```
> array double a[2][2] = {2, 4, 3, 7}
> determinant(a)
2.0000
> cdeterminant(a)
complex(2.0000,-0.0000)
```

Condition Number

The condition number of a matrix measures the sensitivity of the solution of a system of linear equations to errors in the data. It gives an indication of the accuracy of the results from matrix inversion and numerical solution of the linear system of equations solution. A condition number near 1 indicates a well-conditioned matrix. If a matrix is ill-conditioned, the condition number approaches infinity. Function **condnum**() with the prototype of

```
double condnum(array double complex a[&][&]);
```

returns the condition number of matrix *a*.

Function **rcondnum**() with the prototype of

```
double rcondnum(array double complex a[&][&]);
```

calculates an estimate for the reciprocal of the condition of matrix *a* in 1-norm. Compared to **condnum**(), function **rcondnum**() is a more efficient, but less reliable, method of estimating the condition of a matrix.

For example,

```
> array double a[2][2] = {2, 4, 3, 7}
> array double b[2][2] = {2, 4, 2.001, 4.001}
> condnum(a)
38.9743
> condnum(b)
20006.0010
> rcondnum(a)
```

```
0.0182
> 1/condnum(a)
0.0257
> rcondnum(b)
0.0000
> 1/condnum(b)
0.0001
```

Trace

The trace is defined as the sum of diagonal elements of a matrix. Function **trace**() with the prototype of

```
double trace(array double a[&][&]);
```

returns the trace of matrix *a*. For matrices of complex numbers, their trace shall be calculated by function **ctrace**() with the prototype of

```
double complex ctrace(array double complex a[&][&]);
```

For example,

```
> array double a[2][2] = {2, 4, 3, 7}
> trace(a)
9.0000
> ctrace(a)
complex(9.0000,0.0000)
> array double b[2][3] = {1, 1, 1, 1, 1, 1}
> trace(b)
2.0000
```

Diagonal

Function **diagonal**() with the prototype of

```
double diagonal(array double a[&][&], ... /* [int k] */)[:];
```

produces a column vector formed from the elements of the *k*th diagonal of matrix *a*. If the optional argument *k* is missing, the function returns the diagonal of the matrix. For matrices of complex numbers, their diagonal shall be calculated by function **ctrace**() with the prototype of

```
double complex cdiagonal(array double complex a[&][&],
                         ... /* [int k] */)[:];
```

For example,

```
> array double a[4][3] = {1, 2, 3, \
                          4, 5, 6, \
                          7, 8, 9, \
                          4, 4, 4}
> diagonal(a)
1.0000 5.0000 9.0000
> diagonal(a, -1)
4.0000 8.0000 4.0000
> cdiagonal(a, 1)
complex(2.0000,0.0000) complex(6.0000,0.0000)
```

Rank

Rank is defined as the number of linearly independent rows or columns of a matrix. Function **rank**() with the prototype of

```
int rank(array double complex a[&][&]);
```

returns the rank of matrix *a*. The algorithm used inside the function is based on the singular value decomposition. It is the number of non-zero singular values with tolerance $tol = \max(m, n) \times \max(S) \times$ `DBL_EPSILON`.

For example,

```
> array double a[2][2] = {2, 4, 3, 7}
> array double b[2][3] = {1, 2, 3, 2, 4, 6}
> rank(a)
2
> rank(b)
1
```

24.10.2 Manipulation of Matrices

In addition to generic function **transpose**() for transpose of a matrix. Functions **fliplr**(), **flipud**(), **rot90**(), can be used to manipulate matrices.

Flip Matrices in Left/Right Direction

Function **fliplr**() with the prototype of

```
int fliplr(array double complex y[&][&], array double complex x[&][&]);
```

flips matrix *x* in left/right direction with rows being preserved and columns flipped. Matrix *y*, with the same data type and size as *x*, contains the flipped result of input matrix *x*.

For example,

```
> array double y[2][4], x[2][4]={1, 2, 3, 4, \
                                 5, 6, 7, 8}
> fliplr(y, x)
> y
4.0000 3.0000 2.0000 1.0000
8.0000 7.0000 6.0000 5.0000
```

Flip Matrices in Up/Down Direction

Similar to function **fliplr**(), function **flipud**() with the prototype of

```
int flipud(array double complex y[&][&], array double complex x[&][&]);
```

flips matrix *x* in up/down direction with columns being preserved and rows flipped. Matrix *y*, with the same data type and size as *x*, contains the flipped result of input matrix *x*.

For example,

```
> array double y[4][2], x[4][2]={1, 2, \
                                 3, 4, \
                                 5, 6, \
                                 7, 8}
> flipud(y, x)
> y
7.0000 8.0000
5.0000 6.0000
3.0000 4.0000
1.0000 2.0000
```

Rotate Matrices

Function **rot90**() with the prototype of

```
int rot90(array double complex y[&][&], array double complex x[&][&],
          ... /* [int  k] */);
```

rotates matrix *x* $k * 90$ degrees. For a positive *k*, the matrix is rotated counter clockwise. For a negative *k*, the matrix is rotated in clockwise. Array argument y is a two-dimensional matrix of the same data type and size of matrix *x*. Matrix *y* contains the rotation of input matrix *x*.

For example,

```
> array double y[4][2], x[2][4]={1, 2, 3, 4, \
                                 5, 6, 7, 8}
> rot90(y, x)
> y
4.0000 8.0000
3.0000 7.0000
2.0000 6.0000
1.0000 5.0000
> rot90(x, x, 2)
> x
8.0000 7.0000 6.0000 5.0000
4.0000 3.0000 2.0000 1.0000
```

24.10.3 Special Matrices

Programming with special matrices in Ch are presented in this section.

Identity Matrix

Function **identitymatrix**() with the prototype of

```
array double identitymatrix(int n)[:][:];
```

returns an $n \times n$ indentity matrix. For example,

```
> identitymatrix(3)
1.0000 0.0000 0.0000
0.0000 1.0000 0.0000
0.0000 0.0000 1.0000
```

Diagonal Matrix

Function **diagonalmatrix**() with the prototype of

```
array double diagonalmatrix(array double v[&], ... /*[int k]*/)[:][:];
```

returns a square matrix of order $n + abs(k)$, with the elements of *v* on the *k*th diagonal. $k = 0$ represents the main diagonal, $k > 0$ above the main diagonal, and $k < 0$ below the main diagonal. By default, *k* is 0. Function **diagonalmatrix**() creates a main diagonal matrix. For a diagonal matrix of complex type, function **cdiagonalmatrix**() with the prototype of

```
array double cdiagonalmatrix(array double complex v[&], ...
                             /* [int k] */)[:][:];
```

shall be used.

For example,

```
> array double v[2] = {1, 2}
> diagonalmatrix(v)
1.0000 0.0000
0.0000 2.0000
> diagonalmatrix(v, 1)
0.0000 1.0000 0.0000
0.0000 0.0000 2.0000
0.0000 0.0000 0.0000
> diagonalmatrix(v, -1)
0.0000 0.0000 0.0000
1.0000 0.0000 0.0000
0.0000 2.0000 0.0000
```

Triangular Matrix

Function **triangularmatrix**() with the prototype of

```
array double triangularmatrix(string_t pos, array double a[&][&],
                              ... /* [int k] */)[:][:];
```

returns a triangular matrix of matrix *a*. The dimension of the returned matrix is the same as the input matrix *a*. For *pos* of `"upper"`, the function returns the upper triangular part of matrix *a*, on and above the *k*th diagonal of the matrix. Optional argument *k* indicates the offset of the triangular matrix to the upper *k*th diagonal of the matrix. For *pos* of `"lower"` the function returns the lower part of matrix *a*, on and below the *k*th diagonal of the matrix. Optional argument *k* indicates the offset of the triangular matrix to the lower *k*th diagonal of the matrix. By default, *k* is 0. Function **diagonalmatrix**() creates a main diagonal matrix. For a triangular matrix of complex type, function **ctriangularmatrix**() with the prototype of

```
array double complex ctriangularmatrix(string_t pos,
      array double complex a[&][&], ... /* [int k] */)[:][:];
```

shall be used.

For example,

```
> array double a[4][3] = {1,2,3, \
                          4,5,6, \
                          7,8,9, \
                          6,3,5}
> triangularmatrix("upper", a)
1.0000 2.0000 3.0000
0.0000 5.0000 6.0000
0.0000 0.0000 9.0000
0.0000 0.0000 0.0000
> triangularmatrix("upper", a, 1)
0.0000 2.0000 3.0000
0.0000 0.0000 6.0000
0.0000 0.0000 0.0000
0.0000 0.0000 0.0000
> triangularmatrix("upper", a, -1)
1.0000 2.0000 3.0000
4.0000 5.0000 6.0000
0.0000 8.0000 9.0000
0.0000 0.0000 5.0000
> triangularmatrix("lower", a)
1.0000 0.0000 0.0000
4.0000 5.0000 0.0000
7.0000 8.0000 9.0000
6.0000 3.0000 5.0000
> triangularmatrix("lower", a, 1)
1.0000 2.0000 0.0000
4.0000 5.0000 6.0000
7.0000 8.0000 9.0000
6.0000 3.0000 5.0000
> triangularmatrix("lower", a, -1)
0.0000 0.0000 0.0000
4.0000 0.0000 0.0000
7.0000 8.0000 0.0000
6.0000 3.0000 5.0000
```

Companion Matrix

Function **companionmatrix**() with the prototype of

```
array double companionmatrix(array double v[&])[:][:];
```

returns the companion matrix from array *v* of the coefficients of a polynomial. For array *v* of size n, the first row of the companion matrix is -**v**[1:n]/v[0]. The eigenvalues of companion matrix are the roots of the polynomial. For a triangular matrix of complex type, function **ccompanionmatrix**() with the prototype of

```
array double complex ccompanionmatrix(array double complex v[&])[:][:];
```

shall be used.

For example, the companion matrix of the polynomial $2x^3 + 3x^2 + 4x + 5$ can be obtained by the following commands.

```
> #define N 4
> array double v[N] = {2,3,4,5}
> array double a[N-1][N-1]
> a = companionmatrix(v)
-1.5000 -2.0000 -2.5000
1.0000 0.0000 0.0000
0.0000 1.0000 0.0000
```

Householder Matrix

Given a vector **x**, the Householder matrix **H** is defined as

$$\mathbf{H} = \mathbf{I} - \beta \mathbf{v}\mathbf{v}^{\mathbf{T}}$$

where **I** is an identity matrix and vector **v** is the same size as vector **x**. A Householder matrix **H** satisfies the equation

$$\mathbf{H}\mathbf{x} = -\text{sign}(\mathbf{x}[0]) * \text{norm}(\mathbf{x}) * \mathbf{E}$$

where vector $\mathbf{E} = [1, 0, 0, \cdots, 0]$ is of the same size as vector **x**. If **x** is a complex vector, the Householder matrix **H** is defined as

$$\mathbf{H} = \mathbf{I} - \beta \mathbf{v}\mathbf{v}^{\mathbf{H}}$$

and sign(**x**[0]) is defined as

$$\text{sign}(\mathbf{x}[0]) = \frac{\mathbf{x}[0]}{\text{abs}(\mathbf{x}[0])}$$

Function **householdermatrix**() with the prototype of

```
int householdermatrix(array double complex x[&],
                      array double complex v[&], ... /* [double *beta] */);
```

passes a vector *x* as an input argument and gets the vector *v* and optional output value *beta* for a Householder matrix. For example, a Householdermatrix of a vector of real type can be calculated by the following commands.

```
> array double x[5] = {-0.3, 54, 25.3, 25.46, 83.47}
> array double e[5]={1,0,0,0,0}
> array double v[5], h[5][5]
> double beta
> householdermatrix(x,v,&beta)
> h = identitymatrix(5) - beta*v*transpose(v)
> v
-105.9959 54.0000 25.3000 25.4600 83.4700
> beta
0.0001
> h*x+sign(x[0])*norm(x,"2")*e
0.0000 0.0000 0.0000 -0.0000 -0.0000
```

A Householder matrix from a vector of complex type can be calculated by the following commands.

Table 24.4: Special matrices.

Cauchy	ChebyshevVandermonde	Chow	Circul	Celement
Dramadah	DenavitHartenberg	DenavitHartenberg2	Fiedler	Frank
Gear	Hadamard	Hankel	Hilbert	InverseHilbert
Magic	Pascal	Rosser	Toeplitz	Vandermonde
Wilkinson				

```
> array double complex x[3] = {complex(-0.3, 0.5), 54, 25.3}
> array double e[3]={1,0,0}
> array double complex v[3], h[3][3]
> double beta
> householdermatrix(x,v,&beta)
> h = identitymatrix(3) - beta*v*conj(transpose(v))
> printf("%.3f", v)
complex(-30.982,51.637) complex(54.000,0.000) complex(25.300,0.000)
> beta
0.0003
> v = h*x+x[0]/abs(x[0])*norm(x,"2")*e
> printf("%.3f", v)
complex(-30.682,51.137) complex(0.000,0.000) complex(0.000,0.000)
```

Special Matrix

Function **specialmatrix**() with the prototype of

```
array double specialmatrix(string_t name, ...
            /* [type1 arg1, type2 arg2, ...] */)[:][:];
```

returns a special matrix. The argument *name* can be one of special matrices listed in Table 24.4. Optional arguments *arg1*, *arg2*, etc. may be required for a particular special matrix. The number of arguments and their data types are different for different special matrices.

As an example, a Hilbert matrix, $\mathbf{H}$, has elements $\mathbf{H}[i][j] = 1/(i+j+1)$. It is a famous example of an ill-conditioned matrix. A Hilbert matrix can be generated by a function call of `specialmatrix("Hilbert", n)`, where argument *n* specifies the degree of the matrix. That is, the size of Hilbert matrix is $n \times n$. A 4-by-4 Hilbert matrix can be created by the following command.

```
> specialmatrix("Hilbert", 4)
1.0000 0.5000 0.3333 0.2500
0.5000 0.3333 0.2500 0.2000
0.3333 0.2500 0.2000 0.1667
0.2500 0.2000 0.1667 0.1429
```

Detailed description of each special matrix can be found the chapter about numerical analysis in *The Ch Language Environment — Reference Guide*.

24.10.4 Matrix Analysis

Unlike functions **fevalarray**() and **cfevalarray**which applies a function to each element of an array, Ch functions in this section can be used to evaluate mathematical functions with variables of matrix type.

Function **funm**() with the prototype of

```
int funm(array double y[&][&], double (*func)(double),
         array double x[&][&]);
```

evaluates the function in matrix version, specified by argument *func*. In this function, the input square matrix *x* shall be type double and the specified function shall be prototyped as

```
double func(double);
```

The output square matrix *y* with the same size of matrix *x* can be real or complex type as required. Function **funm**() returns 0 on success and -1 on failure. For a matrix of complex type, function **cfunm**() with the prototype of

```
int cfunm(array double complex y[&][&],
          double complex (*func)(double complex),
          array double complex x[&][&]);
```

shall be used.

For example, the polynomial of

$$\mathbf{Y} = \mathbf{X}^3 + 2\mathbf{X}^2 + 3\mathbf{X} + 4$$

with matrix **X** of

$$\mathbf{X} = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

can be calculated by the following commands.

```
> array double p[4] = {1,2,3,4}
> array double x[2][2] = {5,6,7,8}, y[2][2]
> double func(double x) { return polyeval(p, x);}
> funm(y, func, x)
> y
1034.0000 1200.0000
1400.0000 1634.0000
```

Matrix analysis functions **polyevalm**(), **sqrtm**(), **expm**(), and **logm**() have been implemented in Ch to handle polynomial, square root, exponential, and natural logarithmic functions with matrix variables, respectively. These functions have the prototypes of

```
int polyevalm(array double complex y[&][&],
              array double complex p[&],
              array double complex x[&][&]);
int sqrtm(array double complex y[&][&], array double complex x[&][&]);
int expm(array double complex y[&][&], array double complex x[&][&]);
int logm(array double complex y[&][&], array double complex x[&][&]);
```

For example, the previous polynomial of a matrix can be calculated by the function **polyevalm**(). Applications of complementary functions **expm**() versus **logm**() and **sqrtm**() versus X^2 are also demonstrated in the commands below.

```
> array double p[4] = {1,2,3,4}
> array double x[2][2] = {5,6,7,8}, y[2][2]
> polyevalm(y, p, x)
> y
1034.0000 1200.0000
1400.0000 1634.0000
> expm(y, x)
> y
199464.8244 232291.9543
271007.2800 315610.8016
> logm(y, y)
> y
5.0000 6.0000
7.0000 8.0000
> array double complex z[2][2]
> sqrtm(z, x)
> z
complex(1.4044,0.2389) complex(1.6355,-0.1759)
complex(1.9081,-0.2052) complex(2.2222,0.1510)
> array double p2[3] = {1, 0, 0}
> polyevalm(y, p2, z)
> y
5.0000 6.0000
7.0000 8.0000
```

24.11 Matrix Decomposition

24.11.1 LU Decomposition

ludecomp() an LU decomposition of a square matrix **A** is calculated using partial pivoting with row interchanges. The factorization has the form

$$\mathbf{A} = \mathbf{PLU},$$

where **P** is a permutation matrix, **L** a lower triangular matrix with unit diagonal elements, and **U** an upper triangular matrix.

Function **ludecomp**() with the prototype of

```
int ludecomp(array double complex a[&][&],array double complex l[&][&],
        array double complex u[&][&], ... /*[array int p[&][&]] */);
```

decomposes a general n-by-n matrix according to the above LU decomposition formula.

For example,

```
> array double l[3][3], u[3][3], a[3][3] = {2, 1, -2, \
                                             4, -1, 2, \
                                             2, -1, 2}
> array int p[3][3]
> ludecomp(a, l, u)
> l
```

```
0.5000 1.0000 0.0000
1.0000 0.0000 0.0000
0.5000 -0.3333 1.0000
> u
4.0000 -1.0000 2.0000
0.0000 1.5000 -3.0000
0.0000 0.0000 0.0000
> l*u
2.0000 1.0000 -2.0000
4.0000 -1.0000 2.0000
2.0000 -1.0000 2.0000
> ludecomp(a, l, u, p)
> l
1.0000 0.0000 0.0000
0.5000 1.0000 0.0000
0.5000 -0.3333 1.0000
> u
4.0000 -1.0000 2.0000
0.0000 1.5000 -3.0000
0.0000 0.0000 0.0000
> p
0 1 0
1 0 0
0 0 1
> p*l*u
2.0000 1.0000 -2.0000
4.0000 -1.0000 2.0000
2.0000 -1.0000 2.0000
```

24.11.2 Singular Value Decomposition

The singular value decomposition is formulated as

$$\mathbf{A} = \mathbf{USV^T}$$

Where S is an m-by-n matrix with zero for each element except for its diagonal elements of size min(m,n), U is an m-by-m orthogonal matrix, and V is an n-by-n orthogonal matrix. The diagonal elements of S are the singular value of matrix **A**; they are real, non-negative, and are in descending order. The first min(m,n) column of **U** and **V** are the left and right singular vectors of matrix **A**.

Function **svd**() with the prototype of

```
int svd(array double complex a[&][&], array double s[&],
        array double complex u[&][&], array double complex vt[&][&]);
```

computes the singular value of a real or complex m-by-n matrix, and the left and right singular vectors.

For example, singular decomposition of an n-by-n matrix can be performed by the following commands.

```
> array double a[2][2] = {1, 2, \
                          3, 4}
```

```
> array double s[2], u[2][2], v[2][2]
> svd(a, s, u, v)
> s
5.4650 0.3660
> u
-0.4046 -0.9145
-0.9145 0.4046
> v
-0.5760 0.8174
-0.8174 -0.5760
> u*diagonalmatrix(s)*transpose(v)
1.0000 2.0000
3.0000 4.0000
```

Singular decomposition of an m-by-n matrix can be performed by the following commands.

```
> array double a[2][3] = {7, 8, 1, \
                          3, 6, 4}
> int m=2, n=3, mn=min(m,n), i
> array double s[mn], sm[m][n], u[m][m], v[n][n]
> svd(a, s, u, v)
> for(i=0; i<mn; i++) sm[i][i] = s[i]
> s
12.8515 3.1367
> u
-0.8189 -0.5739
-0.5739 0.8189
> v
-0.5800 -0.4976 0.6450
-0.7777 0.1027 -0.6202
-0.2424 0.8613 0.4465
> u*sm*transpose(v)
7.0000 8.0000 1.0000
3.0000 6.0000 4.0000
```

24.11.3 Cholesky Decomposition

Cholesky decomposition factors a symmetric, positive, definite matrix into two matrices. For a symmetric positive finite matrix **A** of real type, Cholesky decomposition can produce matrix **L** so that

$$\mathbf{A} = \mathbf{L^T L}$$

for upper triangle factorization or

$$\mathbf{A} = \mathbf{L L^T}$$

for lower triangle factorization, where $\mathbf{L}^T$ is the transpose of matrix **L**. For a symmetric positive finite matrix **A** of complex type, Hermitian $\mathbf{L}^H$ of matrix **L** shall be used instead of $\mathbf{L}^T$.

Function **choldecomp**() with the prototype of

```
int choldecomp(array double complex a[&][&],
               array double complex l[&][&], ... /* [char mode] */);
```

performs Cholesky decomposition. Array argument **l** contains the upper or lower triangle of a symmetric, positive, definite matrix **a**. The **string_t** *mode* specifies the calculation of upper or lower triangle matrix. The character of 'L' or 'l' is for lower triangle factorization, otherwise, the upper triangle matrix is calculated. By default, the upper triangle matrix is calculated. Function **choldecomp**() returns 0 on success, negative value on failure, and positive value i indicates that the leading minor of order i is not positive definite and the factorization could not be completed.

For example, Cholesky decomposition of a symmetric, positive, definite matrix of real type into an upper triangle matrix can be calculated by the following commands.

```
> array double l[2][2], a[2][2] = {1, 1, \
                                   1, 2}
> choldecomp(a, l);
> l
1.0000 1.0000
0.0000 1.0000
> transpose(l)*l
1.0000 1.0000
1.0000 2.0000
```

Cholesky decomposition of a symmetric positive definite matrix of complex type into the lower triangle matrix can be calculated by the following commands.

```
> array double complex l[2][2],
              a[2][2] = {complex(2,0), complex(0,-1), \
                         complex(0,1), complex(2,0)}
> choldecomp(a, l, 'L');
> l
complex(1.4142,0.0000) complex(0.0000,0.0000)
complex(0.0000,0.7071) complex(1.2247,0.0000)
> l*conj(transpose(l))
complex(2.0000,0.0000) complex(0.0000,-1.0000)
complex(0.0000,1.0000) complex(2.0000,0.0000)
```

24.11.4 QR Decomposition

QR decomposition factors matrix **A** into two matrices so that

$$\mathbf{A} = \mathbf{QR},$$

where **Q** is an orthogonal matrix of real type or unitary matrix of complex type and **R** is an upper triangular matrix. Assume the size of matrix **A** is $m \times n$. If $m \leq n$, there is only one type of matrix **Q** and **R**. The size of matrix **Q** is $m \times m$ and matrix **R** is $m \times n$. If $m > n$, there are two types of matrices for **Q** and **R**. One is full size in which matrix **Q** is $m \times m$ and **R** is $m \times n$. The other is economy size in which matrix **Q** is $m \times n$ and **R** is $n \times n$.

Function **qrdecomp**() with the prototype of

```
int qrdecomp(array double complex a[&][&],
    array double complex q[&][&], array double complex r[&][&]);
```

performs QR decomposition. The unitary matrix *q* and upper triangular matrix *a* are obtained from matrix *a*. For array *a* of size $m \times n$, if $m > n$, function **qrdecomp**() checks the array sizes of arguments *q* and *r*, and automatically selects the corresponding output type.

