INTRODUCCION

Lenguaje de bajo nivel: es, por ejemplo el Assembler, que está integrado por la representación simbólica de las instrucciones de máquina que ejecuta la computadora. Por lo tanto, al programar en Assembler se estaría trabajando directamente con ellas, aunque de manera simbólica. El lenguaje NO suministra estructuras de control.
Lenguaje de alto nivel: sus elementos **no se relacionan** de una forma muy directa con el modo en que la computadora ejecuta el programa. Soporta el concepto de tipo de datos (es decir define a un conjunto de valores que puede tener una variable y a un conjunto de operaciones que se pueden hacer sobre ellas), proporciona estructuras de control, órdenes de entrada y salida, etc. Todo esto hace que, si bien un programa en lenguaje de alto nivel es "más fácil" de escribir, es menos eficiente (deberá ser traducido a lenguaje de máquina por un compilador o intérprete).

Para satisfacer la necesidad de contar con un lenguaje que fuera de fácil utilización, pero que al mismo tiempo tuviera la misma eficacia que el Assembler, se diseña el C como un lenguaje de nivel intermedio. En definitiva va a combinar la estructura del lenguaje de alto nivel con la potencia y eficacia del lenguaje de bajo nivel.

El lenguaje no fue inventado directamente, sino que resulta de un proceso que podemos analizar brevemente en distinto momentos:
Década del 60:
Técnicos de los laboratorios de la American Telephone and Telegraph (Thompson y Ritchie) **desarrollaron en assembler**, para un equipo PDP-11 (de Digital Equipment Corporation) un sistema operativo multiusuario (esto es que permite que una computadora sea usada simultáneamente por varios usuarios, cada una a través de una terminal) que llamaron **UNIX** (la Microsoft lo desarrollaría con el nombre de XENIX).

Década del 70:
- En 1972, basándose en dos lenguajes de reciente invención (el BCPL y el B) **Dennis Ritchie** diseña, para ser soportado en UNIX, uno nuevo, que llamó **C**.
  Por su parte, el UNIX presentaba grandes inconvenientes cuando era portado de una máquina a otra (por los distintos lenguajes de máquina), entonces Thompson y Ritchie lo reescriben casi totalmente en C, dejando muy pocas rutinas en Assembler. Por lo tanto, ahora lo único que había que hacer para portar UNIX a otras máquinas era escribir un compilador de C.
- En 1978 Kernigham y Ritchie publican el libro "El lenguaje de programación C" donde se describe el lenguaje del sistema operativo UNIX. Esta versión fue tomada durante muchos años como el estándar ***(contaba con 27 palabras claves)***.

Década del 80:
Con la popularidad de los microprocesadores aparecen para ellos, distintas versiones de C que, si bien eran altamente compatibles presentaban discrepancias importantes.
- En 1983 el ANSI (American National Standards Institute o Instituto de estándares americano) se aboca a la tarea de crear un estándar que defina el lenguaje C.

Década del 90:
- En 1990 el estándar ANSI de C ***que contiene 32 palabras claves*** es definitivamente adoptado, y entonces están disponibles para el público las primeras copias.

- A partir de 1990 los distintos compiladores que surgen se adaptan al estándar ANSI. Aparece, por ejemplo el Turbo C ***que posee 43 palabras claves*** y posee un compilador rápido y eficiente.

En 1986 Bjarne Stroustrup basándose en el C desarrolla el C++. Por lo tanto es una versión ampliada del C. Justamente el nombre C++ representa esta naturaleza evolutiva con respecto al C (++ es el operador incremento en C). Además de los recursos de este lenguaje que permiten soportar programación estructurada, tiene otros más, que proporcionan, por ejemplo, mecanismos eficientes para definir nuevos tipos de datos y puede soportar programación orientada a objetos.

USOS DEL C

- Inicialmente se lo utilizó para la creación de software de sistema (sistemas operativos, compiladores, editores)
- Actualmente se lo usa para programación de uso general.

CARACTERISTICAS DEL C

* Es un lenguaje no muy altamente estructurado
  Es estructurado porque permite:
  - trabajar con funciones (subrutina independiente) definiendo en ellas las tareas del programa y codificándolas por separado (haciendo que los programas sean modulares). Una vez creada la función se la puede usar en distintas situaciones.
  - usar bloques de códigos (grupo de sentencias lógicamente relacionadas que se tratan como unidad). En C se encierran entre llaves.

  Es poco estructurado porque no permite:
  - crear funciones dentro de funciones.
- Es un lenguaje de formato libre, lo que implica que no es importante la ubicación de las sentencias sino su sintaxis.
- Permite la manipulación de bits, bytes y direcciones (o sea los elementos básicos con que funciona la computadora)
- Es un lenguaje no muy tipificado: tiene incorporados sólo 5 tipos de datos básicos pero permite cualquier conversión de tipos (por ejemplo los enteros y los caracteres pueden ser entremezclados en la mayoría de las expresiones).

- NO cuenta con instrucciones de entrada-salida, ni con métodos propios para el manejo de archivos. Estos mecanismos de alto nivel son aportados por funciones especiales.

VENTAJAS

* potente: dispone de una gran variedad de operadores y comandos.
* portátil: puede ser corrido en distintos procesadores.
* se presta para las técnicas de programación estructurada.
* alta velocidad de ejecución: por el hecho de ser compilable.
* excelente biblioteca estándar de funciones

compilación

PROG. FUENTE	**PROG. OBJETO**
en C	ejecutable

ESTRUCTURA DE UN PROGRAMA

Un programa C puede verse como un **conjunto de bloques, llamados *funciones.***

Función: conjunto de 1 o más sentencias C, designadas para realizar una tarea. La comunicación entre funciones se hace por medio de argumentos, de algún valor calculado que se envía a la función que realizó la llamada y por variables globales (externas o permanentes).
Todas las funciones tienen *nombre* y *argumentos*

identifica a la función

información con la cual la función debe hacer algo (si son más de uno van separados por coma)

main: Es fundamental, pues le indica al compilador el lugar donde tiene que comenzar la ejecución del programa (debe existir siempre)
NO TIENE ARGUMENTOS
main( )

otros nombres

.

ELEMENTOS DEL LENGUAJE
Comentarios: texto significativo que acompaña al código (programa fuente) encerrado por caracteres especiales cuya función es indicarle al compilador que debe ignorar su contenido:
/* .... */ para los comentarios de más de una línea
// ... para los comentarios de sólo una línea

Identificadores: nombre que debe comenzar con una letra y puede continuar o no con otra letra, número o el carácter subrayado (_). Hay diferencia entre una letra mayúscula y una minúscula (no es lo mismo **ab** que **Ab**).

Palabras reservadas (o palabras claves): tienen un significado estricto y predeterminado para el lenguaje (no pueden ser usadas como identificadores). Se escriben en minúscula (ELSE no es palabra clave, si lo es else).

Operadores: símbolos que permiten generar expresiones de distintos tipos. El C define muchos más operadores que otros lenguajes.

Sentencias: enunciado que especifica cierta operación ejecutable. Termina en punto y coma (;) y no reconoce al <ENTER> como un límite, pudiendo escribirse más de una por renglón.
Según la acción que representan podemos hablar de sentencias de:

- ✓ asignación
- ✓ decisión
- ✓ iteración
- ✓ transferencia de control

Directivas del preprocesador: en una primera fase del proceso de traducción (programa fuente a programa objeto) los compiladores usan un preprocesador que realiza distintas manipulaciones en el archivo fuente, antes de compilarlo realmente. Trabaja únicamente con el texto del fuente y no tiene en cuenta ningún aspecto sintáctico ni semántico del lenguaje. Las operaciones que lleva a cabo se realizan a nivel léxico y son: la sustitución o eliminación de ciertas partes del fuente o la inclusión de otros textos en un punto del mismo. El control del preprocesador está dado por unas instrucciones especiales que el programador **debe** haberle dado. Estas son las conocidas como **directivas del preprocesador.**

Cada directiva va precedida por el signo # y debe aparecer en una línea del programa, que NO finalizará en punto y coma ya que no se trata de una sentencia (la directiva no es una palabra clave del lenguaje) sino sólo una instrucción para el compilador. Las dos más usadas son:

- # include indica al compilador que tiene que **incluir** en el programa el contenido de un archivo.
  C soporta 2 formatos: **# include "nombre del archivo"** o **# include <nombre del archivo>**

  El compilador lee este archivo e inserta sus líneas en la posición de la directiva

- # define define una cadena de caracteres que puede ser reemplazada por otra a lo largo de todo el programa. Se lo puede usar para definir nombres simbólicos que representen valores constantes. Un ejemplo podría ser: # define XZ 20

  Cuando el preprocesador encuentra el identificador XZ en el programa fuente lo sustituye, automáticamente, por la cadena 20 (el preprocesador **NO** lo convierte en número, eso lo hace el compilador).
  Otros ejemplos: # define PI 3.14159 # define MAX "hola"

- Existen otras directivas que:

- Permiten compilar selectivamente parte de un código fuente de un programa (compilación condicional)
- Hacen que se detenga la compilación y emita un mensaje de error.

