EXPERT INSIGHT

# C++ STL Cookbook

Master the latest STL features in C++23 and 26 with practical recipes for modern C++ development



Second Edition

Bill Weinman

packt

# C++ STL Cookbook

Second Edition

Master the latest STL features in C++23 and 26 with practical recipes for modern C++ development

**Bill Weinman**

‹packt›

# C++ STL Cookbook

Second Edition

Copyright © 2026 Packt Publishing

*All rights reserved*. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.
Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.
Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

**Portfolio Director:** Kunal Chaudhari
**Relationship Lead:** Denim Pinto
**Project Manager:** K. Loganathan
**Content Engineer:** Deepayan Bhattacharjee
**Technical Editor:** Aditya Bharadwaj
**Copy Editor:** Deepayan Bhattacharjee
**Indexer:** Pratik Shirodkar
**Production Designer:** Ajay Patule
**Growth Lead**: Vinishka Kalra

First published: May 2022
Second edition: May 2026

Production reference: 1260526

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK.

ISBN 978-1-83620-425-1
www.packtpub.com

# Contributors

## About the author

**Bill Weinman** built his first computer in 1972, using a 4-bit Intel microprocessor and parts from a local radio store. Over a 45+ year career, he has consulted for major corporations and government organizations, including NASA, IBM, Bank of America, the US Navy, and KWHY-TV, where he designed the world's first broadcast lower-third stock ticker.

An early contributor to the original HTTP specification and the Apache web server project, since the late 1990s Mr. Weinman has written popular books and courseware on C, C++, Perl, Python, SQL, and more. His courses have consistently been ranked in the top 10 on lynda.com and LinkedIn Learning's training sites.

*This book represents the hard work of many talented people. Deep appreciation goes to Deepayan Bhattacharjee, K. Loganathan, Denim Pinto, and the wonderful staff at Packt. I am grateful for the support of my assistants, RiKi and Kurstin, who keep my life running smoothly as I focus on research and writing, and my family and friends, without whom none of this would make any sense at all.*

# About the reviewer

**Desislav Andreev** is experienced software engineer, lecturer, and a book author with a PhD in artificial intelligence systems and quantum machine learning. He has a demonstrated history of working in the automotive software engineering sector, virtualization, and in the area of the higher education.

Desislav is also skilled in creating complex system and software architectures, designing core infrastructural solutions, and building systems for autonomous driving. He's interested in computer graphics, operating systems, supercomputers, systems engineering, and even simple software tools for everyday tasks.

Being a strong professional he's currently working as a lead core infrastructure developer. He is an author of several publications in IEEE, respected journals and international scientific conferences. He's also an author of Packt's *C++ Programming for Linux Systems*.

*First, I'd like to thank Packt and Bill Weinman for this opportunity – it was an amazing experience, and it was quite thrilling for me to wait for each chapter!*

*I also want to thank my great family for putting up with me and giving me enough time to do my side quests! My love goes to my wonderful wife – Svetla, my precious daughters – Kalina and Maria, and my sweet dog - Robyn.*

# Table of Contents

**Preface** xiii

**About this book** ... xiii

Prerequisite knowledge • xiii

The STL's power is derived from templates (a brief primer) • xiv

This book uses the C++23 standard • xv

Hiding the std:: namespace • xv

Type aliases with using • xvi

Abbreviated function templates • xvi

Use the STL to solve real-world problems • xvii

**Free benefits with your book** ... xxii

**Chapter 1: Introduction to New C++23 Features** 1

**Simplify your code with the std module** ... 1

**Format text with the new print() and println() functions** ... 4

**Use synchronous generators** ... 5

**Use mdspan as a multidimensional array** ... 8

**Initialize containers from ranges** ... 14

**Find content in strings and ranges with contains()** ... 16

**Subscribe to Deep Engineering** ... 21

**Chapter 2: Best Practices** 23

**Use braced initialization** ... 23

**Use span to make C-arrays safer** ... 28

**Use structured binding to return multiple values** ... 31

**Initialize variables within if and switch conditionals** ... 35

**Use template argument deduction for simplicity and clarity** ... 38

**Use RAII to preserve memory integrity ..... 42**

## Chapter 3: STL Containers 47

**A quick overview of the STL container types ..... 48**

Sequential containers • 48

Associative containers • 48

Container adapters • 49

**Use uniform erase functions to delete elements from a container ..... 50**

**Delete items from an unsorted vector ..... 55**

**Access vector elements safely ..... 58**

**Keep vector elements sorted ..... 61**

**Efficiently insert elements into a map ..... 64**

**Efficiently modify keys in a map ..... 70**

**Use unordered_map with custom keys ..... 74**

**Sort and filter user input with set ..... 77**

**Create a simple RPN calculator with deque ..... 80**

**Count word frequency with map ..... 87**

**Find long sentences with a vector of vectors ..... 92**

**Create an efficient ToDo list using multimap ..... 95**

**Subscribe to Deep Engineering ..... 98**

## Chapter 4: STL Compatible Iterators 99

**Iterators are fundamental ..... 99**

Iterator categories • 102

Iterator concepts • 102

**Create an iterable range ..... 105**

**Use traits with your iterators for STL compatibility ..... 109**

**Use iterator adaptors to populate STL containers ..... 112**

**Iterate backward with a reverse adaptor ..... 116**

**Iterate objects of unknown length with a sentinel ..... 120**

**Create a generator with iterators ..... 123**

**Create a generator with coroutines ..... 128**

**Build a zip iterator adaptor** ..... **136**
**Create a random-access iterator** ..... **143**
*Operator overloads • 145*
*Validation code • 146*

**Chapter 5: Lambda Expressions** **151**

**Lambda expressions** ..... **151**
Closures • 152
**Use lambdas for scoped, reusable code** ..... **153**
*Capture list • 157*
*Parameters • 158*
*The mutable modifier (optional) • 158*
*The constexpr specifier (optional) • 159*
*The noexcept attribute (optional) • 159*
*The trailing return type (optional) • 159*
**Use lambdas as predicates with algorithms** ..... **160**
**Create a polymorphic wrapper with std::function** ..... **164**
**Concatenate lambdas with recursion** ..... **168**
**Combine predicates with logical conjunction** ..... **170**
**Pass common parameters to multiple lambdas** ..... **171**
**Create a simple jump table with mapped lambdas** ..... **173**
**Subscribe to Deep Engineering** ..... **177**

**Chapter 6: STL Algorithms** **179**

**Copy from one iterator to another** ..... **180**
**Join container elements into a string** ..... **184**
**Sort containers with std::sort** ..... **188**
**Modify containers with std::transform** ..... **194**
**Search containers with std::find** ..... **197**
*The new contains() method • 201*
**Limit container values with std::clamp** ..... **202**
**Sample data sets with std::sample** ..... **205**

**Use ranges for more powerful algorithms ..... 208**
**Generate permutations of data sequences ..... 214**
**Merge sorted containers ..... 217**

**Chapter 7: Strings and Formatting 221**

**Use string_view as a lightweight string object ..... 223**
**Format text with the format library ..... 228**
**Print strings with std::print and std::println ..... 234**
**Concatenate strings ..... 238**
*Benchmarks • 240*
*Why the performance discrepancy? • 243*
*Why would I choose one over another? • 244*
**Transform strings ..... 244**
**Trim whitespace from strings ..... 248**
**Read strings from user input ..... 250**
**Count words in a file ..... 254**
**Initialize complex structures from file input ..... 255**
**Customize a string class with char_traits ..... 259**
**Parse strings with regular expressions ..... 263**
**Subscribe to Deep Engineering ..... 266**

**Chapter 8: Utility Classes 267**

**Manage optional values with std::optional ..... 267**
**Use std::any for type safety ..... 272**
**Store different types with std::variant ..... 276**
Differences from the union primitive • 276
**Time events with std::chrono ..... 282**
*The clock classes • 287*
*std::duration class • 287*
**Use fold expressions for generic tuples ..... 288**
Fold expressions • 288
**Manage allocated memory with std::unique_ptr ..... 292**

**Share objects with std::shared_ptr ..... 299**
**Use weak pointers with shared objects ..... 303**
**Share object members with aliased smart pointers ..... 308**
**Compare random number engines ..... 310**
**Comparison of random number distribution generators ..... 314**

**Chapter 9: Concurrency and Parallelism 319**

**Sleep a process for a specific amount of time ..... 320**
There's more... • 321
**Use std::thread for concurrency ..... 321**
There's more... • 326
**Use std::async for concurrency ..... 328**
Simplify promise and future with std::async() • 333
**Run STL algorithms in parallel with execution policies ..... 334**
**Use mutex and lock to safely share data ..... 337**
There's more... • 345
**Share flags and values with std::atomic ..... 345**
Specializations • 348
*Pointers and smart pointers • 348*
*Floating-point types • 349*
*Integral types • 349*
Standard aliases • 349
Lock-free variations • 350
There's more... • 351
**Initialize threads with std::call_once ..... 352**
**Resolve the producer-consumer problem with std::condition_variable ..... 354**
**Implement multiple producers and consumers ..... 358**
**Suspend and resume execution with coroutines ..... 363**
**Use coroutines for a producer-consumer solution ..... 370**
**Subscribe to Deep Engineering ..... 375**

**Chapter 10: Using the File System** **377**

**Specialize std::formatter for the path class** ..... **378**
**Use manipulation functions with path** ..... **382**
**List files in a directory** ..... **386**
**Search directories and files with a grep utility** ..... **397**
**Rename files with regex and directory_iterator** ..... **403**
**Create a disk usage counter** ..... **408**

**Chapter 11: More Practical Ideas** **415**

**Create a trie class for search suggestions** ..... **415**
**Calculate the error sum of two vectors** ..... **423**
There's more... • 427
**Build your own algorithm: split** ..... **427**
There's more... • 432
**Leverage existing algorithms: gather** ..... **433**
**Remove consecutive whitespace** ..... **436**
**Convert numbers to words** ..... **438**
There's more... • 446
**Subscribe to Deep Engineering** ..... **447**

**Chapter 12: C++26 and the Future** **449**

**Static reflection** ..... **449**
**Contracts** ..... **451**
**Structural pattern matching** ..... **452**
**Other C++26 improvements** ..... **453**

**Chapter 13: Unlock Your Exclusive Benefits** **455**

**Unlock this Book's Free Benefits in 3 Easy Steps** ..... **456**

**Other Books You May Enjoy** **461**

**Index** **465**

# Preface

## About this book

The *C++ STL Cookbook* provides recipes to help you get the most out of the C++ STL (Standard Template Library), including new features introduced with C++23.

C++ is a rich and powerful language. Built upon C, with extensions for type safety, generic and object-oriented programming, C++ is essentially a low-level language. The STL provides a broad set of higher-level classes, functions, and algorithms to make your programming job easier, more effective, and less prone to error.

I view C++ as five languages cobbled together. The formal specification includes 1) the entire *C language*, 2) C's cryptic-yet-powerful *macro preprocessor,* 3) a feature-rich *class/object* model, 4) a *generic programming* model called *Templates*, and finally, built upon C++ classes and templates, 5) *the Standard Template Library (STL)*.

## Prerequisite knowledge

This book presumes that you have a basic understanding of C++, including syntax, structure, data types, classes and objects, templates, and the STL.

The recipes and examples in this book presume that you understand the need to `#include` certain headers to use library functions. The recipes don't usually list all the necessary headers, preferring to focus on the techniques at hand. You're encouraged to download the example code, which has all the necessary `#include` directives and other relevant front matter.

You may download the example code from GitHub: `https://github.com/PacktPublishing/CPP-STL-Cookbook-Second-Edition`.

These assumptions mean that when you see a piece of code like this:

```
println("hello, world");
```

You should already know that you'll need to put this code in a `main()` function, you'll need to `#include` the `<print>` header, and `print` is a function in the `std::` namespace:

```
#include <print>
int main() {
```

```
    std::println("hello, world");
}
```

## The STL's power is derived from templates (a brief primer)

*Templates* are how C++ does *generic programming,* code that's independent of type while retaining type safety. C++ templates allow you to use tokens as placeholders for types and classes, like this:

```
template<typename T>
T add_em_up(T& lhs, T& rhs) {
    return lhs + rhs;
}
```

A template may be used for both classes and functions. In this template function, `T` represents a *generic type,* which allows the function to be used in the context of any compatible class or type:

```
int a {72};  // see braced initialization below
int b {47};
print("{}\n" , add_em_up<int>(a, b)); // see print() below
```

This invokes the template function with an `int` type. This same code can be used with any type or class that supports the `+` operator.

When the compiler sees a *template invocation,* like `add_em_up<int>(a, b)`, it creates a *specialization*. This is what makes the code type safe. When you invoke `add_em_up()` with an `int` type, the specialization will look something like this:

```
int add_em_up(int& lhs, int& rhs) {
    return lhs + rhs;
}
```

The specialization takes the template and replaces all instances of the `T` placeholder with the type from the invocation, in this case, `int`. The compiler creates a separate specialization of the template each time it's invoked with a different type. The specialization happens at compile time and is otherwise invisible to the author of the code.

STL *containers,* like `vector`, `stack`, or `map`, along with their *iterators* and other supporting functions and algorithms, are built with templates so they may be used generically while maintaining type safety. This is what makes the STL so flexible. Templates are the *T* in S*T*L.

## This book uses the C++23 standard

The C++ language is standardized by the International Organization for Standardization (ISO) on a roughly three-year cycle. The current standard is called C++23 (which was preceded by C++20, C++17, C++14, and C++11). The ISO C++23 formal standard (ISO/IEC 14882:2024) was published in October 2024.

C++23 builds upon C++20 with many refinements and some new features. While not as large an update as C++20, the refinements in C++23 are significant. Improvements include the new `print()` and `println()` functions, which are built upon the `format` library, an all-inclusive `std` module, updates to the `ranges` and `views` libraries, a `generator` class, and more.

The most visible updates are the `print()` and `println()` functions and the new `std` module.

## Hiding the std:: namespace

In most instances, the exercises in this book will hide the `std::` namespace. This is for page space and readability considerations. We know that most STL identifiers are in the `std::` namespace. I will normally use some form of the `using` declaration to avoid cluttering the examples with repetitive prefixes. For example, when using `println()` you can presume I've included a `using` declaration like this:

```
using std::println;  // println is now available sans prefix
println("Hello, Jimi!");
```

I usually *will not show* the `using` declaration in the recipe listings. This allows us to focus on the purpose of the example.

It is poor practice to import the entire `std::` namespace in your code. You should *avoid* a `using namespace` declaration like this:

```
using namespace std;    // bad. don't do this.
println("Hello, Jimi!");
```

The `std::` namespace includes thousands of identifiers and there's no good reason to clutter your namespace with them. The potential for collisions is not trivial and can be hard to track down. When you want to use a name or names without the `std::` prefix, the preferred method is to import the name(s) explicitly, one at a time, as above.

To further avoid namespace collisions, I often use a separate namespace for classes that will be re-used. I tend to use `namespace bw` for my personal namespace. You may use a name that works for you.

## Type aliases with using

This book uses the `using` directive for type aliases instead of `typedef`.

STL classes and types can be verbose at times. A templated iterator class, for example, may look like this:

```
vector<pair<int, string>>::iterator
```

Long type names are not just hard to type; they are prone to errors.

One common technique is to abbreviate long type names with `typedef`:

```
typedef vector<pair<int, string>>::iterator vecit_t
```

This declares an alias for our unwieldy iterator type. `typedef` is inherited from C and its syntax reflects that.

Beginning with C++11, the `using` keyword may be used to create a type alias:

```
using vecit_t = vector<pair<int, string>>::iterator;
```

In most circumstances, a `using` alias is equivalent to `typedef`. The most significant difference is that a `using` alias may be templated:

```
template<typename T>
using vec = vector<T>;
vec<int> x {};
```

For these reasons, and for the sake of clarity, this book prefers the `using` directive for type aliases.

## Abbreviated function templates

Beginning with C++20, an *abbreviated function template* may be specified without the template header. For example:

```
void printc(const auto& c) {
    for (auto i : c) {
        println("{}", i);
    }
}
```

The `auto` type in a parameter list works like an anonymous template `typename`. It is equivalent to:

```
template<typename T>
void printc(const T& c) {
    for (auto i : c) {
        println("{}", i);
    }
}
```

Though formally introduced with C++20, abbreviated function templates had already been supported by the major compilers for some time. This book uses abbreviated function templates in many of the examples.

## Use the STL to solve real-world problems

The recipes in this book use the STL to provide real-world solutions to real-world problems. They have been designed to rely exclusively on the STL and C++ standard libraries, with no non-standard libraries. This should make it easy for you to experiment and learn without the distractions of installing and configuring third-party code.

Now, let's go have some fun with the STL. *Happy learning!*

## Who this book is for

This book is for *intermediate to advanced* C++ programmers who want to get more out of the C++23 Standard Template Library. Basic knowledge of C++ and related concepts is necessary to get the most out of this book.

# What this book covers

*Chapter 1*, *Introduction to New C++23 Features*, introduces the new STL features in C++23. It aims to familiarize you with the new language features so you may use them with the STL.

*Chapter 2*, *Best Practices*, discusses modern STL features added in recent C++ versions.

*Chapter 3*, *STL Containers*, covers the STL's comprehensive library of containers.

*Chapter 4*, *STL Compatible Iterators*, shows how to use and create STL-compatible iterators.

*Chapter 5*, *Lambda Expressions*, covers the use of lambdas with STL functions and algorithms.

*Chapter 6*, *STL Algorithms*, provides recipes for using and creating STL-compatible algorithms.

*Chapter 7*, *Strings and Formatting*, describes the STL string and formatter classes.

*Chapter 8*, *Utility Classes*, covers STL utilities for date and time, smart pointers, optionals, and more.

*Chapter 9*, *Concurrency and Parallelism*, describes support for concurrency, including threads, async, atomic types, and more.

*Chapter 10*, *Using the File System*, covers the `std::filesystem` classes and how to put them to use with the latest advancements from C++23.

*Chapter 11*, *More Practical Ideas*, provides a few more solutions, including a trie class, string split, and more. This provides advanced examples of how to put the STL to use for real-world problems.

*Chapter 12*, *C++26 and the Future*, examines features proposed for the upcoming C++26 update.

## The recipes in this book use the GCC compiler

Unless otherwise noted, most of the recipes in this book have been developed and tested using the GCC compiler, version 17.2, the latest stable version as of this writing.

As of mid 2026, C++23 is not yet fully implemented on any available compiler. Of the three major compilers, *GCC* (GNU), *MSVC* (Microsoft), and *Clang* (Apple), the MSVC compiler is generally furthest along in implementing the new standard. Occasionally, we may run into a feature that is implemented on MSVC or another compiler, but not on GCC, in which case I will note which compiler I used. If a feature is not yet implemented on any available compiler, I will explain that I was unable to test it.

| Where possible, code has been tested on one or more of these compilers | |
|---|---|
| GCC 15.2 | Debian Linux 6.17 |
| LLVM/Clang 17.0 | macOS 15.7 |
| Microsoft C++ 19.44 | Windows 11 |

I strongly recommend that you install GCC to follow along with the recipes in this book. GCC is freely available under the GNU General Public License (GPL). The easiest way to get the latest version of GCC is to install *Debian Linux* (also GPL) and use `apt` with the `testing` repository.

**If you are using the digital version of this book, we suggest you type the code yourself or download the code from the GitHub repository (link in the next section). This will avoid errors due to copying and pasting formatted code from the e-book.**

## Download the example code files

You can download the example code files for this book from GitHub at `https://github.com/PacktPublishing/CPP-STL-Cookbook-Second-Edition`. In the event of updates and errata, code will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

## Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://packt.link/gbp/9781836204251`.

## Conventions used

There are several text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, file names, file extensions, path names, dummy URLs, user input, and social media handles. Here is an example: "The `insert()` method takes an `initializer_list` and calls the private function `_insert()`".

A block of code is set as follows:

```
int main() {
    Frac f {5, 3};
    println("Frac: {}", f);
}
```

Any command-line input or output is written as follows:

```
$ ./producer-consumer
Got 0 from the queue
Got 1 from the queue
Got 2 from the queue
finished!
```

**Bold**: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select **System info** from the **Administration** panel."

**Note**

Warnings or important notes appear like this.

**Tip**

Tips and tricks appear like this.

# Sections

In this book, you will find several headings that appear frequently (*How to do it...*, *How it works...*, *There's more...*, and *See also...*).

To give clear instructions on how to complete a recipe, use these sections as follows:

## How to do it

This section contains the steps required to follow the recipe.

## How it works

This section usually consists of a detailed explanation of what happened in the previous section.

## There's more...

This section consists of additional information about the recipe in order to make you more knowledgeable about the recipe.

## See also...

This section provides helpful links to other useful information about the recipe.

# Get in touch

Feedback from our readers is always welcome.

**General feedback**: If you have questions about any aspect of this book or have any general feedback, please email us at `customercare@packt.com` and mention the book's title in the subject of your message.

**Errata**: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit `http://www.packt.com/submit-errata`, click **Submit Errata**, and fill in the form.

**Piracy**: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

**If you are interested in becoming an author**: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `http://authors.packt.com/`.

## Join us on Discord!

Read this book alongside other users, developers, experts, and the author himself.

Ask questions, provide solutions to other readers, chat with the authors via Ask Me Anything sessions, and much more. Scan the QR or visit the link to join the community.



`https://packt.link/deep-engineering-cpp`

# Free benefits with your book

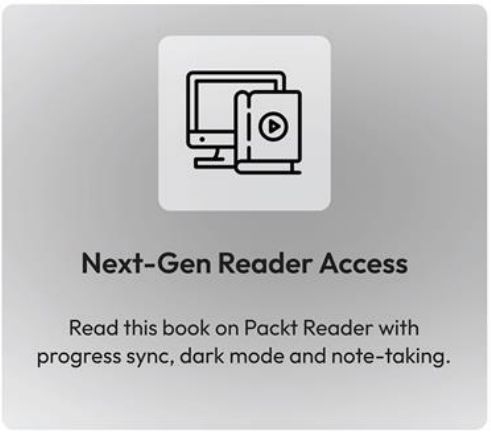
This book comes with free benefits to support your learning. Activate them now for instant access (see the "*How to Unlock*" section for instructions).

Here's a quick overview of what you can instantly unlock with your purchase:









## How to Unlock

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.




*Note: Keep your invoice handy. Purchases made directly from Packt don't require an invoice.*

## Share your thoughts

Once you've read *C++ STL Cookbook*, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback.



`https://packt.link/r/1836204256`

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

# 1

# Introduction to New C++23 Features

C++23 is not as large an update as C++20, yet it does add some significant new features. These include the new `print()` and `println()` functions, a new `generator` class, and multidimensional support for subscripts and views.

We'll cover some of these more impactful features in the following recipes:

- Simplify your code with the `std` module
- Format text with the new `print()` and `println()` functions
- Use synchronous generators
- Use `mdspan` as a multidimensional array
- Initialize containers from ranges
- Find content in strings and ranges with `contains()`

As of mid-2024, many of these new features, as well as some of the features introduced with C++20, are not yet supported on all the major C++ compilers. As we cover each feature, I'll point out which compilers support it.

This chapter aims to familiarize you with these new features in C++23, so you may use them in your own projects and understand them when you encounter them.

## Simplify your code with the std module

C++20 introduced the concept of *modules*. Modules represent a more compact and efficient alternative to including headers in your code. The promise of the modules feature is that it will eliminate the need for `#include` headers in most circumstances. For that promise to be

fulfilled, the standard library needs to be available as modules. C++23 provides the `std` module for that purpose. How to do it.

This recipe demonstrates the usage of the `std` module.

- We'll start with a simple *Hello, World* file:

```
#include <print>
#include <string>

int main() {
    std::string whoami {"I'm Spartacus"};
    std::println("Hello, World, {}", whoami);
}
```

When compiled and run, it produces this output:

```
Hello, World, I'm Spartacus
```

- When using the `std` module, you no longer need to use the `#include` directives. Now, our code can look like this:

```
import std;

int main() {
    std::string whoami {"I'm Spartacus"};
    std::println("Hello, World, {}", whoami);
}
```

When compiled and run, this produces the same output:

```
Hello, World, I'm Spartacus
```

## How it works

This feature uses the *modules* specification, introduced with C++20, to provide the entire *C++ Standard Library* in one pre-compiled module. Because modules are imported at link time, they do not import any code or symbols that are not directly referenced in the target code. This makes modules both safer and more efficient than `#include` directives. As a side benefit, using modules also eliminates the need for the cumbersome *include guards* that are necessary in `#include` files.

The `std` module is a powerful feature that can mitigate long lists of STL `#include` directives:

```
#include <print>
#include <iostream>
#include <stack>
#include <deque>
#include <map>
#include <string>
#include <utility>
#include <cctype>
#include <cmath>
#include <iterator>
#include <limits>
```

All of that may now be replaced with one `import` statement:

```
import std;
```

## There's more...

Many traditional `#include` headers have legacy *C Standard Library* symbols declared in the global namespace (e.g., `printf` or `puts`). The C++23 standard provides an alternative module named `std.compat` which provides those global namespace declarations. You may use this anywhere you would use the `std` module:

```
import std.compat;
```

Otherwise, both `std` and `std.compat` are the same.

**Note**

As of late 2025, the C++23 `std` module specification is not yet implemented on some compilers. In fact, some compilers do not yet implement C++20 modules. Therefore, this book and the accompanying examples will continue to use `#include` headers. I look forward to having more complete implementations in the future.

# Format text with the new print() and println() functions

When the `format` library was developed for C++20, the intent was to transition away from using `cout` and `iostream` for displaying formatted text. A `print()` function was planned but was not ready in time for the C++20 specification.

Available now with the C++23 update, the `print()` and `println()` functions are defined in the `<print>` header.

> **Note**
>
> ***Compiler Support***
>
> As of mid-2024, all major production compilers support the `print()` and `println()` functions. This code was tested on the current versions of Apple Clang, GCC, and MS Visual C++.

## How to do it

The `print()` function has finally been included with C++23, along with its sibling, `println()` which prints a line ending without requiring the string to include a *newline* `\n` character.

- Here's a simple example of the `print()` function in use:

```
#include <print>
#include <string>

using std::print;
using std::string;

int main() {
    string whoami {"I'm Spartacus"};
    print("Hello, World, {}\n", whoami);
}
```

The `using` statement brings the `print` symbol into the current namespace. I find this convenient with functions and objects that I use frequently.

The `print()` function uses a formatting string, which is passed to the `formatter` class introduced with C++20. This provides you with all the formatting features of the `format()` function.

- Alternatively, you can use `println()` to end the line, without including the `\n` character in your string:

```
println("Hello, World, {} ", whoami);
```

This usage is functionally equivalent to the previous example, but a newline `\n` character is automatically added to the end of the format string.

## How it works

The `print()` functions use the `formatter` class from the C++20 specification to expand any variables or positional arguments. It formats the output according to the options specified in the *format string*.

Because the `print()` functions bypass the cumbersome `iostream` library, these functions are far more efficient than using `format` with `cout`. Performance and code size are comparable to `printf()` but with the type-safety benefits of the `format` library.

These functions are intended to be used instead of `cout` for formatting and displaying text on the console or writing to a file stream. You should almost always favor using `print()` or `println()` where you would have previously used `cout`.

## See also...

A good reference for `format` string options is available at: `https://en.cppreference.com/w/cpp/utility/format/spec`

# Use synchronous generators

A *generator* is a type of iterator that generates a sequence of values. It's an extremely useful and powerful pattern. Until now, it's been common practice to use an iterator class to create a generator in C++. With C++23 we have a new `generator` class that uses *coroutines* to create a *synchronous* generator, which suspends execution, and resumes when called again. This makes generators far more convenient in C++.

The `generator` class is defined in the `<generator>` header.

**Note**

***Compiler Support***

As of late 2025, of the three major C++ compilers, only GCC and Microsoft Visual C++ support the `generator` specification. This feature is not yet supported on Apple Clang. This code was tested on GCC 16.2 and MSVC 19.44.

***Note***

When the `ranges` library is used in examples, this book uses *namespace aliases* for convenience and brevity:

```
namespace ranges = std::ranges;

namespace views = ranges::views;
```

This allows us to save trees by using `ranges::to` instead of `std::ranges::to` and `views::iota` instead of `std::ranges::views::iota`.

It's also quite a bit easier to read.

## How to do it

An iterator class is commonly used to provide a generator. However, it is now much simpler with C++23's new `generator` class.

- Here's an example that generates a sequence of prime numbers:

```
template<typename T>
std::generator<T> primes() {
    constexpr auto isprime = [](const T& n) {
        for (T i {2}; i * i <= n; ++i) {
            if (n % i == 0) return false;
        }
        return true;
    };
    for (T n : views::iota(T{2})) {
        if (!isprime(n)) continue;
        co_yield n;
    }
}
```

In this example, the `generator` class is used to generate a sequence of prime numbers. We use a template declaration to make it type-agnostic.

The `co_yield` statement (introduced with C++20 as part of the *coroutines* specification) returns the value from each iteration. The next time the generator is called, execution continues after the `co_yield` statement.

We use a lambda expression called `isprime()` to check if a value is prime. And the `iota` range factory is used to generate the sequential values.

- We can now call the `primes()` generator in a `for` loop, just like any iterator or range:

```
int main() {
    for (auto n : primes<uint64_t>() | views::take(25)) {
        print("{} ", n);
    }
    print("\n");
}
```

The `for` loop calls the `primes()` generator and pipes it to the `take()` view. This leverages the `ranges` library to limit our output to the first 25 iterations.

The output looks like this:

```
    2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

This gives us the first 25 prime numbers, as expected.

## How it works

The `generator` class creates a wrapper around a coroutine. When we create a `generator` object, we need to pass it a coroutine that produces the desired result:

```
std::generator<long> gen_fib() {
    long a{0};
    long b{1};

    while(true) {
        co_yield a;
        auto next = a + b;
        a = b;
        b = next;
```

```
    }
}
```

In this example, the generator is declared as type `long`. Within the curly braces, we declare a *coroutine* function. The `co_yield` statement suspends execution until the function is called again. This is what makes it a coroutine.

Notice that the `co_yield` statement is not the last line in the function. It is not an exit point. It simply returns the value to the caller, and when called again it picks up execution at the next statement.

We can call this with a `for` loop:

```
int main() {
    for (auto n : gen_fib() | views::take(10)) {
        print("{} ", n);
    }
    print("\n");
}
```

Our output is:

```
0 1 1 2 3 5 8 13 21 34
```

This gives us the first 10 values in the Fibonacci sequence.

The generator pattern is extremely valuable, and this new `generator` class makes it very simple to implement.

The `generator` class is defined in the `<generator>` header.

### See also...

See *Chapter 4*, *STL Compatible Iterators,* for further generator recipes.

## Use mdspan as a multidimensional array

Defined in C++23, the new `mdspan` class provides a multidimensional view of any contiguous container, including primitive C arrays.

The `mdspan` class uses the new C++23 multidimensional extension to the subscript operator `[]` to provide access to individual elements in a multidimensional space.

The multidimensional space is an abstraction superimposed over a contiguous container or primitive array.

Consider this primitive array:

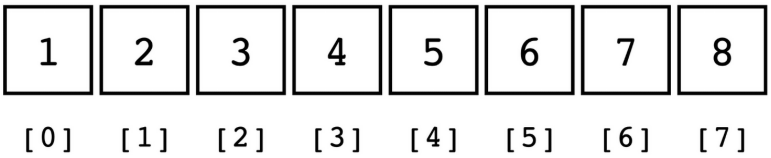



*Figure 1.1: A primitive array suitable for use with* `mdspan`

This array is a simple one-dimensional vector of integers, increasing in value from 1 to 8. Each element in the array is accessed using a scalar index, from `[0]` for the first element to `[7]` for the eighth element.

Using `mdspan`, we can view this exact same data as a two-dimensional array:

```
mdspan m(array, 2, 4);
```

Now we can access this same data using two-dimensional subscripts:

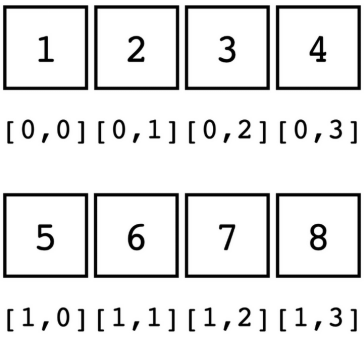



*Figure 1.2: A two-dimensional view using* `mdspan`

This gives us a two-dimensional view of our array. The first dimension exposes elements `[0]` to `[3]` as `m[0,0]` to `m[0,3]`, and the second exposes elements `[4]` to `[7]` as `m[1,0]` to `m[1,3]`.

The `mdspan` class is defined in the `<mdspan>` header.

**Note**

***Compiler support***

As of late 2025, `mdspan` is supported by the *Apple clang and Microsoft Visual C++* compilers. The code in this section has been tested on Apple clang 17.0 and MSVC 19.44.

## How to do it

The `mdspan` class requires a contiguous data set, such as a primitive array, for its source data. STL containers that conform to this requirement include `array` and `vector`.

- For this example, we'll use an STL `vector` object as our source data:

  ```
  vector v {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
  ```

  We can access the underlying data of a `vector` using its `data()` method. This provides the contiguous data required by `mdspan`:

  ```
  mdspan mds2(v.data(), 2, 6);
  ```

  The `mdspan` object is named `mds2` to indicate that it's a two-dimensional view. The first argument is a pointer to the contiguous array, as returned by the `data()` method of the `vector` object. The subsequent arguments are the *extents* of the view dimensions. Each extent specifies the number of elements in that dimension.

  In this example, we have two dimensions with extents of 2 and 6. The product of the extents must not exceed the number of underlying elements. Our source `vector` has 12 elements and the product of 2 × 6 is 12, so the product of our extents and the size of our source data are equal.

  This creates a two-dimensional `mdspan` object, with 2 elements in one dimension, and each of those elements has its own span of 6 elements.

- We can now view our data in two dimensions using the new multidimensional subscript operator:

  ```
  for (size_t i1 {0}; i1 < mds2.extent(0); ++i1) {
      for (size_t i2 {0}; i2 < mds2.extent(1); ++i2) {
          print("[{},{}] {:02d} ", i1, i2, mds2[i1,i2]);
      }
  ```

```
    println("");
}
```

The output looks like this:

```
[0,0] 01 [0,1] 02 [0,2] 03 [0,3] 04 [0,4] 05 [0,5] 06
[1,0] 07 [1,1] 08 [1,2] 09 [1,3] 10 [1,4] 11 [1,5] 12
```

We format the output to include the subscripts so we can clearly see the position of each element in the two-dimensional array. Each element's integer value is formatted with leading zeros, using the formatter specification `{:02d}`, so the values align visually.

- We can view this same data set in three dimensions by simply creating another `mdspan` object:

```
mdspan mds3(v.data(), 2, 3, 2);
```

For this object, We pass the same data and specify three extents. This creates a new three-dimensional view into the same source data.

- We can print the `mds3` view using the same technique we used for two dimensions:

```
for (size_t i1 {0}; i1 < mds3.extent(0); ++i1) {
    for (size_t i2 {0}; i2 < mds3.extent(1); ++i2) {
        for (size_t i3 {0}; i3 < mds3.extent(2); ++i3) {
            print("[{},{},{}] {:03d} ",
                  i1, i2, i3, mds3[i1,i2,i3]);
        }
        println("");
    }
}
```

That gives us this output:

```
[0,0,0] 001 [0,0,1] 002
[0,1,0] 003 [0,1,1] 004
[0,2,0] 005 [0,2,1] 006
[1,0,0] 007 [1,0,1] 008
[1,1,0] 009 [1,1,1] 010
[1,2,0] 011 [1,2,1] 012
```

Both views may coexist because the underlying data remains in its original location. The `mdspan` class merely provides a `view` into that data set.

## How it works

The constructor for the `mdspan` class looks like this:

```
template<class... OtherIndexTypes>
mdspan(data_handle_type p, OtherIndexTypes... exts);
```

The first argument is a pointer to a contiguous data source. The `exts` argument(s) is (are) *variadic template* argument(s) that take one or more *extents*, which specify the dimensions of the `mdspan` object.

The C++23 specification provides new multidimensional support for the subscript [ ] operator. This means that we may now use comma-separated values in the subscript operator.

The multidimensional subscript operator uses a variadic template to pass arguments as a parameter pack. We can demonstrate this capability with a simplified example of a class that overloads a multidimensional subscript operator:

```
template <typename T>
struct multidim {
    template <typename... I>
    const T operator[](I... indices) const {
        return (indices + ...);
    }
};
```

This example uses a fold expression to return the sum of the indices passed to the subscript [ ] operator. We can call it like this:

```
multidim<int> m1 {};
println("multidim[1,1,1,1,1]: {}", m1[1,1,1,1,1]);
println("multidim[5,6,7]: {}", m1[5,6,7]);
```

And we get this output:

```
multidim[1,1,1,1,1]: 5
multidim[5,6,7]: 18
```

The `mdspan` class uses this new feature to provide support for multiple indices into the data, like this:

```
x = o[1,2,3];
```

This allows the `mdspan` object to use the subscript operator to access values in multiple dimensions.

## There's more...

Because the `mdspan` class uses an external data store and does not own the underlying data itself, it returns data elements as references. This allows you to both read and write those elements.

That means that if we start with a simple ordinal data set:

```
vector v {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
```

We can modify that data set like this:

```
for (size_t i1 {0}; i1 < mds2.extent(0); ++i1) {
    for (size_t i2 {0}; i2 < mds2.extent(1); ++i2) {
        mds2[i1, i2] = mds2[i1, i2] * 10 + (int)i2;
    }
}
```

That modifies the data in the underlying `vector` object, so when we look at it:

```
for (auto n : v) {
    print("{} ", n);
}
```

We now get this output:

```
10 21 32 43 54 65 70 81 92 103 114 125
```

While it's important to understand that this allows the data to be mutable, it's not always a good idea. If you want to minimize the risk of side effects, you should qualify your source data as `const`:

```
const vector v {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
```

It's generally good practice to `const`-qualify any variables that you don't need to modify.

# Initialize containers from ranges

Beginning with C++23, we can now initialize a container directly from a `range` or `view`.

Until now, to initialize a container from a `range` we could use a `for` loop:

```
vector<int> v {};
for (auto n : views::iota(1, 11)) {
    v.push_back(n);
}
```

Or we could construct the `vector` from the `range` iterators:

```
auto r = views::iota(1, 11);
vector<int> v(r.begin(), r.end());
```

In C++23, this becomes much easier with the new `ranges::to` class:

```
auto v = views::iota(1, 11) | ranges::to<vector<int>>();
```

This new pattern is both simpler and clearer. The `ranges::to` class is defined in the `<ranges>` header.

## How to do it

The `ranges::to` class takes any `range` or `view` as input via the *pipe syntax* and returns a new non-`view` object.

- For convenience, we first declare namespace aliases for `ranges` and `views`:

  ```
  namespace ranges = std::ranges;
  namespace views = ranges::views;
  ```

- We start by populating a `vector` from a `view`:

  ```
  auto vec = views::iota(1, 11)
      | views::transform([](const auto n){ return n * 5; })
      | ranges::to<vector>();
  for (auto n : vec) print("{} ", n);
  print("\n");
  ```

In this example, `views::iota(1, 11)` generates a sequence of integers from `1` to `10`. We then use `views::transform` with a *lambda function* to multiply each integer by `5`. Finally, `ranges::to<vector>()` returns a new `vector` for our initialization.

The function call operator (parentheses) after `ranges::to<>` is required for use with the pipe syntax:

```
auto vec = r | ranges::to<vector>;   // syntax error
auto vec = r | ranges::to<vector>(); // correct usage
```

This produces the output:

```
5 10 15 20 25 30 35 40 45 50
```

- You may use this technique to initialize any sequential container:

```
auto lst = vec | views::reverse
    | ranges::to<list<double>>();
for (auto d : lst) print("{} ", d);
print("\n");
```

In this example, we use `views::reverse` to reverse the order of the sequence, and `ranges::to<list<double>>()` to construct a `list` of `double`, which produces the output:

```
50 45 40 35 30 25 20 15 10 5
```

Let's take a look at how this works.

## How it works

The `ranges::to` class is designed to work in a variety of contexts. To return a container `C` from range `R`, it considers a series of scenarios and chooses the first valid method from the list:

- Construct `C` from `R`
- Construct `C` from `R` using a tagged range constructor (`from_range_t`)
- Construct `C` from the range iterators `R.begin()`, `R.end()`

- Construct `C`, then insert each element of `R` at the end of `C`
- If `C` is a `range` whose value type is itself a `range` (and is not a `view`), and `R`'s value type is also a `range`, for each element of `R`, a value of `to<range_value_t>` is inserted at the end of `C`

This allows a more unified and concise syntax for constructing containers from ranges.

# Find content in strings and ranges with contains()

C++23 provides a new strategy for searching within a `string`, `string_view`, or `range` object. The new `string::contains()` and `string_view::contains()` methods check whether a `string` or `string_view` contains a given substring. The `ranges::contains()` algorithm looks for an element or sub-range in a given `range`. These functions return a Boolean `true` or `false` value to indicate if the provided value(s) are contained in the target.

This new method of searching a container is simpler and more convenient than using the `find()` method. The `find()` method returns a position, or the special `string::npos` value to indicate that the substring was not found. That means that if we just want to know if a string is contained within another string, we need to compare against the `npos` value like this:

```
string s {"https://example.com"};
if (s.find("https://") != string::npos) {
    println("good!");
}
```

For this purpose, the `contains()` method is both simpler and more readable:

```
string s {"https://example.com"};
if (s.contains("https://")) {
    println("good!");
}
```

The same is true for the `ranges::contains()` algorithm:

```
const vector<int> v {1, 2, 3, 4, 5};
if (ranges::contains(v, 3)) {
    println("got it!");
}
```

This provides a concise solution for a simple search.

The `ranges::contains()` algorithm is defined in the `<algorithm>` header.

## How to do it

The `string::contains()` method provides a simple `true` or `false` result when searching a `string` object for a substring.

- We can search for content in a `string`:

```
const string s {"Big Light In Sky Appears In East"};
if (s.contains("Sky")) {
    println("found");
}
```

- The `string_view` class also has a `contains()` method:

```
const string_view sv {"Why is abbreviated such a long word?"};
if (sv.contains("word")) {
    println("found");
}
```

- Or we can search for a single character:

```
const string s2 {"abcdefg"};
if (s2.contains('f')) {
    println("found");
}
```

- We can search for an element in a `range` using `ranges::contains()`:

```
const auto r = views::iota(1,11);
if (ranges::contains(r, 7)) {
    println("found");
}
```

- Or we can search for a sub-range within a range, using `ranges::contains_subrange()`:

```
const std::vector<int> v1 {1, 2, 3, 4, 5};
const std::vector<int> v2 {2, 3, 4};
if (ranges::contains_subrange(v1, v2)) {
```

```
        println("found");
    }
```

These new C++23 features make searching within containers more concise and readable.

## How it works

The `string::contains()` function is a method of the `string` class. This means that it is called on the `string` object with the member operator:

```
if (s.contains("Sky")) {
    ...
}
```

The function signatures for `string::contains()` look like this:

```
bool contains( std::string_view sv ) const noexcept;
bool contains( char c ) const noexcept;
bool contains( char* s ) const;
```

This provides overloads for passing different types to the function. The first overload takes a `std::string_view` object. When we pass a `std::string` object it gets promoted to a `string_view` for simple access to the raw underlying C-string data.

The second overload takes a `char` type. This allows us to search for an individual character in a string:

```
if (s.contains('r')) {
    println("char 'r' found");
}
```

The third overload takes a primitive C-string:

```
if (s.contains("Sky")) {
    println("c-string \"Sky\" found");
}
```

This overload prevents the C-string from being promoted to a more complex `std::string` object.

The `ranges::contains()` algorithm is found in the `<algorithm>` header. This is a stand-alone template function, not a class member function. The function signatures look like this:

```
template< input_iterator I, sentinel_for<I> S, class T,
    class Proj = std::identity >
constexpr bool contains( I first, S last, const T& value,
    Proj proj = {} );

template< input_range R, class T, class Proj = identity >
constexpr bool contains( R&& r, const T& value,
    Proj proj = {} );
```

This function provides two overloads. The first uses iterators for the source container. The second takes a `range` object. We will consider the `range` overload:

```
constexpr bool contains( R&& r, const T& value,
    Proj proj = {} );
```

We use this as we would any algorithm in the `ranges` library:

```
const auto r = views::iota(1,11);
if (ranges::contains(r, 7)) {
    println("found");
}
```

This allows us to succinctly search for a value in a range.

The third argument `proj` is a *projection*. This allows us to use a lambda expression to modify the values from the range `r` before the comparison is performed:

```
const auto r = views::iota(1, 11);
if (ranges::contains(r, 14, [](auto n){return n * 2;})) {
    println("found");
}
```

Using this simple lambda, we double each value in the range `r` before comparing it. Now the value `7` will match the value `14` in the `ranges::contains` call.

## There's more...

The `ranges::contains_subrange()` algorithm may be used to test for a sub-range within a range. Its function signatures specify two ranges:

```
template< forward_iterator I1, sentinel_for<I1> S1,
    forward_iterator I2, sentinel_for<I2> S2,
    class Pred, class Proj1, class Proj2 >
constexpr bool contains_subrange( I1 first1, S1 last1, I2 first2,
    S2 last2, Pred pred = {},
    Proj1 proj1 = {}, Proj2 proj2 = {} );

template< forward_range R1, forward_range R2,
    class Pred, class Proj1, class Proj2 >
constexpr bool contains_subrange( R1&& r1, R2&& r2,
    Pred pred = {}, Proj1 proj1 = {}, Proj2 proj2 = {} );
```

The two ranges `r1` and `r2` are often referred to as the *needle* and the *haystack*, where we can think of the function as searching for a needle in a haystack. The range `r1` is the haystack and `r2` is the needle. Likewise, the projections `proj1` and `proj2` operate on the haystack and the needle, respectively.

This allows us to search for one range within another:

```
const std::vector<int> v1 {1, 2, 3, 4, 5};
const std::vector<int> v2 {2, 3, 4};
if (ranges::contains_subrange(v1, v2)) {
    println("found");
}
```

This works just like `ranges::contains`, but searches for a range rather than a single value.

The `contains_subrange()` algorithm also takes a predicate, which can be used as an alternative comparison function, just like other functions in the `algorithm` library.

Using the two projections, we can modify values before we compare them:

```
const std::vector<int> v3 {10, 20, 30, 40, 50};
const std::vector<int> v4 {2, 3, 4};
if (ranges::contains_subrange(v3, v4, {},
        [](const auto& n){return n / 5;},
        [](const auto& n){return n * 2;})) {
```

```
    println("found");
}
```

In this example, we leave the comparison predicate empty with `{}` and we provide two projections. The first projection modifies values read from the haystack `v3`, and the second projection modifies values read from the needle `v4`.

While there are many options available, in practice you'll most likely use the simplest forms which provide concise syntax for finding a substring within a string, or an object within a sequence.

## Subscribe to Deep Engineering

Join thousands of developers and architects who want to understand how software is changing, deepen their expertise, and build systems that last.

Deep Engineering is a weekly expert-led newsletter for experienced practitioners, featuring original analysis, technical interviews, and curated insights on architecture, system design, and modern programming practice.

Scan the QR or visit the link to subscribe for free.



`https://packt.link/deep-engineering-newsletter`

# 2

# Best Practices

Modern C++ is powerful and versatile, and the STL draws heavily upon those qualities. The best practices in this chapter will help you avoid common pitfalls and improve the clarity, maintainability, and performance of your code:

- Use braced initialization
- Use span to make c-arrays safer
- Use structured binding to return multiple values
- Initialize variables within if and switch conditions
- Use template argument deduction for simplicity and clarity
- Use RAII to preserve memory integrity

In this chapter, we'll explore some simple strategies to ensure our code is elegant, stable, and scalable.

## Use braced initialization

One of the more significant differences between C and C++ is the distinction between object construction and initialization. Because C++ has classes, object construction and initialization are necessarily more complex than in C, which does not have classes. As we dive more deeply into the STL, it is vitally important to understand this distinction and to write code that expresses it clearly.

Source code is the bridge between the programmer and the compiler. That makes it necessary to write code that is both effective and readable. If we fail to understand how our objects are constructed, from both classes and primitive types, we invite confusion, which can lead to maddeningly elusive bugs.

## How to do it

Modern C++ provides several ways to initialize objects. The two most common are *copy initialization* and *braced initialization*:

```
std::string name = "Jimi Hendrix";  // copy initialization
std::string name {"Jimi Hendrix"};  // braced initialization
```

Copy initialization uses the assignment operator =. Traditionally, we think of this as an *assignment operator*, but in this context, it serves as a *converting constructor*. This misunderstanding can lead to unintended type conversions with loss of data, which can be extremely difficult to debug.

Braced initialization uses the *list initialization operator* `{}` (introduced in C++11) to avoid those side effects. It's a good habit to get into and you'll see it a lot in this book.

It's also worth noting that the special case of `T{}` is guaranteed to be zero-initialized:

```
T x;      // uninitialized            bad  :(
T x = 0;  // zero (copy initialized)  good :)
T x {};   // zero (zero-initialized)  best :D
```

Zero initialization with empty braces offers a useful shortcut for initializing new variables.

## How it works

Let's start by looking at some primitive initializations. Here's a simple `int` initialization:

```
int x {};
print("x is {}\n", x);
```

The empty braces `{}` are a special case which are guaranteed to initialize the `int` to zero.

The output is as expected:

```
x is 0
```

Now, let's consider this traditional initialization:

```
int x = 42;
```

This uses the = (assignment) operator to assign the value 42 to the int x, which gives us the result:

```
x is 42
```

Now let's assign an incompatible type:

```
int x = 73.9;
```

This gives us no error and silently results in a wrong value:

```
x is 73
```

This effect is called a *narrowing conversion*. The literal `73.9` is a `double`, which is incompatible with type `int`. Under the legacy rules of copy initialization, the compiler narrows the value by quietly discarding the fractional part. While usually an undesirable effect, this behavior is retained in modern C++ for compatibility with legacy code.

Braced initialization doesn't allow narrowing. If we use braced initialization with an incompatible type, we get an error:

```
int x {73.9};
```

Now we get this error from GCC:

```
error: narrowing conversion of '7.3900000000000006e+1' from 'double' to 'int' [-
Wnarrowing]
```

(Error messages may differ with different compilers.)

Using braced initialization, incompatible initialization is not allowed. Braced initialization is simpler, safer, and the rules are more general and less ambiguous.

## There's more...

To better understand the rules of initialization in C++, let's illustrate with a simple `vector`. First, we'll need a function to print the values of a `vector`:

```
void pvec(const auto& v) {
    if (v.empty()) {
        println("[empty]");
    } else {
```

```
        for (const auto& i : v) print("{} ", i);
        println("");
    }
}
```

This is an *abbreviated function template* that prints the values of a `vector` (or any compatible container) using the C++23 `println()` function.

Now we initialize our `vector`. We'll start with an empty set of braces:

```
vector<int> v {};
pvec(v);
```

This gives us the result:

```
[empty]
```

The empty braces zero-initialize the `vector` according to the rules of the `std::vector` class.

We can use a comma-separated list of values within the braces:

```
vector<int> v {1, 2, 3, 4, 5};
```

The result is as expected:

```
1 2 3 4 5
```

The comma-separated list in braces is interpreted as a `std::initializer_list` object. The `vector` constructor uses it to create a `vector` object with the specified values.

Now, consider the case of two values in braces:

```
vector<int> v {5, 5};
```

This gives us the result:

```
5 5
```

This is still interpreted as an `initializer_list`. But if we use parentheses instead of braces:

```
vector<int> v (5, 5);
```

Our result is very different:

```
5 5 5 5 5
```

It is vitally important to understand this distinction.

In the context of initializing a class, the list of values in braces `{}` is a literal `initializer_list` object, which invokes an `initializer_list` constructor, that is, a constructor which takes an `initializer_list` object as a parameter.

On the other hand, the values in parentheses are treated as parameters for a different constructor. In the case of the `vector` class, it calls a constructor with this signature:

```
vector( size_type count, const T& value );
```

The first parameter is a count, and the second is a value. This constructs a `vector` object with five integer `5`'s.

For this reason, it's generally a bad idea to initialize values with parentheses. While this may work as expected:

```
int x(5);
```

It does not translate to class constructors, and in many contexts, it will be parsed as a function call. On the other hand, the braced initialization

```
int x{5};
```

will work in all contexts. It's a good habit to get into and we'll use it throughout this book.

**Note**

***Initializing auto variables***

One notable exception to the rule is the `auto` type declaration.

Because `auto` uses *type deduction*, it cannot always distinguish between a literal `initializer_list` and braced initialization.

Prior to C++17, If braces were used with an `auto` type declaration, it would create an `initializer_list` object:

```
auto x {42}; // C++11-14 creates an initializer_list
```

> To avoid ambiguity, and for compatibility with older standards, unless we mean to create an `initializer_list` object, it's best practice to use the assignment operator = when declaring an `auto` variable:
>
> `auto x = 42; // for auto declarations`

# Use span to make C-arrays safer

A C-array is a primitive structure for storing a contiguous sequence of scalar data. As a simple primitive, it has no bounds checking. The `std::span` class is a simple wrapper that creates a `view` over a contiguous sequence of objects. A `span` object is inherently safer than a C-array as it carries size information which can be used for bounds checking of the underlying data. The `span` doesn't own any of its own data. It refers to the data in the underlying structure, which may be a C-array, a `vector`, or an STL `array`. Think of it as `string_view` for arrays.

## How to do it

We can create a span from any compatible contiguous-storage structure. The most common use case will involve a C-array. For example, if we pass a C-array directly to a function, the array is passed as a pointer, and the function has no easy way to know the size of the array:

```
void parray(int* a);  // loses size information
```

If we define the function with a `span` parameter, we can still pass the C-array and it will be promoted to `span`. Here's a template function that takes a `span` and prints out the size in elements and in bytes:

```
template<typename T>
void pspan(span<T> s) {
    print("number of elements: {}\n", s.size());
    print("size of span: {}\n", s.size_bytes());
    for(auto e : s) print("{} ", e);
    print("\n");
}
```

We can pass a C-array to this function and it's automatically promoted to `span`:

```
int main() {
    int carray[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

```
    pspan<int>(carray);
}
```

Output:

```
number of elements: 10
number of bytes: 40
1 2 3 4 5 6 7 8 9 10
```

The purpose of `span` is to encapsulate raw data of a contiguous array to provide a measure of safety and utility, with a minimum of overhead.

## How it works

The `span` class itself doesn't own any data. The data belongs to the underlying data structure. The `span` is essentially a view over the underlying data. It also provides some useful member functions.

Defined in the `<span>` header, the `span` class looks something like this:

```
template<typename T, size_t Extent = std::dynamic_extent>
class span {
    T * data;
    size_t count;
public:
    ...
};
```

The `Extent` template parameter is a `constexpr` of type `size_t`, which is computed at compile time. It's either the number of elements in the underlying data or the default `std::dynamic_extent` constant, which indicates that the size is variable. This allows `span` to use an underlying structure like a `vector`, which may not always be the same size.

All member functions are `constexpr` and `const` qualified. Member functions include:

| Public member function | Return value |
|---|---|
| `T& front()` | The first element |
| `T& back()` | The last element |
| `T& operator[]` | An indexed element |

| **Public member function** | **Return value** |
|---|---|
| `T* data()` | A pointer to the beginning of the sequence |
| `iterator begin()` | An iterator to the first element |
| `iterator end()` | An iterator following the last element |
| `iterator rbegin()` | A reverse iterator to the first element |
| `iterator rend()` | A reverse iterator following the last element |
| `constexpr iterator cbegin()` | A `const`-qualified iterator to the first element (new for C++23) |
| `constexpr iterator cend()` | A `const`-qualified iterator following the last element (new for C++23) |
| `constexpr iterator crbegin()` | A `const`-qualified reverse iterator to the first element (new for C++23) |
| `constexpr iterator crend()` | A `const`-qualified reverse iterator following the last element (new for C++23) |
| `size_t size()` | The number of elements in the sequence |
| `size_t size_bytes()` | The number of bytes in the sequence |
| `bool empty()` | True if empty |
| `span<T> first<count>()`<br>`span<T> first(count)` | A sub-span consisting of the first count elements of the sequence |
| `span<T> last<count>()`<br>`span<T> last(count)` | Returns a sub-span of the last count elements |
| `span<T> subspan(offset, count)` | Returns a sub-span consisting of count elements beginning at offset |

*Table 2.1:* `span` *class member functions*

**Note**

The `span` class `[ ]` operator performs no bounds checking. If we try to access element `n+1` in a `span` of `n` elements, the result is *undefined*, which is tech for, "Bad. Don't do that."

# Use structured binding to return multiple values

*Structured binding* makes it easy to unpack the values of a structure into separate variables, improving both the readability and safety of our code.

With structured binding we can directly assign member values to variables like this:

```
std::pair<int,int> things_pair {47, 9};
auto [thing1, thing2] = things_pair;
println("{} {}", thing1, thing2);
```

Output:

```
47 9
```

## How to do it

*Structured binding* works with aggregated values, such as `pair`, `tuple`, `array`, and `struct`, or with your own custom types. Beginning with C++20, this also works with bit-fields.

- This example uses a C array:

  ```
  int nums[] { 1, 2, 3, 4, 5 };
  auto [ a, b, c, d, e ] = nums;
  println("{} {} {} {} {}", a, b, c, d, e);
  ```

  Output:

  ```
  1 2 3 4 5
  ```

- Because structured binding uses *automatic type deduction*, its type must be `auto`. The names of the individual variables are within the square brackets:

  ```
  auto [ a, b, c, d, e ] = x;
  ```

For example, consider a C-array with five `int` values. These five values are assigned to the variables (`a`, `b`, `c`, `d`, and `e`) using structured binding.

```
int carray[] {1, 2, 3, 4, 5};
auto [ a, b, c, d, e ] = carray;
println("{} {} {} {} {}", a, b, c, d, e);
```

Output:

```
1 2 3 4 5
```

- This also works with an STL `array` object:

```
array<int,5> nums { 1, 2, 3, 4, 5 };
auto [ a, b, c, d, e ] = nums;
println("{} {} {} {} {}", a, b, c, d, e);
```

Output:

```
1 2 3 4 5
```

- Or we can use it with a `tuple`:

```
tuple<int, double, string> nums {1, 2.7, "three"};
auto [ a, b, c ] = nums;
println("{} {} {}", a, b, c);
```

Output:

```
1 2.7 three
```

- When we use it with a `struct` it will take the variables in the order they're defined:

```
struct Things { int i{}; double d{}; string s{}; };
Things nums {1, 2.7, "three"};
auto [ a, b, c ] = nums;
println("{} {} {}", a, b, c);
```

Output:

```
1 2.7 three
```

- We can use a reference with a structured binding, which allows us to modify the values in the bound container, while avoiding duplication of the data:

```
array<int,5> nums {1, 2, 3, 4, 5};
auto& [ a, b, c, d, e ] = nums;
println("{} {}", nums[2], c);
c = 47;
println("{} {}", nums[2], c);
```

Output:

```
3 3
47 47
```

- Because the variables are bound as a reference, we can assign a value to `c` and it will change the value in the array as well (`nums[2]`)
We can declare the array `const` to prevent values from being changed:

```
const array<int,5> nums { 1, 2, 3, 4, 5 };
auto& [ a, b, c, d, e ] = nums;
c = 47;    // this is now an error
```

Or we can declare the binding `const` for the same effect, while allowing the array to be changed elsewhere and still avoid copying data:

```
array<int,5> nums { 1, 2, 3, 4, 5 };
const auto& [ a, b, c, d, e ] = nums;
c = 47;    // this is also an error
```

## How it works

Structured binding uses *automatic type deduction* to unpack the structure into our variables. It determines the type of each value independently and assigns a corresponding type to each variable.

Because structured binding uses *automatic type deduction*, we cannot specify a type for the binding. We must use `auto`. We should get a reasonable error message if we try to use a type for the binding, for example:

```
array<int,5> nums { 1, 2, 3, 4, 5 };
int [ a, b, c, d, e ] = nums;
```

Output:

```
error: structured binding declaration cannot have type 'int'
note: type must be cv-qualified 'auto' or reference to cv-qualified 'auto'
```

Above is the error from GCC when I try to use `int` with the structured binding declaration.

It's common to use structured binding for the return type from a function:

```
struct div_result {
    long quo;
    long rem;
};
div_result int_div(const long & num, const long & denom) {
    struct div_result r{};
    r.quo = num / denom;
    r.rem = num % denom;
    return r;
}
int main() {
    auto [quo, rem] = int_div(47, 5);
    println("quotient: {}, remainder {}",
      quo, rem);
}
```

Output:

```
quotient: 9, remainder 2
```

Because the `map` container classes return a `pair` for each element, it can be convenient to use structured binding to retrieve key/value pairs:

```
map<string, uint64_t> inhabitants {
    { "humans",   7000000000 },
```

```
    { "pokemon", 17863376 },
    { "klingons",   24246291 },
    { "cats",    1086881528 }
};
// I like commas
string make_commas(const uint64_t num) {
    string s = std::to_string(num);
    for (auto i = s.size(); i > 3; i -= 3) {
        s.insert(i - 3, ",");
    }
    return s;
}
int main() {
    for(const auto & [creature, pop] : inhabitants) {
        println("there are {} {}",
            make_commas(pop), creature);
    }
}
```

Output:

```
there are 1,086,881,528 cats
there are 7,000,000,000 humans
there are 24,246,291 klingons
there are 17,863,376 pokemon
```

Using structured binding to unpack structures should make our code clearer and easier to maintain.

# Initialize variables within if and switch conditionals

Beginning with C++17, `if` and `switch` have initialization syntax, much like the `for` loop has had since C99. This allows us to limit the scope of variables to within the condition.

## How to do it

You may be accustomed to code like this:

```
const string artist {"Jimi Hendrix"};
size_t pos {artist.find("Jimi")};
if(pos != string::npos) {
    println("found");
```

```
} else {
    println("not found");
}
```

This leaves the variable pos exposed outside the scope of the conditional statement, where it needs to be managed to mitigate collisions with symbols in the rest of the code.

Now we can put the initialization expression inside the `if` condition:

```
if(size_t pos {artist.find("Jimi")}; pos != string::npos) {
    println("found");
} else {
    println("not found");
}
```

This confines the scope of the pos variable to the scope of the conditional, which keeps our namespace clean and manageable.

## How it works

The initializer expression can be used in either `if` or `switch` statements. Here are some examples of each.

- Use an initializer expression with an `if` statement:

  ```
  if (auto var {init_value}; condition) {
      // var is visible
  } else {
      // var is visible
  }
  // var is NOT visible
  ```

  The variable defined in the initializer expression is visible within the scope of the entire `if` statement, including the `else` clause. Once control flows out of the `if` statement scope, the variable will no longer be visible, and any relevant destructors will be called.

- Use an initializer expression with a `switch` statement:

  ```
  switch (auto var {init_value}; var) {
  case 1: ...
  case 2: ...
  case 3: ...
  ```

```
...
default: ...
}
// var is NOT visible
```

The variable defined in the initializer expression is visible within the scope of the entire `switch` statement, including all the `case` clauses and the `default` clause, if included. Once control flows out of the `switch` statement scope, the variable will no longer be visible, and any relevant destructors will be called.

## There's more...

One interesting use case is to limit the scope of a `lock_guard` that's locking a mutex. This becomes simple with an initializer expression:

```
if (lock_guard<mutex> lg {my_mutex}; condition) {
    // interesting things happen here
}
```

The `lock_guard` locks the mutex in its constructor and unlocks it in its destructor. Now the `lock_guard` will be automatically destroyed when it runs out of the scope of the `if` statement. In the past we would have had to delete it or enclose the whole `if` statement in an extra block of braces.

Another use case could be using a legacy interface that has output parameters, like this one from *SQLite*:

```
if (sqlite3_stmt** stmt,
    auto rc = sqlite3_prepare_v2(db, sql, -1, &_stmt, nullptr); !rc) {
          // do SQL things
} else {  // handle the error
    // use the error code
    return 0;
}
```

Here I can keep the statement handle `stmt` and the error code `rc` localized to the scope of the `if` statement. Otherwise, I would need to manage those objects globally.

Using initializer expressions will help keep our code tight and uncluttered, more compact, and easier to read. Refactoring and managing our code will also become easier.

# Use template argument deduction for simplicity and clarity

Template argument deduction occurs when template types are clear enough to be understood by the compiler without the use of template arguments. There are certain rules to this feature, but it's mostly intuitive.

## How to do it

Template argument deduction happens automatically when we use a template with clearly compatible arguments. Let's consider some examples.

- In a function template, argument deduction usually looks something like this:

```
template<typename T>
const char * f(const T a) {
    return typeid(T).name();
}
int main() {
    println("T is {} ", f(47));
    println("T is {}", f(47L));
    println("T is {}", f(47.0));
    println("T is {}", f("47"));
    println("T is {}", f("47"s));
}
```

Output:

```
T is int
T is long
T is double
T is char const *
T is class std::basic_string<char...
```

Because the types are easily discernible, there is no reason to specify a template parameter like `f<int>(47)` in the function call. The compiler can deduce the `<int>` type from the argument. The simpler function calls are intuitive and easy to read.

> **Note**
>
> The above output shows meaningful type names where most compilers will use shorthand, like `i` for `int` and `PKc` for `const char*`, and so on.

- This works just as well for multiple template parameters:

```
template<typename T1, typename T2>
string f(const T1 a, const T2 b) {
    return format("{} {}", typeid(T1).name(),
        typeid(T2).name());
}
int main() {
    println("T1 T2: {}", f(47, 47L));
    println("T1 T2: {}", f(47L, 47.0));
    println("T1 T2: {}", f(47.0, "47"));
}
```

Output:

```
T1 T2: int long
T1 T2: long double
T1 T2: double char const *
```

Here the compiler is deducing types for both `T1` and `T2`.

- Beginning with C++17, we can also use template parameter deduction with classes. So this will work:

```
pair p(47, 47.0);     // deduces to pair<int, double>
tuple t(9, 17, 2.5);  // deduces to tuple<int, int, double>
```

This eliminates the need for `std::make_pair()` and `std::make_tuple()` as we can now initialize these classes directly without the explicit template parameters. The `std::make_*` helper functions remain available for backward compatibility.

## How it works

Let's define a class so we can see how this works:

```
template<typename T1, typename T2, typename T3>
class Thing {
    T1 v1{};
    T2 v2{};
    T3 v3{};
public:
    explicit Thing(T1 p1, T2 p2, T3 p3)
    : v1{p1}, v2{p2}, v3{p3} {}
    string s() {
        return format("{}, {}, {}\n",
            typeid(v1).name(),
            typeid(v2).name(),
            typeid(v3).name()
        );
    }
};
```

This is a template class with three types and three corresponding data members. It has a function `s()` which returns a formatted string with the three type names.

Without template parameter deduction, I would have to instantiate an object of this type like this:

```
Thing<int, double, string> thing1{1, 47.0, "three" }
```

Now I can do it more like this:

```
Thing thing1{1, 47.0, "three" }
```

This is both simpler and more reliable.

When I call the `s()` method on the `thing1` object, I get this result:

```
println("thing1: {}", thing1.s());
```

Output:

```
thing1: int, double, char const *
```

Of course, your compiler may report something different but equivalent.

Before C++17, template parameter deduction didn't apply to classes, so we needed a helper function, which may have looked like this:

```
template<typename T1, typename T2, typename T3>
Thing<T1, T2, T3> make_thing(T1 p1, T2 p2, T3 p3) {
    return Things<T1, T2, T3>(p1, p2, p3);
}
...
auto thing2(make_thing(1, 47.0, "three"));
println("thing2: {}", thing2.s());
```

Output:

```
thing2 int, double, char const *
```

The STL includes a few of these helper functions, like `make_pair()` and `make_tuple()`, etc. These are now obsolescent but will be maintained for compatibility with older code.

## There's more...

Consider the case of a constructor with a parameter pack:

```
template <typename T>
class Sum {
    T v{};
public:
    template <typename... Ts>
    Sum(Ts&& ... values) : v {(values + ...)} {}
    const T& value() const { return v; }
};
```

Notice the *fold expression* in the constructor `(values + ...)`. This is a C++17 feature that applies an operator to all the members of a parameter pack. In this case, it initializes `v` to the sum of the parameter pack.

The constructor for this class accepts an arbitrary number of parameters, where each parameter may be a different class. For example, we could call it like this:

```
Sum s1 { 1u, 2.0, 3, 4.0f };  // unsigned, double, int,
                              // float
Sum s2 { "abc"s, "def" };     // std::string, c-string
```

This, of course, doesn't compile. The template argument deduction fails to find a common type for all those different parameters. We get an error message to the effect of:

```
cannot deduce template arguments for 'Sum'
```

We can fix this with a *template deduction guide*. A deduction guide is a helper pattern to assist the compiler with a complex deduction. Here's a guide for our constructor:

```
template <typename... Ts>
Sum(Ts&& ... ts) -> Sum<std::common_type_t<Ts...>>;
```

This tells the compiler to use the `std::common_type_t` trait, which attempts to find a common type for all the parameters in the pack. Now our argument deduction works, and we can see what types it settled on:

```
Sum s1 { 1u, 2.0, 3, 4.0f };  // unsigned, double, int,
                              // float
Sum s2 { "abc"s, "def" };     // std::sring, c-string
auto v1 = s1.value();
auto v2 = s2.value();
println("s1 is {} {}, s2 is {} {}",
        typeid(v1).name(), v1, typeid(v2).name(), v2);
```

Output:

```
s1 is double 10, s2 is class std::string abcdef
```

# Use RAII to preserve memory integrity

The term, *Resource Acquisition Is Initialization* (RAII), refers to a code design pattern where each class is responsible for managing its own resources, often memory resources.

Resource leaks are a common source of both bugs and security breaches. By allocating resources during initialization (object construction), we guarantee that those resources are

completely under the control of the class. And by freeing those resources during object destruction, we guarantee that those resources are properly released. This prevents a dominant source of resource leaks and cybersecurity attacks.