For example, QR decomposition of a matrix of real type can be calculated by the following commands.

```
> array double q1[3][3], r1[3][2], q2[3][2], r2[2][2],
> array double a[3][2] = {1, 5, \
                          -7, 4, \
                          3, 2}
> qrdecomp(a, q1, r1)
> q1
-0.1302 -0.8351 -0.5345
0.9113 -0.3132 0.2673
-0.3906 -0.4523 0.8018
> r1
-7.6811 2.2132
0.0000 -6.3326
0.0000 0.0000
> transpose(q1)*q1
1.0000 -0.0000 -0.0000
-0.0000 1.0000 -0.0000
-0.0000 -0.0000 1.0000
> q1*r1
1.0000 5.0000
-7.0000 4.0000
3.0000 2.0000
> qrdecomp(a, q2, r2)
> q2
-0.1302 -0.8351
0.9113 -0.3132
-0.3906 -0.4523
> r2
-7.6811 2.2132
0.0000 -6.3326
> transpose(q2)*q2
1.0000 -0.0000
-0.0000 1.0000
> q2*r2
1.0000 5.0000
-7.0000 4.0000
3.0000 2.0000
```

QR decomposition of matrix of complex type can be calculated by the following commands.

```
> array double complex q[2][2], r[2][2]
> array double complex a[2][2] = {complex(1,2), 5, \
                                  3, 3}
> qrdecomp(a, q, r)
> q
```

```
complex(-0.2673,-0.5345) complex(0.7171,-0.3587)
complex(-0.8018,0.0000) complex(-0.0000,0.5976)
> r
complex(-3.7417,0.0000) complex(-3.7417,2.6726)
complex(0.0000,0.0000) complex(3.5857,0.0000)
> conj(transpose(q))*q
complex(1.0000,0.0000) complex(-0.0000,0.0000)
complex(-0.0000,-0.0000) complex(1.0000,0.0000)
> q*r
complex(1.0000,2.0000) complex(5.0000,0.0000)
complex(3.0000,-0.0000) complex(3.0000,0.0000)
```

24.11.5 Hessenberg Decomposition

The Hessenberg matrix **H** for a square matrix **A** of real type is defined as

$$\mathbf{H} = \mathbf{P}^T\mathbf{A}\mathbf{P}$$

where matrix **P** is an orthogonal matrix with $\mathbf{P}^T\mathbf{P}$=**I**. For matrix **A** of complex type, the Hessenberg matrix *H* is defined as

$$\mathbf{H} = \mathbf{P}^\mathbf{H}\mathbf{A}\mathbf{P}$$

where matrix **P** is unitary with $\mathbf{P}^H\mathbf{P}$=**I**. Each element of a Hessenberg matrix below the first subdiagonal is zero. If the matrix is symmetric or Hermitian, the form is tridiagonal. This matrix has the same eigenvalues as the original one, but less computation is needed to calculate them.

Function **hessdecomp**() with the prototype of

```
int hessdecomp(array double complex a[&][&],
               array double complex h[&][&], ...
               /* [array double complex p[&][&]] */);
```

decomposes the matrix *a* into the Hessenberg matrix *h* and orthogonal or unitary matrix *p*. The square matrix *a* could be any supported arithmetic data type. The output matrix *h* is the same dimension and data type as the input *a*. If the input *a* is of real type, the optional output *p* shall only be of type double. If the input *a* is complex type, *p* shall be **complex** or **double complex** type.

For example, a Hessenberg matrix can be calculated by the following commands.

```
> array double a[3][3] = {0.8, 0.2, 0.1,\
                          0.1, 0.7, 0.3,\
                          0.1, 0.1, 0.6}
> array double h[3][3], p[3][3]
> hessdecomp(a, h, p)
> h
1.0000 0.0000 0.0000
0.0000 -0.7071 -0.7071
0.0000 -0.7071 0.7071
> p
0.8000 -0.2121 -0.0707
-0.1414 0.8500 -0.0500
```

```
0.0000 0.1500 0.4500
> transpose(p)*a*p
1.0000 0.0000 0.0000
0.0000 -0.7071 -0.7071
0.0000 -0.7071 0.7071
```

24.11.6 Schur Decomposition

The Schur matrix **T** for a square matrix **A** of real type is defined as

$$\mathbf{A} = \mathbf{QTQ}^T$$

where matrix **Q** is an orthogonal matrix with $\mathbf{Q}^T\mathbf{Q}$=**I**. For matrix **A** of complex type, the Schur matrix *T* is defined as

$$\mathbf{A} = \mathbf{QTQ}^H$$

where matrix **Q** is unitary with $\mathbf{Q}^H\mathbf{Q}$=**I**.

Function **schurdecomp**() with the prototype of

```
int schurdecomp(array double complex a[&][&],
    array double complex q[&][&], array double complex t[&][&])
```

decomposes the matrix *a* into the Schur matrix *t* and orthogonal or unitary matrix *q*. The square matrix *a* could be any supported arithmetic data type. The output matrices *t* and *q* shall be the same dimension and data type as the input *a*.

For example, a Schur matrix can be calculated by the following commands.

```
> array double t[2][2], q[2][2], a[2][2] = {8, -3, \
                                            -5, 9}
> schurdecomp(a, q, t)
> t
4.5949 2.0000
0.0000 12.4051
> q
0.6611 -0.7503
0.7503 0.6611
> transpose(q)*q
1.0000 0.0000
0.0000 1.0000
> q*t* transpose(q)
8.0000 -3.0000
-5.0000 9.0000
```

24.12 Linear Equations

24.12.1 Linear System of Equations

A linear system of equations

$$\mathbf{Ax} = \mathbf{b}$$

can be solved by various functions in Ch. If matrix **A** is an n-by-n square matrix of double type, it can be solved by function **linsolve**(). Function **linsolve**() uses LU decomposition with partial pivoting and row interchanges to factor matrix **A** as $\mathbf{A} = \mathbf{PLU}$, where P is a permulation matrix, L is unit lower triangular, and U is upper triangular. The factored form of **A** is then used to solve the system of equations for **x**. Function **linsolve**() with the prototype of

```
int linsolve(array double x[:], array double a[:][:], array double b[:])
```

take three arguments *x, a* and *b*, corresponding to **x, A** and **b** in the linear system of equation $\mathbf{Ax} = \mathbf{b}$, respectively. They shall be arrays of double type. The function **linsolve**() returns 0 if the equation can be solved successfully, otherwise it returns -1. Function **clinsolve**() can be used to solve a system of complex linear equations.

For example,

```
> array double a[3][3] = {3, 0, 6,\
                          0, 2, 1,\
                          1, 0, 1}
> array double x[3], b[3] = {2, 13, 25}
> linsolve(x, a, b)
> x
49.3333 18.6667 -24.3333
> a*x
2.0000 13.0000 25.0000
```

24.12.2 Over-Determined or Under-Determined Linear System of Equations

For a linear system of equations

$$\mathbf{Ax} = \mathbf{b},$$

if A with dimension m-by-n is not a square matrix, it can be solved by the linear least-squares method, which minimizes the squared error of $(\mathbf{A} * \mathbf{x} - \mathbf{b})^T(\mathbf{A} * \mathbf{x} - \mathbf{b})$. Function **llsqsolve**() with the prototype of

```
int llsqsolve(array double complex x[&],
    array double complex a[&][&], array double complex b[&]);
```

takes three arguments *x, a* and *b*, corresponding to **x, A** and **b** in the linear system of equation $\mathbf{Ax} = \mathbf{b}$, respectively. They can be arrays of complex numbers. The number of elements for *x* shall be the same as the number of columns in matrix *a*. The number of elements for *b* shall be the same as the number of rows in matrix *a*. The function **llsqsolve**() returns 0 if the equation can be solved successfully, otherwise it returns -1.

For example, the solution of a linear system of equations based on the least-squares method can be calculated by the following commands.

```
> array double a[2][3] = {3, 5, 6,\
                          7, 2, 1}
> array double x[3], b[2] = {1, 2}
> llsqsolve(x, a, b)
> x
0.2742 0.0440 -0.0070
> a*x
1.0000 2.0000
```

```
> array double a2[3][2] = {3, 5, \
                           6, 7, \
                           2, 1}
> array double x2[2], b2[3] = {1, 2, 3}
> llsqsolve(x2, a2, b2)
> x2
1.4278 -0.8299
> a2*x2
0.1340 2.7577 2.0258
```

Function **llsqnonnegsolve**() with the prototype of

```
int llsqnonnegsolve(array double x[&], array double a[&][&],
       array double b[&], ... /* [double tol, array double w[&]] */);
```

can be used to solved a linear system of equations $\mathbf{Ax} = \mathbf{b}$ using the least-squares method, subject to the constraint that the solution vector *x* has non-negative elements. That is, $x[i] \geq 0$ for $i = 0, 1, \cdots, n-1$. Function **llsqnonnegsolve**() takes three arguments *x*, *a* and *b*, corresponding to **x, A** and **b** in the linear system of equation $\mathbf{Ax} = \mathbf{b}$, respectively. The optional argument *tol* specifies the tolerance of the solution. If the user does not specify this argument or specify zero, $tol = 10 * max(m, n) * norm(u, "1") *$ FLT_EPSILON is used by default. FLT_EPSILON is defined in header file **float.h**. The optional array argument *w* with n elements contain a vector where $w[i] < 0$ when $x[i] = 0$ and $w[i] \cong 0$ when $x[i] > 0$.

For example, the non-negative solution of a linear system of equations based on the least-squares method can be calculated by the following commands.

```
> array double a[2][3] = {3, 5, 6,\
                          7, 2, 1}
> array double x[3], w[3], b[2] = {1, 2}
> llsqnonnegsolve(x, a, b)
> x
0.2821 0.0000 0.0257
> a*x
1.0000 2.0000
> array double a2[3][2] = {3, 5, \
                           6, 7, \
                           2, 1}
> array double x2[2], b2[3] = {1, 2, 3}
> llsqnonnegsolve(x2, a2, b2)
> x2
0.4286 0.0000
> a2*x2
1.2857 2.5714 0.8571
```

Function **llsqcovsolve**() with the prototype of

```
int llsqcovsolve(array double x[&], array double a[&][&],
       array double b[&], array double v[&][&], ...
       /* [array double p[&]] */);
```

can be used to solved a linear system of equations $\mathbf{Ax} = \mathbf{b}$, with a normally distributed error (with zero mean) and covariance *v* using the least-squares method. This is an overdetermined linear least-squares

problem. So the number of rows m must be larger than the number of columns n. If optional output vector *p* is specified, it will pass out the standard errors of *x*.

For example,

```
> array double a[3][2] = {3, 5, \
                          6, 7, \
                          2, 1}
> array double x[2], b[3] = {1, 2, 3}
> array double v=[3][3] = {1, 1, 3, \
                           1, 2, 5, \
                           3, 5, 6}
> llsqcovsolve(x, a, b, v)
> x
0.3402 -0.3521
> a*x
-0.7396 -0.4231 0.3284
```

24.12.3 Inverse and Pseudo Inverse Matrices

The inverse A^{-1} of a square matrix **A** is defined as

$$\mathbf{A^{-1}A} = \mathbf{AA^{-1}} = \mathbf{I}.$$

To obtain its inverse, the matrix **A** shall not be singular. The inverse matrix A^{-1} of matrix **A** of real type can be calculated by function **inverse**() with the prototype of

```
array double inverse(array double a[:][:], ...
                     /* [int *status] */)[:][:];
```

The computation of an inverse matrix is based on the LU decomposition of the original matrix. The optional argument *status* gives the status of calculation. If the calculation is successful,*status* is 0, otherwise *status* is negative value.

The inverse A^{-1} of a square matrix **A** of complex type can be calculated by function **cinverse**() with the prototype of

```
array double complex cinverse(array double complex a[:][:], ...
                              /* [int *status] */)[:][:];
```

For example, the inverse of a matrix can be used to solve the linear system of equation using **x** = A^{-1}**b** by the following commands.

```
> array double a[3][3] = {3, 0, 6,\
                          0, 2, 1,\
                          1, 0, 1}
> array double ai[3][3], b[3] = {2, 13, 25}
> ai=inverse(a)
-0.3333 -0.0000 2.0000
-0.1667 0.5000 0.5000
0.3333 0.0000 -1.0000
> ai*b
49.3333 18.6667 -24.3333
```

The Moore_Penrose pseudo inverse matrix **B** of matrix **A** has to meet the following four conditions:

$$\mathbf{ABA} = \mathbf{A},$$
$$\mathbf{BAB} = \mathbf{B},$$
$$\mathbf{AB} \text{ is Hermitian}$$
$$\mathbf{BA} \text{ is Hermitian}$$

where A can be of singular square matrix or even non-square matrix.

Function **pinverse**() with the prototype of

```
array double pinverse(array double a[:][:])[:][:];
```

calculates the Moore_Penrose pseudo inverse of matrix *a* of real type.

For example,

```
> int M = 2, N = 3
> array double a[2][3] = {7, 8, 1, \
                          3, 6, 4}
> array double p[3][2]
> p = pinverse(a)
0.1280 -0.1040
0.0308 0.0615
-0.1422 0.2357
> a*p*a
7.0000 8.0000 1.0000
3.0000 6.0000 4.0000
> p*a*p
0.1280 -0.1040
0.0308 0.0615
-0.1422 0.2357
```

24.12.4 Linear Spaces

Function **orthonormalbase**() with the prototype of

```
int orthonormalbase(array double complex orth[&][&],
                    array double complex a[&][&]);
```

calculates the the orthonormal bases for matrix **a** of dimension m-by-n. The columns of *orth* are orthonormal, and have the same space as the columns of *a*. The numbers of rows for arrays *orth* and *a* are the same. The number of columns of *orth* is the rank of *a*. Function **orthonormalbase**() returns 0 on success and -1 on failure.

The null space *S* of matrix **A** meets the following conditions.

$$\mathbf{S^T S} = I$$
$$\mathbf{AS} = \mathbf{0}$$

Function **nullspace**() with the prototype of

```
int nullspace(array double complex null[&][&],
              array double complex a[&][&]);
```

calculates the orthonormal bases of the null space of matrix **a** of dimension m-by-n. The columns of *null* are orthonormal. The numbers of rows for arrays *null* and *a* are the same. The number of columns of *null* is the number of columns for array *a* minus the value of the rank of *a*. Function **nullspace**() returns 0 on success and -1 on failure.

For example, the orthonormal bases and orthonormal bases of the null space of a singular matrix with rank 2 can be calculated by the following commands.

```
> #define M 3
> #define N 3
> array double a[M][N] = {1, 2, 3, \
                          4, 5, 6, \
                          7, 8, 9}
> int r
> r = rank(a)
2
> array double orth[M][r]
> orthonormalbase(orth, a)
> orth
-0.2148 0.8872
-0.5206 0.2496
-0.8263 -0.3879
> transpose(orth)*orth
1.0000 -0.0000
-0.0000 1.0000
> array double null[M][N-r]
> nullspace(null, a)
> null
-0.4082
0.8165
-0.4082
> transpose(null)*null
1.0000
```

24.13 Eigenvalues and Eigenvectors

The eigenvalues λ and eigenvectors $\mathbf{V}$ of square matrix $\mathbf{A}$ are defined as

$$\mathbf{AV} = \lambda\mathbf{V}$$

If matrix A is not symmetrical or not all elements are of real type, its eigenvalues λ and eigenvectors $\mathbf{V}$ could be complex numbers.

Function **eigen**() with the prototype of

```
int eigen(... /* double [complex] a[:][:],
                 double [complex] evalues[:],
                 double [complex] evectors[:][:],
                 [char *mode] */);
```

calculates the eigenvalues `evalues` and eigenvectors `evectors` of matrix *a* with dimension n-by-n. The syntaxes for calling this function is as follows.

```
eigen(a, evalues);
eigen(a, evalues, evectors);
eigen(a, evalues, evectors, mode);
```

The computed eigenvalues and eigenvectors are passed as arguments of `evectors` and `evectors` of the function, respectively. Arrays `a` `evalues`, and `evectors` shall be double or double complex type. The computed eigenvectors are normalized so that the norm of each eigenvector equals 1. Argument *mode* is used to indicate if a preliminary balancing step before the calculation is performed or not. Generally, balancing improves the condition of the input matrix, enabling more accurate computation of the eigenvalues and eigenvectors. But, it also may lead to incorrect eigenvectors in some special cases. By default, a preliminary balancing step is taken.

For example, for the matrices below

$$\mathbf{A} = \begin{bmatrix} 0.8 & 0.2 & 0.1 \\ 0.2 & 0.7 & 0.3 \\ 0.1 & 0.3 & 0.6 \end{bmatrix}, \mathbf{B} = \begin{bmatrix} 0.8 & 0.2 & 0.1 \\ 0.1 & 0.7 & 0.3 \\ 0.1 & 0.1 & 0.6 \end{bmatrix}, \mathbf{C} = \begin{bmatrix} 3 & 9 & 23 \\ 2 & 2 & 1 \\ -7 & 1 & -9 \end{bmatrix},$$

matrix **A** is symmetrical with real eigenvalues, **B** is non-symmetrical with real eigenvalues, and **C** is non-symmetrical with complex eigenvalues. The eigenvalues and eigenvectors of these three matrices can be calculated by the following commands.

```
> array double a[3][3] = {0.8,0.2,0.1, 0.2,0.7,0.3, 0.1,0.3,0.6}
> array double b[3][3] = {0.8,0.2,0.1, 0.1,0.7,0.3, 0.1,0.1,0.6}
> array double c[3][3] = {3, 9, 23, 2, 2, 1, -7, 1, -9}
> array double evalues[3], evectors[3][3]
> array double complex zvalues[3], zvectors[3][3]
> eigen(a, evalues)
> evalues
1.1088 0.6526 0.3386
> eigen(b, evalues, evectors)
> evalues
1.0000 0.6000 0.5000
> evectors
-0.7448 -0.7071 0.4082
-0.5793 0.7071 -0.8165
-0.3310 0.0000 0.4082
> eigen(c, evalues)
> evalues
NaN NaN 3.2417
> eigen(c, zvalues, zvectors)
> printf("%.3f", zvalues)
complex(-3.621,10.647) complex(-3.621,-10.647) complex(3.242,0.000)
> printf("%.3f", zvectors)
complex(0.854,0.000) complex(0.854,0.000) complex(0.604,0.000)
complex(-0.024,-0.125) complex(-0.024,0.125) complex(0.744,0.000)
complex(-0.236,0.445) complex(-0.236,-0.445) complex(-0.285,0.000)
```

24.14 Fast Fourier Transforms

A pair of transform and inverse transform are defined below.

$$Y(k) = \sum_{j=0}^{N-1} x(j)\omega_N^{jk}, (k = 0, 1, \cdots, N-1);$$

$$x(j) = \sum_{k=0}^{N-1} Y(k)\omega_N^{-jk}, (j = 0, 1, \cdots, N-1);$$

where $\omega_N = e^{(-2\pi i)/N}$ is an nth root of unity. Functions **fft**() and **ifft**() with the prototype of

```
int fft(array double complex &y, array double complex &x, ...
        /* [int n [int dim [&]]] */);
int ifft(array double complex &x, array double complex &y, ...
         /* [int n [int dim [&]]] */);
```

evaluate the above transform and inverse transform using the fast Fourier transform (FFT) algorithm. It can be used for one-, two-, and three-dimensional FFT. The multi-dimensional (maximum dimension is three) arrays *x* and *y* can be of any supported arithmetic data type and size. Conversion of the data to double complex is performed internally. Array *y* with the same dimension as array *x* contains the result of fast Fourier transform. The optional argument *n* of int type is used to specify the number of points for FFT of one-dimensional data. If the length of the array in the second argument as input data is less than n, the input array is padded with trailing zeros to lengthen n. If the length of the array in the second argument as input data is greater than n, the data in the input array is truncated. For multi-dimensional data, the optional array argument *dim* of int type contains the value for user specified FFT points. The dimensions of input array in the second argument are contained in array *dim*. For example, for a three-dimensional data *x[m][n][l]*, the FFT points of the array are specified by m, n, l. Then the array *dim* is given values of $\text{dim}[0] = m, \text{dim}[1] = n$ and $\text{dim}[2] = l$. Similar to the case of one-dimensional array, if the length of the array in the second argument as input data is less than that specified in array dim, the input array is padded with trailing zeros to the length specified by array *dim*. If the length of the array in the second argument as input data is greater than that specified in array dim, the data in the input array are truncated. If no optional argument is passed, the number of FFT points is obtained from the input array in the second argument. There is no constraint that n, or size of array, needs to be a power of 2 for functions **fft**() and **ifft**(). Functions **fft**() and **ifft**() return 0 on success and -1 on failure. Function **unwrap**() is useful for adjusting the phase angles of complex numbers obtained from the FFT and inverse FFT by changing its absolute jump greater than π to its $2 * \pi$ complement.

For example, the FFT and inverse FFT of the one-dimensional array $\mathbf{x} = (0, 0.25, 0.5, 0.75, 1)$ can be calculated by the following commands.

```
> array double x[5]
> array double complex x1[5], yy1[5], x2[3], y2[3]
> lindata(0, 1, x)
> x
0.0000 0.2500 0.5000 0.7500 1.0000
> fft(yy1,x)
> printf("%.3f",yy1)
complex(2.500,0.000) complex(-0.625,-0.860) complex(-0.625,-0.203)\
```

```
complex(-0.625,0.203) complex(-0.625,0.860)
> ifft(x1,yy1)
> printf("%.3f",x1)
complex(0.000,0.000) complex(0.250,0.000) complex(0.500,0.000) \
complex(0.750,0.000) complex(1.000,0.000)
> fft(y2,x,3)
> printf("%.3f",y2)
complex(0.750,0.000) complex(-0.375,-0.217) complex(-0.375,0.217)
> ifft(x2,y2,3)
> printf("%.3f",x2)
complex(0.000,0.000) complex(0.250,0.000) complex(0.500,0.000)
> fft(yy1,x2,5)
> printf("%.3f",yy1)
complex(0.750,0.000) complex(-0.327,0.532) complex(-0.048,-0.329) \
complex(-0.048,0.329) complex(-0.327,-0.532)
> ifft(x1,yy1)
> printf("%.3f",x1)
complex(0.000,0.000) complex(0.250,0.000) complex(0.500,0.000)
complex(0.000,0.000) complex(0.000,0.000)
```

The FFT and inverse FFT of two-dimensional array

$$\mathbf{x} = \begin{bmatrix} 0 & 0.2 \\ 0.4 & 0.6 \\ 0.8 & 1 \end{bmatrix}$$

can be calculated by the following commands.

```
> array double x[3][2]
> array double complex yy1[3][2], x1[3][2]
> array double complex y2[2][2], x2[2][2]
> int dim[2] = {2, 2}
> lindata(0, 1, x)
> x
0.000000 0.200000
0.400000 0.600000
0.800000 1.000000
> fft(yy1, x)
> printf("%.3f",yy1)
complex(3.000,0.000) complex(-0.600,0.000)
complex(-1.200,-0.693) complex(0.000,-0.000)
complex(-1.200,0.693) complex(0.000,0.000)
> ifft(x1, yy1)
> printf("%.3f",x1)
complex(0.000,0.000) complex(0.200,0.000)
complex(0.400,0.000) complex(0.600,0.000)
complex(0.800,0.000) complex(1.000,0.000)
> fft(yy1, x, dim)
> printf("%.3f",yy1)
```

```
complex(1.200,0.000) complex(-0.400,0.000)
complex(-0.800,0.000) complex(0.000,0.000)
complex(0.000,0.000) complex(0.000,1.200)
> fft(y2, x, dim)
> printf("%.3f",y2)
complex(1.200,0.000) complex(-0.400,0.000)
complex(-0.800,0.000) complex(0.000,0.000)
> ifft(x2, y2, dim)
> printf("%.3f",x2)
complex(-0.000,0.000) complex(0.200,0.000)
complex(0.400,0.000) complex(0.600,0.000)
> ifft(x2, y2)
> printf("%.3f",x2)
complex(-0.000,0.000) complex(0.200,0.000)
complex(0.400,0.000) complex(0.600,0.000)
```

24.15 Convolution and Filtering

The convolution of two functions $x(t)$ and $y(t)$, denoted as $x * y$, is defined by

$$x * y \equiv \int_{-\infty}^{\infty} x(\tau)y(t-\tau)d\tau$$

in the time domain with $x * y = y * x$. According to the theorem of convolution, if $X(f)$ and $Y(f)$ are Fourier transforms of $x(t)$ and $y(t)$, that is,

$$x(t) \Longleftrightarrow X(f) \qquad \text{and} \qquad y(t) \Longleftrightarrow Y(f)$$

then

$$x * y \Longleftrightarrow X(f)Y(f)$$

If functions are digitized as two arrays, *x* of size n and *y* of size m, the sizes of two convolution arrays *x* and *y* are expanded to $m + n - 1$ and zero padded internally. The FFT algorithm is used to compute the discrete Fourier transform of *x* and *y*. Multiplying two transforms together component by component, then using the inverse FFT algorithm to take the inverse discrete Fourier transform of the products, the result is the convolution of arrays *x* and *y*.

Function **conv**() with the prototype of

```
int conv(array double complex c[&],
         array double complex x[&], array double complex y[&]);
```

calculates the convolution of two arrays *x* of size n and *y* of size m. If both arrays *x* and *y* are real type, the result is a one-dimensional array *c* of size $n + m - 1$. If either one of *x* and *y* is complex type, the result *c* is complex type. If *x* and *y* are considered as two vectors of polynomial coefficients, the convolution of *x* and *y* is equivalent to the multiplication of these two polynomials.

Deconvolution is the opposite operation of convolution. Function **deconv**() with the prototype of

```
int deconv(array double complex u[&], array double complex v[&],
       array double complex q[&], ... /* array double complex r[&]);
```

deconvolves vector *v* out of vector *u* using long division. If *u* and *v* are considered as two vectors of polynomial coefficients, the deconvolution of *u* out of *v* is equivalent to the polynomial division. The quotient is stored in vector *q* and remainder in optional argument *r* so that u = conv(v, q)+r. If both arrays *u* of size n and *v* of size m are real type, the quotient *q* of size $n - m + 1$ and remainder *r* of size n are real type. If either one of *u* and *v* is complex type, the results *q* and *r* are complex type.