VARIABLES

Como en otros lenguajes de programación, las variables, en C se corresponden con una posición de memoria y poseen:

- un `nombre` que las identifica y permite así, referirse al contenido de una dirección particular de memoria.
- un `tipo` que las describe, para interpretar el valor contenido en esa dirección.

`Nombre:` se puede formar por letras mayúsculas, letras minúsculas, dígitos y el carácter subrayado (_).

- El primer carácter debe ser indefectiblemente una letra.
- En general son significativos los primeros 31 caracteres del nombre (esto difiere según el compilador)..
- No pueden usarse palabras claves

Los siguientes son algunos nombres de variables válidos, todos distintos entre sí:

NUM	N_UM3
Num	n_um3
num	Numm
N123	Numero
n123	N

`Tipo:` es la representación concreta de un concepto, en cuanto a los valores que puede asumir y al conjunto de operaciones que puede realizar en directa correspondencia con los recurso de hardware. El C soporta 4 **tipos de datos** básicos (llamados también integrados) que forman parte del lenguaje y corresponden a las unidades fundamentales más comunes de almacenamiento:

PALABRA CLAVE	TIPO	SIGNIFICADO
char	carácter	carácter
int	entero	número entero con signo
Flota	flotante de simple precisión	número con signo, con o sin parte fraccionaria
double	flotante de doble precisión	número con signo, con o sin parte fraccionaria con más dígitos significativos que el tipo anterior

El rango de valores que pueden representarse con cada tipo, depende del diseño del hardware; en cuanto a la cantidad de bits dispuestos para cada uno.

Así, generalmente se dispone de 16 bits para contener un entero, por lo que los valores posibles pertenecen al intervalo $[-2^{15}, 2^{15} - 1]$ (o sea [-32768, 32767]). Pero si se dispone de 32 bits para un entero, los valores posibles pertenecerán al intervalo $[-2^{31}, 2^{31} - 1]$ (o sea [-2.147.483.648, 2.147.483.647]).

En cambio si se dispone de 32 bits para contener un número en punto flotante de simple precisión, los valores posibles serán $[\ 3{,}4 \times 10^{-38},\ 3{,}4 \times 10^{+38}]$.

Para almacenar un carácter del juego de caracteres ASCII, generalmente se dispone de 8 bits. El C permite usar una variable de tipo **char** como un “entero pequeño”, este será el que corresponda a la sarta de 8 cifras binarias; según la arquitectura de la

máquina ese número puede ser con o sin signo, aunque esto no afecta a la impresión del carácter en sí.

El C posee además un tipo de dato de propósito especial cuyo significado es *"sin valor"* y cuya palabra clave es **void**. Desde el punto de vista sintáctico este tipo se comporta como un tipo fundamental. No obstante, se le puede utilizar sólo como parte de un tipo derivado; no existen variables de tipo **void**. Sirve para especificar que una función no devuelve un valor.

Mencionaremos 4 de los modificadores de tipos que existen en C:

short ..... reduce a la mitad la longitud de almacenamiento
long ..... duplica la longitud de almacenamiento
unsigned ..... número sin signo (siempre positivo)
signed ..... número con signo (positivo o negativo)

Así, la variedad de tipos se amplía: **short int**, **long int**, **signed char**, etc. Permite, en el caso de los enteros, una notación abreviada: **short**, **long** o **unsigned** pues el compilador sobreentiende el **int**.

El efecto que produzcan los modificadores tiene que ver con el entorno de trabajo. Por ejemplo, la mayoría de los compiladores C de Pcs usan enteros de 16 bits (ya que este es el tamaño de palabra de la familia de procesadores 8086). El estándar ANSI de C establece que el menor tamaño aceptable para un **int** es de 16 bits y que para un **short int** es 16 bits; por eso en muchos entornos no hay diferencias entre un **int** y un **short int**.

El modificador **long** se puede aplicar a **int** o a **double.**

La diferencia entre un entero con signo (**int**) y un entero sin signo (**unsigned int**) está en la forma de interpretar el bits más significativo:

* **int**: el compilador C generará un código que asume que el bit más significativo de se usa como indicador de signo (0 para el positivo, 1 para el negativo), y que para los negativos generalmente aplica el método de complemento a 2; por lo tanto son 15 los bits que representan al número.
* **unsigned int** : no existe un bits para el signo, por lo tanto son 16 los bits que representan al número.

Si una variable declarada como **int** contiene 0111111111111111 será interpretado como el +32767 (valor decimal de las 15 últimas cifras binarias con signo positivo, ya que la primer cifra es 0), pero si los bits contenidos fueran 1111111111111111 se trataría de un -1 porque, asumiendo el formato complemento a 2, los 15 últimos bits corresponden al binario 000000000000001 (el primer 1 estaría indicando que se representa a un negativo).

Si en cambio el 0111111111111111 está contenido en una variable declarada como **unsigned int** será interpretado también como el +32767 (valor decimal de las 16 cifras binarias), pero si los bits contenidos fueran 1111111111111111 se trataría del +65535 (valor decimal de las 16 cifras binarias).

El modificador **signed** aplicado a **int** sería redundante, no así en el caso de los **char** en una implementación en la que este tipo de dato no lleve signo.

Para elegir fácilmente el tipo de dato adecuado, es necesario conocer muy bien la arquitectura de la máquina. En general los conceptos fundamentales a tener presente en cuanto a tamaño son:

tamaño *char* **<** tamaño *short int* **<** tamaño *int* **<** tamaño *long int*
tamaño *float* **<** tamaño *double* **<** tamaño *long double*

Considerando el tamaño más común y mínimo al que responde una máquina, las combinaciones de los distintos tipos de datos y modificadores nos permitirían trabajar en variados rangos:

TIPO	TAMAÑO (bits)	RANGO
char	8	-128, 127
unsigned char	8	0, 255
signed char	8	-128, 127
int	16	-32768, 32767
unsigned int	16	0, 65535
short int	16	-32768, 32767
long int	32	-2.147.483.648, 2.147.483.647
float	32	$3,4 \times 10^{-38}$, $3,4 \times 10^{+38}$
double	64	$1,7 \times 10^{-307}$, $1,7 \times 10^{+308}$
long double	80	$3,4 \times 10^{-4932}$, $3,4 \times 10^{+4932}$

Ejemplo:

```
char m;
m = 139;
m = 1398;
```

DECLARACION DE VARIABLES

En C todas las variables se deben declar antes de usarlas.

Al declarar una variable, se le está especificando:

- al programa, **cómo debe interpretar los contenidos de la memoria** y por lo tanto se establece cuánta necesita.
- al compilador, el **significado** de los símbolos de operación que se le apliquen a los valores contenidos en esas direcciones de memoria.

La declaración de variables se hace por medio de una sentencia con el siguiente formato:

```
TIPO  nombre de la variable;
```

Se puede declarar más de una variable del mismo tipo separándolas por comas.

Ejemplo:

```
float x, p, luf;
```

¿DONDE SE DECLARAN LAS VARIABLES?

Dentro de una función = variables locales, privadas o automáticas.

- sólo las pueden referenciar las sentencias que están **dentro** de esa función
- sólo existen mientras se está ejecutando la función en la que están declaradas (se crea cuando se llama a esa función y se destruye cuando se sale de ella)
- su nombre puede ser usado por otras funciones

Fuera de **todas** las funciones = variables globales, externas o permanentes.

- son conocidas a lo largo de todo el programa (las puede usar cualquier trozo de código)
- mantienen su valor durante toda la ejecución
- para que una variable global que pertenece a un archivo fuente pueda ser usada por otro debe contener la cláusula *extern*.