RAII in C++ relies on the fact that class destructors are always called when objects go out of scope. For example, here's a class which prints a message when it's constructed and destroyed:

```
class C {
    int i {};
public:
    C() { println("default constructor"); }
    ~C() { println("destructor"); }
};
```

This function creates an object that goes out of scope when the function returns:

```
void f1() {
    C a {};
    println("this is function f1()");
}
```

When we call this function, we get constructor and destructor messages:

```
int main() {
    println("call f1()");
    f1();
    println("return from f1()");
}
```

This gives us the output:

```
call f1()
default constructor
this is function f1()
destructor
return from f1()
```

This example shows the object being constructed in function `f1()` and then calling its destructor upon going out of scope, when function `f1()` completes its execution.

To implement RAII, we leverage this behavior to ensure that any resources acquired (initialized) by an object are automatically released in its destructor.

It's worth noting that, while we may associate resource leakage with memory leaks, this also applies to files, sockets, databases, locks, and other resources that require management.

## How to do it

A common example of an RAII class is a smart pointer. When we create a smart pointer, it allocates memory and returns a handle for access:

```
std::unique_ptr<int> p = std::make_unique<int>();
```

The `std::make_unique` function constructs a `std::unique_ptr` object `p`. The `unique_ptr` constructor allocates memory for an object of type `int`. When `p` goes out of scope, its destructor frees up the memory allocated by its constructor.

Let's start with a rudimentary unique pointer class:

```
template <typename T>
class uptr {
    T* m{};
public:
    uptr() {
        println("uptr constructor");
        m = new T;
    }
    ~uptr() {
        println("uptr destructor");
        delete m;
    }
    T& operator*() const {
        return *m;
    }
    T& operator*(const T& v) {
        *m = v;
        return *m;
    }
};
```

Our class is called `uptr`. The constructor allocates memory for an object of type `T` using the `new` operator. The destructor uses `delete` to free up that memory and return it to the pool. The constructor and destructor print messages indicating when the object is created and destroyed.

We can now create an object of type `uptr` in a function with limited scope:

```
void f2() {
    uptr<int> p {};
    *p = 42;
    println("p is {}", *p);
}
```

When we call this function, we can see how the smart pointer is created and destroyed:

```
int main() {
    println("calling f2()");
    f2();
    println("returned from f2()");
}
```

Our output:

```
calling f2()
uptr constructor
p is 42
uptr destructor
returned from f2()
```

The constructor is called when the object is created in `f2()`, and the destructor is called when we return from `f2()`. This ensures that the associated memory is always managed properly without intervention.

## How it works

Whenever an object is created, its class *constructor* is invoked. When the object goes out of scope, it is automatically destroyed and its *destructor* is invoked. By allocating resources in the constructor and freeing those resources in the destructor, we mitigate the need to manage those resources manually. This sequence of events provides the framework for RAII.

An object cannot be created in C++ without calling a constructor. We can ensure that our resources are properly initialized by always specifying a constructor:

```
class uptr {
    T* m{};
public:
    uptr() {        // default constructor
```

```
            m = new T;
        }
    };
```

C++ also guarantees that the destructor is called upon the destruction of any object. By specifying a destructor, we may complete the RAII pattern and guarantee that our resources are properly released:

```
    ~uptr() {       // destructor
        delete m;
    }
```

A class may have many constructors but only one destructor. It is important to ensure that our data is always properly initialized in all of our constructors:

```
    uptr(const T& v) {
        println("uptr(value) constructor");
        m = new T;
        *m = v;
    }
```

This maintains the integrity of the RAII pattern and its protections.

## Get this book's PDF version and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.



UNLOCK NOW

*Note: Keep your invoice handy. Purchases made directly from Packt don't require an invoice.*

# 3

# STL Containers

In this chapter, we will focus on the container classes in the STL. In short, a *container* is an object that contains a collection of other objects, or *elements*. The STL provides a complete suite of container types that form the foundation of the STL itself.

In this chapter we will cover the following recipes:

- Use uniform erasure functions to delete elements from a container
- Delete items from an unsorted `vector`
- Access `vector` elements safely
- Keep `vector` elements sorted
- Efficiently insert elements into a `map`
- Efficiently modify keys in a `map`
- Use `unordered_map` with custom keys
- Sort and filter user input with set
- Create a simple RPN calculator with `deque`
- Count word frequency with `map`
- Find long sentences with a vector of vectors
- Create an efficient ToDo list using a multimap

The heart of the STL is in its container classes. The STL leverages the power of Templates (the T in STL) to provide containers that work transparently with a broad spectrum of classes and types.

# A quick overview of the STL container types

The STL provides a comprehensive set of container types, including *sequential containers*, *associative containers*, and *container adapters*. Here's a brief overview.

## Sequential containers

Sequential containers provide an interface where the elements are arranged in sequence. While you may use the elements sequentially, some of these containers use contiguous storage and others do not. The STL includes these sequential containers:

- The `array` is a fixed-size sequence that holds a specific number of elements in contiguous storage. Once allocated, it cannot change size. This is the simplest and fastest contiguous storage container.
- The `vector` is like an array, but a `vector` can shrink and grow. Its elements are stored contiguously, so changing its size may involve the expense of allocating memory and moving data. A `vector` may keep extra space in reserve to mitigate that cost. Inserting and deleting elements from anywhere other than the *back* of a `vector` will trigger realignment of the elements to maintain contiguous storage.
- The `list` is a doubly-linked list structure that allows elements to be inserted and deleted in constant *O(1)* time. Traversing the list happens in linear *O(n)* time. A single-linked variant is available as `forward_list`, which iterates forward only. A `forward_list` uses less space and is somewhat more efficient than a doubly-linked `list`, but iterates only in one direction.
- The `deque` (commonly pronounced, *deck*) is a **d**ouble-**e**nded **que**ue. It's a sequential container that can be expanded or contracted on both ends. A `deque` allows random access to its elements, much like a `vector`, but does not guarantee contiguous storage.

## Associative containers

An associative container associates a key with each element. Elements are referenced by their key, rather than their position in the container. STL associative containers include these classes:

- The `set` is an associative container where each element is also its own key. Elements are ordered, usually by some sort of binary tree. Elements in a `set` are immutable and cannot be modified, but they can be inserted and removed. Elements in a `set` must be *unique*; duplicates are not allowed. A `set` iterates in order according to its sorting operators.

- The `multiset` is like a `set` with non-unique keys, where duplicates are allowed.
- The `unordered_set` is like a `set` that does not iterate in order. Elements are not sorted in any specific order but are organized according to their hash values for fast access.
- The `unordered_multiset` is like an `unordered_set` with non-unique keys, where duplicates are allowed.
- The `map` is an associative container for key-value pairs, where each *key* is mapped to a specific *value* (or *payload*). The types of the key and value may be different. Keys are unique but values are not. A `map` iterates in the order of its keys according to its sorting operators.
- The `multimap` is like a `map` with non-unique keys, where duplicate keys are allowed.
- The `unordered_map` is like a map that does not iterate in order.
- The `unordered_multimap` is like an `unordered_map` with non-unique keys, where duplicates are allowed.

## Container adapters

A container adapter is a class which encapsulates an underlying container. The container class provides a specific set of member functions to access the underlying container elements. The STL provides these container adapters:

The `stack` provides a **LIFO** (last-in, first-out) interface where elements may be added and extracted from only one end of the container. The underlying container may be one of `vector`, `deque`, or `list`. If no underlying container is specified, the default is `deque`.

The `queue` provides a **FIFO** (first-in, first-out) interface where elements may be added at one end of the container and extracted from the other end. The underlying container may be one of `deque` or `list`. If no underlying container is specified, the default is `deque`.

The `priority_queue` keeps the greatest value element at the top, according to a *strict weak ordering*. It provides a constant time lookup of the greatest value element, at the expense of logarithmic time insertion and extraction. The underlying container may be one of `vector` or `deque`. If no underlying container is specified, the default is `vector`.

# Use uniform erase functions to delete elements from a container

Before C++20, the *erase-remove idiom* was commonly used to efficiently delete elements from an STL container. This was a little cumbersome, but not a great burden. It was common to use a function like this for the task:

```
template<typename Tc, typename Tv>
void remove_value(Tc & c, const Tv v) {
    auto remove_it = std::remove(c.begin(), c.end(), v);
    c.erase(remove_it, c.end());
}
```

The `std::remove()` function is from the <algorithm> header. `std::remove()` searches for the specified value and removes it by shifting elements forward from the end of the container. It does not change the size of the container. It returns an iterator past the end of the shifted range. We then call the container's `erase()` function to delete the remaining elements.

This two-step process is now reduced to one step with the new uniform erasure function:

```
std::erase(c, 5);   // same as remove_value() function
```

This one function call does the same thing as the `remove_value()` function we wrote above.

There's also a version that uses a predicate function. For example, to remove all even-numbered values from a numeric container:

```
std::erase_if(c, [](auto x) { return x % 2 == 0; });
```

Let's look at the uniform erasure functions in a bit more detail.

## How to do it

- There are two forms of the uniform erasure functions. The first form, called `erase()`, takes two parameters: a container and a value:

  ```
  erase(container, value);
  ```

The container may be any of the sequential containers (`vector`, `list`, `forward_list`, `deque`), except `array`, which cannot change size.

- The second form, called `erase_if()`, takes a container and a predicate function:

```
erase_if(container, predicate);
```

This form works with any of the containers that work with `erase()`, plus the associative containers, `set`, `map`, and their multi-key and unordered variants.

The functions `erase()` and `erase_if()` are defined, as non-member functions, in the header for the corresponding container. There is no need to include another header.

Let's look at some examples:

- First, let's define a simple function to print the size and elements of a sequential container:

```
void printc(const auto& r) {
    print("size: {}: ", r.size());
    for( auto& e : r ) print("{} ", e);
    print("\n");
}
```

The `printc()` function uses the C++23 `print()` function to print the size and values of an STL container.

- Here's a `vector` with 10 integer elements, printed with our `printc()` function:

```
vector v {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
printc(v);
```

Output:

```
size: 10: 0 1 2 3 4 5 6 7 8 9
```

We see that the vector has 10 elements. Now we can use `erase()` to remove all elements with the value 5:

```
erase(v, 5);
printc(v);
```

Output:

```
size: 9: 0 1 2 3 4 6 7 8 9
```

- The `vector` version of the `std::erase()` function is defined in the `<vector>` header. After the `erase()` call, the element with the value `5` has been removed and the vector has 9 elements.
  This gives us the exact same results with a `list` container:

```
list l{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
printc(l);
erase(l, 5);
printc(l);
```

Output:

```
size: 10: 0 1 2 3 4 5 6 7 8 9
size: 9: 0 1 2 3 4 6 7 8 9
```

- The `list` version of the `std::erase()` function is defined in the `<list>` header. After the `erase()` call, the element with the value `5` has been removed and the `list` has 9 elements.
  We can use `erase_if()` to remove all the even-numbered elements with a simple predicate function:

```
vector v{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
printc(v);
erase_if(v, [](auto x) { return x % 2 == 0; });
printc(v);
```

Output:

```
size: 10: 0 1 2 3 4 5 6 7 8 9
size: 5: 1 3 5 7 9
```

The `erase_if()` function also works with associative containers, like `map`:

```
void print_assoc(const auto& r) {
    print("size: {}: ", r.size());
    for (auto& [k, v] : r) print("{}:{} ", k, v);
```

```
    print("\n");
}

int main() {
    map<int, string> m{ {1, "uno"}, {2, "dos"},
        {3, "tres"}, {4, "quatro"}, {5, "cinco"} };
    print_assoc(m);
    erase_if(m,
        [](auto&p){auto&[k,v]=p;
        return k % 2 == 0; }
    );
    print_assoc(m);
}
```

Output:

```
  size: 5: 1:uno 2:dos 3:tres 4:quatro 5:cinco
  size: 3: 1:uno 3:tres 5:cinco
```

Because each element of a `map` is returned as a `pair`, we need a different function to print them. The `print_assoc()` function unpacks the `pair` elements with a *structured binding* in the `for` loop. We also use a structured binding in the predicate function of `erase_if()` to isolate the key for filtering the even numbered elements.

## How it works

The `erase()` and `erase_if()` functions are simply wrappers that perform the *erase-remove idiom* in one step. They perform the same operations as a function like this:

```
template<typename Tc, typename Tv>
void remove_value(Tc & c, const Tv v) {
    auto remove_it = std::remove(c.begin(), c.end(), v);
    c.erase(remove_it, c.end());
}
```

If we consider a simple `vector` of `int`, called `vec`, with the following values:

```
vector vec {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

We can visualize `vec` as a one-row table of `int` values:

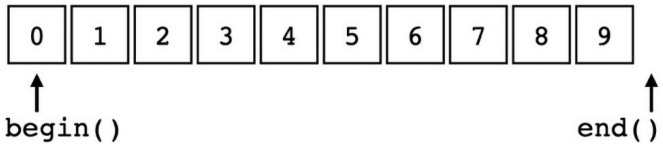



*Figure 3.1:* `begin()` *and* `end()` *iterators*

The `begin()` iterator points at the first element, and the `end()` iterator points *past* the last element. This configuration is standard for all STL sequential containers.

When we call `remove(c.begin(), c.end(), 5)`, the algorithm searches for matching elements, starting at the `begin()` iterator. For each matching element that it finds, it shifts the next element into its place. It continues searching and shifting until it reaches the `end()` iterator. The result is a container where all the remaining elements are at the beginning, without the deleted elements, and in their original order. The `end()` iterator is unchanged, and the remaining elements are *undefined*. We can visualize the `remove()` operation like this:

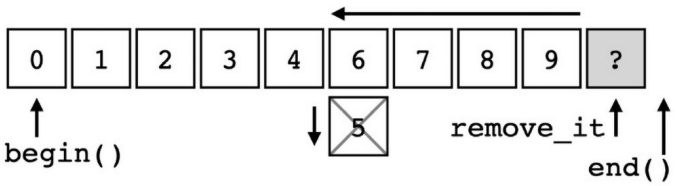



*Figure 3.2: Removing an element*

The `remove()` function returns an iterator (`remove_it`) that points to the first element *past* the elements that were shifted. The `end()` iterator remains as it was before the `remove()` operation. To further illustrate, if we were to remove all even-numbered elements using `remove_if()`, our result would look like this:

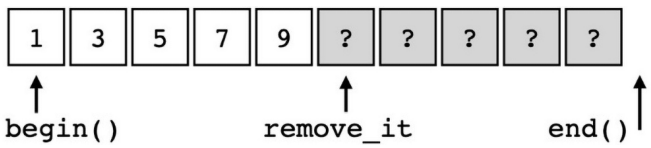



*Figure 3.3: After removing even-numbered elements*

In this case, all that remains are the five odd-numbered elements followed by five elements of *undefined* value.

The container's `erase()` function is then called to erase the remaining elements:

```
c.erase(remove_it, c.end());
```

The container's `erase()` function is called with the `remove_it` and `end()` iterators to delete all the undefined elements.

The `erase()` and `erase_if()` functions call both the `remove()` function and the container's `erase()` function, in order to perform the *erase-remove idiom* in one step.

**Note**

The `remove()` functions avoid resizing the underlying container because it would require additional compute resources to reallocate the underlying container to a smaller size. In circumstances where resizing is required, we can accomplish it in a separate operation.

# Delete items from an unsorted vector

Using the uniform erasure functions (or the *erase-remove idiom*) to delete items from the middle of a `vector` takes linear *O(n)* time. This is because elements must be shifted from the end of the `vector` to close the gap of the deleted items. If the order of items in the `vector` is not important, we can optimize this process to take constant *O(1)* time. Here's how.

## How to do it

This recipe takes advantage of the fact that removing an element from the end of a `vector` is quick and easy.

- Let's start by defining a function to print out a vector:

```
void printc(const auto& r) {
    print("size({}) ", r.size());
    for( auto& e : r ) print("{} ", e);
    print("\n");
}
```

- In our `main()` function we define a `vector` of `int` and print it using `printc()`:

```
int main() {
    vector v {12, 196, 47, 38, 19};
    printc(v);
}
```

Output:

```
  size(10) 0 1 2 3 4 5 6 7 8 9
```

- Now we'll write a function that will delete an element from the `vector`:

```
template<typename T>
void quick_delete(T& v, size_t idx) {
    if (idx < v.size()) {
        if (idx != v.size() - 1) {
            v[idx] = std::move(v.back());
        }
        v.pop_back();
    }
}
```

The `quick_delete()` function takes two arguments, a vector `v` and an index `idx`. We first check to make sure our index is within boundaries. Then we call the `std::move()` function from the `<algorithms>` header to move the last element of the `vector` to the position of our index. Finally, the `v.pop_back()` function is called to shorten the `vector` from the back.

- Let's also create a version of `quick_delete()` for use with an iterator instead of an index:

```
template<typename T>
void quick_delete(T& v, typename T::iterator it) {
    if (it < v.end()) {
        if (it != std::prev(v.end())) {
            *it = std::move(v.back());
        }
        v.pop_back();
    }
}
```

This version of `quick_delete()` operates from an iterator instead of an index. Otherwise, it works the same as the indexed version.

- Now we can call it from our `main()` function:

```
int main() {
    vector v{ 12, 196, 47, 38, 19 };
    printc(v);
    auto it = std::ranges::find(v, 47);
    quick_delete(v, it);
    printc(v);
    quick_delete(v, 1);
    printc(v);
}
```

And the output will look like this:

```
size(5) 12 196 47 38 19
size(4) 12 196 19 38
size(3) 12 38 19
```

The first call to `quick_delete()` uses an iterator from the `std::ranges::find()` algorithm. This deletes the value `47` from the `vector`. Notice the value from the back of the `vector` (`19`) takes its place. The second call to `quick_delete()` uses an index (`1`) to delete the second element from the `vector` (`196`). Again, the value from the back of the `vector` takes its place.

## How it works

The `quick_delete()` function uses a simple trick to delete elements from a `vector` quickly and efficiently. The element at the back of the `vector` is moved (*not copied*) into the position of the element to be deleted. The deleted element is discarded in the process. Then, the `pop_back()` function shortens the `vector` by one element from the end.

This takes advantage of the fact that deleting the element at the back of the `vector` is especially cheap. The `pop_back()` function operates at constant complexity, as it only needs to change the `end()` iterator.

This diagram shows the state of the `vector` before and after the `quick_delete()` operation:

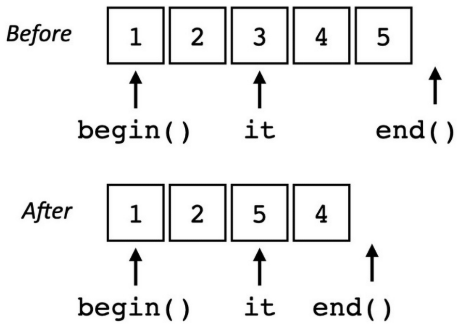



*Figure 3.4: Before and after quick_delete()*

The `quick_remove()` operation simply *moves* the element from the back of the `vector` into the position of the iterator (`it`), then shortens the `vector` by one element. It's important to use `std::move()` instead of an assignment to move the element. The move operation is much faster than a copy-assignment, especially for large objects.

If you don't require ordered elements, this is an extremely efficient technique. It happens in constant *O(1)* time and without touching any other elements.

## Access vector elements safely

The `vector` is one of the most widely used containers in the STL, and for good reason. It's just as convenient as an `array` but far more powerful and flexible. It's common practice to use the `[]` operator to access elements in a vector like this:

```
vector v {19, 71, 47, 192, 4004};
auto& i = v[2];
```

The `vector` class also provides a member function for the same purpose:

```
auto& i = v.at(2);
```

The result is the same but there is an important difference. The `at()` function does bounds checking and the `[]` operator does not. This is intentional, as it allows the `[]` operator to maintain compatibility with the original C-array. Conversely, the bounds checking of the `at()` method requires a tiny bit of extra compute time, which may or may not be important in a given application. Let's examine this in somewhat more detail.

## How to do it

There are two ways to access an element with an index in a `vector`. The `at()` member function does bounds checking, and the `[]` operator does not.

- Here's a simple `main()` function that initializes a `vector` and accesses an element:

```
int main() {
    vector v {19, 71, 47, 192, 4004};
    auto& i = v[2];
    println("element is {}", i);
}
```

Output:

```
element is 47
```

Here, we used the `[]` operator to directly access the third element in the `vector`. As with most sequential objects in C++, the index starts at `0` so the third element is number 2.

- The `vector` has five elements, numbered `0` through 4. If I were to try to access element number `5` that would be beyond the boundary of the vector:

```
vector v {19, 71, 47, 192, 4004};
auto& i = v[5];
println("element is {}", i);
```

Output:

```
element is 0
```

This result is extremely deceiving. It's a common error, since humans tend to count from 1, not 0. But there is no guarantee that an element past the end of the vector has any particular value.

- Even worse, the `[]` operator will silently allow you to *write* to a position beyond the end of the vector:

```
vector v {19, 71, 47, 192, 4004};
v[5] = 2001;
```

```
auto& i = v[5];
println("element is {}", i);
```

Output:

```
element is 2001
```

We have now written to memory that is not under our control and the compiler has *silently* allowed it, with no error messages or crashes. But do not be fooled—this is extremely dangerous code, and it *will* cause problems at some point in the future. Out-of-bounds memory access is one of the primary causes of security breaches.

- The solution is to use the `at()` member function wherever possible, instead of the `[]` operator:

```
vector v {19, 71, 47, 192, 4004};
auto& i = v.at(5);
println("element is {}", i);
```

Now we get a runtime exception:

```
terminate called after throwing an instance of 'std::out_of_range'
  what():  vector::_M_range_check: __n (which is 5) >= this->size() (which
is 5)
Aborted
```

The code compiles without error, but the `at()` function checks the boundaries of the container and throws a *run-time exception* when you try to access memory outside of those boundaries. This is the exception message from code compiled with the GCC compiler. The message will be different in different environments.

## How it works

The `[]` operator and the `at()` member function do the same job: they provide direct access to container elements based on their indexed position. The `[]` operator does it without any bounds checking, so it may be a tiny bit faster in some intensely iterative applications.

That said, the `at()` function *should be your default choice*. While the bounds checking may take a few CPU cycles, it's cheap insurance. For most applications the benefit is well worth the cost.

**Tip**

While the `vector` class is commonly used as a direct-access container, the `array` and `deque` containers also support both the `[]` operator and the `at()` member function. These caveats apply there as well.

## There's more...

In some applications you may not want your application to just *crash* when an out-of-bounds condition is encountered. In this case, you can *catch* the exception, like this:

```
int main() {
    vector v {19, 71, 47, 192, 4004};
    try {
        v.at(5) = 2001;
    } catch (const std::out_of_range & e) {
        println("Ouch!\n{}", e.what());
    }
    println("end element is {}", v.back());
}
```

Output:

```
Ouch!
vector
end element is 4004
```

The `try` block catches the exception specified in the `catch` clause; in this case, the exception is `std::out_of_range`. The `e.what()` function returns a C-string with the error message from the STL library. Each implementation will have different messages.

Keep in mind that all of this applies to `array` and `deque` containers as well.

# Keep vector elements sorted

The `vector` is a sequential container that keeps elements in the order in which they were inserted. It does not sort elements, nor change their order in any way. Other containers, such as `set` and `map`, keep elements sorted, but those containers are not random-access and may not have the features you need. You can, however, keep your `vector` sorted. It just requires a little bit of management.

## How to do it

The idea with this recipe is to create a simple function, `insert_sorted()`, that inserts an element into the correct position in a `vector` to keep the `vector` sorted.

- For convenience, we'll start with *type aliases* for a `vector` of `string` and a vector of `int`:

```
using Vstr = std::vector<std::string>;
using Vint = std::vector<int>;
```

  I like the type aliases here because the exact details of the `vector` are not so important as its application.

- Then we can define a couple of support functions:

```
// print a vector
void printv(const auto& v) {
    for(const auto& e : v) {
        print("{} ", e);
    }
    print("\n");
}
// is it sorted?
void psorted(const auto& v) {
    if(ranges::is_sorted(v)) print("sorted: ");
    else print("unsorted: ");
    printv(v);
}
```

  The `printv()` function is simple enough; it prints the elements of the `vector` on one line.

  The `psorted()` function uses the `ranges` version of the `is_sorted()` algorithm to tell us if the `vector` is sorted. Then it calls `printv()` to print the `vector`.

- Now we can initialize a `Vstr` vector in our `main()` function:

```
int main() {
    Vstr v{
        "Miles",
        "Hendrix",
        "Beatles",
```

```
        "Zappa",
        "Shostakovich"
    };
    psorted(v);
}
```

Output:

```
  unsorted: Miles Hendrix Beatles Zappa Shostakovich
```

At this point we have a `Vstr` vector with the names of some interesting musicians, in no particular order.

- We can sort our `vector` using the `ranges` version of the `sort()` algorithm

```
ranges::sort(v);
psorted(v);
```

Output:

```
  sorted: Beatles Hendrix Miles Shostakovich Zappa
```

- We want to be able to insert items into the `vector` so that they're already in sorted order. The `insert_sorted()` function does this for us:

```
void insert_sorted(Vstr& v, const string& s) {
    const auto pos {std::ranges::lower_bound(v, s)};
    v.insert(pos, s);
}
```

The `insert_sorted()` function uses the `ranges` version of the `lower_bound()` algorithm to get an iterator for the `insert()` function that keeps the vector sorted.

- Now we can use the `insert_sorted()` function to insert more musicians into the `vector`:

```
insert_sorted(v, "Ella");
insert_sorted(v, "Stones");
```

Output:

```
    sorted: Beatles Ella Hendrix Miles Shostakovich Stones Zappa
```

## How it works

The `insert_sorted()` function is used to insert elements into a sorted `vector` while maintaining its order:

```
void insert_sorted(Vstr& v, const string& s) {
    const auto pos {std::ranges::lower_bound(v, s)};
    v.insert(pos, s);
}
```

The `lower_bound()` algorithm finds the first element *not less than* the argument. We then use the iterator returned by `lower_bound()` to insert an element at the correct position.

In this case we're using the `ranges` version of `lower_bound()`, but either version will work.

## There's more...

The `insert_sorted()` function can be made more generic by using a template. This version will work with other container types, such as `set`, `deque`, and `list`.

```
template<typename C, typename E>
void insert_sorted(C& c, const E& e) {
    const auto pos {std::ranges::lower_bound(c, e)};
    c.insert(pos, e);
}
```

**Note**

Note that the `std::sort()` algorithm (and its derivatives) requires a container that supports random access. Not all STL containers fulfil this requirement. Notably, `std::list` does not.

# Efficiently insert elements into a map

The `map` class is an associative container that holds *key-value pairs*, where keys must be unique within the container.

There are a number of ways to populate a `map` container. Consider a `map` defined like this:

```
map<string, string> m;
```

We can add an element with the `[]` operator:

```
m["Miles"] = "Trumpet"
```

We can use the `insert()` member function:

```
m.insert({"Hendrix", "Guitar"});
```

Here we use *template argument deduction* to pass a `std::pair` of values to the `m.insert()` function.

Or, we can use the `emplace()` member function:

```
m.emplace("Krupa", "Drums");
```

I tend to gravitate toward the `emplace()` function. Introduced with C++11, `emplace()` uses *perfect forwarding* to create the new element in place by forwarding parameters directly to the container's constructor. This is quick, efficient, and easy to code.

Though it's certainly an improvement over the other options, the problem with `emplace()` is that it constructs an object even when it's not needed. This involves calling the constructors, allocating memory, moving data around, and then discarding that temporary object.

To solve this problem, C++17 provided the new `try_emplace()` function which only constructs the *value object* if it's needed. This is especially important with large objects or many emplacements.

## How to do it

Each element of a `std::map` is a key-value pair. Within the `pair` structure, the elements are named `first` and `second`, but their purpose in the map is *key* and *value*. I tend to think of the value object as the *payload*, as this is usually the point of the `map`.

To search for an existing key, the C++17 `try_emplace()` function must construct the *key* object; this cannot be avoided. But it need not construct the *value* object unless and until it's needed

for insertion into the `map`. This creates valuable efficiency in the case of key collisions, especially with large payloads. Let's take a look:

- First, we create a payload class. For demonstration purposes, this class has a simple `std::string` payload and displays a message when constructed:

```
struct BigThing {
    string value;
    BigThing(const char * v) : value(v) {
        println("BigThing constructed {}", value);
    }
};
using Mymap = map<string, BigThing>;
```

  This `BigThing` class has only one member function, a constructor that displays a message when the object is constructed. We'll use this to keep track of how often a `BigThing` object is constructed. In practice, of course, this class would be larger and use more resources.

  Each `map` element will consist of a `pair` of objects, a `std::string` for the key and a `BigThing` object for the payload. `Mymap` is just a convenience alias. This allows us to focus on function rather than form.

- We'll also create a `printm()` function to print the contents of the `map`:

```
void printm(const Mymap& m) {
    for(auto& [k, v] : m) {
        print("[{}:{}] ", k, v.value);
    }
    print("\n");
}
```

  This uses the C++23 `print()` function to print out the `map`, so we can keep track of the elements as we insert them.

- In our `main()` function we create the `map` object and insert some elements:

```
int main() {
    Mymap m;
    m.emplace("Miles", "Trumpet");
    m.emplace("Hendrix", "Guitar");
    m.emplace("Krupa", "Drums");
```

```
    m.emplace("Zappa", "Guitar");
    m.emplace("Liszt", "Piano");
    printm(m);
}
```

Output:

```
BigThing constructed Trumpet
BigThing constructed Guitar
BigThing constructed Drums
BigThing constructed Guitar
BigThing constructed Piano
[Hendrix:Guitar] [Krupa:Drums] [Liszt:Piano] [Miles:Trumpet] [Zappa:Guitar]
```

Our output shows the construction of each of the payload objects, and then the output from the `printm()` function call.

- I used the `emplace()` method to add the elements to the `map`, and each payload element was constructed just once. We can use the `try_emplace()` method and the result will be the same:

```
Mymap m;
m.try_emplace("Miles", "Trumpet");
m.try_emplace("Hendrix", "Guitar");
m.try_emplace("Krupa", "Drums");
m.try_emplace("Zappa", "Guitar");
m.try_emplace("Liszt", "Piano");
printm(m);
```

Output:

```
BigThing constructed Trumpet
BigThing constructed Guitar
BigThing constructed Drums
BigThing constructed Guitar
```

```
BigThing constructed Piano
[Hendrix:Guitar] [Krupa:Drums] [Liszt:Piano] [Miles:Trumpet] [Zappa:Guitar]
```

- The difference between `emplace()` and `try_emplace()` shows up when we try to insert new elements with duplicate keys:

```
println("emplace(Hendrix)");
m.emplace("Hendrix", "Singer");
println("try_emplace(Zappa)");
m.try_emplace("Zappa", "Composer");
printm(m);
```

Output:

```
emplace(Hendrix)
BigThing constructed Singer
try_emplace(Zappa)
[Hendrix:Guitar] [Krupa:Drums] [Liszt:Piano] [Miles:Trumpet] [Zappa:Guitar]
```

The `emplace()` function tried to add an element with a duplicate key (`"Hendrix"`). It failed but *still constructed* the payload object (`"Singer"`). The `try_emplace()` function also tried to add an element with a duplicate key (`"Zappa"`). It failed and *did not* construct the payload object.

This recipe demonstrates the distinctions between `emplace()` and `try_emplace()` in a `map` object. The efficiency of `try_emplace()` may be significant with larger payloads.

## How it works

The `try_emplace()` function signature is similar to that of `emplace()`, so it should be easy to retrofit legacy code. Here's the `try_emplace()` function signature:

```
pair<iterator, bool> try_emplace( const Key& k, Args&&... args );
```

At first glance, this looks different from the `emplace()` signature:

```
pair<iterator,bool> emplace( Args&&... args );
```

The distinction is that `try_emplace()` uses a separate parameter for the *key* argument, which allows it to be isolated for construction. Functionally, because of *template argument deduction*, `try_emplace()` can be a drop-in replacement:

```
m.emplace("Miles", "Trumpet");
m.try_emplace("Miles", "Trumpet");
```

The return value of `try_emplace()` is the same as that of `emplace()`, a `pair` representing an iterator and a `bool`:

```
    const auto key = "Zappa";
    const auto payload = "Composer";
    if(auto [it, success] = m.try_emplace(key, payload);
            !success) {
        println("update");
        it->second = payload;
    }
    printm(m);
```

Output:

```
update
BigThing constructed Composer
[Hendrix:Guitar] [Krupa:Drums] [Liszt:Piano] [Miles:Trumpet] [Zappa:Composer]
```

Here we used *structured binding* (`auto [it, success] =`) with an `if` *initializer statement* to test the return value and conditionally update the payload. Notice that it still just constructs the payload object once.

It's worth noting that the `try_emplace()` function also works with `unordered_map`. We change our alias and everything works the same except unordered:

```
using Mymap = unordered_map<string, BigThing>;
```

The advantage of `try_emplace()` is that it only constructs the payload object *if and when* it's ready to store it in the `map`. In practice, this should save significant resources at run-time. We should always favor `try_emplace()` over `emplace()`.

# Efficiently modify keys in a map

A `map` is an associative container that stores *key-value pairs*. The container is ordered by the keys. The keys must be unique and they are `const`-qualified, so they cannot be changed.

For example, if I populate a `map` and attempt to change the key, I'll get an error at compilation time:

```
map<int, string> mymap {
    {1, "foo"}, {2, "bar"}, {3, "baz"}
};
auto it = mymap.begin();
it->first = 47;
```

Output from `gcc`; your error may differ:

```
error: assignment of read-only member ...
    5 |     it->first = 47;
      |
```

If you need to re-order a map container, you may do so by swapping keys using the `extract()` method.

Beginning with C++17, the `extract()` member function allows elements of a `map` to be extracted from the sequence without touching the payload. Once extracted, the key is no longer `const`-qualified and may be modified.

Let's look at an example.

## How to do it

In this example we'll define a `map` that represents contestants in a race. At some point during the race, the order changes, and we need to modify the keys of the `map`.

- We start by defining an alias for the `map` type:

  ```
  using Racermap = map<unsigned int, string>;
  ```

  This allows us to use the type consistently throughout our code.

- We write a function for printing out the `map`:

```
void printm(const Racermap &m) {
    println("Rank:");
    for (const auto& [rank, racer] : m) {
        println("{}:{}", rank, racer);
    }
}
```

We can pass the `map` to this function at any time to print out the current rankings of our contestants.

- In our `main()` function we define a `map` with the initial state of our racers:

```
int main() {
    Racermap racers {
        {1, "Mario"}, {2, "Luigi"}, {3, "Bowser"},
        {4, "Peach"}, {5, "Donkey Kong Jr"}
    };
    printm(racers);
    node_swap(racers, 3, 5);
    printm(racers);
}
```

The key is an `int` indicating the rank of the racer. The value is a `string` with the name of the racer. We call `printm()` to print the current rank.

At some point, one of the racers falls behind and another racer takes the opportunity to move up in the rankings. The `node_swap()` function will swap the ranking of two racers:

```
template<typename M, typename K>
bool node_swap(M& m, K k1, K k2) {
    if (k1 == k2) return true;
    auto node1 = m.extract(k1);
    auto node2 = m.extract(k2);
    if (node1.empty() || node2.empty()) {
        if (!node1.empty()) m.insert(move(node1));
        if (!node2.empty()) m.insert(move(node2));
        return false;
    }
```

```
    swap(node1.key(), node2.key());
    m.insert(move(node1));
    m.insert(move(node2));
    return true;
}
```

This function uses the `map::extract()` method to extract the specified elements from the `map`. These extracted elements are called *nodes*.

A *node* is a new concept introduced with C++17. It allows an element to be extracted from a `map`-type structure without touching the element itself. The node is unlinked, and a *node handle* is returned. Once extracted, the node handle provides *writable* access to the key via the node's `key()` function. We can then swap the keys and insert them back into the `map`, without ever having to copy or manipulate the payload.

- We can call `node_swap()` at the end of `main()` to swap the positions of two racers like this:

```
    node_swap(racers, 3, 5);
    printm(racers);
```

Now our output looks like this:

```
Rank:
1:Mario
2:Luigi
3:Bowser
4:Peach
5:Donkey Kong Jr
Rank:
1:Mario
2:Luigi
3:Donkey Kong Jr
4:Peach
5:Bowser
```

This is all made possible by the `extract()` method and the new `node_handle` class. Let's take a closer look at how this works.

## How it works

This technique uses the new `extract()` method, which returns a `node_handle` object. As the name suggests, a `node_handle` is a handle to a *node*, which consists of an associative element and its related structures. The extract function *disassociates* the node, while leaving it in place, and returns a `node_handle` object. This has the effect of removing the node from the associative container without touching the data itself. The `node_handle` allows us to access the disassociated node.

The `node_handle` has a member function, `key()`, which returns a *writable* reference to the node key. This allows us to change the key, while it's disassociated from the container.

## There's more...

There are a few things to keep in mind when using `extract()` and a `node_handle`:

- If the key is not found, the `extract()` function returns an empty node handle. You can test if a node handle is empty with the `empty()` function or with its boolean operator:

  ```
  auto node = mapthing.extract(key);
  if(node.empty()) {
      // node handle is empty
  }
  ```

  or...

  ```
  if (!node) {
      // node handle is empty
  }
  ```

- There are two overloads of the `extract()` function:

  ```
  node_type extract(const key_type& x);
  node_type extract(const_iterator position);
  ```

  We used the first form, by passing a key. You may also use an iterator, which should not require a lookup.

- Keep in mind that previous to C++23 you could not make a reference from a literal, so a call like `extract(1)` could crash with a segmentation fault
- Keys must remain unique when inserted into a `map`

For example, if I try to change a key to a value already in the map:

```
auto node_x = racers.extract(racers.begin());
node_x.key() = 5;  // 5 is Donkey Kong Jr
auto status = racers.insert(move(node_x));
if (!status.inserted) {
    println("insert failed, dup key: {}",
        status.position->second);
}
```

The insert fails and we get the error message:

```
  insert failed, dup key: Donkey Kong Jr
```

In this example I've passed the `begin()` iterator to `extract()`. I then assigned the key a value that's already in use (`5, Donkey Kong Jr`). The insert failed and the resulting `status.inserted` is false. `status.position` is an iterator to the found key. In the `if()` block, I used `println()` to print the value the inserted key.

# Use unordered_map with custom keys

With an ordered `map`, the type of the key must be sortable, which means it must at least support the less-than `<` comparison operator. Suppose you want to use an associative container with a custom type that is not sortable. For example, a vector where `(0, 1)` is not smaller or larger than `(1, 0)`; it simply points in a different direction. In such cases, you may use the `unordered_map` type. Let's look at how to do this.

## How to do it

For this recipe we'll create an `unordered_map` object that uses *x/y* coordinates for the key. We will need a few support functions for this.

- First, we'll define a structure for the coordinates:

```
struct Coord {
    int x {};
    int y {};
};
```

This is a simple structure with two members, `x` and `y`, for the coordinates.

- Our map will use the `Coord` structure for the key, and an `int` for the value:

```
using Coordmap = unordered_map<Coord, int>;
```

The `using` alias makes it convenient to use our `unordered_map`.

- To use the `Coord` struct as a key, we need a couple of overloads. These are required for use with an `unordered_map`. First, we'll define an equality comparison operator:

```
bool operator==(const Coord& lhs, const Coord& rhs) {
    return lhs.x == rhs.x && lhs.y == rhs.y;
}
```

This is a simple function that compares `x` members and `y` members with each other.

- We also need a `std::hash` class specialization. This makes it possible to retrieve map elements with the key:

```
namespace std {
    template<>
    struct hash<Coord> {
        size_t operator()(const Coord& c) const noexcept {
            size_t h1{std::hash<int>{}(c.x)};
            size_t h2{std::hash<int>{}(c.y)};
            return h1 ^ (h2 << 1);
        }
    };
}
```

This provides a specialization for the default `hash` class used by the `std::unordered_map` class. It must be in the `std` namespace.

- We'll also write a `print_Coordmap()` function to print a `Coordmap` object:

```
void print_Coordmap(const Coordmap& m) {
    for (const auto& [key, value] : m) {
        print("{{ ({}, {}): {} }} ",
            key.x, key.y, value);
    }
    print("\n");
}
```

This prints the *x/y* key and the value using the C++23 `print()` function. Notice the use of the double braces, `{{` and `}}`, to print single braces.

- Now that we have all our support functions, we can write the `main()` function

```
int main() {
    Coordmap m {
        { {0, 0}, 1 },
        { {0, 1}, 2 },
        { {2, 1}, 3 }
    };
    print_Coordmap(m);
}
```

Output:

```
  { (2, 1): 3 } { (0, 1): 2 } { (0, 0): 1 }
```

Because it's an *unordered* map, the order of the elements may be different for you.

At this point, we've defined a `Coordmap` object that accepts `Coord` objects for the keys and maps them to arbitrary values.

- We can also access individual members based on the `Coord` keys:

```
Coord k{ 0, 1 };
print("{{ ({}, {}): {} }}\n", k.x, k.y, m.at(k));
```

Output:

```
  { (0, 1): 2 }
```

Here we define a `Coord` object named `k`, and we use that with the `at()` function to retrieve a value from the `unordered_map`.

## How it works

The `unordered_map` class relies on a hash class to look up elements from the key. We normally instantiate an object like this:

```
std::unordered_map<key_type, value_type> my_map;
```

What's not obvious here is that, because we haven't defined one, `unordered_map` is using a *default hash class*. The full template type definition of the `unordered_map` class looks like this:

```
template<
    class Key,
    class T,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator< std::pair<const Key,
      T> >
> class unordered_map;
```

The template provides default values for `Hash`, `KeyEqual`, and `Allocator`, so we don't always include them in our definitions. In our example, we've provided a specialization for the default `std::hash` class.

The STL contains specializations of `std::hash` for most of the standard types, like `string`, `int`, and so on. For it to work with our `Coord` class, we must provide a specialization.

We could have passed a functor to the template parameter, like this:

```
std::unordered_map<coord, value_type, my_hash_type> my_map;
```

That certainly would work. However, using a specialization for `std::hash` provides a more general solution which may be used with other hash-based containers, such as `unordered_set`, without repeatedly specifying a separate hash functor.

## Sort and filter user input with set

The `set` class is an associative container where each element is a single value, which is both key and payload. Elements in a `set` are maintained in sorted order and duplicate elements are not allowed.

The `set` container is often misunderstood, and it does have fewer and more specific uses than more general containers such as `vector` and `map`. One common use for a `set` is to filter duplicates from a set of values.

## How to do it

In this recipe we will read words from the *standard input* and filter out the duplicates.

- We start by defining an alias for an `istream` iterator. We'll use this to get input from the standard input.

```
using input_it = istream_iterator<string>;
```

- In the `main()` function, we'll define a `set` for our words:

```
set<string> words;
```

  The `set` is defined as a set of `string` elements.

- We define a pair of iterators for use with the `inserter()` function:

```
input_it it {std::cin};
input_it end {};
```

  The `end` iterator is initialized with its default constructor. This is known as the *end-of-stream* iterator. At the end of input this iterator will compare equal with the `cin` iterator.

- The `std::inserter()` function is used to insert elements into the `set` container:

```
std::copy(it, end, std::inserter(words, words.end()));
```

  We use `std::copy()` to conveniently copy words from the input stream.

- Now we can print out our `set` to see the results:

```
for(const string & w : words) {

    print("{} ", w);
}
```

```
print("\n");
```

- We can run the program by piping a bunch of words to its input:

```
$ echo "a a a b c this that this foo foo foo" | ./set-words
a b c foo that this
```

The `set` has eliminated the duplicates and retained a sorted list of the words that were inserted.

## How it works

The `set` container is the heart of this recipe. It only holds unique elements. When you insert a duplicate, that insert will fail and you end up with a sorted list of each unique element.

But that's not the only interesting part of this recipe.

The `istream_iterator` is an input iterator that reads objects from a stream. We instantiated the input iterator like this:

```
istream_iterator<string> it {std::cin};
```

Now we have an input iterator of type `string` from the `cin` stream. Every time we dereference this iterator, it will return one word from the input stream.

We also instantiated another `istream_iterator`:

```
istream_iterator<string> end {};
```

This calls the default constructor, which gives us a special *end-of-stream* sentinel iterator. When the input iterator reaches the end of the stream, it will compare equal with the `end` sentinel. This is convenient for completing loops, such as the one created by the `copy()` algorithm.

The `copy()` algorithm takes three iterators: the beginning and end of the range to copy, and a destination iterator:

```
std::copy(it, end, std::inserter(words, words.end()));
```

The `inserter()` function takes a container and an iterator for the insertion point and returns an `insert_iterator` of the appropriate type for the container and its elements.

This combination of `copy()` and `inserter()` makes it easy to copy elements from a stream into the `set` container.

# Create a simple RPN calculator with deque

In this recipe, we will build a simple RPN calculator using the `std::deque` container as a last-in, first-out (LIFO) stack.

A reverse polish notation (RPN) calculator is a stack-based calculator that uses postfix notation, where the operator follows the operands. It's commonly used in printing calculators and, notably, the most popular electronic calculator of all time, the HP 12C.

After becoming familiar with its operational modality, many people prefer an RPN calculator. (I've been using the HP 12C and 16C since they were first introduced in the early 1980s.) As an example of the difference between an algebraic and an RPN calculator, using conventional algebraic notation, to add 1 and 2 you would type `1 + 2`. Using RPN, you type `1 2 +`. The operator comes *after* the operands.

Using an algebraic calculator, you press an **=** key to indicate that you want a result. With an RPN calculator this is unnecessary because the operator processes immediately, serving both purposes. On the other hand, an RPN calculator often requires an extra keypress (*Enter*) to push an operand onto the stack.

We can easily implement an RPN calculator using a stack-based data structure. For example, consider an RPN calculator with a four-position stack:

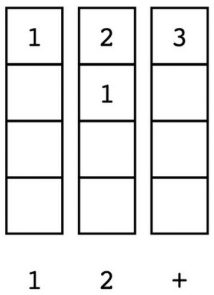



*Figure 3.5: RPN addition operation*

Each operand is pushed onto the stack as they are entered. When the operator is entered, the operands are popped off the stack, operated upon, and the result is pushed back onto the stack. The result may then be used in the next operation. For example, consider the case of *(3+2)×3*:

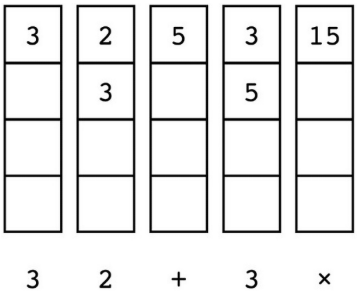



*Figure 3.6: RPN stack operations*

One advantage of RPN is that you can leave operands on the stack for future calculations, reducing the need for separate memory registers. Consider the case of *(9×6)+(2×3)*:

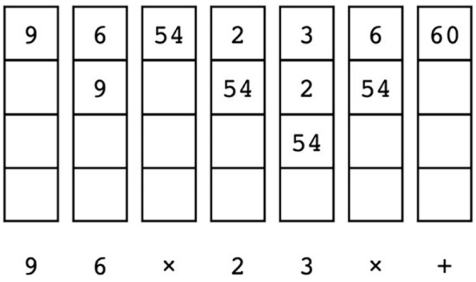



*Figure 3.7: RPN multiple stack operations*

Notice that we first perform the operations within the parentheses, then the final operation on the intermediate results. This may seem more complex at first, but it makes a lot of sense once you get used to it.

Now, let's build a simple RPN calculator using the STL `deque` container.

**Note**

The word "Polish" in the term, Reverse Polish Notation, refers to the nationality of logician Jan Łukasiewicz, who invented Polish Notation in 1924.

## How to do it

For this implementation, we'll use a `deque` container for our stack. Why not use a `stack` container? The `stack` class is a container-adapter, which uses another container (usually a `deque`) for its storage. For our purposes, `stack` doesn't provide any tangible advantage over

deque. And deque allows us to iterate over and display the RPN stack, like a paper tape calculator.

- First, we create a class to *encapsulate* our RPN calculator. There are a few advantages to using a class here. Encapsulation provides safety, reusability, extensibility, and a clean interface. We'll call this class `RPN`.

```
class RPN {
    deque<double> stk{};
    constexpr static double zero {0.0};
    constexpr static double inf
        {std::numeric_limits<double>::infinity()};
...
};
```

The deque data store, named `stk`, is in the private area of the class to protect it. This is where we store the RPN stack.

The `zero` constant is used throughout the class, both as a return value and as a comparison operand. The `inf` constant is used for a divide-by-zero error. These constants are declared `constexpr static` so they don't take up space in every instance.

- We don't need an explicit constructor or destructor because the deque class manages its own resources. So, our `public` interface consists of just three functions:

```
// process an operand/operator
double op(const string & s) {
    if (is_numeric(s)) {
        double v{stod(s)};
        stk.push_front(v);
        return v;
    }
    else return optor(s);
}

// empty the stack
void clear() {
    stk.clear();
}

// print the stack
```

```
string get_stack_string() const {
    string s{};
    for(auto v : stk) {
        s += format("{} ", v);
    }
    return s;
}
```

The `double op()` function is the main entry point for the `RPN` class. It takes a `string`, with either a number or an operator. If it's a number, it's converted to a `double` and pushed onto the stack. If it's an operator, we call `optor()` to perform the operation. This is the main logic of the class.

The `void clear()` function simply calls `clear()` on the deque to empty the stack.

And finally, the `get_stack_string()` function returns the contents of the stack in a `string`.

- In the `private` section, we have the supporting utilities that make the interface work. The `pop_get2()` function pops two operands from the stack and returns them as a pair. We use this as operands for the operators:

```
pair<double, double> pop_get2() {
    if (stk.size() < 2) return {zero, zero};
    double v1 {stk.front()};
    stk.pop_front();
    double v2 {stk.front()};
    stk.pop_front();
    return {v2, v1};
}
```

- The `is_numeric()` function checks to see if the string is entirely numeric

```
    bool is_numeric(const string& s) {
        size_t pos{};
        try {
            std::stod(s,&pos);
            return pos == s.size();
        }
        catch (...) {
            return false;
```

```
        }
    }
```

The `stod()` function converts a `string` to a `double`. But in this case, we're simply using it to determine if a string is numeric. The `stod()` function throws an exception when it fails, so we simply catch any exception and return false for non-numeric values.

- The `optor()` function performs the operators. We use a `map` container to map an operator to a corresponding lambda function:

```
double optor(const string& op) {
    map<string, double (*)(double, double)> opmap {
        {"+", [](double l, double r){ return l + r; }},
        {"-", [](double l, double r){ return l - r; }},
        {"*", [](double l, double r){ return l * r; }},
        {"/", [](double l, double r){ return l / r; }},
        {"^", [](double l, double r){ return pow(l, r); }},
        {"%", [](double l, double r){ return fmod(l, r); }}
    };

    if (opmap.find(op) == opmap.end()) return zero;
    auto [l, r] = pop_get2();

    // don't divide by zero
    if (op == "/" && r == zero) stk.push_front(inf);
    else stk.push_front(opmap.at(op)(l, r));
    return stk.front();
}
```

The `map` container with lambda functions makes a quick and easy jump table. We use the `find()` function in `map` to test if we have a valid operator. After a test for divide-by-zero, the `map` is dereferenced, and the operator is called. The result of the operation is pushed onto the stack and returned.

- Those are all the function members of the `RPN` class. Now we can call it from our `main()` function:

```
int main() {
    RPN rpn;

    for (string o{}; cin >> o; ) {
```

```
            rpn.op(o);
            auto stack_str = rpn.get_stack_string();
            println("{}: {}", o, stack_str);
        }
    }
```

We test this is by piping a string into the program from the command line. We use a `for` loop to fetch each word from the `cin` stream and pass it to `rpn.op()`. I like the `for` loop here, as it's easy to contain the scope of the o variable. We then print the stack using the `get_stack_string()` function after each command line item.

- We can run the program by piping in an expression like this:

```
$ echo "9 6 * 2 3 * +" | ./rpn
9: 9
6: 6 9
*: 54
2: 2 54
3: 3 2 54
*: 6 54
+: 60
```

This is actually quite simple. With the comments, the `RPN` class is less than 70 lines of code. The full `rpn.cpp` source code is in the GitHub repository.

## How it works

The `RPN` class operates by first determining the nature of each chunk of input. If it's a number, we push it onto the stack. If it's an operator, we pop two operands off the top of the stack, apply the operation, and push the result back on the stack. If we don't recognize the input, we just ignore it.

The `deque` class is a double-ended queue. To use it as a stack, we pick an end and both push and pop from that same end. I chose the `front` end of the deque, but it would work just as well from the `back`. We just need to do everything from the same end.

If we determine that an input is numeric, we convert it to a `double` and push it onto the front of the `deque` using `push_front()`.

```
if (is_numeric(s)) {
    double v {stod(s, nullptr)};
    stk.push_front(v);
```

```
    return v;
}
```

When we need to use values from the stack, we pop them off the front of the deque. We use `front()` to get the value and then `pop_front()` to pop it off the stack.

```
    pair<double, double> pop_get2() {
        if (stk.size() < 2) return {zero, zero};
        double v1{stk.front()};
        stk.pop_front();
        double v2{stk.front()};
        stk.pop_front();
        return {v2, v1};
    }
```

Using a `map` for our operators makes it easy to both check if an operator is valid and to execute the operation.

```
map<string, double (*)(double, double)> opmap {
    {"+", [](double l, double r){ return l + r; }},
    {"-", [](double l, double r){ return l - r; }},
    {"*", [](double l, double r){ return l * r; }},
    {"/", [](double l, double r){ return l / r; }},
    {"^", [](double l, double r){ return pow(l, r); }},
    {"%", [](double l, double r){ return fmod(l, r); }}
};
```

We can test for the validity of an operator by using the `find()` function:

```
if (opmap.find(op) == opmap.end()) return zero;
```

Because `opmap` uses a lambda as its value, we can call the operator by dereferencing `opmap` with the `at()` function:

```
opmap.at(op)(l, r)
```

We both call the operator lambda and push the result onto the `deque` in one statement:

```
stk.push_front(opmap.at(op)(l, r));
```

## There's more...

In this recipe, we use the `cin` stream to feed operations to the RPN calculator. It would be just as easy to do this with an STL container.

```
int main() {
    RPN rpn;
    vector<string> opv{ "9", "6", "*", "2", "3", "*", "+"
      };
    for(auto o : opv) {
        rpn.op(o);
        auto stack_str = rpn.get_stack_string();
        println("{}: {}", o, stack_str);
    }
}
```

Output:

```
9: 9
6: 6 9
*: 54
2: 2 54
3: 3 2 54
*: 6 54
+: 60
```

By putting the RPN calculator in a class with a clean interface, we've created a flexible tool that can be used in many different contexts.

# Count word frequency with map

This recipe uses the unique key property of the `map` container to count duplicate words from a stream of text.

The STL `map` container is an *associative* container. It consists of elements organized in *key-value pairs*. The keys are used for lookup and must be unique.

## How to do it

There are a few parts to this task that we can solve separately.

1. We need to get the text from a file. We'll use the `cin` stream for this.

2. We need to separate words from punctuation and other non-word content. We'll use the `regex` (Regular Expression) library for this.
3. We need to count the frequency of each word. This is the main objective of the recipe. We'll use the STL `map` container for this.
4. Finally, we need to sort the results, first by frequency and then alphabetically by word within frequency. For this we'll use the STL `sort` algorithm with a `vector` container.

Even with all those tasks, the resulting code is relatively short, just about 70 lines with headers and all. Let's dive in:

- We'll start with some aliases for convenience:

  ```
  namespace ranges = std::ranges;
  namespace regex_constants = std::regex_constants;
  ```

  For namespaces within the `std::` space, I like to make aliases that are shorter, but still let me know that I'm using a token in a particular namespace. Especially with the `ranges` namespace, which often re-uses the names of existing algorithms.

- We store the regular expression in a constant. I don't like to clutter up the global namespace because that can lead to collisions. I tend to use a namespace based on my initials for things like this:

  ```
  namespace bw {
      constexpr const char * re {"(\\w+)"};
  }
  ```

  It's easy enough to get it later using `bw::re`, and that tells me exactly what it is.

- At the top of `main()`, we define our data structures:

  ```
  map<string, int> wordmap {};
  vector<pair<string, int>> wordvec {};
  regex word_re(bw::re);
  size_t total_words {};
  ```

  Our main `map` is called `wordmap`. We have a `vector` named `wordvec` that we'll use as a sorting container. And finally, our `regex` class, `word_re`.

Most of the work happens in a `for` loop. We read text from the `cin` stream, apply the `regex`, and store words in the `map`:

```
for (string s {}; cin >> s; ) {
    auto words_begin = sregex_iterator(s.begin(), s.end(),
            word_re);
    auto words_end = sregex_iterator();

    for (auto r_it = words_begin; r_it != words_end;
            ++r_it) {
        smatch match {*r_it};
        auto word_str = match.str();

        ranges::transform(word_str, word_str.begin(),
            [](unsigned char c){ return tolower(c); });

        auto [map_it, result] =
            wordmap.try_emplace(word_str, 0);
        auto&[w,count]=*map_it;
        ++total_words;
        ++count;
    }
}
```

I like a `for` loop for this because it allows me to contain the scope of the `s` variable.

1. We start by defining iterators for the `regex` results. This allows us to distinguish multiple words even when surrounded only by punctuation. The `for(r_it...)` loop returns individual words from the `cin` string.
2. The `std::smatch` type is a specialization of a `regex` string match class. It gives us the next word from our `regex`.
3. We then use the `transform` algorithm to make the words lowercase – so we can count words regardless of case. (For example, `"The"` is the same word as `"the"`.)
4. Next, we use `try_emplace()` to add the word to the `map`. If it's already there, it will not be replaced.
5. Finally, we increment the count for the word in the `map` with `++count`.

Now we have the words and their frequency counts in our `map`. But they're in alphabetical order and we want them in descending order of frequency. For this, we put them in a vector and sort them:

```
auto unique_words = wordmap.size();
wordvec.reserve(unique_words);
ranges::move(wordmap, back_inserter(wordvec));
ranges::sort(wordvec, [](const auto& a, const auto& b) {
    if (a.second != b.second)
        return (a.second > b.second);
    return (a.first < b.first);
});

println("total word count: {}", total_words);
println("unique word count: {}", unique_words);
```

`wordvec` is a vector of pairs, with the word and the frequency count. We use the `ranges::move()` algorithm to populate the `vector`, then the `ranges::sort()` algorithm to sort the `vector`. Notice that the *predicate lambda function* sorts first by the count (descending) and then by the word (ascending).

- Finally, we print the results:

```
    for (int limit {20}; auto& [w, count] : wordvec) {
        println("{}: {}", count, w);
        if (--limit == 0) break;
    }
```

I set a limit to print only the first `20` entries. You can comment out the `if(--limit == 0) break;` line to print the whole list.

- In the example files, I've included a text file with a copy of *The Raven*, by Edgar Allan Poe. This beautiful poem is in the public domain. We can use it to test the program:

```
$ ./word-count < the-raven.txt
total word count: 1098
unique word count: 439
56: the
38: and
32: i
24: my
```

```
21: of
17: that
17: this
15: a
14: door
11: chamber
11: is
11: nevermore
10: bird
10: on
10: raven
9: me
8: at
8: from
8: in
8: lenore
```

*The Raven* has `1,098` words total, and `439` of them are unique.

## How it works

The core of this recipe is the use of a `map` object to count duplicate words. But there are other parts that merit consideration.

We use the `cin` stream to read text from the *standard input*. By default, `cin` will skip *whitespace* when reading into a `string` object. By putting a string object on the right-hand side of the `>>` operator (`cin >> s`) we get chunks of text separated by whitespace. This is a good enough definition of a word-at-a-time for many purposes, but we need linguistic words. And for that we will use a regular expression.

The `regex` class provides a choice of regular expression *grammars* and it defaults to *ECMA* grammar. In the ECMA grammar, the regular expression `"(\w+)"` is a shortcut for `"([A-Za-z0-9_]+)"`. This will select words that include the Latin/Roman alphabet, both upper- and lower-case, the ten Arabic numerals, and the underscore _ character.

As we get each word from the `regex` engine, we use the `map` object's `try_emplace()` method to conditionally add the word to our `wordmap`. If the word is not in the map, we add it with a count of `0`. If the word is already in the map, the count is untouched. We increment the count later in the loop, so it's always correct.

After the `map` is populated with all the unique words from the file, we transfer it to a `vector` using the `ranges::move()` algorithm. The `move()` algorithm makes this transfer quick and

efficient. Then we can sort the `vector` using `ranges::sort()`. The *predicate lambda function* for sorting includes comparisons for both sides of the `pair`, so we end up with a result that's sorted by both word count (descending) and the word itself (ascending).

**Tip**

Regular expressions are a language unto themselves. To learn more about regular expressions, I recommend the book *Mastering Regular Expressions* by Jeffrey Friedl.

# Find long sentences with a vector of vectors

It can be useful for a writer to make sure they are using a variety of sentence lengths, or to ensure none of their sentences are too long. Let's build a tool that evaluates a text file for sentence length.

Choosing the appropriate container is key when using the STL. If you need something ordered, it's often best to use an associative container, such as `map` or `multimap`. In this case, however, since we need a custom sort, it's easier to sort a `vector`.

The `vector` is generally the most flexible of the STL containers. Whenever another container type seems appropriate, but is missing one important capability, the `vector` is often an effective solution. In this case, where we need a custom sort, the vector works great.

This recipe uses a *vector of vectors*. The inner `vector` stores the words of a sentence, and the outer `vector` stores the inner `vectors`. As you'll see, this affords a lot of flexibility while retaining all the relevant data.

## How to do it

This program needs to read words from `stdin`, find the ends of sentences, store and sort the sentences, then print out the results.

- In `main()`, we start by defining the *vector of vectors*:

  ```
  vector<vector<string>> vv_sentences(1);
  ```

  This defines a `vector` of elements typed `vector<string>` named `vv_sentences`. The `vv_sentences` object is initialized with one empty `vector` for the first sentence. The inner vectors will each hold a sentence of words.

- Now we can process the stream of words:

```
for (string s{}; cin >> s; ) {
    vv_sentences.back().emplace_back(s);
    if (is_eos(s)) {

        vv_sentences.emplace_back()
    }
}
```

The `for` loop returns one word at a time from the input stream. The `back()` method on the `vv_sentences` object is used to access the current vector of words, and the current word is added using `emplace_back()`. Then we call `is_eos()` to see if this was the end of a sentence. If so, we add a new empty vector to `vv_sentences` to start the next sentence.

- Because we always add a new empty vector to the end of `vv_sentences` after each end-of-sentence character, we will usually end up with an empty sentence `vector` at the end. Here we check for this, and delete it if necessary:

```
// delete back if empty
if (vv_sentences.back().empty())
    vv_sentences.pop_back();
```

- Now we can sort the `vv_sentences` vector by the size of the sentences:

```
sort(vv_sentences, [](const auto& l, const auto& r) {
        return l.size() > r.size();
    });
```

This is why the `vector` is so convenient for this project. It's quick and easy to sort using the `ranges::sort()` algorithm with a simple *predicate lambda* for sorting by size in descending order.

- Now we can print our result:

```
for (const auto& v : vv_sentences) {
    constexpr auto WLIMIT = 10;
    size_t size = v.size();
    size_t limit {WLIMIT};
```

```
        print("{}: ", size);
        for (const auto& s : v) {
            print("{} ", s);
            if (--limit == 0) {
                if (size > WLIMIT) print("...");
                break;
            }
        }
        print("\n");
    }
    print("\n");
```

The outer loop and the inner loop correspond to the outer and inner vectors. We simply loop through the vectors and print out the size of the inner vector with `print("{}: ", size)` and then each word with `print("{} ", s)`. We don't want to print the very long sentences in their entirety, so we define a limit of 10 words and print an ellipsis if there's more.

- The output looks like this, using the first few paragraphs of this recipe for input:

```
$ ./sentences < sentences.txt
27: It can be useful for a writer to make sure ...
19: Whenever another container type seems appropriate, but is missing
one ...
18: If you need something ordered, it's often best to use ...
17: The inner vector stores the words of a sentence, and ...
16: In this case, however, since we need a descending sort, ...
16: In this case, where we need our output sorted in ...
15: As you'll see, this affords a lot of flexibility while ...
12: Let's build a tool that evaluates a text file for ...
11: The vector is generally the most flexible of the STL ...
9: Choosing the appropriate container key when using the STL.
7: This recipe uses a vector of vectors.
```

## How it works

We read the text file one word at a time using `cin`:

```
for (string s{}; cin >> s; ) {
    ...
}
```

This avoids the overhead of reading a large file into memory all at once. The `vector` will already be large, containing all the words of the file. It's not necessary to also hold the entire text file in memory. In the rare case that a file is too large, it would be necessary to find another strategy or use a database.

Finding punctuation is simple using the `find_first_of()` method in the `string` class:

```
for (string s {}; cin >> s; ) {
    vv_sentences.back().emplace_back(s);
    if (s.find_first_of(end_punct) != s.npos) {
        vv_sentences.emplace_back();
    }
}
```

This provides a simple and efficient method of separating sentences.

The *vector of vectors* may look complex at first glance, but it's no more complicated than using two separate vectors:

```
vector<vector<string>> vv_sentences {vector<string>{}};
```

This declares an *outer* `vector`, with *inner* elements of type `vector<string>`. The outer `vector` is named `vv_sentences`. The inner vectors are anonymous; they require no name. This definition initializes the `vv_sentences` object with one element, an empty `vector<string>` object.

The current inner `vector` will always be available as `vv_sentences.back()`:

```
vv_sentences.back().emplace_back(s);
```

When we've completed one inner `vector`, we simply create a new one with:

```
vv_sentences.emplace_back();
```

This creates a new anonymous `vector<string>` object and *emplaces* it at the back of the `vv_sentences` object.

## Create an efficient ToDo list using multimap

An ordered task list (or a *ToDo list*) is a common computing application. Formally stated, it's a list of tasks associated with a priority, sorted in reverse numerical order.

You may be tempted to use a `priority_queue` for this, because as the name implies, it's already sorted in priority (reverse numerical) order. The disadvantage of a `priority_queue` is that it has no iterators, so it's difficult to operate upon individual elements without pushing and popping items to and from the queue.

For this recipe, we'll use a `multimap` for the ordered list. The `multimap` is an *associative container* which keeps items in order, and it can be accessed using *reverse iterators* for the proper sort order.

## How to do it

This is a short and simple recipe that initializes a `multimap` and prints it in reverse order.

- We start with a type alias for our `multimap`:

  ```
  using todomap = multimap<int, string>;
  ```

  Our `todomap` is a `multimap` with an `int` key and a `string` payload.

- And we use a namespace alias for `ranges`:

  ```
  namespace views = std::views;
  ```

- We use a small utility function for printing the `todomap` in reverse order:

  ```
  void rprint(const todomap& todo) {
      for (auto const& [priority, task] :
        todo | views::reverse) {
          println("{}: {}", priority, task);
      }
      print("\n");
  }
  ```

  This uses `views::reverse` view to print the `todomap` in reverse order.

  The `main()` function is short and sweet:

  ```
  int main() {
      todomap todo {
          {1, "wash dishes"},
          {0, "watch teevee"},
          {2, "do homework"},
          {0, "read comics"}
  ```

```
    };
    rprint(todo);
}
```

We initialize the `todomap` with tasks. Notice that the tasks are not in any particular order, but they do have assigned priority values in their keys. The `rprint()` function will print them in descending priority order.

- The output looks like this:

```
$ ./todo
2: do homework
1: wash dishes
0: read comics
0: watch teevee
```

The ToDo list prints out in priority order, just as we need it.

## How it works

This is a short and simple recipe. It uses a `multimap` container to hold items for a prioritized list.

The only trick is in the `rprint()` function:

```
void rprint(const todomap& todo) {
    for (auto const& [priority, task] : todo | views::reverse) {
        println("{}: {}", priority, task);
    }
    print("\n");
}
```

While it's not possible to change the sort order of a multimap, `views::reverse` provides a reverse-ordered view that allows us to print our prioritized list in descending order.

## Subscribe to Deep Engineering

Join thousands of developers and architects who want to understand how software is changing, deepen their expertise, and build systems that last.

Deep Engineering is a weekly expert-led newsletter for experienced practitioners, featuring original analysis, technical interviews, and curated insights on architecture, system design, and modern programming practice.

Scan the QR or visit the link to subscribe for free.



`https://packt.link/deep-engineering-newsletter`

# 4

# STL Compatible Iterators

Iterators are a fundamental concept in the STL. Iterators are implemented with the semantics of C pointers, using the same increment, decrement, and dereference operators. The pointer idiom is already familiar and it allows *algorithms* such as `std::sort` and `std::transform` to work with both STL containers and primitive memory buffers.

This chapter contains the following recipes:

- Create an iterable range
- Use traits with your iterators for STL compatibility
- Use iterator adaptors to populate STL containers
- Iterate backward with a reverse adaptor
- Iterate objects of unknown length with a sentinel
- Create a generator with iterators
- Create a generator with coroutines
- Build a zip iterator adaptor
- Create a random-access iterator

Before we dive into the recipes, let's first take a quick look at how iterators work.

## Iterators are fundamental

The STL uses iterators to navigate the elements of its container classes. Most containers include `begin()` and `end()` iterators. These are usually implemented as member functions that return an iterator object. The `begin()` iterator points to the initial container element, and the `end()` iterator points *past* the final element:

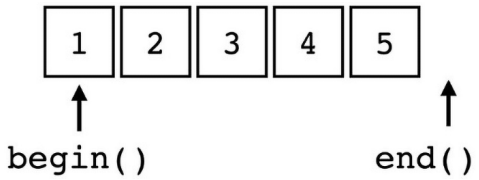



*Figure 4.1: The* `begin()` *and* `end()` *iterators*

The `end()` iterator may function as a *sentinel* to mark the ending boundary for containers of indeterminate length. We'll see some examples of that in this chapter.

Most STL containers define their own specific iterator type. For example, for a `vector` of `int`:

```
std::vector<int> v;
```

The iterator type would be defined as:

```
std::vector<int>::iterator v_it;
```

You can see how this could easily get out of hand. If we had a `vector` of `vector` of `string`:

```
std::vector<std::vector< std::string>> v;
```

Its iterator type would be:

```
std::vector<std::vector< std::string>>::iterator v_it;
```

Fortunately, C++11 gave us *automatic type deduction* and the `auto` type. By using `auto`, we rarely need to use the full iterator type definition. For example, if we need an iterator in a `for` loop, we can use the `auto` type:

```
for (auto v_it = v.begin(); v_it != v.end(); ++v_it) {
    println("{}", *v_it);
}
```

Notice the use of the dereference operator * to access the elements from the iterator. This is the same syntax you would use to dereference a pointer:

```
const int a[] {1, 2, 3, 4, 5};
size_t count {sizeof(a) / sizeof(int)};
for (const int* p = a; count > 0; ++p, --count) {
```

```
    println("{}", *p);
}
```

This also means that you can use a range-based `for` loop with either a primitive array:

```
const int a[] {1, 2, 3, 4, 5};
for (auto e : a) {
    println("{}", e);
}
```

Or with an STL container:

```
std::vector<int> v {1, 2, 3, 4, 5};
for (auto e : v) {
    println("{}", e);
}
```

The range-based `for` loop is just a shorthand for a C-style `for` loop with iterators:

```
auto begin_it = std::begin(v);
auto end_it = std::end(v);
for (; begin_it != end_it; ++begin_it) {
  auto e = *begin_it;
  println("{}", e);
}
```

Because iterators use the same syntax as a primitive pointer, the range-based `for` loop works the same with either container.

Notice that the range-based `for` loop calls `std::begin()` and `std::end()`, instead of directly calling the `begin()` and `end()` member functions. The `std::` functions call the member functions to get the iterators. So, why not just call the member functions? The `std::` non-member functions are designed to also work with primitive arrays. That's why a `for` loop works with an array:

```
const int arr[] {1, 2, 3, 4, 5};
for (auto e : arr) {
    println("{}", e);
}
```

Output:

```
1 2 3 4 5
```

For most purposes, I tend to favor the member functions `begin()` and `end()` because they are more explicit. Others favor the `std::` non-member functions because they are more general. Six or half a dozen; I suggest you pick a style and stick with it.

## Iterator categories

Prior to C++20, iterators were divided into categories based on their capabilities:

<table>
<tr><th colspan="5">Iterator Category</th><th>Iterator Capability</th></tr>
<tr><td rowspan="5">Contiguous Iterator</td><td rowspan="5">Random Access Iterator</td><td rowspan="4">Bidirectional Iterator</td><td rowspan="3">Forward Iterator</td><td>Input Iterator</td><td>● Read<br>● Increment once</td></tr>
<tr><td></td><td>● Increment multiple times</td></tr>
<tr><td colspan="2"></td><td>● Decrement</td></tr>
<tr><td colspan="3"></td><td>● Random access</td></tr>
<tr><td colspan="4"></td><td>● Contiguous storage (such as an array)</td></tr>
<tr><td colspan="6">When any of the above iterators can also write, they are called mutable iterators.</td></tr>
<tr><td colspan="5">Output Iterator</td><td>● Write<br>● Increment once</td></tr>
</table>

*Table 4.1: Iterator categories*

These categories are hierarchical, where the more capable iterators inherit the capabilities of the less capable iterators. In other words, the *input iterator* can read and increment once. The *forward iterator* has the capabilities of the Input Iterator *plus* it can increment multiple times. The *bidirectional iterator* has those capabilities *plus* it can decrement. And on down the list.

The *output iterator* can write and increment once. If any of the other iterators can also write, it is considered a *mutable iterator*.

## Iterator concepts

While the pre-C++20 iterator categories remain valid, C++20 introduced *concepts* and *constraints*, which provide compile-time semantics for clearer diagnostics. A concept is simply a

named constraint that restricts the types of arguments to a template function or class and helps the compiler choose appropriate specializations.

Beginning with C++20, the STL defines iterators in terms of concepts instead of categories. Each of these concepts is in the `std::` namespace.

| Concept | Description |
|---|---|
| `indirectly_readable` | An iterator can be read by the dereference operator, `*`. This includes pointers, smart pointers, and input iterators. |
| `indirectly_writable` | The object reference of the iterator is writable. |
| `weakly_incrementable` | This can be incremented with `++` but does not preserve equality. For example, where `a == b`, `++a` may not equal `++b`. |
| `incrementable` | This can be incremented with `++` and equality is preserved. |
| `input_or_output_iterator` | An iterator can be incremented and dereferenced. Every iterator must satisfy this concept. |
| `sentinel_for` | A sentinel iterator is used to find the end of an object of indeterminate size, such as an input stream. |
| `sized_sentinel_for` | A sentinel iterator may be used with another iterator and the `-` operator to determine its distance in constant time. |
| `input_iterator` | An iterator that may be read and incremented. |
| `output_iterator` | An iterator that may be written to and incremented. |
| `forward_iterator` | This modifies `input_iterator` to include `incrementable`. |
| `bidirectional_iterator` | This modifies `forward_iterator` by adding the ability to decrement with the `--` operator. It preserves equality. |
| `random_access_iterator` | This modifies `bidirectional_iterator` by adding support for the `+`, `+=`, `-`, `-=`, and `[]` operators. |
| `contiguous_iterator` | This modifies `random_access_iterator` to indicate contiguous storage. |

*Table 4.2: C++20 iterator concepts*

You can use these concepts to constrain the arguments of a template:

```
template<typename T>
requires std::random_access_iterator<typename T::iterator>
void printc(const T & c) {
        for(auto e : c) {
        println("{}", e);
    }
    print("\n");
    println("element 0: {}", c[0]);
}
```

This function requires a `random_access_iterator`. If we call it with a `list`, which is not a random-access container, the compiler will give us an error.