For example, given two polynomial functions $x(t)$ and $y(t)$ as,

$$\begin{aligned} x(t) &= t^5 + 2 * t^4 + 3 * t^3 + 4 * t^2 + 5 * t + 6; \\ y(t) &= 6 * t + 7; \end{aligned}$$

the result of convolution of $x(t) * y(t)$ or multiplication of two polynomials $x(t)$ and $y(t)$ is

$$c(t) = x(t) * y(t) = 6 * t^6 + 19 * t^5 + 32 * t^4 + 45 * t^3 + 58 * t^2 + 71 * t + 42$$

The deconvolution of $y(t)$ out of $c(t)$ or $x(t)$ out of $c(t)$ is the division of polynomial $c(t)$ by $y(t)$ or $c(t)$ by $x(t)$. These calculations can be performed by the following commands.

```
> array double x[6]={1,2,3,4,5,6},y[2]={6,7}, c[6+2-1]
> conv(c,x,y)
> printf("%.2f", c)
6.00 19.00 32.00 45.00 58.00 71.00 42.00
> deconv(c,y,x)
> printf("%.2f", x)
1.00 2.00 3.00 4.00 5.00 6.00
> deconv(c,x,y)
> printf("%.2f", y)
6.00 7.00
```

Two-dimensional convolution is analogous in form to one-dimensional convolution. Thus for two functions $f(x, y)$ and $g(x, y)$, we define

$$f(x, y) * g(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(\alpha, \beta) g(x - \alpha, y - \beta) d\alpha d\beta$$

If $F(u, v)$ and $G(u, v)$ are two-dimensional Fourier transforms of $f(x, y)$ and $g(x, y)$, respectively, according to the convolution theorem, the following relation is valid

$$f(x, y) * g(x, y) \Longleftrightarrow F(u, v)G(u, v)$$

Function **conv2**() with the prototype of

```
    int conv2(array double complex c[&][&], array double complex f[&][&],
              array double complex g[&][&], ... /* [string_t method] */);
);
```

calculates the two-dimensional convolution of matrices *f* and *g*. In the calculation, if the size of *f* is n_a-by-n_b and the size of *g* is m_a-by-m_b, both matrices *f* and *g* are expanded to $(n_a + m_a - 1)$-by-$(n_b + m_b - 1)$ and zero padded internally. The FFT algorithm is used to compute the discrete Fourier transform of *x* and *y*. Multiplying two transforms together component by component, then using the inverse FFT algorithm to take the inverse discrete Fourier transform of the products, the answer is the convolution of matrices *f* and *g*. The size of the array *c* depends on the optional argument *method*. If the value of optional argument

```
#include <math.h>
#include <chplot.h>
#include <numeric.h>

int main() {
   int i, j;
   array double g[3][3]={{-1,0,1},{-2,0,2},{-1,0,1}};
   array double x[16], y[16], z1[256], f[16][16], H[18][18], V[18][18],Z[18][18], Z1[256];

   linspace(x,0,16);
   linspace(y,0,16);
   for(i=3; i<13; i++)
      for(j=3; j<13; j++) {
         z1[i*16+j]=1;
         f[i][j] = 1;
   }
   plotxyz(x,y,z1);                    /* original image */
   conv2(H,f,g);
   conv2(V,f,transpose(g));
   Z = H .* H + V .* V;                /* magnitude of the pixel value */
   for(i = 0; i<16; i++)
      for(j=0; j<16; j++)
         Z1[i*16+j] = Z[i+1][j+1];
   plotxyz(x,y,Z1);                    /* edge finded image */
}
```

Program 24.9: A program using **conv2**().

method is `"full"`, the size of *c* in each dimension is equal to the sum of the corresponding dimensions of the input matrices, minus 1. If the value of optional argument *method* is `"same"`, *c* contains the central part of 2D convolution with the same size as matrix *f*. If the value of optional argument *method* is `"valid"`, *c* contains only those parts of 2-D convolution that are computed without the zero-padded edges. The size of *c* is $(m_a - m_b + 1) \times (n_a - n_b + 1)$ where the size of *f* must be bigger than the size of *g*. By default, the *method* is `"full"`.

For example, in image processing, a Sobel filter is a simple approximation to the concept of edge detection. Convoluting the original two-dimensional image data with a Sobel filter

$$g_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

detects the edges of an image in the x-direction. Similarly, when convoluting a Sobel filter

$$g_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

which is the transpose of matrix g_x, edges in y-direction can be detected. This edge detection using function **conv2**() can be performed by Program 24.9 with output displayed in Figure 24.9

Function **filter**() with the prototype of

```
int filter(array double complex v[&], array double complex u[&],
    array double complex x[&], array double complex y[&], ...
    /* [array double complex zi[&], array double complex zf[&]] */);
```

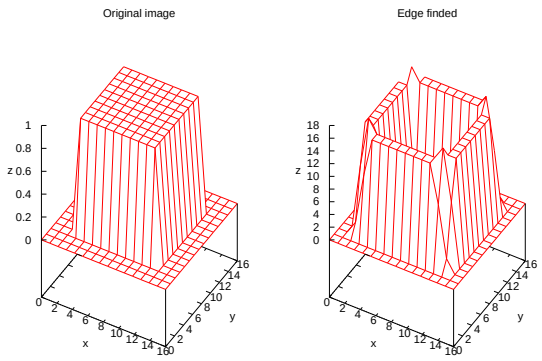


Figure 24.9: The result from two-dimensional convolution function **conv**2().

filters the data in vector *x* with a filter represented by vectors *u* and *v* to create the filtered data *y*. The filter is a direct form II transposed implementation of the standard difference equation:

$$\begin{aligned} y(n) &= v_0 * x(n) + v_1 * x(n-1) + ... + v_{nb} * x(n-nb-1) \\ &\quad -u_1 * y(n-1) - ... - u_{na} * y(n-na-1) \end{aligned}$$

The input-output description of this filtering operation in the z-transform domain is a rational transfer function

$$Y(z) = \frac{v_0 + v_1 z^{-1} + \cdots + v_{nb-1} z^{-nb-1}}{1 + u_1 z^{-1} + \cdots + u_{na-1} z^{-na-1}} X(z)$$

The numerator coefficients with zeros of the system transfer function *v*, denominator coefficients with poles of the system transfer function *u*, and vector *x* of input data can be of any supported arithmetic data type and size. Conversion of the data to double complex is performed internally. Vector *y*, which is the same size as *x*, contains the result of the filtered output. The optional arguments *zi* and *zf* are used to set the initial values of delays and get the final delays of the filter. They shall be double complex data type. Assume that the sizes of vectors *u* and *v* are n_a and n_b, respectively. The sizes of vectors *zi* and *zf* shall be $(\max(n_a, n_b) - 1)$ and $\max(n_a, n_b)$, respectively. The leading coefficient of denominator u_0 must be non-zero, as the other coefficients are divided by u_0.

Application of function **filter**() is illustrated in Program 24.10. This example shows how to use a FFT algorithm to find the spectrum of signals that are buried in noises and use a filter algorithm to filter undesired signals. Figure 24.10 shows the original and filtered signals in the time domain. The figure on the left hand side is the original signals with three sinusoidal components at frequencies of 5, 15, 30 Hz, which are buried in white noises simulated with the uniform random number generator function **urand**(). The figure on the right hand side is signals after a sixth order IIR filter with a passband 10 to 20 Hz. The filter with coefficients *u* and *v* given inside Program 24.10 will keep the 15 Hz sinusoidal signals and, get rid of the 5 and 30 Hz sinusoids and other white noises. The signals in the frequency domain are obtained by a fast Fourier transform using function **fft**(). Figure 24.11 shows the original and filtered signals in the frequency domain. The figure on the left hand side shows the original signals with three separate main frequency spectrum and some noise frequency. After filtering, the signals as shown on the right hand side of the figure contain mainly 15 Hz frequencies and a few noise components.

Function **filter**2() with the prototype of

```
#include <stdio.h>
#include <math.h>
#include <chplot.h>
#include <numeric.h>

#define N 512

int main() {
   array double t[N], x[N], y[N], Pyy[N/2], f[N/2], u[7], v[7];
   array double complex Y[N];
   int i;
   class CPlot plot;

   u[0]=1;u[1]=-5.66792131;u[2]=13.48109005;u[3]=-17.22250511;
   u[4]=12.46418230;u[5]=-4.84534157;u[6]=0.79051978;
   v[0]=0.00598202; v[1]=-0.02219918; v[2]=0.02645738; v[3]=0;
   v[4]=-0.02645738;v[5]=0.02219918;v[6]=-0.00598202;

   linspace(t,0,N-1);
   t = t/N;
   for (i=0; i<N; i++) {
      x[i] = sin(2*M_PI*5*t[i]) + sin(2*M_PI*15*t[i]) + sin(2*M_PI*t[i]*30);
      x[i]=x[i]+3*(urand(NULL)-0.5);
   }

   filter(v,u,x,y);
   plotxy(t,x,"Time domain original signal","Time (second)","Magnitude ");
   plotxy(t,y,"Time domain filtered signal","Time (second)","Magnitude ");

   fft(Y,x);
   for (i=0; i<N/2; i++)
      Pyy[i] = abs(Y[i]);
   linspace(f,0,N/2);
   plotxy(f,Pyy,"Frequency domain original signal","frequency (Hz)","Magnitude (db)");
   fft(Y,y);
   for (i=0; i<N/2; i++)
      Pyy[i] = abs(Y[i]);
   linspace(f,0,255);
   plotxy(f,Pyy,"Frequency domain filtered signal","frequency (Hz)","Magnitude (db)");
}
```

Program 24.10: A program using **filter**().

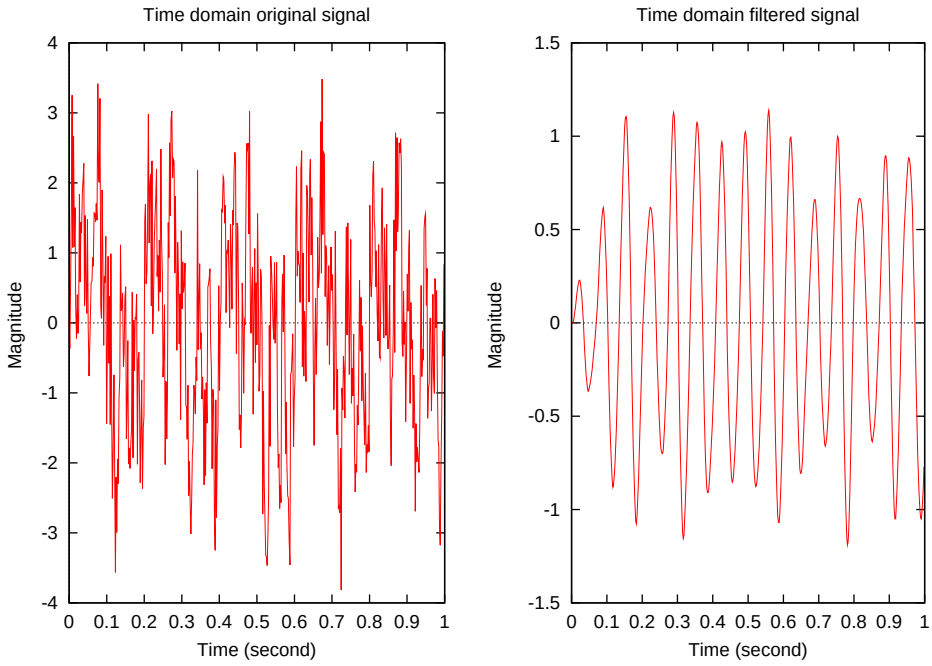


Figure 24.10: The original and filtered signals in the time domain.

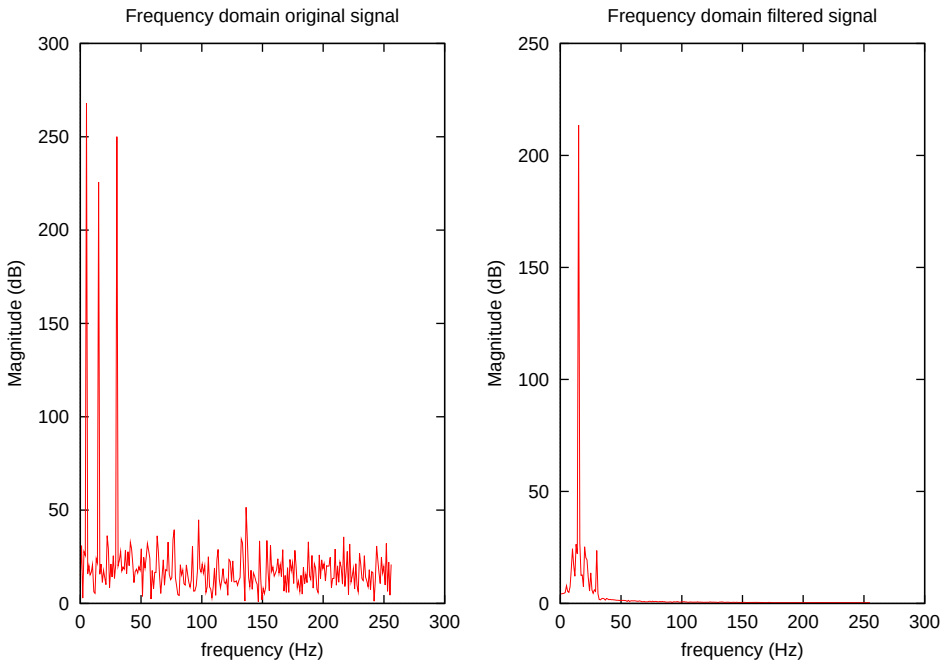


Figure 24.11: The original and filtered signals in the frequency domain.

```
#include <math.h>
#include <chplot.h>
#include <numeric.h>

int main() {
   int i, j;
   array double u[3][3]={{1,2,1},{2,4,2},{1,2,1}};
   array double s[3][3]={{1,2,1},{2,4,2},{1,2,1}};
   array double x[16],y[16],z1[256],z[16][16],Z[18][18],Z1[256];

   linspace(x,0,16);
   linspace(y,0,16);
   for(i=3; i<13; i++)
      for(j=3; j<13; j++) {
         z1[i*16+j]=1;
         z[i][j] = 1;
   }
   plotxyz(x,y,z1);
   filter2(Z,u,z,"full");
   for(i = 0; i<16; i++)
      for(j=0; j<16; j++)
         Z1[i*16+j] = Z[i+1][j+1];
   plotxyz(x,y,Z1);
}
```

Program 24.11: A program using **filter**2().

```
int filter2(array double complex y[&][&], array double complex u[&],
      array double complex x[&], ...  /* [string_t method] */);
```

computes the full two-dimensional convolution of the FIR filter with the input matrix *x* of size $(n_x \times m_x)$. Inside function **filter**2(), the input filter *u* is rotated 180 degrees first, function **conv2**() is then called to implement the filtering operation. Given the size of the filter matrix *u* of $(n_u \times m_u)$, the size of array *y* for the filtered data depends on the optional argument *method*. If the value of optional argument *method* is `"same"`, *y* contains the central part of 2D convolution with the same size as matrix *x*. By default, the *method* is `"same"`. If the value of optional argument *method* is `"full"`, the size of *y* in each dimension is equal to the sum of the corresponding dimensions of the input matrices, minus 1. If the value of optional argument *method* is `"valid"`, *y* contains only those parts of 2-D convolution that are computed without the zero-padded edges. The size of *y* is $(m_x - m_u + 1) \times (n_x - n_u + 1)$, where the size of *x* must be bigger than the size of *u*.

For example, in image processing, a Sobel filter below

$$g_x = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

can be used to smooth an image. Program 24.11 illustrates how to use the above Sobel mask to convolute an image. The original image and filtered one using function **filter**2() are displayed in Figure 24.12

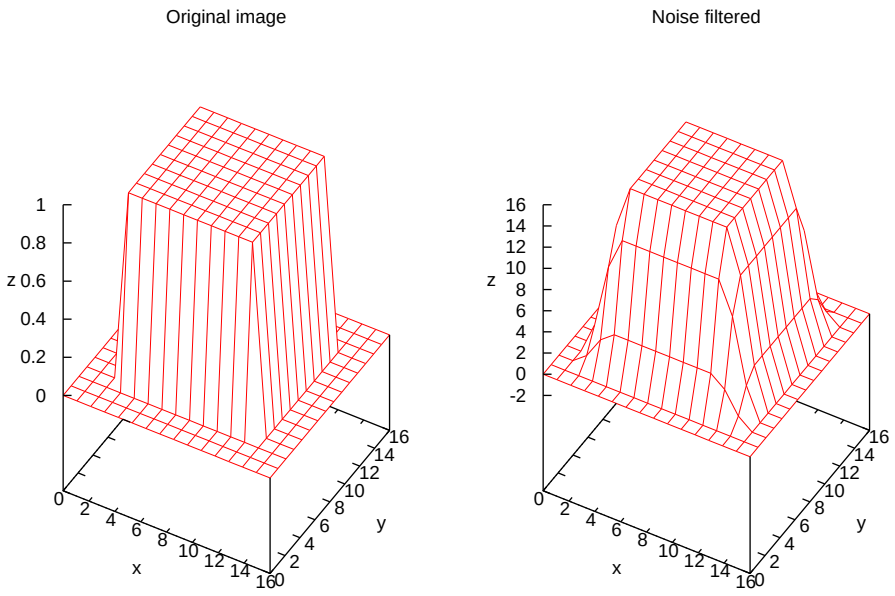


Figure 24.12: The original image and filtered one using function **filter**2().

24.16 Cross Correlation

Given two functions $x(t)$ and $y(t)$, and their corresponding Fourier transforms $X(f)$ and $Y(f)$, the cross correlation of these two functions, denoted as $xcorr(x, y)$, is defined by

$$xcorr(x, y) \equiv \int_{-\infty}^{\infty} x(t + \tau)y(t)d\tau$$

$$xcorr(x, y) \iff X(f)Y^*(f)$$

The asterisk denotes a complex conjugate. The correlation of a function with itself is called its autocorrelation. In this case

$$xcorr(x, x) \iff |X(f)^2|$$

In the numerical implementation, the sizes of two data sequences *x* of the same size n are expanded to $(2n - 1)$ and padded with zeroes internally. The FFT algorithm is used to compute the discrete Fourier transforms of *x* and *y* first. Then, multiply the complex conjugate of transform of *y* with the transform of *x* component by component. Finally, take the inverse FFT of the products, and the result is the cross correlation $xcorr(x, y)$ for arrays *x* and *y*.

Function **xcorr**() with the prototype of

```
int xcorr(array double complex c[&],
          array double complex x[&], array double complex y[&]);
```

calculates the cross correlation of two arrays *x* and *y* of the same size n. If both arrays *x* and *y* are real type, the cross correlation *c* is a one-dimensional array of size $(2n - 1)$. If either one of *x* and *y* is complex type, the result *c* is complex type.

For example, for two sequences *x*[n] and *y*[n], analytically, the cross correlation of *x* and *y* can be calculated by

$$c[k] = \sum_{j=\max\{1,k-n+1\}}^{\min\{k,m\}} y[j]x[n+j-k]; \qquad k = 1, 2, \cdots, n+m-1$$

Given $x = \{1, 2, 3, 4\}$ and $y = \{3, 2, 0, 1\}$, then

$$\begin{aligned}
c[1] &= y[1]x[4] = 12 \\
c[2] &= y[1]x[3] + y[2]x[4] = 17 \\
c[3] &= y[1]x[2] + y[2]x[3] + y[3]x[4] = 12 \\
c[4] &= y[1]x[1] + y[2]x[2] + y[3]x[3] + y[4]x[4] = 11 \\
c[5] &= y[2]x[1] + y[3]x[2] + y[4]x[3] = 5 \\
c[6] &= y[3]x[1] + y[4]x[2] = 2 \\
c[7] &= y[4]x[1] = 1
\end{aligned}$$

The above cross correlation can be calculated by the following commands.

```
> #define N 4
> array double x[1:N] = {1, 2, 3, 4}
> array double y[1:N] = {3, 2, 0, 1}, c[1:2*N-1]
> xcorr(c,x,y)
> printf("%.2f", c)
12.00 17.00 12.00 11.00 5.00 2.00 1.00
> c[1]
12.000000
> c[7]
1.000000
```

Chapter 25

Bibliography

1. ANSI, *ANSI Standard X3.9-1978, Programming Language FORTRAN* (revision of ANSI X2.9-1966), American National Standards Institute, Inc., NY, 1978.

2. Cheng, H. H., The Ch Language Environment Homepage, `http://www.softintegration.com/docs/ch/ .`

3. Cheng, H. H., CGI Programming in C, *C/C++ Users Journal*, Vol. 14, No. 11, November, 1996, pp. 17-21.

4. Cheng, H. H., Scientific Computing in the Ch Programming Language, *Scientific Programming*, Vol. 2, No. 3, Fall, 1993, pp. 49-75.

5. Cheng, H. H., Handling of Complex Numbers in the Ch Programming Language, *Scientific Programming*, Vol. 2, No. 3, Fall, 1993, pp. 76-106.

6. Cheng, H. H., Extending C with Arrays of Variable Length, *Computer Standards and Interfaces*, Vol. 17, 1995, pp. 375-406.

7. Cheng, H. H., Extending C and FORTRAN for Design Automation, *ASME Trans., Journal of Mechanical Design*, Vol. 117, No. 3, 1995, pp. 390-395.

8. Cheng, H. H., Programming with Dual Numbers and its Applications in Mechanisms Design, *Engineering with Computers, An International Journal for Computer-Aided Mechanical and Structural Engineering*, Vol. 10, No. 4, 1994, pp. 212-229.

9. Cheng, H. H., Adding References and Nested Functions to C for Modular and Parallel Programming, The ANSI C Standard Committee X3J11.1 Meeting, NCEG, X3J11.1/93-044, October 22, 1993.

10. Churchill, R. V. and Brown, J. W., Churchill, R. V. and Brown, J. W., *Complex Variables and Applications*, Fourth edition, McGraw-Hill Book Co., NY, 1984.

11. IEEE, *ANSI/IEEE Standard 754-1985, IEEE Standard for Binary Floating-Point Arithmetic*, Institute of Electrical and Electronic Engineers, Inc., Piscataway, NJ, 1985.

12. ISO, *ISO/IEC Standard 9899:1990, Programming Language C*, International Standards Organization, Geneva, 1990.

13. ISO/IEC, *Information Technology, Programming Languages - FORTRAN*, 1539:1991E, ISO, Geneva, Switzerland.

14. Joy, W., *An Introduction to the C Shell*, Department of Electrical Engineering and Computer Science, University of California at Berkeley, 1980.

15. Kahan, W., Branch Cuts for Complex Elementary Functions, or Much Ado about Nothing's Sign Bit, *The State of the Art in Numerical Analysis* (ed. Iserles & Powell), 1987, Oxford Univ. Press; *Proc. of the Joint IMA/SIAM Conference*, April 14-18, 1986.

16. Kernighan, B. W. and Ritchie, D. M., *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, second edition, 1988.

17. Marsden, J. E., *Basic Complex Analysis*, W. H. Freeman and Company, San Francisco, 1973.

18. The MathWorks, Inc., *MATLAB Function Reference*, Version 6, 2001.

19. Stroustrup, B., *The C++ Programming Language*, Addison-Wesley, Publishing Company, Inc., 1987.

Appendix A

Known Problems and Platform Specific Features

A.1 Platform Specific Features

Most functions in Ch are supported across different platforms. Functions which are not supported in specific platforms are listed in *The Ch Language Environment — Reference Guide*. Other platform specific features and problems are described below.

A.1.1 Solaris

1. The system variable `_ignoreeof` can not be set to true, because EOF is used to terminate a login shell at the startup session.

A.1.2 Windows NT/2000/XP/Vista/Windows 7

1. Users of the Ch language environment can work in Windows NT/2000/XP/Vista/Windows 7 as if they were in a Unix environment. A forward slash / is treated as a directory notation. The disk drive can be specified by prefixing the path with "drive:", For example, drive C can be used as **C:/Ch/bin/ch**. All native DOS commands in Windows command shell are supported in Ch shell.

2. The native MS-DOS commands in Table A.1 under a Ch shell are exactly the same as they are in Windows. For example, options of a command can be specified using a forward slash (/).

3. The native MS-DOS commands in Table A.2 under a Ch shell in Windows NT/2000/XP/Vista/Windows 7 are exactly the same as they are in Windows NT/2000/XP/Vista/Windows 7. For example, options of a command can be specified using a forward slash (/).

A.2 Functions Not Supported in Specific Platforms

See "Functions Not Supported in Specific Platforms" in Appendix of *The Ch Language Environment — Reference Guide.*

Table A.1: DOS commands in Ch shell.

Command	Description
cls	Clear the screen.
copy	Copy one or more files to another location.
del	Delete one or more files.
dir	Display a list of files and subdirectories in a directory.
endlocal	
erase	Delete one or more files.
ren	Rename a file or files.
	rd is valid only at command prompt.
time	Display or set the system time.
title	Set the title for the command prompt window.
type	display the content of a file.
ver	Display the Windows version number.
verify	Tells Windows whether to verify that your files are.
	written correctly to a disk.
vol	Display a disk volume label and serial number.