Suponemos un programa C formado por una función main( ) y otra función de nombre F invocada desde la anterior. El programa fuente podría esquematizarlo de la siguiente forma:

```
declaración de una variable a

main()
 {
    inicialización de la variable a;
    declaración e inicialización de la variable b;
    llamada a la función F enviando a la variable b;
   }


definición de la función F
 {
   uso del valor de la variable a y del transferido desde la
   variable b;
    }
```

Si organizamos el mismo programa fuente de otra forma resultaría:

```
declaración de una variable a

main()
```

```
 {
    inicialización de la variable a;
    declaración e inicialización de la variable b;
    llamada a la función F enviando a la variable b;
   }


definición de la función F
 {
   uso del valor de la variable a
   y del transferido desde la
   variable b;
    }
```

```
declaración de una variable a

main()
 {
    inicialización de la variable a;
    declaración e inicialización de la variable b;
    llamada a la función F enviando a la variable b;
   }


          definición de la función F
           {
             extern declaración de la variable a;
             uso del valor de la variable a y
             del transferido desde la variable b;
               }
```

Las variables globales son muy útiles cuando muchas funciones del programa usan los mismos datos, aunque donde se pueda, hay que usar las locales debido, entre otras cosas, a que:

⇒ las variables globales usan memoria todo el tiempo que el programa está en ejecución, no sólo cuando se necesitan (importante si la memoria es un recurso escaso)

⇒ usar una variable global donde podría ir una local hace a la función menos general

⇒ usar muchas variables globales puede acarrear problemas de programación (especialmente en programas grandes) cuando se modifica el valor de alguna de ellas.

¿COMO SE INICIALIZAN LAS VARIABLES?

Inicializar significa especificar un valor de comienzo. Para aquellas que no se inicialicen se debe asumir que contienen valores desconocidos, “basura”, porque si bien hay algunos compiladores que inicializan automáticamente a cero, no conviene tomarlo como algo seguro.

La inicialización se realiza mediante la asignación de un valor constante (nunca otra variable) que puede realizarse en la sentencia de declaración o en una sentencia por separado

Ejemplo:

```
int cant = 97;        int cant;
                      cant = 97;
```

La ventaja de inicializar en la declaración en lugar de usar una sentencia de asignación separada, está en que el compilador producirá un código más rápido.

¿COMO PUEDEN SER LAS VARIABLES?

En C las variables pueden corresponder a datos:

- simples
- derivados : se pueden componer a partir de los datos integrados mediante operadores especiales o palabras claves, formarán: arreglo, puntero, estructura

[ ] * struct

PUNTEROS

En una computadora, la memoria se organiza en celdas numeradas o direccionadas consecutivamente. Así, cada variable tiene una dirección que indica la posición del primer byte que ella ocupa. Entonces, si por ejemplo en la dirección 200 hay guardado un valor float, la siguiente dirección es la 204 (suponiendo que un float se almacena en 4 byte).

El C le permite al programador trabajar con estas direcciones, pues se las puede almacenar en unas variables especiales que están definidas sólo para eso y que, para diferenciarlas de las otras, reciben el nombre de **puntero** o **apuntador**.

De esta forma, el puntero, por el hecho de guardar la dirección de una variable, estará haciendo siempre referencia al valor *contenido* en ella. Por eso al operar con el puntero se estará, en realidad, afectando al valor almacenado en la variable a la que él referencia

.

Si **x** es una variable que representa la edad y **p** es un puntero de **x**, esto implica que **p** hará referencia a la edad.

Esta situación esquematizada sería:

..... 2376 2377 **2378** 2379 ..... 5600 .....

x es una variable que contiene al carácter ? y está ubicada en la dirección 2378 de la memoria.
p es una variable puntero de **x** y contiene el valor 2378 (señala o apunta a la dirección 2378 que está ocupada por la variable **x**)

Declaración de un puntero

TIPO DE DATO * nombre

.
Ejemplo: int *z;
float *x;

Un puntero debe:

- ✓ ser declarado como puntero: float *x
- ✓ apuntar a alguien: x = &s

Ejemplos:

```
float s                 float s,
float *x;               float *x = &s;        equivale a la asignación s = 4.53
x = &s;        ó        *x = 4.53;
*x = 4.53;
```

```
float *x;
x = &s;        equivale a la asignación s = a
*x = a;
```

```
float *x;
x = &s;        equivale a la asignación a = s
a = *x;
```

```
char s = 'a';
char * x = &s;      equivale a la asignación a = s
char a = * x;
```

*VARIABLE PUNTERO equivale a la VARIABLE cuya dirección está almacenada en el puntero.

Un puntero al que no se le asigna un valor inicial (o sea una dirección para empezar) NO APUNTA A NADA. Si se le asignara 0 el compilador lo aceptaría sin considerarlo un error, pero los resultados serían imprevisibles ya que, por convenio, para el C una dirección 0 es inválida. En lugar de 0 se usa generalmente la constante simbólica NULL que está definida en el archivo de cabecera STDIO.H.

Los punteros se utilizan, por ejemplo, para implementar listas enlazadas y árboles binarios; aunque ejemplificaremos con programas sencillos en los que no sería estrictamente necesario usarlos.

ARRAY O ARREGLOS

Es un conjunto finito y ordenado de datos homogéneos que se agrupan bajo un mismo nombre.

Es “finito” porque se trata de una cantidad específica (puede ser grande o pequeña, pero *debe existir*).

Es “ordenado” porque están dispuestos de tal manera que existe un elemento cero, un elemento primero, uno segundo, uno tercero y así sucesivamente.

Es “homogéneo” porque todos los elementos son de igual tipo; por ejemplo puede contener valores enteros o caracteres pero *nunca ambos*.

Por lo tanto los arreglos constituyen un modo adecuado de gestionar grupos de datos relacionados

Un arreglo tiene:

⇒ un nombre que lo identifica
⇒ un rango que define su tamaño

El *límite inferior* de un arreglo es el índice más pequeño (siempre cero). El *límite superior* es el índice más grande (el rango + 1).

Una característica importante es que ni el *límite inferior* ni el *superior* (y por lo tanto tampoco el rango) **pueden cambiarse durante la ejecución de un programa**. Una técnica muy útil es la de declarar al *límite superior* como constante, tal que se minimiza el trabajo requerido para posteriores modificaciones del tamaño del arreglo.

Declaración de un arreglo unidimensional:

```
TIPO DE DATO  nombre [rango]
```

Declaración de un arreglo bidimensional:

```
TIPO DE DATO  nombre [rango de filas] [rango de columnas]
```

Declaración de un arreglo multidimensional:

Para añadir una dimensión bastará con especificar su tamaño entre corchetes. Si bien se pueden crear arreglos de más de 3 dimensiones esto se hace raramente debido a que la cantidad de memoria que consumen aumenta exponencialmente con cada dimensión adicional.

Notación de un arreglo:

Cada elemento se señala mediante un subíndice del nombre, que debe ser un valor **entero** representado por una constante, una variable o una expresión y se lo especifica entre corchetes.

Ejemplos:

a[5] ..... es el elemento del arreglo que está a cinco posiciones del comienzo.

a[3] [1] ..... es el elemento del arreglo que está a tres posiciones del comienzo en el sentido de las filas y a una del comienzo en el sentido de las columnas.

Valores que pueden almacenarse en un arreglo:

Dependen de como se lo declare pudiendo ser:

- numéricos
- de caracteres

El C NO tiene incorporado el tipo de dato cadena (string), los soporta usando arreglos unidimensionales de caracteres, con la particularidad de que el último, que indicará la terminación de la cadena, es el carácter nulo ('\0'). Por lo tanto **al declarar un arreglo de caracteres, el rango deberá exceder en 1 a la cadena más larga que pueda contener**, para dejar sitio al carácter nulo.

– Cuando **se escribe una constante de caracteres** el compilador pone el '\0' automáticamente, para indicar el final.

Ejemplo: # define A "sábados y domingos"

– Cuando **se lee una cadena**, depende de la función usada para ello (hay varias maneras de preveer el ingreso de cadenas) que deba programarse o no el almacenamiento del '\0' para dejar marcado el final.

Funcionamiento del compilador C ante un arreglo:

* no hace ninguna comprobación de los límites de un arreglo, por lo tanto aunque se lo declare con un RANGO el compilador permite el acceso a elementos "inexistentes" aunque por supuesto ocasionará unos resultados desastrosos, produciendo normalmente un fallo en el programa.
  ES ASUNTO DEL PROGRAMADOR ASEGURARSE DE QUE NO SE SOBREPASEN LOS LÍMITES DEL ARREGLO.