```
int main() {
    list<int> c {1, 2, 3, 4, 5};
    printc(c);
}
```

The `list` iterator type does not support the `random_access_iterator` concept. So, the compiler gives me an error:

```
error: no matching function for call to 'printc(std::__cxx11::list<int>&)'
   27 |     printc(c);
      |     ~~~~~~^~~
note: candidate: 'template<class T>  requires  random_access_iterator<typename
T::iterator> void printc(const T&)'
   16 | void printc(const T & c) {
      |      ^~~~~~
note:   template argument deduction/substitution failed:
note: constraints not satisfied
```

This is the error output from GCC. Your errors may look different.

If I call the same template function with a `vector`, which is a random-access container:

```
int main() {
    vector<int> c{ 1, 2, 3, 4, 5 };
    printc(c);
}
```

Now it compiles and runs without errors:

```
$ ./working
1 2 3 4 5
element 0: 1
```

While there are different types of iterators for different types of capabilities (and concepts), the complexity is there to support ease of use.

By using these concepts, diagnostics are moved from run time to the compiler, resulting in smaller, more efficient, and more secure code.

# Create an iterable range

This recipe describes a simple class that generates an iterable range, suitable for use with the range-based `for` loop. The idea is to create a `sequence generator` that iterates from a beginning value to an ending value.

To accomplish this task, we need an iterator class, along with the object interface class.

## How to do it

There are two major parts to this recipe: the main interface class `Seq`, and a nested `iterator` class.

- First, we'll define the `Seq` class. It only needs to implement the `begin()` and `end()` member functions:

```
template<typename T>
class Seq {
    T seq_start {};
    T seq_end {};
public:
    Seq(T start, T end) : seq_start {start}, seq_end {end}
      {}
    iterator begin() const { return iterator {seq_start}; }
    iterator end() const { return iterator {seq_end}; }
};
```

The constructor sets up the `seq_start` and `seq_end` variables. These are used to construct the `begin()` and `end()` iterators, respectively. The member functions `begin()` and `end()` return `iterator` objects.

- The `iterator` class is normally defined inside the public section of the container class. This is called a *member class* or a *nested class*. We'll insert it right after the `Seq` constructor:

```
class iterator {
    T it_value{};
public:
    explicit iterator(T position = 0) : it_value {position}
        {}

    T operator*() const { return it_value; }

    iterator& operator++() {
        ++it_value;
        return *this;
    }

    bool operator!=(const Seq::iterator& other) const {
        return it_value != other.it_value;
    }
};
```

  It's traditional to name the iterator class `iterator`. This allows it to be referenced as `Seq<`*type*`>::iterator`.

  The `iterator` constructor is qualified `explicit` to avoid implicit conversions.

  The `it_value` variable is maintained by the iterator. This is used to return a value from the pointer dereference.

  The minimum requirement for supporting the range-based `for` loop is a dereference operator `*`, a pre-increment operator `++`, and the not-equal comparison operator `!=`.

- Now we can write a `main()` function to test our sequence generator:

```
int main() {
    Seq<int> r(100, 110);

    for (auto v : r) {
        print("{} ", v);
    }
```

```
    print("\n");
}
```

This constructs a `Seq` object and prints out its sequence.

The output looks like this:

```
$ ./seq
100 101 102 103 104 105 106 107 108 109
```

## How it works

The point of this recipe is to make a sequence generator that works with a range-based `for` loop. Let's first consider the equivalent code for the range-based `for` loop:

```
{
    auto begin_it = std::begin(container);
    auto end_it   = std::end(container);
    for (; begin_it != end_it; ++begin_it){
        auto e = *begin_it;
        println("{}", e);
    }
}
```

From this equivalent code, we can deduce the minimum requirements for an object to support the range-based `for` loop:

- `begin()` and `end()` iterators
- Iterator support for the not-equal comparison `!=` operator
- Iterator support for the prefix increment `++` operator
- Iterator support for the dereference `*` operator

Our main `Seq` class interface only has three public member functions: the constructor, and the `begin()` and `end()` iterators:

```
Seq(T start, T end) : seq_start {start}, seq_end {end} {}
iterator begin() const { return iterator {seq_start}; }
iterator end() const { return iterator {seq_end}; }
```

The implementation of the `Seq::iterator` class carries the actual payload:

```
class iterator {
T it_value{};
```

This is the common configuration because the payload is only accessed through iterators.

We've implemented only the three operators we need:

```
T operator*() const { return it_value; }

iterator& operator++() {
    ++it_value;
    return *this;
}

bool operator!=(const Seq::iterator& other) const {
    return it_value != other.it_value;
}
```

This is all we need to support the range-based `for` loop:

```
Seq<int> r(100, 110);
for (auto v : r) {
    print("{} ", v);
}
print("\n");
```

Output:

```
100 101 102 103 104 105 106 107 108 109
```

## There's more...

It's traditional, but not required, to define the iterator as a member class of the container. This allows the `iterator` type to be subordinate to the container type:

```
Seq<int>::iterator it = r.begin();
```

This is less important post-C++11 because of the `auto` type, but it's still considered best practice.

# Use traits with your iterators for STL compatibility

Many STL algorithms require iterators to conform to certain traits. Unfortunately, these requirements may be inconsistent across compilers, systems, and C++ versions.

For our purposes, we'll use the class from the *Create an iterable range* recipe to illustrate the issue. You may find this makes more sense if you read that recipe before continuing.

In `main()`, if I add a call to the `std::minmax_element()` algorithm:

```
Seq<int> r {100, 110};
auto [min_it, max_it] = minmax_element(r.begin(), r.end());
print("{} - {}\n", *min_it, *max_it);
```

This does not compile. The error messages are vague, cryptic, and cascading, but if you look closely, you'll see that our iterator does not meet the requirements to be compatible with this algorithm.

Okay, let's fix that.

**Note**

The gcc compiler gives a cyrptic error like:

```
In instantiation of 'constexpr std::pair<_FIter, _FIter>
std::__minmax_element(_ForwardIterator, _ForwardIterator, _Compare)
[with _ForwardIterator = Seq<int>::iterator; _Compare =
__gnu_cxx::__ops::_Iter_less_iter]':
```

...followed by a cascade of other errors. You can see that it has something to do with the `_ForwardIterator` token. This is a clue, but not very helpful unless you're familiar with this issue.

## How to do it

We need to make a few simple additions to our iterator to make it compatible with the algorithm. Our iterator needs to meet the minimum requirements for a *forward iterator*, so let's start there:

- We have almost all the operators necessary for a forward iterator. The only one we're missing is the equality comparison operator `==`. We can easily add this to our iterator with an `operator==()` overload:

  ```
  bool operator==(const iterator& other) const noexcept {
      return it_value == other.it_value;
  }
  ```

  Interestingly, this makes the code compile and run on some systems, but not on `Clang`, where we get the error message:

  ```
    No type named 'value_type' in 'std::iterator_traits<Seq<int>::iterator>'
  ```

  This tells me that we need to set up the traits in the iterator.

- The `iterator_traits` class looks for a set of *type definitions* (implemented as `using` aliases) in the `iterator` class:

  ```
  public:
      using iterator_concept  = std::forward_iterator_tag;
      using iterator_category = std::forward_iterator_tag;
      using value_type        = std::remove_cv_t<T>;
      using difference_type   = std::ptrdiff_t;
      using pointer           = const T*;
      using reference         = constT&;
  ```

  I tend to put these at the top of the `public:` section of the `iterator` class, where they'll be easy to find.

  This gives us a fully conforming *forward iterator* class, and the code runs on all the compilers I have.

## How it works

The `using` statements are traits that may be used to define what capabilities the iterator can perform. Let's look at each of them:

```
using iterator_concept  = std::forward_iterator_tag;
using iterator_category = std::forward_iterator_tag;
```

The first two are the *category* and the *concept*, and both are set to `forward_iterator_tag`. This value indicates that the iterator conforms to the forward iterator specification.

Some code doesn't look at those values, and instead looks for individual settings and capabilities:

```
using value_type        = std::remove_cv_t<T>;
using difference_type   = std::ptrdiff_t;
using pointer           = const T*;
using reference         = constT&;
```

The `value_type` alias is set to `std::remove_cv_t<T>`, which is the type of the value, with any `const` qualifier removed.

The `difference_type` alias is set to `std::ptrdiff_t`, a special type for pointer differences.

The `pointer` and `reference` aliases are set to `const`-qualified versions of the pointer and reference, respectively.

Defining these type aliases is a basic requirement for most iterators.

## There's more...

It's worth noting that defining these traits allows us to use concept-restricted templates with our iterator. For example:

```
template<typename T>
requires std::forward_iterator<typename T::iterator>
void printc(const T & c) {
    for(auto v : c) {
        print("{} ", v);
    }
    print("\n");
}
```

This function that prints our sequence is restricted by the `forward_iterator` concept. If our class did not qualify, it wouldn't compile.

We can also use the `ranges::` versions of the algorithms:

```
auto [min_it, max_it] = ranges::minmax_element(r);
```

This makes it more convenient to use our iterators.

We can test for `forward_range` compatibility with a static assertion:

```
static_assert(ranges::forward_range<Seq<int>>);
```

# Use iterator adaptors to populate STL containers

An iterator is essentially an abstraction. It has a specific interface and is used in specific ways. But beyond that, it's just code and it can be used for other purposes. An *iterator adaptor* is a class that looks like an iterator but does something else.

The STL comes with an assortment of iterator adaptors. Often used with the `algorithm` library, they are quite useful. The STL iterator adaptors generally fall into three categories:

**Insert iterators**, or *inserters*, are used to insert elements into a container.

**Stream iterators** read from and write to a stream.

**Reverse iterators** reverse the direction of an iterator.

We will briefly cover each of these in this recipe.

## How to do it

In this recipe, we'll consider a few examples of STL iterator adaptors.

- We'll start with a simple function to print the contents of a container:

  ```
  void printc(const auto & v, const string_view s = "") {
      if (!s.empty()) print("{}: ", s);
      for (auto e : v) print("{} ", e);
      print("\n");
  }
  ```

  The `printc()` function allows us to easily view the results of our algorithms. It includes an optional `string_view` argument for a description.

- In our `main()` function, we'll define a couple of deque containers. We're using deque containers so we can insert at both ends:

```
int main() {
    deque<int> d1 {1, 2, 3, 4, 5};
    deque<int> d2(d1.size());
    copy(d1.begin(), d1.end(), d2.begin());
    printc(d1);
    printc(d2, "d2 after copy");
}
```

Output:

```
  1 2 3 4 5
  d2 after copy: 1 2 3 4 5
```

We defined deque `d1` with five `int` values, and `d2` with space for the same number of elements. The `copy()` algorithm will not allocate space, so `d2` must have room for the elements.

The `copy()` algorithm takes three iterators: the `begin` and `end` iterators indicate the range of elements to copy from, and the `begin` iterator of the destination range. It does not check the iterators to make sure they're valid. (Try this without allocating space in a `vector` and you should get a *segmentation fault* error.)

We call `printc()` on both containers to show the results.

- The `copy()` algorithm is not always convenient for this. Sometimes you want to copy and add elements to the end of a container. It would be nice to have an algorithm that calls `push_back()` for each element. This is where an iterator adaptor is useful. Let's add some code at the end of `main()`:

```
copy(d1.begin(), d1.end(), std::back_inserter(d2));
printc(d2, "d2 after back_inserter");
```

Output:

```
  d2 after back_inserter: 1 2 3 4 5 1 2 3 4 5
```

`back_inserter()` is an *insert iterator adaptor* that calls `push_back()` for each item assigned to it. You can use it anywhere an output iterator is expected.

- There's also a `front_inserter()` adaptor for when you want to insert at the front of a container:

```
deque<int> d3 {47, 73, 114, 138, 54};
copy(d3.begin(), d3.end(), std::front_inserter(d2));
printc(d2, "d2 after front_inserter");
```

Output:

```
d2 after front_inserter: 54 138 114 73 47 1 2 3 4 5 1 2 3 4 5
```

The `front_inserter()` adaptor inserts elements at the front using the container's `push_front()` method. Notice that the elements in the destination are reversed, because each element is inserted before the previous one.

- If we want to insert in the middle, we can use the `inserter()` adaptor:

```
auto it2 = d2.begin() + 2;
copy(d1.begin(), d1.end(), std::inserter(d2, it2));
printc(d2, "d2 after middle insert");
```

Output:

```
d2 after middle insert: 54 138 1 2 3 4 5 114 73 47 ...
```

The `inserter()` adaptor takes an iterator for the insertion begin point.

- *Stream iterators* are convenient for reading from and writing to `iostream` objects. This is `ostream_iterator()`:

```
print("ostream_iterator: ");
copy(d1.begin(), d1.end(), ostream_iterator<int>(cout));
print("\n");
```

Output:

```
ostream_iterator: 12345
```

- And here is `istream_iterator()`:

```
vector<string> vs{};
copy(istream_iterator<string>(cin),
    istream_iterator<string>(), back_inserter(vs));
printc(vs, "vs from istream");
```

Output:

```
$ ./working < five-words.txt
vs from istream: this is not a haiku
```

The `istream_iterator()` adaptor will return an end iterator by default, if no stream is passed.

- *Reverse adaptors* are included with most containers, as member functions `rbegin()` and `rend()`:

```
for(auto it = d1.rbegin(); it != d1.rend(); ++it) {
    print("{} ", *it);
}
print("\n");
```

Output:

```
5 4 3 2 1
```

The reverse adaptors, `rbegin()` and `rend()`, simply iterate in reverse. This is handy when you need to iterate a container from the end to the beginning.

## How it works

Iterator adaptors work by wrapping around an existing container. When you call an adaptor, like `back_inserter()` with a container object:

```
copy(d1.begin(), d1.end(), back_inserter(d2));
```

The adaptor returns an object that *mimics* an iterator, in this case a `std::back_insert_iterator` object, which calls the `push_back()` method on the container object each time a value is assigned to the iterator. This allows the adaptor to be used in place of an iterator, while performing its useful task.

The `istream_adaptor()` also requires a *sentinel*. A sentinel signals the end of an iterator of indeterminate length. When we read from a stream, we don't know how many objects are in the stream until we hit the end. When the stream hits the end, the sentinel will compare equal with the iterator, which signals the end of the stream. The `istream_adaptor()` will create a sentinel when it's called without a parameter:

```
auto it = istream_adaptor<string>(cin);
auto it_end = istream_adaptor<string>();  // creates sentinel
```

This allows us to test for the end of a stream, as we would with any container:

```
for (auto it = istream_iterator<string>(cin);
        it != istream_iterator<string>();
        ++it) {
    print("{} ", *it);
}
print("\n");
```

Output:

```
$ ./working < five-words.txt
this is not a haiku
```

# Iterate backward with a reverse adaptor

A *reverse iterator adaptor* is an abstraction that reverses the direction of an iterator class. It requires a bidirectional iterator.

## How to do it

Most bidirectional containers in the STL include a reverse iterator adaptor. Other containers, such as the primitive C-array, do not. Let's look at some examples:

- Let's start with a `printc()` function, as we've used throughout this chapter:

  ```
  void printc(const auto & c, const string_view s = "") {
      if (s.size()) print("{}: ", s);
      for (auto e : c) print("{} ", e);
      print("\n");
  }
  ```

  This uses a range-based `for` loop to print the elements of a container.

- The range-based `for` loop works even with primitive C-arrays, which have no iterator class. So, our `printc()` function already works with a C-array:

```
int main() {
    int array[] {1, 2, 3, 4, 5};
    printc(array, "c-array");
}
```

We get this output:

```
c-array: 1 2 3 4 5
```

- We can use the `begin()` and `end()` iterator adaptors to create normal forward iterators for the C-array:

```
auto it = std::begin(c);
auto end_it = std::end(c);
while (it != end_it) {
    print("{} ", *it++);
}
```

Output from the `while` loop:

```
1 2 3 4 5
```

- Or we can use the `rbegin()` and `rend()` reverse iterator adaptors to create reverse iterators for the C-array:

```
auto it = std::rbegin(c);
auto end_it = std::rend(c);
while (it != end_it) {
    print("{} ", *it++);
}
```

Now our output is reversed:

```
5 4 3 2 1
```

- We can even create a modified version of `printc()` that prints in reverse:

```
void printr(const auto & c, const string_view s = "") {
    if (s.size()) print("{}: ", s);
    auto it = std::rbegin(c);
    auto end_it = std::rend(c);
    while (it != end_it) {
        print("{} ", *it++);
    }
    print("\n");
}
```

When we call it with the C-array:

```
printr(array, "rev c-array");
```

We get this output:

```
  rev c-array: 5 4 3 2 1
```

- Of course, this works just as well with any bidirectional STL container:

```
vector<int> v {1, 2, 3, 4, 5};
printc(v, "vector");
printr(v, "rev vector");
```

Output:

```
  vector: 1 2 3 4 5
  rev vector: 5 4 3 2 1
```

## How it works

A normal `iterator` class has a `begin()` iterator that points to the first element, and an `end()` iterator that points *past* the last element:

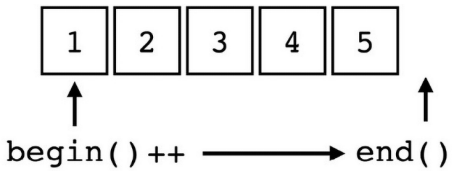



*Figure 4.2: Forward iterator*

You iterate the container by incrementing the `begin()` iterator with the `++` operator, until it reaches the value of the `end()` iterator.

A reverse iterator adaptor *intercepts the iterator interface* and turns it around so the `begin()` iterator points to the last element, and `end()` iterator points *before* the first element. The `++` and `--` operators are also inverted:

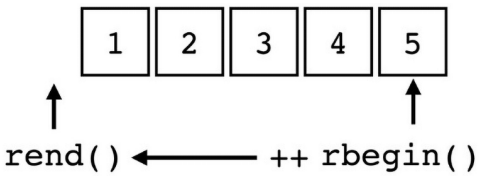



*Figure 4.3: Reverse iterator adaptor*

In the reversed iterator, the `++` operator decrements and the `--` operator increments.

It's worth noting that most bidirectional STL containers already include a reverse iterator adaptor, accessible by member functions `rbegin()` and `rend()`:

```
vector<int> v;
it = v.rbegin();
it_end = v.rend();
```

These iterators will operate in reverse and are suitable for many purposes, such as prioritized lists, sortable displays, or any context where you need to traverse a data set in descending order.

# Iterate objects of unknown length with a sentinel

Some objects don't have a specific length. To know their length, you need to iterate through all their elements. For example, elsewhere in this chapter we've seen a *generator* that doesn't have a specific length. A more common example would be a C-string.

A C-string is a primitive C-array of characters, terminated with a null `'\0'` value.

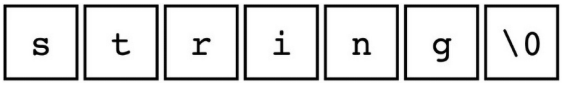



*Figure 4.4: A C-string with its null terminator*

We use C-strings all the time, even if we don't realize it. Any *literal* string in C/C++ is a C-string:

```
std::string s {"string"};
```

Here, the STL string `s` is initialized with a literal string. The literal string is a C-string. If we look at the individual characters in hexadecimal, we'll see the null terminator:

```
for (char c : "string") {
    print("{:02x} ", c);
}
```

We use the format string `{:02x}` to print the character in hexadecimal because the null terminator would be invisible when printed as a character.

The word `"string"` has six letters. The output from our loop shows seven elements in the array:

```
73 74 72 69 6e 67 00
```

The seventh element is the *null terminator*.

The loop sees the primitive C-array of characters, with seven values. The fact that it's a string is an abstraction invisible to the loop. If we want the loop to treat it like a string, we'll need an iterator and a *sentinel*.

A sentinel is an object that signals the end of an iterator of indeterminate length. When the iterator hits the end of the data, the sentinel will compare equal to the iterator.

To see how this works, let's build an iterator for C-strings!

## How to do it

To use a sentinel with a C-string, we need to build a custom iterator. It doesn't need to be complicated, just the essentials for use with a range-based `for` loop: a dereference operator `*`, a pre-increment operator `++`, and the not-equal comparison operator `!=`.

- We'll start with a couple of convenient definitions:

```
using sentinel_t = const char;
constexpr sentinel_t nullchar {'\0'};
```

  The `using` alias for `sentinel_t` is `const char`. We'll use this for the sentinel in our class.

  We also define the constant `nullchar` for the null character terminator.

- Now we can define our iterator type:

```
class cstr_it {
    const char *s {};
public:
    explicit cstr_it(const char *str) : s {str} {}
    char operator*() const { return *s; }

    cstr_it& operator++() {
        ++s;
        return *this;
    }

    bool operator!=(sentinel_t) const {
        return s != nullptr&&*s!=nullchar;
    }

    cstr_it begin() const { return *this; }
    sentinel_t end() const { return nullchar; }
};
```

  This is short and simple. It's the minimum necessary for a range-based `for` loop. Notice the `end()` function returns a `nullchar` and the `operator!=()` overload compares against the `nullchar`. That's all we need for the sentinel.

- Now we can define a function for printing our C-string using the sentinel:

```
void print_cstr(const char * s) {
    print("{}: ", s);
    for (char c : cstr_it(s)) {
        print("{:02x} ", c);
    }
    print("\n");
}
```

In this function we first print the string. Then we print each individual character as a hexadecimal value, using the `cstr_it` iterator to traverse the string.

- Now we can call `print_cstr()` from our `main()` function:

```
int main() {
    const char carray[] {"array"};
    print_cstr(carray);
    const char * cstr {"c-string"};
    print_cstr(cstr);
}
```

The output looks like this:

```
  array: 61 72 72 61 79
  c-string: 63 2d 73 74 72 69 6e 67
```

Notice that there are no extraneous characters and no null terminators. This is because our sentinel tells the `for` loop to stop when it sees the `nullchar` sentinel.

## How it works

The sentinel part of the iterator class is very simple. We can easily use the null terminator as the sentinel value by returning it in the `end()` function:

```
sentinel_t end() const { return nullchar; }
```

Then the not-equal comparison operator can test for it:

```
bool operator!=(sentinel_t) const {
    return s != nullptr&&*s!=nullchar;
}
```

Notice that the parameter is just a type (`sentinel_t`). A parameter type is necessary for the function signature, but we don't need the value. All that's necessary is to compare the current iterator with the sentinel.

This technique should be useful whenever you have a type or class that doesn't have a predetermined endpoint for comparison.

# Create a generator with iterators

A *generator* is an iterator that generates a sequence of values. It does not use a container. It creates values on the fly, returning one value at a time as needed. A C++ generator stands on its own; it does not need to wrap around another object.

In this recipe, we'll build a generator for a *Fibonacci sequence*. This is a sequence where each number is the sum of the previous two numbers in the sequence, starting with 0 and 1:

$$F_0 = 0,\ F_1 = 1$$

and

$$F_n = F_{n-1} + F_{n-2}$$

for $n > 1$

*Figure 4.5: Definition of a Fibonacci sequence*

The first ten values of the Fibonacci sequence, not counting zero, are: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55. This is a close approximation of the *golden ratio* found in nature.

## How to do it

A Fibonacci sequence is often created with a *recursive loop*. Recursion in a generator can be difficult and resource-intensive, so instead we just save the previous two values in the sequence and add them together. This approach is more efficient.

- First we define a function to print the sequence:

```
void printc(const auto & v, const string_view s = "") {
    if (!s.empty()) print("{}: ", s);
    for (auto e : v) print("{} ", e);
    print("\n");
}
```

We've used this `printc()` function before. It prints an iterable range, along with a description string, if provided.

- Our class begins with a *type alias*, and a few object variables, all in the `private` section

```
template<typename T>
class fib_generator {
    T stop {};
    T count {};
    T n1 {0};
    T n2 {1};
```

The `stop` variable will be used later as a *sentinel*. It's set to the number of values to generate. `count` is used to keep track of how many values we've generated. `n1` and `n2` are the previous two sequence values, used for calculating the next value.

- Still in the `private` scope, we add a simple function for calculating the next value in the Fibonacci sequence

```
constexpr void do_fib() {
    const T prev_n2 = n2;
    n2 += n1;
    n1 = prev_n2;
}
```

- Now in the `public` scope, we have a simple constructor with a default value:

```
public:
    explicit fib_generator(T fib_stop = 0)
        : stop {fib_stop} {}
```

This constructor is used without an argument to create a sentinel. The `fib_stop` argument initializes the `stop` variable to hold the number of values to generate.

- To complete the public functions we need the operator overloads expected of a *forward iterator*, along with `begin()` and `end()` iterator functions:

```
T operator*() const { return n2; }

constexpr fib_generator& operator++() {
    do_fib();
    ++count;
    return *this;
}
```

```
fib_generator operator++(int) {
    auto temp = *this;
    ++*this;
    return temp;
}

bool operator!=(const fib_generator& o) const {
    return count != o.count;
}

bool operator==(const fib_generator& o) const {
    return count == o.count;
}

const fib_generator& begin() const { return *this; }

const fib_generator end() const {
    auto sentinel = fib_generator();
    sentinel.count = stop;
    return sentinel;
}

fib_t size() { return stop; }
```

Though the post-increment ++ and equality comparison == operators are not required for this example, it's good practice to include them. These are required when the iterator is used with the `algorithms` library.

There's also a simple `size()` function which can be useful if you need to initialize a target container for a copy operation.

- Now we can use the generator in our main function with a simple call to `printc()`:

```
int main() {
    printc(fib_generator(10));
}
```

This creates an anonymous `fib_generator` object to pass to the `printc()` function.

- We get this output with the first `10` Fibonacci numbers, not including zero:

```
1 1 2 3 5 8 13 21 34 55
```

## How it works

The `fib_generator` class operates as a forward iterator, simply because it provides all the necessary interface functions:

```
template<typename T>
fib_generator {
public:
    T operator*() const;
    constexprfib_generator&operator++();
    fib_generator operator++(int);
    bool operator!=(constfib_generator&o)const;
    bool operator==(constfib_generator&o)const;
    const fib_generator& begin() const;
    const fib_generator end() const;
};
```

For the purposes of a range-based `for` loop, this is an iterator because it looks like an iterator.

The value is calculated in the `do_fib()` function:

```
constexpr void do_fib() {
    const T prev_n2 = n2;
    n2 += n1;
    n1 = prev_n2;
}
```

This simply adds `n1 + n2`, stores the result in `n2`, and stores the old `n2` in `n1`, setting it up for the next iteration.

The dereference operator * returns the value from `n2`, which is the next value in the sequence:

```
T operator*() const { return n2; }
```

The `end()` function creates a *sentinel* `fib_generator` object, where the `count` variable is equal to the `stop` variable:

```
const fib_generator end() const {
    auto sentinel = fib_generator();
    sentinel.count = stop;
    return sentinel;
}
```

Now the not-equal comparison operator `!=` can easily detect the end of the sequence:

```
bool operator!=(const fib_generator& o) const {
    return count != o.count;
}
```

## There's more...

If we want to make our generator work with the `algorithm` library, we need to provide the `traits` aliases. These go at the top of the `public` scope:

```
public:
    using iterator_concept  = std::forward_iterator_tag;
    using iterator_category = std::forward_iterator_tag;
    using value_type        = std::remove_cv_t<fib_t>;
    using difference_type   = std::ptrdiff_t;
    using pointer           = const fib_t*;
    using reference         = const fib_t&;
```

Now we can use our generator with algorithms:

```
fib_generator fib(10);
auto x = fib | ranges::views::transform(
    [](unsigned long x){ return x * x; });
printc(x, "squared:");
```

This uses the `ranges::views` version of the `transform()` algorithm to square each value from the generator. The resulting object can be used wherever you would use an iterator. We get this output from the `printc()` call:

```
squared:: 1 1 4 9 25 64 169 441 1156 3025
```

## Create a generator with coroutines

*Coroutines* are an asynchronous concurrency paradigm. Introduced in C++20, coroutines allow functions to pause and resume, providing a mechanism for *lazy evaluation*. Simply stated, lazy evaluation means that a calculation is not performed until required. This makes coroutines well-suited for implementing generators, especially when significant complexity is involved in generating their values.

C++23 provides a `generator` class that uses the `coroutine` library. Using the `generator` class, a simple generator may look like this:

```
std::generator<long> gen_seq(long count) {
    for (long i {}; i < count; ++i) {
        co_yield i;
    }
}

int main() {
    for (auto i : gen_seq(10)) {
        print("{} ", i);
    }
    print("\n");
}
```

Output:

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

The `std::generator` class is defined in the `<generator>` header. The `co_yield` statement, introduced in C++20, makes the `gen_seq()` function a *coroutine*. Upon reaching the `co_yield` statement, the `gen_seq()` function returns a `std::generator` object to the caller with the generated value. A *coroutine function* must have a valid *coroutine class* as its return type. The `co_yield` statement suspends execution of the coroutine function to allow *lazy evaluation* of its generated values. When called again, the coroutine function continues where it left off.

In this recipe, we will create our own *generator* coroutine class using the `coroutine` library. While it often makes sense to simply use the provided `generator` library class, you may find it useful to understand how it implements the `coroutine` specification to make it work.

This recipe is more complex than the iterator generator in this chapter. It is also more complex than using the C++23 `generator` STL class. The value is in learning how coroutines and iterators can work together to create effective solutions to common problems.

We'll cover more details about coroutines in *Chapter 9*, *Concurrency and Parallelism*.

**Note**

As of mid-2026, the `std::generator` class is not yet implemented on Apple Clang. This code has been tested on GCC 15.2 and Microsoft MSVC 1944.

## How to do it

In this recipe, we'll create a simple coroutine generator class that works like the `std::generator` class provided in the C++23 STL. This will give us some insight into how to implement an iterator with a coroutine.

Coroutines are a relatively new concept in C++ and the concept may seem a bit abstract at first. There are a number of concepts involved with which you may not be familiar, especially if you're new to asynchronous programming. You may want to take your time with this or simply use the `std::generator` class if that's available to you.

- We start with the `co_gen` class. This is a template class, and it begins like any coroutine class:

```
template<typename T>
class co_gen {
public:
    struct promise_type;    // forward declaration
    using handle_type =
        std::coroutine_handle<promise_type>;
    handle_type coro {};
```

  Our class is called `co_gen`, for *coroutine generator*. This class has another class, `promise_type`, nested within it. The `promise_type` class requires a *forward declaration* so it can be used as a template parameter for the coroutine handle, which is named `coro`.

- The `coroutine_handle` template class manages the coroutine itself. We use `handle_type` as an alias for `coroutine_handle` with `promise_type` as its template parameter. The constructor takes a `handle_type` parameter and initializes `coro` with the handle of the coroutine:

```
co_gen(handle_type h) : coro(h) {}
```

- There are three more small functions in the `public` interface of our class:

```
~co_gen() {
    if (coro) coro.destroy();
}

operator bool() {
    coro.resume();
    return !coro.done();
}

T operator()() {
    return coro.promise().value;
}
```

  The destructor calls `coro.destroy()`, which destroys the coroutine.

  The `bool()` operator calls the coroutine `resume()` function, so the generator may produce a new value. It returns the negated `done()` value from the `coroutine_handle` object to indicate completion of the coroutine. This allows us to check the status of the coroutine conveniently from a `while` loop.

  The function call operator, `operator()()`, returns the most recent value from the coroutine via the `promise` object.

- Coroutines use a `promise_type` object for managing values and state
  The `promise_type` class is nested within the `co_gen` class. This provides the interface between the asynchronous coroutine and its caller:

```
struct promise_type {
    T value;

    co_gen get_return_object() {
        return {handle_type::from_promise(*this)};
```

```
    }

    std::suspend_always initial_suspend() {
        return {};
    }

    std::suspend_always final_suspend() noexcept {
        return {};
    }

    void return_void() {}

    std::suspend_always yield_value(T v) {
        value = v;
        return {};
    }

    void unhandled_exception() {
        std::exit(1);
    }
};
```

The `promise_type` class is defined as a `struct` because it has only `public` scope member functions.

The `get_return_object()` method returns a pointer to the promise object.

The `initial_suspend()` and `final_suspend()` methods return a `suspend_always` object, to indicate that this is a *lazy* execution coroutine, as opposed to a *greedy* (strict evaluation) coroutine.

The `return_void()` and `unhandled_exception()` methods are required for compliance with the coroutine standard.

The `yield_value()` method stores the value in the `promise` object and returns a `suspend_always` object.

- Now that we have a `co_gen` class, we can use it from a coroutine function, just as we would with the `std::generator` class:

```
template<typename T>
co_gen<T> gen_seq(T count) {
```

```
    for (T i {}; i < count; ++i) {
        co_yield i;
    }
}
```

The coroutine return type is a `co_gen` object, which allows us to invoke the coroutine from `main()` like this:

```
int main() {
    using ulong = unsigned long;
    auto gen = gen_seq<ulong>(10);
    while (gen) {
        print("{} ", gen());
    }
    print("\n");
}
```

This gives us the output:

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

## How it works

A coroutine is simply a concurrency paradigm that allows lazy evaluation in C++. The `coroutine_handle` type provides a hook to the suspended or executing coroutine, and the `promise_type` stores state and value to interface with the caller.

The `co_gen` class provides the hooks for the coroutine, but the coroutine itself is in the `gen_seq()` function:

```
template<typename T>
co_gen<T> gen_seq(T count) {
    for (T i {}; i < count; ++i) {
        co_yield i;
    }
}
```

The return type of the `gen_seq()` function is the `co_gen` class, but the function does not have a `return` statement. The `co_yield` statement indicates that the function is a coroutine and

returns each value in sequence to the caller. The `promise_type` class is defined within the `co_gen` class to provide the necessary hooks into the coroutine system:

```
struct promise_type {
    T value;
    suspend_always initial_suspend() { return {}; }
    suspend_always final_suspend() noexcept { return {}; }
    void return_void() {}
    void unhandled_exception() { exit(1); }

    co_gen get_return_object() {
        return {handle_type::from_promise(*this)};
    }

    std::suspend_always yield_value(T v) {
        value = v;
        return {};
    }
};
```

We've implemented the promise methods necessary for the coroutine:

- `get_return_object()` returns a pointer to the promise object
- `initial_suspend()` and `final_suspend()` return a `suspend_always` object
- `return_void()` and `unhandled_exception()` are required for compliance with the coroutine standard
- `yield_value()` stores the value in the `promise` object and returns a `suspend_always` object

This provides the basic operation of the coroutine generator. But it still doesn't operate as an iterator. We can now add iterator functionality to our generator.

## There's more...

In order to make the generator work with `ranges`, `views`, and range-based `for` loops, like the C++23 `generator` class, we need to add an iterator.

The `iterator` class is defined within the `co_gen` class. This makes the generator work as an iterable object, just as it would for a container:

```
class iterator {
     handle_type coro {};
 public:
     using iterator_concept  = std::forward_iterator_tag;
     using iterator_category = std::forward_iterator_tag;
     using value_type = std::remove_cv_t<T>;
     using difference_type = std::ptrdiff_t;
     using pointer = T*;
     usingreference=T&;

     iterator() : coro(nullptr) {}
     iterator(handle_type h) : coro(h) {}

     T& operator*() const {
         return coro.promise().value;
     }

     T* operator->() const {
         return&coro.promise().value;
     }

     iterator& operator++() {
         coro.resume();
         if (coro.done()) coro = nullptr;
         return *this;
     }

     iterator operator++(int) {
         iterator temp = *this;
         ++*this;
         return temp;
     }

     bool operator==(const iterator& other) const {
         return coro == other.coro;
     }

     bool operator!=(const iterator& other) const {
```

```
        return !(*this == other);
    }
};

iterator begin() {
    if (coro) coro.resume();
    if (coro.done()) return iterator{nullptr};
    return iterator{coro};
}

iterator end() {
    return iterator{nullptr};
}
```

Most of the `iterator` class looks exactly like we would expect of a container iterator. The main differences are in our interface between the `coroutine_handle` and the `promise` object.

A few of the iterator methods are of particular interest:

- The `coro` object is a handle to the coroutine. We use this to access the `promise` object which contains the state and data for the coroutine.
- The dereference operator `operator*()` returns the `value` from the promise object.
- The increment operator `operator++()` calls `coro.resume()` to resume a suspended coroutine.
- The `begin()` and `end()` methods are defined outside the scope of the `iterator` class, so they may operate on the generator class itself. The `begin()` method calls `coro.resume()` to resume a suspended coroutine.

We can now use our `co_gen` class just as we would the C++23 `generator` class. Here's a Fibonacci generator:

```
template<typename T>
co_gen<T> gen_fib(T count) {
    T n1 {}, n2 {1};
    auto do_fib = [&n1, &n2] {
        const T prev_n2 = n2;
        n2 += n1;
        n1 = prev_n2;
        return n1;
    };
```

```
    for (T i = 0; i < count; ++i) {
        co_yield do_fib();
    }
}
```

The Fibonacci calculation is encapsulated in a *lambda expression* for convenience. We can call this coroutine generator from `main()` as we would any range or container:

```
for (auto i : gen_fib<ulong>(10)) {
    print("{} ", i);
}
print("\n");
```

Output:

```
1 1 2 3 5 8 13 21 34 55
```

We can even pipe it into standard ranges and views:

```
auto gen2 = gen_fib<ulong>(10);
for (auto i : gen2 | views::transform([](ulong x){
        return x * x; })) {
    print("{} ", i);
}
print("\n");
```

This provides us with all the functionality of the C++23 `generator` class.

# Build a zip iterator adaptor

Many scripting languages include a function for *zipping* two sequences together. A typical zip operation will take two input sequences and return a pair of values for each position in both inputs.

Consider the case of two sequences – they may be containers, iterators, or initialization lists:

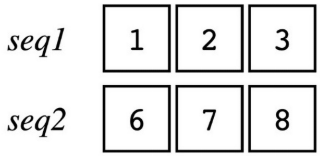



*Figure 4.6: Containers to be zipped*

We want to *zip* them together to make a new sequence with pairs of elements from the first two sequences:

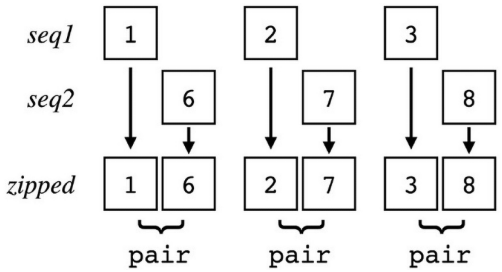



*Figure 4.7: Zip operation*

In this recipe we will use many of the techniques discussed in this chapter to build a zip iterator with an iterator adaptor.

## How to do it

In this recipe we'll build a *zip* iterator adaptor that takes two containers of the same type and zips the values into `std::pair` objects:

- In our `main()` function we want to call our adaptor with two vectors:

```
int main()
{
    vector<std::string> vec_a {"Bob", "John", "Joni"};
    vector<std::string> vec_b
        {"Dylan", "Williams", "Mitchell"};
    print("zipped: ");
    for(const auto& [a, b] : zip_iterator(vec_a, vec_b)) {
        print("[{}, {}] ", a, b);
    }
    print("\n");
}
```

This allows us to use the `zip_iterator` in place of the individual `vector` iterators.

And we expect an output like this:

```
    zipped: [Bob, Dylan] [John, Williams] [Joni, Mitchell]
```

- Our iterator adaptor is in a class called `zip_iterator`. We'll start with some type aliases for convenience:

```
template<typename T>
class zip_iterator {
    using val_t = typename T::value_type;
    using ret_t = std::pair<val_t, val_t>;
    using it_t = typename T::iterator;
```

These allow us to conveniently define objects and functions.

- We don't store any data in our iterator. We only store copies of the target containers' `begin()` and `end()` iterators:

```
it_t ita {};
it_t itb {};
it_t ita_begin {};
it_t itb_begin {};
it_t ita_end {};
it_t itb_end {};
```

`ita` and `itb` are iterators from the target containers. The other four iterators are used to generate the `begin()` and `end()` iterators for the `zip_iterator` adaptor.

- We also have a private constructor:

```
zip_iterator(it_t ita, it_t itb) : ita {ita}, itb {itb} {}
```

This is used later to construct adaptor objects, specifically for `begin()` and `end()` iterators.

- In the `public` section, we start with the iterator *traits* type definitions:

```
public:
    using iterator_concept  =
        std::forward_iterator_tag;
    using iterator_category =
        std::forward_iterator_tag;
```

```
    using value_type        = std::pair<val_t, val_t>;
    using difference_type   = long int;
    using pointer           = const val_t*;
    using reference         = const val_t&;
```

- The constructor sets up all the private iterator variables:

```
zip_iterator(T& a, T& b) :
    ita {a.begin()},
    itb {b.begin()},
    ita_begin {ita},
    itb_begin {itb},
    ita_end {a.end()},
    itb_end {b.end()}
{}
```

- We define the operator overloads necessary for a forward iterator:

```
zip_iterator& operator++() {
    ++ita;
    ++itb;
    return *this;
}

bool operator==(const zip_iterator& o) const {
    return ita == o.ita || itb == o.itb;
}

bool operator!=(const zip_iterator& o) const {
    return !operator==(o);
}

ret_t operator*() const {
    return {*ita, *itb};
}
```

- And finally, the `begin()` and `end()` functions return the respective iterators:

```
zip_iterator begin() const {
    return zip_iterator(ita_begin, itb_begin);
}
```

```
zip_iterator end() const {
    return zip_iterator(ita_end, itb_end);
}
```

These are made simple by the stored iterators and the private constructor.

- Now let's expand our `main()` function for testing:

```
int main() {
    vector<std::string> vec_a {"Bob", "John", "Joni"};
    vector<std::string> vec_b
        {"Dylan", "Williams", "Mitchell"};

    print("vec_a: ");
    for(const auto& e : vec_a) print("{} ", e);
    print("\n");

    print("vec_b: ");
    for(const auto& e : vec_b) print("{} ", e);
    print("\n");

    print("zipped: ");
    for(const auto& [a, b] : zip_iterator(vec_a, vec_b)) {
        print("[{}, {}] ", a, b);
    }
    print("\n");
}
```

- This gives us the output we're looking for:

```
vec_a: Bob John Joni
vec_b: Dylan Williams Mitchell
zipped: [Bob, Dylan] [John, Williams] [Joni, Mitchell]
```

## How it works

The *zipped iterator adaptor* is an example of how flexible the iterator abstraction can be. We take the iterators of two containers and use them in one aggregated iterator. Let's see how this works.

The main constructor for the `zip_iterator` class takes two container objects. For the purposes of this discussion, we'll refer to these objects as the *target* objects.

```
zip_iterator(T& a, T& b) :
    ita {a.begin()},
    itb {b.begin()},
    ita_begin {ita},
    itb_begin {itb},
    ita_end {a.end()},
    itb_end {b.end()}
{}
```

The constructor initializes the `ita` and `itb` variables from the target `begin()` iterators. These will be used to navigate the target objects. The target `begin()` and `end()` iterators are also saved for later use.

These variables are defined in the private section:

```
it_t ita {};
it_t itb {};

// for begin() and end() objects
it_t ita_begin {};
it_t itb_begin {};
it_t ita_end {};
it_t itb_end {};
```

The `it_t` type is defined as the type of the target iterator class:

```
using val_t = typename T::value_type;
using ret_t = std::pair<val_t, val_t>;
using it_t = typename T::iterator;
```

The other aliased types are `val_t` for the type of the target value and `ret_t` for the return `pair`. These type definitions are used for convenience throughout the class.

The `begin()` and `end()` functions use a private constructor that only initializes the `ita` and `itb` values:

```
zip_iterator begin() const
    { return zip_iterator(ita_begin, itb_begin); }
```

```
zip_iterator end() const
    { return zip_iterator(ita_end, itb_end); }
```

The `private` constructor looks like this:

```
zip_iterator(it_t ita, it_t itb) : ita {ita}, itb {itb} {}
```

This is a constructor that takes `it_t` iterators for parameters. It only initializes `ita` and `itb` so they can be used in the comparison operator overloads.

The rest of the class just acts like a normal iterator, but it operates on iterators from the target class:

```
zip_iterator& operator++() {
    ++ita;
    ++itb;
    return *this;
}
bool operator==(const zip_iterator& o) const {
    return ita == o.ita || itb == o.itb;
}
bool operator!=(const zip_iterator& o) const {
    return !operator==(o);
}
```

The dereference operator returns a `std::pair` object (`ret_t` is an alias for `std::pair<val_t, val_t>`). This is the interface for retrieving a value from the iterator.

```
ret_t operator*() const {
    return {*ita, *itb};
}
```

## There's more...

The `zip_iterator` adaptor can be used to easily zip objects into a `map`:

```
map<string, string> name_map{};

for(const auto& [a, b] : zip_iterator(vec_a, vec_b)) {
    name_map.try_emplace(a, b);
}
```

```
print("name_map: ");
for(const auto& [a, b] : name_map) {
    print("[{}, {}] ", a, b);
}
print("\n");
```

When we add this code to `main()`, we get this output:

```
name_map: [Bob, Dylan] [John, Williams] [Joni, Mitchell]
```

# Create a random-access iterator

This recipe is an example of a full-featured contiguous/random-access iterator. This is the most complete type of iterator for a container. A random-access iterator includes all the features of the other types of container iterators, along with its random-access capabilities.

While I felt it was important to include a complete iterator in this chapter, with over 200 lines of code this example is somewhat larger than the other examples in this book. I'll cover the essential components of the code here.

You may download the full source code at `https://github.com/PacktPublishing/CPP-STL-Cookbook-Second-Edition/blob/main/Chap04/container-iterator.cpp`.

## How to do it

To start, we need a container for our iterator. We'll use a simple array for this, and we'll call our class, `Container`. The `iterator` class is nested within the `Container` class.

All of this is designed to be consistent with the STL container interfaces.

- `Container` is defined as a `template` class. Its `private` section has only two elements:

  ```
  template<typename T>
  class Container {
      std::unique_ptr<T[]> c_ptr {};
      size_t n_elements{};
  ```

  We use a `unique_pointer` for the data. We let the *smart pointer* manage its own memory. This mitigates the need for a `~Container()` destructor. The `n_elements` variable keeps the size of our container.

  For a discussion of *smart poniters*, see *Chapter 8*, *Utility Classes*.

- In the public section we have two constructors. The first constructor allocates space without populating the elements:

```
explicit Container(size_t sz) : n_elements{sz} {
    c_ptr = std::make_unique<T[]>(n_elements);
}
```

  The `make_unique()` call constructs empty objects for element.

- The second constructor populates the container. This is a *delegating constructor* that calls the first constructor to allocate space for the container.

```
Container(initializer_list<T> i_lst) :
      Container(i_lst.size()) {
    std::copy(i_lst.begin(), i_lst.end(), c_ptr.get());
}
```

- The `size()` method returns the number of elements:

```
size_t size() const {
    return n_elements;
}
```

- The `operator[]()` overload returns an indexed element *without bounds checking*:

```
const T& operator[](const size_t index) const {
    return c_ptr[index];
}
```

- The `at()` method returns an indexed element *with bounds checking*:

```
T& at(const size_t index) const {
    if (index > n_elements - 1) {
        throw std::out_of_range(
            "Container::at(): index out of range"
        );
    }
    return c_ptr[index];
}
```

This is consistent with STL usage. The `at()` function is the preferred method of addressing an indexed element.

- The `begin()` and `end()` methods call the iterator constructor with the address of the container data

```
iterator begin() const { return iterator(c_ptr.get()); }
iterator end() const {
    return iterator(c_ptr.get() + n_elements);
}
```

The `unique_ptr::get()` function returns the address from the smart pointer.

- The `iterator` class is nested within the `Container` class as a `public` member

```
class iterator {
    T* ptr;
```

The iterator class has one private member, a pointer that's initialized in the `begin()` and `end()` methods of the `Container` class.

- The iterator constructor takes a pointer to the container data

```
iterator(T* ptr = nullptr) : ptr{ptr} {}
```

We provide a default value because the standard requires a default constructor.

## Operator overloads

This iterator provides operator overloads for the following operators: ++, *postfix* ++, `--`, *postfix* `--`, [ ], *default comparison* <=> *(C++20)*, ==, *, ->, +, *non-member* +, *numeric* -, *object* -, +=, and -=. We'll cover a few notable overloads here. See the source code for all of them.

- The C++20 default comparison operator <=> provides the functionality of the full suite of comparison operators, except the equality == operator:

```
// default comparison operator (C++20)
const auto operator<=>(const iterator& o) const {
    return ptr <=> o.ptr;
}
```

This is a C++20 feature, so it requires a compliant compiler and library.

- There are two + operator overloads. These support *it* + *n* and *n* + *it* operations.

```
iterator operator+(const size_t n) const {
    return iterator(ptr + n);
}

// non-member operator (n + it)
friend const iterator operator+(const size_t n,
        const iterator& o) {
    return iterator(o.ptr + n);
}
```

  The `friend` declaration is a special case. When used in a template class member function, it's the equivalent of a non-member function. This allows a non-member function to be defined in the class context.

- The - operator also has two overloads. We need to support both a numeric operand and an iterator operand.

```
iterator operator-(const size_t n) {
    return iterator(ptr - n);
}
size_t operator-(const iterator& o) {
    return ptr - o.ptr;
}
```

This allows both *it* – *n* and *it* – *it* operations. There's no need for a non-member function, as *n* – *it* is not a valid operation.

## Validation code

The C++23 specification requires a specific set of operations and results for a valid random-access iterator. I've included a `unit_tests()` function in the source code to validate those requirements.

The `main()` function creates a `Container` object and performs some simple validation functions.

- First, we create a `Container<string>` object `x` with ten values

```
Container<string> x{"one", "two", "three", "four", "five",
    "six", "seven", "eight", "nine", "ten" };
println("Container x size: {}", x.size());
```

The output gives the number of elements:

```
Container x size: 10
```

- We display the elements of the container with a range-based **for** loop:

```
println("Container x:");
for (auto e : x) {
    print("{} ", e);
}
print("\n");
```

Output:

```
Container x:
one two three four five six seven eight nine ten
```

- Next, we test several direct access methods:

```
println("direct access elements:");
println("element at(5): {}", x.at(5));
println("element [5]: {}", x[5]);
println("element begin + 5: {}", *(x.begin() + 5));
println("element 5 + begin: {}", *(5 + x.begin()));
println("element begin += 5: {}", *(x.begin() += 5));
```

Output:

```
direct access elements:
element at(5): six
element [5]: six
element begin + 5: six
```

```
element 5 + begin: six
element begin += 5: six
```

- We test the container with a `ranges::views` pipe and `views::reverse`:

```
println("views pipe reverse:");
auto result = x | views::reverse;
for (auto v : result) print("{} ", v);
print("\n");
```

Output:

```
views pipe reverse:
ten nine eight seven six five four three two one
```

- Next, we create a `Container` object with `10` uninitialized elements:

```
Container<string> y(x.size());
println("Container y size: {}\n", y.size());
for (auto e : y) {
    print("[{}] ", e);
}
print("\n");
```

Output:

```
Container y size: 10
[] [] [] [] [] [] [] [] [] []
```

- Finally, the `unit_tests()` function runs the tests from the standard and provides the following output:

```
unit tests ============================
*a: one
*b: six
n: 5
true: (a += n) == b
true: std::addressof(a += n) == std::addressof(a)
true: (a + n) == (a += n)
true: (a + n) == (n + a)
true: a + (i + j) == (a + i) + j
```

```
true: a + 0 == a
true: --b == (a + (n - 1))
true: (b += -n) == a && (b -= n) == a
true: std::addressof(b -= n) == std::addressof(b)
true: (b - n) == (b -= n)
true: a[n] == *b
true: (a <= b)
```

## How it works

Although it's a lot of code, this iterator is no more complicated than a smaller iterator. Most of the code is in the operator overloads, which are mostly just one or two lines of code each.

The data is managed by a *smart pointer*, which is simplified by the fact that it's a flat array and doesn't require expansion or compression. We'll cover smart pointers in more detail in *Chapter 8*, *Utility Classes*.

While the STL provides a flat `std::array` class, as well as other more complex data structures, you will find it valuable to demystify the workings of a complete iterator class.


# Get this book's PDF version and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.



UNLOCK NOW

*Note: Keep your invoice handy. Purchases made directly from Packt don't require an invoice.*

# 5

# Lambda Expressions

The C++11 standard introduced *lambda expressions* (sometimes called *lambda functions*, or simply *lambdas*). This feature allows an anonymous function to be used in the context of an expression. Lambdas may be used in function calls, containers, variables, and other expression contexts. It may sound innocuous, but it's remarkably useful.

In this chapter, we will cover the use of lambdas with the STL, in the following recipes:

- Lambdas and closures
- Use lambdas for scoped reusable code
- Use lambdas as predicates with algorithms
- Create a polymorphic wrapper with std::function
- Concatenate lambdas with recursion
- Combine predicates with a logical conjunction
- Pass common parameters to multiple lambdas
- Create a simple jump table with mapped lambdas

Let's start with a brief review of lambda expressions.

## Lambda expressions

A lambda is essentially an anonymous function as a literal expression:

```
auto la = []{ return "Hello"; };
```

The variable `la` may now be used as if it were a function:

```
println("{}", la());
```

It can be passed to another function:

```
f(la);
```

It can be passed to another lambda:

```
const auto la = []{ return "Hello"; };
const auto lb = [](auto a){ return a(); };
println("{}", lb(la));
```

Output:

```
Hello
```

Or it can be passed anonymously (as a literal):

```
const auto lb = [](auto a){ return a(); };
println("{}", lb([]{ return "Hello"; }));
```

## Closures

The term *closure* is often applied to any anonymous function. Strictly speaking, a closure is a function that allows the use of symbols outside its own lexical scope.

You may have noticed the square brackets in the definition of a lambda:

```
auto la = []{ return "Hello"; };
```

The square brackets are used to specify a list of *captures*. Captures are outside variables that are accessible from within the scope of the lambda body. If I try to use an outside variable without listing it as a capture, I'll get a compilation error:

```
const char * greeting{ "Hello\n" };
const auto la = []{ return greeting; };
println("{}", la());
```

When I try to compile this with GCC, I get the following error:

```
In lambda function:
error: 'greeting' is not captured
```

This is because the body of the lambda has its own lexical scope, and the `greeting` variable is outside of that scope.

I can specify the `greeting` variable in a *capture*. This allows the variable into the scope of the lambda:

```
const char * greeting{ "Hello\n" };
const auto la = [greeting]{ return greeting; };
println("{}", la());
```

Now it compiles and runs as expected:

```
$ ./working
Hello
```

This ability to capture variables outside its own scope is what makes a lambda a *closure*. People use the term in different ways, and that's fine, so long as we can understand each other. Still, it's good to know what the term means.

Lambda expressions allow us to write good, clean generic code. They allow the use of *functional programming* patterns, where we can use lambdas as functional parameters to algorithms and even other lambdas.

# Use lambdas for scoped, reusable code

Lambda expressions can be defined and stored for later use. They can be passed as parameters, stored in data structures, and called in different contexts with different parameters. They are as flexible as functions, but with the mobility of data.

## How to do it

We'll start with a simple program to test various configurations of lambda expressions:

- First, we define a `main()` function that we can use to experiment with lambdas:

  ```
  int main() {
      ... // code goes here
  }
  ```

- Inside the `main()` function, we'll declare a couple of lambdas

The basic definition of a lambda requires a pair of square brackets and a block of code in curly brackets:

```
auto one = [](){ return "one"; };
auto two = []{ return "two"; };
```

Notice that the first example one includes parentheses after the square brackets, and the second example `two` does not. The empty parameter parentheses are often included but are not required. The return type is inferred by the compiler.

- I can call these functions in any context that will take a C-string:

```
println("{}", one());
println("{}", two());
```

- In many cases, the compiler can determine the return type from *automatic type deduction*. Otherwise, you can specify the return type with the **->** operator:

```
auto one = [] -> const char * { return "one"; };
auto two = [] -> auto { return "two"; };
```

Lambdas use *trailing return type* syntax. This consists of the **->** operator followed by the *type specification*. If the return type is not specified, it is considered `auto`.

Let's define a lambda to print out the values from our other lambdas:

```
auto p = [](auto v){ println("{}", v()); };
```

The `p()` lambda expects a lambda (or function) as its parameter `v` and calls it in its function body.

The `auto` type parameter makes this lambda an *abbreviated template*. Before C++20, this was the only way to template a lambda. Beginning with C++20, you may specify template parameters after the capture brackets. This is the equivalent with a template parameter:

```
auto p = []<typename T>(T v){ println("{}", v()); };
```

The abbreviated `auto` version is simpler and more common. It works well for most purposes.

- Now we can pass an anonymous lambda to this lambda in the function call:

```
p([]{ return "lambda call lambda"; });
```

The output is:

```
  lambda call lambda
```

- If we need to pass parameters to an anonymous lambda, we can put them in parentheses after the lambda expression:

```
println("{}",
    [](auto l, auto r){ return l + r; }(47, 73));
```

The function parameters, 47 and 73, are passed to the anonymous lambda in parentheses after the function body.

- You can access variables from the outside scope of the lambda by including them as *captures* in the square brackets:

```
int num {1};
p([num]{ return num; });
```

- Or you can capture them by reference:

```
int num {};
autoinc=[&num]{ return ++num; };
for (size_t i {}; i < 5; ++i) {
    print("{} ", inc());
}
print("\n");
```

The output is as follows:

```
  1 2 3 4 5
```

*Capture by reference* allows you to modify a captured variable.

- You can also define a local capture variable that maintains its state:

```
auto counter = [n = 0]() mutable { return ++n; };
for (size_t i {}; i < 5; ++i) {
```

```
    print("{} ", counter());
}
print("\n");
```

Output:

```
  1, 2, 3, 4, 5
```

The `mutable` specifier allows the lambda to modify its captures. Lambda captures default to `const`-qualified.

As with the trailing return type, any *specifier* requires the parameter parentheses.

- The lambda supports two types of *default capture*:

```
int a = 47;
int b = 73;
auto l1 = []{ return a + b; };
```

If I try to compile this code, I get an error that includes:

```
  note: the lambda has no capture-default
```

One type of default capture is indicated by an equal sign:

```
auto l1 = [=]{ return a + b; };
```

This will capture all the symbols in the lambda's scope. The equal sign performs *capture by copy*. It will capture a copy of the objects as if they were copied with an assignment operator.

The other default capture uses an ampersand for *capture by reference*:

```
autol1=[&]{returna+b; };
```

This is a default capture that captures by reference.

The default captures only use symbols when they are referenced, so they're not as messy as they may look. That said, I recommend explicit captures where possible as they generally improve readability.

**Note**

Prior to C++23, whenever a *return type*, *qualifier*, *modifier*, or *exception attribute* was included, the parentheses were required even without parameters:

```
auto one = []() -> const char * { return "one"; };
```

As of C++23, empty parentheses are no longer required in any circumstances:

```
auto one = [] -> const char * { return "one"; };
```

## How it works

The syntax of a lambda expression is as follows:

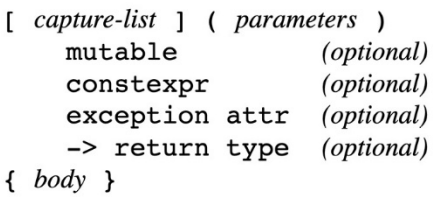



*Figure 5.1: Syntax of a lambda expression*

The only required parts of a lambda expression are the capture list and the body, which may be empty:

```
[]{}
```

This is the minimal lambda expression. It captures nothing and does nothing.

Let's consider each of the parts.

### Capture list

The *capture-list* specifies what we capture, if anything. It cannot be omitted, but it may be empty. We can use `[=]` to capture all variables *by copy* or `[&]` to capture all variables *by reference*, within the scope of the lambda.

You may capture individual variables by listing them in the brackets:

```
[a, b]{ return a + b; }
```

The specified captures default to *capture by copy*. You may capture by reference with the reference operator:

```
[&a, &b]{ return a + b; }
```

When you capture by reference, you may modify the referenced variable.

**Note**

You cannot capture object members directly. You may capture `this` or `*this` to dereference class members.

## Parameters

As with a function, parameters are specified in parentheses:

```
[](int a, int b){ return a + b };
```

If there are no parameters, the parentheses are not required. Prior to C++23, the parentheses were always required if a *return type*, *qualifier*, *modifier*, or *exception attribute*, was included:

```
[]() -> int { return 47 + 73 };
```

As of C++23, the parentheses are optional if empty.

## The mutable modifier (optional)

A lambda expression defaults to `const`-qualified unless you specify the `mutable` modifier. This allows the lambda to be used in `const` contexts. It also means that any captured-by-copy variables cannot be modified. For example:

```
[a]{ return ++a; };
```

This will fail to compile with an error message like this:

```
In lambda function:
error: increment of read-only variable 'a'
```

With the `mutable` modifier, the lambda is no longer `const`-qualified and the captured variable may be changed:

```
[a]() mutable { return ++a; };
```

## The constexpr specifier (optional)

You may use `constexpr` to explicitly specify that you want your lambda to be considered a `constant expression`. This means that it may be evaluated at compile time:

```
constexpr int z {42};
constexpr auto x = [z] constexpr { return z; };
```

If the lambda meets the requirements, it may be considered `constexpr` even without the specifier. In practice, the compiler should take care of this for you and you should never need to use the specifier.

## The noexcept attribute (optional)

You can use the `noexcept` specifier to declare that your lambda does not throw any exceptions:

```
auto x = [] noexcept { return 42; };
```

## The trailing return type (optional)

By default, the lambda return type is deduced from the `return` statement, as if the return type was `auto`. You may optionally specify a *trailing return type* with the `->` operator:

```
[](int a, int b) -> long { return a + b; };
```

In this example, the return type is `long`. It would have otherwise been deduced as `int` without the specified return type.

**Note**

***Compiler compatibility***

At the time of this writing, current versions of Microsoft Visual C++ have yet to implement the C++23 lambda syntax. Mostly, this means that parameter parentheses are required according to C++20 rules. Under current versions (both release and preview), this will not compile:

```
auto one = [] -> const char * { return "one"; };
```

It requires empty parentheses, as in C++20:

```
auto one = []() -> const char * { return "one"; };
```

If you're using the Microsoft compiler, please be aware of this limitation.

# Use lambdas as predicates with algorithms

Some functions in the `algorithm` library require the use of a *predicate function*. A predicate is a function that tests a condition and returns a Boolean `true`/`false` response.

For example, the `count_if()` function from the `algorithm` library returns the number of elements in a sequence which match a condition. The `count_if()` function has this signature:

```
size_t count_if(InputIt first, InputIt last, UnaryPred p);
```

The condition is specified by the predicate function `p`. The predicate may be any callable object, including a function pointer, functor (a function object), or a lambda. In modern C++, it's most common to use a lambda for this. A predicate function may be *unary* or *binary*. A unary predicate takes one argument and a binary predicate takes two arguments. In this recipe, we'll consider both unary and binary predicates using lambdas.

## How to do it

For this recipe, we will experiment with the `count_if()` and `sort()` algorithms using different types of predicates:

- First, let's create a function for use as a predicate with `count_if()`. A predicate takes a certain number of arguments and returns a `bool`. A predicate for `count_if()` takes one argument:

  ```
  bool is_div4(const int i) {
      return i % 4 == 0;
  }
  ```

  This predicate checks an `int` to see if it's divisible by 4.

- In `main()`, we'll define a vector of `int` values, and use it to test our predicate function with `count_if()`:

  ```
  int main() {
      vector<int> v {1, 7, 12, 20, 4, 9, 4, 8};
      auto count4 = count_if(v.begin(), v.end(), is_div4);
      println("numbers divisible by 4 (is_div4): {}",
        count4);
  }
  ```

The output is as follows:

```
  numbers divisible by 4 (is_div4): 5
```

(The 5 divisible numbers are: 4, 4, 8, 12, and 20.)

The `count_if()` algorithm uses the predicate function to determine which elements of the sequence to count. It calls the predicate with each element as a parameter, and only counts the element if the predicate returns `true`.

In this case, we used a function as a predicate.

- We could also use a *functor* as a predicate:

```
struct is_div4 {
    bool operator()(const int i) const {
        return i % 4 == 0;
    }
};
```

The only change here is that we need to use an *instance* of the class as the predicate:

```
auto count4 = count_if(v.begin(), v.end(), is_div4());
```

The advantages of a functor are that it can carry context, and it can access class and instance variables. This was the common way to use predicates before C++11 introduced lambda expressions.

- With a lambda expression, we have the best of both worlds: the simplicity of a function and the power of a functor. We can use a lambda as a variable:

```
auto is_div4 = [](int i){ return i % 4 == 0; };
auto count4 = count_if(v.begin(), v.end(), is_div4);
```

Or we can use an anonymous lambda:

```
auto count = count_if(v.begin(), v.end(),
    [](const int i) { return i % 4 == 0; });
```

- We can take advantage of the lambda capture by wrapping the lambda in a function and using that function context to produce the same lambda with different parameters:

```
auto is_div_by(const int divisor) {
    return [divisor] (const int i)
        { return i % divisor == 0; };
}
```

This function returns a predicate lambda with the divisor from the capture context.

We can then use that predicate with `count_if()`:

```
for (int i : { 3, 4, 5 }) {
    auto pred = is_div_by(i);
    auto count = count_if(v.begin(), v.end(), pred);
    println("numbers divisible by {}: {} (is_div_by)", i,
      count);
}
```

Each call to `is_div_by()` returns a predicate with a different divisor from `i`. Now we get this output:

```
  numbers divisible by 3 (is_div_by): 2
  numbers divisible by 4 (is_div_by): 5
  numbers divisible by 5 (is_div_by): 2
```

- A *binary predicate* is simply a predicate that takes two arguments. For example, the `ranges::sort()` algorithm may use an optional predicate as a custom comparator:

```
auto comp = [](int a, int b) { return a < b; };
println("unsorted: {}", v);
ranges::sort(v, comp);
println("sorted: {}", v);
```

Here we use a lambda for a custom comparator predicate that sorts our vector in descending order. The arguments are passed to the predicate by the `sort()` function which uses the predicate to compare values in its sorting algorithm.

This gives us the output:

```
unsorted: [1, 7, 12, 20, 4, 9, 4, 8]
sorted: [20, 12, 9, 8, 7, 4, 4, 1]
```

## How it works

The type of a function pointer is represented as a pointer followed by the function call `()` operator:

```
void (*)()
```

You can declare a function pointer and initialize it with the name of an existing function:

```
void (*fp)() = func;
```

This declaration creates a variable named `fp` with the type of `void(*)()`, a pointer to a function that returns `void`. Once declared, a function pointer may be dereferenced and used as if it were the function itself:

```
func();  // do the func thing
```

A lambda expression has the same type as a function pointer:

```
void (*fp)() = []{ return 42; };
```

This means that wherever you use a function pointer with a certain signature, you may also use a lambda with the same signature. This allows function pointers, functors, and lambdas to work interchangeably:

```
bool (*fp)(int) = is_div4;
bool (*fp)(int) = [](int i){ return i % 4 == 0; };
```

Because of this interchangeability, an algorithm such as `count_if()or sort()` may accept a function, functor, or lambda where it expects a predicate with a particular function signature.

This applies to any algorithm that uses a predicate.

# Create a polymorphic wrapper with std::function

The `std::function` class is a thin polymorphic wrapper for functions. It can store, copy, and invoke references to any function, lambda expression, or other functor. It's polymorphic in that it allows you to store functions and lambdas with different signatures in the same container, while maintaining the context of lambda captures. It's useful in places where you would like to store a reference to a function or lambda.

## How to do it

This recipe uses the `std::function` class to store different specializations of a lambda in a `vector`:

- This recipe is contained in the `main()` function, where we start by declaring three containers of different types:

```
int main() {
    deque<int> d;
    list<int> l;
    vector<int> v;
```

  These containers, `deque`, `list`, and `vector`, will be referenced by a *template* lambda.

- We declare a simple `print_c` lambda function for printing out the containers:

```
auto print_c = [] (const auto& c) {
    for (const auto i : c) print("{} ", i);
    print("\n");
};
```

  The `print_c` lambda function takes a container parameter and uses a `for` loop to print its elements.

- Now we declare another lambda that returns an *anonymous* lambda:

```
auto push_c = [] (auto& container) {
    return [&container] (const auto value) {
        container.push_back(value);
    };
};
```

This is a lambda that returns another lambda. The `push_c` lambda takes a reference to a container, which is then *captured* by the anonymous lambda. The anonymous lambda calls the `push_back()` member on the captured container. The return value from `push_c` is the anonymous lambda.

- Now we declare a `vector` of `std::function` elements, and populate it with three instances of `push_c()`:

```
const vector<function<void(int)>>
    consumers {push_c(d), push_c(l), push_c(v)};
```

The `vector` object is named `consumers`. Its template parameter specifies a signature using the `function` class. The template parameter on the `function` class is a generic function that takes one parameter. The type of the parameter in the `function` signature uses `int` as a generic placeholder.

Each of the elements in the initializer list is a function call to the `push_c` lambda. `push_c` returns an instance of the anonymous lambda, which gets stored in the `vector` via the `function` wrapper. The `push_c` lambda is called with the three containers, `d`, `l`, and `v`, a `deque`, a `list`, and a `vector` respectively. The containers are passed as *captures* with the anonymous lambda.

- Now we loop through the `consumers` vector and call each of the lambda elements `10` times, populating the three containers with integers `0`–`9` in each container:

```
for (auto &c : consumers) {
    for (int i{0}; i < 10; ++i) {
        c(i);
    }
}
```

- Now our three containers, the `deque`, `list`, and `vector`, should all be populated with integers. Let's print them out:

```
print_c(d);
print_c(l);
print_c(v);
```

Our output should be:

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

## How it works

Lambdas are often used with indirection, and this recipe is a good example of that. For example, the `push_c` lambda returns an anonymous lambda:

```
auto push_c = [] (auto& container) {
    return [&container] (const auto value) {
        container.push_back(value);
    };
};
```

This anonymous lambda gets stored in the `vector` of `function` objects:

```
const vector<function<void(int)>>
    consumers {push_c(d), push_c(l), push_c(v)};
```

This defines the `consumers` container. It is initialized with three elements, where each element is the lambda returned by a call to `push_c`. The anonymous lambda gets stored in the vector, not the `push_c` lambda.

The `vector` definition uses the `function` class as the type of the elements. The `function` constructor takes any callable object and stores its reference as the `function` target:

```
template< class F >
function(F&&f);
```

When its function call `()` operator is invoked, the `function` object calls the target function with the intended parameters:

```
for (auto& c : consumers) {
    for (int i{0}; i < 10; ++i) {
        c(i);
    }
}
```

This calls each of the anonymous lambdas, as stored in the `consumers` container, 10 times, thus populating the `d`, `l`, and `v` containers.

## There's more...

The nature of the `std::function` class makes it useful for many purposes. You can think of it as a polymorphic function container. It can store a standalone function:

```
void hello() {
    println("hello");
}

int main() {
    const function<void(void)> h = hello;
    h();
}
```

It can store a member function, using `std::bind` to bind function parameters:

```
struct hello {
    void greeting() const { println("Hello Human"); }
};
int main() {
    hello human{};
    const function<void(void)> h =
        std::bind(&hello::greeting,&human);
    h();
}
```

Or it can store any executable object:

```
struct hello {
    void operator()() const { println("Hello Human"); }
};
int main() {
    const function<void(void)> h = hello();
    h();
}
```

The output is as follows:

```
Hello Human
```

# Concatenate lambdas with recursion

You can stack lambdas so that the output of one is the input of the next, using a simple recursive function. This creates a simple way to build one function upon another, allowing the lambdas to serve as modular building blocks.

## How to do it

This is a short and simple recipe that uses one recursive function to do most of the work:

- We'll start by defining the concatenation function `concat()`:

```
template <typename T, typename ...Ts>
        auto concat(const T t, const Ts ...ts) {
    if constexpr (sizeof...(ts) > 0) {
        return [=](const auto ...parameters) {
            return t(concat(ts...)(parameters...));
        };
    } else  {
        return t;
    }
}
```

  This function returns an anonymous lambda, which in turn calls the function again, until the parameter pack is exhausted.

- In the `main()` function, we create a couple of lambdas and call the `concat()` function with them:

```
int main() {
    auto twice = [](const auto i) { return i * 2; };
    auto thrice = [](const auto i) { return i * 3; };

    auto combined = concat(thrice, twice,
        std::plus<int>{});
    println("{}", combined(2, 3));
}
```

The `concat()` function is called with three parameters: two lambdas, and the `std::plus()` function.

As the recursion unravels, the functions are called right-to-left, starting with `plus()`. The `plus()` function takes two arguments and returns the sum. The return value from `plus()` is passed to `twice()`, and its return value is passed to `thrice()`. The result is then printed to the console with `println()`.

Output:

```
30
```

## How it works

Let's take a closer look at the `concat()` function. It is simple, but it may be confusing due to the *recursion* and the *indirection* of the returned lambda:

```
template <typename T, typename ...Ts>
auto concat(const T t, const Ts ...ts) {
    if constexpr (sizeof...(ts) > 0) {
        return [=](const auto ...parameters) {
            return t(concat(ts...)(parameters...));
        };
    } else  {
        return t;
    }
}
```

The `concat()` function is called with a *template parameter pack*. The `sizeof...` operator (with ellipsis) returns the number of elements in the parameter pack. This is used to test for the end of the recursion. Because the `sizeof...` operator returns a constant that must be evaluated at compile time, the `if()` expression must use the `constexpr` modifier.

The `concat()` function returns a lambda. The lambda recursively calls the `concat()` function in the `return` expression. Because the first argument of `concat()` is not part of the parameter pack, each recursive call peels off the first element of the pack.

The outer `return` statement returns the lambda. The inner `return` calls the function that was passed to `concat()` and returns its value.

Feel free to take this apart and study it. There's value in this technique.

# Combine predicates with logical conjunction

This example wraps a lambda in a function to create a custom conjunction for use with an algorithm predicate.

## How to do it

The `copy_if()` algorithm requires a predicate that takes one parameter. In this recipe, we create a predicate lambda from three other lambdas:

- First, we'll write the `combine()` function. This function returns a lambda for use with the `copy_if()` algorithm:

```
template <typename F, typename A, typename B>
auto combine(F binary_func, A a, B b) {
    return [=](auto param) {
        return binary_func(a(param), b(param));
    };
}
```

  The `combine()` function takes three function parameters – a binary conjunction and two predicates – and returns a lambda that calls the conjunction with the two predicates.

- In the `main()` function, we create the lambdas for use with `combine()`:

```
auto begins_with = [](const string &s)
    { return s.find("a") == 0; };
auto ends_with = [](const string &s)
    { return s.rfind("b") == s.length() - 1; };
auto bool_and = [](const auto& l, const auto& r)
    { returnl&&r };
```

  The `begins_with` and `ends_with` lambdas are simple filter predicates to find strings that begin with `'a'` and end with `'b'`, respectively. The `bool_and` lambda is the conjunction.

- Now we can call the `copy_if` algorithm with `combine()`:

```
std::copy_if(istream_iterator<string>{cin}, {},
             ostream_iterator<string>{cout, " "},
             combine(bool_and, begins_with, ends_with));
```

```
print("\n");
```

The `combine()` function returns a lambda that combines the two predicates with the conjunction.

The output looks like the following:

```
$ echo aabb bbaa foo bar abazb | ./conjunction
aabb abazb
```

## How it works

The `std::copy_if()` algorithm requires a predicate function that takes one parameter, but our conjunction requires two parameters, each of which requires one parameter. We resolve this with a function that returns a lambda specifically for this context:

```
template <typename F, typename A, typename B>
auto combine(F binary_func, A a, B b) {
    return [=](auto param) {
        return binary_func(a(param), b(param));
    };
}
```

The `combine()` function creates a lambda from three parameters, each of which is a function. The returned lambda takes the one parameter that's required of `copy_if()`'s predicate function. Now we can call `copy_if()` with the `combine()` function:

```
std::copy_if(istream_iterator<string>{cin}, {},
             ostream_iterator<string>{cout, " "},
             combine(bool_and, begins_with, ends_with));
```

This passes the combined lambda to the algorithm so it can operate within that context.

# Pass common parameters to multiple lambdas

You can easily create multiple instances of a lambda with different capture values by wrapping the lambda in a function. This allows you to call different versions of a lambda with the same input.

## How to do it

This is a simple example of a lambda that wraps a value in different types of braces:

- We'll start by creating the wrapper function `braces()`:

```
auto braces (const char a, const char b) {
    return [a, b](const auto v) {
        print("{}{}{} ", a, v, b);
    };
}
```

  The `braces()` function wraps a lambda that returns a three-value string, where the first and last values are characters passed to the lambda as captures, and the middle value is passed as a parameter.

- In the `main()` function, we use `braces()` to create four lambdas, using four different sets of braces:

```
auto a = braces('(', ')');
auto b = braces('[', ']');
auto c = braces('{', '}');
auto d = braces('|', '|');
```

- Now we can call our lambdas from a simple `for()` loop:

```
for (auto i : {1, 2, 3, 4, 5}) {
    for (auto x : {a, b, c, d}) x(i);
    print("\n");
}
```

  In this pair of nested `for` loops, the outer loop counts from 1 to 5, passing an integer to the inner loop. The inner loop calls the lambdas with the different sets of braces.

  Both loops use an `initializer_list` as the container in a range-based `for` loop. This is a convenient technique for looping through a small set of values.

- The output from our program looks like this:

```
(1) [1] {1} |1|
(2) [2] {2} |2|
(3) [3] {3} |3|
```

```
(4) [4] {4} |4|
(5) [5] {5} |5|
```

The output shows each of the integers, in each combination of braces.

## How it works

This is a simple example of how to use a wrapper for a lambda. The `braces()` function constructs a lambda using the braces passed to it:

```
auto braces (const char a, const char b) {
    return [a, b](const auto v) {
        print("{}{}{} ", a, v, b);
    };
}
```

By passing the `braces()` function parameters to the lambda, it can return a lambda with that context. So, each of the assignments in the main function carries those parameters with it:

```
auto a = braces('(', ')');
auto b = braces('[', ']');
auto c = braces('{', '}');
auto d = braces('|', '|');
```

When these lambdas are called with a digit, they will return a string with that digit in the corresponding braces.

# Create a simple jump table with mapped lambdas

A jump table is a useful pattern when you want to select an action from a user or other input. Jump tables are useful as state machines and are often implemented in `if/else` or `switch` structures. In this recipe, we'll build a concise jump table using only an STL `map` and anonymous lambdas.

## How to do it

It's easy to build a simple jump table from a `map` and lambdas. The `map` provides simple indexed navigation and the lambda can be stored as payload. Here's how to do it:

- First, we'll create a simple `prompt()` function to get input from the console:

```
const char prompt(const char * p) {
    constexpr size_t BUFLEN {4};
```

```
    constexpr int NL {'\n'};
    char bufin[BUFLEN] {};
    int bufchar {};

    // fold char to uppercase
    auto char_upper = [](char c) ->char {
        if (c >= 'a'&&c<= 'z') return c - ('a' - 'A');
        else return c;
    };

    // flush the input buffer
    auto flush_stdin = []{
        int c {};
        while(c!=NL&&c!=EOF)c=getchar();
    };

    std::print("{} > ", p);
    for (size_t i {};
          i <BUFLEN&&(bufchar=getchar())!=NL;
          ++i) {
        bufin[i] = (char) bufchar;
    }

    if (bufchar != NL) flush_stdin();
    if (bufchar == EOF) exit(0);

    const char r0 = bufin[0];
    const char r1 = bufin[1];

    if (r0 == 0 || r0 == NL) return 0;
    else if (r1 != 0 && r1 != NL) {
        println("Response too long");
        return 0;
    }
    else return char_upper(r0);
}
```

The C-string parameter is used as a prompt. `std::getchar()` is called to get input from the user. The response is stored in `bufin`, checked for length, then if it's one character in length, it's converted to uppercase and returned.

We use a lambda for the uppercase conversion, and another for flushing the input buffer. Using lambdas for this allows us to keep these functions in the local scope of the `prompt()` function.

- We create a `jump()` function for the jump table. Here, we declare and initialize a `map` of lambdas:

```
const bool jump(const char select) {
    using jumpfunc = void(*)();

    static const std::map<char, jumpfunc> jumpmap {
        { 'A', []{ println("func A"); } },
        { 'B', []{ println("func B"); } },
        { 'C', []{ println("func C"); } },
        { 'D', []{ println("func D"); } },
    };

    const auto it = jumpmap.find(select);
    if (it != jumpmap.end()) it->second();
    else {
        println("Invalid response");
        return false;
    }
    return true;
}
```

The `map` container is loaded with anonymous lambdas for the jump table. These lambdas could easily call other functions or perform simple tasks. We store our map in static storage so it's only allocated once, while keeping it within the limited scope of the `jump()` function.

The `using` alias is for convenience. We're using the function pointer type `void(*)()` for the lambda payload. If you prefer, you could use `std::function()` if you need more flexibility or if you just find it more readable. It has very little overhead:

```
using jumpfunc = std::function<void()>;
```

To select a target, we call `find()` on the `map` object, then call the lambda with `it->second()`.

- In the `main()` function, we prompt for user input and select an action from the `map`:

```
int main() {
    const char* pstr {"What to do? (A/B/C/D/X)"};
    for (auto key = prompt(pstr);
          key != 'X';
          key = prompt(pstr)) {
        if (key) jump(key);
    }
    println("Bye!");
}
```

This is how we use the `map`-based jump table. We loop until `'X'` is selected for exit. We call `prompt()` with a prompt string and `jump()` with the keypress.

## How it works

The `map` container makes an excellent jump table. It's concise and easy to navigate:

```
using jumpfunc = void(*)();
map<const char, jumpfunc> jumpmap {
    { 'A', []{ cout << "func A\n"; } },
    { 'B', []{ cout << "func B\n"; } },
    { 'C', []{ cout << "func C\n"; } },
    { 'D', []{ cout << "func D\n"; } },
    { 'X', []{ cout << "Bye!\n"; } }
};
```

Anonymous lambdas are stored as payload in the `map` container. The keys are the character responses from the menu of actions.

You can test the validity of a key and select a lambda in one action:

```
const auto it = jumpmap.find(select);
if (it != jumpmap.end()) it->second();
else {
    println("Invalid response");
    return false;
```

```
    }
    return true;
```

This is a simple, elegant solution, where we would have otherwise used awkward branching code.

## Subscribe to Deep Engineering

Join thousands of developers and architects who want to understand how software is changing, deepen their expertise, and build systems that last.

Deep Engineering is a weekly expert-led newsletter for experienced practitioners, featuring original analysis, technical interviews, and curated insights on architecture, system design, and modern programming practice.

Scan the QR or visit the link to subscribe for free.



`https://packt.link/deep-engineering-newsletter`

# 6

# STL Algorithms

Much of the power of the STL is in the standardization of container interfaces. If a container has a particular capability, there's a good chance that the interface for that capability is standardized across container types. This standardization makes it possible for a library of *algorithms* to operate seamlessly across containers and sequences sharing a common interface.

For example, if we want to sum all the elements in a `vector` of `int`, we could use a loop:

```
vector<int> x {1, 2, 3, 4, 5};
long sum {};
for (int i : x) sum += i;
```

Or we could use an algorithm:

```
vector<int> x {1, 2, 3, 4, 5};
auto sum = accumulate(x.begin(), x.end(), 0);
```

This same syntax works with other containers:

```
deque<int> x {1, 2, 3, 4, 5};
auto sum = accumulate(x.begin(), x.end(), 0);
```

The algorithm version is easier to read and easier to maintain, and an algorithm is often more efficient than the equivalent loop.

Beginning with C++20, the `ranges` library provides a set of alternative algorithms that operate with *ranges* and *views*. This book will demonstrate those alternatives where appropriate.

Most of the algorithms are in the `algorithm` header. Some numeric algorithms, notably `accumulate()`, are in the `numeric` header, and some memory-related algorithms are in the `memory` header.

In this chapter, we will cover STL algorithms in the following recipes:

- Copy from one iterator to another
- Join container elements into a string
- Sort containers with `std::sort`
- Modify containers with `std::transform`
- Search containers with `std::find`
- Limit container values with `std::clamp`
- Sample data sets with `std::sample`
- Use ranges for more powerful algorithms
- Generate permutations of data sequences
- Merge sorted containers

# Copy from one iterator to another

The *copy algorithms* are generally used to copy from and to containers, but in fact, they work with iterators, which is far more flexible.

## How to do it

In this recipe, we will experiment with `std::copy` and `std::copy_n` to get a good understanding of how they work:

- Let's start with a function to print a container:

```
void printc(auto& c, string_view s = "") {
    if (s.size()) print("{}: ", s);
    for (auto e : c) print("[{}] ", e);
    print("\n");
}
```

- In `main()`, we define a `vector` and print it with `printc()`:

```
int main() {
    vector<string> v1 {"alpha", "bravo", "charlie",
        "delta", "echo"};
```

```
    printc(v1, "v1");
}
```

We get this output:

```
v1: [alpha] [bravo] [charlie] [delta] [echo]
```

- Now, let's create a second `vector` with enough space to make a copy of the first `vector`:

```
vector<string> v2(v1.size());
```

This creates an empty `vector` with enough room for all the elements from `v1`.

- We copy `v1` to `v2` using the `std::copy()` algorithm:

```
std::copy(v1.begin(), v1.end(), v2.begin());
printc(v2);
```

The `std::copy()` algorithm takes two iterators for the range of the copy source, and one iterator for the destination. In this case, we give it the `begin()` and `end()` iterators of `v1` to copy the entire `vector`. The `begin()` iterator of `v2` serves as the destination for the copy.

Our output is now:

```
v1: [alpha] [bravo] [charlie] [delta] [echo]
v2: [alpha] [bravo] [charlie] [delta] [echo]
```

- The `copy()` algorithm does not allocate space for the destination. So, `v2` must already have the space for the copy. Alternatively, you can use the `back_inserter()` iterator adapter to insert the elements at the back of the `vector`:

```
vector<string> v2 {};
std::copy(v1.begin(), v1.end(), back_inserter(v2));
```

- We can also use the `ranges::copy()` algorithm to copy an entire *range*. A container object serves as a range so we can use `v1` as the source. We still use an iterator for the destination:

```
vector<string> v2 {};
ranges::copy(v1, v2.begin());
```

This also works with `back_inserter()`:

```
vector<string> v2 {};
ranges::copy(v1, back_inserter(v2));
```

- You can copy a certain number of elements using `copy_n()`:

```
vector<string> v3 {};
std::copy_n(v1.begin(), 3, back_inserter(v3));
printc(v3, "v3");
```

In the second argument, the `copy_n()` algorithm is a *count* for the number of elements to copy. The output is:

```
  v3: [alpha] [bravo] [charlie]
```

- There's also a `copy_if()` algorithm that uses a Boolean *predicate function* to determine which elements to copy:

```
vector<string> v4 {};
std::copy_if(v1.begin(), v1.end(), back_inserter(v4),
    [](string& s){ return s.size() > 4; });
printc(v4, "v4");
```

There's also a `ranges` version of `copy_if()`:

```
vector<string> v4 {};
ranges::copy_if(v1, back_inserter(v4),
    [](string& s){ return s.size() > 4; });
printc(v4, "v4");
```

The output includes only strings longer than 4 characters:

```
v4: [alpha] [bravo] [charlie] [delta]
```

Notice that the value `echo`, which is not longer than 4 characters, is excluded.

- You can use any of these algorithms to copy to or from any sequence, including a stream iterator:

```
ostream_iterator<string> out_it(cout, " ");
ranges::copy(v1, out_it)
print("\n");
```

Output:

```
alpha bravo charlie delta echo
```

## How it works

The `std::copy()` algorithm is very simple. An equivalent function would look like this:

```
template<typename I_it, typename O_it>
O_it copy(I_it begin_it, I_it end_it, O_it dest_it) {
    while (begin_it != end_it) {
        *dest_it++ = *begin_it++;
    }
    return dest_it;
}
```

The `copy()` function uses the destination iterator's assignment operator to copy from the input iterator to the output iterator until it reaches the end of the input range.

There is also a version of this algorithm called `std::move()`, which moves elements instead of copying them:

```
std::move(v1.begin(), v1.end(), v2.begin());
printc(v1, "after move: v1");
printc(v2, "after move: v2");
```

This performs a move instead of copy assignment. After the move operation, the elements in `v1` will be empty, and the elements that were in `v1` are now in `v2`. The output looks like this:

```
after move1: v1: [] [] [] [] []
after move1: v2: [alpha] [beta] [gamma] [delta] [epsilon]
```

There is also a `ranges` version of the `move()` algorithm that performs the same operation:

```
ranges::move(v1, v2.begin());
```

The power of these algorithms lies in their simplicity. By letting the iterators manage the data, these simple, elegant functions allow you to seamlessly copy or move between any of the STL containers that support the required iterators.

# Join container elements into a string

Sometimes, there is no algorithm in the library to accomplish a task at hand. We can use iterators, with the same techniques as the `algorithms` library, to easily write one.

For example, we often need to join elements from a container, with separators, into a string. One common solution is to use a simple `for()` loop:

```
for(auto v : c) print(", ");
```

The problem with this otherwise simple solution is that it leaves a trailing separator:

```
vector<string> lads {"paul", "john", "george",
    "ringo", "billy"};
for (const auto v : lads) print("{}, ", v);
print("\n");
```

Output:

```
paul, john, george, ringo, billy,
```

This may be fine in a testing environment, but in any production system, that trailing comma is unacceptable.

The `ranges::views` library has a `join()` function, but it doesn't do what we want:

```
auto lads_view = lads | views::join;
```

This returns a `ranges::view` object, which requires a separate step to display or create a string. We can cycle through the view with a `for()` loop:

```
for (const auto v : lads_view) print("{}:", v);
print("\n");
```

The output looks like this:

```
p:a:u:l:j:o:h:n:g:e:o:r:g:e:r:i:n:g:o:b:i:l:l:y:
```

This is a completely flattened view, with each element of each string separated into its own `view` element. It may be useful in some contexts, but it's not what we're looking for, and it doesn't solve our problem at all.

Since the `algorithms` library does not have a function that suits our needs, we'll write one.

## How to do it

For this recipe, we will take the elements of a container and join them into a string with separators.

- In our `main()` function, we declare a `vector` of strings:

```
vector<string> lads{ "john", "paul", "george",
  "ringo", "billy" };
```

- Now, we can write a simple `join()` function that uses a `string` object to join elements with a separator:

```
namespace bw {
    template<typename I>
    string join(I it, I end_it, string_view sep = "") {
        string ostr {};
        if (it != end_it) ostr = format("{}", *it++);
        while (it != end_it) {
            ostr = format("{}{}{}", ostr, sep, *it++);
        }
        return ostr;
    }

    string join(const auto& c, string_view sep = "") {
        return join(c.begin(), c.end(), sep);
```

```
    }
}
```

I've put this in my own `bw` namespace to avoid name collisions.

Notice there's two specializations of the `join()` function. One takes iterators, and one takes a container. The container version calls the iterator version with the iterators from the container.

We use the `format()` function to format the elements of the container. This allows us to use the `format` library's specializations for various common types, or to create our own specializations to represent our data as strings.

We call it with `println()` like this:

```
println("{}", bw::join(lads, ", "));
```

Output:

```
john, paul, george, ringo, billy
```

**Note**

By using new C++23 features, such as `print()` and `format()`, and eliminating the use of the `iostream` library, this version of the `join()` function is about half the size and significantly more efficient than the one in the previous edition of this book.

## How it works

Most of the work in this recipe is done by the iterators:

```
template<typename I>
string join(I it, I end_it, const string_view sep = "") {
    string ostr {};
    if (it != end_it) ostr = format("{}", *it++);
    while (it != end_it) {
        ostr = format("{}{}{}", ostr, sep, *it++);
    }
    return ostr;
}
```

Separators are inserted after the first element, between each of the successive elements, and end before the final element. This means we can either add a separator *before* each element, skipping the first, or *after* each element, skipping the last. The logic is simpler if we process the first element separately. We do that in the line just before the `while` loop:

```
if (it != end_it) ostr = format("{}", *it++);
```

Once we have the first element out of the way, we can simply add a separator before each remaining element:

```
while (it != end_it) {
    ostr = format("{}{}{}", ostr, sep, *it++);
}
```

Because we return a `string` object, it's easy to call this in various contexts:

```
println("{}", bw::join(lads, ", "));
```

Output:

```
john, paul, george, ringo, billy
```

## There's more...

As with any of the library algorithms, this `join()` function will work with any container that supports *forward iterators*. For example, here's a `list` of `double` constants from the `numbers` library:

```
namespace num = std::numbers;
list<double> constants {num::pi, num::e, num::sqrt2};
println("{}", bw::join(constants, ", "));
```

Output:

```
3.141592653589793, 2.718281828459045, 1.4142135623730951
```

It will even work with a `ranges::view` object:

```
auto lads_view = views::join(lads);
println("{}", bw::join(lads_view, ":"));
```

Output:

```
j:o:h:n:p:a:u:l:g:e:o:r:g:e:r:i:n:g:o:b:i:l:l:y
```

This gives us the completely flattened `views::join` result, joined with separators by our `join()` function.

# Sort containers with std::sort

The problem of how to efficiently sort comparable elements is essentially solved. For most applications, there's no reason to re-invent this wheel. The STL provides an excellent sorting solution via the `std::sort()` algorithm. While the standard does not specify a sorting algorithm, it does specify a worst-case complexity of $O(n \log n)$, when applied to a range of $n$ elements.

Just a few decades ago, the *quicksort* algorithm was considered a good compromise for most uses and was generally faster than other comparable algorithms. Today we have *hybrid* algorithms that choose between different approaches according to the circumstances, often switching algorithms on the fly. Most current C++ libraries use a hybrid approach with some combination of *introsort* and *insertion sort*. `std::sort()` provides excellent performance under most common circumstances, such as partially ordered data, random data, and larger general-purpose data sets.

## How to do it

In this recipe, we'll examine the `std::sort()` algorithm. The `sort()` algorithm works with any container with random-access iterators. For simplicity, we will use a `vector` of `int` in this recipe:

- We'll start with a function to test if a container is sorted:

  ```
  void check_sorted(const auto &c) {
      if (!is_sorted(c.begin(), c.end())) print("un");
      print("sorted: ");
  }
  ```

  This becomes even simpler with `ranges::is_sorted()`:

  ```
  void check_sorted(const auto &c) {
      if (!ranges::is_sorted(c)) print("un");
      print("sorted: ");
  }
  ```

Using the `is_sorted()` algorithm this prints either `"sorted:"` or `"unsorted:"`, according to the result.

- We need a function to print our `vector`:

```
void printc(const auto &c) {
    check_sorted(c);
    for (auto& e : c) print("{} ", e);
    print("\n");
}
```

This function calls `check_sorted()` to display the status of the container before the values.

- Now we can define and print a `vector` of `int` in the `main()` function:

```
int main() {
    vector<int> v {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    printc(v);
    ...
}
```

The output looks like this:

```
  sorted: 1 2 3 4 5 6 7 8 9 10
```

- In order to test the `std::sort()` algorithm, we need an unsorted `vector`. Here's a simple function to randomize our container:

```
void randomize(auto& c) {
    static std::random_device rd;
    static std::default_random_engine rng(rd());
    std::shuffle(c.begin(), c.end(), rng);
}
```

The `std::random_device` class uses your system's hardware *entropy* source. Most modern systems have one, otherwise the library will simulate it. This can be a useful tool for testing purposes.

The `std::default_random_engine()` function generates random numbers from the entropy source. This is the default engine used by `std::shuffle()` to randomize the container.

We can now call `randomize()` with our container and print the result:

```
randomize(v);
printc(v);
```

Output:

```
  unsorted: 6 3 4 8 10 1 2 5 9 7
```

Of course, your output will be different because it's randomized. In fact, I get a different result every time I run it:

```
for(int i {3}; i; --i) {
    randomize(v);
    printc(v);
}
```

Output:

```
  unsorted: 3 1 8 5 10 2 7 9 6 4
  unsorted: 7 6 5 1 3 9 10 2 4 8
  unsorted: 4 2 3 10 1 9 5 6 8 7
```

- To sort the `vector`, we simply call `std::sort()`:

```
std::sort(v.begin(), v.end());
printc(v);
```

Or we can use the simpler `ranges::sort()`:

```
std::ranges::sort(v);
```

printc(v);Output:

```
  sorted: 1 2 3 4 5 6 7 8 9 10
```

By default, the `sort()` algorithm uses the < operator to sort the elements in the range specified by the supplied iterators.

- The `partial_sort()` algorithm will sort part of the container:

```
randomize(v);
auto middle {v.begin() + (v.size() / 2)};
std::partial_sort(v.begin(), middle, v.end());
printc(v);
```

`partial_sort()` takes three iterators: beginning, middle, and end. It sorts the container such that the elements before the middle are sorted. The elements after the middle are not guaranteed to be in the original order. Here's the output:

```
  unsorted: 1 2 3 4 5 10 7 6 8 9
```

Notice that the first five elements are sorted, and the rest are not.

- The `partition()` algorithm does not sort anything. It rearranges the container so that certain elements appear at the front of the container:

```
randomize(v);
printc(v);
std::partition(v.begin(), v.end(),
    [](int i) { return i > 5; });
printc(v);
```

The third argument is a *predicate* lambda that determines which elements will be moved to the front.

Output:

```
  unsorted: 4 6 8 1 9 5 2 7 3 10
  unsorted: 10 6 8 7 9 5 2 1 3 4
```

Notice that the values greater than 5 are moved to the front of the container.

- The `sort()` algorithms support an optional comparison function that may be used for non-standard comparisons. For example, given a class called `things`, with a `string` called `s` and an `int` called `i`:

```
struct things {
    string s;
    int i;
    string str() const {
        return format("({}, {})", s, i);
    }
};
```

We can create a `vector` of `things` objects:

```
vector<things> vthings { {"button", 40}, {"hamburger", 20},
    {"blog", 1000}, {"page", 100}, {"science", 60} };
```

We'll need a function to print them out:

```
void print_things(const auto& c) {
    for (auto& v : c) print("{} ", v.str());
    print("\n");
}
```

- Now we can sort and print the `vector` of `things`, using `std::ranges::sort()`:

```
std::ranges::sort(vthings,
        [](const things &lhs, const things &rhs) {
    return lhs.i < rhs.i;
});
print_things(vthings);
```

Output:

```
  (hamburger, 20) (button, 40) (science, 60) (page, 100) (blog, 1000)
```

Notice the comparison function sorts on the `int` member, so the result is sorted by `i`. We could instead sort on the `string` member:

```
std::ranges::sort(vthings,
        [](const things &lhs, const things &rhs) {
    return lhs.s < rhs.s;
});
print_things(vthings);
```

Now it's sorted by `s` and we get this output:

```
  (blog, 1000) (button, 40) (hamburger, 20) (page, 100) (science, 60)
```

## How it works

The `sort()` functions work by applying a sorting algorithm to a range of elements indicated by two iterators, for the beginning and end of the range.

By default, these algorithms use the `<` operator to compare elements. Optionally, they may use a *comparison function*, often provided as a lambda:

```
std::sort(vthings.begin(), vthings.end(),
        [](const things& lhs, const things& rhs) {
    return lhs.i_ < rhs.i_;
});
```

There is also a `ranges` version which takes a container as a single argument instead of separate iterators:

```
std::ranges::sort(vthings,
        [](const things &lhs, const things &rhs) {
    return lhs.i < rhs.i;
});
```

The comparison function takes two arguments and returns a `bool`. It has a signature equivalent to this:

```
bool cmp(const Type1& a, constType2&b);
```

The `sort()` functions use `std::swap()` to move elements. This is efficient in both compute cycles and memory usage, as it relieves the need to allocate space for reading and writing the

objects being sorted. This is also why the `partial_sort()` and `partition()` functions cannot guarantee the order of unsorted elements.

# Modify containers with std::transform

The `std::transform()` algorithm is remarkably powerful and flexible. One of the more commonly deployed algorithms in the library, it applies a *function* or *lambda* to each element in a container, storing the results in another container while leaving the original in place.

Given its power, it's deceptively simple to use.

## How to do it

In this recipe, we will explore a few applications for the `std::transform()` function:

- We'll start with a simple function that prints the contents of a container:

```
void printc(const auto& c, string_view s = "") {
    if (s.size()) print("{}: ", s);
    for (const auto& e : c) print("{} ", e);
    print("\n");
}
```

  We'll use this function to view the results of our transformations.

- In the `main()` function, we declare a couple of `vectors`:

```
int main() {
    vector<int> v1 {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    vector<int> v2 {};
    printc(v1, "v1");
    ...
}
```

  The `vectorv1` contains ten elements, `int` values `1` through `10`. The `vectorv2` is empty.

  This prints out the contents of `v1`:

```
v1: 1 2 3 4 5 6 7 8 9 10
```

- We can use the `transform()` function to insert the square of each value into `v2`:

```
println("squares:");
transform(v1.begin(), v1.end(), back_inserter(v2),
    [](int x) { return x * x; });
printc(v2, "v2");
```

The `transform()` function takes four arguments. The first two are the `begin()` and `end()` iterators for the source range. The third argument is the `begin()` iterator for the destination range. Because the `vectorv2` is empty, we use the `back_inserter()` algorithm to insert the results in `v2`. The fourth argument is the function to apply to the transformation. In this case, we're using a lambda to square the value.

Output:

```
squares:
v2: 1 4 9 16 25 36 49 64 81 100
```

- Of course, we can use `transform()` with any type. Here's an example that converts a `vector` of `string` objects to lowercase. First, we need a function to return the lowercase value of a string:

```
string str_lower(string s) {
    auto char_lower = [](char c) -> char {
        if(c >= 'A'&&c<= 'Z') return c + ('a' - 'A');
        else return c;
    };
    for (auto&c:s)c=char_lower(c);
    return s;
}
```

- This uses a lambda for the actual character conversion, rather than the `tolower()` function from the legacy C Standard Library which requires type casting to work with `char`-typed values in C++.
- Now we can use the `str_lower()` function with transform:

```
vector<string> vstr1 {"Mercury", "Venus", "Earth", "Mars",
    "Jupiter", "Saturn", "Uranus", "Neptune", "Pluto"};
vector<string> vstr2 {};
```

```
printc(vstr1, "vstr1");
println("str_lower:");
transform(vstr1.begin(), vstr1.end(), back_inserter(vstr2),
    [](string& x) { return str_lower(x); });
printc(vstr2, "vstr2");
```

This calls `str_lower()` for each element in `vstr1` and inserts the resulting lowercase string into `vstr2`. The result is:

```
vstr: Mercury Venus Earth Mars Jupiter Saturn Uranus Neptune Pluto
str_lower:
vstr: mercury venus earth mars jupiter saturn uranus neptune pluto
```

(Yes, Pluto will always be a planet to me.)

- There's also a `ranges` version of `transform`:

```
println("ranges squares:");
auto view1 = views::transform(v1,
    [](int x) { return x * x; });
printc(view1, "view1");
```

The `ranges` version has a more succinct syntax and returns a `view` object, rather than populating another container.

## How it works

The `std::transform()` function works very much like `std::copy()`, with the addition of the user-provided function. Each element in the input range is passed to the function, and the return value from the function is copy-assigned to the destination iterator. This makes `transform()` a singularly useful and powerful algorithm.

It's worth noting that `transform()` does not guarantee the elements will be *processed* in order. In other words, while the output order is preserved, the functor (or lambda) may be called out of order. If you need to ensure the order of the transformation, you will want to use a `for` loop instead:

```
v2.clear();    // reset vector v2 to empty state
for(auto e : v1) v2.push_back(e * e);
printc(v2, "v2");
```

Output:

```
v2: 1 4 9 16 25 36 49 64 81 100
```

# Search containers with std::find

The `algorithm` library contains a set of functions for finding elements in a container. The `std::find()` function, and its derivatives, search sequentially through a container and return an iterator pointing to the first matching element, or the `end()` iterator if there's no match.

## How to do it

The `find()` algorithm works with any container that satisfies the *Forward* or *Input* iterator qualifications. For this recipe, we'll use `vector` containers. The `find()` algorithm searches sequentially for the first matching element in a container. In this recipe, we'll walk through a few examples:

- Let's start by declaring a `vector` of `int` in the `main()` function:

```
int main() {
    const vector<int> v {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    ...
}
```

- We can now search for the element with the value 7:

```
auto it1 = find(v.begin(), v.end(), 7);
if (it1 != v.end()) println("found: {}", *it1);
else println("not found");
```

  The `find()` algorithm takes three arguments: `begin()` and `end()` iterators, and the value to search. It returns an iterator to the first element it finds, or the `end()` iterator if the search failed to find a match.

  Output:

```
found: 7
```

- We can also search for something more complex than a scalar. The object needs to support the equality comparison operator ==. Here's a simple `struct` with an `operator==()` overload:

```
struct City {
    string name {};
    unsigned pop {};
    bool operator==(const City& o) const {
        return name == o.name;
    }
    string str() const {
        return format("[{}, {}]", name, pop);
    }
};
```

The struct is named `City` and it has two members, `name` and `pop`, for the name and population of the city.

The `operator=()` overload only compares the `name` members. This allows us to search with only a `name`.

I've also included an `str()` function that returns a `string` representation of a `City` element.

- Now we can declare a `vector` of `City` elements:

```
const vector<City> c{
    { "London", 9425622 },
    { "Berlin", 3566791 },
    { "Tokyo",  37435191 },
    { "Cairo",  20485965 }
};
```

- We can search the `vector` of `City` just as we did with the `vector` of `int`:

```
auto it2 = find(c.begin(), c.end(), City({"Berlin", {}}));
if (it2 != c.end()) println("found: {}", it2->str());
else println("not found");
```

Output:

```
found: [Berlin, 3566791]
```

Notice that we passed a `City` object, with only a `name` parameter, to the `find()` function. If we want to compare against a string literal (a `const` C-string), we could add another `operator==()` overload:

```
bool operator==(const char* cstr) const {
    return name == cstr;
}
```

Now we can perform our search with a string literal:

```
auto it3 = find(c.begin(), c.end(), "Berlin");
if (it3 != c.end()) println("found: {}", it3->str());
else println("not found");
```

Output:

```
found: [Berlin, 3566791]
```

- If we want to search on the pop member instead of `name`, we can use the `find_if()` function with a predicate:

```
auto it4 = find_if(begin(c), end(c),
    [](const City& item) { return item.pop > 20'000'000; });
if (it4 != c.end()) println("found: {}", it4->str());
else println("not found");
```

Note the use of apostrophe `'` separators in the literal numeric values. This is a C++14 feature which makes large numbers easy to read. In modern C++, `20'000'000` is treated the same as `20000000`.

The predicate tests the pop member for elements with a pop value greater than 20,000,000, so we get this output:

```
found: [Tokyo, 37435191]
```

- Notice that the result from `find_if()` returns only the first element that satisfies the predicate, even though there are two elements in the `vector` with pop values greater than 20,000,000.
  The `find()` and `find_if()` functions return only one iterator. The `ranges` library provides `ranges::views::filter()`, a *view adapter* that will give us all the matching iterators without disturbing our `vector`:

```
auto vw1 = views::filter(c,
    [](const City& c){ return c.pop > 20'000'000; });
for (const City& e : vw1) println("{}", e.str());
```

This gives us both matching elements in our output:

```
[Tokyo, 37435191]
[Cairo, 20485965]
```

## How it works

The `find()` and `find_if()` functions search sequentially through a container, checking each element until they find a match. If they find a match, they return an iterator pointing to that match. If they reach the `end()` iterator without finding a match, they return the `end()` iterator to indicate no match was found.

The `find()` function takes three arguments: `begin()` and `end()` iterators, and a search value. The signature looks like this:

```
template<class InputIt, class T>
constexpr InputIt find(InputIt, InputIt, const T&)
```

The `find_if()` function uses a predicate instead of a value:

```
template<class InputIt, class UnaryPredicate>
constexpr InputIt find_if(InputIt, InputIt, UnaryPredicate)
```

These functions are simple, and work with any Forward iterator, so they don't require random access. This also means they work with linear $O(n)$ complexity. Faster performance would require more complex iterators.

## There's more...

Both `find()` functions search sequentially and return when they find the first match. If you want to find more matching elements, you can use the `filter()` function from the `ranges` library:

```
template<ranges::viewable_range R, class Pred>
constexpr ranges::view auto ranges::views::filter(R&&,Pred&&);
```

The `filter()` function returns a *view*, a non-destructive window into the container with only the filtered elements. We can then use the view as we would any other container:

```
auto vw1 = views::filter(c,
    [](const City& c){ return c.pop > 20000000; });
for (const City& e : vw1) println("{}", e.str());
```

Output:

```
[Tokyo, 37435191]
[Cairo, 20485965]
```

## The new contains() method

Beginning with C++23, some STL containers include a `contains()` method which returns a simple `true` or `false` indicating whether a value exists in the container.

```
auto needle = "n";
for (auto [city, pop] : c) {
    if (name.contains(needle)) {
        println("{} found in {}", needle name);
    }
}
```

In this example we use the `string_view::contains()` method to discover which cities have the letter `n` in their name.

Output:

```
n found in London
n found in Berlin
```

The `string` and `string_view` classes, along with the `map` and `set` containers, now include a `contains()` method.

# Limit container values with std::clamp

Introduced with C++17, the `std::clamp()` function can be used to limit the range of a numeric scalar to within minimum and maximum values. The function is optimized to use *move semantics*, where possible, for maximum speed, and efficiency.

The `clamp()` function is in the `<algorithm>` header.

## How to do it

We can use `clamp()` to constrain the values of a container by using it in a loop or with the `transform()` algorithm. Let's look at some examples.

- We start with a simple function for printing out the values of a container:

  ```
  void printc(auto& c, string_view s = "") {
      if (s.size()) print("{}: ", s);
      for (auto e : c) print("{:>5} ", e);
      print("\n");
  }
  ```

  Notice the *format string* `"{:>5} "`. This right-aligns each value to 5 spaces, for a tabular view.

- In the `main()` function, we'll define an *initializer list* for use with our container. This allows us to use the same values more than once:

  ```
  int main() {
      const auto il = {0, -12, 2001, 4, 5, -14,
          100, 200, 30000};
      ...
  }
  ```

  That's a nice range of values to work with `clamp()`.

- Let's also define some constants for use as our limits:

```
constexpr int ilow {0};
constexpr int ihigh {500};
```

We'll use these values in our calls to `clamp()`.

- Now we can define a container in our `main()` function. We'll use a `vector` of `int`:

```
vector<int> voi {il};
println("vector voi before:");
printc(voi);
```

Using the values from our initializer list, the output is:

```
vector voi before:
    0   -12  2001     4     5   -14   100   200 30000
```

- Now we can use a `for` loop with `clamp()` to limit the values to between `0` and `500`:

```
println("vector voi after:");
for (auto&e:voi)e=clamp(e,ilow,ihigh);
printc(voi);
```

This applies the `clamp()` function to each value in the container, using `0` and `500` for the low and high limits, respectively. Now, the output is:

```
vector voi before:
    0   -12  2001     4     5   -14   100   200 30000
vector voi after:
    0     0   500     4     5     0   100   200   500
```

After the `clamp()` operation, the negative values are `0` and the values greater than `500` are `500`.

- We can do the same thing with the `transform()` algorithm, using `clamp()` in a lambda. This time we'll use a `list` container:

```
list<int> loi {il};
println("list loi before:");
printc(loi);
```

```
transform(loi.begin(), loi.end(), loi.begin(),
    [=](auto e){ return clamp(e, ilow, ihigh); });
println("list loi after:");
printc(loi);
```

The output is the same as in the version with a `for` loop:

```
list loi before:
    0   -12  2001     4     5   -14   100   200 30000
list loi after:
    0     0   500     4     5     0   100   200   500
```

## How it works

The `clamp()` algorithm is a simple function that looks something like this:

```
template<class T>
constexpr const T& clamp( const T& v, const T& lo,
        const T& hi ) {
    return less(v, lo) ? lo : less(hi, v) ? hi : v;
}
```

If the value of `v` is less than `lo`, it returns `lo`. If `hi` is less than `v`, it returns `hi`. The function is fast and efficient.

In our examples, we used a `for` loop to apply `clamp()` to a container:

```
For (auto&v:voi)v=clamp(v,ilow,ihigh);
```

We also used the `transform()` algorithm with `clamp()` in a lambda:

```
transform(loi.begin(), loi.end(), loi.begin(),
    [=](auto v){ return clamp(v, ilow, ihigh); });
```

In my experiments, both versions gave the same results, and both produced similar code from the GCC compiler. There was a slight difference in compiled size (the version with the `for` loop was smaller, as expected) and a negligible difference in performance.

In general, I prefer the `for` loop, but the `transform()` version may be more flexible in some circumstances.

# Sample data sets with std::sample

The `std::sample()` algorithm takes a random sample of a sequence of values and populates a destination container with the sample. It is useful when analyzing a larger set of data, where the random sample is taken to be representative of the whole.

A sample set allows us to approximate the characteristics of a large set of data, without analyzing the full set. This provides efficiency in exchange for accuracy, a fair trade-off in many circumstances.

## How to do it

In this recipe, we use an array of 200,000 random integers with a *standard normal distribution*. We'll sample a few hundred values to create a histogram of the frequency of each value.

- We'll start with a simple function to return a rounded `int` from a `double`. The standard library lacks such a function and we'll need it later:

```
int iround(const double& d) {
    return static_cast<int>(std::round(d));
}
```

  The standard library provides several versions of `std::round()`, including one that returns a `long int`. But we need an `int`, and this is a simple solution that avoids compiler warnings about narrowing conversion while hiding away the unsightly `static_cast`.

- In the `main()` function, we start with some useful constants:

```
int main() {
    constexpr size_t n_data {200000};
    constexpr size_t n_samples {500};
    constexpr int mean {};
    constexpr size_t dev {3};
    ...
}
```

  We have values for `n_data` and `n_samples`, used for the size of the data and sample containers, respectively. We also have values for `mean` and `dev`, the *mean* and *standard deviation* parameters for the distribution of random values.

- We now set up our *random number generator* and *distribution* objects. These are used to initialize the source data set:

```
std::random_device rd {};
std::mt19937 rng(rd());
std::normal_distribution<> dist(mean, dev);
```

The `random_device` object provides access to the hardware random number generator. The `mt19937` class is an implementation of the *Mersenne Twister* random number algorithm, a high-quality algorithm that will perform well on most systems with a data set of the size we're using. The `normal_distribution` class provides a distribution of random numbers around the *mean* with the *standard deviation* provided.

- Now we populate an array with an `n_data` number of random `int` values:

```
array<int, n_data> data {};
for (auto&e:data)e=iround(dist(rng));
```

The `array` container is fixed in size, so the template parameters include a `size_t` value for the number of elements to allocate. We use a `for` loop to populate the array.

The `rng` object is the hardware random number generator. This is passed to `dist()`, our `normal_distribution` object, and then to `iround()`, our integer rounding function.

- At this point, we have an array with 200,000 data points. That's a lot to analyze, so we use the `sample()` algorithm to take a sample of 500 values:

```
array<int, n_samples> samples {};
sample(data.begin(), data.end(), samples.begin(),
    n_samples, rng);
```

We define another `array` object to hold the samples. This one is `n_samples` in size. Then we use the `sample()` algorithm to populate the array with `n_samples` random data points.

- We create a histogram to analyze the samples. A `map` structure is perfect for this so we can easily map the frequency of each value:

```
std::map<int, size_t> hist {};
for (const int i : samples) ++hist[i];
```

The `for` loop takes each value from the `samples` container and uses it as a key in the `map`. The increment expression `++hist[i]` counts the number of occurrences of each value in the sample set.

- We print out the histogram using a `format` string in the `println()` function:

```
constexpr size_t scale {3};
println("{:>3} {:>5} {:<}/{}", "n", "count",
    "graph", scale);
for (const auto& [value, count] : hist) {
    println("{:>3} ({:>3}) {}", value, count,
        string(count / scale, '*'));
}
```

The `format` specifiers make space for a certain number of characters. The angle bracket specifies right alignment.

The `string(count, char)` constructor creates a `string` object with a character repeated the number of times specified, in this case, *n* asterisk characters `*`, where *n* is `count/scale`, the frequency of a value in the histogram, divided by the `scale` constant.

The output looks like this:

```
$ ./sample
  n count graph/3
-9 (  2)
-7 (  5) *
-6 (  9) ***
-5 ( 22) *******
-4 ( 24) ********
-3 ( 46) ***************
-2 ( 54) ******************
-1 ( 59) *******************
  0 ( 73) ************************
  1 ( 66) **********************
  2 ( 44) **************
  3 ( 34) ***********
  4 ( 26) ********
  5 ( 18) ******
  6 (  9) ***
  7 (  5) *
```

```
8 (  3) *
9 (  1)
```

This is a nice graphical representation of the histogram. The first number is the value, the second number is the frequency of the value, and the asterisks are a visual representation of the frequency, where each asterisk represents `scale` (3) occurrences in the sample set.

Your output will differ each time you run the code.

## How it works

The `std::sample()` function selects a specific number of elements from random locations in the source container and copies them to the destination container.

The signature of `sample()` looks like this:

```
OutIter sample(SourceIter, SourceIter, OutIter,
SampleSize,RandNumGen&&);
```

The first two arguments are `begin()` and `end()` iterators on a container with the full data set. The third argument is an iterator for the destination of the samples. The fourth argument is the sample size, and the final argument is a random number generator function.

The `sample()` algorithm uses *uniform distribution*, so each data point has the same chance of being sampled.

# Use ranges for more powerful algorithms

*Ranges* are an abstraction that allow *algorithms* to work with any sequence of elements. A range is essentially the normalization of `begin()` and `end()` iterators as the boundaries of an operational sequence in the STL. Many classes in the `ranges` library also return `view` objects. A `view` is a non-owning adapter that provides sequential access to underlying data.

The `ranges` library was introduced with C++20 and expanded in C++23. It provides alternative, more efficient, and convenient versions of most of the functions in the `algorithms` library.

**Note**

At the time of this writing, the most complete implementation of the `ranges` library is in the Preview Edition of Microsoft Visual C++. The examples in this recipe have been tested on that compiler.

In this recipe we will consider a few representative examples of algorithms in the `ranges` library.

## How to do it

The `ranges` library includes alternative versions of a number of functions from the `algorithms` library. Let's consider some examples.

- First, we'll need a few common elements for these examples. We'll use these namespace aliases for `ranges` and `views`:

  ```
  namespace ranges = std::ranges;
  namespace views = ranges::views;
  ```

  We'll need a function to print a range:

  ```
  void p_range(auto r) {
      for (const auto& v : r) print("{} ", v);
      print("\n");
  }
  ```

  This uses an abbreviated function template to deduce the type of the range.

  Because the range may be a non-owning `view`, we cannot use the `const` qualifier or a reference for the range `r` parameter, but we can for the value `v` in the `for` loop.

- We'll also define a couple of lambdas for use with the algorithms:

  ```
  auto even = [](auto i) { return 0 == i % 2; };
  auto square = [](auto i) { return i * i; };
  ```

  `even` is a predicate that filters even-numbered values and `square` returns the square of a numeric value.

  In the `main()` function, we'll use a simple range of integers for these examples:

  ```
  const auto r = views::iota(0, 10);
  ```

- The `views::iota()` function returns a `view` object representing a range of integers. This view is lazy, with its members generated as they are needed.

We can print the range `r` using our `p_range()` function:

```
p_range(r);
```

We get this output:

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

The `views::iota(0, 10)` function call returns a *bounded range* of `10` integers beginning with `0`. The first argument is the start of the series and the second argument is the number of elements.

- We can populate a container from the range using `ranges::copy()`:

```
vector<int> v1 (r.size());
ranges::copy(r, v1.begin());
p_range(v1);
```

We initialize the `vector` with the size of the `range` object to create a vector with room for the range. The `ranges::copy` algorithm takes two arguments, the range and an iterator. We use the `begin()` iterator from our `vector` as the target of the copy operation. We then print the `vector` with `p_range()` for this output:

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

- Or we can use `back_inserter()` instead of initializing the `vector` with the size of the range:

```
ranges::copy(r, std::back_inserter(v2));
```

This gives us the same result:

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

- The ranges library provides a new *piping* syntax, using the | operator (*bitwise or*) to chain together range adapters, similar to how Unix command pipes work on the command line:

```
vector<int> v3 = r
  | views::transform([](auto i) { return i * 2; })
```

```
    | ranges::to<vector<int>>();
p_range(v3);
```

This creates a new `vector`, `v3`, and populates it with the output of `r`, piped into the `transform` algorithm with a lambda that multiplies by `2`, piped into the `ranges::to` algorithm to populate the `vector`. The output is:

```
0 2 4 6 8 10 12 14 16 18
```

The piping syntax is nicely readable and produces efficient code due to the lazy evaluation of ranges and views.

- We can scramble `vectorv1` using the `ranges::shuffle` algorithm. First, we need a random number generator, which we can create using this function:

```
auto rng() {
    std::random_device rd {};
    return std::mt19937(rd());
}
```

Now we can use `ranges::shuffle` to scramble `v1`:

```
ranges::shuffle(v1, rng());
p_range(v1);
```

My output looks like this:

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

Your output will be different. In fact, each time you run it you should get a different randomization. Let's try running it `5` times:

```
for (int i {}; i < 5; ++i) {
    ranges::shuffle(v1, rng());
    p_range(v1);
}
```

Output:

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

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

Of course, your result will differ each time you run it.

- Now we can sort `v1` with `ranges::sort()`:

```
ranges::sort(v1);
p_range(v1);
```

After the `sort`, `v1` looks like this:

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

- We've seen the `views::iota()` function used with two arguments, where the second argument is the length of the series.
If we call `views::iota()` without a second argument, `iota()` will generate an indefinite, or *unbounded*, series. We can use `views::take()` to take a specific number of values from an unbounded view:

```
p_range(views::iota(0) | views::take(10));
```

With just one argument, `iota()` generates an unbounded view. The `take(10)` call returns the first `10` elements from the `iota()` view. Here, we pass the pipeline directly to our `p_range()` function and we get the first `10` elements of the view:

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

- Most of the functions in the `algorithm` library now have ranges analogs. Here's an example that pipes together `filter`, `transform`, and `reverse`:

```
p_range(r | views::filter(even) | views::transform(square) |
views::reverse);
```

This gives us the following result:

```
64 36 16 4 0
```

## How it works

A *range* is an abstraction that represents a sequence of elements with `begin()` and `end()` iterators. The purpose is to make it easier for algorithms to operate on any iterable sequence, including containers, views, or any iterable object.

The `std::ranges` library provides a set of tools and algorithms that operate on ranges directly, rather than requiring separate iterators. For example, the `std::sort()` algorithm requires iterators. Its signature looks like this:

```
template< class It >
void sort( It first, It last );
```

It is typically called with the `begin()` and `end()` iterators of a container:

```
vector<int> v {3, 9, 7, 4, 6, 5, 2, 1, 0, 8};
std::sort(v.begin(), v.end());
for (const auto& i : v) print("{} ", i);
print("\n");
```

The arguments to the `std::sort()` algorithm are iterators which specify the range to be sorted.

The `std::ranges::sort()` version of the same algorithm looks like this:

```
template< ranges::random_access_range R >
sort(R&&r);
```

This means that we can simply pass the container directly to `std::ranges::sort()` instead of its iterators:

```
vector<int> v {3, 9, 7, 4, 6, 5, 2, 1, 0, 8};
std::ranges::sort(v);
for (const auto& i : v) print("{} ", i);
print("\n");
```

The output of both of these examples is the same:

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

For many circumstances, this syntax is simpler and more concise.

A *view* is a further abstraction that exposes iterators for a non-owning sequence of values. From the caller's perspective, it looks very much like a range, but its values are owned by an underlying object. *View adapters* can be chained using the piping syntax, which leverages the bitwise or operator | in much the same way that streams leverage the bitwise left-shift operator <<. This allows us to chain views together like this:

```
for (const auto& i : views::iota(0, 10) |
  views::filter(even) |
  views::transform(square) |
  views::reverse) {
    print("{} ", i);
}
print("\n");
```

Output:

```
64 36 16 4 0
```

As these new ranges and views versions of the algorithm library become more widely implemented, they will allow us to write more expressive and intuitive code.

# Generate permutations of data sequences

There are many use cases for permutations, including testing, statistics, research, and more. The `next_permutation()` algorithm generates permutations by re-ordering a container to the next *lexicographical* permutation.

## How to do it

For this recipe, we'll print out the permutations of a set of three strings:

- We start by creating a short function for printing the contents of a container:

  ```
  void printc(const auto& c, string_view s = "") {
      if (s.size()) print("{}: ", s);
      for (const auto& e : c) print("{} ", e);
      print("\n");
  }
  ```

  We'll use this simple function to print our data set and permutations.

- In the `main()` function, we declare a `vector` of `string` objects and sort it with the `sort()` algorithm.

```
int main() {
    vector<string> vs {"dog", "cat", "velociraptor"};
    sort(vs.begin(), vs.end());
    ...
}
```

The `next_permutation()` function requires a sorted container.

- Now we can list the permutations with `next_permutation()` in a `do` loop:

```
do {
    printc(vs);
} while (next_permutation(vs.begin(), vs.end()));
```

The `next_permutation()` function modifies the container and returns `true` if there is another permutation, or `false` if not.

The output lists six permutations of our three pets:

```
cat dog velociraptor
cat velociraptor dog
dog cat velociraptor
dog velociraptor cat
velociraptor cat dog
velociraptor dog cat
```

## How it works

The `std::next_permutation()` algorithm generates *lexicographical* permutations of a set of values, that is, permutations based on dictionary ordering. The input must be sorted because the algorithm steps through permutations in lexicographical order. So, if you start with a set like 3, 2, 1, it will terminate immediately as this is the last lexicographical order of those three elements.

For example:

```
vector<string> vs {"velociraptor", "dog", "cat"};
do {
```

```
    printc(vs);
} while (next_permutation(vs.begin(), vs.end()));
```

This gives us the following output:

```
velociraptor dog cat
```

While the term *lexicographical* implies alphabetical ordering, the implementation uses standard comparison operators, so it works on any sortable values.

Likewise, if values in the set repeat, they are only counted according to *lexicographical* order. Here we have a `vector` of `int` with two repeating sequences of five values:

```
vector<int> vi {1, 2, 3, 4, 5, 1, 2, 3, 4, 5};
sort(vi.begin(), vi.end());
printc(vi, "vi sorted");
long count {};
do {
    ++count;
} while (next_permutation(vi.begin(), vi.end()));
print("number of permutations: {}\n", count);
```

Output:

```
vi sorted: 1 1 2 2 3 3 4 4 5 5
number of permutations: 113400
```

There are 113,400 permutations of these values. Notice that it's not *10!* (3,628,800) because some values repeat. Since *3,3* and *3,3* sort the same, they are not different *lexicographical* permutations.

In other words, if I list the permutations of this short set:

```
vector<int> vi2 {1, 3, 1};
sort(vi2.begin(), vi2.end());
do {
    printc(vi2);
} while (next_permutation(vi2.begin(), vi2.end()));
```

We get only three permutations, not *3!* (6), because of the repeating values:

```
1 1 3
1 3 1
3 1 1
```

# Merge sorted containers

The `std::merge()` algorithm takes two sorted sequences and creates a third merged and sorted sequence. This technique is often used as part of a *merge sort*, allowing very large amounts of data to be broken down into chunks, sorted separately, and merged into one sorted target.

## How to do it

For this recipe, we'll take two sorted `vector` containers and merge them into a third `vector` using `std::merge()`.

- We start with a simple function to print the contents of a container:

```
void printc(const auto& c, string_view s = "") {
    if (s.size()) print("{}: ", s);
    for (const auto& e : c) print("{} ", e);
    print("\n");
}
```

  We'll use this to print the source and destination sequences.

- In the `main()` function, we'll declare our source vectors, along with the destination vector, and print them out:

```
int main() {
    vector<string> vs1 {"dog", "cat", "velociraptor"};
    vector<string> vs2 {"kirk", "sulu", "spock"};
    vector<string> dest {};
    printc(vs1, "vs1");
    printc(vs2, "vs2");
    ...
}
```

The output is:

```
vs1: dog cat velociraptor
vs2: kirk sulu spock
```

- Now we can sort our vectors and print them again:

```
sort(vs1.begin(), vs1.end());
sort(vs2.begin(), vs2.end());
printc(vs1, "vs1 sorted");
printc(vs2, "vs2 sorted");
```

Output:

```
vs1 sorted: cat dog velociraptor
vs2 sorted: kirk spock sulu
```

Now that our source containers are sorted, we can merge them for our final merged result:

```
merge(vs1.begin(), vs1.end(), vs2.begin(), vs2.end(),
    back_inserter(dest));
printc(dest, "dest");
```

Output:

```
dest: cat dog kirk spock sulu velociraptor
```

This output represents the merge of the two sources into one sorted vector.

## How it works

The `merge()` algorithm takes `begin()` and `end()` iterators from both sources and an output iterator for the destination:

```
OutputIt merge(InputIt1, InputIt1, InputIt2, InputIt2, OutputIt)
```

It takes the two input ranges, performs its merge/sort operation, and sends the resulting sequence to the output iterator.

## Get this book's PDF version and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.



UNLOCK NOW

*Note: Keep your invoice handy. Purchases made directly from Packt don't require an invoice.*

# 7

# Strings and Formatting

The STL `string` class is a powerful, full-featured tool for storing, manipulating, and displaying character-based data. It has much of the convenience you would find in a high-level scripting language while providing the agility and efficiency you expect from C++.

The `string` class is based on `basic_string`, a contiguous container class that may be instantiated with any character type. Its template class signature looks like this:

```
template<
    typename CharT,
    typename Traits = std::char_traits<CharT>,
    typename Allocator = std::allocator<CharT>
> class basic_string;
```

The `Traits` and `Allocator` template parameters are usually left to their default values.

The underlying storage of `basic_string` is a contiguous sequence of `CharT`, and can be accessed with the `data()` member function:

```
const std::basic_string<char> s {"hello"};
const char * sdata = s.data();
for(size_t i{}; i < s.size(); ++i) {
    print("{} ", sdata[i]);
}
print("\n");
```

Output:

```
h e l l o
```

The `data()` member function returns a `CharT*` that points to the underlying array of characters. Since C++11, the array returned by `data()` is null-terminated, making `data()` equivalent to `c_str()`.

The `basic_string` class includes many of the methods you would find in other contiguous-storage classes, including `insert()`, `erase()`, `push_back()`, `pop_back()`, and others. These methods operate on the underlying array of `CharT`.

`std::string` is a *type alias* for `std::basic_string<char>`:

```
using std::string = std::basic_string<char>;
```

For most purposes, you'll use `std::string`.

Before C++20, string formatting had been a weak point in the STL. We were given the imperfect choice between the cumbersome STL `iostreams` or the archaic legacy `printf()`. Beginning with C++20 and the `format` library, STL string formatting has finally grown up. Closely based on Python's `str.format()` method, the new `format` library is fast and flexible, providing many of the advantages of both `iostreams` and `printf()`, along with good memory management and type safety.

In this chapter, we will cover these subjects and more in the following recipes:

- Use `string_view` as a lightweight string object
- Format text with the `format` library
- Print strings with `std::print` and `std::println`
- Concatenate strings
- Transform strings
- Trim whitespace from strings
- Read strings from user input
- Count words in a file
- Initialize complex structures from file input
- Customize a string class with `char_traits`
- Parse strings with regular expressions

# Use string_view as a lightweight string object

The `string_view` class provides a lightweight alternative to the `string` class. Instead of maintaining its own data store, `string_view` operates on a *view* of a C-string. This makes `string_view` smaller and more efficient than `std::string`. It's useful in cases where you need a string object but don't need the more memory- and computation-intensive features of `std::string`.

For example, we've been using a function like this in many of our recipes:

```
void printc(const auto& c, string_view s = "") {
    if (s.size()) print("{}: ", s);
    for (const auto& e : c) print("{} ", e);
    print("\n");
}
```

This function prints the contents of a container, along with an optional description string, like this output from the permutations example in *Chapter 6*:

```
dest: cat dog kirk spock sulu velociraptor
```

We pass the description string to the `printc()` function as a literal C-string:

```
printc(dest, "dest");
```

The literal string is used to construct the `string_view` object `s` in `printc()`. `string_view` is convenient here because we're able to concisely check the `size()` method to see if there is a description to print:

```
if (s.size()) print("{}: ", s);
```

We could have instead used a reference to a `std::string`:

```
void printc(const auto& c, const string& s = "")
```

This would construct a `string` object from the literal c-string.

Because the `string` object keeps its own copy of the data it must first be copied, which uses both space and compute cycles. On the other hand, `string_view` simply refers to the original

C-string for its data, thus providing much of the utility of `std::string` without the additional cost of copying and storing the data.

The underlying data is available via the `string_view::data()` method, which returns a `const`-qualified pointer.

## How to do it

The `string_view` class looks deceptively similar to the STL `string` class, but it works a bit differently. Let's consider some examples:

- Here's an STL `string` initialized from a C-string (array of `char`):

```
char text[] {"hello"};
string greeting {text};
text[0] = 'J';
println("{} {}", text, greeting);
```

Output:

```
Jello hello
```

Notice that we modify the `text` array with `text[0] = 'J'`. This does not change the characters in the original `char` array. This is because `string` object carries its own copy of the underlying data.

- When we do the same with a `string_view`, we get a different result:

```
char text[] {"hello"};
string_view greeting {text};
text[0] = 'J';
println("{} {}", text, greeting);
```

Output:

```
Jello Jello
```

The `string_view` object creates a *view* of the underlying data. It does *not* make its own copy. This results in significant efficiencies but also allows for side effects.

- Because `string_view` doesn't copy the underlying data, the source data must remain in scope for the duration of the `string_view` object. So, this does not work:

```
string_view sv() {
    const char text[] {"hello"};  // temporary storage
    string_view greeting {text};
    return greeting;
}

int main() {
    auto greeting = sv();     // data out of scope
    println("{}", greeting);  // output undefined
}
```

  Because the underlying data is out of scope in the `main()` function, the `greeting` object in `main()` is no longer valid by the time we use it, resulting in undefined output and possible memory leaks.

- The `string_view` class has constructors that work for different types of underlying containers. This includes character arrays (`const char*`), contiguous *ranges* (including `std::string`), and other `string_view` objects. For example, constructing a `string_view` from a `string` uses a *ranges* constructor:

```
string str {"hello"};
string_view greeting {str};
println("{}", greeting);
```

  Output:

```
hello
```

- There is also a `string_view` literal operator `sv`, defined in the `std::literals` namespace:

```
using namespace std::literals;
auto sv = "hello"sv;
println("{}", sv.substr(1,4));
```

  This constructs a `constexpr string_view` object and calls its method `substr()` to return the 4 values starting at index 1 (the second character).

Output:

```
ello
```

## How it works

The `string_view` class is effectively an *iterator adaptor* on a contiguous sequence of characters. The implementation typically has two members: a `const CharT*` and a `size_t`. It works by wrapping a `contiguous_iterator` around the source data.

This means that you can use it like `std::string` for many purposes, with a few important distinctions:

- The copy constructor does not copy the data. This means that when you make a copy of a `string_view`, each copy operates on the same underlying data:

```
char text[] {"hello"};
string_view sv1 {text};
string_view sv2 {sv1};
string_view sv3 {sv2};
string_view sv4 {sv3};
println("{} {} {} {}", sv1, sv2, sv3, sv4);
text[0] = 'J';
println("{} {} {} {}", sv1, sv2, sv3, sv4);
```

Output:

```
hello hello hello hello
Jello Jello Jello Jello
```

- Keep in mind that when you pass a `string_view` to a function, it uses the copy constructor:

```
void f(string_view sv) {
    if (sv.size()) {
        char* x = (char*) sv.data();  // dangerous
        x[0] = 'J';  // modifies the source
    }
    println("f(sv): {} {}", (void*) sv.data(), sv);
}

int main() {
```

```
    char text[] {"hello"};
    string_view sv1 {text};
    println("sv1: {} {}", (void*) sv1.data(), sv1);
    f(sv1);
    println("sv1: {} {}", (void*) sv1.data(), sv1);
}
```

Output:

```
sv1: 0x7fff595bcb0a hello
f(sv): 0x7fff595bcb0a Jello
sv1: 0x7fff595bcb0a Jello
```

Notice that the address of the underlying data (returned by the `data()` member function) is the same for all instances of the `string_view`. That's because the `string_view` class does not make a copy of the underlying data. Even though the `string_view` member pointer is `const`-qualified, it's still possible to *cast away* the `const` qualifier, though that's not recommended because it could cause unintended side effects. But it is worth noting that the data is never copied.

- Because the `string_view` class does not own its source data, it lacks methods that directly operate on the underlying string. Methods such as `append()`, `operator+()`, `push_back()`, `pop_back()`, `replace()`, and `resize()`, which are supported in `string`, are not supported in `string_view`.

If you need to concatenate strings with the **+** operator, you'll need a `std::string` object. For example, this does not work with `string_view`:

```
sv1 = sv2 + sv3 + sv4; // does not work
```

You'll need to use `string` instead:

```
char text[] {"hello"};
string str1 {text};
string str2 {str1};
string str3 {str2};
string str4 {str3};
str1 = str2 + str3 + str4; // works
println("{}", str1);
```

Output:

```
hellohellohello
```

We may access the underlying data of a `string_view` object with its `data()` method.