Table A.2: DOS commands in Ch shell in Windows NT

Command	Description
move	Move files or directory (In Windows NT only).
start	Start a separate window to run a program or command.
	(in Windows NT only).

Appendix B

Comparison with C and Implementation-Defined Behaviors

B.1 New C99 Features Supported in Ch

Ch supports all features in the C89 Standard. Ch also supports the following major new features added in the ISO C99 Standard.

1. The features of IEEE 754 standard for floating-point arithmetic are transparent to users. Real numbers are represented in the entire real line with metanumbers ± 0.0, $\pm$Inf, and NaN. Mathematical functions with real numbers are defined in the entire real domain.

2. Data types of float complex, double complex, and long double complex.

3. Data types of long long and unsigned long long.

4. Variable length arrays (VLAs).

5. Generic functions for polymorphism.

6. Mixed executable code and declaration as shown below:

```
x = 4;
int n = 2*x;
int a[n];
```

7. C++ style comment symbol `//` is added.

8. The identifier `__func__` inside a function or member function of a class contains the name of the function.

9. Variable argument list macros use the ellipsis notation in the arguments and an identifier **__VA_ARGS__** in the replacement list. For example,

```
#define debug(...)    printf(__VA_ARGS__)
debug("x = %d\n", x);
```

   results in

```
printf("x = %d\n", x);
```

10. Support the header file `stdbool.h` with type `bool`, and macros `true` and `false`.

11. Support the header files **complex.h, fenv.h, inttypes.h, iso646.h, stdbool.h, stdin.h, tgmath.h**, as well as **wchar.h** and **wctype.h**.

12. The implicit int declaration is removed.

```
fun(int i) { // ERROR: implicit int type for fun
   ...
}
int fun2(i) { // ERROR: implicit int type for i
   ...
}
```

13. Add `va_copy()` in the header file `stdarg.h.`

14. Keywords `restrict` and `inline` are recognized.

15. The hexadecimal floating-point contants. For example,

```
> 0X2P3
16.0000
> 0x1.1p0
1.0625
> 0x1.1p1F
2.12
```

B.2 Summary of Extensions to C

1. C interpreter. Ch is an integrated language environment. C programs can readily run in Ch across different platforms without lengthy compile/link/execute/debug cycles.

2. Classes in C++.

3. Command interpreter. Ch is similar to C-shell in command mode.

4. Safe Ch can be used for across-platform network computing.

5. All mathematical functions, I/O functions, and other generic functions are polymorphic.

6. Complex operations and complex functions are polymorphic. They are defined in the entire complex domain with complex metanumbers ComplexInf and ComplexNaN. Different branches of multiple-valued complex functions can be obtained by mathematical functions with optional arguments. There is only *one* complex infinity and *one* complex-not-a-number.

7. Consistent implicit and explicit data type conversions among char, unsigned char, short, unsigned short, int, unsigned int, float, double, float complex, double complex, as well as their metanumbers.

8. Built-in operations and functions will guarantee the delivery of correct numerical values or NaN, ComplexNaN.

9. String through the string type **string_t** is a first-class object and can be used for symbolic computing. New string functions **str2ascii()**, **str2mat()**, **stradd()**, **strgetc()**, **strputc()**, and **strrep()** are added.

10. Functions can be nested and recursively nested. Functions can also be mutually recursive, i.e, recursive functions can call each other.

11. Class/struct/union/enum tags and class/struct/union/enum variables in Ch share the same namespace.

12. Using goto statement to transfer the program execution from a nested function to a higher level outer function where the label is defined.

13. The index of a switch statement can be a string in addition to integral constants.

14. A computational array is treated as a first-class object like in Fortran 90.

15. Arrays of variable length including deferred-shape arrays, assumed-shape arrays, and pointer to assumed-shape arrays are supported.

16. Arrays of adjustable range. The range of subscript for an index of array can be adjusted.

17. A member of class/struct/union can be a pointer to assumed-shape arrays.

18. Reference for simple data types char, short, int, float, double, as well as data types qualified by signed, unsigned, long, complexcan be declared. Functions can be called by reference.

19. Variables of different data type can be passed to arguments of functions by reference through variables or arrays.

20. Array of reference is supported. Arrays of different shape and data type can be passed to arguments of functions with array of reference type.

21. Functions and variables can be defined only once at the given scope and lexical level. It is guaranteed that all function calls to a function are governed by a prototype, that all the prototypes for the same function are compatible, and that all the prototypes match the function definition even for a program that is divided into many separate files. The same is true for external variables. Data types in all declarations and the definition of an external variable must be compatible across different files. For variables of array, both shape and extent of an external array at different declarations and its definition must also be compatible.

22. Storage-class specifiers `__declspec(local)` and `__declspec(global)` are added.

23. The C shell style foreach-loop is supported.

24. Variables and memories allocated by memory allocation functions are automatically initialized.

25. Relational *exclusive or* operator `^^` is supported.

26. Functional type cast operations, such as float(3), are supported.

27. If the rvalue of the shifting operation is negative, the shifting direction for lvalue is reversed.

28. The identifier `__class__` inside a member function of a class contains the name of the class. The identifier `__class_func__` inside a member function of a class contains the names of both class and member function.

29. Several system variables such as `_path`, `_fpath`.

30. Integral operations do not use the longest data type. Operations in preprocessor commands and regular commands use the same execution environment. All integral/floating-point operations and generic functions are available in preprocessing as if they were used in program execution.

31. For safe operation, the array boundary is checked when an element of array is accessed. The string length is checked when an array of chars is updated as a string.

32. A short-hand parameter list of function such as `int func(double a, b, c)` is supported. If a typedefed identifier is used as an identifier for a function parameter list, it shall be preceded with a type specifier.

33. Many generic functions are built into the language.

34. Toolkits and many function files such as `plotxy(), plotxyz(), plotxyf(), plotxyzf()` are available.

35. The pointer freed by the generic function **free**() is reset to NULL.

36. NULL is a default keyword. It has both status of int data type and pointer to void data type. As int, it has a value of 0. As pointer to void, it points to nowhere. It can be used in preprocess directives

```
ifdef NULL is true
ifndef NULL is false
defined(NULL) is true
```

37. C requires that a variadic function have at least one argument specified. C++/Ch doesn't require it.

```
int func(...); // ok in C++/Ch, bad in C.
```

38. The functions **printf()** and **scanf()** can be used without the format control string. The default I/O format described in section 20.4 will be used. For example,

```
int i = 90;
printf(i, " ", 2*i, "\n");
printf("Please input a number\n");
scanf(&i);
```

39. Programs can be written as scripts without the function main().

B.3 Implementation Notes

B.3.1 Unlimited Properties

The following properties are not limited in Ch, but limited by the available memory of a particular computer system where Ch runs.

1. Dimension of array.

2. Nesting levels of compound statements, iteration structures of for-loop, do-loop, while-loop, and selection control switch statement.

3. Nesting levels of functions.
4. Depth of class/struct/union and scope nesting.
5. Nesting levels of conditional inclusion.
6. Nesting levels of parenthesized expressions with a full expression.
7. Number of macro identifiers.
8. Number of parameters in a function definition.
9. Number of arguments in a function call.
10. Number of parameters in one marco definition.
11. Arguments in one macro invocation.
12. Size of objects for array/class/struct/union.
13. Number of members in a single class/struct/union.
14. Number of elements in a single enum.
15. Levels of nested class/struct/union definitions in a single struct-declaration-list.
16. Number of function call stack depth.
17. Number of typedef definitions.
18. Number of class/struct/enum/union definitions.

B.3.2 Defined Properties

Many implementation-dependent behaviors in C are well defined in Ch across different platforms for the maximum portability. These so-called implementation-dependent properties are defined as follows.

1. The data type *char* is signed with 1 byte.
2. The data type *short* occupies 2 bytes.
3. The data type *int* occupies 4 bytes.
4. The data type *long* is the same as *int*.
5. The data type *long long* occupies 64 bits.
6. The data type *float* occupies 4 bytes.
7. The data type *double* occupies 8 bytes.
8. The data type *complex* occupies 8 bytes, 4 bytes for real and imaginary part each.
9. The data type *double complex* occupies 16 bytes, 8 bytes for real and imaginary part each.
10. Nesting levels for `#included` files is 32.

11. The maximum number of characters in a character string is 5119.
12. The maximum number of characters in a logical source line is 5119.
13. The maximum number of characters of a file name is 5119.
14. The maximum number of significant initial characters in an internal identifier or a macro name is 5119.
15. The maximum number of significant initial characters in an external identifier is 5119.
16. The maximum number of external identifiers is 32767.
17. The maximum number of identifiers with block scope declared in one block is 32767.
18. The maximum number of case labels for a switch statement is INT_MAX-1.
19. The number of case labels for a switch statement is INT_MAX-1.
20. The maximum extent for an array dimension is the value of INT_MAX.
21. The upper limit of a dimension of array is the value of INT_MAX-1.
22. The *if-else if-else* statements can be nested. Each layer can have a maximum of 127 branches.
23. Up to three indirection pointers can be declared.

B.3.3 Temporarily Features

The following tricks are used to temporarily ease the porting of C code to Ch. They will be C conforming in the future.

1. The **sizeof** operator in ISO C is treated as a generic function in Ch. If the operand of sizeof operator is an expression with multiple operands or the result of sizeof operation is not used as the last operand in an expression, the operand of sizeof operator shall be enclosed inside a pair of parentheses. For example,

   ```
   double a[3];
   int l = sizeof a/sizeof a[1]*4-sizeof a;
   ```

   shall be changed to

   ```
   double a[3];
   int l = sizeof(a)/sizeof(a[1])*4-sizeof a;
   ```

2. The preprocessor directives starting with '#' are C conforming except that

   (1) macro appearing in their own expansion are reexpanded. That means the following code is invalid:

```
#define sqrt(x) ((x)<0? sqrt(-x) : sqrt(x))  // Error

enum {
  _MACRO1,
#define _MACRO1 _MACRO1   // Error
  _MACRO2
#define _MACRO2 _MACRO2   // Error
};
```

(2) The escape character `'\0'` may not work properly in conversion of tokens to strings as shown in the following example:

```
#define print2(a, b) printf(#a "<" #b "=%d\n", (a)<(b) )
int main() {
  print2('\0', 10);
  print2('\\',10);
  print2('\n',10);
}
```

The output from Ch is

```
'<10=1
'\'<10=0
'\n'<10=0
```

According to C, the output should be

```
'\0'<10=1
'\\'<10=0
'\n'<10=0
```

3. Type qualifiers **register** and **volatile** are ignored. The type qualifier **restrict** in function argument list is ignored.

4. The following generic functions fscanf(), sscanf(), cannot be used as an rvalue in an assignment statement and passed as a function argument of pointer to function. For example, the following code is not valid in Ch.

```
int (*fp)(char *, char *, ...);
void func(int (*fp)(char *, char *, ...));
fp = fscanf;   /* Bad in Ch, ok in C */
func(fscanf);  /* Bad in Ch, ok in C */
```

5. The size of **long double** is 8 bytes. The type **long double** is treated the same as **double**. The size of **long double complex** is 16 bytes. The type **long double complex** is treated the same as **double complex**. The size of **long double** will be 16 bytes and **long double complex** 32 bytes in the future.

B.3.4 Incompatibility between Ch and C

Ch is designed to be a superset of C. However, the following features of C are not supported in the current version of Ch. This is not a limitation of the Ch language environment. Rather, it reflects the programming needs of our current applications. Besides, due to the interpretive nature of the current implementation of the Ch programming language, some C features which are designed for compilers and linkers will not be necessary in Ch. The unsupported C features and incompatibilities between Ch and ISO C are summarized as follows.

1. **Function Prototype**

   The old-style function definition known as the K&R C is supported. But, the function definition, except for function main(), in Ch must begin with a type identifier even for a function that returns an int. In C, if the type qualifier is omitted, the return data type of the function is int by default. The reason is that the type checking in Ch is more rigorous as far as the returned data type of a function is concerned, which can help users to detect some hidden bugs of the program. Also, because Ch is a script file, the default int for function declaration cannot be implemented without ambiguity. For example,

   ```
   fun();    /* function fun() invoked or declaration of int fun()? */
   fun() {
       ...
   }
   ```

2. There are more keywords in Ch than in C. The following keywords in C++ have been added in Ch.
   class delete new private public this
   The following additional keywords have been added in Ch.
   ComplexInf ComplexNaN Inf NaN NULL foreach fprintf printf scanf string_t

3. Struct tag and variables use different namespaces in C. Like C++, struct tag are typedefed and put in variable namespace in Ch for object-oriented programming.

   ```
   struct tag2 {
       int i;
       /* enum tag1 is local to tag2 in C++/CH, global in C */
       enum tag1 {rich, poor, thief} e;
       int f();
   };
   ```

4. Initialization of an array with its dimension of 4 or higher. But, Ch supports arrays with dimension of 4 or higher, just not initialization when arrays are declared.

5. Very complicated (and brain-damaging) declarations (not understandable by average human beings) may not be valid in Ch.

6. In C, the number of indirection of pointers can be up to 12. In Ch, the multiple indirection of pointer up to 3 is allowed. In other words, only single, double and triple pointers can be defined, e.g., `int *ptr1, *ptr2, ***ptr3`. The rationale behind Ch is that a multiple indirection of pointer greater than two is not necessary in practical applications. There is always a better way to achieve the same programming goal for a complicated quadruple pointer indirection or higher.

7. Use

```
char c, s1[] = "ABC";
c = s1[i];
```

   instead of

```
char c;
c = "ABC"[i];
```

8. IEEE arithmetics

```
Function             C         Ch
-----------------------------------
hypot(+/-Inf, NaN)   Inf       NaN
hypot(NaN, +/-Inf)   Inf       NaN
pow(1, NaN)          1.0       NaN
pow(NaN, +/-0.0)     1.0       NaN
-----------------------------------
```

9. In C9x, when a complex number is cast to a real floating-point number implicitly, the real part of the complex number is used. In Ch, when a complex number is cast to a real floating-point number implicitly, if the imaginary part is not zero, the resulting real number is NaN. The real part of a complex number can be obtained by generic function **real()**. For example, for the following code

```
#include <complex.h>
int main() {
    double d;
    complex z;

    z = complex(1, 0);
    d = z; // d is 1 in both C9x and Ch
    z = complex(1, 2);
    d = z; // d is 1 in C9x, d is NaN in Ch
}
```

10. A character constant such as 'c' has type **int** in C. Like C++, a character constant has type **char** in Ch.

11. The dollar sign character `'$'` cannot be used in an identifier in Ch. But, it can be used in C.

B.4 Tips for Porting C to Ch

Ch is designed as a superset of C. Because of extensions to C, some obsolete features in C should be avoided so that programs can run in both C and Ch.

1. Non-portable code:

```
funct()
{ ...}
```

Portable code:

```
int funct()
{ ...}
```

2. Non-portable code:

```
funct(a, b, c)
int a;
int b;
char *c;
{...}
```

Portable code:

```
int funct(a, b, c)
int a;
int b;
char *c;
{...}
```

3. Non-portable code:

```
funct(int a, int b, char *c)
{...}
```

Portable code:

```
int funct(int a, int b, char *c)
{...}
```

4. For included the header file, add

```
#ifndef FILENAME_H
#define FILENAME_H
...
#endif
```

5. Do not use the following functions **fscanf(), sscanf()** as an rvalue in an assignment statement, or passed as a function argument of pointer to function. For example, the following code is not valid in Ch.

```
int (*fp)(char *, ...);
void func(int (*fp)(char *, ...));
fp = fscanf;   /* Bad in Ch, ok in C */
func(fscanf);  /* Bad in Ch, ok in C */
```

6. Do not use new keywords **ComplexInf, ComplexNaN, Inf, NULL, NaN, array, class, delete, foreach, fprintf, new, printf, private, public, scanf, string_t, this** in Ch as variables. Use `'which identifier'` to check if the `identifier` is a keyword or not. In some cases, you may use the technique similar to the code below to handle the namespace conflict.

```
#if defined(__cplusplus) || defined(c_plusplus) || defined(_CH_)
    int c_class;
#else
    int class;
#endif
```

7. Do not use pointers with more than triple indirections.

For portability of your code, do not use the following C features for the time being.

1. Do not initialize arrays with more than three dimensions.
2. Do not use the following functions defined in header file wchar.h.

```
extern int fwprintf(FILE *, const wchar_t *, ...);
extern int fwscanf(FILE *, const wchar_t *, ...);
extern int swprintf(wchar_t *, size_t, const wchar_t *, ...);
extern int swscanf(const wchar_t *, const wchar_t *, ...);
extern int vfwprintf(FILE *, const wchar_t *, __va_list);
extern int vwprintf(const wchar_t *, __va_list);
extern int vswprintf(wchar_t *, size_t, const wchar_t *, __va_list);
extern int wprintf(const wchar_t *, ...);
extern int wscanf(const wchar_t *, ...);
```

3. Do not use `"string"[i]`.
4. Do not use `sizeof expr`, instead use `sizeof(expr)`.

Appendix C

Comparison with C++

C.0.1 Features in Both C++ and Ch

1. Member function.
2. Mixed code and declaration.
3. The `this->` pointer.
4. Reference type and pass-by-reference.
5. Function-style type conversion.
6. Class.
7. Private/public data and functions. Members of a class definition are assumed to be private until a `public` declaration is given.
8. Static member of class/struct/union.
9. const member functions.
10. The new and delete operators.
11. The constructor and destructor.
12. Polymorphic functions.
13. The scope resolution operator `::` for member function definitions, static members, and global variables such as `::g` for global variable *g* in a local scope.
14. The I/O cout, cerr, cin with endl and ends.
15. The following using directive for cout, cerr, cin, endl, and ends.

```
using std::cout;
using std::cin;
using std::cerr;
using std::endl;
using std::ends;
```

or

```
using namespace std;
```

16. Arguments for variadic functions are optional.

```
int func(...); // ok in C++/Ch, bad in C.
```

C.0.2 Extensions to C++ Classes in Ch

1. Classes inside member functions.
2. Nested functions with classes.
3. Pass member function to argument of pointer-to-function type of functions.

C.0.3 C++ Features not Supported in Ch

1. Classes with member function definitions.

   class member function definitions should be outside a class declaration. For example, the following code won't work.

```
class tag {
    int i;
  public:
    tag() {
      i=0;
    }
    void f() {
      i++;
    }
};
```

   It should be changed as

```
class tag {
    int i;
  public:
    tag();
    void f();
};
tag::tag() {
   i =0;
}
void tag::f(void) {
   i++;
}
```

2. Virtual functions and pure virtual functions.

3. Friend function.
4. Inheritance.
5. Multi-inheritance.
6. Protected data and functions.
7. Function overloading.
8. Operator overloading.
9. Template.
10. Exception.
11. File stream I/O.
12. Function return reference type.
13. Function with default argument value.
14. Copy constructor.
15. Conversion function.
16. asm keyword.
17. linkage specifier such as 'extern C'.
18. Operators `::*` and `::->`.
19. Shorthand initialization.

```
...
myclass ob1 = myclass(i);   // ok in  both C++ and Ch
myclass ob2 = myclass(2);   // ok in  both C++ and Ch
myclass ob3(i);             // ok in C++, bad in Ch
myclass ob4(2);             // ok in C++, bad in Ch
```

20. Declaration in expression. For example,

```
if(char *c = malloc(8)) {   // ok in C++, bad in Ch
   /* use c here */
}.
for(int i = 0; i<10; i++) { // ok in C++, bad in Ch
   /* use i here */
}
```

21. Namespaces.
22. Run-time type information.
23. New cast notion.
24. New style of including header file without postfix *.h*. For example,

```
#include <stdio>
```

C.0.4 Differences Between C++ and Ch

1. The constructor of a member of a class type inside another class is not called in Ch automatically upon instantiation. A function return class can be used to assign the value of the class. For example,

```
#include <stdio.h>

class tag1 {
public:
        tag1();
        class tag1 fun();
};
tag1::tag1() {
   printf("hello from tag1 constructor\n");
}
class tag1 tag1::fun() {
  class tag1 t; /* constructor is called */

  return t;
}

class tag2 {
public:
        class tag1 document; /* constructor is NOT called */
        tag2();
};
tag2::tag2() {
   printf("hello from tag2 constructor\n");
}

int main(){
  class tag2 window;
  window.document = window.document.fun();
}
```

Appendix D

Comparison with C Shell

D.1 Syntax

Both Ch and C shells have their local shell variables besides the environment variables. Table D.1 gives a brief syntax comparison between Ch and C shell. The semantics of Ch commands listed in Table D.1 are the same as those of the corresponding commands of C shell. Commands **alias**, **history**, **remvar** and **unalias** in Ch are valid in interactive command shells only. Directive `#pragma remvar(var)` is valid inside Ch programs only. Table D.2 gives comparison of control flow between C shell and Ch.

Table D.1: Selected syntax comparisons between C shell and Ch.

C shell	Ch shell
`$#argv`	`_argc`
`$argv[*]`	`strjoin(" ", _argv[1], _argv[2], ..., _argv[_argc])`
`$argv`	`strjoin(" ", _argv[1], _argv[2], ..., _argv[_argc])`
`$*`	`strjoin(" ", _argv[1], _argv[2], ..., _argv[_argc])`
`$argv[1-n]`	`strjoin(" ", _argv[1], _argv[2], ..., _argv[n])`
`$0`	`_argv[0]`
`$argv[n]`	`_argv[n]`
`$1 $2 ... $9`	`_argv[1] _argv[2] ... _argv[9]`
`$argv[$#argv]`	`_argv[_argc]`
`set prompt = "ch> "`	`_prompt = "ch> "`
`set path = (/usr/bin /bin)`	`_path = "/usr/bin /bin"`
`umask 022`	`umask(022)`
`setenv PATH "/usr/bin /bin"`	`putenv("PATH=/usr/bin /bin")`
`echo $PATH`	`printf("%s\n", getenv("PATH"))`
`echo $PATH`	`echo $(getenv("PATH"))`
`echo $PATH`	`echo $PATH`
`echo ${PATH}`	`echo ${PATH}`
`echo $path`	`printf("%s\n", _path)`
`echo $path`	`echo $_path`
`unsetenv PATH`	`remenv("PATH")`
`printenv PATH`	`getenv("PATH")`
`unset path`	`remvar _path`
`unset path`	`#pragma remvar(_path)`
`unset i`	`remvar i`
`unset i`	`#pragma remvar(i)`
`set i =90`	`int i = 90`
`set i =91`	`i = 91`
`` `cmd $var` ``	`` `cmd $var` ``
`` `cmd $ENVVAR` ``	`` `cmd $ENVVAR` ``
`` `cmd $ENVVAR` ``	`` `cmd $(getenv("ENVVAR"))` ``
`set hostnam = `hostname``	`string_t hostnam = `hostname``
`set host = `hostname``	`_host = `hostname``

Table D.1: Selected syntax comparisons between C shell and Ch (Contd.).

C shell	Ch shell
`alias rm`	`alias rm`
`alias`	`alias`
`alias rm "rm -i"`	`alias("rm", "rm -i")`
`alias f "find . -name \!:1 -print"`	`alias("f", "find . -name _argv[1] -print")`
`alias e "echo \!* \!$ \!#"`	`alias("e", "echo _argv[*] _argv[$] _argv[#]")`
`alias rm`	`alias("rm")`
`alias`	`alias()`
`unalias rm`	`unalias rm`
`unalias rm`	`alias("rm", NULL)`
`eval ls`	`streval("‘ls‘")`
`eval ls`	`system("ls")`
`eval setenv NAME value`	`streval("putenv(\"NAME=value\")")`
`./cmd -option`	`./cmd -option`
`/usr/bin/ls *`	`/usr/bin/ls *`
`"/path with space/cmd" option`	`"/path with space/cmd" option`
`$cmd option`	`$cmd option`
`status`	`_status`
`ls ~ *`	`ls ~ *`
`ls > output`	`ls > output`
`ls | tar -cf tarfile`	`ls | tar -cf tarfile`
`ls $PATH`	`ls $(getenv("PATH"))`
`ls $PATH`	`ls $PATH`
`ls $path`	`ls $_path`
`history = 100`	`_histsize = 100`
`history`	`history`
`!l`	`!l`
`!l -agl`	`!l -agl`
`!3`	`!3`
`!-1`	`!-1`
`!!`	`!!`
`!!`	`!`
`vi `grep -l "str1 str2" *.c``	`vi `grep -l "str1 str2" *.c``
`more .cshrc .login .logout`	`more .chrc .chlogin .chlogout`
`more .cshrc .login .logout`	`more .chsrc .chlogin .chlogout`

D.2 Control Flow

Table D.2: Control flow comparisons between C shell and Ch.