* no se puede asignar en bloque un arreglo completo a otro, hay que hacerlo elemento por elemento o, en el caso especial de los arreglos de caracteres, se usa una función especial contenida en el archivo de cabecera STRING.H, la:
  strcpy(cadena origen, cadena receptora)
  strcpy(cadena origen, cadena receptora, cantidad de caracteres a copiar).

* en el caso especial de los arreglos de caracteres pueden ser impresos en bloque.

  Ejemplos:

```
printf("%s\n", ca);
puts (ca);
```

Inicialización de arreglos:

Igual que ocurre con otro tipo de variables, a los elementos de un arreglo se les puede dar valor inicial.

Si fuera necesario, la inicialización podría realizarse en la declaración. En este caso se especificará la lista de valores que tendrá cada elemento del arreglo; considerando que el orden que ocupen en la lista se corresponderá directamente con el orden en el arreglo. Los valores se enumerarán separados por comas y encerrando toda la lista entre llaves:

Ejemplos:

int tra[4] = {16, 23, 26, 847};	significa que tra[0] tendrá el valor 16, tra[1] el 23 ....
int s[2] [3] = {24, 5, 8, 9, 56, 92 };	significa que s[0] [0] tendrá el valor 24, s[0] [1] el 5, s[0] [2] el 8, ..., s[1] [2] el 92.
char gra[3] = {'R', '2', 'h'};	significa que gra[0] tendrá el valor R, gra[1] el 2.....
char ca[ ] = "R2h"	significa que ca[0] tendrá el valor R, ca[1] el 2.....

Arreglos y punteros

En C existe una estrecha relación entre arreglos y punteros, lo que da gran potencia a la implementación de estos recursos.

Cuando se declara un arreglo, se está definiendo un bloque de *objetos consecutivos*, enumerados desde 0 (cero). Entonces, cuando se hace referencia a un elemento de él, se lo identifica por su posición en el arreglo, que no es lo mismo que la dirección de memoria que tenga. Justamente, en C, el nombre de un array define un puntero al 1º elemento del mismo. Por eso en algunas circunstancias se puede usar el nombre del arreglo sin índice, ya que *el compilador estaría generando un puntero al principio del arreglo*. De allí que es posible la asignación:

nombre de puntero = nombre de un array

para luego, operando con el puntero, acceder a cualquier elemento del arreglo.

Aumentar en 1 un puntero, implica, desde el punto de vista de su funcionamiento, que ahora apuntará a la siguiente variable (si se hubiera incrementado en 2, apuntaría a la subsiguiente). Aunque observando el contenido del puntero, se verá que su valor aumentará, en relación al tipo de variable apuntada. Entonces, si un puntero contiene la dirección de un elemento de un arreglo, aumentar en 1 el valor del puntero implica que apuntará al siguiente elemento del arreglo.

Así: **int z[3]** declara un arreglo de enteros
int *pun declara un puntero a entero
pun = z en **pun** se almacena la dirección del 1º elemento del arreglo, apunta a él.

De esta forma, mencionar luego a ***pun** equivale a decir el valor almacenado en **z[0]**, y mencionar a **pun** será referirse a la dirección de memoria de **z[0]**. Esto ocurre también en el caso de las cadenas de caracteres.

z

dirección

pun

Por supuesto, en el caso de una cadena de caracteres (se define por un arreglo) también *el compilador estaría generando un puntero a la cadena*.

ESTRUCTURA

Es un conjunto finito y ordenado de datos de igual o distinto tipo que se agrupan bajo un mismo nombre. Cada dato es una variable y se constituye en miembro, campo o elemento de la estructura.

Declaración de una estructura

Es una sentencia en la que se describe la forma de la estructura, es decir, en ella se crea una plantilla de la misma definiéndose cada campo o miembro.

```
struct  nombre { TIPO nombre;
                 TIPO nombre;
                 ..... ;
                } ;
```

Definir a una variable como estructura

Implica reservar espacio de almacenamiento para una estructura de determinada forma. Puede hacerse:

- En el momento de la declaración de la estructura:

```
struct  nombre { TIPO nombre;
                 TIPO nombre;
                 ..... ;
                } lista de variables ;
```

- Después de la declaración de la estructura:

```
struct  nombre { TIPO nombre;
                 TIPO nombre;
                 ..... ;
                 };
.............
struct  nombre  lista de variables ;
```

Inicialización de una estructura

Puede inicializarse cada campo, en la definición de la variable estructura, con una lista de valores constantes para ellos:

```
struct  nombre { TIPO nombre;
                 TIPO nombre;
                 ..... ;
                } variable = {valor1, valor2, .....} ;
```

Operaciones sobre una estructura

- Asignación:

var1 = var2

donde var1 es la identificación de una estructura de determinada forma y var2 es la identificación de otra estructura de la misma forma que la anterior (los mismos campos, con los mismos nombres y del mismo tipo). En definitiva las dos variables deben estar definidas por la misma estructura.

Ejemplo:

```
struct xx {
            int a;
            char z;
              };
struct zz{
           int m;
           char w;
           float d;
              };
struct xx una, dos;
struct zz tres;
.......;
.......;
una = dos;
tres = una;
```

- Acceso a cada miembro:

variable estructura.nombre del campo

Ejemplo:

```
int z;
struct xx {
            int a;
            char z;
              } uno;
uno.a = 4;
z = uno.a;
```

- Comparación:

  NO se puede realizar

var1 operador relacional var2

donde var1 y var2 son estructuras

CONVERSION DE TIPOS DE DATOS

Los operandos de tipos distintos, que aparecen en una expresión, se convierten a un mismo tipo en forma automática, de acuerdo con ciertas reglas o en forma forzada.

♦ CONVERSION AUTOMATICA DE TIPO IMPLICITA

Hay que diferenciar entre *el tipo de dato del resultado de una expresión* (regla nº 1) y *el tipo de dato con que ese resultado se almacena* en la variable (regla nº 2).

Regla nº 1: en una expresión aritmética donde se vinculan por lo menos 2 operandos, si estos son de distinto tipo, el compilador C, previo a la resolución de la operación igualará los tipos de datos convirtiéndolos automáticamente segun una escala de dominio de tipos que podríamos expresar como:

char
int
float
double

Esto implica decir que los *char* se convierten a *int* tomando el valor ASCII del carácter en cuestión; los *int* se convierten a *float* y estos a *double*.

Ejemplos:

- 10 + 3.8.....Mezcla de **int** y **float**

El resultado es **13.8**

- 'A' + 5.....Mezcla de **char** e **int**.

El resultado es **70**

- 'A' + 5.0.....Mezcla de **char** y **float**

El resultado es **70.0**

- 'A' + 5.9.....Mezcla de **char** y **float**

El resultado es **70.9**

- 'A' + 3.8 + 5.....Mezcla de **char**, **int** y **float**

El resultado es **73.8**

Regla nº 2: en una asignación, independientemente de los tipos de datos que intervienen se almacenará el resultado de la expresión convirtiéndose automáticamente al tipo de la variable de asignación.

Expresando las asignaciones por sus tipos, podríamos resumir:

int = char.....se almacena el código ASCII del caracter
int = float.....se almacena como entero truncando los decimales
float = double....se almacena en simple precisión redondeando los decimales
float = int.....se almacena el valor entero con parte decimal 0
float = char.....se almacena el código ASCII del caracter con su parte decimal 0
char = float.....se almacena el caracter correspondiente truncando los decimales

Ejemplos:

- a = 10 + 3.8

1º El resultado es **13.8**
2º si **a** es **int** se almacena **13**
si **a** es **float** se almacena **13.8**
si **a** es **char** se almacena el carácter **"nueva linea"**

- a = 'A' + 5.0

1º El resultado es **70.0**
2º si **a** es **int** se almacena el **70**
si **a** es **float** se almacena el **70.000000**
si **a** es **char** se almacena el carácter **F**

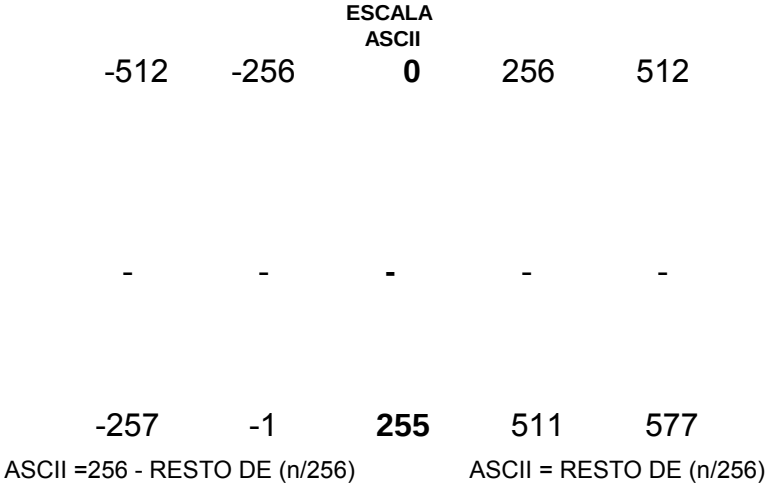
- a = 'A' + 5.9

1º El resultado es **70.9**
2º si **a** es **int** se almacena el **70**
si **a** es **float** se almacena el **70.9**
si **a** es **char** se almacena el carácter **F**