```
auto str_data = sv1.data();
for (auto i = 0; i < sv1.size(); ++i) {
print("{} ".format(str_data[i]))
}
print("\n")
```

Output:

```
J e l l o
```

# Format text with the format library

Before C++20, if you wanted to format text, you could use either the legacy `printf` functions or the STL `iostream` library. Both have their strengths and flaws.

The `printf`-based functions are inherited from C and have proven efficient, flexible, and convenient for over 50 years. The formatting syntax can look a bit cryptic, but it's simple enough once you get used to it.

```
printf("Hello, %s\n", c_string);
```

The main weakness in `printf` is its lack of type safety. The common `printf()` function (and its relatives) use C's *variadic arguments* model to pass parameters to a formatter. This works great when it works, but it can cause serious problems when a parameter type doesn't match its corresponding format specifier. Modern compilers do as much type-checking as they can, but the model is inherently flawed and the protection can only go so far.

The STL `iostream` library brings type safety at the expense of readability and run-time performance. The `iostream` syntax is unusual, yet familiar. It overloads the *bitwise left-shift operator* (`<<`) to allow a chain of objects, operands, and *formatting manipulators*, which produce the formatted output.

```
cout << "Hello, " << str << endl;
```

The weakness of `iostream` is its complexity, in both syntax, and implementation. Building a formatted string can be verbose and obscure. Many of the formatting manipulators must be reset after use, or they create cascading formatting errors that can be difficult to debug. The library itself is vast and complex, resulting in code significantly larger and slower than its `printf` equivalent.

This left C++ programmers with little option but to choose between two flawed systems. The new `format` library, introduced with C++20, provides an elegant and efficient solution to this dilemma.

The `format` library is in the `<format>` header. As of mid-2024, `format` is implemented and supported on all the major compiler systems. The `format` library is modeled on the `str.format()` method from Python 3. *Format strings* are substantially the same as those in Python and, for most purposes, they should be interchangeable. Let's examine some simple examples:

## How to do it

- In its simplest form, the `format()` function takes a `string_view` format string and a *variadic parameter pack* of arguments. It returns a `string`. Its function signature looks like this:

```
template<typename... Args>
string format(string_view fmt, const Args&...args);
```

- The `format()` function returns a `string` with a textual representation of virtually any type or value. For example:

```
string who {"everyone"};
int ival {42};
double pi {std::numbers::pi};
auto s1 = format("Hello, {}!", who);
auto s2 = format("Integer: {}", ival);
auto s3 = format("π: {}", pi);

for (auto s : {s1, s2, s3}) {
    println("{}", s);
}
```

Output:

```
Hello, everyone!
Integer: 42
π: 3.141592653589793
```

The *format string* uses braces {} as a placeholder. With no format specifiers, the braces are effectively a type-safe *placeholder* which will convert a value of any compatible type to a reasonable string representation.

- You can include multiple placeholders in your format string, like this:

```
auto s4 = format("Hello {} {}", ival, who);
```

Output:

```
Hello 42 everyone
```

- You can specify the order of the replacement values. This could be useful for internationalization:

```
auto s5 = format("Hello {1} {0}", ival, who);
auto s6 = format("Hola {0} {1}", ival, who);
```

Output:

```
Hello everyone 42
Hola 42 everyone
```

- You can align values, left (**<**), right (**>**), or center (**^**), with or without a fill character:

```
auto s7 = format("{:.<10}", ival);
auto s8 = format("{:.>10}", ival);
auto s9 = format("{:.^10}", ival);
```

Output:

```
  42........
  ........42
  ....42....
```

- You can set the decimal precision of values:

  ```
  auto s10 = format("π: {:.5}", pi);  // π: 3.1416
  ```

  And much, much more.

  It's a rich and complete formatting specification that provides the type-safety of `iostream` with the performance and simplicity of `printf`, for the best of both worlds.

## How it works

The `format` library is the basis of the `print()` function, which was introduced with C++23. The `format()` function itself returns a `string` object.

The `format` library uses a few separate functions to do its work. By way of example, we could create our own `print()` function using components from the `format` library:

```
template<typename... Args>
void print(const string_view fmt_str, Args&&... args) {
    auto fmt_args = std::make_format_args(args...);
    auto outstr = std::vformat(fmt_str, fmt_args);
    fputs(outstr.c_str(), stdout);
}
```

This uses the same arguments as the `format()` function, and it demonstrates the use of `make_format_args()` and `vformat()` to format output. The first argument is a `string_view` object for the format string. This is followed by a variadic parameter pack for the arguments.

The `make_format_args()` function takes the parameter pack and returns an object with *type-erased values* suitable for formatting. This object is then passed to `vformat()`, which returns a `string` suitable for printing. We use `fputs()` from the C standard library to print the formatted string to the console, but you could just as easily store the string in a database or use it for any purpose.

In C++, *type-erased values* are objects that hide their concrete type behind a uniform interface. This allows a class like `std::format` to operate on values of different types without knowing their types at compile time.

## There's more...

It's nice to have the ability to format strings and primitives, but for the `format` library to be fully functional, it needs customization to work with your own classes.

For example, here's a simple `struct` with two members: a *numerator* and *denominator*. We'd like this to print as a fraction:

```
struct Frac {
    long n;
    long d;
};
int main() {
    Frac f {5, 3};
    print("Frac: {}\n", f);
}
```

When I compile this, it leads to a cascade of errors to the effect of, "No user-defined conversion operator...". Okay, so, let's fix it!

When the `format` system encounters an object for *conversion*, it looks for a *specialization* of a `formatter` object with the corresponding type. Standard specializations are included for common objects such as strings and numbers.

It's quite simple to create a specialization for our `Frac` type:

```
template<typename T>
struct Frac {
    T n;
    T d;
};

template <typename T>
struct std::formatter<Frac<T>>: std::formatter<int> {
    template<typename FormatContext>
    auto format(const Frac<T>& o, FormatContext& ctx) const {
        return format_to(ctx.out(), "{}/{}", o.n, o.d);
```

```
    }
};
```

**Note**

In a previous edition of this book the `const` qualifier was omitted from the `format` function overload. This was an error that was tolerated by early implementations of the format library. The `const` qualifier is now required by all major implementations.

The `format()` function is an overload of the `std::format()` function that takes an object of the type we want to specialize for, in this case a `Frac` object, and a `FormatContext` object which is inherited from the `formatter`. This version of `format()` returns an *end iterator*. The `format_to()` function takes that iterator, the format string, and the variadic parameters. In this case, the variadic parameters are the two values from our `Frac` class, the numerator and denominator.

All we need to do is provide a simple format string `"{0}/{1}"` and the numerator and denominator values. The `0` and `1` indicate the position of the parameters. This is not strictly necessary but they could come in handy in other contexts.

Now that we have a specialization for `Frac`, we can pass our object to `print()` to get a readable result:

```
int main() {
    Frac f {5, 3};
    print("Frac f {{5, 3}}: {}\n", f);
}
```

Output:

```
Frac f {5, 3}: 5/3
```

Note the double braces `{{` and `}}` in the format string. You can use double braces to print a single brace without having it interpreted as a formatting parameter by the library.

The C++20 `format` library solves a long-standing problem by providing a type-safe text formatting library that is both efficient and convenient.

## See also...

A good reference for `format` string options is available at:

`https://en.cppreference.com/w/cpp/utility/format/spec`

# Print strings with std::print and std::println

While cout has had its place since the beginning of the STL, it has been outdated for many years. The advent of type-safe variadic templates and parameter packs with C++11 made it possible to replace cout with a more succinct and efficient alternative, like `printf()` but with type safety.

New for C++23, the `print()` and `println()` functions leverage the `format` library to provide efficient string formatting and printing for both console and file streams. This was originally proposed for C++20 but was not ready in time for ratification.

Because the `print()` functions bypass the cumbersome `iostream` library, these functions are far more efficient than using format with `cout`. Performance and code size are comparable to `printf()` but with the type-safety benefits of the `format` library.

The `print()` and `println()` functions are defined in the `<print>` header.

## How to do it

The `print()` and `println()` functions are simple and powerful.

- To use the `print()` and `println()` functions, we need to include the `<print>` header:

```
#include <print>
```

- I also like to set up a `using` declaration to allow use of the `print` and `println` symbols without the `std::` namespace:

```
using std::print, std::println;
```

  It's often considered best practice to keep to one symbol per `using` declaration, but I make an exception when the symbols are so closely related.

- Now we can more easily use these functions in our code:

```
int main() {
    const char* greeting {"earthlings"};
    print("hello, {}!\n", greeting);
}
```

Our output:

```
hello, earthlings!
```

- The `println()` function works just like `print()` but it sends a newline after printing the string:

```
print("one");
print("two");
print("three");
print("\n");
println("one");
println("two");
println("three");
```

Output:

```
onetwothree
one
two
three
```

The `println()` variant is useful in many circumstances where you want to print a line of text at a time.

- Beginning with C++23, we can also use these functions to write to a file stream:

```
const char* greeting {"earthlings"};
const char* filename {"output.txt"};
std::ofstream ofile(filename);
if (ofile.is_open()) {
    println("Writing to {}", filename);
    println(ofile, "hello, {}!", greeting);
    ofile.close();
    println("done.");
} else {
    println(stderr, "failed to open file {}", filename);
}
```

Output:

```
Writing to output.txt
done.
```

Contents of `output.txt`:

```
hello, earthlings!
```

This makes printing to a file just as simple as printing to the console.

**Note**

As of mid-2026, writing to an `ostream` file stream with `print()` or `println()` is supported only on GCC and not yet implemented on Apple Clang or Microsoft Visual C++.

## How it works

The `print()` and `println()` functions leverage the `formatter` class to expand variables and positional arguments to format the output according to the options specified in a *format string*. The function signature is effectively the same as that of `format()`, with the distinction being only the return type:

```
template< class... Args >
std::string format(std::format_string<Args...> fmt, Args&&...args);

template< class... Args >
void print(std::format_string<Args...> fmt, Args&&...args);

template< class... Args >
void println(std::format_string<Args...> fmt, Args&&...args);
```

This means that you may use `print()` and `println()` in the same context as `format()`, eliminating the need for `cout` in most, if not all, circumstances. In other words, where you may have used `cout` with format:

```
cout << format("hello, {}", greeting) << endl;
```

You may now just use `print()` or `println()`:

```
println("hello, {}", greeting);
```

With these functions being part of the standard, you should now favor using `print()` and `println()` wherever you would have previously used `cout`.

## There's more...

It's worth mentioning that any specializations for the formatter work seamlessly with `print()` and `println()`.

For example, the specialization we used in the `format` recipe will also work with `print()` and `println()`:

```
template<typename T>
struct Frac {
    T n;
    T d;
};

template <typename T>
struct std::formatter<Frac<T>>: std::formatter<int> {
    template<typename FormatContext>
    auto format(const Frac<T>& o, FormatContext& ctx) const {
        return format_to(ctx.out(), "{}/{}", o.n, o.d);
    }
};
```

This is the same code we used in the `format` example, and it works just as well with `print()` and `println()`:

```
Frac<int> f {5, 3};
println("Frac f {{5, 3}}: {}", f);
```

Output:

```
Frac f {5, 3}: 5/3
```

This makes it easy to format output for your own classes on the console or in files.

# Concatenate strings

There are several ways to concatenate strings in C++. In this recipe, we look at a few of the most common: the `string` class `+` operator, the `string::append()` method, the `ostringstream` class `<<` operator, and the `format()` function. Each of these has its advantages, disadvantages, and use cases.

## How to do it

In this recipe, we examine ways to concatenate strings. We will then perform some benchmarks and consider the different use cases.

- We'll start with a couple of `std::string` objects:

  ```
  string a {"a"};
  string b {"b"};
  ```

  The `string` objects are constructed from literal C-strings.

  The `std::string` class's C-string constructor makes a copy of the literal string and uses the local copy as the underlying data for the `string` object.

- Now, let's construct a new empty string object and concatenate **a** and **b** with a separator:

  ```
  string x {};
  x += a + ", " + b;
  println("{}", x);
  ```

  Here, we used the `string` object's `+=` and `+` operators to concatenate the `a` and `b` strings, along with literal string `", "`. The resulting string has the elements concatenated together:

  ```
  a, b
  ```

- We could instead use the `append()` method of the `string` class:

  ```
  string x {};
  x.append(a);
  x.append(", ");
  ```

```
x.append(b);
println("{}", x);
```

This gives us the same result:

```
a, b
```

- Or we could construct an `ostringstream` object, which uses the stream interface:

```
ostringstream x {};
x << a << ", " << b;
println("{}", x.str());
```

Note that we use `x.str()` to print the `ostringstream` because format does not have a specialization for iostreams.

We get the same result:

```
a, b
```

- We could also use the `format()` function:

```
string x {};
x = format("{}, {}", a, b);
println("{}", x);
```

Again, we have the same result:

```
a, b
```

- Or, because `println()` uses the format library, we can skip the `format` step altogether and use `println()` directly with no intermediate `string` object:

```
println("{}, {}", a, b);
```

And we get the same output:

```
a, b
```

## How it works

The `string` object has two distinct methods for concatenating a string: the + operator and the `append()` member function.

The `append()` member function adds data to the end of the `string` object's data. It must allocate and manage memory to accomplish this.

The + operator uses the `operator+()` overload to construct a new `string` object with the old and new data and returns the new object.

The `ostringstream` object works like an `ostream` but stores its output in a `string` object. The `<<` operator returns a `basic_ostream` object.

The C++20 `format()` function uses a format string with variadic arguments and returns a newly constructed `string` object.

It's worth noting here that the `<print>` header also includes the `<format>` header, so we need only `#include <print>` which gets us all the code in both headers.

## There's more...

How do you decide which concatenation strategy is right for your code? We can start with some benchmarks.

### Benchmarks

I performed these tests using GCC 15.2 with `-O2` optimization on a VM running Debian Linux 6.17. Your results will, of course, be different.

- First, we'll create a `timer` function using the `<chrono>` library:

```
void timer(string(*f)()) {
    auto t1 = high_resolution_clock::now();
    string s {f()};
    auto t2 = high_resolution_clock::now();
    duration<double, std::milli> ms = t2 - t1;
    println("{}", s);
    println("duration: {} ms", ms.count());
}
```

  The `timer` function calls the function passed to it, marking the time before and after the function call. It then displays the duration using `cout`.

- Now, we create a function that concatenates strings, using the `append()` member function:

```
string append_string() {
    println("== append_string:");
    string a {"a"};
    string b {"b"};
    long n {};
    while (++n) {
        string x {};
        x.append(a);
        x.append(", ");
        x.append(b);
        if(n >= 10000000) return x;
    }
    return "error";
}
```

For benchmarking purposes, this function repeats the concatenation 10 million times. We call this function from `main()` with `timer()`:

```
int main() {
    timer(append_string);
}
```

We get this output:

```
  == append_string:
  a, b
  duration: 74.882582 ms
```

So, on this system, our concatenation ran 10 million iterations in about 75 milliseconds.

- Now, let's create the same function with the + operator overload:

```
string concat_string() {
    println("== concat_string:");
    string a {"a"};
    string b {"b"};
    long n {};
    while (++n) {
```

```
        string x {};
        x += a + ", " + b;
        if(n >= 10000000) return x;
    }
    return "error";
}
```

Our benchmark output:

```
== concat_string:
a, b
duration: 174.573135 ms
```

This version performed 10 million iterations in about 175 milliseconds.

- Now, let's try it with `ostringstream`:

```
string concat_ostringstream() {
    println("== ostringstream:");
    string a {"a"};
    string b {"b"};
    long n {};
    while (++n) {
        ostringstream x {};
        x << a << ", " << b;
        if(n >= 10000000) return x.str();
    }
    return "error";
}
```

Our benchmark output:

```
== ostringstream:
a, b
duration: 1434.781101 ms
```

This version ran 10 million iterations in about 1.4 seconds.

- Finally, here's the `format()` version:

```
string concat_format() {
    println("== append_format:");
```

```
    string a {"a"};
    string b {"b"};
    long n {};
    while (++n) {
        string x {};
        x = format("{}, {}", a, b);
        if(n >= 10000000) return x;
    }
    return "error";
}
```

Our benchmark output:

```
== append_format:
a, b
duration: 929.482062 ms
```

The `format()` version ran 10 million iterations in about 930 milliseconds.

Summary of the results:

| Concatenation Method | Benchmark in milliseconds |
|---|---|
| `append()` | 75 ms |
| `operator+()` | 175 ms |
| `ostringstream` | 1,435 ms |
| `format()` | 929 ms |

*Table 7.1: A comparison of concatenation performance*

## Why the performance discrepancy?

We can see from these benchmarks that the `ostringstream` version takes many times longer than the `string`-based versions.

The `append()` method is fastest. It must allocate memory but does not construct new objects. Some optimizations may be possible due to repetition.

The `+` operator overload calls the `append()` method. The additional function call makes it incrementally slower than the `append()` method.

The `ostringstream` operator << creates a new `ostream` object for each operation. With the complexity of the stream object, along with managing the stream state, this makes it much slower than any of the `string`-based versions.

The `format()` version creates one new `string` object but without the overhead of the `iostream` system.

**Note**

These benchmark results will vary widely on different processors, operating systems, and library implementations. For example, append() tested closer to operator+() on the latest Apple Clang and even slower on MSVC. If performance is important to your application you will want to run your own benchmarks on your target system.

### Why would I choose one over another?

Some measure of personal preference will be involved. The operator overloads (+ or <<) can be convenient. Performance may or may not be an issue for you.

The `ostringstream` class specializes the << operator for each different type, so it's able to operate in circumstances where you may have different types calling the same code.

The `format()` function offers the same type-safety and customization options as the `ostringstream` class, yet is significantly faster.

The `string` object's + operator is fast, easy to use, and easy to read, but is incrementally slower than `append()`.

The `append()` version is fastest but requires a separate function call for each item.

For many purposes, I prefer the `format()` function or the `string` object's + operator. I'll use `append()` if every bit of speed matters.

## Transform strings

The `std::string` class is a *contiguous container*, much like `std::vector` or `std::array`. It supports the `contiguous_iterator` concept and all corresponding algorithms.

The `string` class is a specialization of `basic_string` with a `char` type. This means that the elements of the container are of type `char`. Other specializations are available, but `string` is most common.

Because it is fundamentally a contiguous container of `char` elements, `string` may be used with the `transform()` algorithm, or any other technique that operates on a `contiguous_iterator` concept.

## How to do it

There are several ways to perform transformations, depending on the application. This recipe will explore a few of them.

- We'll start with a few predicate functions. A predicate function takes a transformation element and returns a related element. For example, here is a simple predicate that returns an uppercase character:

```
const char char_upper(const char c) {
    if (c >= 'a'&&c<= 'z') return c - ('a' - 'A');
    else return c;
};
```

  This function depends on the arithmetic relationship between uppercase and lowercase letters in the ASCII code set.

  We could use the `std::toupper()` function but it returns an `int` and would require a *cast* to work with the `char` elements in a `string`. When working in ASCII, I prefer to use this simple function instead.

  Here is the corresponding `char_lower()` function:

```
const char char_lower(const char c) {
    if (c >= 'A'&&c<= 'Z') return c + ('a' - 'A');
    else return c;
};
```

- The `rot13()` function is a fun transformation predicate for demonstration purposes. It's a simple substitution cypher, *not suitable for encryption* but commonly used for *obfuscation*:

```
const char rot13s(const char& x) {
    auto rot13c =
      [](const char x, const char a) -> const char {
        return a + (x - a + 13) % 26;
    };
    if (x >= 'A'&&x<= 'Z') return rot13c(x, 'A');
```

```
    if (x >= 'a'&&x<= 'z') return rot13c(x, 'a');
    return x;
}
```

This uses a lambda expression to encapsulate the rotation itself. The rotation is accomplished with a simple modulus operation.

- We can use these predicates with the `transform()` algorithm:

```
int main() {
    string s {"hello jimi"};
    println("{}", s);

    ranges::transform(s, s.begin(), char_upper);
    println("{}", s);
    ...
```

I chose the `ranges::transform()` function for its simplicity. We pass the `char_upper()` predicate which applies the transformation to each element of `s` and puts the result back in `s`:

Output:

```
hello jimi
HELLO JIMI
```

- Instead of `transform()`, we could use a simple `for` loop with the predicate function:

```
for (auto& c : s) c = rot13(c);
println("{}", s);
```

Starting with our uppercase string object, the result is:

```
URYYB WVZV
```

- The fun thing about the `rot13` cypher is that it unscrambles itself. Because there are 26 letters in the *ASCII* alphabet, rotating 13 and then rotating 13 again results in the original string. Let's transform to lowercase and `rot13` again to restore our string:

```
for (auto& c : s) c = rot13(char_lower(c));
println("{}", s);
```

Output:

```
hello jimi
```

Because of their uniform interface, the predicate functions may be *chained* as parameters of each other. We could also use `char_lower(rot13s(c))` with the same result.

- If your requirement is too complex for a simple character-by-character transformation, you may use `string` iterators as you would with any contiguous container. Here's a simple function that transforms a lowercase string to *Title Case* by capitalizing the first character and every character that follows a space:

```
const string& title_case(string& s) {
    auto begin = s.begin();
    auto end = s.end();
    *begin++ = char_upper(*begin);  // first element
    bool space_flag {false};
    for (auto it {begin}; it != end; ++it) {
        if (*it == ' ') {
            space_flag = true;
        } else {
            if (space_flag) *it = char_upper(*it);
            space_flag = false;
        }
    }
    return s;
}
```

Because it returns a reference to the transformed string, we can call it with `print()`, like this:

```
println("{}", title_case(s));
```

Output:

```
Hello Jimi
```

## How it works

The `std::basic_string` class, and its specializations (including `string`), are supported by iterators fully compliant with `contiguous_iterator`. This means that any technique that works with any contiguous container also works with `string`.

**Note**

1. These transformations will not work with `string_view` objects because the underlying data is `const`-qualified.

2. For some legacy purposes, including older Windows APIs, the `std::wstring` class has the same capabilities but uses wide characters. Keep in mind that the ASCII-based transformations shown in this recipe may not work the same with other character encodings.

# Trim whitespace from strings

It is common for input from users to include extraneous whitespace characters at one or both ends of a string. This can be problematic, so we often need to remove them. In this recipe, we'll use the `string` class methods, `find_first_not_of()` and `find_last_not_of()`, to trim whitespace from the ends of a string.

## How to do it

The `std::string` class includes methods for finding elements that are, or are not, included in a list of characters. We'll use these methods to trim whitespace characters from a `string`:

- We start by defining `string` with input from a hypothetical ten-thumbed user:

```
int main() {
    string s {" \t  ten-thumbed input   \t   \n \t "};
    println("[{}]", s);
    ...
```

Our input has a few extra tab `\t` and newline `\n` characters before and after the content. We print it with surrounding brackets to show the whitespace:

```
[       ten-thumbed input
      ]
```

- Here's a `trimstr()` function to remove all the whitespace characters from both ends of a `string`:

```
string trimstr(const string& s) {
    constexpr const char * whitespace {" \t\r\n\v\f"};
    if (s.empty()) return s;
    const auto first = s.find_first_not_of(whitespace);
    if (first == string::npos) return {};
    const auto last = s.find_last_not_of(whitespace);
    return s.substr(first, (last - first + 1));
}
```

We define our set of whitespace characters as *space*, *tab*, *return*, *newline*, *vertical tab*, and *form feed*. While some of these are more common than others, that's the canonical set.

This function uses the `find_first_not_of()` and `find_last_not_of()` methods of the `string` class to find the first and last elements that are *not* a member of the set.

- Now, we can call the function to get rid of all that unsolicited whitespace:

```
s = trimstr(s);
println("[{}]", s);
```

Output:

```
  [ten-thumbed input]
```

## How it works

The `string` class's various `find...()` member functions return a position as a `size_t` value:

```
size_t find_first_not_of( const CharT* s, size_type pos = 0 );
size_t find_last_not_of( const CharT* s, size_type pos = 0 );
```

The return value is the zero-based position of the first matching character (*not* in the `s` list of characters) or the special value, `string::npos`, if not found. `npos` is a static member constant that represents an invalid position.

We test for `(first == string::npos)` and return an empty string `{}` if there is no match. Otherwise, we use the `first` and `last` positions with the `s.substr()` method to return the string without whitespace.

# Read strings from user input

The STL provides character-based input from the standard input stream using the `std::cin` object. The `cin` object is a global *singleton* that reads input from the console as an `istream` input stream.

By default, `cin` reads *one word at a time* until it reaches the end of the stream:

```
string word{};
print("Enter words: ");
while(cin >> word) {
    print("[{}] ", word);
}
print ("\n");
```

Output:

```
$ ./working
Enter words: big light in sky
[big] [light] [in] [sky]
```

This is of limited usefulness and it may lead some to dismiss `cin` as minimally functional. While `cin` definitely has its quirks, it can be wrangled into providing line-oriented input.

## How to do it

To get basic line-oriented functionality from `cin`, there are two significant behaviors that need to be understood. One is the ability to get a line at a time, instead of a word at a time. The other is the ability to reset the stream after an error condition. Let's look at these in some detail:

- First, we need to prompt the user for input. Here's a simple prompt function:

```
bool prompt(const string_view s,
  const string_view s2 = "") {
    if (s2.size()) print("{} ({}): ", s, s2);
    else print("{}: ", s);
    fflush(stdout);
    return true;
}
```

The `fflush()` function is part of the C Standard. This ensures that the output is written to the display or output file immediately. This is necessary because the `std::print()` function does not guarantee that the output is flushed.

- The `cin` class has a `getline()` method that gets a line of text from the input stream and puts it in a C-string array:

```
constexpr size_t MAXLINE {1024 * 10};
char s[MAXLINE]{};
const char * p1 {"Words here"};
prompt(p1);
cin.getline(s, MAXLINE, '\n');
println("{}", s);
```

Output:

```
$ ./working
Words here: big light in sky
big light in sky
```

The `cin.getline()` method takes three arguments:

```
getline(char* s, size_t count, char delim );
```

The first argument is a C-string buffer for the destination. The second is the size of the C-string buffer. The third is the delimiter for the end of the line.

The function will not put more than `count – 1` characters in the buffer, leaving room for a *null terminator*.

The delimiter defaults to the newline `'\n'` character. I have specified it here for completeness.

- The STL also provides a stand-alone `getline()` function that works with an STL `string` object:

```
string line{};
const char * p1a{ "More words here" };
prompt(p1a, "p1a");
getline(cin, line, '\n');
println("{}", line);
```

Output:

```
$ ./working
More words here (p1a): slated to appear in east
slated to appear in east
```

The stand-alone `std::getline()` function takes three arguments:

```
getline(basic_istream&∈,string&str,chardelim);
```

The first argument is the input stream. The second is a reference to a `string` object. And the third is the end-of-line delimiter.

If not specified, the delimiter defaults to the newline `'\n'` character.

I usually find the standalone `getline()` more convenient than the `cin.getline()` method.

- We can use `cin` to get a specific type from the input stream. To do this, we must be able to handle an error condition.
  When `cin` encounters an error, it sets the stream to an error condition and stops accepting input. To retry input after an error, we must reset the state of the stream. Here's a function that resets the input stream after an error:

```
void clearistream() {
    int c {};
    cin.clear();
    while (c != '\n'&&c!=EOF)c=getchar();
}
```

The `cin.clear()` function resets the error flags on the input stream but leaves text in the buffer. We then clear the buffer by reading a line and discarding it. The C Standard `getchar()` function is the most reliable way to clear the input buffer.

- We can accept numeric input by using `cin` with numeric type variables:

```
double a {};
double b {};
for (prompt(p2); !(cin >> a >> b); prompt(p2)) {
    println("not numeric");
    clearistream();
}
```

```
println("You entered {} and {}", a, b);
clearistream();
```

Output:

```
$ ./working
Please enter two numbers: a b
not numeric
Please enter two numbers: 47 73
You entered 47 and 73
```

The `cin >> a >> b` expression accepts input from the console and attempts to convert the first two words to types compatible with `a` and `b` (`double`). If it fails, we call `clearistream()` and try again. We call `clearistream()` after the loop to ensure the input buffer is cleared.

- We can use the `getline()` separator parameter to get comma-separated input:

```
line = "";
prompt(p3);
while(line.empty()) getline(cin, line);
stringstream ss(line);
while(getline(ss, word, ',')) {
    if(word.empty()) continue;
    println("word: [{}]", trimstr(word));
}
```

Output:

```
$ ./working
Comma-separated words: this, that, other
word: [this]
word: [that]
word: [other]
```

Because this code runs after the numbers code, and because `cin` is messy, there may still be a line ending in the buffer. The `while(line.empty())` loop will optionally eat any empty lines.

We use a `stringstream` object to process the words, so we don't have to do it with `cin`. This allows us to use `getline()` to get one line without waiting for the end-of-file state.

Then, we call `getline()` on the `stringstream` object to parse out words separated by commas. This gives us words but with leading and/or trailing whitespace. We use the `trimstr()` function from the *Trim whitespace from strings* recipe in this chapter to trim the whitespace.

## How it works

The `std::cin` class is more useful than it may appear, but it can be a challenge to use. It tends to leave line endings on the stream, and in the case of errors, it can end up ignoring input.

The solution is to use `getline()` and, when necessary, put the line into a `stringstream` for convenient parsing.

# Count words in a file

By default, the `basic_istream` class reads one word at a time. We can take advantage of this property to use an `istream_iterator` to count words.

## How to do it

This is a simple recipe to count words using an `istream_iterator`:

- We'll start with a function to count words using an `istream_iterator` object:

```
size_t wordcount(auto& is) {
    using it_t = std::istream_iterator<string>;
    return std::distance(it_t(is), it_t());
}
```

The `distance()` function takes two iterators and returns the number of steps between them. The `using` statement creates an alias `it_t` for the `istream_iterator` class with a `string` specialization. We then call `distance()` with an iterator, initialized with the input stream `it_t(is)`, and another with the default constructor, which gives us an end-of-stream sentinel.

- We call `wordcount()` from `main()`:

```
int main() {
    const char* fn {"the-raven.txt"};
```

```
    auto infile = std::ifstream(fn, std::ios_base::in);
    println("There are {} words in the file.",
      wordcount(infile));
}
```

This calls `wordcount()` with our `fstream` object and prints the number of words in the file. Using the text of Edgar Allan Poe's *The Raven*, we get this output:

```
There are 1068 words in the file.
```

## How it works

Because `basic_istream` defaults to word-by-word input, the number of iterations in a file will be the number of words. We use the `distance()` function to measure the number of steps between the two iterators. The distance between the beginning iterator and the sentinel iterator resolves to the number of words in the file.

# Initialize complex structures from file input

One strength of the *input stream* is its ability to parse different types of data from a text file and convert them to their corresponding fundamental types. Here's a simple technique for importing data into a container of structures using an input stream.

## How to do it

In this recipe we'll take a data file and import its disparate fields into a vector of structs. The data file represents cities with their populations and map coordinates.

- This is `cities.txt`, the data file we'll read:

```
Las Vegas
661903 36.1699 -115.1398
New York City
8850000 40.7128 -74.0060
Berlin
3571000 52.5200 13.4050
Mexico City
21900000 19.4326 -99.1332
Sidney
5312000 -33.8688 151.2093
```

The city name is on a line by itself. The second line is population, followed by longitude and latitude. This pattern repeats for each of the five cities.

- We'll define our filename in a constant so we can open it later:

```
constexpr const char * fn {"cities.txt"};
```

Here's a `City` struct to hold the data:

```
struct City {
    string name;
    unsigned long population;
    double latitude;
    double longitude;
};
```

- We would like to read the file and populate a vector of `City` objects:

```
vector<City> cities;
```

Here's where the input stream makes this easy. We can simply overload the `operator>>` for `istream` like this:

```
std::istream&operator>>(std::istream& in, City& c) {
    in >> std::ws;
    std::getline(in, c.name);
    in >> c.population >> c.latitude >> c.longitude;
    return in;
}
```

The `std::ws` input manipulator discards leading whitespace from the input stream.

We use `getline()` to read the city name, as it could be one or more words.

This leverages the `>>` operator for the population (`unsigned long`), and latitude and longitude (both `double`) elements to populate the correct type.

Now we can open the file and use the `>>` operator to read the file directly into the vector of City:

```
ifstream infile(fn, std::ios_base::in);
if(!infile.is_open()) {
```

```
        println("failed to open file {}", fn);
        return 1;
    }
    for(City c{}; infile >> c;) cities.emplace_back(c);
```

- We can display the `vector` using `println()`:

```
for (const auto& [name, pop, lat, lon] : cities) {
    println("{:.<15} pop {:<10} coords {}, {} ",
        name, make_commas(pop), lat, lon);
}
```

Output:

```
$ ./initialize_container < cities.txt
Las Vegas...... pop 661,903    coords 36.1699, -115.1398
New York City.. pop 8,850,000  coords 40.7128, -74.006
Berlin......... pop 3,571,000  coords 52.52, 13.405
Mexico City.... pop 21,900,000 coords 19.4326, -99.1332
Sidney......... pop 5,312,000  coords -33.8688, 151.2093
```

The `make_commas()` function is from *Chapter 2* of this book:

```
string make_commas(const uint64_t num) {
    auto s = std::to_string(num);
    for (auto i = s.size(); i > 3; i -= 3) {
        s.insert(i - 3, ",");
    }
    return s;
}
```

## How it works

The heart of this recipe is the `istream` class `operator>>` overload:

```
std::istream&operator>>(std::istream& in, City& c) {
    in >> std::ws;
    std::getline(in, c.name);
    in >> c.population >> c.latitude >> c.longitude;
    return in;
}
```

By specifying our `City` class in the function header, this function will be called every time a `City` object appears on the right-hand side of an input stream >> operator:

```
City c{};
infile >> c;
```

This allows us to specify exactly how the input stream reads data into a `City` object.

## There's more...

When you run this code on a Windows system, you'll notice that the first word of the first line gets corrupted. That's because Windows always includes a *byte order mark* (BOM) at the head of any UTF-8 file. So, when you read a file on Windows, the BOM will be included in the first object you read. The BOM is anachronistic, but as of this writing there is no way to stop Windows from employing it.

The solution is to call a function that checks the first three bytes of a file for the BOM. The BOM for UTF-8 is EF BB BF. Here's a function that searches for, and skips, a UTF-8 BOM:

```
// skip BOM for UTF-8 on Windows
void skip_bom(auto& fs) {
    const unsigned char boms[]{ 0xef, 0xbb, 0xbf };
    bool have_bom{ true };
    for(const auto& c : boms) {
        if((unsigned char)fs.get() != c) have_bom = false;
    }
    if(!have_bom) fs.seekg(0);
    return;
}
```

This reads the first three bytes of the file and checks them for the UTF-8 BOM signature. If any of the three bytes do not match, it resets the input stream to the beginning of the file. If the file has no BOM, there's no harm done.

You simply call this function before you begin reading from a file:

```
int main() {
    ...
    ifstream infile(fn, std::ios_base::in);
    if(!infile.is_open()) {
        cout << format("failed to open file {}\n", fn);
        return 1;
```

```
    }
    skip_bom(infile);
    for(City c{}; infile >> c;) cities.emplace_back(c);
    ...
}
```

This will ensure that the BOM is not included in the first string of the file.

> **Note**
>
> Because the `cin` input stream is not seekable, the `skip_bom()` function will not work on the `cin` stream. It will only work with a seekable text file.

# Customize a string class with char_traits

The string class is an alias of the `basic_string` class, with the signature:

```
class basic_string<char, std::char_traits<char>>;
```

The first template parameter provides the type of a character. The second template parameter provides a character traits class, which provides basic character and string operations for the specified character type. We normally use the default char_traits class.

We can modify the behavior of a string by providing our own custom character traits class.

## How to do it

In this recipe we will create a *character traits class* for use with `basic_string` that will ignore case for comparison purposes.

- First, we'll need a function to convert characters to a common case. We'll use lowercase here, but it's an arbitrary choice. Uppercase would work just as well:

  ```
  constexpr char char_lower(const char& c) {
      if (c >= 'A'&&c<= 'Z') return c + ('a' - 'A');
      else return c;
  }
  ```

  This function must be constexpr (for C++20 and later), so the existing `std::tolower()` function won't work here. Fortunately, it's a simple solution to a simple problem.

- Our traits class is called `ci_traits` (*ci* stands for case-independent). It inherits from `std::char_traits<char>`:

```
class ci_traits : public std::char_traits<char> {
public:
    ...
};
```

The inheritance allows us to override only the functions that we need.

- The comparison functions are called `lt()`, for less-than, and `eq()`, for equal-to:

```
static constexpr bool lt(char_type a, char_type b) noexcept {
    return char_lower(a) < char_lower(b);
}
static constexpr bool eq(char_type a, char_type b) noexcept {
    return char_lower(a) == char_lower(b);
}
```

Notice that we compare using the *lowercase* versions of characters.

- There's also a `compare()` function, that compares two C-strings. It returns `+1` for greater-than, `-1` for less-than, and `0` for equal-to. We can use the spaceship `<=>` operator for this:

```
static constexpr int compare(const char_type* s1,
        const char_type* s2, size_t count) {
    for (size_t i {0}; i < count; ++i) {
        auto diff { char_lower(s1[i]) <=> char_lower(s2[i]) };
        if(diff > 0) return 1;
        if(diff < 0) return -1;
        }
    return 0;
}
```

- Finally, we need to implement a `find()` function. This returns a pointer to the first instance of a found character, or `nullptr` if not found:

```
static constexpr const char_type* find(const char_type* p,
        size_t count, const char_type& ch) {
    const char_type find_c{ char_lower(ch) };
```

```
    for (size_t i {0}; i < count; ++i) {
        if(find_c == char_lower(p[i])) return p + i;
    }
    return nullptr;
}
```

- Now that we have a `ci_traits` class, we can define an alias for our string class:

```
using ci_string = std::basic_string<char, ci_traits>;
In our main() function, we define a string and a ci_string:
int main() {
    string s {"Foo Bar Baz"};
    ci_string ci_s {"Foo Bar Baz"};
    ...
```

- Let's compare two ci_string objects with different cases:

```
ci_string compare1{"CoMpArE StRiNg"};
ci_string compare2{"compare string"};
if (compare1 == compare2) {
    println("Match! {} == {}", compare1, compare2);
} else {
    println("no match {} != {}", compare1, compare2);
}
```

Output:

```
Match! CoMpArE StRiNg == compare string
```

The comparison works as expected.

- Using the `find()` function on the `ci_s` object, we search for a lowercase b and find an uppercase B:

```
size_t found = ci_s.find('b');
cout << format("found: pos {} char {}\n", found, ci_s[found]);
```

Output:

```
found: pos 4 char B
```

## How it works

This recipe works by replacing the `std::char_traits` class in the template specialization of the string class with a `ci_traits` class of our own. The `basic_string` class uses the traits class for its fundamental character-specific functions, like comparisons and searching. When we replace it with our own class, we can change these fundamental behaviors.

## There's more...

We can also override the `assign()` and `copy()` member functions to create a class that stores lowercase characters:

```
class lc_traits : public std::char_traits<char> {
public:
    static constexpr void assign( char_type& r,
            const char_type& a ) noexcept {
        r = char_lower(a);
    }
    static constexpr char_type* assign( char_type* p,
            std::size_t count, char_type a ) {
        for (size_t i{}; i < count; ++i) p[i] = char_lower(a);
        return p;
    }
    static constexpr char_type* copy(char_type* dest,
            const char_type* src, size_t count) {
        for (size_t i{0}; i < count; ++i) {
            dest[i] = char_lower(src[i]);
        }
        return dest;
    }
};
```

Now we can create an `lc_string` alias, and the object stores lowercase characters:

```
using lc_string = std::basic_string<char, lc_traits>;
...
lc_string lc_s{"Foo Bar Baz"};
println("lc_string: {}", lc_s);
```

Output:

```
lc_string: foo bar baz
```

# Parse strings with regular expressions

*Regular Expressions* (commonly abbreviated as *regex*) are often used for lexical analysis and pattern-matching on streams of text. They are common in Unix text processing utilities, such as `grep`, `awk`, and `sed`, and are an integral part of the *Perl* language. There are a few common variations on the syntax. A POSIX standard was approved in 1992, while other common variations include *Perl* and *ECMAscript* (JavaScript) dialects. The C++ `regex` library defaults to the *ECMAscript* dialect.

The `regex` library was first introduced to the STL with C++11. It can be very useful for finding patterns in text files.

To learn more about Regular Expression syntax and usage, I recommend the book, *Mastering Regular Expressions*, by Jeffrey Friedl.

## How to do it

For this recipe we will extract hyperlinks from an HTML file. A hyperlink is coded in HTML like this:

```
<a href="http://example.com/file.html">Text goes here</a>
```

We will use a `regex` object to extract both the link and the text, as two separate strings.

- Our example file is called `"the-end.html"`. It's taken from my web site (`https://old.bw.org/end/`), and included in the *GitHub* repository:

  ```
  const char * fn {"the-end.html"};
  ```

- Now we define our regex object with a regular expression string:

  ```
  const std::regex
      link_re {"<a href=\"([^\"]*)\"[^<]*>([^<]*)</a>"};
  ```

  Regular expressions can look intimidating at first, but actually they're rather simple.

  This is parsed as follows:

  - Match the whole string

    - Find the substring `<a href="`
    - Store everything up to the next " as sub-match 1
    - Skip past the > character
    - Store everything up to the string `</a>` as sub-match 2

- Now we read our file entirely into a string:

```
string in{};
std::ifstream infile(fn, std::ios_base::in);
for (string line {}; getline(infile, line);) in += line;
```

  This opens the HTML file, reads it line-by-line, and appends each line to the `string` object `in`.

- To extract the link strings, we set up a `sregex_token_iterator` to step through the file and extract each of the matched elements:

```
std::sregex_token_iterator it {in.begin(), in.end(),
    link_re, {1, 2}};
```

  The 1 and 2 correspond to the sub-matches in the regular expression.

- We have a corresponding function to step through the results with the iterator:

```
template<typename It>
void get_links(It it) {
    for (It end_it {}; it != end_it; ) {
        const string link {*it++};
        if (it == end_it) break;
        const string desc {*it++};
        cout << format("{:.<24} {}\n", desc, link);
    }
}
```

  We call the function with the `regex` iterator:

```
get_links(it);
```

And we get this result with our descriptions and links:

```
Bill Weinman............ https://bw.org/
courses................. https://bw.org/courses/
music................... https://bw.org/music/
books................... https://packt.com/
back to the internet.... https://duckduckgo.com/
```

## How it works

The STL regex engine operates as a *generator* that evaluates and yields one result at a time. We set up the iterator using `sregex_iterator` or `sregex_token_iterator`. `sregex_token_iterator` supports sub-matches. `sregex_iterator` does not.

The parentheses in our regex serve as *sub-matches*, numbered 1 and 2 respectively.

```
const std::regex link_re {"<a href=\"([^\"]*)\"[^<]*>([^<]*)</a>"}
```

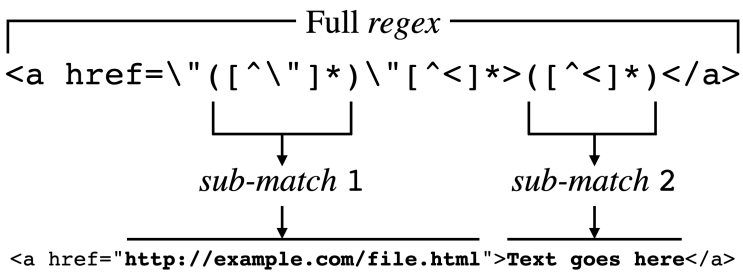



*Figure 7.1: Regular expression with sub-matches*

This allows us to match a `string` and use parts of that `string` as our results:

```
sregex_token_iterator it {in.begin(), in.end(), link_re, {1, 2}};
```

The sub-matches are numbered beginning with `1`. Sub-match `0` is a special value that represents the entire match.

Once we have our iterator, we use it as we would any iterator:

```
for (It end_it {}; it != end_it; ) {
    const string link {*it++};
    if (it == end_it) break;
    const string desc {*it++};
```

```
    println("{:.<24} {} ", desc, link);
}
```

This simply steps through our results via the regex iterator giving us the formatted output:

```
Bill Weinman..... https://bw.org/
courses.......... https://bw.org/courses/
music............ https://bw.org/music/
books............ https://packt.com/
back to the internet.... https://duckduckgo.com/
```


## Subscribe to Deep Engineering

Join thousands of developers and architects who want to understand how software is changing, deepen their expertise, and build systems that last.

Deep Engineering is a weekly expert-led newsletter for experienced practitioners, featuring original analysis, technical interviews, and curated insights on architecture, system design, and modern programming practice.

Scan the QR or visit the link to subscribe for free.



https://packt.link/deep-engineering-newsletter

# 8

# Utility Classes

The C++ Standard Library includes an assortment of utility classes designed for specific tasks. You've probably seen many of these classes in other recipes in this book.

This chapter covers a broad range of utilities, including time measurement, generic types, smart pointers, and more, in the following recipes:

- Manage optional values with `std::optional`
- Use `std::any` for type safety
- Store different types with `std::variant`
- Time events with `std::chrono`
- Use fold expressions for variadic tuples
- Manage allocated memory with `std::unique_ptr`
- Share objects with `std::shared_ptr`
- Use weak pointers with shared objects
- Share object members with aliased smart pointers
- Compare random number engines
- Compare random number distribution generators

## Manage optional values with std::optional

Introduced with C++17, the `std::optional` class holds an *optional value*.

There are any number of scenarios where this can be useful, including long-running processes where a result may not yet be available, API callbacks where a function may or may not return a

value, lazy initialization where an object is constructed only when needed, or any use case where a value may or may not be valid.

Consider the case where you have a function that may or may not return a value. For example, a function that checks if a number is *prime*, but returns the first factor if there is one. This function should return either a value or a `bool` status. We could create a struct that carries both value and status:

```
struct factor_t {
    bool is_prime;
    long factor;
};

factor_t factor(long n) {
    factor_t r{};
    for(long i = 2; i <= n / 2; ++i) {
        if (n % i == 0) {
            r.is_prime = false;
            r.factor = i;
            return r;
        }
    }
    r.is_prime = true;
    return r;
}
```

It's a clumsy solution, but it works and it's not uncommon.

This is made much simpler with the optional class:

```
optional<long> factor(long n) {
    for (long i = 2; i <= n / 2; ++i) {
        if (n % i == 0) return {i};
    }
    return {};
}
```

Now we just return a value, or a *not* value.

We can call it like this:

```
long a {42};
long b {73};
auto x = factor(a);
auto y = factor(b);
if (x) println("lowest factor of {} is {}", *a, *x);
else println("{} is prime", *a);
if (y) println("lowest factor of {} is {}", *b, *y);
else println("{} is prime", *b);
```

Our output is:

```
lowest factor of 42 is 2
73 is prime
```

The optional class allows us to easily return both status and value.

## How to do it

In this recipe we'll look at some examples of how to use the `optional` class:

- The optional class is quite simple. We construct an optional value using standard template notation:

  ```
  optional<int> a {42};
  println("{}", *a);
  ```

  We access the value of the optional with the pointer dereference operator *.

  Output:

  ```
    42
  ```

- Test if the optional has a value using its bool operator:

  ```
  if (a) println("{}", *a);
  else println("no value ");
  ```

  If a were constructed without a value:

  ```
  optional<int> a {};
  ```

The output would reflect the else condition:

```
no value
```

- We can further simplify this by declaring a *type alias*:

```
using oint = std::optional<int>;
oint a {42};
oint b {73};
```

If we want to operate on `oint` objects, with `oint` objects as the result, we can provide operator overloads:

```
oint operator+(const oint& a, const oint& b) {
    if (a && b) return *a + *b;
    return {};
}
```

It may also be useful to provide an overload for the `optional` operator with a non-`optional` operand:

```
oint operator+(const oint& a, const int b) {
    if (a) return *a + b;
    return {};
}
```

This same technique would apply for any of the common arithmetic operators.

Now we can operate on the `oint` objects directly:

```
auto sum {a + b};
if (sum) {
    println("{} + {} = {}", *a, *b, *sum);
} else {
    println("no value");
}
```

Output:

```
42 + 73 = 115
```

Notice that when we declare b with the default constructor:

```
oint b {};
```

The result is now an optional with no value. The `if(sum)` test returns `false`, and we get the else branch output:

```
  no value
```

## How it works

The `std::optional` class is made for simplicity. It provides operator overloads for many common functions. It also includes member functions for further flexibility.

The `optional` class provides an `operator bool` overload for determining if the object has a value:

```
optional<int> n{ 42 };
if (n) ... // has a value
```

Or you may use the `has_value()` member function:

```
if (n.has_value()) ... // has a value
```

To access the value, you may use the `operator*` overload:

```
x = *n;  // * retruns the value
```

Or you may use the `value()` member function:

```
x = n.value();  // * retruns the value
```

The `reset()` member function destroys the value and resets the state of the optional object.

```
n.reset();      // no longer has a value
```

## There's more...

The optional class provides exception support with the `value()` method:

```
b.reset();
try {
```

```
    println("{}", b.value());
} catch(const std::bad_optional_access& e) {
    println("b.value(): {}", e.what());
}
```

Output:

```
b.value(): bad optional access
```

**Note**

Only the `value()` method throws an exception. The behavior of the * operator is *undefined* for an invalid value.

## Use std::any for type safety

Introduced with C++17, the `std::any` class provides a type-safe container for a single object of any type.

For example, this is a default-constructed any object:

```
any x {};
```

This object has no value. We can test that with the `has_value()` method:

```
if (x.has_value()) println("have value ");
else println("no value");
```

Output:

```
no value
```

We assign a value to the any object with the assignment operator:

```
x = 42;
```

Now the any object has a value and a type:

```
if (x.has_value()) {
    println("x has type: {}", x.type().name());
    println("x has value: {}", any_cast<int>(x));
```

```
} else {
    println("no value");;
}
```

Output:

```
x has type: i
x has value: 42
```

The `type()` method returns a `type_info` object. The `type_info::name()` method returns an implementation-defined name for the type in a C-string. In this case, for GCC, the `i` means `int`.

We use the non-member function `any_cast<`*type*`>()` to cast the value for use.

We can re-assign the `any` object with different values of different types:

```
using namespace std::literals;
x = "abc"s;
println("x is type {} with value {}",
    x.type().name(), any_cast<string>(x));
```

Notice the use of the suffix after the string `"abc"s`. This uses the `std::literals` namespace to create a `string` object from a literal C-string.

Output:

```
x is type NSt7__cxx1112basic_string... with value abc
```

I've abbreviated the long type-name from GCC but you get the idea. The same `any` object that once held an `int` now contains an STL string object.

The main usefulness of the `any` class is in creating a *polymorphic* function. Let's examine how to do that in this recipe:

## How to do it

In this recipe we'll build a polymorphic function using the `any` class. A polymorphic function is one that can take objects of different types in its parameters.

- Our polymorphic function takes an `any` object and prints its type and value:

```
void p_any(const any& a) {
    if (!a.has_value()) {
```

```
        println("None.");
    } else if (a.type() == typeid(int)) {
        println("int: {}", any_cast<int>(a));
    } else if (a.type() == typeid(string)) {
        println("string: \"{}\"",
          any_cast<const string&>(a));
    } else if (a.type() == typeid(list<int>)) {
        print("list<int>: ");
        for (auto& i : any_cast<const list<int>&>(a)) print("{} ",
             i);
        print("\n");
    } else {
        println("something else: {}", a.type().name());
    }
}
```

The `p_any()` function first tests to see if the object has a value. It then tests the `type()` method against various types and takes appropriate action for each type.

Before the any class, we would have had to write four different specializations for this function and we still wouldn't be able to easily handle the default case.

- We call this function from `main()` like this:

```
p_any({});
p_any(47);
p_any("abc"s);
p_any(any(list{ 1, 2, 3 }));
p_any(any(vector{ 1, 2, 3 }));
```

Output:

```
None.
int: 47
string: "abc"
list<int>: 1 2 3
something else: St6vectorIiSaIiEE
```

Our polymorphic function handles the various types with a minimum of code.

## How it works

The `std::any` copy constructor and assignment operator use *direct initialization* to make a non-`const` copy of the target object as the *contained* object. The type of the contained object is stored separately as a `typeid` object.

Once initialized, the `any` object has the following methods:

- `emplace()` replaces the contained object, constructing the new object in place
- `reset()` destroys the contained object
- `has_value()` returns `true` if there is a contained object
- `type()` returns a `typeid` object, representing the type of the contained object
- `operator=()` replaces the contained object by copy or move

The `any` class also supports the following non-member functions:

- `any_cast<T>()` is a template function that provides type-safe access to the contained object
  Keep in mind that the `any_cast<T>()` function returns a copy of the contained object. You may use `any_cast<T&>()` to return a reference.
- `std::swap()` specializes the `std::swap` algorithm

If you try to cast an `any` object with an incompatible type, it throws a bad_any_cast exception:

```
X = std::string("foo");
try {
    println("{}", any_cast<int>(x));
} catch (std::bad_any_cast& e) {
    println("any: {}", e.what());
}
```

Output:

```
any: bad any_cast
```

**Note**

It's worth understanding that `std::any::any_cast<>()` is typed at compile time. This is why it requires a type as a template argument. While `std::any` supports run-time inspection through its `type()` method, extraction still requires the destination type to be known at compile time.

# Store different types with std::variant

Introduced with C++17, the `std::variant` class may hold different values, one at a time, where each value must fit in the same allocated memory space. It's useful for holding alternative types for use in a single context.

## Differences from the union primitive

The variant class is a *tagged union*. It differs from the `union` primitive in that only one type may be in effect at a time.

The primitive `union` type, inherited from C, provides shared storage where the same storage may be accessed as different types. For example:

```
union ipv4 {
    uint32_t int32;
    struct {
        uint8_t a, b, c, d;
    } quad;
} addr;

addr.int32 = 0x2A05A8C0;
println("ip addr dotted quad: {}.{}.{}.{} ",
    addr.quad.a, addr.quad.b, addr.quad.c, addr.quad.d);
println("ip addr int32 (LE): {:08X} ", addr.int32);
```

Output:

```
ip addr dotted quad: 192.168.5.42
ip addr int32 (LE): 2A05A8C0
```

In this example, the `union` has two members, a `struct`, and a `uint32_t`, where the `struct` has four `uint8_t` members. This gives us two different interpretations of the same 32-bit memory

space. We can view the same IPv4 address as either a 32-bit unsigned integer, or four 8-bit unsigned integers in the common *dotted quad notation*. This provides a form of bitwise polymorphism that can be useful at the systems level.

Because this `union` example operates on the byte level, it relies on the *endianness* of the underlying machine. For example, a 32-bit integer value is commonly stored as four 8-bit bytes. In a *little-endian* system the bytes are stored least-significant byte first. On a *big-endian* system the order is reversed.

This code will behave as expected on little-endian machines but will give different results on big-endian systems. While IPv4 addresses are universally transmitted in network byte order (big-endian), they are commonly stored in the native byte order of the host system. Most modern systems use little-endian representation, yet notable exceptions remain, such as the IBM/360 family of mainframes.

**Note**

Unlike the C standard, the C++ standard specifies that only the most recently written member of a `union` is considered active. Reading any other member is formally considered undefined behavior. In practice, however, this form of type-punning is widely supported by mainstream compilers and remains common in systems-level code.

The `std::variant` class is different from a `union` in that it's a *tagged union*, where each value is tagged with its type. If we store a value as `uint32_t`, we may only access it as `uint32_t`. This makes `variant` type safe, but *not a replacement* for `union`.

## How to do it

In this recipe we demonstrate the use of `std::variant` with a small catalog of household pets of various species.

- We'll start with a simple class to hold an `Animal`:

```
class Animal {
    string_view a_name{};
    string_view a_sound{};
    Animal();
public:
    Animal(string_view n, string_view s) : a_name {n}, a_sound {s} {}
    void speak() const {
        println("{} says {}", a_name, a_sound);
```

```
    }
    void sound(string_view s) {
        a_sound = s;
    }
};
```

The name of the `Animal` and the sound that it makes are passed in the constructor.

- Individual species classes inherit from Animal:

```
class Cat : public Animal {
public:
    Cat(string_view n) : Animal(n, "meow") {}
};

class Dog : public Animal {
public:
    Dog(string_view n) : Animal(n, "arf!") {}
};

class Wookie : public Animal {
public:
    Wookie(string_view n) : Animal(n, "grrraarrgghh!") {}
};
```

Each of these classes set the sound for their specific species by calling the parent constructor.

- Now we can define our variant type in an alias:

```
using v_animal = std::variant<Cat, Dog, Wookie>;
```

This variant can hold any of the types, `Cat`, `Dog`, or `Wookie`.

- In `main()`, we create a `list` using our `v_animal` alias as the type:

```
int main() {
    list<v_animal> pets {
        Cat {"Hobbes"}, Dog {"Fido"}, Cat {"Max"},
        Wookie {"Chewie"}
    };
```

Each element in the list is of a type included in the variant definition.

- The variant class provides several different ways to access elements. First, we'll look at the `std::visit()` function.
  The `std::visit()` function is defined in the `variant` header. It calls a *functor* with an object contained in the `variant`. First, let's define a functor that accepts any of our pets:

```
struct animal_speaks {
    void operator()(const Dog& d) const { d.speak(); }
    void operator()(const Cat& c) const { c.speak(); }
    void operator()(const Wookie& w) const { w.speak(); }
};
```

This is a simple functor class with overloads for each of the `Animal` sub-classes. We call it via `visit()` with each of our list elements:

```
for (const v_animal& a : pets) {
    visit(animal_speaks{}, a);
}
```

We get this output:

```
Hobbes says meow
Fido says arf!
Max says meow
Chewie says grrraarrgghh!
```

- The `variant` class also provides an `index()` method:

```
for (const v_animal &a : pets) {
    auto idx {a.index()};
    if (idx == 0) get<Cat>(a).speak();
    if (idx == 1) get<Dog>(a).speak();
    if (idx == 2) get<Wookie>(a).speak();
}
```

Output:

```
Hobbes says meow
Fido says arf!
```

```
  Max says meow
  Chewie says grrraarrgghh!
```

Each variant object is indexed, based on the order in which the types were declared in the template arguments. Our `v_animal` type was defined with `std::variant<Cat, Dog, Wookie>`, and these types are indexed as `0` – `2` in that order.

- The `get_if<T>()` function tests a given element against a type:

```
for (const v_animal& a : pets) {
    if (const auto c {get_if<Cat>(&a)}; c) {
        c->speak();
    } else if (const auto d {get_if<Dog>(&a)}; d) {
        d->speak();
    } else if (const auto w {get_if<Wookie>(&a)}; w) {
        w->speak();
    }
}
```

Output:

```
  Hobbes says meow
  Fido says arf!
  Max says meow
  Chewie says grrraarrgghh!
```

The `get_if<T>()` function returns a pointer if the type of the element matches `T`, otherwise it returns `nullptr`.

- Finally, the `holds_alternative<T>()` function returns `true` or `false`. We can use this to test a type against an element, without returning the element:

```
size_t n_cats{}, n_dogs{}, n_wookies{};
for(const v_animal& a : pets) {
    if(holds_alternative<Cat>(a)) ++n_cats;
    if(holds_alternative<Dog>(a)) ++n_dogs;
    if(holds_alternative<Wookie>(a)) ++n_wookies;
}
println("there are {} cat(s), "
                "{} dog(s), "
```

```
                "and {} wookie(s)",
                n_cats, n_dogs, n_wookies);
```

Output:

```
there are 2 cat(s), 1 dog(s), and 1 wookie(s)
```

## How it works

The `std::variant` class is a single-object container. An instance of `variant<X, Y, Z>` must hold exactly one object of type `X`, `Y`, or `Z`. It holds both the value and the type of its current object.

The `index()` method tells us the type of the current object:

```
if (v.index() == 0) // if variant is type X
```

The `holds_alternative<T>()` non-member function returns true if `T` is the type of the current object.

```
if (holds_alternative<X>(v))  // if current variant obj is type X
```

We can retrieve the current object with the `get()` non-member function:

```
auto o {get<X>(v)};  // current variant obj must be type X
```

We can combine the test for type and retrieval with the `get_if()` non-member function:

```
auto* p {get_if<X>(v)};  // nullptr if current obj not type X
```

The `visit()` non-member function invokes a callable object with the current variant object as its single parameter:

```
visit(f, v);  // calls f(v) with current variant obj
```

The `visit()` function is the only way to retrieve an object without testing its type. In combination with a functor that can handle each type, this can be very flexible:

```
struct animal_speaks {
    void operator()(const Dog& d) const {d.speak();}
    void operator()(const Cat& c) const {c.speak();}
```

```
    void operator()(const Wookie& v) const {v.speak();}
};
main() {
    for (const v_animal& a : pets) {
        visit(animal_speaks{}, a);
    }
}
```

Output:

```
Hobbes says meow
Fido says arf!
Max says meow
Chewie says grrraarrgghh!
```

# Time events with std::chrono

The `std::chrono` library provides tools for measuring and reporting time and intervals.

The `chrono` library can be used for working with dates and times, measuring durations, and other time-related purposes.

Using the chrono library, this recipe explores techniques for timing events.

## How to do it

The `system_clock` class is used for reporting current date and time. The `steady_clock` and `high_resolution_clock` classes are used for timing events. Let's look at the differences between these clocks.

- Because these names can be long and unwieldy, we'll use some type aliases throughout this recipe:

```
using std::chrono::system_clock;
using std::chrono::steady_clock;
using std::chrono::high_resolution_clock;
using std::chrono::duration;
using seconds = duration<double>;
using milliseconds = duration<double, std::milli>;
using microseconds = duration<double, std::micro>;
using fps24 = duration<unsigned long, std::ratio<1, 24>>;
```

The `duration` class represents an interval between two points in time. These aliases are convenient for using different intervals.

- We can get the current time and date, using the `system_clock` class:

```
auto t = system_clock::now();
println("system_clock::now is {:%F %T %Z}", t);
```

The `system_clock::now()` function returns a `time_point` object. The `<chrono>` library includes a `format()` specialization for `time_point` that uses `strftime()` format specifiers.

The output is:

```
  system_clock::now is 2025-12-26 19:23:21.412144 UTC
```

The number of digits after the decimal will vary on different systems.

We can also use `system_clock` to time an event. First, we need something to time. Here's a function that counts primes:

```
constexpr uint64_t MAX_PRIME {0x8FFFFF};

uint64_t count_primes() {
    constexpr auto isprime = [](uint64_t n) {
        if (n == 2) return true;
        if (n < 2 || n % 2 == 0) return false;
        for (uint64_t i{3}; i <= n / i; i += 2) {
            if (n % i == 0) return false;
        }
        return true;
    };

    uint64_t count {MAX_PRIME >= 2 ? 1 : 0};  // count 2
    uint64_t start {3};
    uint64_t end {MAX_PRIME};

    for (uint64_t i {start}; i <= end; i += 2) {
        if (isprime(i)) ++count;
    }
```

```
    return count;
}
```

This function counts the prime numbers between `3` and `0x8FFFFF` (9,437,183), which should take a few seconds on most modern systems.

- Now, we can write a timer function to time our `count_primes()`:

```
seconds timer(uint64_t(*f)()) {
    auto t1 {system_clock::now()};
    uint64_t count{ f() };
    auto t2 {system_clock::now()};
    seconds secs {t2 - t1};
    println("there are {} primes in range", count);
    return secs;
}
```

This function takes a function argument `f` and returns a `duration<double>`. We use `steady_clock::now()` to mark the time before and after the call to `f()`. We take the difference between the two times and return it in a duration object.

- We call our `timer()` from `main()` like this:

```
int main() {
    auto secs {timer(count_primes)};
    println("time elapsed: {:.3f} sec", secs.count());
    ...
```

This passes the `count_primes()` function to `timer()` and stores the duration object in secs.

Output:

```
there are 629,686 primes in range
time elapsed: 12.928 seconds
```

The `count()` method on the duration object returns the duration in the specified units, in this case, a double representing *seconds* of duration.

This result was produced on a VM running *Debian* with GCC. The exact time will vary on different systems.

- The `system_clock` class is designed to provide the current *wall clock* time. While its resolution may support timing purposes, it is not guaranteed to be *monotonic*. In other words, it may not always provide consistent *ticks* (timing intervals).
  The `chrono` library provides a clock more suitable for timing events in `steady_clock`. It has the same interface as `system_clock`, but provides more reliable ticks for timing purposes:

  ```
  seconds timer(uint64_t(*f)()) {
      auto t1 {steady_clock::now()};
      uint64_t count {f()};
      auto t2 {steady_clock::now()};
      seconds secs {t2 - t1};
      println("there are {} primes in range", count);
      return secs;
  }
  ```

  `steady_clock` is designed to provide reliably consistent, monotonic ticks, suitable for timing events. It uses a relative time reference, so it's not useful for wall clock time. While `system_clock` measures beginning with a fixed point in time (e.g., 1 January 1970, 00:00 UTC), `steady_clock` uses a relative time.

  Another option is `high_resolution_clock`, which provides the shortest tick period available on a given system, but is not implemented consistently across different systems. It may be an alias for `system_clock` or `steady_clock`, and it may or may not be monotonic. `high_resolution_clock` is not recommended for general purpose use.

- Our `timer()` function returns `seconds`, which is an alias for `duration<double>`:

  ```
  using seconds = duration<double>;
  ```

  The `duration` class takes an optional second template parameter, a ratio:

  ```
  template<class Rep, class Period = std::ratio<1>>
  class duration;
  ```

  The `<chrono>` header provides convenience types for many decimal ratios, including `milli` and `micro`:

  ```
  using milliseconds = duration<double, std::milli>;
  using microseconds = duration<double, std::micro>;
  ```

If we require something else, we may provide our own. For example:

```
using fps24 = duration<unsigned long, std::ratio<1, 24>>;
```

`fps24` represents the number of frames of film shot at the standard 24 frames per second. The ratio is 1/24 of a second.

This allows us to easily convert between different ranges of duration:

```
println("time elapsed: {:.3f} sec", secs.count());
println("time elapsed: {:.3f} ms",
    milliseconds(secs).count());
println("time elapsed: {:.3e} µs",
    microseconds(secs).count());
println("time elapsed: {} frames at 24 fps",
    floor<fps24>(secs).count());
```

Output:

```
time elapsed: 12.928 sec
time elapsed: 12927.994 ms
time elapsed: 1.293e+07 µs
time elapsed: 310 frames at 24 fps
```

Because the `fps24` alias uses `unsigned long` instead of `double`, a type conversion is required. The `floor` function provides this by discarding the fractional part. `round()` and `ceil()` are also available in this context.

- For convenience, the `chrono` library provides `format()` specializations for the standard duration ratios:

```
println("time elapsed: {:.3}", secs);
println("time elapsed: {:.3}", milliseconds(secs));
println("time elapsed: {:.3}", microseconds(secs));
```

Output:

```
time elapsed: 12.928s
time elapsed: 12928ms
time elapsed: 1.2928e+07µs
```

These results will, of course, vary on different systems.

## How it works

There are two major pieces of the `chrono` library at work here: the various *clock* classes and the `duration` class.

### The clock classes

The clock classes include:

- `system_clock` – provides wall clock time.
- `steady_clock` – provides guaranteed monotonic ticks for duration measurements
- `high_resolution_clock` – provides the shortest available tick period. May be an alias for `system_clock` or `steady_clock` on some systems.

We use `system_clock` to display the current time and date. We use `steady_clock` to measure intervals.

Each of the clock classes has a `now()` method that returns a `time_point`, representing the current value of the clock. `now()` is a static member function, so it's called on the class without instantiating an object:

```
auto t1 {steady_clock::now()};
```

### std::duration class

The `duration` class is used to hold a time interval, that is, the difference between two `time_point` objects. It is generally constructed with a `time_point` object's subtraction operator.