Description	C shell	Ch shell

while-loop	`while(expr)`
	`  commands`
	`end`
		`while(expr) {`
		`  commands`
		`}`

foreach-loop	`foreach token(wordList)`
	`#use $token`
	`  commands`
	`end`
		`foreach(token; wordList) {`
		`// use token`
		`  commands`
		`}`

if	`if(expr)`
	`  commands`
	`endif`
		`if(expr) {`
		`  commands`
		`}`

if-else if -else	`if(expr1) then`
	`  commands1`
	`else if(expr2) then`
	`  commands2`
	`else`
	`  commands3`
	`endif`
		`if(expr1) {`
		`  commands1`
		`}`
		`else if(expr2) {`
		`  commands2`
		`}`
		`else {`
		`  commands3`
		`}`

goto	`goto label` `label: statements`	`goto label` `label: statements`

switch	`switch(expr)`
	`  case pattern1:`
	`  commands1`
	`  breaksw`
	`  case pattern2:`
	`  commands2`
	`  breaksw`
	`  default:`
	`  defaultCommands`
	`endsw`
		`switch(expr) {`
		`  case pattern1:`
		`  commands1`
		`  break;`
		`  case pattern2:`
		`  commands2`
		`  break;`
		`  default:`
		`  defaultCommands`
		`  break;`
		`}`

Appendix E

Comparison with MATLAB

Table E.1: Symbols used for comparison of MATLAB and Ch.

Symbol	Data Type
x	scalar or computational array of real or complex type.
A, *Ai*, *B*, *Bi*	computational arrays of real or complex type.
Av, *Avi*, *Bv*, *Bvi*	one-dimensional computational arrays of real or complex type.
R, *Ri*	computational arrays of real type.
Rv, *Rvi*	one-dimensional computational arrays of real type.
I, *Ii*	computational arrays of integral type.
Iv, *Ivi*	one-dimensional computational arrays of integral type.
C, *Ci*	arrays of char type.
Z, *Zi*	computational arrays of complex type.
Zv, *Zvi*	one-dimensional computational arrays of complex type.
s	scalar of real or complex type.
z	scalar complex type.
r	scalar real type.
f	scalar floating-point type.
i	scalar integral type.
p	pointer type.
str	a string.

E.1 Operators

Table E.2: Comparison of operators in MATLAB and Ch.

Operator	MATLAB	Ch
∼	∼*A*	**!***A*
	∼*s*	**!***s*
+	+*A*	+*A*
	+*s*	+*s*
−	−*A*	−*A*
	−*s*	−*s*
<	*A*<*B*	*A*<*B*
	A<*s*	*A*<*r*
	s<*A*	*r*<*A*
∼=	*A*∼=*B*	*A*! =*B*
	A∼=*s*	*A*! =*s*
	s∼=*A*	*s*! =*A*
|	*A*|*B*	*A*||*B*
	A|*s*	*A*||*r*
	s|*A*	*r*||*A*
&	*A*& *B*	*A*&& *B*
	A& *s*	*A*&& *s*
	s& *A*	*s*&& *A*
+	*A*+*B*	*A*+*B*
	A+*s*	*A*+*s*
	s+*A*	*s*+*A*
	A= *A*+*B*	*A*= *A*+*B*
	A= *A*+*B*	*A*+= *B*
−	*A*−*B*	*A*−*B*
	A−*s*	*A*−*s*
	s−*A*	*s*−*A*
	A= *A*−*B*	*A*−= *B*
∗	*A*∗*B*	*A*∗*B*
	A∗*s*	*A*∗*s*
	s∗*A*	*s*∗*A*
	A= *A*∗*B*	*A*= *A*∗*B*
	A= *A*∗*B*	*A*∗= *B*
/	*A*= *A1*/*B1*	*A*=*A1*∗**inverse**(*B1*)
	A/*s*	*A*/*s*
	A= *A*/*s*	*A*= *A*/*s*
	A= *A*/*s*	*A*/= *s*
^	*i1*^*i2*	pow(*i1*,*i2*)
	s1^*s2*	pow(*s1*,*s2*)
	I^*i*	pow(*I*,*i*)
	A^*i*	pow(*A*,*i*)

Table E.2: Comparison of operators in MATLAB and Ch (continued).

Operator	MATLAB	Ch
\	*A*=*A1**B1*	*A*=**inverse**(*A1*)∗*B1*
\	*Av*=*A**Bv*'	**linsolve**(*Av*,*A*,*Bv*)
		Av=**inverse**(*A*)∗*Bv*
		llsqsolve(*Bv*, *A*,*Av*)
'	*A*'	**transpose**(*A*)
'	*Z*'	**transpose**(**conj**(*Z*))
.∗	*A*.∗*B*	*A*.∗*B*
./	*A*./*B*	*A*./*B*
./	*s*./*B*	*s*./*A*
.^	*A*.^*B*	**pow**(*A*, *B*)
.^	*A*.^*s*	**pow**(*A*, (**array double** [n])*s*)
.^	*s*.^*A*	**pow**((**array double** [n])*s*, *A*)
.\	*A*.*B*	(not valid)

E.2 Functions and Constants

Table E.3: Comparison of functions in MATLAB and Ch.

Function	MATLAB	Ch
abs	*r*=**abs**(*s*)	*r*=**abs**(*s*)
	A=**abs**(*A*)	*A*=**abs**(*A*)
	Iv=**abs**('*str*')	**array** int *Iv*[**strlen**("*str*")]= {'*s*','*t*','*r*'}
acos	*x*=**acos**(*x*)	*x*=**acos**(*x*)
acosh	*x*=**acosh**(*x*)	*x*=**acosh**(*x*)
addpath	**addpath**('/new/dir',/new/dir2')	_path=**stradd**("/new/dir1;/new/dir2;", _path)
all	*i*=**all**(*A*)	*i*=**sum**(**!***A*=0)==0
angle	*r*=**angle**(*z*)	*r*=**carg**(*z*)
any	*i*=**any**(*A*)	*i*=**sum**(*A*! =0)! =0
asin	*x*=**asin**(*x*)	*x*=**asin**(*x*)
asinh	*x*=**asinh**(*x*)	*x*=**asinh**(*x*)
atan	*x*=**atan**(*x*)	*x*=**atan**(*x*)
atan2	*r*=**atan2**(*r1*,*r2*)	*r*=**atan2**(*r1*,*r2*)
atanh	*x*=**atanh**(*x*)	*x*=**atanh**(*x*)
balance	[*A*, *B*] = **balance**(*A1*)	**balance**(*A1*, *A*, *B*)
base2dec	*i*=**base2dec**('*str*',*i2*)	*i*=**strtol**("*str*", **NULL**, *i2*)
bin2dec()	**bin2dec**('*01010*')	
blanks	*str*=**blank**(*n*)	" "
ceil	*x*=**ceil**(*x*)	*s*=**ceil**(*s*)
choly	*A*= **chol**(*A1*)	**choldecomp**(*A1*, *A*)
clear	**clear** name	**remvar** name
	clear name	#pragma remvar(name)
compan	*A*=**compan**(*Av*)	*R*=**companionmatrix**(*Rv*)
		Z=**ccompanionmatrix**(*Zv*)
cond	*r*=**cond**(*A*)	*r*=**condnum**(*A*)
condest	*r*=**condest**(*A*)	*r*=**condnum**(*A*)
conj	*x*=**conj**(*x*)	*x*=**conj**(*x*)
conv	*Av*= **conv**(*Av1*, *Av2*)	**conv**(*Av*, *Av1*, *Av1*)
conv2	*A*= **conv2**(*A1*, *A2*)	**conv2**(*A*, *A1*, *A2*)
corrcoef	*R*= **corrcoef**(*R1*)	**corrcoef**(*R*, *R1*)
corr2	*r*= **corr2**(*R1*, *R2*)	*r*= **correlation2**(*R1*, *R2*)
cos	*x*=**cos**(*x*)	*x*=**cos**(*x*)
cosh	*x*=**cosh**(*x*)	*x*=**cosh**(*x*)
cov	*R*= **cov**(*R1*)	**covariance**(*R*, *R1*)
cross	*Rv*=**cross**(*Rv1*,*Rv2*)	*Rv*=**cross**(*Rv1*,*Rv2*)
cumprod	*A*= **cumprod**(*A1*')	**cumprod**(*A*, *A1*)
	A= **cumprod**(*A1*)	**cumprod**(*A*, **transpose**(*A1*))
	Av=**cumprod**(*Av1*)	**cumprod**(*Av*, *Av1*)
cumsum	*A*= **cumsum**(*A1*')	**cumsum**(*A*, *A1*)
	A= **cumsum**(*A1*)	**cumsum**(*A*, **transpose**(*A1*))
	Av= **cumsum**(*Av1*)	**cumsum**(*Av*, *Av1*)
dec2base()		

Table E.3: Comparison of functions in MATLAB and Ch (continued).

Function	MATLAB	Ch
dec2bin	*i*=**dec2bin**(*i2*)	**printf**("%*b*", *i2*)
		printf("%*10b*", *i2*)
dec2hex	*str*=**dec2hex**(*x*)	**sprintf**(*str*,"%*x*", *r*)
deconv	*A*=**deconv**(*A1*,*B1*)	**deconv**(*A*,*A1*,*B1*)
	[*A*, *B*]=**deconv**(*A1*,*B1*)	**deconv**(*A*,*A1*,*B1*,*B*)
deblank()	**deblank**('*str*')	(not valid)
det	*s*=**det**(*A*)	*r*=**determinant**(*A*)
		z=**cdeterminant**(Z)
diag	*Av*=**diag**(*A*)	*Rv*=**diagonal**(*R*)
		Zv=**diagonal**(*Z*)
	A=diagb(*Av*)	*R*=**diagonalmatrix**(*Rv*)
		Z=**cdiagonalmatrix**(*Zv*)
diff	*A*=**diff**(*A*)	*Rv*=**difference**(*Rv*)
		r=**derivative**(*func*,*r*)
		R=**derivatives**(*func*,*R*)
disp	**disp**(*i*)	**printf**("%d", *i*) or **printf**(i)
	disp(*f*)	**printf**("%f", *f*) or **printf**(f)
	disp(*str*)	**printf**("%s", *str*) or **printf**(str)
	disp(*A*)	**printf**("%d", *A*) or **printf**(A)
display	**display**(*i*)	**printf**("%d", *i*) or **printf**(i)
	display(*f*)	**printf**("%f", *f*) or **printf**(f)
	display(*str*)	**printf**("%s", *str*) or **printf**(str)
	display(*A*)	**printf**("%d", *A*) or **printf**(A)
dot	*x*=**dot**(*Av1*,*Av2*)	*r*=**dot**(*Rv1*,*Rv2*)
eig	*Av*=**eig**(*A1*)	**eigensystem**(*Av*, *NULL*,*A1*)
	[*Av*, *B*]=**eig**(*A1*)	**eigensystem**(*Av*, *B*,*A1*)
	[*Av*, *B*]=**eig**(*A1*,'nobalance')	**eigensystem**(*Av*, *B*, *A1*, "nobalance")
eps	**eps**	**#include**<**float.h**>
		FLT_EPSILON, DBL_EPSILON
eval	**eval**('cmd')	**system**("cmd")
	eval('expr')	**streval**("expr")
	eval(*try*,*catch*)	
eye	*R*=**eye**(*i*)	*R*=**identitymatrix**(*i*)
exp	*x*=**exp**(*x*)	*x*=**exp**(*x*)
expm	*A*= **expm**(*A1*)	**expm**(*A*,*A1*)
fclose	**fclose**	see **fclose**()
feval	**feval**('*fun*')	**streval**("*fun*")
feof	**feof**	see **feof**()
ferror	**ferror**	see **perror**() and other I/O functions
fft	*Av*= **fft**(*Av1*)	**fft**(*Av*, *Av1*)
	Av= **fft**(*Av1*, *i*)	*A*= **fft**(*A1*, *i*)
fft2	*A*=**fft2**(*A1*)	**fft**(*A*, *A1*)
	A= **fft**(*A1*, *i1*, *i2*)	*Iv*[0]=*i1*, *Iv*[*1*] =*i2*, *A*= **fft**(*A1*, *Iv*)
fftn	*A*=**fftn**(*A1*)	**fft**(*A*, *A1*) /* 3D only */
	A= **fftn**(*A1*, *i*)	*Iv*[0]=*Iv*[*1*]=*Iv*[*2*]=*i*, **fft**(*A*,*A1*,*Iv*) /* 3D only */
fftshift		

Table E.3: Comparison of functions in MATLAB and Ch (continued).

Function	MATLAB	Ch
fgetl	*str*= **fgetl**(*fid*)	*i*=**strlen**(*str*)
		getline(*fid*, *str*, *i*)
fgets	**fgets**	see **fgets**()
fix	*i*=**fix**(*r*)	*i*=*r*
filter	*Av*= **filter**(*Bv1*, *Bv2*, *Av1*)	**filter**(*Bv1*, *Bv2*, *Av1*, *Av*)
filter2	*A*= **filter2**(*A1*, *B1*)	**filter2**(*A*, *A1*, *B1*)
finite	*i*=**finite**(*s*)	*i*=**isfinite**(*s*)
	I=**finite**(*A*)	**fevalarray**(*I*,**isfinite**,*R*);
find()	*i*=**find**(*x*)	
	[*r*,*c*]=**find**(*x*)	
	I= **find**(*A*)	*i*=**findvalue**(*I*, *A*) /* i is # of values found */
findstr()	*i*=**findstr**('*str1*', '*str2*')	*p*=**strstr**(*str1*, *str2*)
fliplr	*A*= **fliplr**(*A1*)	**fliplr**(*A*, *A1*)
flipud	*A*= **flipud**(*A1*)	**flipud**(*A*, *A1*)
floor	**floor**(*x*)	**floor**(*x*)
flops	**flops**	(Not valid)
fmin	*r*= **fmin**('*fun*', *r1*, *r2*)	**fminimum**(*r*3, *r*, *fun*, *r1*, *r2*)
fmins	*Rv*= **fmins**('*fun*', *Rv1*)	**fminimums**(*r*3, *Rv*, *fun*, *Rv1*)
fplot	**fplot**('*fun*', [*r1r2*])	**fplotxy**(*fun*,*r1*,*r2*)
		fplotxyz()
		CPlot:MemberFunctions()
fprintf	**fprintf**()	see **fprintf**()
fread	**fread**()	see **fread**()
frewind	**frewind**()	see **frewind**()
fscanf	**fscanf**()	see **fscanf**()
fseek	**fseek**()	see **fseek**()
ftell	**ftell**()	see **ftell**()
funm	*R*= **funm**(*R1*,'*fun*')	**funm**(*R*, *fun*, *R1*)
	Z= **funm**(*Z*,'*fun*')	**cfunm**(*Z*, /* complex */ cfun, *Z1*)
fwrite	**fwrite**()	see **fwrite**()
fsolve	*R*= **fsolve**('*fun*', *Ri*)	**fsolve**(*R*, *fun*, *Ri*)
fzero	*r*= **fzero**('*fun*', *r1*)	**fzero**(*r*, *fun*, *r1*)
gallery	*A*=**gallery**(*name*,*arg1*)	*A*=**specialmatrix**(Name,*arg1*)
	A=**gallery**(*name*,*arg1*,*arg2*)	*A*=**specialmatrix**(*Name*,*arg1*,*arg2*)
	A=**gallery**(*name*,*arg1*,*arg2*,*arg3*)	*A*=**specialmatrix**(*Name*,*arg1*,*arg2*,*arg3*)
	A=**gallery**('*caychy*',*Av1*)	**specialmatrix**("*Cauchy*",*Av1*)
	A=**gallery**('*caychy*',*Av1*,*Av2*)	**specialmatrix**("*Cauchy*",*Av1*,*Av2*)
	A=**gallery**('*chebvand*',*Av1*)	**specialmatrix**("*ChebyshevVandemonde*",*Av1*)
	A=**gallery**('*chebvand*',*i*,*Av1*)	**specialmatrix**("*ChebyshevVandemonde*",*Av1*,*i*)
	A=**gallery**('*chow*',*i*)	**specialmatrix**("*Chow*",*i*)
	A=**gallery**('*chow*',*i*,*r1*)	**specialmatrix**("*Chow*",*i*,*r1*)
	A=**gallery**('*chow*',*i*,*r1*,*r2*)	**specialmatrix**("*Chow*",*i*,*r1*,*r2*)
	A=**gallery**('*circul*',*Av1*)	**specialmatrix**("*Circul*",*Av1*)
	A=**gallery**('*clement*',*i1*,*i2*)	**specialmatrix**("*Clement*",*i1*,*i2*)
		specialmatrix("*DenavitHartenberg*",*r1*,*r2*,*r3*,*r*4)
		specialmatrix("*DenavitHartenberg2*",*r1*,*r2*,*r3*,*r*4)

Table E.3: Comparison of functions in MATLAB and Ch (continued).

Function	MATLAB	Ch
	A=**gallery**('*dramadah*',*i1*)	**specialmatrix**("*Dramadah*",*i1*)
	A=**gallery**('*dramadah*',*i1*,*i2*)	**specialmatrix**("*Dramadah*",*i1*,*i2*)
	A=**gallery**('*fiedler*',*Av1*)	**specialmatrix**("*Fiedler*",*Av1*)
	A=**gallery**('*frank*',*i1*)	**specialmatrix**("Frank",*i1*)
	A=**gallery**('*frank*',*i1*,*i2*)	**specialmatrix**("*Frank*",*i1*,*i2*)
	A=**gallery**('*gearmat*',*i1*)	**specialmatrix**("Gear",*i1*)
	A=**gallery**('*wilk*',*i1*)	**specialmatrix**("*Wilkinson*",*i1*)
	AV=**gallery**('*house*',*Av1*)	**householdermatrix**(*Av1*,*Av*)
	[*AV*,*r*]=**gallery**('*house*',*Av1*)	**householdermatrix**(*Av1*,*Av*,*r*)
gcd()	*I*= **gcd**(*I1*, *I2*)	**gcd**(*I1*, *I2*, *I*)
	[*I*,*I*3,*I*4]=**gcd**(*I1*,*I2*)	**gcd**(*I1*, *I2*, *I*, *I*3, *I*4)
hadamard	*A*=**hadamard**(*i1*)	**specialmatrix**("*Hadamard*",*i1*)
hankel	*A*=**hankel**(*Av1*)	**specialmatrix**("*Hankel*",*Av1*)
	A=**hankel**(*Av1*,*Av1*)	**specialmatrix**("*Hankel*",*Av1*,*Av2*)
hex2dec	*i*=**hex2dec**('*str*')	*i*=**strtol**("*str*", *NULL*, *16*)
hex2num()	*r*=**hex2num**('*str*')	
hess	[*B*, *A*] = **hess**(*A1*)	**hessdecomp**(*A1*, *A*, *B*)
hilb	*A*=**hilb**(*i1*)	**specialmatrix**("*Hilbert*",*i1*)
hist	**hist**()	**histogram**()
i	**i**	**#include**<**complex.h**>
		I
ifft	*Av*=**ifft**(*Av1*)	**ifft**(*Av*, *Av1*)
	Av=**ifft**(*Av1*, *i*)	**ifft**(*A*, *A1*, *i*)
ifft2	*A*=**ifft2**(*A1*)	**ifft**(*A*,*A1*)
	A= **ifft**(*A1*, *i1*, *i2*)	*Iv*[0]=*i1*, *Iv*[two] =*i2*, **ifft2**(*A*, *A1*, *Iv*)
ifftn	*A*=**ifftn**(*A1*)	**ifft**(*A*, *A1*) /* 3D only */
	A= **iftn**(*A1*, *i*)	*Iv*[0]=*Iv*[*1*]=*Iv*[*2*]=*i*, **ifft**(*A*, *A1*,*Iv*)
imag	*r*=**imag**(*s*)	*r*=**imag**(*s*)
	R=**imag**(*A*)	*R*=**imag**(*A*)
inf	**inf**	Inf
input	*s*=**input**(*str*)	*r*=**getnum**(*str*,*r*)
int2str	*str*=**int2str**(*i*)	**sprintf**(*str*,"%d"', *i*)
interp1	*Ri*=**interp1**(*R1*, *R2*, *Ri1*)	**interp1**(*Ri*, *Ri1*, *R1*, *R2*, "*linear*")
	Ri=**interp1**(*R1*, *R2*, *Ri1*,'*linear*')	**interp1**(*Ri*, *Ri1*, *R1*, *R2*, "*linear*")
	Ri=**interp1**(*R1*, *R2*, *Ri1*,'*spline*')	**interp1**(*Ri*, *Ri1*, *R1*, *R2*, "*spline*")
interp2	*Ri*=**interp2**(*R1*, *R2*, *R*3, *Ri1*, *Ri2*)	**interp2**(*Ri*, *Ri1*, *Ri2*, *R1*, *R2*, *R*3, "*linear*")
	Ri=**interp2**(*R1*, *R2*, *R*3, *Ri1*, *Ri2*, '*linear*')	**interp2**(*Ri*, *Ri1*, *Ri2*, *R1*, *R2*, *R*3, "*linear*")
	Ri= **interp2**(*R1*, *R2*, *R*3, *Ri1*, *Ri2*, '*spline*')	**interp2**(*Ri*, *Ri1*, *Ri2*, *R1*, *R2*, *R*3, "*spline*")
inv	*A*=**inv**(*A*)	*R*=**inverse**(*R*)
		Z=**cinverse**(*Z*)
invhilb	*A*=**invhilb**(*i1*)	**specialmatrix**("*InverseHilbert*",*i1*)
isempty	*i*=**isempty**(*A*)	(not valid)
isglobal	*i*=**isglobal**(*x*)	**#include**<chshell.h >
		isvar("*x*")==**CH_SYSTEMVAR**

Table E.3: Comparison of functions in MATLAB and Ch (continued).

Function	MATLAB	Ch
ishold	*i*=**ishold**	(not valid)
isieee	*i*=**isieee**	(not valid)
isinf	*i*=**isinf**(*s*)	*i*=**isinf**(*s*)
	I=**isinf**(*A*)	**fevalarray**(*I*,**isinf**,*R*);
isletter	*i*=**isletter**(*s*)	*i*=**isalpha**(*s*)
	I=**isletter**(*A*)	**fevalarray**(*I*,**isalpha**,*R*);
isnan	*i*=**isnan**(*s*)	*i*=**isnan**(*s*)
	I=**isnan**(*A*)	**fevalarray**(*I*,**isnan**,*R*);
isreal	*i*=**isreal**(*s*)	*i*=**elementtype**(*x*) != **elementtype**(**complex**) && **elementtype**(*x*) != **elementtype**(**double complex**)
isspace	*i*=**isspace**(*s*)	*i*=**isspace**(*s*)
	I=**isspace**(*A*)	**fevalarray**(*I*,**isspace**,*R*);
issparse	*i*=**issparse**(*x*)	
isstr	*i*=**isstr**(*x*)	*i*=**elementtype**(*x*) == **elementtype**(*string_t*)
isstudent	*i*=**isstudent**	*i*=**isstudent**()
isunix	*i*=**isunix**	**#ifndef _WIN32_**
isvms	*i*=**isvms**	(not valid)
invhilb()		
j	**j**	**#include**<**complex.h**> I
lasterr	**lasterr**(")	see **perror**(), **strerror**()
lcm()	*I*= **lcm**(*I1*,*I2*)	**lcm**(*I*,*I1*,*I2*)
length	*i*=**length**(*A*)	*i*=**max**(**shape**(*A*))
linspace	*x*=**linspace**(*first*,*last*,*n*)	**lindata**(*first*,*last*, *x*)
loglog	*x*=**loglog**(*x*,*y*)	plot.loglog(*x*,*y*)
loglog	*x*=**loglog**(*x*,*y*)	plot.data2D(*x*,*y*) plot.scaleType(PLOT_AXIS_X, PLOT_SCALETYPE_LOG) plot.scaleType(PLOT_AXIS_Y, PLOT_SCALETYPE_LOG)
logspace	*x*=**logspace**(*first*,*last*,*n*)	**logdata**(*first*,*last*,*x*)
log	*x*=**log**(*x*)	*x*=**log**(*x*)
log10	*x*=**log10**(*x*)	*x*=**log10**(*x*)
log2	*x*=**log2**(*x*)	*x*=**log**(*x*)/**log**(*2*)
	r=**log2**(*r*)	*r*=**log2**(*r*)
logm	*A*=**logm**(*A1*)	**logm**(*A*,*A1*)
lower	*str*=**lower**('*str*')	see **tolower**()
lscov	*R*= **lscov**(*A1*,*R1*,*A2*)	**llsqcovsolve**(*R*, *A1*,*R1*,*A2*)
	[*R*,*R2*]=**lscov**(*A1*,*R1*,*A2*)	**llsqcovsolve**(*R*, *A1*,*R1*,*A2*,*R2*)
lu	[*R1*,*R2*]=**lu**(*A*)	**ludecomp**(*A*,*R1*,*R2*)
	[*R1*,*R2*,*I*]=**lu**(*A*)	**ludecomp**(*A*,*R1*,*R2*,*I*)
magic()	*A*=**magic**(*i1*)	**specialmatrix**("*Magic*",*i1*)
	Zv= **mean**(*Z*)	**cmean**(*Z*, *Zv*)
max	*r*=**max**(*s1*,*s2*)	*r*=**max**(*r1*,*r2*)
	r=**max**(*Av*)	*r*=**max**(*R*)
	Rv=**max**(*A*)	*Rv*=**maxv**(*R*) *Rv*=**transpose**(**maxv**(**transpose**(*R*))

Table E.3: Comparison of functions in MATLAB and Ch (continued).