- 'A' + 3.8 + 5

1º El resultado es **73.8**
2º si **a** es **int** se almacena el **73**
si **a** es **float** se almacena el **73.8**
si **a** es **char** se almacena el carácter **Y**

Cuando una expresión aritmética que se asigna a una variable char, da un resultado mayor que 255 o menor que 0, el carácter que corresponde depende de la relación existente entre éste y la escala ASCII. Por eso si representamos con una recta de números la sucesión de éstos códigos y comparamos con ella a cualquier número fuera de la escala, tendremos para un valor **n**:



Ejemplos:

- a = 500 + b

si en **b** hay un **77** y **a** es char, se guardará en ella la letra **A**, pues el resto de la división entera entre 577 y 256 es 65 (RESTO DE (n/256))

- a = b

si en **b** hay un **-1471** y **a** es char, se guardará en ella la letra **A**, pues el resto de la división entera entre 1471 y 256 es 191 y la diferencia entre 256 y éste da 65 (256 - RESTO DE (n/256))

♦ CONVERSION FORZADA DE TIPO EXPLICITA O MOLDES (CAST)

Si algún operando de una expresión debe ser usado momentáneamente con otro tipo, más allá del que adquirió automáticamente de acuerdo con las reglas anteriores, es posible provocar esa conversión. El operando debe ser afectado por el nuevo tipo de dato de la siguiente forma:

(nuevo tipo de dato) OPERANDO

Ejemplo:

```
int r;
float x,z;
x = 98.2;
r = (int) x;
z = x;
```

Aquí el molde hace que el valor de **x** se convierta en un entero.

CONSTANTES

Son valores fijos que el programa NO puede cambiar. Puede tratarse de valores numéricos o no numéricos

- Numéricos: el compilador decide qué tipo de constante es. Por omisión elige el tipo de dato compatible más pequeño que pueda contener a una constante numérica, salvo en el caso de las constantes de punto flotante que las supone de tipo double. Así, por defecto, 20 será int (aunque hubiera podido caber en un char), el 63000 es un unsigned int.
  De cualquier forma, si la complejidad del programa lo requiriera, el C le permite al programador especificar el tipo de la constante usando un sufijo:
  F o **f**..... float
  L o **l** ..... long
  U o **u**..... unsigned int
  Ejemplos: 1234L ó 1234l, 1234UL

  Los números con parte fraccionaria pueden, si fuera necesario, expresarse en notación científica, siguiendo el formato: número E signo exponente . Por Ejemplo: 34E-7.

  El C permite usar constantes pertenecientes a distintos sistemas de numeración
  ⇒ decimal: corresponde a la representación en base 10
  ⇒ octal: corresponde a la representación en base 8 y se indica usando un prefijo 0 (cero). Por ejemplo: 025 que equivale al $21)_{10}$.
  ⇒ hexadecimal: corresponde a la representación en base 16 y se indica usando el prefijo 0x ó 0X. Por ejemplo: 0x25 que equivale al $37)_{10}$.

- no numéricos
  - de carácter: es **un carácter delimitado por apóstrofos**. El valor de una constante de carácter es el número correspondiente al código del procesador (generalmente el ASCII). Hay algunos caracteres no imprimibles que se representan como constantes de carácter mediante una secuencia de escape que aparece como 2 caracteres, pero en realidad son sólo uno (‘\n’, ‘\0’, etc.)
  - cadena de caracteres: es una secuencia de **ningún, uno o más caracteres delimitado por comillas**. El compilador ubica un carácter nulo (‘\0’) al final de cada cadena para que los programas puedan encontrar su final. Técnicamente una cadena de caracteres es un vector cuyos elementos son caracteres.
    NOTA: No es lo mismo ‘x’ que “x”; en el primer caso se trata de un único carácter, x, cuyo valor es el que le corresponde en el conjunto de caracteres del procesador y en el segundo caso se trata de una cadena que tiene 2 caracteres, una x y un \0.

CONSTANTES SIMBÓLICAS

Puede dársele a una constante, un nombre simbólico. Así, el compilador reemplazará todas sus apariciones, por el valor correspondiente. Para ello debe definirse la constante, al comienzo del programa, mediante la construcción:

```
# define  NOMBRE    VALOR
```

donde NOMBRE es una cadena de caracteres, generalmente en mayúscula (para poder distinguir de los nombres de variable) que simbolizará al VALOR a lo largo del programa. De esta forma haciendo una modificación sólo en esta sentencia se puede alterar la definición de la constante.

Ejemplo:

```
# define  A  8
# define  COL  9
```

OPERADORES

Combinados con distintos operandos pueden generar expresiones que siguen las reglas del álgebra, pudiendo los paréntesis alterar el orden de evaluación que le correspondería por la prioridad propia de cada operador.

♦ aritméticos

Suma ..... +
Resta ..... -
Multiplicación ... *
División ..... /
Módulo ..... % (resto de la división entera)

Con estos se pueden generar expresiones aritméticas cuyos operandos pueden ser: constantes, variables u otro tipo de expresiones.

– Cuando en una expresión se combinan distintos operadores, la prioridad en la evaluación es:
  1º * / %
  2º + -
– Cuando en una expresión se combinan operadores de igual prioridad la evaluación es de izquierda a derecha.

Ejemplos:

4 / (k + 2)	a - 5 * g
7 + r % paso + y	tr + y / (z - 9)
(a > b) * 3	78.9 + r + 9

♦ relacionales

Mayor..... >
Menor ..... <
Mayor o igual ... >=
Menor o igual ... <=
Igual..... ==
Distinto..... !=

Con estos se pueden generar expresiones relacionales cuyos operandos pueden ser: constantes, variables u otro tipo de expresiones.

EL RESULTADO DE UNA EXPRESION RELACIONAL ES *UN NUMERO ENTERO*: 1 para verdadero y 0 para falso

– Cuando en una expresión se combinan distintos operadores, la prioridad en la evaluación es:
  1º > < >= <=
  2º == !=
– Cuando en una expresión se combinan operadores de igual prioridad la evaluación es de izquierda a derecha.

– Cuando en una expresión se combinan operadores aritméticos y relacionales, la prioridad en la evaluación es:

1º aritméticos
2º relacionales

Ejemplos:

re == 6	3 > t == 1
a > b < c	a > b * 3
a == b > c	m % 3 == 0

♦ lógicos
Conjunción (y) ..... &&
Opción (o) ||
Negación (no) ..... !

Con estos se pueden generar expresiones lógicas cuyos operandos pueden ser:
relaciones, variables u otro tipo de expresiones.

- Cuando en una expresión se combinan distintos operadores, la prioridad en la evaluación es:
  1º !
  2º &&
  3º ||

Ejemplo: Verdadero o Falso y Verdadero da Verdadero

- Cuando en una expresión se combinan operadores de igual prioridad la evaluación es de izquierda a derecha (Verdadero y Falso y Falso da Falso).
- Cuando en una expresión se combinan operadores relacionales y lógicos la prioridad en la evaluación es:
  1º relacionales
  2º lógicos
- Cuando en una expresión se combinan operadores aritméticos, relacionales y lógicos la prioridad en la evaluación es:
  1º aritméticos
  2º relacionales
  3º lógicos

Ejemplos:

```
a > 0  &&  f <= 4
z  && dw == 0
!a
m > 2  &&  m < 10  ||  m == 1
s < 7  ||  s == 32 != 0
w < -3  ||  w >12
! ( w >= -3 &&  w <= 12)
```

♦ para manejo de bits
Sólo pueden ser aplicados a operandos enteros (**int**, **char**, **short** y **long**)

- de desplazamiento de bits
  Sirve para resolver algunas operaciones de multiplicación o división cuando el multiplicando o el divisor se puede expresar como potencia de 2 (2,4,...,64,1/8, 1/16,...). Como, debido al funcionamiento interno de las C.P.U. las operaciones de corrimiento de bits son más rápidas que las aritméticas equivalente, es una gran ventaja que el lenguaje tenga este operador.
  El formato general es :

VARIABLE operador EXPRESION ENTERA

  Provoca que todos los bits que representan al valor VARIABLE se desplacen tantas posiciones como se indica en EXPRESION ENTERA, por lo tanto por un extremo se perderán bits y por el

otro se completará con ceros. Al producirse el corrimiento cambia el valor representado en potencias de 2.