```
duration<double> secs {t2 - t1};
```

The `time_point` subtraction operator doubles as a constructor for a `duration` object:

```
template<class C, class D1, class D2>
constexpr duration<D1,D2>
operator-(const time_point<C,D1>& pt_lhs,
    const time_point<C,D2>& pt_rhs);
```

The `duration` class has template parameters for type-representation and a `ratio` object:

```
template<class Rep, class Period = std::ratio<1>>
class duration;
```

The `Period` template parameter defaults to a ratio of 1:1, which is `seconds`.

The library provides `ratio` aliases (such as `micro`, `milli`, etc.) for powers-of-ten from `atto` (1/1,000,000,000,000,000,000) through `exa` (1,000,000,000,000,000,000/1). This allows us to create standard durations as we did in our example:

```
using milliseconds = duration<double, std::milli>;
using microseconds = duration<double, std::micro>;
```

The `count()` method gives us the duration in `Rep` type:

```
constexpr Rep count() const;
```

This allows us to easily access the duration for display or other purposes:

```
println("duration: {}", secs.count());
```

# Use fold expressions for generic tuples

The `std::tuple` class is essentially a more complex, less convenient, `struct`. The interface for `tuple` is cumbersome, although *class template argument deduction* and *structured binding* has made it somewhat easier.

The one real advantage of `tuple` is that it can be used with *fold expressions* to support varying numbers of elements of varying types.

## Fold expressions

Designed to make it easier to expand a variadic parameter pack, *fold expressions* were introduced with C++17. Prior to fold expressions, expanding a parameter pack required a recursive function:

```
template<typename T>
void f(T final) {
    print("\n");
}
```

```
template<typename T, typename... Args>
void f(T first, Args... args) {
    print("{}", first);
    f(args...);
}

int main() {
    f("hello", ' ', 47, ' ', "world");
}
```

Output:

```
hello 47 world
```

Using a fold expression this is much simpler:

```
template<typename... Args>
void f(Args... args) {
    (std::print("{}", args), ...);
}
```

Output:

```
hello 47 world
```

There are four types of fold expressions:

- Unary right fold: `(args op ...)`
- Unary left fold: `(... op args)`
- Binary right fold: `(args op ... op init)`
- Binary left fold: `(init op ... op args)`

The expression in the example above is a *unary right fold*.

```
(std::print("{}", args), ...);
```

It's *unary* because there is one operator, the comma, and it's a *right* fold because the ellipsis is to the right of the operator.

This expands to:

```
print("{}", "hello");
print("{}", ' ');
print("{}", 47);
print("{}", ' ');
print("{}", "world");
```

Fold expressions are a great convenience for many purposes. Let's look at how we can use them with tuples.

## How to do it

In this recipe we'll create a template function that operates on a `tuple` with varying numbers and types of elements.

- The heart of this recipe is a function that takes a `tuple` of unknown size and type and prints each element with `print()`:

  ```
  template<typename... T>
  void print_t(const std::tuple<T...>& tup) {
      std::apply([](const auto&... elems) {
          ((std::print("{} ", elems)), ...);
      }, tup);
      std::print("\n");
  }
  ```

  This function uses `std::apply` to translate the `tuple` into a series of arguments passed to the lambda. Then, within the lambda, we use a *unary right fold* expression to call `print()` with each argument.

  We could use a separate function in place of the lambda, but I prefer to keep it in a single scope.

- We can now call this from `main()` with a variety of tuples of a mixture of types:

  ```
  int main() {
      tuple lables {"ID", "Name", "Scale"};
      tuple employee {123456, "John Doe", 3.7};
      tuple nums {1, 7, "forty-two", 47, 73L, -111.11};

      print_t(lables);
  ```

```
    print_t(employee);
    print_t(nums);
}
```

The fold expression operates with however many elements, and however many different types. Each parameter is passed to the `print()` function with its type intact, allowing the formatter to apply the correct formatting.

Output:

```
ID Name Scale
123456 John Doe 3.7
1 7 forty-two 47 73 -111.11
```

## How it works

The challenge with `tuple` is its restrictive interface. You can retrieve elements with `std::tie()`, with *structured bindings*, or the `std::get<>` function. None of these techniques are useful if you don't know the number and type of elements in the `tuple`.

We get around this limitation by using the `std::apply()` function:

```
template< class F, class Tuple >
constexpr decltype(auto) apply( F&& f, Tuple&& t );
```

The `std::apply()` function takes a function argument `f` and a `tuple t`. It calls the function with each of the arguments of the tuple. This effectively unpacks the `tuple` and passes its elements as fully typed arguments to the function.

We could pass a function, functor, or lambda to `apply()`. I chose a lambda for its simplicity and to keep everything in the scope of the function:

```
std::apply([](const auto&... elems) {
    ((std::print("{} ", elems)), ...);
}, tup);
```

This keeps the solution simple and self-contained.

The *unary right fold* expression takes each element of the lambda's parameter pack and passes it to the `print()` function.

Keep in mind that, as with template functions in general, the compiler will generate a separate specialization of this function for each combination of tuple parameters.

## There's more...

We can use this technique for other tasks. For example, here's a function that returns the sum of all the int values in a tuple of unknown size:

```
template<typename... T>
constexpr int sum_t(const tuple<T...>& tup) {
    return std::apply([](const auto&... elems) {
        return (elems + ...);
    }, tup);
}
```

This uses the same technique as the example in our recipe, using `std::apply` to unpack the `tuple` and pass its elements as parameters to a lambda. Then we can use a simple *unary right fold* with a + operator to return the sum of the elements.

We can call this with several tuple objects of varying numbers of int values:

```
    tuple ti1 {1, 2, 3, 4, 5};
    tuple ti2 {9, 10, 11, 12, 13, 14, 15};
    tuple ti3 {47, 73, 42};
    auto sum1 {sum_t(ti1)};
    auto sum2 {sum_t(ti2)};
    auto sum3 {sum_t(ti3)};
    println("sum of ti1: {}", sum1);
    println("sum of ti2: {}", sum2);
    println("sum of ti3: {}", sum3);
```

Output:

```
sum of ti1: 15
sum of ti2: 84
sum of ti3: 162
```

# Manage allocated memory with std::unique_ptr

*Smart pointers* are an excellent tool for managing allocated *heap memory*.

Heap memory is managed at the lowest level by the C functions `malloc()` and `free()`. `malloc()` allocates a block of memory from the heap, and `free()` returns it to the heap. These functions do not perform initialization and they do not call constructors or destructors. If you

fail to return allocated memory to the heap with a call to `free()`, the behavior is undefined and often leads to memory leaks and security vulnerabilities.

C++ provides the `new` and `delete` operators to allocate and free heap memory in place of `malloc()` and `free()`. The `new` and `delete` operators call object constructors and destructors but still do not manage memory. If you allocate memory with `new` and fail to free it with `delete`, you will likely get memory leaks.

Introduced with C++14, *smart pointers* comply with the RAII idiom, *Resource Acquisition Is Initialization*. This means that when memory is allocated for an object, that object's constructor is called, and when the object's destructor is called, the memory is automatically returned to the heap.

For example, when we create a new smart pointer with `make_unique()`:

```
{   // beginning of scope
    auto p = make_unique<Thing>(); // memory alloc'd, ctor called,
    process_thing(p);              // p is unique_ptr<Thing>
}   // end of scope, dtor called, memory freed
```

`make_unique()` allocates memory for a `Thing` object, calls the `Thing` default constructor, constructs a `unique_ptr<Thing>` object, and returns the `unique_ptr`. When `p` goes out of scope, the `Thing` destructor is called, and the memory is automatically returned to the heap.

Aside from the memory management, a smart pointer works very much like a primitive pointer:

```
auto x = *p;  // *p derefs the pointer, returns Thing object
auto y = p->thname; // p-> derefs the pointer, returns member
```

`unique_ptr` is a smart pointer that allows only one instance of the pointer. It may be moved but it may not be copied. Let's take a closer look at how to use `unique_ptr`.

## How to do it

In this recipe we examine `std::unique_ptr` with a demonstration class that prints when its constructors and destructor are called.

- First, we'll create a simple demonstration class:

```
class Thing {
    string_view thname {"unk"};
public:
```

```
    Thing() {
        println("default ctor: {}", thname);
    }
    Thing(const string_view& n) : thname(n) {
        println("param ctor: {}", thname);
    }
    ~Thing() {
        println("dtor: {}", thname);
    }
    string_view name() const {
        return thname;
    }
};
```

This class has a default constructor, a parameterized constructor, and a destructor. Each of these has a simple print statement to tell us what was called. I've also included a getter to return the name of the `Thing`.

- When we construct a `unique_ptr` by itself, it does not allocate memory or construct a managed object:

```
int main() {
    unique_ptr<Thing> p1;
    println("end of main()");
}
```

Output:

```
  end of main()
```

- When we use the `new` operator, it allocates memory and constructs a `Thing` object:

```
int main() {
    unique_ptr<Thing> p1 {new Thing};
    println("end of main()");
}
```

Output:

```
default ctor: unk
end of main()
dtor: unk
```

The new operator constructs a `Thing` object by calling the default constructor.

Notice that the `"end of main()"` message occurs before the `dtor` message. The `unique_ptr<Thing>` destructor calls the `Thing` destructor when the smart pointer reaches the end of its scope, which is after the `"end of main()"` message.

The `Thing` default constructor does not initialize the `thname` string, leaving its default value, `"unk"`.

- We can use `make_unique()` to get the same result:

```
int main() {
    auto p1 = make_unique<Thing>();
    println("end of main()");
}
```

Output

```
default ctor: unk
end of main()
dtor: unk
```

The `make_unique()` helper function takes care of the memory allocation and returns a `unique_ptr` object. This is the recommended way to construct a `unique_ptr`.

- Any arguments you pass to `make_unique()` are used in constructing the target object:

```
int main() {
    auto p1 = make_unique<Thing>("Thing 1") };
    println("end of main()");
}
```

Output:

```
  param ctor: Thing 1
  end of main()
  dtor: Thing 1
```

The parameterized constructor assigns a value to `thname` so our `Thing` object is now `"Thing 1"`.

- Let's write a function that takes a `unique_ptr<Thing>` argument:

```
void process_thing(unique_ptr<Thing> p) {
    if (p) println("processing: {}", p->name());
    else println("invalid pointer");
}
```

If we try to pass a `unique_ptr` to this function, we get a compiler error:

```
process_thing(p1);
```

Compiler error:

```
  error: use of deleted function...
```

This fails because the call by value argument to the function call tries to make a copy of the `unique_ptr` object, but the `unique_ptr` copy constructor is *deleted* to prevent copying. The solution is to have the function take a `const&` reference:

```
void process_thing(const unique_ptr<Thing>& p) {
    if (p) println("processing: {}", p->name());
    else println("invalid pointer");
}
```

Output:

```
  param ctor: Thing 1
  processing: Thing 1
  end of main()
  dtor: Thing 1
```

The `const&` reference parameter maintains the single instance the `unique_ptr` object as intended.

- We can call `process_thing()` with a temporary object, which is immediately destroyed at the end of the function scope:

```
int main() {
    auto p1 {make_unique<Thing>("Thing 1")};
    process_thing(p1);
    process_thing(make_unique<Thing>("Thing 2"));
    println("end of main()");
}
```

Output:

```
  param ctor: Thing 1
  processing: Thing 1
  param ctor: Thing 2
  processing: Thing 2
  dtor: Thing 2
  end of main()
  dtor: Thing 1
```

## How it works

A *smart pointer* is simply an object that presents a pointer interface, while owning and managing the resources of another object.

The `unique_ptr` class is distinguished by its deleted copy constructor and copy assignment operator, which prevents the smart pointer from being copied.

You may not copy a `unique_ptr`:

```
auto p2 = p1;
```

Compiler error:

```
error: use of deleted function...
```

But you can *move* a `unique_ptr`:

```
auto p2 = std::move(p1);
process_thing(p1);
process_thing(p2);
```

After the move, p1 is invalid and p2 now *owns* the object which was initialized as "`Thing 1`."

Output:

```
invalid pointer
processing: Thing 1
end of main()
dtor: Thing 1
```

The `unique_ptr` interface has a `reset()` method which calls the object destructor and invalidates the pointer:

```
p1.reset();  // pointer is now invalid
process_thing(p1);
```

Output:

```
dtor: Thing 1
invalid pointer
```

The `reset()` method may also be used to replace the managed object with another of the same type:

```
p1.reset(new Thing("Thing 3"));
process_thing(p1);
```

Output:

```
param ctor: Thing 3
dtor: Thing 1
processing: Thing 3
```

# Share objects with std::shared_ptr

The `std::shared_ptr` class is a smart pointer that owns its managed object and maintains a *use counter* to keep track of the number of references to the object so that the destructor may be called when the last copy of the `shared_ptr` is destroyed. This is distinct from a traditional "garbage collection" scheme which attempts to discover and reclaim unreachable objects automatically. This recipe explores the use of `shared_ptr` to manage memory while sharing ownership of the pointer.

**Note**

For more detail about smart pointers, see the introduction to the recipe "Manage allocated memory with `std::unique_ptr`", earlier in this chapter.

## How to do it

In this recipe we examine `std::shared_ptr` with a demonstration class that prints when its constructors and destructor are called.

- First, we'll create a simple demonstration class:

```
class Thing {
    string_view thname{"unk"};
public:
    Thing() {
        println("default ctor: {}", thname);
    }
    Thing(const string_view& n) : thname(n) {
        println("param ctor: {}", thname);
    }
    ~Thing() {
        println("dtor: {}", thname);
    }
    string_view name() const {
        return thname;
    }
};
```

This class has a default constructor, a parameterized constructor, and a destructor. Each of these has a simple `print()` statement to tell us what was called.

The `shared_ptr` class works very much like other smart pointers, in that it may be constructed with the `new` operator or with its helper `make_shared()` function:

```
int main() {
    shared_ptr<Thing> p1 {new Thing("Thing 1")};
    auto p2 = make_shared<Thing>("Thing 2");
    println("end of main()");
}
```

Output:

```
param ctor: Thing 1
param ctor: Thing 2
end of main()
dtor: Thing 2
dtor: Thing 1
```

The `make_shared()` function is recommended as it manages the construction process and is less prone to error.

As with the other smart pointers, the managed object is destroyed and its memory is returned to the heap when the pointer goes out of scope. In the case of `main()`, this happens *after* the last line of the function, which prints `"end of main()"`.

- The `shared_pointer` class keeps a counter to monitor the number of copies of the pointer. Here's a function to check the use count of a `shared_ptr` object:

```
void check_thing_ptr(const shared_ptr<Thing>& p) {
    if (p) println("{} use count: {}",
        p->name(), p.use_count());
    else println("invalid pointer");
}
```

`name()` is a function member of the `Thing` class, so we access it through the pointer with the member dereference `p->` operator. The `use_count()` function is a member of the `shared_ptr` class, so we access it with the object member `p.` operator.

Let's call this with our pointers:

```
check_thing_ptr(p1);
check_thing_ptr(p2);
```

Output:

```
Thing 1 use count: 1
Thing 2 use count: 1
```

- When we make copies of our pointers, the use count increases but no new objects are constructed:

```
println("make 4 copies of p1: ");
auto pa = p1;
auto pb = p1;
auto pc = p1;
auto pd = p1;
check_thing_ptr(p1);
```

Output:

```
make 4 copies of p1:
Thing 1 use count: 5
```

- When we check any of the other copies, we get the same result:

```
check_thing_ptr(pa);
check_thing_ptr(pb);
check_thing_ptr(pc);
check_thing_ptr(pd);
```

Output:

```
Thing 1 use count: 5
Thing 1 use count: 5
Thing 1 use count: 5
Thing 1 use count: 5
```

Each pointer reports the same use count.

- When the copies go out of scope, they are destroyed and the use count is decremented:

```
{   // new scope
    cout << "make 4 copies of p1:\n";
    auto pa = p1;
```

```
    auto pb = p1;
    auto pc = p1;
    auto pd = p1;
    check_thing_ptr(p1);
}   // end of scope
check_thing_ptr(p1);
```

Output:

```
make 4 copies of p1:
Thing 1 use count: 5
Thing 1 use count: 1
```

- Destroying a copy reduces the use count but does not destroy the managed object. The object is destroyed when the final copy goes out of scope and the use count reaches zero:

```
{
    cout << "make 4 copies of p1:\n";
    auto pa = p1;
    auto pb = p1;
    auto pc = p1;
    auto pd = p1;
    check_thing_ptr(p1);
    pb.reset();
    p1.reset();
    check_thing_ptr(pd);
}   // end of scope
```

Output:

```
make 4 copies of p1:
Thing 1 use count: 5
Thing 1 use count: 3
dtor: Thing 1
```

Destroying `pb` (a copy) and `p1` (the original) leaves three copies of the pointer (`pa`, `pc`, and `pd`), so the managed object remains.

The remaining three pointer copies are destroyed at the end of the scope in which they were created. Then the object is destroyed and its memory returned to the heap.

**Note**

It's important to understand that the `std::shared_ptr` class makes it possible for two objects to refer to each other, creating a condition known as *circular references*. The solution is to use a `std::weak_ptr` for one of the references. We'll discuss this in the next recipe, "Use weak pointers with shared objects".

## How it works

The `shared_ptr` class is distinguished by its management of multiple pointers to the same managed object.

The `shared_ptr` object's copy constructor and copy assignment operator increment a *use counter*. The destructor decrements the use counter until it reaches zero, then destroys the managed object and returns its memory to the heap.

The `shared_ptr` class manages both the managed object and a heap-allocated *control block*. The control block contains the use counter, along with other housekeeping objects. The control block is managed and shared between copies along with the managed object. This allows the original `shared_ptr` object to cede control to its copies, so that the last remaining `shared_ptr` may manage the object and its memory.

# Use weak pointers with shared objects

Strictly speaking, `std::weak_ptr` is not a smart pointer. Rather, it's an *observer* that operates in cooperation with a `shared_ptr`. A `weak_ptr` object does not hold a pointer on its own.

There are circumstances where `shared_ptr` objects may create dangling pointers or race conditions, which could lead to memory leaks or other problems. The solution is to use `weak_ptr` objects with the `shared_ptr`.

## How to do it

In this recipe we examine the use of `std::weak_ptr` with `std::shared_ptr`, using a demonstration class that prints when its constructors and destructor are called.

- We'll start with the same class we've used to demonstrate `shared_ptr` and `unique_ptr`:

```
class Thing {
    string_view thname {"unk"};
public:
    Thing() {
```

```
        println("default ctor: {}", thname);
    }
    Thing(const string_view& n) : thname(n) {
        println("param ctor: {}", thname);
    }
    ~Thing() {
        println("dtor: {}", thname);
    }
    string_view name() const {
        return thname;
    }
};
```

This class has a default constructor, a parameterized constructor, and a destructor. Each of these has a simple `print()` statement to tell us what was called.

- We also need a function to examine a `weak_ptr` object:

```
void get_weak_thing(const weak_ptr<Thing>& p) {
    if (auto sp = p.lock()) {
        println("{}: count {}",
           sp->name(), p.use_count());
    } else println("no shared object");
}
```

A `weak_ptr` does not operate as a pointer on its own, it requires the use of a `shared_ptr`. The `lock()` function returns a `shared_ptr` object, which can then be used to access the managed object.

- Because `weak_ptr` requires an associated `shared_ptr`, we'll start `main()` by creating a `shared_ptr<Thing>` object. When we create a `weak_ptr` object without assigning the `shared_ptr`, the expired flag is initially set:

```
int main() {
    auto thing1 = make_shared<Thing>("Thing 1");
    weak_ptr<Thing> wp1;
    println("expired: {}", wp1.expired());
    get_weak_thing(wp1);

    println("\nend of main()");
}
```

Output:

```
param ctor: Thing 1
expired: true
no shared object

end of main()
dtor: Thing 1
```

The `make_shared()` function allocates memory and constructs a `Thing` object.

The `weak_ptr<Thing>` declaration constructs a `weak_ptr` object without assigning a `shared_ptr`. So, when we check the `expired` flag, it's `true`, indicating that there is no associated `shared_ptr`.

The `get_weak_thing()` function is unable to obtain a lock because there is no `shared_ptr` available.

- When we assign the shared_ptr to the weak_ptr, we can use the weak_ptr to access the managed object:

```
wp1 = thing1;
get_weak_thing(wp1);
```

Output:

```
Thing 1: count 2
```

The `get_weak_thing()` function is now able to obtain a lock and access the managed object. The `lock()` method returns a `shared_ptr`, and the `use_count()` reflects the fact that there is now a second `shared_ptr` managing the `Thing` object.

The new `shared_ptr` is destroyed at the end of the `get_weak_thing()` scope.

- The `weak_ptr` class has a constructor that takes a `shared_ptr` for one-step construction:

```
weak_ptr<Thing> wp2(thing1);
get_weak_thing(wp2);
```

Output:

```
  Thing 1: count 2
```

The `use_count()` is 2 again. Remember that the previous `shared_ptr` was destroyed when its enclosing `get_weak_thing()` scope ended.

When we reset the `shared_ptr`, it's associated `weak_ptr` objects are expired:

```
thing1.reset();
get_weak_thing(wp1);
get_weak_thing(wp2);
```

Output:

```
  dtor: Thing 1
  no shared object
  no shared object
```

After the `reset()`, the use count reaches zero and the managed object is destroyed and the memory released.

## How it works

A `weak_ptr` object is an *observer* that holds a non-owning reference to a `shared_ptr` object. The `weak_ptr` observes the `shared_ptr` so that it knows when the managed object is and is not available. This allows use of a `shared_ptr` in circumstances where you may not always know if the managed object is active.

The `weak_ptr` class has a `use_count()` function that returns the use count of the `shared_ptr`, or zero if the managed object has been deleted.

```
long use_count() const noexcept;
```

`weak_ptr` also has an `expired()` function that reports if the managed object has been deleted:

```
bool expired() const noexcept;
```

The `lock()` function is the preferred way to access the shared pointer. It checks `expired()` to see if the managed object is available. If so, it returns a new `shared_ptr` that shares ownership

with the managed object. Otherwise, it returns an empty `shared_ptr`. It does all that as one atomic operation:

```
std::shared_ptr<T> lock() const noexcept;
```

## There's more...

One important use case for a `weak_ptr` is when there's a possibility of *circular references* to `shared_ptr` objects. For example, consider the case of two classes that link to each other:

```
struct circB; // forward declaration
struct circA {
    shared_ptr<circB> p;
    ~circA() { println("dtor A"); }
};
struct circB {
    shared_ptr<circA> p;
    ~circB() { println("dtor B"); }
};
```

We have `print()` statements in the destructors so we can see when the objects are destroyed. We can now create two objects that point to each other with `shared_ptr`:

```
int main() {
    auto a{ make_shared<circA>() };
    auto b{ make_shared<circB>() };
    a->p = b;
    b->p = a;
    println("\nend of main()");
}
```

When we run this, notice that the destructors are never called:

```
end of main()
```

Because the objects maintain shared pointers that refer to each other, the use counts never reach zero and the managed objects are never destroyed.

We can resolve this problem by changing one of the classes to use a `weak_ptr`:

```
struct circB {
    weak_ptr<circA> p;
    ~circB() { println("dtor B "); }
};
```

The code in `main()` remains the same, and we get this output:

```
end of main()
dtor A
dtor B
```

By changing one `shared_ptr` to `weak_ptr`, we have resolved the circular reference, and the objects are now properly destroyed at the end of their scope.

# Share object members with aliased smart pointers

The `std::shared_ptr` class provides an *aliasing constructor* to share a pointer managed by another unrelated pointer:

```
shared_ptr( shared_ptr<Y>&& ref, element_type* ptr ) noexcept;
```

This returns an aliased `shared_ptr` object that uses the resources of `ref` but returns a pointer to `ptr`. The `use_count` and the `deleter` are shared with `ref`, but `get()` returns `ptr`. This allows us to share a member of a managed object without sharing the entire object, and without allowing the entire object to be deleted while we're still using the member.

## How to do it

In this recipe we create a managed object and share members of that object.

- We'll start with a class for the managed object:

```
struct Animal {
    string name {};
    string sound {};
    Animal(const string& n, const string& a) :
            name{n}, sound{a} {
        println("ctor: {}", name);
    }
    ~Animal() {
```

```
        println("dtor: {}", name);
    }
};
```

This class has two members, `string` objects for the `name` and `sound` of the `Animal`. We also have `print()` statements for the constructor and destructor.

- Now we need a function to create an `Animal`, but share only its `name` and `sound`:

```
auto make_animal(const string& n, const string& s) {
    auto ap = make_shared<Animal>(n, s);
    auto np = shared_ptr<string>(ap, &ap->name);
    auto sp = shared_ptr<string>(ap, &ap->sound);
    return tuple(np, sp);
}
```

This function creates a `shared_ptr` with an `Animal` object, constructed with a `name` and a `sound`. We then create aliased `shared_ptr` objects for the `name` and `sound`. When we return the `name` and `sound` pointers, the `Animal` pointer goes out of scope. But it is *not deleted* because the aliased pointers keep the *use count* from reaching zero.

- In our `main()` function we call `make_animal()` and inspect the results:

```
int main() {
    auto [name, sound] =
        make_animal("Velociraptor", "Grrrr!");
    println("The {} says {}", *name, *sound);
    println("Use count: name {}, sound {}",
        name.use_count(), sound.use_count());
}
```

Output:

```
ctor: Velociraptor
The Velociraptor says Grrrr!
Use count: name 2, sound 2
dtor: Velociraptor
```

We can see that the aliased pointers each show a `use_count` of 2. When the `make_animal()` function creates the aliased pointers, they each increase the use count

of the `Animal` pointer. When the function ends, the `Animal` pointer goes out of scope, leaving its use count at 2, which is reflected in the aliased pointers. The aliased pointers go out of scope at the end of `main()`, which allows the `Animal` object to be destroyed.

## How it works

The *aliased* shared pointer seems a bit abstract, but it's simpler than it appears.

A shared pointer uses a *control block* to manage its resources. One control block is associated with one managed object and is shared among the pointers that share that object. The control block generally contains:

- A pointer to the managed object
- The *deleter*
- The *allocator*
- The number of `shared_ptr` objects that own the managed object (this is the *use count*)
- The number of `weak_ptr` objects that refer to the managed object

In the case of an aliased shared pointer, the control block includes the pointer to the *aliased object*. Everything else remains the same.

Aliased shared pointers participate in the use count, just like non-aliased shared pointers, preventing the managed object from being destroyed until the use count reaches zero. The deleter is not changed, so it destroys the managed object.

**Note**

It is possible to use any pointer to construct an aliased pointer. Usually, the pointer refers to a member within the aliased object. If the aliased pointer does not refer to part of the managed object, you will need to manage its construction and destruction separately.

# Compare random number engines

The `random` library provides a selection of random number generators, each with different strategies and properties. In this recipe, we examine a function to compare the different options by creating a histogram of their outputs.

## How to do it

In this recipe we compare the different random number generators provided by the C++ random library.

- We'll start with some constants to provide uniform parameters for the random number generators:

```
constexpr size_t n_samples {1000};
constexpr size_t n_partitions {10};
constexpr size_t n_max {50};
```

`n_samples` is the number of samples to examine. `n_partitions` is the number of partitions in which to display the samples, and `n_max` is the maximum size of a bar in the histogram (this will vary some due to rounding).

These numbers provide a reasonable display of the differences between the engines. Increasing the ratio of *samples* vs *partitions* tends to smooth out the curves and obscure the differences between the engines.

- This is the function that collects random number samples and displays a histogram:

```
template <typename RNG>
void histogram(const string_view& rng_name) {
    auto p_ratio = (double)RNG::max() / n_partitions;

    RNG rng{};
    vector<size_t> v(n_partitions);
    for (size_t i{}; i < n_samples; ++i) {
        ++v[(size_t)(rng() / p_ratio)];
    }

    auto max_el = std::max_element(v.begin(), v.end());
    auto v_ratio = *max_el / n_max;
    if (v_ratio < 1) v_ratio = 1;

    println("engine: {}", rng_name);
    for (size_t i{}; i < n_partitions; ++i) {
        println("{:02}:{:*<{}}", i + 1, ' ', v[i] / v_ratio);
    }
    print("\n");
}
```

This function stores a histogram of collected samples in a `vector`. It then displays the histogram as a series of asterisks on the console.

- We call `histogram()` from `main()` like this:

```
int main() {
    histogram<std::random_device>("random_device");
    histogram<std::default_random_engine>
        ("default_random_engine");
    histogram<std::minstd_rand0>("minstd_rand0");
    histogram<std::minstd_rand>("minstd_rand");
    histogram<std::mt19937>("mt19937");
    histogram<std::mt19937_64>("mt19937_64");
    histogram<std::ranlux24_base>("ranlux24_base");
    histogram<std::ranlux48_base>("ranlux48_base");
    histogram<std::ranlux24>("ranlux24");
    histogram<std::ranlux48>("ranlux48");
    histogram<std::knuth_b>("knuth_b");
}
```

Output:

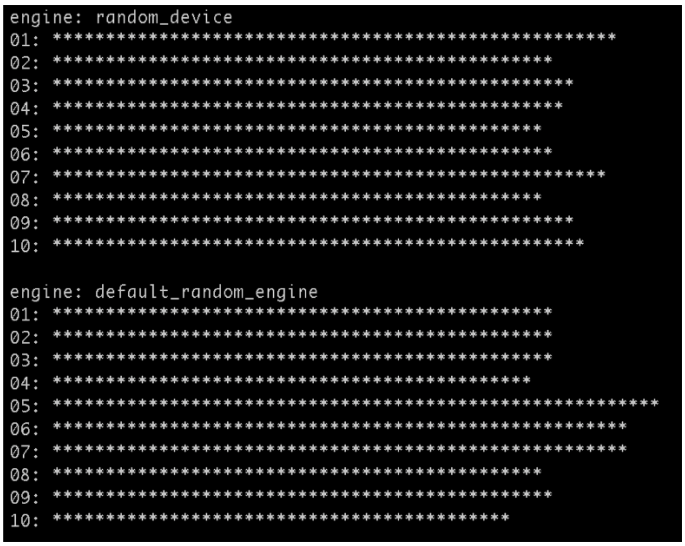



*Figure 8.1: Screenshot of output from first two random number engines*

This screenshot shows histograms of the first two random number engines. Your output will vary.

If we raise the value of `n_samples` to 100,000 you'll see the variance between engines becomes more difficult to discern:

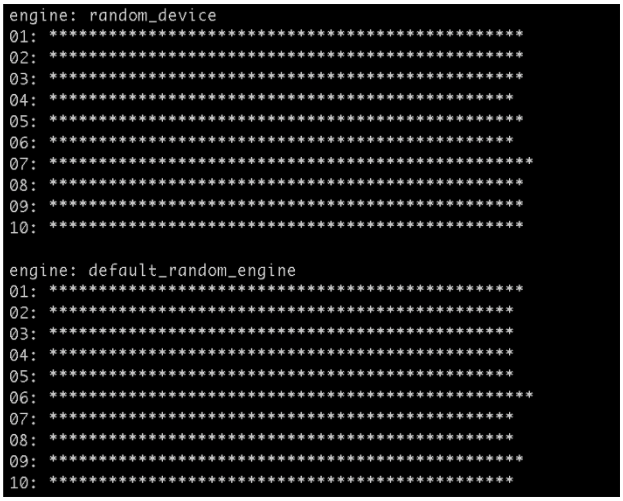



*Figure 8.2: Screenshot of output with 100,000 samples*

## How it works

Each of the random number engines has a functor interface that returns the next random number in the sequence:

```
result_type operator()();
```

The functor returns a random value evenly distributed between the `min()` and `max()` values. Each of the random number engines has this interface in common.

The `histogram()` function takes advantage of this uniformity by using the class of the random number engine in a template:

```
template <typename RNG>
```

(RNG is a common abbreviation for *Random Number Generator*. The library documentation refers to these classes as *engines*, which is synonymous for our purposes.)

We instantiate an object with the RNG class and create a histogram in a vector:

```
RNG rng{};
vector<size_t> v(n_partitions);
for(size_t i{}; i < n_samples; ++i) {
```

```
    ++v[rng() / p_ratio];
}
```

We can easily compare the results of the various random number engines with this technique.

## There's more...

Each of the random number engines in the library has different methodologies and characteristics. When you run the histogram multiple times, you'll notice that most of the engines have the same distribution each time they're run. That's because they are *deterministic*, that is, they generate the same sequence of numbers each time. The `std::random_device` is non-deterministic on most systems. You can use it to seed one of the other engines if you need more variation. It is also common to seed an RNG with the current date and time.

The `std::default_random_engine` is a suitable choice for most purposes.

# Comparison of random number distribution generators

The C++ Standard Library provides a selection of random number distribution generators, each with its own properties. In this recipe we examine a function to compare the different options by creating a histogram of their outputs.

## How to do it

Like the random number engines, the distribution generators have some common interface elements. Unlike the random number engines, the distribution generators have a variety of properties to set. We can create a template function to print a histogram of the various distributions, but their initializations vary significantly.

- We start with some constants:

  ```
  constexpr size_t n_samples {10 * 1000};
  constexpr size_t n_max {50};
  ```

  The `n_samples` constant is the number of samples to generate for each histogram. In this case, 10,000.

  The `n_max` constant is used as a divisor while generating the histograms.

- Our histogram function takes a distribution generator as an argument and prints a histogram for that distribution algorithm:

```
void dist_histogram(auto distro, const string_view& dist_name) {
    std::default_random_engine rng{};
    map<long, size_t> m;

    for (size_t i{}; i < n_samples; ++i)
        ++m[(long)distro(rng)];

    auto max_elm_it = max_element(m.begin(), m.end(),
            [](const auto& a, const auto& b) {
                return a.second < b.second;
            }
        );
    size_t max_elm = max_elm_it->second;
    size_t max_div = std::max(max_elm / n_max, size_t(1));

    println ("{}:", dist_name);
    for (const auto [randval, count] : m) {
        if (count < max_elm / n_max) continue;
        println("{:3}:{:*<{}}", randval, ' ', count / max_div);
    }
}
```

The `dist_histogram()` function uses a `map` to store the histogram. It then displays the histogram as a series of asterisks on the console.

- We call `dist_histogram()` from `main()` like this:

```
int main() {
    dist_histogram(
        std::uniform_int_distribution<int>{0, 9},
        "uniform_int_distribution");
    dist_histogram(
        std::normal_distribution<double>{0.0, 2.0},
        "normal_distribution");
...
```

Calling the `dist_histogram()` function is more complex than it was for the random number generators. Each random distribution class has a different set of parameters, according to its algorithm.

For the full list, see the `distribution.cpp` file in the Github archive.

Partial output:

```
uniform_int_distribution:
  0: ************************************************
  1: ************************************************
  2: ************************************************
  3: ************************************************
  4: ************************************************
  5: ************************************************
  6: ************************************************
  7: ************************************************
  8: ************************************************
  9: ************************************************
normal_distribution:
 -4: *
 -3: *****
 -2: ***********
 -1: ******************
  0: ************************************************
  1: ******************
  2: **********
  3: ****
  4:
bernoulli_distribution:
  0: ***************
  1: ************************************************
discrete_distribution:
  0: *****
  1: ***********
  2: ***********************
  3: ************************************************
```

*Figure 8.3: Screenshot of random distribution histograms*

Each of the distribution algorithms produces very different output. You will want to experiment with the different options for each random distribution generator.

## How it works

Each of the distribution generators has a functor that returns the next value in the random distribution:

```
result_type operator()( Generator& g );
```

The functor takes a random number generator (RNG) object as an argument:

```
std::default_random_engine rng{};
map<long, size_t> m;
for (size_t i{}; i < n_samples; ++i) ++m[(long)distro(rng)];
```

For our purposes, we're using the `std::default_random_engine` for our RNG.

As with the RNG histogram, this is a useful tool to visualize the various distribution algorithms available in the `random` library. You will want to experiment with the various parameters available for each algorithm.


## Get this book's PDF version and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.



UNLOCK NOW

*Note: Keep your invoice handy. Purchases made directly from Packt don't require an invoice.*

# 9

# Concurrency and Parallelism

Concurrency and parallelism refer to the ability to run code in separate *threads of execution*.

More specifically, *concurrency* is the ability to run threads in the background, and *parallelism* is the ability to run threads simultaneously in separate cores of a processor. The run-time library, along with the host operating system, will choose between concurrent and parallel execution models for a given thread on a given hardware environment.

In a modern multi-tasking operating system, the `main()` function already represents a thread of execution. When a new thread is started, it's said to be *spawned* by an existing thread. A group of threads may be called a *swarm*.

In the C++ standard library, the `std::thread` class provides the basic unit of threaded execution. Other classes build upon `thread` to provide *locks*, *mutexes*, and other concurrency patterns. Depending on system architecture, execution threads may run concurrently on one processor or in parallel on separate cores.

In this chapter, we will cover these tools and more, in the following recipes:

- Sleep a process for a specific amount of time
- Use `std::thread` for concurrency
- Use `std::async` for concurrency
- Run STL algorithms in parallel with execution policies
- Use `mutex` and `lock` to safely share data
- Share flags and values with `std::atomic`
- Initialize threads with `std::call_once`
- Resolve the producer-consumer problem with `std::condition_variable`

- Implement multiple producers and consumers
- Suspend and resume execution with coroutines
- Use coroutines for a producer-consumer solution

# Sleep a process for a specific amount of time

The `<thread>` header provides three functions for putting a thread to sleep: `sleep()`, `sleep_for()`, and `sleep_until()`. These functions are in the `std::this_thread` namespace.

This recipe explores the use of these functions, as we will be using them later in this chapter.

## How to do it

Let's look at how to use the `sleep_for()` and `sleep_until()` functions.

- The sleep-related functions are in the `std::this_thread` namespace. Because it has just a few symbols, we'll go ahead and issue `using` directives for `std::this_thread` and `std::chrono_literals`:

  ```
  using namespace std::this_thread;
  using namespace std::chrono_literals;
  ```

  The `chrono_literals` namespace has symbols for representing durations, like `1s` for one second, or `100ms` for 100 milliseconds.

- In `main()`, we'll mark a point in time with `steady_clock::now()` so we can time our test:

  ```
  int main() {
      auto t1 = steady_clock::now();
      println("sleep for 1.3 seconds");
      sleep_for(1s + 300ms);

      println("sleep for 2 seconds");
      sleep_until(steady_clock::now() + 2s);

      duration<double> dur1 = steady_clock::now() - t1;
      println("total duration: {:.5}s", dur1.count());
  }
  ```

The `sleep_for()` function takes a duration object to specify the amount of time to sleep. The argument `(1s + 300ms)` uses `chrono_literal` operators to return a duration object representing 1.3 seconds.

The `sleep_until()` function takes a `time_point` object to specify a specific time to resume from sleep. In this case, the `chrono_literal` operators are used to modify the `time_point` object returned from `steady_clock::now()`.

This is our output:

```
sleep for 1.3 seconds
sleep for 2 seconds
total duration: 3.3004s
```

Your results will vary based on many factors, including system hardware, OS, and scheduling of other resources.

## How it works

The `sleep_for(duration)` and `sleep_until(time_point)` functions suspend execution of the current thread for the specified duration or until the `time_point` is reached, respectively.

The `sleep_for()` function will use the `steady_clock` implementation, if supported. Otherwise, the duration may be subject to time adjustments. Both functions may block for longer due to scheduling or resource delays.

## There's more...

Some systems support a POSIX function `sleep()`, which suspends execution for the number of seconds specified:

```
unsigned int sleep(unsigned int seconds);
```

The `sleep()` function is part of the POSIX C standard and is not part of the C++ standard. If available, POSIX `sleep()` can be found in the `unistd.h` header.

# Use std::thread for concurrency

A *thread* is a unit of concurrent execution. The `main()` function may be thought of as the *main thread of execution*. Within the context of the operating system, the main thread runs concurrently with other threads owned by other processes.

The `std::thread` class is the primary interface for creating threads in the C++ Standard Library. In this recipe we will examine the basics of `std::thread`, and how `join()` and `detach()` determine its execution context.

## How to do it

In this recipe we will create some threads using `std::thread`, and experiment with their execution options.

- We'll start with a convenience function for sleeping a thread, in milliseconds:

```
void sleepms(const unsigned ms) {
    using std::chrono::milliseconds;
    std::this_thread::sleep_for(milliseconds(ms));
}
```

  The `sleep_for()` function takes a `duration` object and blocks execution of the current thread for the specified duration. This `sleepms()` function serves as a convenience wrapper that takes an `unsigned` value for the number of milliseconds to sleep.

- Now, we need a function for our thread. This function sleeps for a variable number of milliseconds, based on an integer parameter:

```
void fthread(const int n) {
    println("This is t{}", n);

    for (size_t i {}; i < 5; ++i) {
        sleepms(100 * n);
        println("t{}: {}", n, i + 1);
    }
    println("Finishing t{}", n);
}
```

  `fthread()` calls `sleepms()` five times, sleeping each time for 100 * n milliseconds.

- We can run this in a separate thread with `std::thread` from `main()`:

```
int main() {
    thread t1(fthread, 1);
    println("end of main()");
}
```

This code compiles but we get this error when we run it:

```
terminate called without an active exception
Aborted
```

(Your error message will vary. This is the error message on Debian with GCC.)

The problem is that the operating system doesn't know what to do with the thread object when it goes out of scope. We must specify if the caller waits for the thread, or if it's detached and runs independently.

- We use the `join()` method to indicate that the caller will wait for the thread to finish:

```
int main() {
    thread t1(fthread, 1);
    t1.join();
    println("end of main()");
}
```

Output:

```
This is t1
t1: 1
t1: 2
t1: 3
t1: 4
t1: 5
Finishing t1
end of main()
```

Now `main()` waits for the thread to finish.

- If we call `detach()` instead of `join()`, then `main()` doesn't wait and the program ends before the thread can run:

```
thread t1(fthread, 1);
t1.detach();
```

Output:

```
end of main()
```

When the thread is detached, we need to give it time to run:

```
thread t1(fthread, 1);
t1.detach();
println("main() sleep 2 sec");
sleepms(2000);
```

Output:

```
main() sleep 2 sec
This is t1
t1: 1
t1: 2
t1: 3
t1: 4
t1: 5
Finishing t1
end of main()
```

- Let's start and detach a second thread and see what happens:

```
int main() {
    thread t1(fthread, 1);
    thread t2(fthread, 2);
    t1.detach();
    t2.detach();
    println("main() sleep 2 sec");
    sleepms(2000);
    println("end of main()");
}
```

Output:

```
main() sleep 2 sec
This is t1
This is t2
t1: 1
t2: 1
t1: 2
t1: 3
t2: 2
```

```
t1: 4
t1: 5
Finishing t1
t2: 3
t2: 4
t2: 5
Finishing t2
end of main()
```

Because our `fthread()` function uses its parameter as a multiplier for `sleepms()`, the second thread runs a bit slower than the first. We can see the timers interlaced in the output.

- If we do this with `join()` instead of `detatch()` we get a similar result:

```
int main() {
    thread t1(fthread, 1);
    thread t2(fthread, 2);
    t1.join();
    t2.join();
    println("end of main()");
}
```

Output:

```
This is t1
This is t2
t1: 1
t2: 1
t1: 2
t1: 3
t2: 2
t1: 4
t1: 5
Finishing t1
t2: 3
t2: 4
t2: 5
Finishing t2
end of main()
```

Because `join()` waits for the thread to finish, we no longer need the 2-second `sleepms()` in `main()` to wait for the threads to finish.

## How it works

A `std::thread` object represents a thread of execution. There is a 1:1 relationship between the object and thread. One thread object represents one thread, and one thread is represented by one thread object. A thread object cannot be copied or assigned but it can be moved.

The `thread` constructor looks like this:

```
explicit thread(Function&&f,Args&&...args);
```

A `thread` is constructed with a function pointer and zero or more arguments. The function is called immediately with the arguments provided:

```
thread t1(fthread, 1);
```

This creates the object `t1` and immediately calls the function `fthread(int)` with the literal value `1` as the argument.

After creating the `thread`, we must use either `join()` or `detach()` on the thread:

```
t1.join();
```

The `join()` method blocks execution of the calling thread until the `t1` thread has completed.

```
t1.detach();
```

The `detach()` method allows the calling thread to continue independently of the t1 thread.

## There's more...

C++20 provides `std::jthread`, which automatically joins the caller at the end of its scope.

```
int main() {
    std::jthread t1(fthread, 1);
    println("end of main()");
}
```

Output:

```
end of main()
This is t1
t1: 1
t1: 2
t1: 3
t1: 4
t1: 5
Finishing t1
```

`jthread` is a thread type that automatically joins on destruction. This allows both the `t1` and `main()` threads to execute independently without an explicit `join()` and without accidentally terminating. This allows a safe, structured shutdown and simpler thread management.

Of course, this works just as safely with multiple threads:

```
int main() {
    std::jthread t1(fthread, 1);
    std::jthread t2(fthread, 2);
    println("end of main()");
}
```

Output:

```
This is t1
This is t2
end of main()
t1: 1
t2: 1
t1: 2
t1: 3
t2: 2
t1: 4
t1: 5
Finishing t1
t2: 3
t2: 4
t2: 5
Finishing t2
```

Notice that the threads in this example run *after* the `"end of main()"` message. This is because the join occurs at the end of the scope of the `jthread()` call.

While the combined `jthread()` can be more convenient than the separate `thread` and `join()`, we lose the ability to specify when the join occurs. If you wish to control when the thread is joined, you'll want to use `thread()` and `join()` explicitly instead of `jthread()`.

# Use std::async for concurrency

`std::async()` runs a target function asynchronously and returns a `std::future` object to carry the target's return value. In this way, `async()` operates much like `std::thread` but allows return values.

Let's consider the use of `std::async()` with a few examples.

## How to do it

In its simplest forms, the `std::async()` function performs much the same task as `std::thread`, without the need to call `join()` or `detach()`, and also allows return values via a `std::future` object.

In this recipe we'll use a function that counts the number of primes in a range. We'll use `chrono::steady_clock` to time the execution of each thread.

- We start with a couple of convenient aliases:

```
using launch = std::launch;
using secs = std::chrono::duration<double>;
```

  `std::launch` has launch policy constants for use with the `async()` call. The `secs` alias is a duration class for timing our primes calculations.

- Our target function counts prime numbers in a range. This is essentially a way to understand the execution policies by eating some clock cycles.

```
struct prime_time {
    secs dur{};
    uint64_t count{};
};

prime_time count_primes(const uint64_t max) {
    prime_time ret {};
    constexpr auto isprime = [](uint64_t n) {
```

```
        if (n == 2) return true;
        if (n < 2 || n % 2 == 0) return false;
        for (uint64_t i{3}; i <= n / i; i += 2) {
            if (n % i == 0) return false;
        }
        return true;
    };
    ret.count = max >= 2 ? 1 : 0;
    uint64_t start {3};
    uint64_t end {max};
    auto t1 = steady_clock::now();
    for (uint64_t i {start}; i <= end ; ++i) {
        if (isprime(i)) ++ret.count;
    }
    ret.dur = steady_clock::now() - t1;
    return ret;
}
```

The `prime_time` structure is for the return value, with elements for `duration` and `count`. This allows us to time the loop itself. The `isprime` lambda returns `true` if a value is prime. We use `steady_clock` to calculate the duration of the loop that counts primes.

- In `main()`, we call our function and report its timing:

```
int main() {
    constexpr uint64_t max_prime {0x4FFFFF};
    auto pt = count_primes(max_prime);
    println("primes: {} {}", pt.count, pt.dur);
}
```

Output (on an Intel NUC, 1.61 GHz 6 core i7):

```
  primes: 364195 5.30279
```

- Now, we can run `count_primes()` asynchronously with `std::async()`:

```
int main() {
    constexpr uint64_t max_prime {0x4FFFFF};
    auto primes1 = async(count_primes, max_prime);
    auto pt = primes1.get();
```

```
    println("primes: {} {}", pt.count, pt.dur);
}
```

Here we call `async()` with our `count_primes` function and the `max_prime` parameter. This runs `count_primes()` in the background.

`async()` returns a `std::future` object, which carries the return value of an asynchronous operation. The future object `get()` method blocks execution until the asynchronous function has completed, then returns the `prime_time` object from the function.

This runs with roughly the same timing as we got without `async()`:

```
  primes: 364195 5.40484s
```

- The `async()` function optionally takes execution policy flags as its first parameter:

```
auto primes1 = async(launch::async, count_primes, max_prime);
```

The choices are `async` or `deferred`. These flags are in the `std::launch` namespace.

The `async` flag enables asynchronous operation, and the `deferred` flag enables lazy evaluation. These flags are bitmapped and may be combined with the bitwise or `|` operator.

The default is for both bits to be set, as if `async | deferred` was specified.

- We can run several instances of our function simultaneously with `async()`:

```
int main() {
    constexpr uint64_t max_prime {0x4FFFFF};
    constexpr size_t num_threads {15};
    list<std::future<prime_time>> swarm;

    println("start parallel primes");
    auto t1 {steady_clock::now()};
    for (size_t i {}; i < num_threads; ++i) {
        swarm.emplace_back(
          async(launch::async, count_primes, max_prime)
        );
    }
```

```
    for (auto& f : swarm) {
        static size_t i{};
        auto pt = f.get();
        println("primes({:02}): {} {:.5}",
          ++i, pt.count, pt.dur);
    }

    secs dur_total{ steady_clock::now() - t1 };
    println("total duration: {:.5}s", dur_total.count());
}
```

We know that async returns a `future` object. So, we can run 15 threads by storing the future objects in a container. Here's the output on a 6-core i7 running Windows:

```
start parallel primes
primes(01): 364195 4.56607s
primes(02): 364195 4.56273s
primes(03): 364195 4.55887s
primes(04): 364195 4.55954s
primes(05): 364195 5.37375s
primes(06): 364195 5.43635s
primes(07): 364195 5.34973s
primes(08): 364195 4.56051s
primes(09): 364195 4.56134s
primes(10): 364195 4.5587s
primes(11): 364195 5.40484s
primes(12): 364195 4.58699s
primes(13): 364195 5.38693s
primes(14): 364195 3.82778s
primes(15): 364195 3.82953s
total duration: 8.3893s
```

Even though the 6-core i7 cannot run all the processes in separate cores, it still completes 15 instances in about 8.4 seconds.

It looks like it finishes the first 13 threads in about 5.4 seconds and then takes another 3 seconds to finish the last 2 threads. This likely takes advantage of the processor's hyper-threading technology that allows 2 threads to run in one core under some circumstances.

When we run the same code on a 32-core Apple M1 Ultra under macOS we get this result:

```
start parallel primes
primes(01): 364195 0.476
primes(02): 364195 0.477
primes(03): 364195 0.477
primes(04): 364195 0.476
primes(05): 364195 0.477
primes(06): 364195 0.485
primes(07): 364195 0.480
primes(08): 364195 0.476
primes(09): 364195 0.496
primes(10): 364195 0.483
primes(11): 364195 0.480
primes(12): 364195 0.480
primes(13): 364195 0.476
primes(14): 364195 0.476
primes(15): 364195 0.476
total duration: 0.49679s
```

The 32-core M1 Ultra easily runs all 15 parallel processes simultaneously and gets through the set in under 500 milliseconds.

## How it works

The key to understanding `std::async` is in its use of `std::promise` and `std::future`.

The `promise` class allows a thread to store an object that may later be retrieved asynchronously by a future object.

For example, if we have a function like this:

```
void f() {
    println("this is f()");
}
```

We could run it with `std::thread` like this:

```
int main() {
    std::thread t1(f);
    t1.join();
```

```
    println("end of main()");
}
```

That works fine for a simple function with no return value. When we want to return a value from `f()`, we can use `promise` and `future`.

We pass the promise object to our function:

```
void f(std::promise<int> value) {
    println("this is f()");
    value.set_value(47);
}
```

And then use `promise` and `future` objects in the `main()` thread to pass and return values from `f()`:

```
int main() {
    std::promise<int> value_promise;
    std::future<int> value_future = value_promise.get_future();
    std::thread t1(f, std::move(value_promise));
    t1.detach();
    println("value is {}", value_future.get());
    println("end of main()");
}
```

Notice that a `promise` object cannot be copied, so we need to use `std::move` to pass it to the function.

The `promise` object serves as a bridge to a `future` object, which allows us to retrieve the value when it becomes available.

## Simplify promise and future with std::async()

`std::async()` is a helper function to simplify the creation of the `promise` and `future` objects. With `async()`, we can accomplish the same task like this:

```
int f(int value) {
    println("this is f()");
    return value;
}

int main() {
```

```
    auto value_future = async(launch::async, f, 47);
    println("value is {}", value_future.get());
    println("end of main()");
}
```

Result:

```
this is f()
value is 47
end of main()
```

That's the value of `async()`. For many purposes, it makes the use of `promise` and `future` trivially simple.

# Run STL algorithms in parallel with execution policies

Beginning with C++17, many of the standard STL algorithms can run with *parallel execution*. This feature allows an algorithm to split its work into sub-tasks to run simultaneously on multiple CPU cores. These algorithms accept an execution policy object that specifies the kind of parallelism applied to the algorithm. This feature requires hardware support.

## How to do it

Execution policies are defined in the `<execution>` header and in the `std::execution` namespace. In this recipe we will test the available policies using the `std::transform()` algorithm.

- For timing purposes, we'll use the `duration` object with the `std::milli` ratio, so we can measure in milliseconds:

  ```
  using dur_t = duration<double, std::milli>;
  ```

  For demonstration purposes, we'll start with a `vector` of `int` with 20 million random values:

  ```
  int main() {
      std::vector<unsigned> v(20'000'000);
      std::random_device rng;
      println("generate randoms");
  ```

```
    for (auto &i : v) i = rng() % 0xFFFF'FFFF;
    ...
```

Now we apply a simple transformation:

```
auto mul2 = [](int n) {return n * 2;};
auto t0 = steady_clock::now();
std::transform(v.begin(), v.end(), v.begin(), mul2);
dur_t dur0 = steady_clock::now() - t0;
println("no policy: {:.3}ms", dur0.count());
```

The `mul2` lambda simply multiplies a value by 2. The `transform()` algorithm applies `mul2` to every member of the `vector`.

This transformation does not specify an execution policy.

Output:

```
no policy: 8.74ms
```

- We can specify an execution policy in the first argument of the algorithm:

```
std::transform(execution::seq,
    v.begin(), v.end(), v.begin(), mul2);
```

The `seq` policy means that the algorithm shall not be parallelized. This is the same as no execution policy.

Output:

```
execution::seq: 10.9ms
```

Notice that the duration is somewhat slower than without a policy. This is because the `seq` policy may inhibit inlining and other compiler optimizations.

- The `execution::par` policy allows the algorithm to parallelize its workload:

```
std::transform(execution::par,
    v.begin(), v.end(), v.begin(), mul2);
println("execution::par: {:.3}ms", dur2.count());
```

Output:

```
  execution::par: 6.2ms
```

The algorithm runs somewhat faster with the parallel execution policy, which allows the implementation to share its work among different processor cores.

The `execution::par_unseq` policy allows unsequenced parallel execution of the workload:

```
std::transform(execution::par_unseq,
    v.begin(), v.end(), v.begin(), mul2);
println("execution::par_unseq: {:.3}ms", dur3.count());
```

Output:

```
  execution::par_unseq: 5.99ms
```

Here, we notice another increase in performance with this policy.

The `execution::par_unseq` policy allows the implementation greater flexibility in assigning work to the different cores. However, it also imposes stricter requirements of the algorithm. The algorithm must not perform operations that rely on ordering or synchronized behavior.

## How it works

The execution policies interface doesn't specify how the algorithm workloads are parallelized. It's designed to work with a diverse set of hardware and processors under varying loads and circumstances. It may be implemented entirely in the library or may rely on compiler or hardware support.

Parallelization will show the most improvement on algorithms that do more than *O(n)* work. For example, `sort()` shows a dramatic improvement. Here's a `sort()` with no parallelization:

```
auto t0 = steady_clock::now();
std::sort(v.begin(), v.end());
dur_t dur0 = steady_clock::now() - t0;
println("sort: {:.3}ms ", dur0.count());
```

Output:

```
sort: 835ms
```

With `execution::par` we see significant performance gains:

```
std::sort(execution::par, v.begin(), v.end());
```

Output:

```
sort: 158ms
```

The improvement with `execution::par_unseq` is incrementally better:

```
std::sort(execution::par_unseq, v.begin(), v.end());
```

Output:

```
sort: 152ms
```

It's a good idea to do a lot of testing when using the parallelized algorithms. If your algorithm or predicates do not lend themselves well to parallelization, you may end up with minimal performance gains or unintended side-effects.

**Note**

At the time of writing, execution policies are poorly supported by GCC; and not yet supported by Apple Clang. This recipe was tested using MSVC on a 6-core Intel i7.

# Use mutex and lock to safely share data

The term *mutex* refers to *mutually exclusive* access to shared resources. A mutex is commonly used to avoid data corruption and race conditions, due to multiple threads of execution attempting to access the same data. A mutex will typically use *locks* to restrict access to one thread at a time.

The STL provides `mutex` and `lock` classes in the `<mutex>` header.

## How to do it

In this recipe we will use a simple `Animal` class to experiment with *locking* and *unlocking* a `mutex`.

- We start with creating a `mutex` object

  ```
  std::mutex animal_mutex {};
  ```

The mutex is declared in global scope so it's accessible to all the relevant objects.

- Our Animal class has a name and a list of friends:

```
class Animal {
    using friend_t = list<Animal>;
    string_view s_name {"unk"};
    friend_t l_friends {};
public:
    Animal() = delete;
    Animal(const string_view n) : s_name {n} {}
    ...
}
```

Adding and deleting friends will be a useful test case for our `mutex`.

- The equality operator is the only operator we'll need:

```
bool operator==(const Animal& o) const {
    return s_name.data() == o.s_name.data();
}
```

The `s_name` member is a `string_view`, so we can test the address of its data store for equality. This is a simple and useful shortcut.

- The `is_friend()` method tests if another `Animal` is in the `l_friends` list:

```
bool is_friend(const Animal& o) const {
    for (const auto& a : l_friends) {
        if (a == o) return true;
    }
    return false;
}
```

- The `find_friend()` method returns an optional, with an iterator to the `Animal` if found:

```
optional<friend_t::iterator>
find_friend(const Animal& o) noexcept {
    for (auto it {l_friends.begin()};
      it != l_friends.end(); ++it) {
        if (*it == o) return it;
```

```
    }
    return {};
}
```

- The `display()` method prints `s_name` along with the names of each of the `Animal` objects in the `l_friends` list:

```
void display() const noexcept {
    auto n_animals {l_friends.size()};
    print("Animal: {}, friends: ", s_name);
    if (!n_animals) print("none");
    else {
        for (auto n : l_friends) {
            print("{}", n.s_name);
            if (--n_animals) print(", ");
        }
    }
    print("\n");
}
```

- The `add_friend()` method adds an `Animal` object to the `l_friends` list:

```
bool add_friend(Animal& o) noexcept {
    println("add_friend {} -> {}", s_name, o.s_name);
    if (*this == o) return false;
    if (!is_friend(o)) {
        l_friends.emplace_back(o);
    }
    if (!o.is_friend(*this)) {
        o.l_friends.emplace_back(*this);
    }

    return true;
}
```

- The `delete_friend()` method removes an `Animal` object from the `l_friends` list:

```
bool delete_friend(Animal& o) noexcept {
    println("delete_friend {} -> {}", s_name, o.s_name);
    if (*this == o) return false;
```

```
    if (auto it = find_friend(o)) {
        l_friends.erase(it.value());
    }
    if (auto it = o.find_friend(*this)) {
        o.l_friends.erase(it.value());
    }

    return true;
}
```

- In the `main()` function we create some `Animal` objects:

```
int main() {
    auto cat1 = std::make_unique<Animal>("Felix");
    auto tiger1 = std::make_unique<Animal>("Hobbes");
    auto dog1 = std::make_unique<Animal>("Astro");
    auto rabbit1 = std::make_unique<Animal>("Bugs");
    ...
```

- We call `add_friends()` on our objects with `async()`, to run them in separate threads:

```
auto a1 = std::async([&]{cat1->add_friend(*tiger1); });
auto a2 = std::async([&]{cat1->add_friend(*rabbit1); });
auto a3 = std::async([&]{rabbit1->add_friend(*dog1); });
auto a4 = std::async([&]{rabbit1->add_friend(*cat1); });
a1.wait();
a2.wait();
a3.wait();
a4.wait();
```

  We call `wait()` to allow our threads to complete before continuing.

- We create a lambda with `async()` to call `display()` with `wait()` to see our `Animal` objects and their relationships:

```
auto p1 = std::async([&]{cat1->display(); });
auto p2 = std::async([&]{tiger1->display(); });
auto p3 = std::async([&]{dog1->display(); });
auto p4 = std::async([&]{rabbit1->display(); });
p1.wait();
p2.wait();
```

```
p3.wait();
p4.wait();
```

- And finally, we call `delete_friend()` to remove one of our relationships:

```
auto a5 = std::async([&]{
    cat1->delete_friend(*rabbit1);
});
a5.wait();
auto p5 = std::async([&]{cat1->display(); });
auto p6 = std::async([&]{rabbit1->display(); });
```

- At this point, our output looks like this:

```
add_friend Bugs -> Felix
add_friend Felix -> Hobbes
add_friend Felix -> Bugs
add_friend Bugs -> Astro
Animal: Felix, friends: Bugs, Hobbes
Animal: Hobbes, friends: Animal: Bugs, friends: FelixAnimal: Astro,
friends: Felix
, Astro
Bugs
delete_friend Felix -> Bugs
Animal: Felix, friends: Hobbes
Animal: Bugs, friends: Astro
```

This output is somewhat scrambled. It will be different each time you run it. It may be fine sometimes, but don't let that fool you. We need to add some `mutex` locks to control access to the data.

- One way to use `mutex` is with its `lock()` and `unlock()` methods. Let's add them to the `add_friend()` function:

```
bool add_friend(Animal& o) noexcept {
    println("add_friend {} -> {}", s_name, o.s_name);
    if (*this == o) return false;
    animal_mutex.lock();
    if (!is_friend(o)) {
        l_friends.emplace_back(o);
    }
```

```
    if (!o.is_friend(*this)) {
        o.l_friends.emplace_back(*this);
    }
    animal_mutex.unlock();
    return true;
}
```

The `lock()` method attempts to acquire a lock on the `mutex`. If the `mutex` is already locked, it will wait (*block execution*) until the `mutex` is unlocked.

- We also need to add a lock to `delete_friend()`:

```
bool delete_friend(Animal& o) noexcept {
    println("delete_friend {} -> {}", s_name, o.s_name);
    if (*this == o) return false;
    animal_mutex.lock();
    if (auto it = find_friend(o)) {
        l_friends.erase(it.value());
    }
    if (auto it = o.find_friend(*this)) {
        o.l_friends.erase(it.value());
    }
    animal_mutex.unlock();
    return true;
}
```

- And we need to add a lock to `display()` so that data is not changed while printing the output:

```
void display() const noexcept {
    animal_mutex.lock();
    auto n_animals {l_friends.size()};
    print("Animal: {}, friends: ", s_name);
    if (!n_animals) print("none");
    else {
        for (auto n : l_friends) {
            print("{}", n.s_name);
            if (--n_animals) print(", ");
        }
    }
    print("\n");
```

```
    animal_mutex.unlock();
}
```

Now our output is sensible:

```
add_friend Bugs -> Felix
add_friend Bugs -> Astro
add_friend Felix -> Hobbes
add_friend Felix -> Bugs
Animal: Felix, friends: Bugs, Hobbes
Animal: Hobbes, friends: Felix
Animal: Astro, friends: Bugs
Animal: Bugs, friends: Felix, Astro
delete_friend Felix -> Bugs
Animal: Felix, friends: Hobbes
Animal: Bugs, friends: Astro
```

Your output may have the lines in a different order due to asynchronous operation but the details should be the same.

- The `lock()` and `unlock()` methods are rarely called directly. The `std::lock_guard` class manages locks with a proper RAII (*Resource Acquisition Is Initialization*) pattern that automatically releases the lock upon destruction. Here's the `add_friend()` method with `lock_guard`:

```
bool add_friend(Animal& o) noexcept {
    println("add_friend {} -> {}", s_name, o.s_name);
    if (*this == o) return false;

    std::lock_guard<std::mutex> l(animal_mutex);
    if (!is_friend(o)) {
        l_friends.emplace_back(o);
    }
    if (!o.is_friend(*this)) {
        o.l_friends.emplace_back(*this);
    }
    return true;
}
```

The `lock_guard` object is created and holds a lock until it is destroyed. Like the `lock()` method, `lock_guard` also blocks until a lock is available.

- Let's apply `lock_guard` to the `delete_friend()` and `display()` methods
  Here is `delete_friend()`:

```
bool delete_friend(Animal& o) noexcept {
    println("delete_friend {} -> {}", s_name, o.s_name);
    if (*this == o) return false;
    std::lock_guard<std::mutex> l(animal_mutex);
    if (auto it = find_friend(o)) {
        l_friends.erase(it.value());
    }
    if (auto it = o.find_friend(*this)) {
        o.l_friends.erase(it.value());
    }

    return true;
}
```

And here is `display()`:

```
void display() const noexcept {
    std::lock_guard<std::mutex> l(animal_mutex);
    auto n_animals {l_friends.size()};
    print("Animal: {}, friends: ", s_name);
    if (!n_animals) print("none");
    else {
        for (auto n : l_friends) {
            print("{}", n.s_name);
            if (--n_animals) print(", ");
        }
    }
    print("\n");
}
```

Our output remains coherent:

```
add_friend Felix -> Hobbes
add_friend Bugs -> Astro
add_friend Felix -> Bugs
```

```
add_friend Bugs -> Felix
Animal: Felix, friends: Bugs, Hobbes
Animal: Astro, friends: Bugs
Animal: Hobbes, friends: Felix
Animal: Bugs, friends: Astro, Felix
delete_friend Felix -> Bugs
Animal: Felix, friends: Hobbes
Animal: Bugs, friends: Astro
```

As before, your output may have the lines in a different order due to asynchronous operation.

## How it works

It's important to understand that a `mutex` does not lock data; it blocks execution. As shown in this recipe, when a `mutex` is applied in object methods it can be used to *enforce* mutually exclusive access to data.

When one thread locks a `mutex`, with either `lock()` or `lock_guard`, that thread is said to *own* the `mutex`. Any other thread that tries to lock the same `mutex` will be blocked until it is unlocked by the owner.

The `mutex` object must not be destroyed while it's owned by any thread. Likewise, a thread must not be destroyed while it owns a mutex. An RAII-compliant wrapper like `lock_guard` will help ensure this doesn't happen.

## There's more...

While `std::mutex` provides an exclusive `mutex` suitable for many purposes, the STL does provide a few other choices.

- `shared_mutex` allows more than one thread to simultaneously own a `mutex`.
- `recursive_mutex` allows one thread to stack multiple locks on a single `mutex`.
- `timed_mutex` provides a timeout for `mutex` blocks. Both `shared_mutex` and `recursive_mutex` also have timed versions available.

# Share flags and values with std::atomic

The `std::atomic` class encapsulates a single object and guarantees it to be *atomic*, such that concurrent access to that object does not cause a race condition. Memory order semantics may also be specified, controlling how operations on different threads are ordered and made visible. It is typically used for low-level synchronization and lock-free communication among threads.

`std::atomic` defines an *atomic type* from its `template` type. The type must be *trivial*. A type is trivial if it occupies contiguous memory, has no user-defined constructor, and has no virtual member functions. All primitive types are trivial.

While it is possible to construct a trivial type, `std::atomic` is most often used with simple primitives, such as `bool`, `int`, `long`, `float`, and `double`.

## How to do it

This recipe uses a simple function that loops over a counter to demonstrate sharing atomic objects. We will spawn a swarm of these loops as threads that share atomic values.

- Atomic objects are often placed in the global namespace. They must be accessible to all the threads that need to share their value:

```
std::atomic<bool> ready {};
std::atomic<uint64_t> g_count {};
std::atomic_flag winner {};
```

  The ready object is a `bool` that gets set `true` when all the threads are ready to start counting.

  The `g_count` object is a global counter. It is incremented by each of the threads.

  The `winner` object is a special `atomic_flag` type. It is used to indicate which thread finishes first.

- We use a couple of constants to control the number of threads and the number of loops for each thread:

```
constexpr int max_count {1000};
constexpr int max_threads {100};
```

  I've set it to run 100 threads and count 1,000 iterations in each thread.

- The `countem()` function is spawned for each thread. It loops `max_count` times and increments `g_count` for each iteration of the loop. This is where we use our atomic values:

```
void countem (int id) {
    while (!ready) std::this_thread::yield();
    for (int i {}; i < max_count; ++i) ++g_count;
    if (!winner.test_and_set()) {
```

```
        println("thread {:02} won!", id);
    }
};
```

The `ready` atomic value is used to synchronize the threads. Each thread will call `yield()` until the ready value is set true. The `yield()` function yields execution to other threads.

Each iteration of the `for` loop increments the `g_count` atomic value. The final value should be equal to `max_count * max_threads`.

After the loop is complete, the `test_and_set()` method of the winner object is used to report the winning thread. `test_and_set()` is a method of the `atomic_flag` class. It sets the flag and returns the boolean value from before it is set.

- We've used the `make_commas()` function before. It displays a number with thousands separators:

```
string make_commas(const uint64_t num) {
    auto s = std::to_string(num);
    for (auto i = s.size(); i > 3; i -= 3) {
        s.insert(i - 3, ",");
    }
    return s;
}
```

The `main()` function spawns the threads and reports the results:

```
int main() {
    vector<std::thread> swarm;
    println("spawn {} threads", max_threads);
    for (int i {}; i < max_threads; ++i) {
        swarm.emplace_back(countem, i);
    }
    ready = true;
    for (auto&t:swarm)t.join();
    println("global count: {}", make_commas(g_count));
}
```

Here we create a `vector<std::thread>` object to hold the threads.

In the for loop, we use `emplace_back()` to create each thread in the `vector`.

Once the threads have been spawned, we set the `ready` flag so the threads may start their loops.

Output:

```
spawn 100 threads
thread 67 won!
global count: 100,000
Every time you run it a different thread will win.
```

## How it works

The `std::atomic` class encapsulates an object to synchronize access among multiple threads.

The encapsulated object must be a *trivial type*, which means it occupies contiguous memory, has no user-defined constructor, and has no virtual member functions. All primitive types are trivial.

It is possible to use a simple struct with atomic:

```
struct Trivial {
    int a;
    int b;
};
std::atomic<Trivial> triv1;
```

While this usage is possible, it's not practical. Anything beyond setting and retrieving compound values loses the benefits of atomicity and ends up requiring a *mutex*. The `atomic` class is best suited for *scalar* values.

## Specializations

There are specializations of the atomic class for a few different purposes:

1. Pointers and smart pointers
2. Floating point types
3. Integral types

## Pointers and smart pointers

The `std::atomic<U*>` specialization includes support for atomic pointer arithmetic operations, including `fetch_add()` for addition and `fetch_sub()` for subtraction.

## Floating-point types

When used with the floating-point types `float`, `double`, and `long double`, `std::atomic` includes support for atomic floating-point arithmetic operations, including `fetch_add()` for addition and `fetch_sub()` for subtraction.

## Integral types

When used with one of the integral types, `std::atomic` provides support for additional atomic operations, including `fetch_add()`, `fetch_sub()`, `fetch_and()`, `fetch_or()`, and `fetch_xor()`.

# Standard aliases

The STL provides type aliases for all the standard scalar integral types. This means that instead of these declarations in our code:

```
std::atomic<bool> ready{};
std::atomic<uint64_t> g_count{};
We could use:
std::atomic_bool ready{};
std::atomic_uint64_t g_count{};
```

There are 46 standard aliases, one for each of the standard integral types:

| | |
|---|---|
| `atomic_bool` | `atomic_uint64_t` |
| `atomic_char` | `atomic_int_least8_t` |
| `atomic_schar` | `atomic_uint_least8_t` |
| `atomic_uchar` | `atomic_int_least16_t` |
| `atomic_short` | `atomic_uint_least16_t` |
| `atomic_ushort` | `atomic_int_least32_t` |
| `atomic_int` | `atomic_uint_least32_t` |
| `atomic_uint` | `atomic_int_least64_t` |
| `atomic_long` | `atomic_uint_least64_t` |

| | |
|---|---|
| `atomic_ulong` | `atomic_int_fast8_t` |
| `atomic_llong` | `atomic_uint_fast8_t` |
| `atomic_ullong` | `atomic_int_fast16_t` |
| `atomic_char8_t` | `atomic_uint_fast16_t` |
| `atomic_char16_t` | `atomic_int_fast32_t` |
| `atomic_char32_t` | `atomic_uint_fast32_t` |
| `atomic_wchar_t` | `atomic_int_fast64_t` |
| `atomic_int8_t` | `atomic_uint_fast64_t` |
| `atomic_uint8_t` | `atomic_intptr_t` |
| `atomic_int16_t` | `atomic_uintptr_t` |
| `atomic_uint16_t` | `atomic_size_t` |
| `atomic_int32_t` | `atomic_ptrdiff_t` |
| `atomic_uint32_t` | `atomic_intmax_t` |
| `atomic_int64_t` | `atomic_uintmax_t` |

## Lock-free variations

Most modern architectures provide *atomic CPU instructions* for performing atomic operations. `std::atomic` should use hardware support for atomic instructions, where supported by your hardware. Some atomic types may not be supported on some hardware. `std::atomic` may use a *mutex* to ensure thread-safe operations for those specializations, causing threads to block while waiting for other threads to complete operations. Specializations that use hardware support are said to be *lock-free* because they don't require a mutex.

The `is_lock_free()` method checks whether a specialization is lock-free:

```
println("is g_count lock-free? {}",g_count.is_lock_free());
```

Output:

```
is g_count lock-free? true
```

This result will be `true` for most modern architectures.

There are a few guaranteed lock-free variations of `std::atomic` available. These specializations guarantee the use of the most efficient hardware atomic operations for each purpose:

`std::atomic_signed_lock_free` is an alias for the most efficient lock-free specialization of a signed integral type.

`std::atomic_unsigned_lock_free` is an alias for the most efficient lock-free specialization of an unsigned integral type.

The `std::atomic_flag` class provides a lock-free atomic boolean type.

**Note**

In my tests, I found significant performance differences between `std::atomic<uint64_t>` and `std::atomic<uint32_t>` on Windows, where the 64-bit atomic was over 10x slower than the 32-bit atomic, even when compiled for the 64-bit processor. I did not see the same performance hits on Linux or MacOS.

On some platforms, notably Windows, 64-bit atomic operations are implemented more conservatively than 32-bit atomics. Though both are lock-free, stronger ordering guarantees are used for 64-bit atomics which can make them significantly slower under heavy contention.

## There's more...

When multiple threads read and write variables simultaneously, one thread may observe the changes in a different order than they were written. `std::memory_order` specifies how memory access is ordered around an atomic operation.

`std::atomic` provides methods for accessing and changing its managed value. Unlike the associated operators, these access methods provide for a memory order to be specified. For example:

```
g_count.fetch_add(1, std::memory_order_seq_cst);
```

In this case, `memory_order_seq_cst` specifies *sequentially consistent* ordering. So, this call to `fectch_add()` will add 1 to the value of `g_count` with sequentially consistent ordering.

Other possible memory order constants are:

- `memory_order_relaxed`
  Relaxed operation: no synchronization or ordering constraints imposed, only atomicity is guaranteed.
- `memory_order_consume`
  This is a *consume operation*. Access in the current thread dependent on the value cannot be reordered before this load.
- `memory_order_acquire`
  This is an *acquire operation*. Access cannot be reordered before this load.
- `memory_order_release`
  This is a *store operation*. Access in the current thread cannot be reordered after this store.
- `memory_order_acq_rel`
  This is both *acquire and release*. Access in the current thread cannot be reordered before or after this store.
- `memory_order_seq_cst`
  This is *sequentially consistent* ordering, either *acquire or release*, depending on the context. Load performs acquire, store performs release, and read/write/modify performs both. All threads observe all modifications in the same order.

If memory order is not specified, `memory_order_seq_cst` is the default.

# Initialize threads with std::call_once

You may need to run the same code in many threads but must initialize that code only once.

One solution would be to call the initialization code before running the threads. This approach can work but has some drawbacks. By separating the initialization, it may be called when unnecessary, or it may be missed when necessary.

The `std::call_once` function provides a more robust solution. `call_once` is in the `<mutex>` header.

## How to do it

In this recipe we use a print function for the initialization, so we can clearly see when it's called.

- We'll use a constant for the number of threads to spawn:

```
constexpr size_t max_threads {25};
```

- We also need a `std::once_flag` to synchronize the `std::call_once` function:

```
std::once_flag init_flag;
```

  The `std::once_flag` is in static memory and global scope.

- Our initialization function simply prints a string to let us know it's been called:

```
void do_init(size_t id) {
    print("do_init ({}): ", id);
}
```

- Our worker function, `do_print()`, uses `std::call_once` to call the initialization function, then prints its own id:

```
void do_print(size_t id) {
    std::call_once(init_flag, do_init, id);
    print("{} ", id);
}
```

  In `main()`, we use a `list` container to manage a set of `jthread` objects:

```
int main() {
    list<jthread> spawn;
    for (size_t id{}; id < max_threads; ++id) {
        spawn.emplace_back(do_print, id);
    }
}
```

Our output shows the initialization happens first, and once:

```
do_init (8): 12 0 2 1 9 6 13 10 11 5 16 3 4 17 7 15 8 14 18 19 20 21 22 23
24
```

Notice that it's not always the first spawned thread (`0`) that ends up calling the initialization function, but it is always called first. If you run this repeatedly, you'll see thread `0` gets the initialization often, but not every time. You'll see thread `0` in the initialization more often on a system with fewer cores.

## How it works

`std::call_once` is a template function that takes a flag, a *callable* (function or functor), and a parameter pack of arguments:

```
template<class Callable, class... Args>
void call_once(once_flag&flag,Callable&&f,Args&&...args);
```

The callable `f` is called exactly one time. Even if `call_once` is called concurrently from several threads, `f` is still called once and only once.

This requires a `std::once_flag` object for coordination. The `once_flag` constructor sets its state to indicate that the callable has not yet been called.

When `call_once` invokes the callable, any other calls on the same `once_flag` are blocked until the callable returns. After the callable returns, the `once_flag` is set and any subsequent calls to `call_once` return without invoking `f`.

# Resolve the producer-consumer problem with std::condition_variable

The simplest version of the *producer-consumer problem* is where you have one process that *produces* data, and another that *consumes* data, using one *buffer* or container to hold the data. This requires coordination between the producer and consumer, to manage the buffer and prevent unwanted side-effects.

## How to do it

In this recipe we consider a simple solution to the producer-consumer problem using `std::condition_variable` to coordinate the processes.

- We begin with some namespace and alias declarations for convenience:

```
using namespace std::chrono_literals;
namespace this_thread = std::this_thread;

using guard_t = std::lock_guard<std::mutex>;
using lock_t = std::unique_lock<std::mutex>;
```

  The `guard_t` and `lock_t` aliases make it easier to use `lock_guard` and `unique_lock` without error.

- We use a couple of constants:

```
constexpr size_t num_items {10};
constexpr auto delay_time {200ms};
```

  Keeping these in one place makes it safer and easier to experiment with different values.

- We're using these global variables to coordinate the data store:

```
std::deque<size_t> q {};
std::mutex mtx {};
std::condition_variable cond {};
bool finished {};
```

  We're using `deque` to hold the data as a first-in-first-out (FIFO) queue.

  The `mutex` is used with the `condition_variable` to coordinate the movement of data from producer to consumer.

  The `finished` flag indicates that there is no more data.

- The producer thread will use this function:

```
void producer() {
    for (size_t i {}; i < num_items; ++i) {
        this_thread::sleep_for(delay_time);
```

```
            guard_t x {mtx};
            q.push_back(i);
            cond.notify_all();
        }

        guard_t x {mtx};
        finished = true;
        cond.notify_all();
    }
```

The `producer()` function loops `num_items` iterations and pushes a number onto the `deque` each time through the loop.

We include a `sleep_for()` call to simulate a delay in producing each value.

The `conditional_variable` requires a `mutex` lock to operate. We use `lock_guard` (via the `guard_t` alias) to obtain the lock, then push the value onto the `deque`, then call `notify_all()` on the `conditional_variable`. This tells the consumer thread that there is a new value available.

When the loop completes, we set the `finished` flag and notify the consumer thread that the producer is completed.

- The consumer thread waits for each value from the producer, displays it on the console, and waits for the finished flag

```
void consumer() {
    while(!finished) {
        lock_t lck {mtx};
        cond.wait(lck, []{
            return !q.empty() || finished;
        });
        while (!q.empty()) {
            println("Got {} from the queue", q.front());
            q.pop_front();
        }
    }
    println("finished!");
}
```

The `wait()` method waits to be notified by the producer. It uses a lambda as a predicate to continue waiting until the `deque` is *not empty* or the `finished` flag is set.

When we get a value, we display it and then pop it from the deque.

- We run this in main with simple thread objects:

```
int main() {
    jthread t1 {producer};
    jthread t2 {consumer};
}
```

Output:

```
Got 0 from the queue
Got 1 from the queue
Got 2 from the queue
Got 3 from the queue
Got 4 from the queue
Got 5 from the queue
Got 6 from the queue
Got 7 from the queue
Got 8 from the queue
Got 9 from the queue
finished!
```

Notice that there's a 200ms delay between each line. This tells us that the producer-consumer coordination is working as expected.

## How it works

The producer-consumer problem requires coordination between writing and reading a buffer or container. In this example, our container is a `deque<size_t>`:

```
std::deque<size_t> q {};
```

The `condition_variable` class can block a thread, or multiple threads, while a shared variable is modified. It may then notify other threads that the value is available.

`condition_variable` requires a mutex to perform the lock:

```
std::lock_guard x {mtx};
q.push_back(i);
cond.notify_all();
```

`std::lock_guard` acquires a lock so we can push a value onto our deque.

The `wait()` method on `condition_variable` is used to block the current thread until it receives a notification:

```
void wait( std::unique_lock<std::mutex>&lock);
void wait( std::unique_lock<std::mutex>& lock,
  Pred stop_waiting );
```

The predicate form of `wait()` is equivalent to:

```
while (!stop_waiting()) {
    wait(lock);
}
```

The predicate form is used to prevent spurious waking while waiting for a specific condition. We use it with a lambda in our example:

```
cond.wait(lck, []{ return !q.empty() || finished; });
```

This prevents the consumer from waking until the `deque` has data or the `finished` flag is set.

The `condition_variable` class has two notification methods:

- `notify_one()` unblocks one waiting thread
- `notify_all()` unblocks all waiting threads

We used `notify_all()` in our example. Because there is only one consumer thread, either notification method would work the same.

**Note**

`unique_lock` is the *only* form of lock that supports the `wait()` method on a `condition_variable` object.

# Implement multiple producers and consumers

The *producer-consumer problem* is really a set of problems. Solutions will differ if the buffer is bounded or unbounded, or if there are multiple producers, multiple consumers, or both.

Let's consider a case with multiple producers, multiple consumers, and a bounded (limited capacity) buffer. This is a common condition.

## How to do it

In this recipe we'll look at a case with multiple producers and consumers, and a *bounded buffer*, using a variety of techniques we've covered in this chapter.

- We'll start with some constants for convenience and reliability:

```
constexpr auto delay_time {50ms};
constexpr auto consumer_wait {100ms};
constexpr size_t queue_limit {5};
constexpr size_t num_items {15};
constexpr size_t num_producers {3};
constexpr size_t num_consumers {5};
```

  - `delay_time` is a duration object, used with sleep_for().
  - `consumer_wait` is a duration object used with the consumer condition variable.
  - `queue_limt` is the buffer limit; the maximum number of items in the deque.
  - `num_items` is the maximum number of items produced per producer.
  - `num_producers` is the number of spawned producers.
  - `num_producers` is the number of spawned consumers.

- Now we need some objects to control the process:

```
deque<string> qs {};
mutex q_mutex {};
condition_variable cv_producer {};
condition_variable cv_consumer {};
bool production_complete {};
```

  - `qs` is a deque of string that holds the produced objects.
  - `q_mutex` controls access to the deque.
  - `cv_producer` is a condition variable that coordinates producers.
  - `cv_consumer` is a condition variable that coordinates consumers.
  - `production_complete` is set true when all producer threads have finished.

- The `producer()` threads run this function:

```
void producer(const size_t id) {
    for (size_t i {}; i < num_items; ++i) {
```

```
        this_thread::sleep_for(delay_time * id);
        unique_lock<mutex> lock(q_mutex);
        cv_producer.wait(lock, [&]{
            return qs.size() < queue_limit;
        });
        qs.push_back(format("pid {}, qs  {}, item {:02}\n",
            id, qs.size(), i + 1));
        cv_consumer.notify_all();
    }
}
```

The passed value `id` is a sequential number used to identify the producer.

The `for` loop repeats `num_item` times. The `sleep_for()` function is used to simulate work required to produce an item.

Then we obtain a `unique_lock` from `q_mutex` and invoke `wait()` on `cv_producer`, using a lambda that checks the size of the `deque` against the `queue_limit` constant. If the `deque` has reached maximum size, the producer waits for consumer threads to reduce the size of the `deque`. This represents the *bounded buffer* limit on the producer.

Once the condition is satisfied, we push an *item* onto the deque. The item is a formatted string with the producer id, the size of `qs`, and an item number (`i + 1`) from the loop control variable.

Finally, we notify the consumers that new data is available, with `notify_all()` on the `cv_consumer` condition variable.

- The `consumer()` threads run this function:

```
void consumer(const size_t id) {
    while (!production_complete) {
        unique_lock<mutex> lock(q_mutex);
        cv_consumer.wait_for(lock, consumer_wait, [&]{
            return !qs.empty();
        });
        if (!qs.empty()) {
            print("cid {}: {}", id, qs.front());
            qs.pop_front();
        }
        cv_producer.notify_all();
```

```
    }
}
```

The passed value `id` is a sequential number used to identify the consumer.

The main `while()` loop continues until `production_complete` is set.

We obtain a `unique_lock` from `q_mutex` and invoke `wait_for()` on `cv_consumer`, with a timeout and a lambda that tests if the `deque` is empty. We need the timeout because it's possible for the producer threads to finish while some of the consumer threads are still running, leaving the `deque` empty.

Once we have a non-empty `deque`, we can print (*consume*) an *item* and pop it off the `deque`.

- In `main()`, we use `async()` to spawn the producer and consumer threads. `async()` conforms to the RAII (*Resource Acquisition Is Initialization*) pattern, so I'll usually prefer it over `thread`, where possible. `async()` returns a future object, so we'll keep a list of `future<void>` objects for process management:

```
int main() {
    list<future<void>> producers;
    list<future<void>> consumers;

    for (size_t i {}; i < num_producers; ++i) {
        producers.emplace_back(async(producer, i));
    }

    for (size_t i {}; i < num_consumers; ++i) {
        consumers.emplace_back(async(consumer, i));
    }
    ...
```

We use `for` loops to create producer and consumer threads.

- Finally, we use a list of future objects to determine when our producer and consumer threads are complete:

```
for (auto& f : producers) f.wait();
production_complete = true;
println("producers done.");
```

```
for (auto& f : consumers) f.wait();
println("consumers done.");
```

We loop through our `producers` container to `wait()` for the `producer` threads to complete. Then we can set the `production_complete` flag. We likewise loop through the consumers container to `wait()` for the `consumer` threads to complete. We could perform any final analysis or completion processes here.

- The output is a bit long to show in its entirety:

```
cid 0: pid 0, qs  0, item 01
cid 0: pid 0, qs  1, item 02
cid 0: pid 0, qs  2, item 03
cid 0: pid 0, qs  3, item 04
cid 0: pid 0, qs  4, item 05
...
cid 4: pid 2, qs  0, item 12
cid 4: pid 2, qs  0, item 13
cid 3: pid 2, qs  0, item 14
cid 0: pid 2, qs  0, item 15
producers done.
consumers done.
```

## How it works

The heart of this recipe is in the use of two `condition_variable` objects to control the `producer` and `consumer` threads asynchronously:

```
condition_variable cv_producer{};
condition_variable cv_consumer{};
```

In the `producer()` function, the `cv_producer` object obtains a `unique_lock`, waits for the `deque` to be available, and notifies the `cv_consumer` object when an item has been produced:

```
void producer(const size_t id) {
    for (size_t i {}; i < num_items; ++i) {
        this_thread::sleep_for(delay_time * id);
        unique_lock<mutex> lock(q_mutex);
        cv_producer.wait(lock, [&]{
            return qs.size() < queue_limit;
        });
```

```
            qs.push_back(format("pid {}, qs  {}, item {:02}\n",
                id, qs.size(), i + 1));
            cv_consumer.notify_all();
        }
    }
```

Conversely, in the `consumer()` function, the `cv_consumer` object obtains a `unique_lock`, waits for the deque to have items, and notifies the `cv_producer` object when an item has been consumed:

```
void consumer(const size_t id) {
    while (!production_complete) {
        unique_lock<mutex> lock(q_mutex);
        cv_consumer.wait_for(lock, consumer_wait, [&]{
            return !qs.empty();
        });
        if (!qs.empty()) {
            print("cid {}: {}", id, qs.front());
            qs.pop_front();
        }
        cv_producer.notify_all();
    }
}
```

These complementary locks, waits, and notifications constitute the balance of coordination between multiple producers and consumers.

## Suspend and resume execution with coroutines

Introduced with C++20, *coroutines* were not yet implemented consistently across platforms in time for the previous edition of this book.

Coroutines allow functions to suspend and resume execution without blocking. This provides support for true, language-level asynchronous programming, without reliance on threads, callbacks, or explicit state management.

Today, all major compilers support coroutines reliably. Libraries have stabilized and debugging and performance have improved. This means that coroutines are practical and portable across platforms.

This recipe demonstrates a simple *cooperative scheduler* built with coroutines. Multiple tasks run on a single thread and explicitly yield control, allowing them to progress without preemption or synchronization.

## How to do it

In this recipe we create a non-blocking cooperative scheduler using a coroutine to execute tasks.

- Every coroutine requires a `promise_type` object to manage the lifecycle and behavior of the coroutine. The `promise_type` is encapsulated in a `task` class for use by our coroutine:

```
class task {
public:
    class promise_type; // forward declaration
    using co_h = std::coroutine_handle<promise_type>;
    co_h handle;

    class promise_type {
    public:
        task get_return_object() {
            return task{
                co_h::from_promise(*this)
            };
        }

        std::suspend_always initial_suspend() noexcept
          { return {}; }
        std::suspend_always final_suspend() noexcept
          { return {}; }
        void return_void() noexcept {}
        void unhandled_exception() { std::terminate(); }
    };

    explicit task(co_h h)
        : handle(h) {}

    // move only, no copying
    task(task&& other) noexcept : handle(other.handle)
        { other.handle = {}; }
    task(consttask&)=delete;
```

```
    task& operator=(consttask&)=delete;
    task& operator=(task&&)=delete;

    ~task() {
        if (handle) handle.destroy();
    }
};
```

The `task` class encapsulates the `promise_type`, which provides hooks for the compiler to call when it encounters a coroutine statement. These hooks provide behaviors for the coroutine statements: `co_await`, `co_yield`, and `co_return`.

The `task` class is move-only, so we delete the copy constructor and copy operators.

- The scheduler itself is in a class called `scheduler`:

```
class scheduler {
    std::deque<std::coroutine_handle<>> ready;

public:
    void schedule(std::coroutine_handle<> h) {
        ready.push_back(h);
    }

    void run() {
        while (!ready.empty()) {
            auto h = ready.front();
            ready.pop_front();
            h.resume();
        }
    }
};
```

The `scheduler` class manages a `deque` of `coroutine_handle` objects in a FIFO (first-in, first-out) queue, providing predictable ordered behavior. The `schedule()` function pushes coroutine handles onto the back of the queue and the `run()` function pops them off the front and resumes each coroutine in turn.

- The `yield` class provides the awaitable protocol for use with the `co_await` statement in the coroutine:

```
class yield {
scheduler&y_sched;
public:
    explicit yield(scheduler& s) noexcept : y_sched(s) {}
    bool await_ready() const noexcept { return false; }
    void await_resume() const noexcept {}

    void await_suspend(std::coroutine_handle<> h) const noexcept {
        y_sched.schedule(h);
    }
};
```

  The constructor initializes a reference to the `scheduler` so that a single schedule queue may be properly managed. The member functions, `await_ready()`, `await_resume()`, and `await_suspend()`, provide hooks for the `co_await` coroutine statement.

- The coroutine itself is called `worker()`:

```
task worker(scheduler& sched, int id) {
    for (int i = 0; i < 3; ++i) {
        println("worker {} step {}", id, i);
        co_await yield {sched};
    }
}
```

  The coroutine itself is rather simple, just a loop and a `co_await` expression.

  `co_await` is syntactically an operator and `yield {sched}` is the operand. The compiler uses the `yield` object for its awaitable protocol to run the scheduler.

- Now we can invoke our coroutine in the `main()` function:

```
int main() {
    scheduler sched {};

    // Create tasks
    auto t1 = worker(sched, 1);
    auto t2 = worker(sched, 2);
```

```
    auto t3 = worker(sched, 3);

    // Schedule initial execution
    sched.schedule(t1.handle);
    sched.schedule(t2.handle);
    sched.schedule(t3.handle);

    // Run event loop
    sched.run();
}
```

We create a `scheduler` object, `sched`, and call `worker(sched)` three times. We take the coroutine handles from the three distinct `task` instances and pass those to the scheduler, then run it with `sched.run()`.

Our output:

```
worker 1 step 0
worker 2 step 0
worker 3 step 0
worker 1 step 1
worker 2 step 1
worker 3 step 1
worker 1 step 2
worker 2 step 2
worker 3 step 2
```

We see that our coroutine tasks are executed by the scheduler in the expeted sequential order.

## How it works

The bare minimum requirement for a functional coroutine is a coroutine function, which must contain a coroutine keyword and an associated promise type. The promise type is associated with the coroutine function's return type. For example:

```
say_hello hello() {
    println("before suspension");
    co_await std::suspend_always{};
    println("after resumption");
}
```

Coroutine keywords include, `co_await`, `co_yield`, and `co_return`. These keywords are associated with functions within the `promise_type` class.

In this minimal example, the compiler will look for `say_hello::promise_type` to define how the coroutine is created, suspended, resumed, and destroyed. This means we need a class called `say_hello` that encapsulates a `promise_type` class:

```
class say_hello {
public:
    class promise_type;
    using co_h = std::coroutine_handle<promise_type>;
    co_h h;

    class promise_type {
    public:
        say_hello get_return_object() {
            return say_hello{
                co_h::from_promise(*this)
            };
        }

        std::suspend_never
        initial_suspend() noexcept { return {}; }

        std::suspend_always
        final_suspend() noexcept { return {}; }

        void return_void() noexcept {}
        void unhandled_exception() {}
    };

    explicit say_hello(co_h h) : h(h) {}
       say_hello(say_hello&& other) noexcept : handle(other.handle)
           { other.handle = {}; }
    say_hello(constsay_hello&)=delete;
    say_hello& operator=(constsay_hello&)=delete;

    ~say_hello() {
        if (h) h.destroy();
    }
};
```

The `say_hello` class provides ownership and manages the lifetime of the coroutine. This includes the `promise_type` required for coroutine operation.

Our `hello()` function uses the coroutine statement:

```
co_await std::suspend_always{};
```

The compiler then uses the *awaitable* protocol defined by `std::suspend_always` to determine whether execution should suspend. The `promise_type` class defines suspension points for coroutine creation and completion. In this example, `promise_type::final_suspend()` returns `std::suspend_always`, causing the coroutine to suspend when it reaches the end of execution:

```
std::suspend_always
say_hello::promise_type::final_suspend() { return {}; }
```

The coroutine in our `scheduler` example contains a few additional moving parts, often included in coroutine implementations:

- The `task` class provides ownership and manages the lifetime of the coroutine frame. This is the class that includes the `promise_type`.
- A `scheduler` class maintains the `deque` of coroutine handles.
- The `yield` class implements the awaitable protocol to hand off execution to the scheduler. The awaitable protocol is implemented by member functions that operate as hooks:

  ```
  class yield {
      scheduler& y_sched;
  public:
      explicit yield(scheduler& s) noexcept : y_sched(s) {}
      bool await_ready() const noexcept { return false; }
      void await_resume() const noexcept {}

      void await_suspend(std::coroutine_handle<> h) const noexcept {
          y_sched.schedule(h);
      }
  };
  ```

  The `await_ready()` function returns `false`, indicating that the coroutine should always suspend when this awaitable is awaited.

The `await_suspend()` function is called when suspension occurs. It passes the coroutine handle to the scheduler. This hands control over scheduling decisions to the scheduler without blocking the current thread.

The `await_resume()` function is invoked when the coroutine is later resumed.

The `worker()` function represents the coroutine itself and returns the `task` object.

This example provides a functional introduction to implementing a coroutine.

# Use coroutines for a producer-consumer solution

Earlier in this chapter we used threads for a producer-consumer solution. We've also seen how to implement a scheduler with coroutines. In this recipe, we'll put all our knowledge to work to solve the producer-consumer problem with coroutines.

## How to do it

In this recipe we use two coroutines, one each for the producer and the consumer. These are managed with a common scheduler.

- The `task` class is consistent with the same class in our `scheduler` recipe. In fact, this will often be the same code for different solutions:

```
class task {
public:
    class promise_type;
    using co_h = std::coroutine_handle<promise_type>;
    co_h handle;

    class promise_type {
    public:
        task get_return_object() {
            return task{
                co_h::from_promise(*this)
            };
        }

        std::suspend_always
            initial_suspend() noexcept { return {}; }
        std::suspend_always final_suspend()
            noexcept { return {}; }
        void return_void() noexcept {}
        void unhandled_exception() { std::terminate(); }
```

```
    };

    explicit task(co_h h) : handle(h) {}
    task(task&& other) noexcept : handle(other.handle)
        { other.handle = {}; }
    task(consttask&)=delete;
    task& operator=(consttask&)=delete;
    task& operator=(task&&)=delete;
    ~task() { if (handle) handle.destroy(); }
};
```

The `task` class encapsulates the `promise_type` which provides hooks for the coroutine.

- The `scheduler` class has some additional functions for managing resources that are shared between the producer and consumer

```
class scheduler {
    std::deque<std::coroutine_handle<>> ready;
    bool f_done {false};
    std::deque<size_t> queue {};
public:
    void schedule(std::coroutine_handle<> h) {
        ready.push_back(h);
    }

    void run() {
        while (!ready.empty()) {
            auto h = ready.front();
            ready.pop_front();
            h.resume();
        }
    }

    void done(bool f) { f_done = f; }
    bool done() const { return f_done; }

    void q_push(const size_t& n) { queue.push_back(n); }
    bool q_empty() const { return queue.empty(); }
    size_t q_pop() {
        size_t n = queue.front();
        queue.pop_front();
```

```
        return n;
    }
};
```

We've added `bool f_done` as a flag for the producer to tell the consumer that it's completed. We also added a deque to hold a queue of the producer's work to be consumed by the consumer. These shared resources have proper encapsulation to coordinate their usage: a setter and getter for the flag and push, pop, and empty functions for the queue.

- The `yield` class is also unchanged from our scheduler recipe

```
class yield {
    scheduler& y_sched;
public:
    explicit yield(scheduler& s) noexcept : y_sched(s) {}
    bool await_ready() const noexcept { return false; }
    void await_resume() const noexcept {}

    void await_suspend(std::coroutine_handle<> h) const noexcept {
        y_sched.schedule(h);
    }
};
```

The `yield` class provides the awaitable protocol for use by the `co_await` statement in the coroutine.

- The `producer` coroutine is simply a `for` loop that pushes data into the production queue:

```
task producer(scheduler& sched) {
    for (size_t i = 0; i < num_items; ++i) {
        sched.q_push(i);
        println("produced {}", i);
        co_await yield{sched};
    }
    sched.done(true);
}
```

It uses `sched.q_push()` to push data onto the queue. The `co_await` statement uses the yield awaitable to pass control to the scheduler. We call `sched.done(true)` when the loop completes.

- The `consumer` coroutine runs a loop until it sees an empty queue *and* the `sched.done()` flag is set

```
task consumer(scheduler& sched) {
    while (!sched.q_empty() || !sched.done()) {
        if (!sched.q_empty()) {
            println("consumed {}", sched.q_pop());
        }
        co_await yield{sched};
    }
}
```

The `||` condition in the `while` loop may seem counterintuitive. In the producer-consumer idiom, it's possible for the queue to be empty while more work may still arrive. Conversely, because the coroutines run asynchronously, it's also possible for the done flag to be set while items remain in the queue. For these reasons the `consumer` must continue running until the queue is empty *and* production is complete.

- The `main()` function calls the coroutine functions and passes the handles to a scheduler object

```
int main() {
    scheduler sched{};

    auto p = producer(sched);
    auto c = consumer(sched);

    sched.schedule(p.handle);
    sched.schedule(c.handle);

    sched.run();
}
```

We take the coroutine handles from the two distinct coroutines and pass those to the scheduler. And then call `sched.run()`.

Our output:

```
produced 0
consumed 0
produced 1
consumed 1
produced 2
consumed 2
produced 3
consumed 3
...
```

## How it works

This recipe builds on our scheduler recipe by reusing the `scheduler` and `task` classes while adding producer-consumer coordination. The key distinction is in the shared data between the `producer` and `consumer` coroutine functions.

The producer must create product and the consumer must have access to that product. This is accomplished by sharing data in the `scheduler` class.

The `scheduler` class has three data members:

```
std::deque<std::coroutine_handle<>> ready;
bool f_done{false};
std::deque<size_t> queue {};
```

The `deque<std::coroutine_handle<>> ready` queue is carried over from the previous recipe. This is simply a queue of coroutine handles to be run by the scheduler.

The `f_done` flag is shared with a setter and getter:

```
void done(bool f) { f_done = f; }
bool done() const { return f_done; }
```

This maintains good encapsulation while allowing the `producer` to tell the `consumer` when the work is complete.

The `std::deque<size_t>` queue contains the work of the producer. This is shared by a set of convenience methods:

```
void q_push(const size_t& n) { queue.push_back(n); }
bool q_empty() const { return queue.empty(); }
size_t q_pop() {
    size_t n = queue.front();
    queue.pop_front();
    return n;
}
```

The `producer` can add product to the queue with `q_push()`, and the `consumer` can check the state of the queue with `q_empty()` or pop items from the queue with `q_pop()`. This adds convenience over the `deque` interface while maintaining the safety of encapsulation.

Coroutines can often simplify concurrency by eliminating the need for locks and condition variables. Because coroutines suspend and resume on a single thread, it's easier to share data without the complexity and risk of synchronization primitives.

While coroutines provide efficient cooperative concurrency, unlike threads they do not inherently provide parallel execution across multiple processor cores. If optimal performance on multiple cores is critical, coroutines may be combined with threads or thread pools.


## Subscribe to Deep Engineering

Join thousands of developers and architects who want to understand how software is changing, deepen their expertise, and build systems that last.

Deep Engineering is a weekly expert-led newsletter for experienced practitioners, featuring original analysis, technical interviews, and curated insights on architecture, system design, and modern programming practice.

Scan the QR or visit the link to subscribe for free.



`https://packt.link/deep-engineering-newsletter`

# 10

# Using the File System

The purpose of the STL `filesystem` library is to normalize file system operations across platforms. The `filesystem` library bridges irregularities and normalizes operations between POSIX/Unix, Windows, and other file systems.

The history of file systems is filled with disparity and inconsistency. Differences in file naming conventions, directory separators, directory structures, permissions conventions, and more, have made cross-platform file system usage difficult in the extreme. Even today, the three major operating systems have significant incompatibilities: Linux uses POSIX conventions including the forward slash (/) to separate directory names; Windows uses a backward slash (\) as a directory separator, limits filename extensions to three characters, and has its own unique permissions system; and while earlier versions of Apple's OS used a colon (:) as a directory separator, the current macOS uses POSIX naming conventions but adds extensions to the permissions system.

The `filesystem` library was adapted from the corresponding *Boost* library and incorporated into the STL with C++17. While there may still be some gaps in the various implementations, the recipes in this chapter have been tested on Linux, Windows, and macOS file systems, and compiled with the latest available versions of the GCC, MSVC, and Clang compilers, respectively.

C++23 introduced several refinements to the `filesystem` library including improved `noexcept` specifications, clarified error-handling semantics, and some defect-report corrections for improved implementation consistency. Problems resolved in the C++23 updates addressed portability and encoding-related issues, including improvements to wide-character and UTF-8 path handling across different platforms.

The library uses the `<filesystem>` header, and the `std::filesystem` namespace is commonly aliased as fs:

```
namespace fs = std::filesystem;
```

The `fs::path` class is at the core of the `filesystem` library. It provides normalized filename and directory path representation across disparate environments. A `path` object may represent a file, a directory, or any object in a file system, even a non-existent or impossible object.

In the following recipes, we cover tools for working with files and directories using the `filesystem` library:

- Specialize `std::formatter` for the path class
- Use manipulation functions with path
- List files in a directory
- Search directories and files with a grep utility
- Rename files with `regex` and `directory_iterator`
- Create a disk usage counter

# Specialize std::formatter for the path class

The `path` class is used throughout the `filesystem` library to represent a file or directory path. On POSIX-conformant systems, like macOS and Linux, the path object uses a `value_type` of char to represent filenames. On Windows, the `value_type` is `wchar_t`. On Windows, `format()` and `print()` will not display primitive strings of `wchar_t` characters. This means there is no simple out-of-the-box way to write code that uses the `filesystem` library and is portable across POSIX and Windows.

We could use preprocessor directives to write specific versions of code for Windows. That may be a reasonable solution for some code bases, but for this book it's messy and does not serve the purpose of simple, portable, reusable recipes.

The elegant solution is to write a `formatter` specialization for the `path` class. This allows us to display `path` objects simply and portably across systems.

## How to do it

In this recipe we'll write a formatter specialization for use with the `fs::path` class:

- We start with a namespace alias for convenience. All the `filesystem` names are in the `std::filesystem` namespace:

```
namespace fs = std::filesystem;
```

  Our `formatter` specialization for the `path` class is simple and succinct:

```
template<>
struct std::formatter<fs::path>:
      std::formatter<std::string> {
    template<typename FormatContext>
    auto format(const fs::path& p,
      FormatContext& ctx) const {
        return std::formatter<std::string,
          char>::format(p.string(), ctx);
    }
};
```

  There's a more complete explanation of `formatter` specialization in *Chapter 7* of this book. Here we're specializing the `formatter` for the `fs::path` type, using its `string()` method to get a printable representation.

- In the `main()` function, we use the command line to pass a filename or path:

```
int main(const int argc, const char** argv) {
    if (argc != 2) {
        fs::path fn {argv[0]};
        println("usage: {} <path>", fn.filename());
        return 0;
    }

    fs::path dir {argv[1]};
    if (!fs::exists(dir)) {
        println("path: {} does not exist", dir);
        return 1;
    }
```

```
    println("path: {}", dir);
    println("filename: {}", dir.filename());
    println("cannonical: {}", fs::canonical(dir));
}
```

The `argc` and `argv` parameters expose the standard command line arguments.

`argv[0]` is always the full directory path and filename for the executable itself. If we don't have the correct number of arguments, we use a `filesystem::path` object to display the filename part from `argv[0]` as part of our *usage* message.

The `fs::exists()` function checks if a directory or file exists.

`dir` is a path object. Thanks to our `formatter` specialization, we can now pass it directly to `println()`. This will display the string representation of the path.

The `filename()` method returns a new path object, which we can pass directly to `println()` because of our specialization.

The `fs::cannonical()` function takes a path object and returns a new path object with the canonical absolute directory path. We pass this path object directly to `println()` and it displays the directory path returned from `cannonical()`.

Output:

```
$ ./formatter ./formatter.cpp
path: ./formatter.cpp
filename: formatter.cpp
cannonical: /home/billw/working/chap10/formatter.cpp
```

## How it works

The `fs::path` class is used throughout the `filesystem` library to represent directory paths and filenames. By providing a `formatter` specialization, we can easily display `path` objects consistently across platforms.

The `path` class provides some useful methods. We can iterate through a path to see its component parts:

```
fs::path p {"~/include/foo.h"};
println("{}", p);
for (const auto& x : p) print("[{}] ", x);
print("\n");
```

Output:

```
~/include/foo.h
[~] [include] [foo.h]
```

The iterator returns a `path` object for each element of the `path`.

Notice that the `path` object does not care if the filename exists in the file system. In this context, it's just parsing the provided string.

We can also get different parts of the path:

```
fs::path p {"~/include/foo.h"};
println("{}", p);
println("{}", p.stem());
println("{}", p.extension());
println("{}", p.filename());
println("{}", p.parent_path());
```

Output:

```
~/include/foo.h
foo
.h
foo.h
~/include
```

Each line of the output represents a distinct component of the filename.

- `p` gives us the full path
- `p.stem()` gives us just the filename without the extension
- `p.extension()` gives us just the extension
- `p.filename()` gives us the full filename without the directory path
- `p.parent_path()` gives us the directory path part without the filename

We will continue to use this `formatter` specialization for the `path` class throughout the rest of this chapter.

# Use manipulation functions with path

The `filesystem` library includes functions for manipulating the contents of `path` objects. In this recipe, we will consider a few of these tools, including:

- The `absolute()` function which returns an absolute path
- The /= operator *appends* which appends a new `path` object
- The `canonical()` function which returns a full canonical directory
- The `equivalent()` function which tests if two relative paths resolve to the same file system object

## How to do it

In this recipe, we will examine some functions that manipulate the contents of `path` objects.

- We start with the `namespace` directive and our `formatter` specialization. We do this in every recipe in this chapter:

```
template<>
struct std::formatter<fs::path>:
      std::formatter<std::string> {
    template<typename FormatContext>
    auto format(const fs::path& p,
      FormatContext& ctx) const {
        return std::formatter<std::string,
          char>::format(p.string(), ctx);
    }
};
```

- We can get the current working directory with the `current_path()` function, which returns a path object:

```
println("current_path: {}", fs::current_path());
```

Output:

```
  current_path: /home/billw/chap10
```

- The `absolute()` function returns an absolute path from a relative `path`:

```
println("absolute(p): {}", fs::absolute(p));
```

Output:

```
absolute(p): /home/billw/chap10/testdir/foo.txt
```

`absolute()` will also dereference symbolic links.

- The += operator *concatenates* a string to the end of a `path`:

```
println("concatenate: {}",
  fs::path {"testdir"} += "foo.txt");
```

Output:

```
concatenate: testdirfoo.txt
```

- The /= operator *appends* a string to the end of a path as a new `path` object:

```
println("concatenate: {}", fs::path
  {"testdir"} /= "foo.txt");
```

Output:

```
append: testdir/foo.txt
```

The `canonical()` function returns the full canonical directory `path`:

```
println("canonical: {}",
  fs::canonical(fs::path{ "." } /= "testdir"));
```

Output:

```
canonical: /home/billw/chap10/testdir
```

- The `equivalent()` function tests whether two relative paths resolve to the same file system entity:

```
println("equivalent: {}",
  fs::equivalent("testdir/foo.txt",
  "testdir/../testdir/foo.txt"));
```

Output:

```
  equivalent: true
```

- The `filesystem` library includes the `filesystem_error` class for exception handling:

```
try {
    fs::path p{ fp };
    println("p: {}", p);
    ...
    println("equivalent: {}",
      fs::equivalent("testdir/foo.txt",
      "testdir/../testdir/foo.txt"));
} catch (const fs::filesystem_error& e) {
    println("{}", e.what());
    println("path1: {}", e.path1());
    println("path2: {}", e.path2());
}
```

The `filesystem_error` class includes methods for displaying the error message, and for getting the `path`(s) involved in the error.

If we introduce an error into the `equivalent()` call, we can see the results of the `fileystem_error` class:

```
println("equivalent: {}",
  fs::equivalent("testdir/foo.txt/x",
  "testdir/../testdir/foo.txt/y"));
```

Output:

```
  filesystem error: cannot check file equivalence: No such file or directory
  [testdir/foo.txt/x] [testdir/../testdir/foo.txt/y]
```

```
path1: testdir/foo.txt/x
path2: testdir/../testdir/foo.txt/y
```

This is the output on Debian with GCC.

The `filesystem_error` class provides additional detail through its `path1()` and `path2()` methods. These methods return path objects.

- You can also use `std::error_code` with some of the `filesystem` functions:

```
fs::path p {fp};
std::error_code e;
println("canonical: {}",
    fs::canonical(p /= "foo", e));
println("error: {}", e.message());
```

Output:

```
canonical:
error: Not a directory
```

- Even though Windows uses a very different file system, this code still works as expected, using Windows file-naming conventions:

```
p: testdir/foo.txt
current_path: C:\Users\billw\chap10
absolute(p): C:\Users\billw\chap10\testdir\foo.txt
concatenate: testdirfoo.txt
append: testdir\foo.txt
canonical: C:\Users\billw\chap10\testdir
equivalent: true
```

## How it works

Most of these functions take a `path` object, an optional `std::error_code` object, and return a `path` object:

```
path absolute(constpath&p);
path absolute(constpath&p,std::error_code&ec);
```

The `equivalent()` function takes two `path` objects and returns a `bool`:

```
bool equivalent( const path& p1, constpath&p2);
bool equivalent( const path& p1, const path& p2,
std::error_code&ec);
```

The path class has operators for concatenate and append. Both operators are destructive. They modify the path on the left-hand side of the operator:

```
p1 += source; // concatenate
p1 /= source; // append
```

For the right-hand side, these operators take either a `path` object, a `string`, a `string_view`, a C-string, or a pair of iterators.

The concatenate operator adds the `string` from the right-hand side of the operator to the end of the `p1path` string.

The append operator adds a separator (e.g., `/` or \), followed by the `string` from the right-hand side of the operator to the end of the `p1path` string.

# List files in a directory

The `filesystem` library provides a `directory_entry` class with directory-related information about a given `path`. We can use this to create useful directory listings.

## How to do it

In this recipe we'll create a directory listing utility using the information in the `directory_entry` class:

- We'll start with our `namespace` alias and `formatter` specialization for displaying `path` objects

```
namespace fs = std::filesystem;
template<>
struct std::formatter<fs::path>: std::formatter<std::string> {
    template<typename FormatContext>
    auto format(const fs::path& p,
      FormatContext& ctx) const {
        return std::formatter<std::string,
          char>::format(p.string(), ctx);
```

```
    }
};
```

- The `directory_iterator` class makes it easy to list a directory:

```
int main() {
    constexpr const char* fn {"."};
    const fs::path fp{fn};
    for (const auto& de : fs::directory_iterator{fp}) {
        print("{} ", de.path().filename());
    }
    print("\n");
}
```

Output:

```
  Makefile working working.cpp zedzedtop.txt alpha.txt
```

- We can add command-line options to make this work like Unix `ls`:

```
int main(const int argc, const char** argv) {
    fs::path fp {argc > 1 ? argv[1] : "."};
    if (!fs::exists(fp)) {
        const auto cmdname {fs::path{argv[0]}.filename()};
        println("{}: {} does not exist", cmdname, fp);
        return 1;
    }
    if (is_directory(fp)) {
        for(const auto& de : fs::directory_iterator{fp}) {
            print("{} ", de.path().filename());
        }
    } else {
        print("{} ", fp.filename());
    }
    print("\n");
}
```

If there is a command line argument, we use it to create a `path` object. Otherwise, we use `"."` for the current directory.

We check if the `path` exists with `if_exists()`. If not, we print an error message and exit. The error message includes `cmdname`, from `argv[0]`.

Next, we check `is_directory()`. If we have a directory, we loop through a `directory_iterator` for each entry. `directory_iterator` iterates over `directory_entry` objects. `de.path().filename()` gets the `path` and `filename` from each `directory_entry` object.

Output:

```
$ ./working
Makefile working working.cpp textfile.txt include alpha.txt zedzedtop.txt
testdir
$ ./working working.cpp
working.cpp
$ ./working foo.bar
working: foo.bar does not exist
```

- If we want our output sorted, we can store our `directory_entry` objects in a sortable container
  Let's create an alias for `fs::directory_entry`. We'll be using this a lot. This goes at the top of the file:

```
using de = fs::directory_entry;
```

At the top of `main()`, we declare a `vector` of `de` objects:

```
vector<de> entries{};
```

Inside the `is_directory()` block, we load the `vector`, sort it, and then display it:

```
if (is_directory(fp)) {
    for(const auto& de : fs::directory_iterator{fp}) {
        entries.emplace_back(de);
    }
    std::sort(entries.begin(), entries.end());
    for (const auto& e : entries) {
        print("{} ", e.path().filename());
    }
} else { ...
```

Now our output is sorted:

```
Makefile alpha.txt include testdir textfile.txt working working.cpp
zedzedtop.txt
```

Notice that `Makefile` is sorted first, apparently out of order. This is because capital letters sort before lowercase in ASCII order.

- If we want a case-insensitive sort, we need a comparison function that ignores case. First, we need a function to return a lowercase string:

```
string strlower(string s) {
    auto char_lower = [](char c) -> char {
        if (c >= 'A'&&c<= 'Z') return c + ('a' - 'A');
        else return c;
    };
    std::transform(s.begin(), s.end(), s.begin(),
      char_lower);
    return s;
}
```

- Now we need a function that compares two `directory_entry` objects, using `strlower()`:

```
bool dircmp_lc(const de& lhs, const de& rhs) {
    const auto lhstr {lhs.path().string()};
    const auto rhstr {rhs.path().string()};
    return strlower(lhstr) < strlower(rhstr);
}
```

- Now we can use `dircmp_lc()` in our sort:

```
std::sort(entries.begin(), entries.end(), dircmp_lc);
```

Our output is now sorted ignoring case:

```
alpha.txt include Makefile testdir textfile.txt working working.cpp
zedzedtop.txts
```

- At this point we have a simple directory listing utility

There's a lot more information available from the filesystem library. Let's create a `print_dir()` function to gather more information and format it for display in the style of Unix `ls`:

```
void print_dir(const de& dir) {
    using fs::perms;
    const auto fpath {dir.path()};
    const auto fstat {dir.symlink_status()};
    const auto fperm {fstat.permissions()};
    const uintmax_t fsize {is_regular_file(fstat)
      ? file_size(fpath) : 0};
    const auto fn {fpath.filename()};

    string suffix{};
    if (is_directory(fstat)) suffix = "/";
    else if ((fperm & perms::owner_exec) != perms::none) {
        suffix = "*";
    }
    println("{}{}", fn, suffix);
}
```

The `print_dir()` function takes a `directory_entry` argument. We then retrieve some useful objects from the `directory_entry`.

`dir.path()` returns a path object.

`dir.symlink_status()` returns a `file_status` object, without following symbolic links.

`fstat.permissions()` returns a perms object. `fsize` is the size of the file and `fn` is the `filename` string. We'll look more closely at each of these as we use them.

Unix `ls` uses trailing characters, after the `filename`, to indicate a directory or an executable. We can test the `fstat` object with `is_directory()` to see if the file is a directory and add a trailing `/` to the filename. Likewise, we can find out if a file is executable by testing the `fperm` object.

- We call `print_dir()` from `main()` in the for loop after `sort()`:

```
std::sort(entries.begin(), entries.end(), dircmp_lc);
for (const auto& e : entries) {
```

```
    print_dir(e);
}
```

Our output now looks like this:

```
alpha.txt
include*
Makefile
testdir/
textfile.txt
working*
working.cpp
zedzedtop.txt
```

Notice the `include*` entry. That's really a symbolic link, not an executable file. Let's notate that properly by following the link to get the target path:

```
string suffix{};
if (is_symlink(fstat)) {
    suffix = " -> ";
    suffix += fs::read_symlink(fpath).string();
}
else if (is_directory(fstat)) suffix = "/";
else if((fperm&perms::owner_exec)!=perms::none)suffix="*";
```

The `read_symlink()` function returns a path object. We take the `string()` representation of the returned `path` object and add it to the suffix for this output:

```
alpha.txt
include -> /home/billw/include
Makefile
testdir/
textfile.txt
working*
working.cpp
zedzedtop.txt
```

- The Unix `ls` command also includes a string of characters to indicate a file's permission bits. It looks something like this: `drwxr-xr-x`.

The first character indicates the type of the file, for example: `d` for directory, `l` for symbolic link, `-` for regular file.

The `type_char()` function returns the appropriate character:

```
char type_char(const fs::file_status& st) {
    using fs::file_type;

    switch (st.type()) {
        case file_type::symlink:    return 'l';
        case file_type::directory:  return 'd';
        case file_type::character:  return 'c';
        case file_type::block:      return 'b';
        case file_type::fifo:       return 'p';
        case file_type::socket:     return 's';
        case file_type::regular:    return '-';
        case file_type::unknown:    return 'o';
        default:                    return '?';
    }
}
```

`filesystem::file_type` is an `enum` class that provides constants for testing against the `file_status` value.

- The rest of the string is in three triplets. Each triplet includes positions for the read, write, and execute permission bits, in the form `rwx`. If a bit is not set, its character is replaced by a dash. There are three triplets for three sets of permissions: owner, user, and other, respectively.

```
string rwx(const fs::perms& p) {
    using fs::perms;
    auto bit2char = [&p](perms bit, char c) {
        return(p&bit)==perms::none?'-':c;
    };

    return { bit2char(perms::owner_read,   'r'),
             bit2char(perms::owner_write,  'w'),
             bit2char(perms::owner_exec,   'x'),
             bit2char(perms::group_read,   'r'),
             bit2char(perms::group_write,  'w'),
             bit2char(perms::group_exec,   'x'),
```

```
            bit2char(perms::others_read,  'r'),
            bit2char(perms::others_write, 'w'),
            bit2char(perms::others_exec,  'x') };
}

std::string rwx(fs::perms p) {
    using fs::perms;

    auto bit2char = [](perms p, perms bit, char c) {
        return(p&bit)!=perms::none?c:'-';
    };

    return {
        bit2char(p, perms::owner_read,   'r'),
        bit2char(p, perms::owner_write,  'w'),
        bit2char(p, perms::owner_exec,   'x'),

        bit2char(p, perms::group_read,   'r'),
        bit2char(p, perms::group_write,  'w'),
        bit2char(p, perms::group_exec,   'x'),

        bit2char(p, perms::others_read,  'r'),
        bit2char(p, perms::others_write, 'w'),
        bit2char(p, perms::others_exec,  'x')
    };
}
```

The `perms` object represents the POSIX permissions bitmap, but it's not necessarily implemented as bits. Each entry must be compared to the `perms::none` value. Our lambda function fulfills this requirement.

We add this definition to the top of our `print_dir()` function:

```
const auto permstr {type_char(fstat) + rwx(fperm)};
```

We update our `format` string:

```
println("{} {}{}", permstr, fn, suffix);
```

And we get this output:

```
-rw-r--r-- alpha.txt
lrwxrwxrwx include -> /home/billw/include
-rw-r--r-- Makefile
drwxr-xr-x testdir/
-rw-r--r-- textfile.txt
-rwxr-xr-x working*
-rw-r--r-- working.cpp
-rw-r--r-- zedzedtop.txt
```

- Now, let's add a size string. The `fsize` value is from the `file_size()` function which returns a `std::uintmax_t` type. This represents the maximum size natural integer on the target system. `uintmax_t` is not always the same as `size_t` and does not always convert easily. Notably, `uintmax_t` is 32 bits on Windows, where `size_t` is 64 bits.

```
string size_string(const uintmax_t fsize) {
    constexpr const uintmax_t kilo {1024};
    constexpr const uintmax_t mega {kilo * kilo};
    constexpr const uintmax_t giga {mega * kilo};
    string s;

    if (fsize >= giga ) return
        format("{}{}", (fsize + giga / 2) / giga, 'G');
    else if (fsize >= mega) return
        format("{}{}", (fsize + mega / 2) / mega, 'M');
    else if (fsize >= kilo) return
        format("{}{}", (fsize + kilo / 2) / kilo, 'K');
    else return format("{}B", fsize);
}
```

I chose to use `1,024` as `1K` in this function, as that appears to be the default on both Linux and BSD Unix. In production, this could be a command-line option.

We update our `format` string in `print_dir()`:

```
println("{} {:>6} {}{}",
  permstr, size_string(fsize), fn, suffix);
```

Now, we get this output:

```
-rw-r--r--    78B alpha.txt
lrwxrwxrwx     0B include -> /home/billw/include
-rw-r--r--   334B Makefile
drwxr-xr-x     0B testdir/
-rw-r--r--   281B textfile.txt
-rwxr-xr-x   146K working*
-rw-r--r--     5K working.cpp
-rw-r--r--   588B zedzedtop.txt
```

**Note**

This utility is designed for POSIX-compliant systems, such as Linux and macOS. It works on a Windows system but the Windows permissions system is significantly different from that of the POSIX system. On Windows, the permission bits will always appear fully set, e.g., `-rwxrwxrwx`.

## How it works

The `filesystem` library carries a rich set of information through its `directory_entry` and related classes. The major classes we used in this recipe include:

- The `path` class represents a file system path, according to the rules of the target system. A `path` is constructed from a `string` or another `path`. It need not represent an existing file path, or even a possible file path. The `path` is parsed to component parts, including a root name, root directory, and an optional series of filenames and directory separators.
- The `directory_entry` class carries a `path` object as a member, and may also store additional attributes including, hard link count, status, symbolic link, file size, and last write time.
- The `file_status` class carries information about the type and permissions of a file. A `perms` object may be a member of `file_status`, representing the permissions structure of a file.
- There are two functions for retrieving a `perms` object from a `file_status`. The `status()` function and the `symlink_status()` function both return a `perms` object. The difference is in how they handle a symbolic link. The `status()` function will follow

a symbolic link and return the perms from the target file. `symlink_status()` will return the perms from the symbolic link itself.

## There's more...

I also wanted to include the last-write time of each file in the directory listing.

The `directory_entry` class has a member function, `last_write_time()`, which returns a `file_time_type` object representing the time stamp of the last time the file was written.

It's recommended that user code should use `std::chrono::clock_cast` to convert time points between clocks. As of the previous edition of this book, none of the implementations available had a working `std::chrono::clock_cast` specialization for this purpose. As of late 2025, GCC and MSVC have working `clock_cast` specializations but Apple's `clang` does not.

Here's a solution that works by casting the time stamp with `chrono::clock_cast`:

```
std::string time_string(const fs::directory_entry& dir) {
    using std::chrono::clock_cast;
    using std::chrono::system_clock;

    auto tp = clock_cast<system_clock>(dir.last_write_time());
    return std::format("{:%F %R}", tp);
}
```

Using this `time_string()` function, we can add to `print_dir()`:

```
const string timestr {time_string(dir)};
```

We can then change the `format` string:

```
print("{} {:>6} {} {}{}\n",
  permstr, sizestr, timestr, fn, suffix);
```

And we get this output:

```
-rw-r--r--    78B 2026-01-06 20:07 alpha.txt
lrwxrwxrwx     0B 2024-02-10 23:44 include -> /home/billw/include
-rw-r--r--   334B 2025-12-19 20:46 Makefile
drwxr-xr-x     0B 2026-01-06 19:36 testdir/
-rw-r--r--   281B 2026-01-06 20:08 textfile.txt
-rwxr-xr-x   188K 2026-01-06 20:27 working*
```

```
-rw-r--r--     5K 2026-01-06 20:26 working.cpp
-rw-r--r--   588B 2026-01-06 20:08 zedzedtop.txt
```

This works on Debian with GCC-15 and Windows 11 with MSVC 14 or later. Do not expect it to work without modification on other systems.

# Search directories and files with a grep utility

To demonstrate traversing and searching directory structures, we create a simple utility that works like Unix `grep`. This utility uses `recursive_directory_iterator` to traverse nested directories and searches files for matches with a regular expression.

## How to do it

In this recipe we will write a simple `grep` utility that traverses directories to search files with a regular expression.

- We'll start with some convenient aliases:

```
namespace fs = std::filesystem;
using de = fs::directory_entry;
using rdit = fs::recursive_directory_iterator;
using match_v = vector<std::pair<size_t, std::string>>;
```

  `match_v` is a vector of regular expression match results.

- We continue using our `formatter` specialization for `path` objects:

```
template<>
struct std::formatter<std::filesystem::path>: std::formatter<std::string> {
    template<typename FormatContext>
    auto format(const std::filesystem::path& p,
      FormatContext& ctx) const {
        return std::formatter<std::string,
          char>::format(p.string(), ctx);
    }
};
```

- We have a simple function for getting regular expression matches from a file:

```
match_v matches(const fs::path& fpath, const regex& re) {
    match_v matches {};
    std::ifstream instrm(fpath.string(),
```

```
        std::ios_base::in);
    string s {};
    for (size_t lineno {1}; getline(instrm, s); ++lineno) {
        if (std::regex_search(s.begin(), s.end(), re)) {
            matches.emplace_back(lineno, move(s));
        }
    }
    return matches;
}
```

In this function, we open the file with `ifstream`, read lines from the file with `getline()`, and match the regular expression with `regex_search()`. Results are collected in the vector and returned.

- We can now call this function from `main()`:

```
int main() {
    constexpr const char * fn {"working.cpp"};
    constexpr const char * pattern {"path"};

    fs::path fpath{ fn };
    regex re {pattern};
    auto regmatches{ matches(fpath, re) };
    for (const auto& [lineno, line] : regmatches) {
        cout << format("{}: {}\n", lineno, line);
    }
    println("found {} matches", regmatches.size());
}
```

In this example, we use constants for the filename and the regular expression pattern. We create `path` and `regex` objects, call the `matches()` function, and print the results.

Our output has line numbers and strings for the matching lines:

```
  25: struct std::formatter<fs::path>: std::formatter<std::string> {
  27:     auto format(const fs::path& p, FormatContext& ctx) {
  32: match_v matches(const fs::path& fpath, const regex& re) {
  34:     std::ifstream instrm(fpath.string(), std::ios_base::in);
  62:     constexpr const char * pattern{ "path" };
  64:     fs::path fpath{ fn };
  66:     auto regmatches{ matches(fpath, re) };
```

- Our utility needs to take command line arguments for the `regex` pattern and filenames. It should be able to traverse directories or take a list of filenames (which may be the result of command-line wildcard expansion). This requires a bit of logic in the `main()` function.
  First, we need one more helper function:

```
size_t pmatches(const regex& re, const fs::path& epath,
      const fs::path& search_path) {
    fs::path target {epath};
    auto regmatches {matches(epath, re)};
    auto matchcount {regmatches.size()};
    if (!matchcount) return 0;

    if (!(search_path == epath)) {
        target = epath.lexically_relative(search_path);
    }
    for (const auto& [lineno, line] : regmatches) {
        println("{} {}: {}", target, lineno, line);
    }
    return regmatches.size();
}
```

  This function calls our `matches()` function and prints the results. It takes a `regex` object and two `path` objects. `epath` is the result of a directory search, and `search_path` is the search directory itself. We'll set these in `main()`.

- In `main()`, we use the `argc` and `argv` command-line arguments and we declare a few variables:

```
int main(const int argc, const char** argv) {
    const char * arg_pat {};
    regex re {};
    fs::path search_path {};
    size_t matchcount {};
    ...
```

  `arg_pat` is for the regular expression pattern from the command line; `re` is the `regex` object; `search_path` is the command line search path argument; and `matchcount` is for counting the matched lines.

- Continuing in `main()`, if we have no arguments, then we print a short usage string:

```
if(argc < 2) {
    auto cmdname {fs::path(argv[0]).filename()};
    println("usage: {} pattern [path/file]", cmdname);
    return 1;
}
```

`argv[1]` is always the invoking command from the command line. `cmdname` uses the `filename()` method to return a `path` with just the `filename` part of the invoking command path.

Next, we parse the regular expression. We use a `try-catch` block to capture any error from the `regex` parser:

```
arg_pat = argv[1];
try {
    re = regex(arg_pat, std::regex_constants::icase);
} catch(const std::regex_error& e) {
    println("{}: {}\n", e.what(), arg_pat);
    return 1;
}
```

We use the `icase` flag to tell the `regex` parser to ignore case.

If `argc == 2`, we have just one argument, which we treat as the regular expression pattern, and we use the current directory for the search path:

```
if(argc == 2) {
    search_path = ".";
        for (const auto& entry : rdit{ search_path }) {
        const auto epath {entry.path()};
        matchcount += pmatches(re, epath, search_path);
    }
}
```

`rdit` is an alias for the `recursive_directory_iterator` class, which traverses the directory tree from the starting path, returning a `directory_entry` object for each file it encounters. We then use create a `path` object and call `pmatches()` to go through the file and print any regular expression matches.

- At this point in `main()`, we know that `argc` is >=2. Now, we handle cases where we have one or more file paths on the command line.