Function	MATLAB	Ch
min	*r*=**min**(*s1*,*s2*)	*r*=**min**(*r1*,*r2*)
	r=**min**(*Av*)	*r*=**min**(*R*)
	Rv=**min**(*A*)	*Rv*=**minv**(*R*)
		Rv=**transpose**(**minv**(**transpose**(*R*))
mean	*r*=**mean**(*R*)	*r*=**mean**(*R*)
	Rv=**mean**(*R*)	*r*=**mean**(*R*, *Rv*)
		mean(**transpose**(*R*), *Rv*)
	z= **mean**(*Z*)	*z*= **cmean**(*Z*)
median	*r*=**median**(*R*)	*r*=**median**(*R*)
	Rv=**median**(*R*)	*r*=**median**(*R*, *Rv*)
		median(**transpose**(*R*), *Rv*)
mod	*i*=**mod**(*i1*,*i2*)	*i*=*i1*%*i2*
	r=**mod**(*r1*,*r2*)	*r*=**fmod**(*r1*,*r2*)
NaN	**NaN**	**NaN**
nargin	**nargin**	(not valid)
nargout	**nargout**	(not valid)
nextpow2	*i*=**nextpow2**(*r*)	*i*=**ceil**(**log2**(*r*))
	i=**nextpow2**(*z*)	*i*=**ceil**(**log2**(**abs**(*z*)))
	i=**nextpow2**(*Av*)	*i*=**ceil**(**log2**((**int**)**shape**(*Av*)))
nnls()	*R*=**nnls**(*A1*,*R1*)	**llsqnonnegsolve**(*R*, *A1*,*R1*)
	R=**nnls**(*A1*,*R1*,*r*)	**llsqnonnegsolve**(*R*, *A1*,*R1*,*R*)
	[*R*, *R2*]=**nnls**(*A1*,*R1*)	**llsqnonnegsolve**(*R*, *A1*,*R1*,0.0, *R2*)
	[*R*, *R2*]=**nnls**(*A1*,*R1*,*r*)	**llsqnonnegsolve**(*R*, *A1*,*R1*,*r*, *R2*)
norm	**norm**(*A*)	**norm**(*A*,"*2*")
	norm(*A*,*1*)	**norm**(*A*,"*1*")
	norm(*A*,*2*)	**norm**(*A*,"*2*")
	norm(*A*,*inf*)	**norm**(*A*,"*i*")
	norm(*A*,'*fro*')	**norm**(*A*,"*f*")
	norm(*Av*)	**norm**(*Av*,"*2*")
	norm(*Av*,*inf*)	**norm**(*Av*,"*i*")
	norm(*Av*,-*inf*)	**norm**(*Av*,"-*i*")
	norm(*Av*,p)	**norm**(*Av*,"*p*")
numtwostr	*str*=**numtwostr**(*s*)	**sprintf**(*str*,"*%*", *s*)
null	*A*=**null**(*Ai*)	**nullspace**(*A*, *Ai*)
ones	**ones**(*i1*)	(**array int a**[*i1*][*i1*])*1*
	ones(*i1*,*i2*)	(**array int a**[i*i1*][*i2*])*1*
	ones(**size**(*A*))	**array int** dim[*2*]=**shape**(*A*);
		(**array int a**[dim[*0*]][dim[*1*]])*1*
ode23	[*Rv1*,*Rv2*]= **ode23**('*fun*',*r1*,*r2*,*Rv*)	**oderungekutta**(*Rv1*,*Rv2*, NULL, *fun*,*r1*,*r2*,*Rv*)
ode45	[*Rv1*,*Rv2*]= **ode45**('*fun*',*r1*,*r2*,*Rv*)	**oderungekutta**(*Rv1*,*Rv2*, NULL, *fun*,*r1*,*r2*,*Rv*)
orth	*A*= **orth**(*Ai*)	**orthonormalbase**(*A*, *Ai*)
pascal()	*A*=**pascal**(*i1*)	**specialmatrix**("*Pascal*",*i1*)
	A=**pascal**(*i1*,*i2*)	**specialmatrix**("*Pascal*",*i1*,*i2*)

Table E.3: Comparison of functions in MATLAB and Ch (continued).

Function	MATLAB	Ch
pi	**pi**	**#include**<**math.h**> **M_PI**
pinv	*A1*=**pinv**(*A*)	*R1*=**pinverse**(*R*)
poly	*Av*=**poly**(*A*)	**charpolycoef**(*Av*,*A*)
	Bv=**poly**(*Av*)	**polycoef**(*Bv*,*Av*)
polyder	*Av*= **polyder**(*Bv*)	**polyder**(*Av*, *Bv*)
polyder2	*Av*= **polyder**(*Av1*, *Bv1*)	**polyder2**(*Av*, NULL, *Av1*, *Bv1*)
	[*Av*,*Bv*] = **polyder**(*Av1*,*Bv1*)	**polyder2**(*Av*,*Bv*,*Av1*,*Bv1*)
polyfit	*Rv*=**polyfit**(*Rv1*, *Rv2*, *i*)	**polyfit**(*Rv*, *Rv1*, *Rv2*)
polyval	*r*= **polyval**(*Rv1*, *r1*)	*r*= **polyeval**(*Rv1*, *r1*)
	r= **polyval**(*Rv1*,*r1*,*Rv*)	*r*= **polyeval**(*Rv1*, *r1*, *Rv*)
	z= **polyval**(*Av1*, *s*)	*z*= **cpolyeval**(*Av1*, *s*)
	z= **polyval**(*Av1*, *s*, *Av*)	z = **cpolyeval**(*Av1*, *s*, *Av*)
	Av= **polyval**(*Bv*, *Av1*)	**polyevalarray**(*Av*, *Bv*, *Av1*)
	A=**polyval**(*Av*,*A1*)	**polyevalm**(*A*,*Av*,*A1*)
polyvalm()	*A*=**polyvalm**(*Av*,*A1*)	**polyevalm**(*A*,*Av*,*A1*)
plot	**plot**(Rva,Rvb)	**plotxy**(*Rv1*,*Rv2*)
		plotxyf(*file*)
		CPlot:MemberFunctions()
plot3	**plot**3(*Rv1*,*Rv2*,*Rv*3)	**plotxyz**(*Rv1*,*Rv2*,*Rv*3)
		plotxyzf(*file*)
		CPlot:MemberFunctions()
prod	*s*=**prod**(*A*)	*r*=**product**(*R*)
		z=**cproduct**(Z)
	Av=**prod**(*A*)	*r*=**product**(*R*, *Rv*)
	Av=**prod**(*A*, 1)	*r*=**product**(*R*, *Rv*)
	Av=**prod**(*A*, 2)	*r*=**product**(**transpose**(*R*), *Rv*)
		z=**cproduct**(*Z*,*Zv*)
		product(**transpose**(*R*), *Rv*)
		cproduct(**transpose**(Z), *Zv*)
quad	*r*=**quad**('*fun*',*r1*,*r2*)	*r*=**integral1**(*fun*,*r1*,*r2*)
quadeight	*r*=**quadeight**('*fun*',*r1*,*r2*)	*r*=**integral1**(*fun*,*r1*,*r2*)
qr	[*A1*, *A2*] = **qr**(*A*)	**qrdecomp**(*A*, *A1*, *A2*)
	[*A1*, *A2*, *A3*] = **qr**(*A*)	
	[*A1*, *A2*]=**qr**(*A*)	**qrdecomp**(*A*,*A1*,*A2*)
	[*A1*,*A2*]=**qr**(*A*,**zero**)	**qrdecomp**(*A*,*A1*,*A2*)
qrdelete	[*A1*, **itBa**] = **qrdelete**(*A*, *B*, *i2*)	**qrdelete**(*A1*, **itBa**, *A*, *B*, *i2*)
qrinsert	[*A1*, **itBa**] = **qrdelete**(*A*, *B*, *i2*, *Av*)	**qrdelete**(*A1*, **itBa**, *A*, *B*, *i2*, *Av*)
rank	*i*= **rank**(*A*)	*i*= **rank**(*A*)
	i= **rank**(*A*, *r*)	*i*= **rank**(*A*)
real	*r*=**real**(*s*)	*r*=**real**(*s*)
	R=**real**(*A*)	*R*=**real**(*A*)
realmax	**realmax**	**#include**<**float.h**> FLT_MAX, DBL_MAX
realmin	**realmin**	**#include**<**float.h**> FLT_MIN, DBL_MIN

Table E.3: Comparison of functions in MATLAB and Ch (continued).

Function	MATLAB	Ch
rem	*r*=**rem**(*r1*,*r2*)	*r*=**fmod**(*r1*,*r2*)
		r=**remainder**(*r1*,*r2*)+*r2*
residue	[*Av*,*Bv*,Rv]=**residue**(*Rv1*,*Rv2*)	**residue**(*Rv1*, *Rv2*, *Av*, *Bv*, *Rv*)
round	*i*=**round**(*r*)	*i*=**round**(*r*)
roots	*Av*=**roots**(*Bv*)	**roots**(*Av*,*Bv*)
rosser	*A*=**rosser**()	**specialmatrix**(”*Rosser*”)
rot90(*A*)		
	A= **rot90**(*A1*)	**rot90**(*A*,*A1*)
	A= **rot90**(*A1*,*i*)	**rot90**(*A*,*A1*,*i*)
rand	*r*=**rand**()	*r*=**urand**()
randn	*R*=**rand**(*i1*, *i2*)	**urand**(*R*)
rcond	*r*= **rcond**(*A*)	*r*= **rcondnum**(*A*)
reshape	**reshape**(*A*, *m*, *n*)	(**array** type [m][n])*A*
rsf2csf	[*A1*, **itBa**] = **rsf2csf**(*A*, *B*)	**rsf2csf**(*A1*, **itBa**, *A*, *B*)
schur	*A1*= **schur**(*A*)	**schurdecomp**(*A*, *A1*, *NULL*)
	[*A1*, *A2*] = **schur**(*A*)	**schurdecomp**(*A*, *A1*, *A2*)
semilogx	*x*=**semilogx**(*x*,*y*)	plot.semilogx(*x*,*y*)
semilogx	*x*=**semilogx**(*x*,*y*)	plot.data2D(*x*,*y*)
		plot.scaleType(PLOT_AXIS_X, PLOT_SCALETYPE_LOG)
semilogy	*x*=**semilogy**(*x*,*y*)	plot.semilogy(*x*,*y*)
semilogy	*x*=**semilogy**(*x*,*y*)	plot.data2D(*x*,*y*)
		plot.scaleType(PLOT_AXIS_Y, PLOT_SCALETYPE_LOG)
sign	*i*=**sign**(*r*)	*i*=**sign**(*r*)
	z=**sign**(*z*)	*z*=*z*/**abs**(*z*)
sqrt	*x*=**sqrt**(*x*)	*x*=**sqrt**(*x*)
sqrtm	*A*= **sqrtm**(*A1*)	**sqrtm**(*A*,*A1*)
size	*Iv*=**size**(*A*)	*Iv*=**shape**(*A*)
	[*i1*,*i2*]=**size**(*A*)	*Iv*=**shape**(*A*); *i1*=*Iv*[0]; *i2*=*Iv*[*1*]
	i=**size**(*A*,*1*)	*i*= (**int**)**shape**(*A*)
	i=**size**(*A*,*2*)	*Iv*=**shape**(*A*); *i*=*Iv*[*1*]
sort	*Av*=**sort**(*Av1*)	**sort**(*Av*, *Av1*)
		sort(*Av*, *Av1*, ”array”)
	A=**sort**(*A1*)	**sort**(*A*, *A1*, ”column”)
	A=**sortrows**(*A1*)	**sort**(*A*, *A1*, ”row”)
	[*Av*, I]=**sort**(*Av1*)	**sort**(*Av*, *Av1*, ”array”, I)
		sort(*Av*, *Av1*, NULL, I)
	[*A*, *I*]=**sort**(*A1*)	**sort**(*A*, *A1*, ”column”, I)
	[*A*, *I*]=**sortrows**(*A1*)	**sort**(*A*, *A1*, ”rows”, I)
spline	*Ri*= **spline**(*R1*, *R2*, *Ri1*)	**interp1**(*Ri*, *R1i*, *R2i*, ”*spline*”)
	ri= **spline**(*R1*, *R2*, *r*)	**CSpline**::**Interp**(r)
	Ri= **spline**(*R1*, *R2*, *Ri1*)	**CSpline**::**Interpm**(*Ri1*, *Ri*)
sprintf	**sprintf**()	see **sprintf**()
sscanf	**sscanf**()	see **sscanf**()
std	*r*=**std**(*R*)	*r*=**std**(*R*)
	Rv=**std**(*R*)	*r*=**std**(R, Rv)
		std(**transpose**(*R*), *Rv*)

Table E.3: Comparison of functions in MATLAB and Ch (continued).

Function	MATLAB	Ch
str2mat	C=**str2mat**('*str1*', '*str2*', ...)	**str2mat**(*C*, *str1*, *str2*)
str2num	*i*=**str2num**('*str*')	*i*=**atol**(*str*), see **strtod**()
strcmp	**strcmp**('*str1*','*str2*')	!**strcmp**("*str1*","*str2*")
strrep	*str*= **strrep**('*str1*', '*str2*', '*str3*')	*str*=**strrep**(*str1*, *str2*, *str3*)
strtok	**strtok**	see **strtok**(), **strtok_r**()
subplot	**subplot**()	**CPlot**::**subplot**()
		CPlot::**getSubplot**()
sum	*r*=**sum**(*A*)	*r*=**sum**(*R*)
		z=**csum**(*Z*)
	Av=**sum**(*A*)	*r*=**sum**(*R*, *Rv*)
		z=**csum**(*Z*,*Zv*)
		sum(**transpose**(*R*), *Rv*)
		csum(**transpose**(*Z*), *Zv*)
svd	*R*= **svd**(*A*)	**svd**(*A*, *R*, *NULL*, *NULL*)
	[*A1*, *R*, *A2*] = **svd**(*A*))	**svd**(*A*, *R*, *A1*, *A2*)
tan	*x*=**tan**(*x*)	*x*=**tan**(*x*)
tanh	*x*=**tanh**(*x*)	*x*=**tanh**(*x*)
toeplititz	*A*=**toeplititz**(*Av1*)	**specialmatrix**("*Toeplititz*",*Av1*)
	A=**toeplititz**(*Av1*,*Av2*)	**specialmatrix**("*Toeplititz*",*Av1*,*Av2*)
trace	*s*=**trace**(*A*)	*r*=**trace**(*R*)
		z=**ctrace**(Z)
trapz	*r*=**trapz**(Rva,Rvb)	see **integral1**()
tril	*A1*=**tril**(*A*)	*A1*=**triangularmatrix**("*lower*",*A*)
	Z1=**tril**(Z)	*Z1*=**ctriangularmatrix**("*lower*",*Z*)
	A1=**tril**(*A*,i)	*A1*=**triangularmatrix**("*lower*",*A*,*i*)
	Z1=**tril**(Z,i)	*Z1*=**ctriangularmatrix**("*lower*",*Z*,*i*)
triu	*A1*=**triu**(*A*)	*A1*=**triangularmatrix**("*upper*",*A*)
	Z1=**triu**(Z)	*Z1*=**ctriangularmatrix**("*upper*",*Z*)
	A1=**triu**(*A*,i)	*A1*=**triangularmatrix**("*upper*",*A*,*i*)
	Z1=**triu**(*Z*,*i*)	*Z1*=**ctriangularmatrix**("*upper*",*Z*,*i*)
vander	*R*=**vander**(*Rv*)	*R*=**vandermatrix**(*Rv*)
	A=**vander**(*Av*)	**specialmatrix**("*Vandermonde*", *Av*)
unwrap()	*A*=**unwrap**(*A1*)	**unwrap**(*A*,*A1*)
	A=**unwrap**(*A1*,r)	**unwrap**(*A*,*A1*,*r*)
upper	*str*=**upper**('str')	see **toupper**()
who	**who**	**showvar**
xcorr	*Av*= **xcorr**(*Av1*, *Av2*)	**xcorr**(*Av*, *Av1*, *Av2*)
xor	**xor**(*A*,*B*)	*A*^^ *B*
	xor(*A*,*s*)	*A*^^ *r*
	xor(s,*A*)	*r*^^ *A*
zeros	**zeros**(*i1*)	(**array int**[*i1*][*i1*]) *0*
	zeros(*i1*,*i2*)	(**array int**[*i1*][*i2*]) *0*
	zeros(**size**(*A*))	**array int** dim[*2*]=shape(*A*);
		(**array int** [dim[*0*]][dim[*1*]]) *0*
2D/3D Plotting functions	see class **CPlot**	

E.3 Control Flow

Table E.4: Control flow comparisons between MATLAB and Ch.

Description	MATLAB	Ch shell

for-loop	`for i=n1:n2:n3`
	`  commands`
	`end`
		`for (i=n1; i<=n3; i+=n2) {`
		`  commands`
		`}`

	`for i=n1:n3`
	`  commands`
	`end`
		`for(i=n1; i<=n3; i++) {`
		`  commands`
		`}`

while-loop	`while expr`
	`  commands`
	`end`
		`while(expr) {`
		`  commands`
		`}`

if	`if expr`
	`  commands`
	`end`
		`if (expr) {`
		`  commands`
		`}`

if-else	`if expr1`
	`  commands1`
	`else`
	`  commands2`
	`end`
		`if (expr1) {`
		`  commands1`
		`}`
		`else {`
		`  commands2`
		`}`

if-else if-else	`if expr1`
	`  commands1`
	`elseif expr2`
	`  commands2`
	`else`
	`  commands3`
	`end`
		`if (expr1) {`
		`  commands1`
		`}`
		`else if(expr2) {`
		`  commands2`
		`}`
		`else {`
		`  commands3`
		`}`

Appendix F

Comparison with Fortran

Many features in Ch are added to bridge the gap between C and FORTRAN for scientific numerical computing. References, complex numbers, generic functions, and variable length arrays in Ch are similar to those in FORTRAN. For example, linguistic features of references in Ch are closely related to equivalence statements, subroutines, and functions in FORTRAN. Users with prior FORTRAN experience can easily adapt to the Ch programming paradigm. This appendix discusses some issues related to port FORTRAN code to Ch.

F.1 Reference in Ch versus Equivalence in FORTRAN

The linguistic features of references in Ch are closely related to equivalence statements. The following equivalence statements in FORTRAN

```
real f1, f2
equivalence (f1, f2), (i, j, k)
```

can be achieved in Ch as follows

```
float f1, &f2 = f1
int i, &j = i, &k = i
```

where float variables `f1` and `f2` share the same memory space whereas there int variables `i, j, and k` share the same space. The equivalence of two arrays in FORTRAN can be achieved by pointers in Ch. For example, the FORTRAN code

```
dimension A(10), B(10), C(20, 20), D(20, 20)
equivalence (A[1], B[1]), (C, D)
```

can be ported to Ch as follows

```
float A[10], *B, C[20,20], (*D)[20]
B = A; D = C;
```

where `B` is a pointer to float and `D` is a pointer to array of 20 floats. Porting the equivalence of a single variable and an element of an array in FORTRAN to Ch is more involving. For example, the FORTRAN code

```
int a[2][3], b[4][5];
a[1][2] = 5;
void funct(int (*c)[3], d[:][:])
{
  d[2][3] = c[1][2];
}
funct(a,b);
printf("b[2][3] = %d \n", b[2][3]); // output: b[2][3] = 5
```

Program F.1: Passing arrays to functions by reference in Ch

```
dimension A(10)
equivalence (A[3], f)
A[3] = 5;
```

may be ported to Ch as follows

```
float A[10], *f
f = &A[2]
A[2] = 5           // *f = 5
```

Although it is allowed to place variables of different data types in an equivalence statement in FORTRAN, there is no provision for consistent handling of this kind of equivalence in the FORTRAN 77 standard.

F.2 Call-by-Reference in Ch and in FORTRAN

Functions in Ch can be called either by value or by reference. Many restrictions in FORTRAN 77 about call-by-reference are relaxed in Ch. For example, FORTRAN does not allow a subroutine to call itself or a function to use itself inside the function. In other words, recursive subroutine calls are not permitted in FORTRAN, let alone nested and recursively nested subroutine calls. In FORTRAN, the data types of actual arguments of a function should be the same as those of the formal arguments of the function. In Ch, the data types of actual arguments of functions can be different from those of the corresponding formal arguments of references in functions. In FORTRAN, when an argument of a subroutine is used as an lvalue inside a subroutine, the actual argument in the calling subroutine must be a variable. A reference variable in Ch can be used as an lvalue inside a function even if the actual argument is not an lvalue, whereas FORTRAN cannot.

Note that when arrays are passed to a function through its arguments, the memory space for arrays in the calling function will be used in the called functions. In other words, arrays in Ch are passed by reference as shown in Program F.1 where the variables `c` and `d`, in the arguments of the function `funct()` are the pointers to array of 3 elements of int and assumed-shape array, respectively.

Since Ch encapsulates all the programming capabilities of FORTRAN 77, porting subroutines and functions in FORTRAN to functions in Ch is not very difficult. For example, the FORTRAN program given in Program F.2 can be ported, without changing the functionality of the program, to a Ch program shown in Program F.3. In Program F.3, the shape 5x6 for the complex array `A` inside the function `FUNCT()` is assumed from its actual argument of the complex array `ZA`. Integral values 5 and 6 are passed to arguments of references `M` and `N`. The complex variable `Z` is passed to the function by reference. Block-structured nested do-loops are used inside the function. Like many other mathematical functions, the function `sin()` in Ch

```
      SUBROUTINE FUNCT(A, M, N, R)
        INTEGER M, N
        COMPLEX A(M, N), R
        INTEGER I, J
        DO 10 I = 1, M
          DO 10 J = 1, N
            A(I,J) = R*R + 3
10          A(I,J) = SIN(A(I,J)/R)
        R = 50
      END

      COMPLEX ZA(5,6), Z
      Z = CMPLX(1,2)
      CALL FUNCT(ZA, 5, 6, Z)
      STOP
      END
```

Program F.2: A FORTRAN program

is a polymorphic function that calculates the result according to the data type of the input argument. In this example, a complex value will be produced by the function `sin()`.

These new features in Ch are intended to bridge the gap between C and FORTRAN so that users with prior FORTRAN experience can readily write Ch programs. Experience indicates that users with prior FORTRAN experience can easily write Ch programs to solve complicated practical engineering problems. It should be pointed out that many programming features such as array syntax and assumed-shape arrays of Fortran 90 have been incorporated into Ch. Fortran 90 is significantly more complicated than FORTRAN 77. Therefore, porting Fortran 90 to Ch is not one of the design goals for Ch.

```
void funct(complex a[:][:], int &m, &n, complex &r) {
  int i, j;
  for(i=0; i<m; i++)
    for(j=0; j<n; j++) {
      a[i][j] = r*r + 3;
      a[i][j] = sin(a[i][j]/r);
    }
  r = 50;
}

complex za[5][6], z;
z = complex(1,2);
funct(za, 5, 6, z);
```

Program F.3: A Ch program ported from a FORTRAN program

Appendix G

Summary of Commonly Used Portable Shell Commands in Ch

This appendix contains a summary of commonly used commands portable across different platforms in Ch under both Unix and Windows. The commands are listed below by category. These commands are very useful for portable shell programming in Ch.