Puede darse:
desplazamiento a la izquierda ..... <<
desplazamiento a la derecha ..... >>

<< cada posición corrida equivale a multiplicar el número binario por $10)_2$ o sea $2)_{10}$; así si se corren 3 lugares equivale a multiplicar por $1000)_2$ o sea $8)_{10.}$ Por ejemplo, un entero almacenado en 2 byte será:

0 0 0 0 0 0 0 0 1 1 0 1 0 0 1 1 = $211)_{10}$

al desplazarlo un bit a la *izquierda* será:

0 0 0 0 0 0 0 0 1 1 0 1 0 0 1 1

0 0 0 0 0 0 0 1 1 0 1 0 0 1 1 0 = $422)_{10}$

En definitiva, un desplazamiento de n bits es igual que multiplicar por $2^{n.}$

>> cada posición corrida equivale a multiplicar el número binario por $10)_2^{-1}$ o sea $1/2)_{10.}$ Así si se corren 3 lugares equivale a multiplicar por $10)_2^{-3}$ o se $1/8)_{10.}$ Por ejemplo, un entero almacenado en 2 byte será:

0 0 0 0 0 0 0 0 1 1 0 1 0 0 1 1 = $211)_{10}$

al desplazarlo un bit a la *derecha* será:

0 0 0 0 0 0 0 0 1 1 0 1 0 0 1 1

0 0 0 0 0 0 0 0 0 1 1 0 1 0 0 1 = $105)_{10}$

En definitiva, un desplazamiento de n bits es igual que dividir por 2^{-n}

Ejemplos:

a >> 3 equivale a la expresión a * 8
a << 3 equivale a la expresión a / 8

- and ..... &
- or ..... |
- xor ..... ^
- complemento a uno..... _

♦ de punteros

Son operadores unarios cuyo formato general es:

Operador OPERANDO

- la dirección del operando ..... &
- el valor de la variable cuya dirección está en el operando ..... *

Como el puntero “apunta” a una variable, esto implica que podrá aparecer en todos los lugares donde puede hacerlo la variable; de allí que el operador * puede usarse al lado izquierdo de una asignación. En este caso indica almacenar un nuevo valor a una variable, utilizando un puntero a ella.

Aritmética de punteros:
Los punteros se pueden usar como otras variables, pero con algunas restricciones:

- las únicas operaciones aritméticas que aceptan son la suma y la resta a través de los operadores +, -, ++ y --.
- se les puede sumar o restar *sólo cantidades enteras.*
- cuando se incrementa o decrementa un puntero, apuntará al siguiente o al anterior atendiendo a su tipo.
- aceptan operadores relacionales

El comportamiento de los operadores en distintas situaciones sería:
(*p)++.....se incrementa el valor al que apunta **p**

*p ++ .....se incrementa el valor del puntero **p** en la cantidad de bytes del tipo de variable a la que apunta y luego obtiene el valor de la nueva posición.

++ p .....se incrementa el valor de **p** en la cantidad de bytes del tipo de variable a la que apunta

♦ coma
Su función es la de indicarle al compilador la sucesión de cosas que debe hacer; es decir la coma se usa para encadenar varias operaciones.

♦ incremento - decremento
Aumenta en 1 el valor del operando..... ++
Disminuye en 1 el valor del operando ..... --

La expresión que resulta de aplicar el operador incremento o decremento es:

operador VARIABLE

VARIABLE operador

Donde operador actúa distinto según su ubicación respecto a la variable:

- prefijo de la variable (operador VARIABLE)
  1º se modifica en 1 el valor de la VARIABLE
  2º la VARIABLE usa su nuevo valor

- sufijo de la variable (VARIABLE operador)
  1º la VARIABLE usa su valor
  2º se modifica en 1 el valor de la VARIABLE

Ejemplos:

++ n equivale a n = n + 1

z = b++ equivale a z = b y luego b = b + 1
x = --r equivale a r = r -1 y luego x = r

♦ asignación ..... =
Provoca el almacenamiento de un valor en una variable. El formato general resulta:

VARIABLE operador VALOR

donde VALOR puede ser una constante, otra variable o una expresión.
Como en otros lenguajes la operación de asignación es una sentencia, pero a diferencia de muchos de ellos se trata, en C, de un operador muy potente.

Variaciones en la sentencia de asignación:

- puede asignársele el mismo valor a distintas variables con la forma:

VARIABLE1 = VARIABLE2 = VARIABLE3 = VALOR

Ejemplo:
a = suma = 3; equivale a a = 3; suma = 3 ;

- permite usar abreviaturas, cuando el valor sea una expresión aritmética que involucre a la misma variable, con la forma:

VARIABLE operador = VALOR

en este caso puede tratarse de +, -, *, /, %
Ejemplos:

a += 3; equivale a a = a + 3;
dato *= 4.2; equivale a dato = dato * 4.2;
f *= x + 3; equivale a f = f * (x + 3);

♦ condicional
Según una condición asigna uno u otro valor ..... ? :

La expresión que resulta de aplicar este operador es:

CONDICION ? VALOR1 : VALOR2

El resultado de esta expresión es VALOR1, si CONDICION es verdadera y VALOR2, si CONDICION es falsa.
Este operador le permite al compilador producir un código más eficiente que si se usa la sentencia equivalente **if....else**
Ejemplos:

(z > b) ? 23 : a+2 equivale a: si el valor almacenado en **z** es mayor que el que está en **b,** el resultado es 23, de lo contrario es 2 más que lo que contiene **a**.

!m ? ‘V’ : ‘F’	equivale a: si el valor guardado en **m** es 1, el resultado es el carácter F, de lo contrario es el carácter V.

ENTRADA/SALIDA

El C, no proporciona ningún mecanismo de comunicación con el usuario. La entrada y la salida se lleva a cabo a través de funciones definidas en las librerías. Son muchas y cada una tiene un funcionamiento específico. Las que manejan la entrada/salida por consola dirigen sus operaciones al teclado o entrada estándar y a la pantalla o salida estándar.

FUNCIONES DEFINIDAS EN EL ARCHIVO DE CABECERA STDIO.H
De salida

PRINTF()

Esta función permite *mostrar* en la pantalla, con un determinado formato (controlado por el programador). La invocación a ella es:

```
printf (cadena de control, arg1, arg2, arg3,....)
```

donde:
cadena de control: es un conjunto de caracteres **entrecomillados** formado por:

- texto que desea imprimirse
- especificaciones u órdenes de formato que define **la forma con que se exhibirán** cada uno de los argumentos.
- secuencia de escape o de salida.

arg1, arg2, arg3: representan a constantes, expresiones o variables cuyos valores se desea imprimir.

Especificación de formato: habrá una especificación para cada argumento, si no hay suficiente o no son del tipo correcto se producen resultados confusos. La sintaxis es la siguiente:

- comienza con el carácter **%**
- termina con un **carácter de conversión**: es una letra que indica con que forma se imprimirán los argumentos de la lista; estos son:
  – para números:

d	entero del sistema decimal
o	entero del sistema octal
x	entero del sistema hexadecimal en minúscula
X.....	entero del sistema hexadecimal en mayúscula
u	entero del sistema decimal sin signo
f	número con coma decimal (flotante) y 6 dígitos de precisión (simple), ó 14 dígitos de precisión (doble)
e	número con coma decimal (flotante) en notación científica
p	puntero (lo muestra en hexadecimal)

  – para caracteres:

c	para un solo carácter
s	para una cadena de caracteres (se imprimirá hasta alcanzar el carácter nulo.

Si el carácter que sigue al % NO es uno de estos se imprimirá el % como tal.

- entre el signo % y el carácter de conversión puede haber:
  - **dígito**: su valor indica el tamaño MINIMO del campo que se usará para imprimir el argumento correspondiente.
    - Si la cantidad de caracteres del argumento (sea número o cadena) es MENOR que el campo asignado se rellena con blancos. En el caso de los números se puede rellenar con 0 (ceros) poniendo un 0 delante del dígito.
    - Si la cantidad de caracteres del argumento es MAYOR que el campo se lo imprime completo y el campo indicado NO es tenido en cuenta.