```
int count{ argc - 2 };
while(count-- > 0) {
    fs::path p {argv[count + 2]};
    if(!exists(p)) {
        println("not found: {}", p);
        continue;
    }
    if(is_directory(p)) {
        for (const auto& entry : rdit {p}) {
            const auto epath {entry.path()};
            matchcount += pmatches(re, epath, p);
        }
    } else {
        matchcount += pmatches(re, p, p);
    }
}
```

The `while` loop handles one or more arguments past the search pattern on the command line. It checks each filename to ensure it exists. Then, if it's a directory, it uses the `rdit` alias for the `recursive_directory_iterator` class to traverse the directory and call `pmatches()` to print any pattern matches in the files.

If it's a single file, it calls `pmatches()` on that file.

- We can run our grep clone with one argument as the search pattern:

```
$ ./bwgrep using
dir.cpp 12: using std::format;
dir.cpp 13: using std::cout;
dir.cpp 14: using std::string;
...
formatter.cpp 10: using std::cout;
formatter.cpp 11: using std::string;
formatter.cpp 13: using namespace std::filesystem;
found 33 matches
```

We can run it with a second argument as a directory to search:

```
$ ./bwgrep using ..
chap04/iterator-adapters.cpp 12: using std::format;
chap04/iterator-adapters.cpp 13: using std::cout;
chap04/iterator-adapters.cpp 14: using std::cin;
...
chap01/hello-version.cpp 24: using std::print;
chap01/chrono.cpp 8: using namespace std::chrono_literals;
chap01/working.cpp 15: using std::cout;
chap01/working.cpp 34:     using std::vector;
found 529 matches
```

Notice that it *traverses the directory tree* to find files in sub-directories.

Or we can run it with a single file argument:

```
$ ./bwgrep using bwgrep.cpp
bwgrep.cpp 13: using std::format;
bwgrep.cpp 14: using std::cout;
bwgrep.cpp 15: using std::string;
...
bwgrep.cpp 22: using rdit = fs::recursive_directory_iterator;
bwgrep.cpp 23: using match_v = vector<std::pair<size_t, std::string>>;
found 9 matches
```

## How it works

While the main task of this utility is regular expression matching, we're concentrating on the technique of recursively processing directories of files.

The `recursive_directory_iterator` object is interchangeable with `directory_iterator`, except `recursive_directory_iterator` operates recursively over all the entries of each subdirectory.

**Tip**

For more about regular expressions, see the recipe: *Parse strings with regular expressions* in *Chapter 7*.

# Rename files with regex and directory_iterator

This is a useful utility that renames files using regular expressions. It uses `directory_iterator` to find the files in a directory and `fs::rename()` to rename them.

## How to do it

In this recipe we create a file rename utility that uses regular expressions:

- We start by defining a few convenient aliases:

```
namespace fs = std::filesystem;
using dit = fs::directory_iterator;
using pat_v = vector<std::pair<regex, string>>;
```

  The `pat_v` alias is a `vector` for use with our regular expressions.

- We also continue to use the `formatter` specialization for `path` objects:

```
template<>
struct std::formatter<std::filesystem::path>:
      std::formatter<std::string> {
    template<typename FormatContext>
    auto format(const std::filesystem::path& p,
          FormatContext& ctx) const {
        return std::formatter<std::string,
          char>::format(p.string(), ctx);
    }
};
```

- We have a function for applying the regular expression replacement to filename strings:

```
string replace_str(string s, const pat_v& replacements) {
    for (const auto& [pattern, repl] : replacements) {
        s = regex_replace(s, pattern, repl);
    }
    return s;
}
```

  Notice that we loop through a `vector` of pattern/replacement pairs, applying the regular expressions successively. This allows us to stack our replacements.

- In `main()`, we first check the command-line arguments:

```
int main(const int argc, const char** argv) {
    pat_v patterns {};

    if (argc < 3 || argc % 2 != 1) {
        fs::path cmdname {fs::path{argv[0]}.filename()};
        println("usage: {} [regex replacement] ...",
          cmdname);
        return 1;
    }
...
```

The command line accepts one or more *pairs of strings*. Each pair of strings includes a *regex* (regular expression) followed by a *replacement*.

- Now we populate the `vector` with `regex` and `string` objects:

```
for (int i {1}; i < argc; i += 2) {
    patterns.emplace_back(argv[i], argv[i + 1]);
}
```

The `pair` constructor constructs the `regex` and `string` objects in place, from the C-strings passed on the command line. These are added to the `vector` with the `emplace_back()` method.

We search the current directory using a `directory_iterator` object:

```
for (const auto& entry : dit {fs::current_path()}) {
    fs::path fpath {entry.path()};
    string rname{
      replace_str(fpath.filename().string(), patterns) };

    if (fpath.filename().string() != rname) {
        fs::path rpath{ fpath };
        rpath.replace_filename(rname);

        if (exists(rpath)) {
            println("Error: cannot rename – destination file exists.");
        } else {
            fs::rename(fpath, rpath);
```

```
            println("{} -> {}", fpath.filename(),
              rpath.filename());
        }
    }
}
```

In this `for` loop, we call `replace_str()` to get the replacement `filename` and then check that the new name is not a duplicate of a file in the directory. We use the `replace_filename()` method on a path object to create a path with the new filename and use `fs::rename()` to rename the file.

- To test the utility, I've created a directory with a few files in it for renaming:

```
$ ls
bwfoo.txt  cpp-version.cpp  jump.cpp  numword-test.cpp  numword.cpp
working.cpp
```

We can do something simple like change `.cpp` to `.Cpp`:

```
$ ../rerename .cpp .Cpp
numword-test.cpp -> numword-test.Cpp
numword.cpp -> numword.Cpp
working.cpp -> working.Cpp
cpp-version.cpp -> cpp-version.Cpp
jump.cpp -> jump.Cpp
```

Let's change them back again:

```
$ ../rerename .Cpp .cpp
cpp-version.Cpp -> cpp-version.cpp
numword.Cpp -> numword.cpp
jump.Cpp -> jump.cpp
working.Cpp -> working.cpp
numword-test.Cpp -> numword-test.cpp
```

Using standard regular expression syntax, I can add "`bw`" to the beginning of each of the filenames:

```
$ ../rerename '^' bw
numword-test.cpp -> bwnumword-test.cpp
numword.cpp -> bwnumword.cpp
```

```
working.cpp -> bwworking.cpp
cpp-version.cpp -> bwcpp-version.cpp
bwfoo.txt -> bwbwfoo.txt
jump.cpp -> bwjump.cpp
```

Notice that it even renamed the files that already had "bw" at the beginning. Let's have it not do that. First, we restore the filenames:

```
$ ../rerename '^bw' ''
bwnumword.cpp -> numword.cpp
bwjump.cpp -> jump.cpp
bwnumword-test.cpp -> numword-test.cpp
bwbwfoo.txt -> bwfoo.txt
bwworking.cpp -> working.cpp
bwcpp-version.cpp -> cpp-version.cpp
```

Now we use a regex that checks if the filename already begins with "bw":

```
$ ../rerename '^(?!bw)' bw
numword-test.cpp -> bwnumword-test.cpp
numword.cpp -> bwnumword.cpp
working.cpp -> bwworking.cpp
cpp-version.cpp -> bwcpp-version.cpp
jump.cpp -> bwjump.cpp
```

Because we use a `vector` of regex/replacement strings, we can stack several replacements:

```
$ ../rerename foo bar '\.cpp$' '.xpp' grep grok
bwnumword.cpp -> bwnumword.xpp
bwjump.cpp -> bwjump.xpp
bwnumword-test.cpp -> bwnumword-test.xpp
bwfoo.txt -> bwbar.txt
bwworking.cpp -> bwworking.xpp
bwcpp-version.cpp -> bwcpp-version.xpp
```

## How it works

The `filesystem` part of this recipe uses `directory_iterator` to return a `directory_entry` object for each file in the current directory:

```
for (const auto& entry : dit {fs::current_path()}) {
    fs::path fpath {entry.path()};
    ...
}
```

We then construct a `path` object from the `directory_entry` object to process the file.

We use the `replace_filename()` method on a path object to create the destination for the rename operation:

```
fs::path rpath {fpath};
rpath.replace_filename(rname);
```

Here we create a duplicate `path` and change its name, giving us both sides for the rename operation:

```
fs::rename(fpath, rpath);
```

On the regular expression side of the recipe, we use `regex_replace()`, which uses regular expression syntax to perform substitutions in a string:

```
s = regex_replace(s, pattern, repl);
```

Regular expression syntax is extremely powerful. It even allows replacements to include sections of the search string:

```
$ ../rerename '(bw)(.*\.)(.*)$' '$3$2$1'
bwgrep.cpp -> cppgrep.bw
bwfoo.txt -> txtfoo.bw
```

By using parentheses in the search pattern, I can easily rearrange parts of a filename.

For more about regular expressions, see the recipe: *Parse strings with regular expressions* in *Chapter 7*.

# Create a disk usage counter

This is a simple utility that totals the size of every file in a directory and its subdirectories. It runs on both POSIX/Unix and Windows filesystems.

## How to do it

This recipe is a utility to report the size of every file in a directory and its sub-directories, along with a total. We'll re-use some of the functions we've used elsewhere in this chapter:

- We'll start with a few convenient aliases:

```
namespace fs = std::filesystem;
using dit = fs::directory_iterator;
using de = fs::directory_entry;
```

- We also use our `formatter` specialization for `fs::path` objects:

```
template<>
struct std::formatter<std::filesystem::path>: std::formatter<std::string> {
    template<typename FormatContext>
    auto format(const std::filesystem::path& p,
      FormatContext& ctx) const {
        return std::formatter<std::string,
          char>::format(p.string(), ctx);
    }
};
```

- For reporting the size of the directory, we'll use this `make_commas()` function:

```
string make_commas(const uintmax_t& num) {
    string s {std::to_string(num)};
    for (long l = s.length() - 3; l > 0; l -= 3) {
        s.insert(l, ",");
    }
    return s;
}
```

  We've used this before. It inserts a comma before every third character from the end.

- To sort our directory, we'll need a lowercase string function:

```
string strlower(string s) {
    auto char_lower = [](char c) -> char {
        if (c >= 'A'&&c<= 'Z') return c + ('a' - 'A');
        else return c;
    };
    std::transform(s.begin(), s.end(), s.begin(),
      char_lower);
    return s;
}
```

- We need a comparison predicate for sorting `directory_entry` objects by lowercase path name:

```
bool dircmp_lc(const de& lhs, const de& rhs) {
    const auto lhstr {lhs.path().string()};
    const auto rhstr {rhs.path().string()};
    return strlower(lhstr) < strlower(rhstr);
}
```

- `size_string()` returns abbreviated values for reporting file size in gigabytes, megabytes, kilobytes, or bytes:

```
string size_string(const uintmax_t fsize) {
    constexpr const uintmax_t kilo {1024};
    constexpr const uintmax_t mega {kilo * kilo};
    constexpr const uintmax_t giga {mega * kilo};

    if (fsize >= giga ) return
      format("{}{}", (fsize + giga / 2) / giga, 'G');
    else if (fsize >= mega) return
      format("{}{}", (fsize + mega / 2) / mega, 'M');
    else if (fsize >= kilo) return
      format("{}{}", (fsize + kilo / 2) / kilo, 'K');
    else return format("{}B", fsize);
}
```

- `entry_size()` returns the size of a file, or if it's a directory, the recursive size of the directory

```
uintmax_t entry_size(const fs::path& p) {
    if (fs::is_regular_file(p)) return fs::file_size(p);
    uintmax_t accum {};
    if (fs::is_directory(p) && !fs::is_symlink(p)) {
        for (auto& e : dit {p}) {
            accum += entry_size(e.path());
        }
    }
    return accum;
}
```

- In `main()`, we start with declarations and test if we have a valid directory to search:

```
int main(const int argc, const char** argv) {
    auto dir {argc > 1 ?
      fs::path(argv[1]) : fs::current_path()};
    vector<de> entries {};
    uintmax_t accum {};

    if (!exists(dir)) {
        println("path {} does not exist", dir);
        return 1;
    }
    if (!is_directory(dir)) {
        println("{} is not a directory", dir);
        return 1;
    }
    println("{}:", absolute(dir));
```

For our directory `path dir`, we use `argv[1]` if we have an argument, otherwise `current_path()` for the current directory.

The vector of `directory_entry` objects is used for sorting our response.

`accum` is used to accumulate values for our final size total.

We make sure `dir` exists and is a directory before proceeding to examine the directory.

- Next, a simple loop to populate the `vector`. Once populated, we sort entries using our `dircmp_lc()` function as a comparison predicate:

```
for (const auto& e : dit {dir}) {
    entries.emplace_back(e.path());
}
std::sort(entries.begin(), entries.end(), dircmp_lc);
```

Now that everything is set up, we can accumulate results from the sorted `vector` of `directory_entry` objects:

```
for (const auto& e : entries) {
    fs::path p {e};
    uintmax_t esize {entry_size(p)};
    string dir_flag {};

    accum += esize;
    if (is_directory(p) && !is_symlink(p)) dir_flag = "▽";
    println("{:>5} {}{}", size_string(esize), p.filename(),
      dir_flag);
}
println("{:->25}", "");
println("total bytes: {} ({})", make_commas(accum), size_string(accum));
```

The call to `entry_size()` returns the size of the file or directory represented in the `directory_entry` object.

If the current entry is a directory (and not a *symbolic link*) we add a symbol to indicate it's a directory. I chose an inverted triangle, like a disclosure symbol. You may use anything here.

After the loop is complete, we display the accumulated size in both bytes with commas, and the abbreviated notation from `size_string()`.

Our output:

```
/home/billw/working/testdir/:
 133M backup ▽
 935B cpp-version.cpp
  20B five-words.txt
```

```
    2K jump.cpp
    2K numword-test.cpp
    2K numword.cpp
    6K the-raven.txt
 154K working
    3K working.cpp
------------------------
total bytes: 139,535,298 (133M)
```

The output is a list of the size of every file and directory in the specified directory. Sub-directories are indicated with the inverted triangle symbol ▽.

## How it works

The `fs::file_size()` function returns a `uintmax_t` value, which is the largest standard unsigned integer defined in the C++ standard. While `size_t` is normally a 64-bit integer on most 64-bit systems, it remains 32 bits on 32-bit builds, even where `uintmax_t` may be 64 bits, notably when using the Windows 32-bit compiler. This can cause compilation to fail due to a type mismatch.

The `entry_size()` function takes a path object and returns a `uintmax_t` value:

```
uintmax_t entry_size(const fs::path& p) {
    if (fs::is_regular_file(p)) return fs::file_size(p);

    uintmax_t accum {};
    if (fs::is_directory(p) && !fs::is_symlink(p)) {
        for(auto& e : dit {p}) {
            accum += entry_size(e.path());
        }
    }
    return accum;
}
```

The `entry_size()` function checks for a regular file and returns the size of the file. Otherwise, it checks for a directory that is not also a symbolic link. We just want the size of the files in a directory, so we don't want to follow symbolic links. (Symbolic links may also cause reference loops, leading to a runaway condition.)

If we find a directory, we loop through it calling `entry_size()` for each file we encounter. This is a recursive loop, so we eventually end up with the size of the directory.

## Get this book's PDF version and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.



UNLOCK NOW

*Note: Keep your invoice handy. Purchases made directly from Packt don't require an invoice.*

# 11

# More Practical Ideas

We've used some useful techniques in this book, like optional values, containers, iterators, algorithms, smart pointers, and more. Let's now apply them to a few more practical ideas. The recipes in this chapter represent concepts that you may find useful, as you work on practical applications in the real world.

In this chapter, we cover the following recipes:

- Create a trie class for search suggestions
- Calculate the error sum of two vectors
- Build your own algorithm: split
- Leverage existing algorithms: gather
- Remove consecutive whitespace
- Convert numbers to words

## Create a trie class for search suggestions

A *trie*, sometimes called a *prefix tree*, is a type of search tree, commonly used for predictive text and other search applications. A trie is a recursive structure designed for depth-first searches, where each *node* is both a key and another trie.

A common use case is a *trie of strings*, where each node is a string in a sentence.

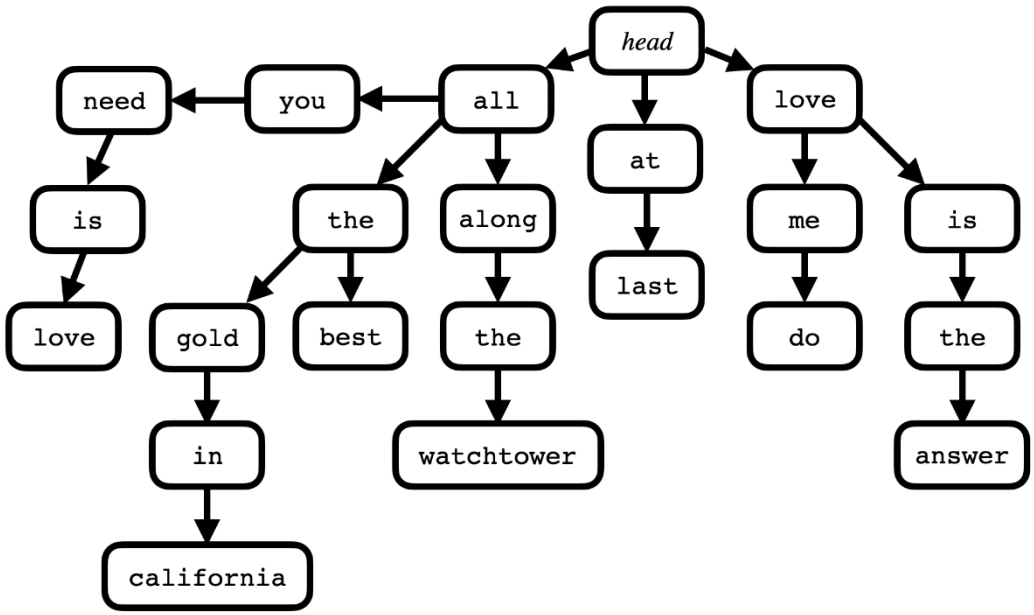



*Figure 11.1: A trie of strings*

We often start a search at the *head* of a trie, looking for sentences that begin with a specific word. In this example, when I search for `all` I get three nodes: `you`, `the`, and `along`. If I search for `love` I get `me` and `is`.

A string trie is commonly used for creating search suggestions. Here we will implement a string trie using `std::map` for the trie structure.

## How to do it

In this recipe we create a recursive `trie` class that stores nodes in a `std::map` container. It's a simple solution for a small in-memory trie. This is a rather large class, so we'll only show the important parts here.

For the full source code, please see the source code on *github*.

- We have one convenience alias:

  ```
  using ilcstr = initializer_list<const char *>;
  ```

  We use `ilcstr` in for searching the trie.

- We'll put this class in a private namespace to avoid collisions:

```
namespace bw {
    using std::map;
    using std::deque;
    using std::initializer_list;
```

We have a few `using` statements in this namespace for convenience.

- The class itself is called trie. It has three data members:

```
class trie {
    using get_t = deque<deque<string>>;
    using nodes_t = map<string, trie>;
    using result_t = optional<const trie*>;

    nodes_t nodes {};
    mutable get_t result_dq {};
    mutable deque<string> prefix_dq {};
```

The trie class has a few local type aliases:

  - `get_t` is a deque of deque of string, used for string results.
  - `nodes_t` is a map of trie classes with string keys.
  - `result_t` is an optional of a pointer to a trie, for returning search results. An empty trie is a valid result, so we use an optional value.

The `nodes` object is used for holding a recursive `map` of nodes, where each node on a `trie` is another `trie`.

- The `public` interface often calls utility functions in the private interface. For example, the `insert()` method takes an `initializer_list` and calls the private function `p_insert()`:

```
void insert(const ilcstr& il) {
    p_insert(il.begin(), il.end());
}
```

The private `p_insert()` function does the work of inserting elements:

```
template <typename It>
void p_insert(It it, It end_it) {
    if (it == end_it) return;
    nodes[*it].p_insert(++it, end_it);
}
```

This facilitates the recursive function calls necessary to navigate the `trie` of `trie` classes. Note that referencing a key that does not appear in a `map` creates an empty element with that key. So, the line that calls `_insert()` on a `nodes` element creates an empty `trie` object if the element doesn't already exist.

- The `get()` method returns a `get_t` object, which is an alias for a `deque` of `deque` of `string`. This allows us to return multiple sets of results:

```
get_t& get() const {
    result_dq.clear();
    deque<string> dq {};
    p_get(dq, result_dq);
    return result_dq;
}
```

The `get()` method calls the private `p_get()` function, which recursively traverses the trie:

```
void p_get(deque<string>& dq, get_t& r_dq) const {
    if(empty()) {
        r_dq.emplace_back(dq);
        dq.clear();
    }
    for(const auto& p : nodes) {
        dq.emplace_back(p.first);
        p.second.p_get(dq, r_dq);
    }
}
```

- The `find_prefix()` function returns a `deque` with all matches to a partial string:

```
deque<string>& find_prefix(const char * s) const {
    p_find_prefix(s, prefix_dq);
    return prefix_dq;
}
```

The public interface calls the private function `p_find_prefix()`:

```
void p_find_prefix(const string& s, auto& pre_dq) const {
    if (empty()) return;
    for (const auto& [k, v] : nodes) {
        if (k.starts_with(s)) {
            pre_dq.emplace_back(k);
            v.p_find_prefix(k, pre_dq);
        }
    }
}
```

The private `p_find_prefix()` function traverses the `trie` recursively, comparing the prefix with beginning of each key. The `starts_with()` method is new with C++20. With an older STL, you could use the `find()` method and check the return value for 0:

```
If (k.find(s) == 0) {
    ...
```

- The `search()` function returns an `optional<const trie*>`, aliased as `result_t`. It has two overloads:

```
result_t search(const ilcstr& il) const {
    return p_search(il.begin(), il.end());
}

result_t search(const string& s) const {
    const ilcstr il {s.c_str()};
    return p_search(il.begin(), il.end());
}
```

These methods pass iterators to the private member function `p_search()`, which does the work of the search:

```
template <typename It>
result_t p_search(It it, It end_it) const {
    if (it == end_it) return {this};
    auto found_it = nodes.find(*it);
    if (found_it == nodes.end()) return {};
    return found_it->second.p_search(++it, end_it);
}
```

The `p_search()` function searches recursively until it finds a match, then returns a node in the `result_t` object. If it finds no match, it returns the non-value `optional`.

- We also have two overloads of a `print_trie_prefix()` function. This function prints the contents of a `trie` from a prefix, used as a search key. One version uses a string for the prefix, the other uses an `initializer_list` of C-strings:

```
void print_trie_prefix(const bw::trie& t,
      const string& prefix) {
    auto& trie_strings = t.get();
    println("results for \"{}...\":", prefix);
    for (auto& dq : trie_strings) {
        print("{} ", prefix);
        for (const auto& s : dq) print("{} ", s);
        print("\n");
    }
}

void print_trie_prefix(const bw::trie& t,
        const ilcstr & prefix) {
    string sprefix {};
    for (const auto& s : prefix) sprefix += format("{} ", s);
    print_trie_prefix(t, sprefix);
}
```

These functions call the `get()` member function to retrieve the results from the trie.

- Now we can test the trie class in the `main()` function. First, we declare a `trie` and insert some sentences:

```
int main() {
    bw::trie ts {};
    ts.insert({ "all", "along", "the", "watchtower" });
    ts.insert({ "all", "you", "need", "is", "love" });
    ts.insert({ "all", "shook", "up" });
    ts.insert({ "all", "the", "best" });
    ts.insert({ "all", "the", "gold", "in", "california" });
    ts.insert({ "at", "last" });
    ts.insert({ "love", "the", "one", "you're", "with" });
    ts.insert({ "love", "me", "do" });
    ts.insert({ "love", "is", "the", "answer" });
    ts.insert({ "loving", "you" });
    ts.insert({ "long", "tall", "sally" });
    ...
```

The `insert()` calls pass an `initializer_list` with C-strings representing the words a sentence. Each of the strings are inserted into the hierarchy of the `trie`.

Now we can search the `trie`. Here's a simple search for the single string `"love"`.

```
const auto prefix = {"love"};
if (auto st = ts.search(prefix); st.have_result) {
    print_trie_prefix(*st.t, prefix);
}
print("\n");
```

This calls `ts.search()` with an `initializer_list` of one C-string, called `prefix`. The result, along with the `prefix` is then passed to the `print_trie_prefix()` function.

The output is:

```
  results for "love...":
  love is the answer
  love me do
  love the one you're with
```

Here's a search for a two-string prefix:

```
const auto prefix = {"all", "the"};
if (auto st = ts.search(prefix); st.have_result) {
    print_trie_prefix(*st.t, prefix);
}
print("\n");
```

Output:

```
  results for "all the ...":
  all the  best
  all the  gold in california
```

And here's a search for a partial prefix, using the `find_prefix()` function:

```
const char * prefix{ "lo" };
auto prefix_dq = ts.find_prefix(prefix);
for(const auto& s : prefix_dq) {
    println("match: {}-> {}", prefix, s);
    if (auto st = ts.search(s); st.have_result) {
        print_trie_prefix(*st.t, s);
    }
}
print("\n");
```

Output:

```
  match: lo -> long
  results for "long...":
  long tall sally
  match: lo -> love
  results for "love...":
  love is the answer
  love me do
  love the one you're with
  match: lo -> loving
  results for "loving...":
  loving you
```

The `find_prefix()` search returned several results, each of which we passed to a search of its own, resulting in several recursive searches.

## How it works

The data for the `trie` class is stored in recursive `map` containers. Each node in the `map` contains another `trie` object, which in turn has its own `map` node.

```
using nodes_t = map<string, trie>
```

The `p_insert()` function takes `begin` and `end` iterators and uses them to recursively call `p_insert()` on new nodes:

```
template <typename It>
void p_insert(It it, It end_it) {
    if (it == end_it) return;
    nodes[*it].p_insert(++it, end_it);
}
```

Likewise, the `p_search()` function recursively calls `p_search()` on the nodes it finds:

```
template <typename It>
result_t p_search(It it, It end_it) const {
    if (it == end_it) return {this};
    auto found_it = nodes.find(*it);
    if (found_it == nodes.end()) return {};
    return found_it->second.p_search(++it, end_it);
}
```

This recursive approach using `std::map` allows us to implement a `trie` class concisely and efficiently.

# Calculate the error sum of two vectors

Given two similar vectors that differ only by quantization or resolution, we can use the `inner_product()` algorithm to calculate an *error sum*, defined as:

$$e = \sum_{i=1}^{n} (a_i - b_i)^2$$

*Figure 11.2: Error sum definition*

Where *e* is the error sum: the sum of the squares of the differences between the series of points *i* to *n* in the two vectors, *a* and *b*.

We can use the `inner_product()` algorithm, from the `<numeric>` header, to calculate the error sum between two `vector` objects.

## How to do it

In this recipe we define two `vector` objects, each with a *sine wave.* One `vector` has values of type `double` and the other has type `int`. This gives us vectors that differ in *quantization*, because the `int` type cannot represent fractional values. We then use `inner_product()` to calculate the error sum between the two:

- In our `main()` function we define our `vectors` and a handy index variable:

```
int main() {
    constexpr size_t vlen {100};
    vector<double> ds(vlen);
    vector<int> is(vlen);
    size_t index{};
    ...
```

`ds` is the `vector` of `double` sine waves and `is` is the `vector` of `int` sine waves. Each `vector` has 100 elements to hold a sine wave. The `index` variable is used to initialize the vector objects.

- We generate the sine wave in the `vector` of `double`, using a loop and a lambda:

```
auto sin_gen = [&index]{
  return 5.0 * sin(index++ * 2 * pi / 100);
};
for (auto& v : ds) v = sin_gen();
```

The lambda captures a reference to the `index` variable so it can be incremented.

The `pi` constant is from the `std::numbers` library.

- We now have a `double` sine wave, and we can use it to derive the `int` version:

```
index = 0;
for (auto& v : is) {
    v = static_cast<int>(round(ds.at(index++)));
}
```

This takes each point from `ds`, rounds it, casts it to an `int`, and updates it in position in the `is` container.

- We display our sine waves with a simple loop:

```
for (const auto& v : ds) print("{:-5.2f} ", v);
print("\n\n");
for (const auto& v : is) print("{:-3d} ", v);
print("\n\n");
```

Our output is the sine waves as data points in the two containers:

```
 0.00  0.31  0.63  0.94  1.24  1.55  1.84  2.13  2.41  2.68  2.94  3.19
 3.42  3.64  3.85  4.05  4.22  4.38  4.52  4.65  4.76  4.84  4.91  4.96
 4.99  5.00  4.99  4.96  4.91  4.84  4.76  4.65  4.52  4.38  4.22  4.05
 3.85  3.64  3.42  3.19  2.94  2.68  2.41  2.13  1.84  1.55  1.24  0.94
 0.63  0.31  0.00 -0.31 -0.63 -0.94 -1.24 -1.55 -1.84 -2.13 -2.41 -2.68
-2.94 -3.19 -3.42 -3.64 -3.85 -4.05 -4.22 -4.38 -4.52 -4.65 -4.76 -4.84
-4.91 -4.96 -4.99 -5.00 -4.99 -4.96 -4.91 -4.84 -4.76 -4.65 -4.52 -4.38
-4.22 -4.05 -3.85 -3.64 -3.42 -3.19 -2.94 -2.68 -2.41 -2.13 -1.84 -1.55
-1.24 -0.94 -0.63 -0.31

  0   0   1   1   1   2   2   2   2   3   3   3   3   4   4   4   4   4
5   5   5   5   5   5   5   5   5   5   5   5   5   5   5   4   4   4   4
4   3   3   3   3   2   2   2   2   1   1   1   0   0   0  -1  -1  -1  -2
-2  -2  -2  -3  -3  -3  -3  -4  -4  -4  -4  -4  -5  -5  -5  -5  -5  -5  -5
```

```
   -5  -5  -5  -5  -5  -5  -5  -5  -4  -4  -4  -4  -4  -3  -3  -3  -3  -2  -2
   -2  -2  -1  -1  -1   0
```

- Now we calculate the error sum using `inner_product()`:

```
double errsum = inner_product(ds.begin(), ds.end(),
    is.begin(), 0.0, std::plus<double>(),
    [](double a, double b){ return pow(a - b, 2); });
print("error sum: {:.3f}\n\n", errsum);
```

The lambda expression returns the $(a_i - b_i)^2$ part of the formula. The `std::plus()` algorithm performs the sum operation.

Output:

```
  error sum: 7.304
```

## How it works...

The `inner_product()` algorithm computes a sum of products on the first input range. Its signature is:

```
T inner_product(InputIt1 first1, InputIt1 last1,
    InputIt2 first2, T init, BinaryOperator1 op1,
    BinaryOperator2 op2)
```

The function takes two binary operator functors, `op1` and `op2`. `op1` is for the *sum* and `op2` is for the *product*. We use `std::plus()` as the sum operator and a lambda as the product operator:

```
double errsum = inner_product(ds.begin(), ds.end(),
    is.begin(), 0.0, std::plus<double>(),
    [](double a, double b){ return pow(a - b, 2); });
```

The `init` parameter can be used as a starting value or bias. We pass it the literal value, 0.0.

The return value is the accumulated sum of the products.

## There's more...

We can calculate an accumulated error sum by putting `inner_product()` in a loop:

```
println("accumulated error:");
for (auto it {ds.begin()}; it != ds.end(); ++it) {
    double accumsum = inner_product(ds.begin(), it,
        is.begin(), 0.0, std::plus<double>(),
        [](double a, double b){ return pow(a - b, 2); });
    print("{:-5.2f} ", accumsum);
}
print("\n");
```

Output:

```
accumulated error:
 0.00  0.00  0.10  0.24  0.24  0.30  0.51  0.53  0.55  0.72  0.82  0.82  0.86
1.04  1.16  1.19  1.19  1.24  1.38  1.61  1.73  1.79  1.82  1.82  1.83  1.83
1.83  1.83  1.83  1.84  1.86  1.92  2.04  2.27  2.42  2.46  2.47  2.49  2.61
2.79  2.83  2.83  2.93  3.10  3.12  3.14  3.35  3.41  3.41  3.55  3.65  3.65
3.75  3.89  3.89  3.95  4.16  4.19  4.20  4.37  4.47  4.48  4.51  4.69  4.82
4.84  4.84  4.89  5.03  5.26  5.38  5.44  5.47  5.48  5.48  5.48  5.48  5.48
5.48  5.49  5.51  5.57  5.70  5.92  6.07  6.12  6.12  6.14  6.27  6.45  6.48
6.48  6.59  6.75  6.77  6.80  7.00  7.06  7.07  7.21
```

This could be useful in some statistical applications.

# Build your own algorithm: split

The STL has a rich algorithm library. Yet, on occasion you may find it missing something you need. One common need is a *split* function.

A split function splits a string on a character separator. For example, here's a Unix /etc/passwd file from a standard Debian installation:

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
```

Each field is separated by a colon `:` character, where the fields are:

- Login name
- Optional encrypted passwd
- User ID
- Group ID
- Username or comment
- Home directory
- Optional command interpreter

This is a standard file in POSIX-based operating systems, and there are others like it.

Most scripting languages include a built-in function for splitting a string on a separator. There are simple ways to do this in C++. Still, `std::string` is just another container in the STL, and a generic algorithm that splits a container on a separator could be a useful addition to the toolbox. So, let's build one.

## How to do it

In this recipe we'll build a generic algorithm that splits a container on a separator and puts the results in a target container.

- Our algorithm is in the `bw` namespace, to avoid collisions with `std`:

```
namespace bw {
    template<typename It, typename Oc, typename V,
        typename Pred>
    It split(It it, It end_it, Oc& dest,
            const V& sep, Pred& f) {
        using SliceContainer = typename Oc::value_type;
        while (it != end_it) {
            SliceContainer dest_elm{};
            auto slice {it};
            while (slice != end_it) {
                if (f(*slice, sep)) break;
                dest_elm.push_back(*slice++);
            }
            dest.push_back(dest_elm);
            if (slice == end_it) return end_it;
            it = ++slice;
        }
        return it;
```

```
    }
};
```

The `split()` algorithm searches a container for separators and collects the separated slices into a new output container, where each slice is a container within the output container.

We want the `split()` algorithm to be as generic as possible, just like those in the algorithm library. This means that all the parameters are templated, and the code will work with a variety of parameter types.

- First, let's look at the template parameters
  - `It` is the input iterator type for the source container.
  - `Oc` is the output container type. This is a container of containers.
  - `V` is the separator type.
  - `Pred` is for the predicate functor.

  Our output type is a container of containers. It needs to hold containers of slices. It could be `vector<string>`, where the string values are slices, or `vector<vector<int>>`, where the inner `vector<int>` contains the slices. This means we need to derive the type of the inner container from the `Oc` container type. We do that with the using declaration in the body of the function.

```
using SliceContainer = typename Oc::value_type;
```

This is also why we cannot use an output iterator for the output parameter. An output iterator cannot determine the type of its contents and its `value_type` is set to `void`.

We use `SliceContainer` to define a temporary container that is added to the output container with the statement:

```
dest.push_back(dest_elm);
```

- The predicate is a binary operator that compares an input element with the separator. We include a default equality operator in the `bw` namespace:

```
constexpr auto eq = [](const auto& el, const auto& sep) {
    return el == sep;
};
```

- We also include a specialization of `split()` that uses the `eq` operator by default:

```
template<typename It, typename Oc, typename V>
It split(It it, const It end_it, Oc& dest, const V& sep) {
    return split(it, end_it, dest, sep, eq);
}
```

- Because splitting string objects is a common use case for this algorithm, we include a helper function for that specific purpose:

```
template<typename Cin, typename Cout, typename V>
Cout& strsplit(const Cin& str, Cout& dest, const V& sep) {
    split(str.begin(), str.end(), dest, sep, eq);
    return dest;
}
```

- We test our split algorithm in `main()`, starting with a string object:

```
int main() {
    constexpr char strsep {':'};
    const string str
      {"sync:x:4:65534:sync:/bin:/bin/sync"};
    vector<string> dest_vs {};
    bw::split(str.begin(), str.end(), dest_vs, strsep,
      bw::eq);
    for (const auto& e : dest_vs) print("[{}] ", e);
    print("\n");
}
```

We use a string from the `/etc/passwd` file to test our algorithm, with this result:

```
  [sync] [x] [4] [65534] [sync] [/bin] [/bin/sync]
```

- It's even simpler using our `strsplit()` helper function:

```
vector<string> dest_vs2 {};
bw::strsplit(str, dest_vs2, strsep);
for (const auto& e : dest_vs2) print("[{}] ", e);
print("\n");
```

Output:

```
   [sync] [x] [4] [65534] [sync] [/bin] [/bin/sync]
```

This would make it easy to parse the `/etc/passwd` file.

- Of course, we can use the same algorithm with any container:

```
constexpr int intsep {-1};
vector<int> vi {1, 2, 3, 4, intsep, 5, 6, 7, 8, intsep, 9,
  10, 11, 12};
vector<vector<int>> dest_vi {};
bw::split(vi.begin(), vi.end(), dest_vi, intsep);
for (const auto& v : dest_vi) {
    string s;
    for (const auto& e : v) s += format("{}", e);
    print("[{}] ", s);
}
print("\n");
```

Output:

```
   [1234] [5678] [9101112]
```

In this example we use a simple vector of `int` with `-1` as the separator value.

## How it works

The split algorithm itself is relatively simple. The magic in this recipe is in the use of templates to make it as generic as possible.

The derived type in the `using` declaration allows us to create a container for use with the output container:

```
using SliceContainer = typename Oc::value_type;
```

This gives us a `SliceContainer` type that we can use to create a container for the slices:

```
SliceContainer dest_elm{};
```

This is a temporary container that is added to the output container for each slice:

```
dest.push_back(dest_elm);
```

## There's more...

Because our algorithm uses the `push_back()` method on the output container, it will not work on a container that doesn't support a `push_back()` method.

We can use a `concept` requiring a `push_back()`-compatible container in our `split` template:

```
template<typename Oc>
concept PushBackContainer =
    requires(Oc c, typename Oc::value_type v) {
        c.push_back(v);
    };
```

We apply the `concept` in our template like this:

```
    template<typename It, PushBackContainer Oc, typename V,
      typename Pred>
```

Now if we try to use our split algorithm with an incompatible output container, it will fail to compile:

```
std::forward_list<string> dest_ls2 {};
bw::strsplit(str, dest_ls2, strsep);
for (const auto& e : dest_ls2) print("[{}] ", e);
print("\n");
```

When we try to compile this code with GCC, we get a cascade of errors that includes this useful message:

```
   26 |     It split(It it, const It end_it, Oc& dest, const V& sep, const Pred&
f) {
      |        ^~~~~
working.cpp:26:8: note: template argument deduction/substitution failed:
working.cpp:26:8: note: constraints not satisfied
```

If we change this to a compatible container, for example, `list`, it now compiles and runs as expected:

```
std::list<string> dest_ls2 {};
```

Output:

```
[sync] [x] [4] [65534] [sync] [/bin] [/bin/sync]
```

# Leverage existing algorithms: gather

`gather()` is an example of an algorithm that leverages existing algorithms.

The `gather()` algorithm takes a pair of container iterators, moves the elements that satisfy a predicate toward a *pivot* position within the sequence, and returns a pair of iterators that contain the elements that satisfy the predicate.

For example, we could use a gather algorithm to sort all the even numbers to the mid-point of a vector:

```
vector<int> vint{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
gather(vint.begin(), vint.end(), mid(vint), is_even);
for (const auto& el : vint) print("{}", el);
```

Our output is:

```
1302468579
```

Notice that the even numbers are all in the middle of the output.

In this recipe, we will implement a `gather()` algorithm using standard STL algorithms.

## How to do it

Our gather algorithm uses the `std::stable_partition()` algorithm to move items before the pivot iterator and again to move items past the pivot.

- We put the algorithm in the `bw` namespace to avoid naming collisions

```
namespace bw {
    using std::stable_partition;
    using std::pair;
    using std::not_fn;
```

```
    template <typename It, typename Pred>
    pair<It, It> gather(It first, It last, It pivot,
      Pred pred) {
        return {stable_partition(first, pivot,
          not_fn(pred)),
        stable_partition(pivot, last, pred)};
    }
...
};
```

The `gather()` algorithm returns a pair of iterators, returned from two calls to `stable_partition()`.

- We also include some helper lambdas in the `bw` namespace:

```
constexpr auto midit = [](auto& v) {
    return v.begin() + (v.end() - v.begin()) / 2;
};
constexpr auto is_even = [](auto i) {
    return i % 2 == 0;
};
constexpr auto is_even_char = [](auto c) {
    if (c >= '0' && c <= '9') return (c - '0') % 2 == 0;
    else return false;
};
```

These three lambdas are as follows:

  - `midit` returns an iterator at the midpoint of a container, for use as a pivot point.
  - `is_even` returns `true` if the value is even, for use as a predicate.
  - `is_even_char` returns Boolean true if the value is a character between `'0'` and `'9'` and is even, for use as a predicate.

- We call `gather()` from the `main()` function with a `vector` of `int` like this:

```
int main() {
    vector<int> vint {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    auto gathered_even = bw::gather(vint.begin(),
      vint.end(), bw::midit(vint), bw::is_even);
    for (const auto& el : vint) print("{}", el);
```

```
    print("\n");
}
```

Our output shows that the even numbers have been gathered in the middle:

```
1302468579
```

- The `gather()` function returns a `pair` of iterators that contain just the even values:

```
auto& [it1, it2] = gathered_even;
for (auto it {it1}; it < it2; ++it) print("{}", *it);
print("\n");
```

Output:

```
02468
```

We can set the pivot point to the `end()` iterators:

```
bw::gather(vint.begin(), vint.end(), vint.begin(),
  bw::is_even);
for (const auto& el : vint) print("{}", el);
print("\n");

bw::gather(vint.begin(), vint.end(), vint.end(),
  bw::is_even);
for (const auto& el : vint) print("{}", el);
print("\n");
```

Output:

```
0246813579
1357902468
```

- Because `gather()` is iterator-based, we can use it with any container. Here's a string of character digits:

```
string jenny {"867-5309"};bw::gather(jenny.begin(), jenny.end(),
jenny.end(),
  bw::is_even_char);
```

```
for (const auto& el : jenny) print("{}", el);
print("\n");
```

This moves all the even digits to the end of the `string`:

Output:

```
7-539860
```

## How it works

The `gather()` function uses the `std::stable_partition()` algorithm to move the elements that match the predicate to the pivot point.

`gather()` has two calls to `stable_partition()`, one with the predicate and one with the predicate negated:

```
template <typename It, typename Pred>
pair<It, It> gather(It first, It last, It pivot, Pred pred) {
    return { stable_partition(first, pivot, not_fn(pred)),
             stable_partition(pivot, last, pred) };
}
```

The iterators returned from the two `stable_partition()` calls are returned in the `pair`.

# Remove consecutive whitespace

When receiving input from users, it's common to end up with excessive consecutive whitespace characters in your strings. This recipe presents a function for removing consecutive spaces, even when it includes tabs or other whitespace characters.

## How to do it

This function leverages the `std::unique()` algorithm to remove consecutive whitespace characters from a string.

- In the `bw` namespace, we start with a function to detect whitespace:

```
template<typename T>
bool isws(const T& c) {
    constexpr const T whitespace[] {" \t\r\n\v\f"};
    for (const T& wsc : whitespace) {
        if (c == wsc) return true;
```

```
    }
    return false;
}
```

This templated `isws()` function should work with any character type.

- The `delws()` function uses `std::unique()` to erase consecutive whitespace in a string:

```
string delws(const string& s) {
    string outstr {s};
    auto its = unique(outstr.begin(), outstr.end(),
        [](const auto &a, const auto &b) {
            return isws(a)&&isws(b);
        });
    outstr.erase(its, outstr.end());
    outstr.shrink_to_fit();
    return outstr;
}
```

`delws()` makes a copy of the input `string`, removes consecutive whitespace, and returns the new `string`.

- We call it with a string from `main()`:

```
int main() {
    const string s {"big     bad    \t   wolf"};
    const string s2 {bw::delws(s)};
    println("[{}]", s);
    println("[{}]", s2);
    return 0;
}
```

Output:

```
[big     bad           wolf]
[big bad wolf]
```

## How it works

This function uses the `std::unique()` algorithm with a comparison lambda to find consecutive whitespace in a `string` object.

The comparison lambda calls our own `isws()` function to determine if we have found consecutive whitespace:

```
auto its = unique(outstr.begin(), outstr.end(),
    [](const auto &a, const auto &b) {
        return isws(a)&&isws(b);
    });
```

We could use the `isspace()` function from the standard library, but it's a standard C function that depends on a narrowing type conversion from `int` to `char`. This may issue warnings on some modern C++ compilers and is not guaranteed to work without an explicit cast. Our `isws()` function uses a templated type and should work on any system and with any specialization of `std::string`.

## Convert numbers to words

Over the course of my career, I've used a lot of programming languages. When learning a new language, I like to have a project to work on that exposes me to the nuances of the language. The `numword` class is one of my favorite exercises for this purpose. I have written it in dozens of languages over the years, including several times in C and C++.

`numword` is a class that spells out a number in words. It can be useful for banking and accounting applications. It looks like this in use:

```
int main() {
    numword nw {};
    uint64_t n {};

    println("n is {}, {}", nw.getnum(), nw);
    nw = 3; println("n is {}, {}", nw.getnum(), nw);
    nw = 47; println("n is {}, {}", nw.getnum(), nw);
    nw = 73; println("n is {}, {}", nw.getnum(), nw);
    nw = 10012; println("n is {}, {}", nw.getnum(), nw);
    n = 100073; println("n is {}, {}", n, bw::numword{n});
    n = 1000000001; println("n is {}, {}", n, bw::numword{n});
    n = 123000000000; println("n is {}, {}", n, bw::numword{n});
    n = 1474142398007; println("n is {}, {}", n, nw(n));
    n = 999999999999999999; println("n is {}, {}", n, nw(n));
    n = 1000000000000000000; println("n is {}, {}", n, nw(n));
}
```

Output:

```
n is 3, three
n is 47, forty-seven
n is 100073, one hundred thousand seventy-three
n is 1000000001, one billion one
n is 123000000000, one hundred twenty-three billion
n is 1474142398007, one trillion four hundred seventy-four billion one hundred
forty-two million three hundred ninety-eight thousand seven
n is 999999999999999999, nine hundred ninety-nine quadrillion nine hundred ninety-
nine trillion nine hundred ninety-nine billion nine hundred ninety-nine million
nine hundred ninety-nine thousand nine hundred ninety-nine
n is 1000000000000000000, error
```

## How to do it

This recipe originated as an exercise in creating production-ready code. For that reason, it's in three different files:

- `numword.h` is the header/interface file for the `numword` class.
- `numword.cpp` is the implementation file for the `numword` class.
- `numword-test.cpp` is the application file for testing the `numword` class.

The class itself is about 180 lines of code so we'll just cover the highlights here. You can find the full source code on *Github* (`https://github.com/PacktPublishing/CPP-STL-Cookbook-Second-Edition/tree/main/chap11/numword`).

In the `numword.h` file, we put the class in the `bw` namespace and start with some using statements:

```
namespace bw {
    using std::string;
    using std::string_view;
    using numnum = uint64_t;
    using bufstr = std::unique_ptr<string>;
```

We use `string` and `string_view` objects throughout the code.

`uint64_t` is our primary integer type because it will hold very large numbers. Because the class is called `numword`, I alias it as `numnum` for this purpose.

`bufstr` is the main output buffer. It's a `string` wrapped in a `unique_ptr`, which handles the memory management for automatic RAII compliance.

- We also have a few constants for various purposes:

```
constexpr numnum maxnum = 999'999'999'999'999'999;
constexpr int zero_i {0};
constexpr int five_i {5};
constexpr numnum zero {0};
constexpr numnum ten {10};
constexpr numnum twenty {20};
constexpr numnum hundred {100};
constexpr numnum thousand {1000};
```

The `maxnum` constant translates to *nine hundred ninety-nine quadrillion nine hundred ninety-nine trillion nine hundred ninety-nine billion nine hundred ninety-nine million nine hundred ninety-nine thousand nine hundred ninety-nine*, which should be sufficient for most purposes.

The rest of the `numnum` constants are used to avoid literals in the code.

The main data structures are `constexpr` arrays of `string_view` objects, representing the words used in the output. The `string_view` class works great for constants:

```
constexpr string_view errnum {"error"};
constexpr string_view _singles[] {
    "zero", "one", "two", "three", "four", "five", "six",
    "seven", "eight", "nine"
};
constexpr string_view _teens[] {
    "ten", "eleven", "twelve", "thirteen", "fourteen",
    "fifteen", "sixteen", "seventeen", "eighteen",
    "nineteen"
};
constexpr string_view _tens[] {
    errnum, errnum, "twenty", "thirty", "forty", "fifty",
    "sixty", "seventy", "eighty", "ninety",
};
constexpr string_view _hundred_string = "hundred";
constexpr string_view _powers[] {
    errnum, "thousand", "million", "billion", "trillion",
```

```
    "quadrillion"
};
```

The words are grouped into sections, useful in translating numbers to words. Many languages use a similar breakdown so this structure should translate easily to those languages.

- The `numword` class has a few private members:

```
class numword {
    bufstr _buf {std::make_unique<string>(string{})};
    numnum _num {};
    bool _hyphen_flag {false};
```

  - `_buf` is the output string buffer. Its memory is managed by a `unique_ptr`.
  - `_num` holds the current numeric value.
  - `_hyphen_flag` is used during the translation process to insert a hyphen between words, rather than a space character.

- These private methods are used to manipulate the output buffer:

```
void clearbuf();
size_t bufsize();
void appendbuf(conststring&s);
void appendbuf(const string_view&s);
void appendbuf(const char c);
void appendspace();
```

- There is also a `pow_i()` private method used to calculate *xy* with `numnum` types:

```
numnum pow_i(const numnum n, const numnum p);
```

`pow_i()` is used to discriminate parts of the numeric value for word output.

- The public interface includes constructors and various ways to call the `words()` method, which does the work of translating a `numnum` to a string of words:

```
numword(const numnum& num = 0) : _num(num) {}
numword(const numword& nw) : _num(nw.getnum()) {}
const char * version() const { return _version; }
void setnum(const numnum& num) { _num = num; }
```

```
numnum getnum() const { return _num; }
numnum operator=(const numnum#);
conststring&words();
const string& words(const numnum#);
const string& operator() (const numnum& num) {
    return words(num);
};
```

- In the implementation file, `numword.cpp`, the bulk of the work is handled in the `words()` member function:

```
const string& numword::words( const numnum& num ) {
    numnum n {num};
    clearbuf();
    if (n > maxnum) {
        appendbuf(errnum);
        return *_buf;
    }
    if (n == 0) {
        appendbuf(_singles[n]);
        return *_buf;
    }
    // powers of 1000
    if (n >= thousand) {
        for(int i{ five_i }; i > zero_i; --i) {
            numnum power{ pow_i(thousand, i) };
            numnum _n {( n - ( n % power ) ) / power};
            if (_n) {
                int index = i;
                numword _nw {_n};
                appendbuf(_nw.words());
                appendbuf(_powers[index]);
                n -= _n * power;
            }
        }
    }
    // hundreds
    if (n >= hundred && n < thousand) {
        numnum _n {( n - ( n % hundred ) ) / hundred};
        numword _nw {_n};
        appendbuf(_nw.words());
```

```
        appendbuf(_hundred_string);
        n -= _n * hundred;
    }
    // tens
    if (n >= twenty && n < hundred) {
        numnum _n {( n - ( n % ten ) ) / ten};
        appendbuf(_tens[_n]);
        n -= _n * ten;
        _hyphen_flag = true;
    }
    // teens
    if (n >= ten && n < twenty) {
        appendbuf(_teens[n - ten]);
        n = zero;
    }
    // singles
    if (n > zero && n < ten) {
        appendbuf(_singles[n]);
    }
    return *_buf;
}
```

Each part of the function peels off part of the number with a *modulus of a power of ten*, recursively in the case of the thousands, and appending strings from the `string_view` constant arrays.

- There are three overloads of `appendbuf()`. One appends a string:

```
void numword::appendbuf(const string& s) {
    appendspace();
    _buf->append(s);
}
```

Another appends a `string_view`:

```
void numword::appendbuf(const string_view& s) {
    appendspace();
    _buf->append(s.data());
}
```

And the third appends a single character:

```
void numword::appendbuf(const char c) {
    _buf->append(1, c);
}
```

- The `appendspace()` method appends a space character or a hyphen, depending on context:

```
void numword::appendspace() {
    if(bufsize()) {
        appendbuf(_hyphen_flag ? _hyphen : _space);
        _hyphen_flag = false;
    }
}
```

- The `numword-test.cpp` file is the testing environment for `bw::numword`. It includes a formatter specialization:

```
template<>
struct std::formatter<bw::numword>: std::formatter<std::string> {
    template<typename FormatContext>
    auto format(const bw::numword& nw, FormatContext& ctx)
      const {
        bw::numword _nw{nw};
        return std::formatter<std::string,
          char>::format(_nw.words(), ctx);
    }
};
```

This allows us to pass a `bw::numword` object directly to `println()`.

- In `main()`, we declare a `bw::numword` object and a `bw::numnum` for use in testing:

```
int main() {
    bw::numword nw {};
    bw:numnum n {};

    bw::print("n is {}, {}\n", nw.getnum(), nw);
    ...
```

The numword object is initialized to zero, giving us this output from our `println()` statement:

```
n is 0, zero
```

We test a variety of ways to call numword:

```
nw = 3; println("n is {}, {}", nw.getnum(), nw);
nw = 47; println("n is {}, {}", nw.getnum(), nw);
...
n = 100073; println("n is {}, {}", n, bw::numword{n});
n = 1000000001; println("n is {}, {}", n, bw::numword{n});
...
n = 474142398123; println("n is {}, {}", n, nw(n));
n = 1474142398007; println("n is {}, {}", n, nw(n));
...
n = 999999999999999999; println("n is {}, {}", n, nw(n));
n = 1000000000000000000; println("n is {}, {}", n, nw(n));
```

Output:

```
n is 3, three
n is 47, forty-seven
...
n is 100073, one hundred thousand seventy-three
n is 1000000001, one billion one
...
n is 474142398123, four hundred seventy-four billion one hundred forty-two
million three hundred ninety-eight thousand one hundred twenty-three
n is 1474142398007, one trillion four hundred seventy-four billion one
hundred forty-two million three hundred ninety-eight thousand seven
...
n is 999999999999999999, nine hundred ninety-nine quadrillion nine hundred
ninety-nine trillion nine hundred ninety-nine billion nine hundred ninety-
nine million nine hundred ninety-nine thousand nine hundred ninety-nine
n is 1000000000000000000, error
```

## How it works

This class is significantly driven by the data structures. By organizing `string_view` objects into arrays, we can easily translate scalar values into corresponding words:

```
appendbuf(_tens[_n]);  // e.g., _tens[5] = "fifty"
```

The rest of it is mostly the math:

```
numnum power {pow_i(thousand, i)};
numnum _n {( n - ( n % power ) ) / power};
if (_n) {
    int index = i;
    numword _nw {_n};
    appendbuf(_nw.words());
    appendbuf(_powers[index]);
    n -= _n * power;
}
```

## There's more...

I also have a utility that uses the `numword` class to tell time in words. Its output looks like this:

```
$ ./saytime
three past five
```

In test mode, it gives this output:

```
$ ./saytime test
00:00 midnight
00:01 one past midnight
11:00 eleven o'clock
12:00 noon
13:00 one o'clock
12:29 twenty-nine past noon
12:30 half past noon
12:31 twenty-nine til one
12:15 quarter past noon
12:30 half past noon
12:45 quarter til one
11:59 one til noon
```

```
23:15 quarter past eleven
23:59 one til midnight
12:59 one til one
13:59 one til two
01:60 OOR
24:00 OOR
```

I leave its implementation as an exercise for the reader.

## Subscribe to Deep Engineering

Join thousands of developers and architects who want to understand how software is changing, deepen their expertise, and build systems that last.

Deep Engineering is a weekly expert-led newsletter for experienced practitioners, featuring original analysis, technical interviews, and curated insights on architecture, system design, and modern programming practice.

Scan the QR or visit the link to subscribe for free.



`https://packt.link/deep-engineering-newsletter`

# 12

# C++26 and the Future

Where C++20 and C++23 introduced significant additions to the STL, C++26 focuses on refinement and defect resolution rather than major new library features. The most notable advances proposed in the current C++26 drafts are in the core language, with features such as *reflection* and *contracts* laying groundwork for future standard library evolution. The STL remains largely stable in this release, gaining a few incremental improvements rather than any new components.

In this chapter, I'll briefly review a few of the more impactful features proposed in the C++26 standard update. Bear in mind that, as of this writing, these features are not implemented in any of the currently released compilers.

## Static reflection

The *static reflection* proposal aims to allow programs to introspect types, members, bases, attributes, and more, without relying on macros or code generation. This represents a foundational language feature that could replace many template metaprogramming hacks with explicit, readable constructs.

- Using reflections, we can iterate over class data members and convert them to JSON, XML, or binary formats without manual glue code. Adding or removing a field automatically updates the serialized form.
- Class members can be mapped to database columns directly from reflected metadata. Table schemas, queries, and migrations can be generated consistently from the C++ type definitions, eliminating duplicated schema descriptions.

- Language bindings (e.g., Python, JavaScript, or C APIs) can be generated directly by C++ code using reflection, rather than external generators. Reflected type information could drive which functions, classes, and fields are exposed and how they are used.
- Tools such as automatic logging, validation, documentation generation, and UI inspection can be expressed as normal C++ libraries. With standardized reflection, these facilities become portable and type-safe without relying on fragile macros or external parsers.

Let's consider a simple hypothetical example, using the proposed reflections syntax:

```
struct Animal {
    std::string name;
    int feet;
    double strength;
};

template <typename T>
void print_members(const T& obj) {
    constexpr auto members = std::meta::members_of(^T);

    for constexpr (auto m : members) {
        std::println(
            "{}: {}",
            std::meta::name_of(m),
            obj.*std::meta::pointer_of(m)
        );
    }
}

int main() {
    Animal tiger {"Hobbes", 4, 7.52};
    print_members(tiger);
}
```

In this example, the template function `print_members()` uses static reflection to print the members of the class passed in `obj`. The functions in the `std::meta` namespace are *compile-time* reflection queries that produce constant expressions. While the returned values can be used in ordinary C++ expressions, the functions themselves operate at compile time.

`meta::members_of()` returns a sequence of reflected members of the given type, representing its non-static data members in declaration order. This allows your code to inspect and operate on a type's structure.

`meta::name_of()` returns the name of a data member and `meta::pointer_of` returns a constant pointer-to-member which can be used to access the value of the data member.

Output:

```
name: Hobbes
feet: 4
strength: 7.52
```

**Note**

This example is hypothetical based on the current proposal. At the time of writing, none of the major compilers yet support the *static reflections* feature, so this code cannot yet be compiled or tested.

## Contracts

Scheduled for inclusion in C++26, *contracts* provide a practical way to make correctness requirements explicit and enforceable. This supersedes many uses of `assert` and other less enforceable techniques such as comments and coding conventions. Contracts can materially improve both code quality and developer effectiveness.

Preconditions and postconditions state what a function requires and guarantees. This reduces misuse and eliminates guesswork for callers. Additionally, the reader no longer must infer intent from defensive code or documentation. The intent is declared where it matters.

For example:

```
int divide(int numerator, int denominator)
    pre(denominator != 0 &&
        numerator % denominator == 0)
    post(r : r * denominator == numerator)
{
    return numerator / denominator;
}
```

Now the correctness logic is located in the function itself. When assumptions change, there is a single authoritative place to update them. This should lead to simpler implementations, fewer latent bugs, and much easier debugging and maintenance.

> **Note**
>
> At the time of writing, none of the major compilers yet support C++26 *contracts* so this code cannot yet be compiled or tested.

## Structural pattern matching

A proposed C++26 feature, *structural pattern matching,* would provide a practical replacement for chains of `if`/`else`, `switch`, and other obtuse unpacking logic, with code that directly reflects structure and intent. It can substantially reduce boilerplate while improving clarity and correctness.

For example, code that may have relied on `if`/`else` chains like this:

```
using value = std::variant<int, double, const char*>;

void print_value(const value& v) {
    if (std::holds_alternative<int>(v)) {
        std::println("int: {}", std::get<int>(v));
    } else if (std::holds_alternative<double>(v)) {
        std::println("double: {:.2f}", std::get<double>(v));
    } else if (std::holds_alternative<const char*>(v)) {
        std::println("string: {}", std::get<const char*>(v));
    }
}

int main() {
    print_value(42);
    print_value(3.14159);
    print_value("hello");
}
```

Could be written more clearly using pattern matching like this:

```
void print_value(const value& v) {
    match (v) {
        case int i:
```

```
            std::println("int: {}", i);
            break;
        case double d:
            std::println("double: {:.2f}", d);
            break;
        case const char* s:
            std::println("string: {}", s);
            break;
    }
}
```

This more clearly expresses the intent of the code without sacrificing any functionality. Each `case` is explicit and self-contained, making control flow easier to view and harder to break during maintenance.

The proposed pattern matching feature would encourage a declarative mindset. Your code describes valid forms, handles them directly, and allows invalid forms to fail early and visibly. The result is shorter, more readable code that is more resistant to subtle bugs – especially as systems evolve over time.

**Note**

While this feature is proposed for C++26, there are ongoing discussions about syntax and methodologies. When finally approved and integrated into the language, it may look different than what we have discussed here.

## Other C++26 improvements

C++26 is expected to be more focused on stability and refinement than the more feature-heavy updates of C++20 and C++23. Some proposed refinements include:

- ***Expanded*** `constexpr`
  Broaden what code may be evaluated at compile time by relaxing constant-evaluation rules and making more library facilities `constexpr`-friendly. This would reduce run-time overhead by allowing more validation and setup logic to run during compilation, catching errors earlier without relying on template metaprogramming.
- ***Safer low-level programming***
  Clarify object lifetimes, evaluation order, and aliasing rules, reducing common sources of undefined behavior in low-level code. These refinements aim to make systems

programming more predictable and analyzable without sacrificing performance or requiring higher-level abstractions.

- ***Concurrency refinements***
  Incremental clarification of the concurrency and memory models, particularly around atomics and execution ordering. This is intended to make concurrent code more predictable and analyzable, improving synchronization without introducing a new concurrency model or breaking existing code.
- ***Library evolution***
  Proposed library changes emphasize refinement and consistency, improving usability and reasoning without disrupting established library design. Specific updates include making more of the standard library `constexpr`-friendly to shift some work from runtime to compile time.

C++26 is still in development and the features discussed here remain subject to change before ratification. That said, it's useful to see where the language is headed: toward clearer semantics, safer low-level code, and more compile-time evaluation. Treat these features as *forward-looking guidance*, not commitments. Be prepared to adapt as proposals are refined or deferred.

## Get this book's PDF version and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.

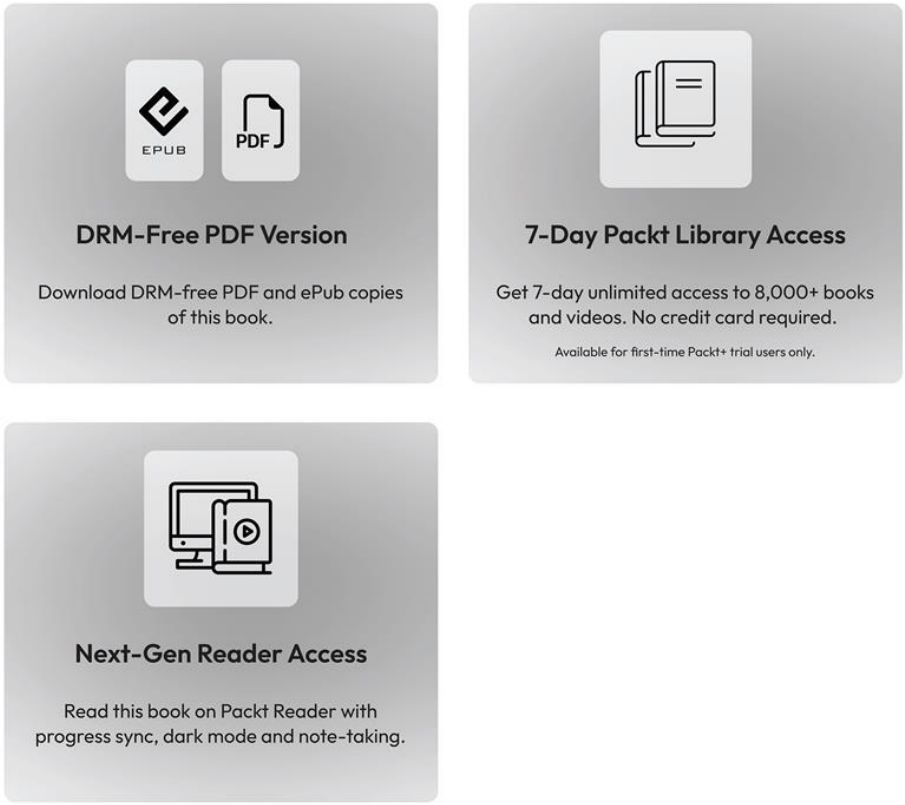



*Note: Keep your invoice handy. Purchases made directly from Packt don't require an invoice.*

# 13

# Unlock Your Exclusive Benefits

Your copy of this book includes the following exclusive benefits:




Follow the guide below to unlock them. The process takes only a few minutes and needs to be completed once.

# Unlock this Book's Free Benefits in 3 Easy Steps

## Step 1

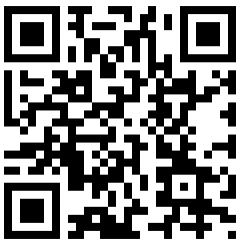
Keep your purchase invoice ready for *Step 3*. If you have a physical copy, scan it using your phone and save it as a PDF, JPG, or PNG.

For more help on finding your invoice, visit `https://www.packtpub.com/en-us/unlock?step=1`.

> **Note**
>
> If you bought this book directly from Packt, no invoice is required. After *Step 2*, you can access your exclusive content right away.

## Step 2

Scan the QR code or go to `packtpub.com/unlock`.



On the page that opens (similar to *Figure 16.1* on desktop), search for this book by name and select the correct edition.

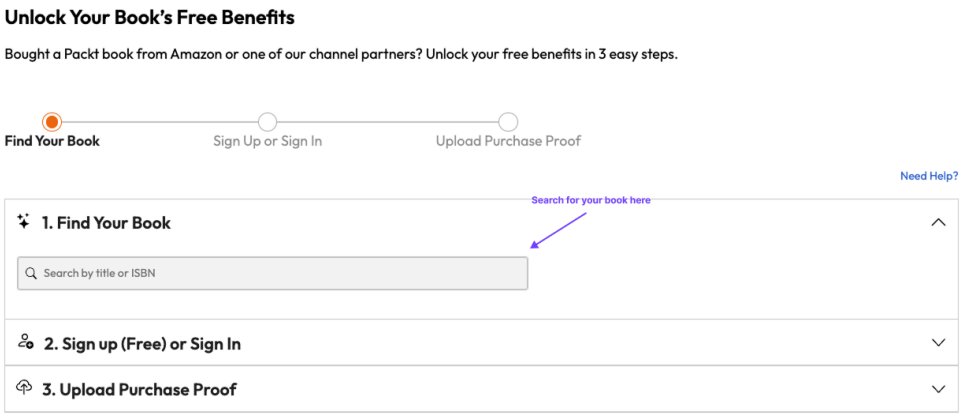



*Figure 13.1: Packt unlock landing page on desktop*

## Step 3

After selecting your book, sign in to your Packt account or create one for free. Then upload your invoice (PDF, PNG, or JPG, up to 10 MB). Follow the on-screen instructions to finish the process.

## Need Help

If you get stuck and need help, visit `https://www.packtpub.com/unlock-benefits/help` for a detailed FAQ on how to find your invoices and more. This QR code will take you to the help page.

**Note**

If you are still facing issues, reach out to `customercare@packt.com`.



packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

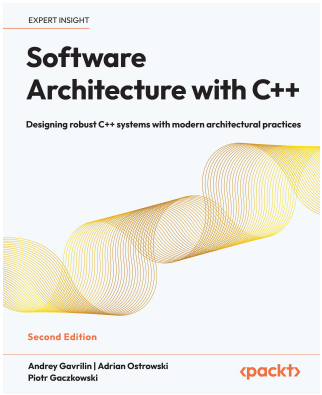
## Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

# Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:




**Software Architecture with C++**

Andrey Gavrilin, Adrian Ostrowski, Piotr Gaczkowski

ISBN: 9781803243016

- Apply architectural fundamentals to design scalable C++ systems
- Use modern C++ features to create maintainable and secure applications
- Implement architectural and system design patterns
- Design testable code and automate quality checks via CI/CD pipelines
- Manage dependencies and build systems using CMake and Conan
- Explore microservices, containers, and cloud-native practices in C++
- Improve observability with logging, tracing, and monitoring tools
- Build deployable C++ systems suited for modern development environments




**C++ Memory Management**

Patrice Roy

ISBN: 9781805129806

- Master the C++ object model to write more efficient and maintainable code
- Automate resource management to reduce manual errors and improve safety
- Customize memory allocation operators to optimize performance for specific applications
- Develop your own smart pointers to manage dynamic memory with greater control
- Adapt allocation behavior to meet the unique needs of different data types
- Create safe and fast containers to ensure optimal data handling in your programs
- Utilize standard allocators to streamline memory management in your containers

## Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packt.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

## Share your thoughts

Now you've finished *C++ STL Cookbook*, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.



`https://packt.link/r/1836204256`

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

# Index


## A

**allocated memory**
- managing, with std::unique_ptr 292 – 298

**associative containers** **48**
- map 49
- multimap 49
- multiset 49
- set 48
- unordered_map 49
- unordered_multimap 49
- unordered_multiset 49
- unordered_set 49

**auto variables**
- initializing 27

**await_ready() function** **369**

**await_resume() function** **370**

**await_suspend() function** **370**

## B

**backward iteration**
- with reverse iterator adaptor 116 – 119

**basic_string class** **221, 222**

**braced initialization**
- rules 25
- using 23 – 27

## C

**C++26**
- improvement 453, 454

**C-array**
- making safe, with span 28 – 31

**C-strings** **120**
- iterator, building with 120 – 122

**char_traits**
- string class, customizing with 259 – 262

**complex structures**
- initializing, from file input 255 – 259

**concat() function** **169**

**concatenation performance**
- discrepancy 243

**consecutive whitespace characters**
- removing 436, 437

**container**
- initializing, from ranges 14, 15
- modifying, with std::transform() algorithm 194 – 196
- uniform erase functions, using to delete elements 50 – 54

**container adapter** **49**

**container elements**
- into string 184 – 187

**container values**
- with std::clamp() function 202 – 204

**contains() method** **201**
- content, finding in ranges 16
- content, finding in strings 16

**content**
- finding, in ranges with contains() method 16
- finding, in strings with contains() method 16

**contracts** **451**

**copy algorithms** **180 – 183**
- working 183, 184

**coroutines** **128**
- execution, resuming 363 – 369
- execution, suspending 363 – 369
- generator, creating with 128 – 136
- using, for producer-consumer solution 370 – 375

**count word frequency**
- with map container 87 – 91

**custom keys**

unordered_map, using 74 – 77

## D

**data sequences permutations**
generating 214 – 217
**data sets**
with std::sample() algorithm 205 – 208
**deque**
RPN calculator, creating 80 – 87
**directory listing** **386 – 396**
**directory_iterator**
using, for files renaming 403 – 407
**disk usage counter**
creating 408 – 411
working 412

## E

**erase_if() function** **52**
**error sum**
calculating 423 – 427

## F

**file**
words, counting 254, 255
**file input**
complex structures, initializing 255 – 259
**files renaming**
with regex and directory_iterator 403 – 407
**find() function** **200**
**find_if() function** **200**
**first-in, first-out (FIFO)** **49**
**flags and values**
sharing, with std::atomic class 345 – 348, 352
**floating-point types** **349**
**fold expressions** **288**
types 289
using, for generic tuples 288 – 292
**format library**
text formatting with 228 – 233
**format text**
with print() function 4
with printIn() function 4

## G

**gather() algorithm**
using 433 – 436
**generator** **5, 123**
creating, with coroutines 128 – 136
creating, with iterators 123 – 128
**generator class**
working 7, 8
**get_stack_string() function** **85**

## I

**if and switch conditionals**
variable, initializing 35 – 37
**inner_product() algorithm** **426**
**insert_sorted() function** **63**
**integral types** **349**
**is_lock_free() method** **350**
**items**
deleting, from unsorted vector 55 – 58
**iterable range**
creating 105 – 108
**iterator adaptors**
using, to populate STL containers 112 – 116
**iterator class** **6**
**iterators**
categories 102
concepts 102 – 104
generator, creating with 123 – 128
traits, using with for STL compatibility 109 – 112
using 99 – 102

## J

**jthread** **327**
**jump table** **173**
creating, with mapped lambdas 173 – 176

# L

**Lambda expressions** **151**
capture-list 157
closure 152, 153
common parameters 171 – 173
concatenating, with recursion 168, 169
constexpr specifier 159
mutable modifier 158
noexcept attribute 159
parameters 158
syntax 157
trailing return type 159
usage 153 – 156
with algorithms 160 – 163
**last-in, first-out (LIFO)** **49**
**lazy evaluation** **128**
**lightweight string object**
string_view class, using as 223 – 228
**locks**
using, to share data 337 – 345
**logical conjunction** **170, 171**

# M

**manipulation functions**
using, with path object 382 – 385
**map class**
elements, inserting 64 – 69
keys, modifying 70 – 74
**mapped lambdas**
jump table, creating 173 – 176
**mdspan class** **8**
external data store, using 13
using, as multidimensional array 8 – 11
working 12, 13
**memory integrity**
preserving, with RAII 42 – 46
**merge sort containers** **217, 218**
**multidimensional array**
mdspan class, using 8 – 11
**multiple producers and consumers**
implementing 358 – 363
**mutually exclusive (mutex)** **337**
using, to share data 337 – 345

# N

**narrowing conversion** **25**
**numword class** **438 – 447**

# O

**object members**
sharing, with aliased smart pointers 308 – 310
**object sharing**
with std::shared_ptr class 299 – 302
**objects of unknown length**
iterating, with sentinel 120 – 123
**optional values**
managing, with std::optional class 267 – 271

# P

**pointers** **348**
**polymorphic wrapper**
creating, with std::function 164 – 167
**prefix tree** **415**
**print() function** **4, 5**
format text 4
strings, printing with 234 – 237
working 5
**println() function** **4, 5**
format text 4
**print_assoc() function** **53**
**printc() function** **51**
**println() function**
strings, printing with 234 – 237
**printv() function** **62**
**producer-consumer problem**
resolving, with std::condition_variable 354 – 358
**promise and future objects**
creating, with std::async() function 333
**psorted() function** **62**

# Q

**quick_delete() function** **56**

# R

**RAII**
using, to preserve memory integrity 42 – 46
**RPN calculator**
creating, with deque 80 – 87
**random number distribution generators**
comparing 314 – 316
**random number engines**
comparing 310 – 314
**random-access iterator**
creating 143 – 145
operator overloads 145, 146
validation code 146 – 148
working 149
**ranges**
container, initializing 14, 15
working 15, 16
**ranges library**
using 208 – 214
**ranges::contains_subrange() algorithm** **20**
**regular expressions (regex)**
strings, parsing with 263 – 265
using, for files renaming 403 – 407
**remove_value() function** **50**
**reverse iterator adaptor** **116**
backward, iterating with 116 – 119

# S

**STL algorithms**
running, with execution policies 334 – 337
**STL container types**
associative containers 48
container adapter 49
overview 48
sequential containers 48
**STL containers**
populating, with iterator adaptors 112 – 116
**STL string class** **221**
**search containers**
with std::find() function 197 – 200
**search directory and files**
with grep utility 397 – 402
**sentence length**
finding, with vector 92 – 95
**sentinel** **120**
objects of unknown length, iterating with 120
using, with C-strings 121, 122
**sequential containers** **48**
array 48
deque 48
list 48
vector 48
**set class**
user input, filtering 77 – 79
user input, sorting 77 – 79
**shared objects**
weak pointers, using 303 – 308
**sleep_for() function** **320**
using 320, 321
**smart pointers** **297, 348**
**sort containers**
with std::sort() algorithm 188 – 193
**span**
used, for making C-array safe 28 – 31
**split() algorithm**
building 427 – 432
**standard aliases** **349**
**static reflection** **449, 450**
**std module**
code simplification 1, 2
working 2, 3
**std::any class**
using, for type safety 272 – 275
**std::async() function** **328 – 333**
**std::atomic class**
flags and values, sharing 345 – 348, 352
floating-point types 349
integral types 349
lock-free variations 350, 351
pointers and smart pointers 348
specializations 348
standard aliases 349
**std::call_once function**
threads, initializing 352 – 354
**std::chrono library**

clock classes 287
duration class 287, 288
time events 282 – 286
**std::condition_variable**
producer-consumer problem, resolving 354 – 358
**std::formatter class** **378 – 380**
working 380, 381
**std::function**
polymorphic wrapper, creating 164 – 168
**std::optional class**
working 271
**std::shared_ptr class**
object sharing 299 – 302
working 303
**std::thread class** **321 – 328**
**std::unique_ptr**
allocated memory, managing 292 – 298
**std::variant class**
union primitive 276, 277
value types, storing 276 – 281
**string class**
customizing, with char_traits 259 – 262
**string::contains() method** **17**
working 18, 19
**string_view class** **223**
using, as lightweight string object 223 – 228
**strings**
concatenating 238 – 242
parsing, with regular expressions (regex) 263 – 265
printing, with print() and println() function 234 – 237
reading, from user input 250 – 254
transforming 244 – 248
whitespace, trimming 248, 249
**structural pattern matching** **452, 453**
**structured binding**
using, to return multiple values 31 – 35
**synchronous generators**
using 5 – 7

# T

**ToDo list**
creating, with multimap 95 – 97
**template argument deduction**
using, for simplicity and clarity 38 – 42
**text formatting**
with format library 228 – 233
**threads**
initializing, with std::call_once function 352 – 354
**time events**
with std::chrono library 282 – 286
**traits**
using, with iterators for STL compatibility 109 – 112
**trie class**
creating 415 – 423
**type safety**
std::any class, using 272 – 275

# U

**uniform erase functions**
using, to delete elements from container 50 – 54
**unordered_map**
using, with custom keys 74 – 77
**user input**
filtering, with set class 77 – 79
sorting, with set class 77 – 79

# V

**value types**
storing, with std::variant class 276 – 281
**vector**
sentence length, finding 92 – 95
**vector elements**
accessing 58 – 61
sorting 61 – 64

# W

**weak pointers**

using, with shared objects 303 – 308

**word counting**
in file 254

**worker() function** **370**

# Z

**zip iterator adaptor** **142**
building 136 – 140
working 140 – 142