G.1 File Systems

cd	change working directory
chgrp	change the group of a file or library
chmod	change the access permission of a file
chown	change the owner of a file or library
cp	copy files or directories
dd	use to block input/output operations on files and performing conversions on them
df	(”disk free”) reports available space and space in use in drives accessible by your system
du	report the amount of disk space used by a hierarchy of directories
find	search for files or directories that match a certain criterion, i.e. name, modification time etc
ln	make a hard link or symbolic link
ls	list contents of directory
mkdir	create a new directory
mv	rename files or move them to other directory
pwd	print name of working directory
rm	remove files or directories
rmdir	remove directories
touch	create a new empty file or update the modification time of an existing file
which	show the full path of commands, functions, and header files

G.2 Binary Files

ar	archive maintenance utility; create libraries to be linked to other programs
nm	list the symbol in an executable, object file or libary
ranlib	generate an index for an ar format archive; enables faster linking
size	print the sizes of sections in object files and executable programs
strings	print the ASCII text strings embedded in binary file
strip	remove debugging symbols and line number information from files to make file smaller

G.3 Text Files

awk	scan one or more files and perform an action on all of the lines that match a particular conditon
cat	concatenate files together and send the result to the standard output
cksum	calculate a CRC checksum for files
csplit	context based file splitter
cut	cut out selected fields of each line of a file
egrep	search a file for a pattern using full regular expressions
expand	replace the tabs in a file with a specified number of spaces
fgrep	search a file for a fixed-character string
fmt	simple text formatters
fold	wrap files to fit in a specified number of columns
grep	include also grep,egrep and fgrep; search for patterns matching on lines in a file
head	print a specified number of lines from the beginning of a file
indent	formatter for C source code
join	conditionally merge two files together based on matching fields in the files
less	a display paginator similar to more and pg in UNIX
make	update a file based on a series of dependency rules stored in a special format
md5sum	calculate a ”fingerprint” for files using the MD5 algorithm
more	copy by the screenful
nl	add line number to files
od	dump the contents of a file in a number format, including octal,hex and several ASCII format
patch	apply changes to files
paste	merge multiple files together into one file by concatenating lines from each file to form one line in the output file
pr	print files
sed	”stream editor” based on the line editor ed; useful for editing streams of data
sort	sort lines in a file based on fields in the file
split	cut files into multiple fixed size pieces
sum	calculate a simple checksum for files
tail	print a specified number of line from the end of a file
tr	character translation
tsort	topological sort
troff	typeset or format documents
unexpand	”unexpand” sequences of a specified number of spaces into tab characters
uniq	remove duplicate lines from a sorted file
vi	screen text editor
wc	count characters, words and/or lines in a file

G.4 Comparing Files

cmp	compare two files and display the line and byte number differences
comm	report common lines in two files
diff	compare files, showing lines that differ in any of serveral formats
diff3	compare three files
sdiff	compare two files side by side and interactively merges them into a third file

G.5 Shell Utilities

basename	echo its argument, a file name, minus any directory name
date	display date and time of the system
dirname	print the directory component of file name
echo	print its argument
env	used to modify the environment in which a program is run
expr	evaluate expressions involving arithmetic, relational, logical and string operators
factor	get all prime factors for a postive integer
hostname	the name of the local host machine
id	display the user and group IDs and names
logname	display the user's login name
pathchk	check path names
printenv	ouput the environment under which a command will execute
sh	the Bourne shell
sleep	stop all actions in a shell for a specified period of time
tee	copy its standard input to files named on the command line and also to standard output
test	check file types and compare values
uname	print name of current system
whoami	display the login name
xargs	construct argument lists and invoke utility

G.6 Archiving Files

cpio	file archiving and backup utility
gzip	file compression program; does both compressing and uncompressing
gunzip	same as gzip
tar	create and manipulate tar archives

Appendix H

Summary of vi Text Editor

This appendix contains a summary of commonly used features of `vi` text editor.

Enter Input Mode

i	before cursor
a	after cursor
I	at start of line
A	at end of line
o	open line above
O	open line below

Move Cursor

l	one space right
h	one space left
j	one line down
k	one line up
$	end of line
^	start of line
0	start of line
w	next word
e	end of word
*n*G	line *n*
G	end of file
H	top of screen
M	middle of screen
L	bottom of screen

Delete

dw	delete words
dd	delete line
dG	delete all lines following the line
d1G	delete all lines before the line
*n*dd	delete *n* lines
D	to end of line
x	delete char at cursor
*n*x	delete *n* chars

Change

cw	change word
cc	change line
C	to end of line
r	replace char at cursor
s	substitute chars at cursor
S	substitute line, same as cc

Screen Control

CTRL-d	scroll forward
CTRL-u	scroll back
CTRL-f	next screen
CTRL-b	previous screen

Copy and Paste

Y	yank line
yy	yank line
*n*yy	yank *n* lines
p	put below
P	put above
"*x*y	yank line into buffer *x*
"*xn*yy	yank *n* lines into buffer *x*
"*x*p	put from buffer *x* below
"*x*P	put from buffer *x* above
"*x*d	delete into buffer x

Other Functions

J	join current and next lines
u	undo
/	search forward
?	search backward
n	next occurrence
N	previous occurrence
.	repeat last action
ZZ	write and quit
ESC	cancel command

Last Line Mode

:w	write
:q	quit
:wq	write and quit
:n	next file
:r *file*	read *file*
:e *file*	edit *file*
:f	file name
:! *command*	execute shell *command*
:*n*	move cursor to line *n*
:set *option*	change options
:set number	show line numbers
:set nonumber	no line number
:[address]s/*old*/*new*/[g]	substitute *old* by *new*
address .	current line
$	last line
%	entire file
g	each occurrence in a line

When in Input Mode

BACKSPACE	delete char
CTRL-w	delete word
ESC	command mode

Appendix I

Porting Code to the Latest Version

I.1 Porting Code to Ch Version 6.0.0.13581

1. Changed the Boolean type ”bool’ from int to unsigned char in stdbool.h.
2. Change the default `complex` type from `float complex` to `double complex`.

I.2 Porting Code to Ch Version 6.0.0.13581

1. Changed the Boolean type ”bool’ from char to int in stdbool.h.
2. Changed

```
CPlot::func2D(double (*func)(double x, void *param),
              void *param, double x0, double xf, int n);
CPlot::func3D(double (*func)(double x, double y, void *param),
              void *param, double x0, double xf, double y0, double yf,
              int nx, int ny);
```

   to

```
CPlot::funcp2D(double x0, double xf, int n,
               double (*func)(double x, void *param), void *param);
CPlot::funcp3D(double x0, double xf, double y0, double  yf,
               int nx, int ny,
               double (*func)(double x, double y, void *param),
               void *param);
```

3. Changed

```
CPlot::origin(double x, double y);
```

   to

```
CPlot::boundingBoxOrigin(double x, double y);
```

4. Changed

```
CPlot::grid(int flag, .../* int type */);
```

to

```
CPlot::grid(int flag, .../* char *option */);
```

Removed

```
PLOT_GRID_POLAR
PLOT_GRID_RECTANGULAR
```

Change

```
plot.grid(PLOT_ON, PLOT_GRID_POLAR);
plot.polarPlot(PLOT_ANGLE_DEG);
```

to

```
plot.grid(PLOT_ON);
 or plot.grid(PLOT_ON, "polar");
 or lot.grid(PLOT_ON, "polar 30"); // the interval of radials is 30 degrees
plot.polarPlot(PLOT_ANGLE_DEG);
```

5. Changed

```
CPlot::arrow(double x_head, double y_head, double z_head,
             double x_tail, double y_tail, double z_tail, ...
             /* [int linetype, int linewidth] */ );
```

to

```
CPlot::arrow(double x_head, double y_head, double z_head,
             double x_tail, double y_tail, double z_tail, ...
             /* [string_t option] */ );
```

Change

```
plot.arrow(x1, y1, z1, x2, y2, z2, 1, 3);
```

to

```
char option[64];
sprintf(option, "linetype 1 linewidth 3");
plot.arrow(x1, y1, z1, x2, y2, z2, option);
```

6. `CPlot::axisRange(int axis, double minx, double max, double incr);`

is obsolete. Use

```
CPlot::axisRange(int axis, double minx, double max);
CPlot::ticsRange(int axis, incr);
```

Index

\" double-quote-character escape sequence, 115
\' signal-quote-character escape sequence, 115
\?, 115
\\ backslash-character escape sequence, 115
\a alert escape sequence, 115
\b escape sequence, 115
\f form-feed escape sequence, 115
\n (newline escape sequence), 115
\r carriage-return escape sequence, 115
\t horizontal-tab escape sequence, 115
\v escape sequence, 115
|| logical inclusive OR operator, 122, 128, 293
| bitwise inclusive OR, 122, 128
| pipe, 73
|= bitwise inclusive OR assignment operator, 122, 128
( ), 122
* indirection operator, 147
* multiplication operator, 122, 123, 226, 227, 289
* pointer indirection operator, 122
* wild characters, 68
*= multiplication assignment operator, 122, 128, 291
+ addition operator, 122, 226, 227
+ sign, 122, 123
+ unary plus, 289
++ increment operator, 122, 134, 292
+= addition assignment operator, 122, 128, 291
, comma operator, 122, 132
- sign, 122, 123, 226, 227
- subtraction operator, 122, 123, 226, 227
- unary minus, 289
-= subtraction assignment operator, 122, 128, 291
. command, 49, 53
. structure member operator, 45, 122, 135, 339
.* array multiplication, 289
.* array multiplication operator, 122
.* array multiplicaton operator, 125
../ parent directory, 68
./ array division, 289
./ array division operator, 122, 125
./ current working directory, 68
.chlogin, 28, 559
.chlogout, 28, 559
.chrc, 28, 33, 36, 407, 413, 559
.chslogin, 28
.chsrc, 28
.cshrc, 559
.login, 559
.logout, 559
/ division operator, 122, 123, 226, 227
/= division assignment operator, 122, 291
:, 275, 278, 298
:: scope resolution operator, 122, 350, 553
= assignment operator, 122, 128, 291
== equal-to operator, 122, 125, 226, 227, 293
? wild character, 68
?: conditional operator, 122, 129, 295
[ ], 122
21
administrator prompt, 47
preprocessor operator, 87, 91
superuser prompt, 47
#!/bin/ch, 80
#!/bin/csh, 80
#!/bin/ksh, 80
#!/bin/sh, 80
preprocessor operator, 87, 91
#define preprocessor directive, 87–89
#defined preprocessor operator, 87
#elif preprocessor directive, 87, 88
#else preprocessor directive, 87
#endif preprocessor directive, 87
#error preprocessor directive, 87, 93
#if preprocessor directive, 87, 88
#ifdef preprocessor directive, 87
#ifndef preprocessor directive, 87
#include preprocessor directive, 87, 88
#line preprocessor directive, 87, 92
#pragma, 36, 87, 93
#undef preprocessor directive, 87, 88
$ Bourne, Korn, BASH shell prompt, 47
$ command name substitution, 66
$ expression substitution, 65, 386
$ variable substitution, 64, 385, 386
$argv, 558
% C shell prompt, 47
% modulus operator, 122
%= modulus assignment operator, 122
&, 303
& address operator, 122, 133, 147, 200, 295, 327
& bitwise AND, 122, 128

& commands in background, 74
&= bitwise AND assignment operator, 122, 128
&& logical AND operator, 122, 128, 293
^ quick substitution, 58
AIX, 97
CHDLL, 96
_CH_, 96
DARWIN, 97
FREEBSD, 97
GLOBALDEF, 96
HPUX, 97
_IOFBF, 370
_IOLBF, 370
_IONBF, 370
LINUXPPC, 97
LINUX, 97
_M64_, 96
QNX, 97
SCH, 96
SOLARIS, 97
WIN32, 97
_X86_, 97
__BIG_ENDIAN__, 97
__DATE__, 96
__FILE__, 96
__LINE__, 92, 96
__LITTLEENDIAN__, 97
__STDC_VERSION__, 96
__STDC__, 96
__TIME__, 96
__VA_ARGS__, 90, 542
__class__, 22, 24, 367
__class_func__, 22, 24, 367
__declspec, 170
__declspec(global), 45
__declspec(local), 45, 111, 170
__func__, 22, 24, 367, 542
__i386__, 97
__ppc__, 97
_argc, 22, 24, 38, 83, 558
_argv, 22, 24, 38, 83, 558
_chlogin, 28
_chlogout, 28
_chrc, 28, 33, 36
_chslogin, 28
_chsrc, 28
_cwd, 21, 22, 24, 401
_cwdn, 21, 22, 24, 401
_environ, 22, 24
_errno, 22, 24
_execv(), 401
_execvp(), 401
_fopen(), 401
_fork(), 401
_formatd, 22, 24, 29
_formatf, 22, 24, 29
_fpath, 21, 22, 24, 36, 95, 401, 404, 545
_fpathext, 22, 24, 34
_fstat(), 401
_histnum, 22, 24
_histsize, 22, 24, 559
_home, 21, 22, 24, 401
_host, 22, 24, 401
_iath, 21
_ignoreeof, 22, 24, 30, 540
_ignoretrigraph, 22, 24
_ipath, 22, 24, 88, 95, 401, 404, 411
_lang, 21, 22, 24
_lc_all, 21, 22, 24
_lc_collate, 21, 22, 24
_lc_ctype, 21, 22, 24
_lc_monetary, 21, 22, 24
_lc_numeric, 21, 22, 24
_lc_time, 21, 22, 24
_logname, 21, 22, 24
_lpath, 21, 22, 24, 95, 401, 404
_lstat(), 401
_new_handler, 22, 24, 347
_path, 21, 23, 25, 33, 49, 95, 401, 404, 545
_pathext, 23, 25
_pipe(), 401
_popen(), 401
_ppath, 23, 25, 401, 409
_prompt, 23, 25, 30, 47
_remove(), 401
_rename(), 401
_setlocale, 23, 25, 338
_shell, 21, 23, 25, 401
_socket(), 401
_socketpair(), 401
_stat(), 401
_status, 23, 25, 38, 558
_stop(), 43
_system(), 401
_term, 21, 23, 25
_tz, 21, 23, 25
_user, 21, 23, 25, 401
_utime(), 401
_warning, 23, 25, 29
! logical negation operator, 122, 128
! logical not operator, 293
!= inequality operator, 122, 125, 226, 227, 293
-- decrement operator, 122, 134, 292
-> structure pointer operator, 45, 122, 135, 339
2>&1 redirect standard output and error stream, 71
< less than operator, 122, 125, 293
< redirect standard input stream, 71
<< left shift, 122, 128

<< redirect standard input stream, 71
<< standard output stream operator, 382
<<= left shift assignment operator, 128
<<= right shift assignment operator, 122
<= less or equal operator, 122, 125, 293
> Ch prompt, 47
> greater than operator, 122, 125, 293
> redirect standard output stream, 71
>= greater or equal operator, 122, 125, 293
>> redirect standard output stream, 71
>> right shift, 122, 128
>> stream extraction operator, 382
>>= left shift assignment operator, 122
>>= right shift assignment operator, 128
^ bitwise exclusive OR operator, 122, 128
^= bitwise exclusive OR assignment operator, 128
^^ logical exclusive OR operator, 122, 128, 293, 544
~ bitwise one's complement, 122, 128
~ home directory, 68
‘ command substitution operator, 69, 122, 135
0B, 116
0X, 116
0b, 116
0x, 116

abs(), 20, 210, 229, 233, 314
accept(), 401
access(), 20, 82, 401
acos(), 20, 210, 229, 233
acosh(), 20, 210, 229, 233
aggregate floating-point types, 103
aio.h, 404
AIX, 97
alias, 31, 61, 62, 558
alias(), 20, 197
all(), 296
and, 128
any(), 296
Aquaterm, 413
argc, 38, 192
argument substitution, 62
argv, 38, 192, 558
array, 104, 105, 112, 249, 270, 282, 544
    assumed-shape array, 104, 105, 259
    computational array, 104, 112
    declaration of array, 249
    deferred-shape array, 104
    deferred-shape arrays, 251
    incomplete array, 105
    lower bounds, 270
    members of struct/union, 105
    range of subscript for an index, 105
    reference, 105
    subscript range, 271
    upper bounds, 270
array element order, 286
array elements, 284
array of assumed-shape, 264
array of reference, 303, 544
array operations, 289
    address operations, 295, 327
    arithmetic operations, 289
    assignment operations, 291
    cast operations, 296
    decrement operations, 292
    increment operations, 292
    logic operations, 293
    relational operations, 293
array reference, 284
array.h, 284, 404
arraycopy(), **181**
arrow(), 414, *see* CPlot, *see* CPlot
asin(), 20, 210, 229, 233, 315
asinh(), 20, 210, 229, 233, 315
assert.h, 404
assumed-shape array, 104, 105, 259, 298, 356, 544
atan(), 20, 210, 229, 233, 315
atan2(), 20, 210, 229, 315
atanh(), 20, 210, 229, 233, 315
atexit(), 20
auto, 19, 45
automatic storage duration, 45, 119, 136
autoScale(), 414, *see* CPlot
awk, 579
axes(), 414
axis(), 414, *see* CPlot
axisRange(), 414, *see* CPlot, *see* CPlot

background command, 74
balance(), 463
barSizd(), 414
basename, 580
BASH, 47, 80
beginparalleltask, 21
binary, 116
bit-field, 107, 340
bool, 543
border(), 414, *see* CPlot
borderOffsets(), 414, *see* CPlot
boundingBoxOrigin(), *see* CPlot
Bourne shell, 47, 80
boxBorder(), 414
boxWidth(), 414
break, 19, 143
buffered I/O, 369
built-in commands, 53

C arrays, 328

C shell, 47, 78, 80, 557, 558
C+, 1
call-by-reference, 156, 575
call-by-value, 156
callback function, 189
calloc(), 149, 239
carg(), 229
case, 19, 139
cat, 579
catch, 21
Cauchy, 512
ccompanionmatrix(), 463, **510**
cd, 53, 578
cdeterminant(), 463
cdiagonal(), 463, **506**
cdiagonalmatrix(), 463, **509**
ceil(), 20, 210, 229, 233, 315
cerr, 382, 553
cfevalarray(), 463, **480**
cfum(), **513**
cfunm(), 463
Ch, ii, 1
ch, 28, 78
Ch options, 31
Ch shell, 28, 47, 80
CH_CARRAYPTRTYPE, 180
CH_CARRAYTYPE, 180
CH_CARRAYVLA, 180
CH_CHARRAYPTRTYPE, 180
CH_CHARRAYTYPE, 180
CH_CHARRAYVLATYPE, 180
CH_UNDEFINETYPE, 180
changeViewAngle(), 414, *see* CPlot
char, 19, 99, 329
CHAR_MAX, 99
CHAR_MIN, 99
character constants, 113
character set, 18
characters, 329
charpolycoef(), 463, **496**
chdebug, 42, 54
chdir, 53
chdir(), 401
Chebyshev, 512
chgrp, 578
CHHOME, 21, 27
chlogin, 28
chmod, 33, 578
choldecomp(), 463, **517**
Cholesky decomposition, 517
Chow, 512
chown, 578
chown(), 401
chparse, 42, 51, 54, 401
chplot.h, 404
chrc, 338, 413
chroot(), 401
chrun, 42, 51, 54, 401
chs, 28, 78, 402
chs.exe, 402
chshell.h, 404
chslogin, 28
cin, 382, 553
cinverse(), 463, **523**
circle(), 414, *see* CPlot
Circul, 512
cksum, 579
class, 19, 51, 106, 269, 343, 543, 549
  classes inside member function, 362
  constructor, 345
  destructor, 345
  member function, 343
  nested class, 361
  polymorphism, 351
  private member, 345
  public member, 345
  scope resolution operator, 350
  static member, 347
  this, 351
Clement, 512
clinsolve(), 463
clock(), 20, 95
CLOCKS_PER_SEC, 95
closedir(), 394
closing a directory, 394
cls, 541
cmean(), 463, **473**
cmp, 580
colorBox(), 414
combination(), 463, **477**
comm, 580
comma operator, 122
command completion, 402
command files, 10, 32
command line options, 31
command mode, 8
command statement, 80
command substitution, 18, 69
command-line arguments, 192
commands in background, 74
comments, 26
communication, 189
companion matrix, 510
companionmatrix(), 463, **510, 511**
complex, 19, 103, 220, 221, 542
  complex functions, 229
  complex variables, 220
  constants, 220

I/O for complex numbers, 225
operations, 226
complex equation, 468
complex functions, 236
complex metanumbers, 221, 232
complex number, 220
complex numbers, 12, 118, 220
complex operations, 226
complex metanumbers, 227
regular complex numbers, 226
complex(), 229, 233
complex.h, 221, 404, 543
ComplexInf, 19, 233, 543, 549
ComplexNaN, 19, 233, 543, 549
complexsolve(), 463, **468**
compound statement, 136
computational array, 13, 104, 112, 282, 286, 328, 544
condition number, 505
conditional inclusion, 87
condnum(), 463
conj(), 20, 229, 233
const, 19, 112
const member functions, 553
constants, 113, 118
complex numbers, 118
pointers, 119
continue, 19, 143
contourLabel(), 414, *see* CPlot
contourLevels(), 414, *see* CPlot
contourMode(), 414, *see* CPlot
conv(), 463, **529**
conv2(), 463, **530**
converting tokens to strings, 91
preprocessor operator, 91
convolution, 529
coordSystem(), 414, *see* CPlot
copy, 541
copyright, i
copysign(), 125
corr2(), 463
corrcoef(), 463, **475**
correlation coefficients, 475
correlation(), **475**
cos(), 20, 210, 229, 233, 315
cosh(), 20, 210, 229, 233, 315
count(), 296
cout, 382, 553
covariance(), 463, **474**
cp, 578
cpio, 580
cpio.h, 404
CPlot
~CPlot, 414
arrow(), 414, **424**
autoScale(), 414
axes(), 414
axis(), 414, **423**
axisRange(), 414, **421**
barSize(), 414
border(), 414, **423**
borderOffsets(), 414
boxBorder(), 414
boxWidth(), 414
changeViewAngle(), 414
circle(), 414, **431**
colorBox(), 414
contourLabel(), 414
contourLevels(), 414
contourMode(), 414
coordSystem(), 414, **451**
CPlot(), 414
data(), 414
data2D(), **413**, 414
data2DCurve(), 414, **417**, **420**
data2DSurface(), **420**
data3D(), 414, **417**
data3DCurve(), 414
data3DSurface(), 414
dataFile(), 414, **420**
dataSetNum(), 414
deleteData(), 414
deletePlots(), 414
dimension(), 414, **421**
displayTime(), 414
enhanceText(), 414
fillStyle(), 414
func2D(), 414
func3D(), 414
funcp2D(), 414
funcp3D(), 414
getLabel(), 414
getOutputType(), 414
getSubplot(), 414, **432**
getTitle(), 414
grid(), 414
isUsed(), 414
label(), 414, **421**
legend(), 415, **424**
legendLocation(), 415, **424**
legendOption(), 415
line(), 415, **431**
lineType(), 415, **438**
margins(), 415
origin(), 415
outputType(), 415, **433**
plotting(), **413**, 415
plotType(), 415, **438**
point(), 415

pointType(), 415, **442**
polarPlot(), 415, **442**
polygon(), 415, **431**
rectangle(), 415, **431**
removeHiddenLine(), 415
scaleType(), 415
showMesh(), 415
size(), 415
size3D(), 415
sizeRatio(), 415, **446**
smooth(), 415
subplot(), 415, **432**
text(), 415, **424**
tics(), 415
ticsDay(), 415
ticsDirection(), 415
ticsFormat(), 415
ticsLabel(), 415
ticsLevel(), 415
ticsLocation(), 415
ticsMirror(), 415
ticsMonth(), 415
ticsPosition(), 415
ticsRange(), 415
title(), 415, **421**
cpolyeval(), 463, **490**
cproduct(), 461, 463, **472**
creat(), 401
createpkg.ch, 411
cross correlation, 536
cross product, 462
cross(), **462**, 463
crypt.h, 404
csplit, 579
csum(), 461, 463, **471**
ctrace(), 463, **506**
ctriangularmatrix(), 463, **509**
ctype.h, 404
cumprod(), 463, **472**
cumsum(), 463, **471**
current shell, 33, 41, 45, **49**, 51, 53
curve fitting, 483
curvefit(), 463, **483**

DARWIN, 97
data analysis, 469
data conversion rules, 223
data type conversion for arrays, 288
data(), 414
data2D(), 414, *see* CPlot
data2DCurve(), 414
data3D(), 414, *see* CPlot
data3DCurve(), 414
data3DSurface(), 414
dataFile(), 414, *see* CPlot
dataSetNum(), 414
date, 580
DBL_MAX, 102
DBL_MIN, 102
DBL_MINIMUM, 102
dd, 578
debug, 42
deconv(), 463, **530**
default, 19, 139
default I/O format, 380
default output format, 29
deferred-shape array, 104, 251, 301, 544
goto statement, 254
members of structures and unions, 256
switch statement, 254
del, 541
delete, 19, 322, 346, 549
deleteData(), 414
deletePlots(), 414, *see* CPlot
DenavitHartenberg, 512
DenavitHartenberg2, 512
derivative(), 463, **498**
derivatives(), 463, **498**
determinant(), 463, **505**
df, 578
diagonal matrix, 506, 509
diagonal(), 463, **506**
diagonalmatrix(), 463, **509**
diff, 580
diff3, 580
difference(), 463, **498**
dimension(), 414, *see* CPlot
dir, 541, 578
directory manipulation, 394
dirent.h, 394, 404
dirname, 580
dirs, 78
DISPLAY, 7, 76
displayTime(), 414, *see* CPlot
dlfcn.h, 404
dlopen(), 20
dlrunfun(), 20, 197
dlsym(), 20
do, 19
do-while, 140
dot command, 45, 53
dot product, 466
dot(), 463, **466**
double, 19, 103
double complex, 103, 542
Dramadah, 512
du, 578
dynamic allocation of memory, 149

dynamic allocation of mwnory, 267

echo, 580
egrep, 579
eigen(), 464, **525**
eigenvalues, 525
eigenvectors, 525
elementtype(), 20, 197, 305, 355
else, 19
else-if, 138
endl, 382, 553
endlocal, 541
endparalleltask, 21
enhanceText(), 414, *see* CPlot
entire domain, 210
enum, 19, 108
enumeration, 341
env, 77, 580
environ, 38, 195
environment variables, 76
    COLUMNS, 7
    DISPLAY, 7
    export, 78
    getenv(), 9, 76
    isenv(), 9, 76
    LINES, 7
    other shells, 78
    putenv(), 7, 9, 76
    remenv(), 9, 76
    setenv, 78
EOF, 30, 390
erase, 541
errno.h, 404
escape characters, 114
eval, 558
evaluation of array elements, 479
event designators, 57
event_t, 21
exclusive, 128, 544
exec, 53, 95, 408
execl(), 401
execle(), 401
execlp(), 401
execv(), 401
execve(), 401
execvp(), 401
exit, 54
exp(), 20, 210, 229, 233, 315
expand, 579
expm(), 464, **513**
export, 78
export plots, 433
expr, 580
expression evaluation, 75
expression statement, 136
expression substitution, 65, 386
extended complex plane, 221
extended finite complex plane, 221
extent, 104
extern, 19, 45