  - **dígito.digito**: el valor del primer dígito indica el tamaño MINIMO del campo que se usará para imprimir el argumento correspondiente, el valor del segundo digito significa:
    - Si se trata de una cadena de caracteres: cantidad MAXIMA de caracteres que serán impresos
    - Si se trata de números flotante: cantidad de decimales
    - Si se trata de números enteros: lo ignora, pues no debiera existir.

  - **–digito** o **–digito.digito**: el valor impreso se justifica a la izquierda en el campo da salida, por omisión del signo –, la salida se justifica a la derecha.

Secuencia de escape (constante de barra invertida): se usan para mostrar caracteres que no se pueden introducir por teclado.
La sintaxis es la siguiente:

\carácter especial

donde el carácter especial puede ser, entre otros:

\r	retorno de carro (al principio de la línea actual) (código ASCII 13)
\n	en la línea siguiente (el compilador lo traduce como una combinación de retorno de carro y salto de línea) (código ASCII 10)
\t	en el siguiente tabulado (código ASCII 9)
\b	en el último carácter impreso (retroceso) (código ASCII 8)
\f	en la página siguiente (código ASCII 12)
\0	imprime el carácter nulo (código ASCII 0)
\"	imprime comillas
\\	imprime la diagonal invertida

Importancia de la especificación de formato adecuada:
printf(“%d”,38); producirá como salida 38

printf(“%d”,65400); producirá como salida –136

Una especificación **d** implica formato de número entero. Por lo tanto, su representación interna será en 2 byte, con el primer bit reservado para el signo y expresado en complemento a 2 para los negativos. Así, el 65400 no “entraría” en los 2 byte. La cadena de 0 y 1 que se corresponde con este número es:

1 1 1 1 1 1 1 1 0 1 1 1 1 0 0 0

1º BYTE 2º BYTE

1º bit = 1 => negativo
15 bit restantes = 111111101111000

complemento a 2

$000000010001000 = 136)_{10}$

PUTCHAR()

Muestra un carácter en la posición actual del cursor. Es equivalente a printf() con una especificación de formato para carácter (%c). La invocación a ella es:

```
putchar (valor)
```

donde:
valor: es una constante, una variable o una expresión.

Ejemplos:

putchar('Z')	Z	
putchar(65)	A	
putchar(a)	A	si **a** fuese una variable entera que contiene al 65
putchar(a)	A	si **a** fuese una variable carácter que contiene a la A
putchar(a+7)	C	si **a** fuese una variable entera que contiene al 60
putchar('\n')		salta un renglón
putchar("ZAEDS")		

PUTS()

Muestra una cadena seguida de un salto de línea. Es equivalente a printf() que contenga una especificación de formato para cadena (%s) seguida de una secuencia de escape para retorno de carro. La invocación a ella es:

```
puts (valor)
```

donde:
valor: es una constante o una variable.

Ejemplo:

puts("AS$&Z")	AS$&Z	
puts(a)	$&Z	si **a** fuese un array de caracteres que contiene $&Z

Vimos que las funciones putchar() y puts() permiten programar salidas iguales a las que pueden prepararse con printf() usando la especificación de formato adecuada en cada caso. La conveniencia o no de usar una u otra función tiene que ver con que, una llamada a printf() se ejecuta a menor velocidad que una invocación a cualquiera de las otras mencionadas. Esta ventaja es observable en aplicación de cierta complejidad y no en programas sencillos.

De entrada

SCANF()

Esta función permite leer un valor desde el teclado, interpretándolo con un determinado formato (el programador "controla" cómo se almacena el valor ingresado por teclado). La invocación a esta función es:

```
scanf (cadena de control, arg1, arg2, arg3,....)
```

donde:

cadena de control: es un conjunto de caracteres **entrecomillados** formado por:

- especificaciones de formato para cada uno de los argumentos de la lista que aparecen a continuación de la cadena de control (arg1, arg2, etc.): indica la **forma como debe interpretarse** el campo de entrada.
- cualquier otro carácter .

arg1, arg2, arg3: representan las direcciones de las variables donde almacenará cada valor tecleado.

Si no hay coincidencia entre la especificación de formato y el tipo de dato ingresado, scanf() NO señala error en el proceso, pero sí se obtendrán resultados inesperados al acceder a la variable en cuestión.

Cuando en una llamada a scanf() se quieren leer varios valores, cualquier carácter (incluido el espacio en blanco) que exista entre las especificaciones de formato le indican al compilador que debe descartarlo para el almacenamiento aunque lo acepta como entrada.

Ejemplo:

```
scanf("%d,%d,%c",&a, &b, &c)

scanf("%d%d%c",&a, &b, &c)
```

scanf(“%d %c”,&a, &b)

scanf(“%d %c”,&a, &b)

Esta función da por terminada la entrada, cuando no tiene más formatos en su cadena de control o cuando alguna entrada no concuerda con la especificación correspondiente.

Ejemplo:

scanf(“%d %c”,&a, &b)

Si la entrada es una cadena, almacenará en el correspondiente arreglo el conjunto de caracteres anteriores a un espacio en blanco, sin asignar la marca de fin de cadena (‘\0’).

Ejemplo:

scanf(“%s”,a,)

GETCHAR()

Esta función devuelve el carácter que se introduce por la terminal estándar (generalmente el teclado); por lo tanto se la usará para **”leer caracteres”**. La invocación a ella es:

```
getchar ()
```

En la mayoría de los compiladores, esta función está implementada de manera que, el valor de la tecla pulsada (entrada) hace eco en la pantalla y se mantiene en un búfer de línea. Esto implica que getchar() devuelve el carácter que se ha tecleado, al programa que la llamó, recién cuando se pulse <ENTER>; por otro lado no controla la cantidad de caracteres tipeados. Así, NO permite generar un entorno interactivo con el usuario.

Ejemplo:

```
main()
{
 char x,z;

 z= getchar();
 x= getchar();
 printf("%d %c\n",z,z);
 printf("%d %c",x,x);      }
```

Cuando la entrada es el carácter de fin de archivo (^ Z), si bien su ASCII es 26, getchar() devuelve el valor **-1**

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

```
main()
{
while (getchar() != -1) ;
      }
```

Podría definirse una *constante simbólica* para el valor de fin

```
# include <stdio.h>
# define T  -1
main()
{
while (getchar() != T)  ;
       }
```

El carácter devuelto por getchar() puede ser:

- recibido en una variable:

```
# include <stdio.h>
main()
{
int c;

c=getchar();
while (c != -1)
         c = getchar();
       }
```

```
# include <stdio.h>
main()
{
char c;

c=getchar();
while (c != -1)
         c = getchar();
       }
```

El usuario no observará diferencias

- impreso con formato:

```
# include <stdio.h>
# define FIN  -1
main()
{
int c;

c = getchar();
while (c != FIN)
         {
         printf("%d\n",c);
         c=getchar();
         }
      }
```

```
#define FIN  -1
main()
{
char c;

c = getchar();
while (c != FIN)
      {
         printf("%c",c);
         c=getchar();
          }
        }
```

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

GETS()

Esta función devuelve él o los caracteres que se tipeen, hasta que se pulse <ENTER>; por lo tanto se la usará para **"leer cadena de caracteres"**. El <ENTER> no forma parte de la cadena devuelta, en su lugar coloca un '\0' como marca de fin de cadena. La invocación a esta función es:

```
gets (arg)
```

donde:
arg: representa la dirección del lugar donde se almacenará el valor tecleado. Se trata entonces del nombre de un arreglo de caracteres, pues es la única forma de poder guardar una cadena.

La función gets() NO realiza comprobación de límite, es decir si la cadena ingresada tiene mayor tamaño que el previsto para el arreglo, no señalará error aunque sí pueden producirse resultados inesperados al momento de usar la cadena guardada.

FUNCIONES DEFINIDAS EN EL ARCHIVO DE CABECERA CONIO.H

De salida

GOTOXY()

Esta función

CLRSCR()

Esta función limpia la pantalla activa.