F_OK, 82
fabs(), 210, 229
factor, 580
factorial(), 464, **477**
false, 137, 543
fast Fourier transforms, 527
fchdir(), 401
fchown(), 401
fchroot(), 401
fcntl.h, 404
fdopen(), 401
fenv.h, 404, 543
fevalarray(), 464, **479**
fflush(), 370, 390
fft(), 464, **527**
fgetc(), 391
fgetpos(), 369
fgets(), 20, 391
fgrep, 579
Fiedler, 512
FILE, 389
file manipulation, 389
file name completion, 402
filename substitution, 67
fillStyle(), 414
filter(), 464, **532**
filter2(), 464, **535**
filtering, 529
find, 578
findvalue(), 464, **477**
flip matrix, 507
fliplr(), 464, **507**
flipud(), 464, **507**
float, 19
float.h, 404
floating-point, 118
floating-point types, 101
floor(), 20, 210, 229, 233, 315
FLT_EPSILON, 215
FLT_MAX, 102, 215
FLT_MIN, 102, 214
FLT_MINIMUM, 102, 214
fminimum(), 464, **487**
fminimums(), 464, **488**
fmod(), 20, 210, 229, 233
fmt, 579
fold, 579

fopen(), 389, 401
for, 19, 141
foreach, 19, 142, 336, 544, 549
FORTRAN, 199, 204, 574
Fortran, 574
fplotxy(), **450**
fplotxyz(), **455**
fpos_t, 369
fprintf, 19, 380, 385, 549
fprintf(), 371, 380
fputc(), 392
fputs(), 392
Frank, 512
fread(), 20, 391, 392
free(), 20, 150, 239, 545
FreeBSD, 97
freopen(), 369
frexp(), 20, 210, 229, 233
fscanf, 381
fscanf(), 20, 376, 381, 391
fseek, 390
fseek(), 393
fsetpos, 390
fsetpos(), 369
fsolve(), 464, **497**
fstat(), 401
ftell(), 393
fully buffered, 369
fully-specified-shape array, 356
fully-specified-shape arrays, 298
func2D(), 414
func3D(), 414
funcp2D(), 414
funcp3D(), 414
function, 111, 157
  communication between functions, 189
  function files, 195
  function prototypes, 160
  functions return computational arrays, 312
  generic functions, 197
  nested functions, 165
  recursive functions, 165
function files, 12, 34
function prototype scope, 266
functions, 312
funm(), 464, **513**
fwide(), 369
fwprintf(), 552
fwrite(), 392
fwscanf(), 552
fzero(), 464, **497**

gawk, 579
gcd(), 464, **467**
Gear, 512
generic array functions, 314
generic functions, 20, 197, 210, 542, 548
getc(), 391
getenv(), 9, 20, 30, 76, 401, 558
gethostname(), 401
getLabel(), 414, *see* CPlot
getnum(), 464, **469**
getOutputType(), 414
gets(), 20, 391
getSubplot(), 414, *see* CPlot
getTitle(), 414, *see* CPlot
glob.h, 404
global, 44, 45, 544
GNUTERM, 413
goto, 19, 44, 144, 254
greatest common divisor, 467
grep, 579
grid(), 414, *see* CPlot, *see* CPlot
grp.h, 404
gunzip, 580

Hadamard, 512
Hankel, 512
head, 579
help, 9, 78
hessdecomp(), 464, **519**
Hessenberg decomposition, 519
hexadecimal, 116
hexadecimal floating-point contants, 118, 543
Hilbert, 512
histogram(), 464, **481**
history, 54, 57, 559
history substitution, 56
HOME, 21
hostname, 580
householdermatrix(), 464
HP_UX, 97
hypot(), 121, 550

I/O Format, 371
  aggregate data type, 385
I/O format, 385
id, 580
identifiers, 21, 44
  linkages, 44
  name spaces, 45
  predefined identifiers, 21
  scope rules, 44
identity matrix, 508
identitymatrix(), 464, **508**
IEEE 754, 542
IEEE 754 standard, 101
if, 19, 137

if-else, 138
ifft(), 464, **527**
imag(), 20, 224, 229, 233, 316
import, 39, 95
importf, 36, 39, 95
inclusive, 128
incomplete array, 105
indent, 579
inet.h, 404
Inf, 19, 102, 383, 542, 549, 550
initialization, 119, 268, 284
inline, 19, 112, 543
input, 369
installpkg.ch, 411
int, 19, 99, 100
INT_MAX, 100
INT_MIN, 100
integer constants, 116
integral1(), 464, **502**
integral2(), 464, **502**
integral3(), 464, **503**
integration, 502
integration2(), 464, **503**
integration3(), 464, **504**
interp1(), 464, **481**
interp2(), 464, **483**
interpolation, 481, 483
inttypes.h, 404, 543
inverse fast Fourier transforms, 527
inverse Hilbert matrix, 512
inverse matrix, 523
inverse(), 464, **523**
ioctl(), 20
iostream.h, 382, 404
isenv(), 9, 76
iskey(), 20
iso646.h, 404, 543
isUsed(), 414, *see* CPlot
iteration statement, 140

join, 579

K&R C, 549
keywords, 19
kill(), 401
Korn shell, 47, 80

label(), 414, *see* CPlot
LANG, 21
LC_ALL, 21
LC_COLLATE, 21
LC_CTYPE, 21
LC_MONETARY, 21
LC_NUMERIC, 21
LC_TIME, 21
lchown(), 401
lcm(), 464, **467**
ldexp(), 20, 210, 229, 233
least common multiple, 467
left shift, 128
legend(), 415, *see* CPlot
legendLocation(), 415, *see* CPlot
legendOption(), 415
less, 579
lexical elements, 18
libintl.h, 404
library, 404
limits.h, 99, 404
lindata(), 464, **469**
line buffered, 369
line(), 415, *see* CPlot
linear spaces, 524
linear system of equations, 520
lineType(), 415, *see* CPlot
link(), 401
linkages of identifiers, 44
linsolve(), 464, **521**
linspace(), 464, **469**
Linux, 97
llsqcovsolve(), 464, **522**
llsqnonnegsolve(), 464, **522**
llsqsolve(), 464, **521**
ln, 578
local, 45, 111, 170, 544
locale.h, 338, 404
log(), 20, 210, 229, 233, 315
log10(), 20, 210, 229, 233, 315
logdata(), 464, **469**
login, 6
logm(), 464, **513**
LOGNAME, 21
logname, 580
logspace(), 464, **469**
long, 19, 99, 100
long double, 548
long double complex, 548
long long, 99, 100, 542, 546
longjmp(), 142
loop, 140
  do-while loop, 140
  for loop , 141
  foreach loop, 142
  while loop, 140
ls, 578
lstat(), 401
LU decomposition, 514
ludecomp(), 464, **514**
lvalue, 233
lvalues related to complex numbers, 233

machine epsilon, 215
macro replacement, 89
Magic, 512
main(), 37, 192
make, 579
malloc(), 149, 239
malloc.h, 404
margins(), 415, *see* CPlot
math.h, 404
matrix, 282
matrix analysis, 512
max(), 20, 197
maximization of functions, 487
maximum value of function, 470
maxloc(), 461, **470**, 477
maxloc(0, 465
maxv(), 465, **470**
MB_CUR_MAX, 113
mbstate_t, 369
mbstowcs(), 116
md5sum, 579
mean value, 473
mean(), 461, 465, **473**
median value, 473
median(), 461, 465, **473**
member function, 553
members of struct/union, 105
memchr(), 331
memcmp(), 331
memcpy(), 20, 330
memmove(), 20, 330
memset(), 20, 333
metanumber, 383
metanumbers, 217
min(), 20, 197
minimization of functions, 487
minimum value of function, 470
minloc(), 461, 465, **470**, 477
minv(), 465, **470**
mkdir, 48, 578
mkdir(), 401
modf(), 20, 210, 229, 233
more, 579
move, 541
mqueue.h, 404
multi-dimensional arrays of fixed length, 244
multibyte characters, 113
multiple files, 39
mv, 578

name spaces of identifiers, 45
NaN, 19, 102, 383, 542, 549, 550
nested function, 189
nested functions, 165, 554
  lexical levels, 167
  nested recursive functions, 171
  prototypes of nested functions, 170
  scopes, 167
netconfig.h, 404
netdb.h, 404
netdir.h, 404
netinet/in.h, 404
new, 19, 149, 322, 346, 549
new.h, 404
nl, 579
nm, 579
nonlinear equation, 497
norm of matrix, 476
norm(), 465, **476**
not, 128
NULL, 19, 150, 311, 545, 549
NULL directive, 93
null space, 525
null statement, 136
nullspace(), 465, **525**
numeric.h, 404

octal, 116
od, 579
oderk(), 465, **499**
offsetof(), 402
one's complement, 128
one-dimensional arrays, 243
open(), 20, 401
opendir(), 394
opening a directory, 394
operator, 19, 112
operators, 122
  address and indirection operators, 133
  arithmetic operators, 125
  assignment operators, 128
  bitwise operators, 128
  cast operators, 131
  comma operator, 122, 132
  conditional operator, 129, 295
  functional type cast operators, 131
  increment and decrement operators, 134
  logical operators, 128
  relational operators, 125
  ternary conditional operator, 122
  unary operator, 122
  unary operators, 133
option of program, 84, 193
ordinary differential equations, 499
origin(), 415, *see* CPlot
orthonormal base, 524
orthonormalbase(), 465, **524**
output, 369

outputType(), 415, *see* CPlot
over-determined, 521
overloading, 112, 282, 555

pack(), 95
package, 39, 95, 409
parse, 51
Pascal, 512
pass by reference, 179
passing array, 243
    passing multi-dimensional arrays, 243
    passing one-dimensional arrays, 243
passing member function, 362
paste, 579
patch, 579
PATH, 21
pathchk, 580
pclose(), 74
pinverse(), 465, **524**
pipe(), 401
pipeline, 73
plot, 413
PLOT_ANGLE_DEG, 442, 454
PLOT_ANGLE_RAD, 442, 454
PLOT_AXIS_X, 421
PLOT_AXIS_X2, 421
PLOT_AXIS_XY, 421
PLOT_AXIS_XYZ, 421
PLOT_AXIS_Y, 421
PLOT_AXIS_Y2, 421
PLOT_AXIS_Z, 421
PLOT_BORDER_ALL, 423
PLOT_BORDER_BOTTOM, 423
PLOT_BORDER_LEFT, 423
PLOT_BORDER_RIGHT, 423
PLOT_BORDER_TOP, 423
PLOT_COORD_CARTESIAN, 451
PLOT_COORD_CYLINDRICAL, 451
PLOT_COORD_SPHERICAL, 451
PLOT_OFF, 423
PLOT_ON, 423
PLOT_OUTPUTTYPE_DISPLAY, 433
PLOT_OUTPUTTYPE_FILE, 433
PLOT_OUTPUTTYPE_STREAM, 433
PLOT_PLOTTYPE_BOXERRORBARS, 439
PLOT_PLOTTYPE_BOXES, 439
PLOT_PLOTTYPE_BOXXYERRORBARS, 439
PLOT_PLOTTYPE_CANDLESTICKS, 439
PLOT_PLOTTYPE_DOTS, 439
PLOT_PLOTTYPE_FILLEDCURVES, 439
PLOT_PLOTTYPE_FINANCEBARS, 439
PLOT_PLOTTYPE_FSTEPS, 439
PLOT_PLOTTYPE_HISTEPS, 439
PLOT_PLOTTYPE_IMPULSES, 439, 450
PLOT_PLOTTYPE_LINES, 439, 450
PLOT_PLOTTYPE_LINESPOINTS, 439, 450
PLOT_PLOTTYPE_POINTS, 439, 450
PLOT_PLOTTYPE_STEPS, 439
PLOT_PLOTTYPE_SURFACES, 450
PLOT_PLOTTYPE_VECTORS, 439, 450
PLOT_PLOTTYPE_XERRORBARS, 439
PLOT_PLOTTYPE_XERRORLINES, 439
PLOT_PLOTTYPE_XYERRORBARS, 439
PLOT_PLOTTYPE_XYERRORLINES, 439
PLOT_PLOTTYPE_YERRORBARS, 439
PLOT_PLOTTYPE_YERRORLINES, 439
PLOT_TEXT_CENTER, 424
PLOT_TEXT_LEFT, 424
PLOT_TEXT_RIGHT, 424
plotting(), 415, *see* CPlot
plotType(), 415, *see* CPlot
plotxy(), **446**, 545
plotxyf(), **448**, 545
plotxyz(), **454**, 545
plotxyzf(), **455**, 545
point(), 415, *see* CPlot
pointer, 104, 147
    arrays of pointers, 151
    pointer to functions, 186
    pointers to pointers, 153, 193
pointer arithmetic, 147
pointer to arrays of assumed shape, 326
pointer to assumed-shape array, 544
pointer to computational arrays, 319
pointers, 119, 189
pointType(), 415, *see* CPlot
polar(), 20, 197, 229, 233
polarPlot(), 415, *see* CPlot
poll.h, 404
polycoef(), **493**
polyder(), 465, **491**
polyder2(), 465, **492**
polyeval(), 465, **490**
polyevalarray(), 465, **490**
polyevalm(), 465, **490**, **513**
polyfit(), 465
polygon(), 415, *see* CPlot
polymorphic functions, 353
polymorphism, 351
    array of reference, 352
    generic mathematical functions, 352
polynomials, 489
    characteristic polynomial of matrix, 496
    curve fitting, 486
    derivative, 491
    evaluation, 490
    factorization, 494
    find coefficients, 493

find roots, 492
pop, 95
popd, 78
popen(), 74, 401
POSIX, 404
pow(), 20, 210, 229, 233, 316, 550
pr, 579
pragma, 39, 338, 409
_fpath, 95
_ipath, 95
_lpath, 95
_path, 95
exec, 95, 408
import, 39, 95, 409
importf, 39, 95, 409
pack(), 95
package, 95, 409
remkey(), 95
remvar(), 95
predefined identifiers, 21
preprocessing directive, 87
printenv, 580
printf, 19, 549
printf(), 371, 380
private, 19, 345, 549, 553
product of array elements, 472
product(), 461, 465, **472**
program execution, 36
multiple files, 39
program mode, 10
program startup, 37
program termination, 38
promotion of scalars to arrays in operations, 296
prompt, 47
administrator prompt, 47
superuser prompt, 47
$ Bourne, Korn, BASH shell prompt, 47
% C shell prompt, 47
> Ch prompt, 47
protected, 21
pseudo inverse matrix, 523
pthread.h, 404
public, 19, 345, 549, 553
punctuators, 26
push, 95
pushd, 78
putc(), 392
putenv(), 7, 9, 30, 76, 401, 558
puts(), 392
PWD, 21
pwd, 48, 53, 54, 57, 65, 578
pwd.h, 404

QNX, 97
qr, 563
QR decomposition, 518
qrdecomp(), 465, **518**
qrdelete, 563
qrinsert, 563
qsort(), 196
quick substitution, 58

R_OK, 82
random numbers, 466
range of subscript for an index, 105
rank, 104
rank of matrix, 507
rank(), 465, **507**
ranlib, 579
rcondnum(), 465
re_comp.h, 404
read(), 20
readdir(), 394
reading a directory, 396
readline.h, 404
real(), 20, 224, 229, 233, 316
realloc(), 149, 239
rectangle(), 415, *see* CPlot
recvevent, 21
redirection, 70
reference, 105, 108, 200
passing arguments by references, 202
references in statements, 200
reference type, 199, 553
regex.h, 404
register, 19, 45, 548
remenv(), 9, 76, 558
remkey, 54
remkey(), 95
remove(), 82, 401
removeHiddenLine(), 415, *see* CPlot
remvar, 54
remvar(), 95
ren, 541
rename(), 401
reserved symbols, 20
residue(), 465, **494**
resize, 7
resize(), 7
restrict, 19, 112, 402, 543, 548
return, 19, 144
rewind, 390
rewinddir(), 394
riemann sphere, 221
right shift, 128
rlimit, 30
rm, 578
rmdir, 578

rmdir(), 401
roots(), 465, **492**
Rosser, 512
rot90(), 465, **508**
rotate matrix, 508
rsf2csf, 563

safe Ch, 28, 78, 400
safe Ch shell, 47
scaleType(), 415, *see* CPlot
scanf, 19, 549
scanf(), 376, 381
sched.h, 404
Schur decomposition, 520
schurdecomp(), 465, **520**
scope resolution operator, 553
scope rules, 44
script files, 11, 34
sdiff, 580
search order, 39
sed, 579
SEEK_CUR, 393
SEEK_END, 393
SEEK_SET, 393
selection statement, 137
semaphore.h, 404
sendevent, 21
set, 558
set_new_handler(), 347
setbuf(), 370
setenv, 78, 558
setjmp(), 142
setjmp.h, 404
setlocale(), 338
setrlimit(), 20, 30, 401
setvbuf(), 370
sh, 580
shape, 104, 309
shape(), 20, 301
SHELL, 21
short, 19, 99
showMesh(), 415, *see* CPlot
showvar, 49, 54, 56
SHRT_MAX, 99
SHRT_MIN, 99
sign function, 466
sign(), 465, **466**
signal.h, 404
signbit(x), 125
signed, 19
signed char, 99
signed int, 99
signed long, 99
signed long long, 99
signed short, 99
simple statement, 136
sin(), 20, 210, 229, 233, 315
singular value decomposition, 515
sinh(), 20, 210, 229, 233, 315
size, 579
size(), 415, *see* CPlot
size3D(), 415, *see* CPlot
size_t, 329
sizeof, 19, 257, 262, 269
sizeof(), 229, 233
sizeRatio(), 415, *see* CPlot
sleep, 580
smooth(), 415
socket(), 401
socketpair(), 401
Solaris, 97
sort, 579
sort data, 477
sort(), 461, 465, **478**
special matrix, 512
    Cauchy, 512
    Chebyshev, 512
    Chow, 512
    Circul, 512
    Clement, 512
    DenavitHartenberg, 512
    DenavitHartenberg2, 512
    Dramadah, 512
    Fiedler, 512
    Frank, 512
    Gear, 512
    Hadamard, 512
    Hankel, 512
    Hilbert, 512
    InverseHilbert, 512
    Magic, 512
    Pascal, 512
    Rosser, 512
    Toeplitz, 512
    Vandermonde, 512
    Wilkinson, 512
specialmatrix(), 465, **512**
split, 579
sqrt(), 20, 210, 229, 233, 315, **513**
sqrtm(), 465
sscanf(), 20, 376, 381
stackvar, 49
standard deviation, 474
start, 541
startup, 6, 27, 400
startup in Linux, 6
startup in Mac OS X, 6
startup in windows, 7

stat(), 395, 401
statement, 136
    break, 143
    case, 139
    compound statement, 136
    continue, 143
    default, 139
    else-if, 138
    expression, 136
    goto, 144
    if, 137
    if-else, 138
    iteration statement, 140
    jump statement, 142
    labeled statement, 145
    loop, 140
    null, 136
    return, 144
    selection statement, 137
    simple statement, 136
    switch, 138
static, 19, 45
static member, 553
static storage duration, 45, 119, 136, 249
statistics, 469
status, 558
std(), 461, 465, **474**
stdarg.h, 180, 353, 404
stdbool.h, 137, 404, 543
stddef.h, 113, 404
stderr, 369
stdin, 369
stdin.h, 543
stdio.h, 389, 404
stdlib.h, 404
stdout, 369
stop, 43
storage duration of objects, 45, 249
str2ascii(), 334, 544
str2mat(), 334, 544
stradd(), 20, 30, 33, 334, 407, 544
strcasecmp(), 333
strcat(), 20, 109, 330
strchr(), 20, 331
strcmp(), 20, 331
strcoll(), 20, 331
strconcat(), 333
strcpy(), 20, 330
strcspn(), 331
stream, 369
strerror(), 20, 333
streval(), 20, 75, 558
strgetc(), 334, 544
string, 109, 545
string literals, 116
string.h, 329, 404
string_t, 19, 334, 549
stringcat(), 109
stringcat2(), 110
strings, 329, 579
strip, 579
strjoin(), 333, 558
strlen(), 20, 333
strncasecmp(), 333
strncat(), 20, 330
strncmp(), 331
strncpy(), 20, 330
stropts.h, 404
strparse(), 75
strpbrk(), 331
strputc(), 334, 544
strrchr(), 331
strrep(), 334, 544
strspn(), 331
strstr(), 331
strtod(), 20
strtok(), 20, 331, 336
strtok_r(), 336
strtol(), 20
strtoul(), 20
struct, 19, 106
structure, 269, 339
strxfrm(), 20, 331
stty, 31
subplot(), 415, *see* CPlot
substitution, 64, 66
    command name substitution, 66
    command substitution, 69, 135
    expression substitution, 65, 386
    filename substitution, 67
    variable substitution, 64, 386
sum, 579
sum of array elements, 471
sum(), 461, 465, **471**
svd(), 465, **515**
swap(), 179, 202
switch, 19, 138
swprintf(), 552
swscanf(), 552
syslog.h, 404
system(), 33, 73, 401

tail, 579
tan(), 20, 210, 229, 233, 315
tanh, 315
tanh(), 20, 210, 229, 233
tar, 580
tar.h, 404

tee, 580
TERM, 21, 30
termios.h, 404
ternary conditional operator, 122
test, 580
text(), 415, *see* CPlot
tgmath.h, 404, 543
this, 19, 549, 553
tics(), 415, *see* CPlot
ticsDay(), 415, *see* CPlot
ticsDirection(), 415, *see* CPlot
ticsFormat(), 415, *see* CPlot
ticsLabel(), 415, *see* CPlot
ticsLevel(), 415, *see* CPlot
ticsLocation(), 415, *see* CPlot
ticsMirror(), 415, *see* CPlot
ticsMonth(), 415, *see* CPlot
ticsPosition(), 415
ticsRange(), 415
time, 541
time.h, 404
title, 541
title(), 415, *see* CPlot
tiuser.h, 404
Toeplitz, 512
token merging in macro expansions, 91
    ## preprocessor operator, 91
toolkit, 408
tools for scientific data analysis, 461
tools for scientific data plotting, 413
touch, 578
tr, 579
trace of matrix, 506
trace(), 465, **506**
transpose(), 20, 197, 291, 317
triangular matrix, 509
triangularmatrix(), 465, **509**
trigraph, 24
trigraphs, 18
troff, 579
true, 137, 543
try, 21
tsort, 579
type, 541
type qualifiers, 112
typedef, 258, 267
typographical conventions, iii
TZ, 21

UCHAR_MAX, 99
UINT_MAX, 100
umask, 558
umask(), 20, 29, 401
unalias, 63, 558
uname, 580
unary operator, 122
unbuffered I/O, 369
under-determined, 521
unexpand, 579
uniform random numbers, 466
union, 19, 107, 269, 340
uniq, 579
unistd.h, 404
unlink(), 401
unset, 558
unsetenv, 558
unsigned, 19
unsigned char, 99
unsigned int, 99, 100
unsigned long, 99
unsigned long long, 99
unsigned short, 99
unwrap, 478
unwrap(), 465
urand(), 465, **466**
USER, 21
USHRT_MAX, 99
using, 382, 553
utime.h, 404

va_arg(), 180
va_arraydim(), 180, 353, 355
va_arrayextent(), 180, 353, 355
va_arraynum(), 180, 353, 355
va_arraytype(), 180, 353, 355
va_copy(), 180, 543
va_count(), 180, 353
va_datatype(), 180, 353, 355
va_end(), 180, 353
va_list, 180
VA_NOARG, 180, 181, 353
va_start(), 180
va_tagname(), 180
Vandermonde, 512
variable length arrays, 248, 542
variable substitution, 64, 386
variadic function, 545, 554
ver, 541
verbatim output block, 385
verify, 541
vfprintf(), 20, 380
vfwprintf(), 552
vi, 579, 581
vim, 579
virtual, 21, 112
VLA, 248
VLAs, 542
void, 19, 108

vol, 541
volatile, 19, 112, 548
vprintf(), 20, 380
vsprintf(), 20, 380
vswprintf(), 552
vwprintf(), 552

W_OK, 82
wait.h, 404
wc, 579
wchar.h, 404, 543
wchar_t, 329, 337
wcstombs(), 116
wctype.h, 404, 543
Web plotting, 457
which, 78, 84, 193, 578
while, 19, 140
whoami, 580
whole arrays, 284
wide characters, 113, 337
wide strings, 116, 338
Wilkinson, 512
Win32, 97
Windows, 540
Windows 2000, 540
Windows 7, 540
Windows Vista, 540
Windows XP, 540
WinMain(), 38
wprintf(), 552
wscanf(), 552

X_OK, 82
xargs, 580
xcorr(), 465, **536**
xhost, 7, 76

zoom plots, 433