De entrada

Las siguientes funciones son usadas como getchar() para **"leer caracteres"**.pero permiten generar un entorno interactivo con el usuario:

GETCHE()

Devuelve, inmediatamente después de pulsada una tecla, el carácter correspondiente, pues no trabaja con búfer de línea de entrada (no necesita del <ENTER> para concretar el ingreso) pero hace eco en la pantalla. Así. La invocación a ella es:

```
getche ()
```

GETCH()

Devuelve, inmediatamente después de pulsada una tecla, el carácter correspondiente pues no trabaja con búfer de línea de entrada (no necesita del <ENTER> para concretar el ingreso) y no hace eco en la pantalla. La invocación a ella es:

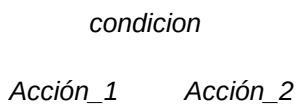
```
getch()
```

CONTROL DE FLUJO

Cuando se habla del control de flujo se hace referencia a sentencias cuya tarea es decidir el orden en el que continuará la ejecución.

DECISION:

- alternativa simple



en C:

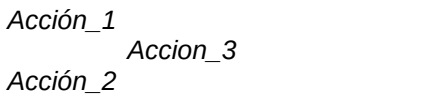
- *condición:* puede ser cualquier expresión válida para el lenguaje (aritmética, condicional, etc.) ya que: **VERDADERO ES TODO VALOR DISTINTO DE 0 Y FALSO ES EL 0**.
- *Acción:* puede ser sentencia simple o compuesta, en este último caso van encerradas entre { }

```
if (expresión)
        Sentencia_1;
    else Sentencia_2;
```

```
if (expresión)
        {Sentencia_1;
         Sentencia_2;
         Sentencia_3}
    else Sentencia_4;
```

```
if /expresión) Sentencia_l;
```

Anidamiento: un compilador estándar permite hasta 15 niveles de anidamiento, aunque puede resultar problemática su construcción y muy confusa su lectura ya que en C el **else** siempre se refiere al **if** precedente salvo que, mediante llaves, se indique lo contrario:




```
if (expresión_1)
      if (expresión_2)
         {Sentencia_1;
              Sentencia_2}
       else
              Sentencia_3;
```

```
if (expresión_1)
      {
         if (expresión_2)
            { Sentencia_1;
              Sentencia_2}
          }
         else
              Sentencia_3;
```

- alternativa múltiple

Los distintos caminos a tomar pueden depender de que:

❖ se verifique o no una igualdad: en este caso se prueba una condición con una lista de constantes y, de haber coincidencia con alguna, transfiere el control a ese punto y desde allí continúa

Acción_1 *Acción_2*

Acción_3 *Acción_4*

en C:

- *condición:* debe ser un entero representado por una variable, una constante, el resultado de una función o de una expresión.
- *Valor:* debe ser una constante. No puede haber 2 iguales.
- *otro valor:* es una alternativa opcional que corresponde al "ninguno de los valores anteriores".

```
switch (expresión)
    {
      case Valor_1:  Sentencia_1;
                     Sentencia_2;
                     Sentencia_3;
                break;
      case Valor_2: Sentencia_4;
                break;
      case Valor_3: case Valor_4:
                     Sentencia_5;
                break;
      default   Sentencia_5;
                }
```

Las sentencias en cada **case** NO van entre { } porque la acción a ejecutar no es una sentencia compuesta sino una secuencia de sentencias.
break produce una salida inmediata, NO puede faltar porque sino se ejecutaría la primer sentencia correspondiente al **case** siguiente.

❖ se verifique o no distintas condiciones: en este caso las condiciones son evaluadas en orden, si una resultara VERDADERA transfiere el control a la acción asociada a ella y termina, de lo contraria continúa probando las otras. Si ninguna de las condiciones es VERDADERA se ejecuta la acción correspondiente al último falso.

Acción_1

Acción_2

Acción_3

Acción_5

Acción_4

```
if (expresión_1)
    Sentencia_1;
        else if (expresión_2)
            Sentencia_2;
                else if (expresión_3)
                    {
                        Sentencia_3;
                        Sentencia_4;
                    }
            else
                Sentencia_5;
```

ITERACION

- con condición de entrada

mientras *condición* **hacer**

Acción

en C:

- *condición* es cualquier expresión válida para el lenguaje (aritmética, condicional, etc.)
- *Acción* puede ser sentencia simple o compuesta, en este último caso van encerradas entre { }

```
while (expresión)
    Sentencia;
```

```
while (expresión)
        {Sentencia_1;
         Sentencia_2;
         Sentencia_3;}
```

Se ejecuta el cuerpo del bucle mientras *expresión* sea VERDADERA (distinto de cero). Puede que no se ejecute nunca.

para *cantidad fija de veces*

Acción

en C:

for **(***expresión_1***;** *expresión_2***;***expresión_3***)** *Sentencia***;**	***for*** **(***expresión_1***;** *expresión_2***;***expresión_3***)** **{** *Sentencia_1***;** *Sentencia_2****;}***

donde:

- *expresión_1* es para inicializar la variable de control del bucle
- *expresión_2* es la condición de repetición. Se evalúa antes de cada posible ejecución del ciclo (inclusive la primera). Este terminará cuando la condición sea falsa (igual a cero); puede que no se realice nunca, si la condición es falsa de entrada. La condición evaluada puede no tener nada que ver con la variable de control.
- *expresión_3* es para definir como se va modificando la variable de control cada vez que se repite el ciclo.

Ejemplo:

```
for (k = 5; k < 5; k++)
        printf("CUERPO");


for (k = getche(); k != '*'; k = getche());


n = 1;
for (k = getche(); k != '*'; k = getche())
        n++;
printf ("EL * SALIO EN EL LUGAR %d", n);
```

En la línea de la sentencia **for** puede faltar cualquiera de las expresiones y estar en otro lugar cumpliendo su función, pero los *punto y coma* (;) deben estar. Si falta la *expresión_2* considera que la condición es siempre verdadera.

Ejemplo:

```
k = 0;
for (; k<5; k++)
        {…… }
for (; ;)
       {……
        …..}
```

La variable que controla el ciclo puede modificarse dentro del cuerpo de la repetición.

En las expresiones puede usarse el operador coma (,) haciendo que las expresiones separadas por ellas se evalúen de izquierda a derecha.

Ejemplo:

```
for (l = 0, k = 4; i < 10; k+= 2, l++)
        printf ("%d ", k);
```

- con condición de salida

repetir
Acción
hasta que *condición*

repetir
Acción
mientras que *condición*

en C:
Se ejecuta el cuerpo del bucle mientras la *condición* sea VERDADERA (distinto de cero). Por lo menos una vez se realizará ya que la condición de iteración se evalúa luego de entrar en el ciclo.

- *condición* es cualquier expresión válida para el lenguaje (aritmética, condicional, etc.)
- *Acción* puede ser sentencia simple o compuesta, en este último caso van encerradas entre { }

```
do
    Sentencia;
while (expresión)
```

```
do
    {
  Sentencia_1;
  Sentencia_2;
  Sentencia_3;
    }
while (expresión);
```

TRANSFERENCIA DE CONTROL

break: Interrumpe el flujo de control ordinario de un bucle (*for, while, do while*) o una alternativa múltiple (*switch*).

return: Detiene la ejecución de la función actual y devuelve el control a la función llamante.

return *espresion*

FUNCION

Para aumentar la legibilidad y facilitar el mantenimiento de programas complejos, como así también para permitir la reutilización de ciertos tramos de código, es necesario aumentar el nivel de estructuración al diseñar el algoritmo. Estas soluciones pueden implementarse fácilmente en C. porque al igual que otros lenguaje cuenta con una herramienta apropiada para ello: *la función*. Algunas funciones las proporcionan los fabricantes de compiladores, pero también pueden ser construidas por el programador.

Una función no es más que un pequeño programa identificable, que toma uno o más valores llamados *argumentos* y produce un valor llamado *resultado*, y cuya ejecución se puede invocar desde otra función, tantas veces como sea necesario.

Al escribir una función hay que tener en cuenta sus 4 componentes:

1) nombre
2) *tipo de valor* que **devuelve** al finalizar su ejecución
3) información que **recibe** antes de comenzar a ejecutarse (*argumentos*).
4) *Cuerpo o conjunto de declaraciones y sentencias* que resuelven una tarea.

Todas las funciones deben ser:

- Definidas o implementadas (sólo una vez): esto es, debe especificarse la forma de llevar a cabo la operación que resuelve. El cuerpo de la función va encerrado entre llaves ({ }) y tiene un encabezado con el siguiente formato:

tipo de retorno NOMBRE DE LA FUNCION (tipo nombre del parámetro formal)

Parámetro formal es la identificación que tendrá, en el cuerpo de la función, el **valor** que necesita recibir para poder ejecutarse. Se corresponde con un lugar de memoria

Todas las partes mencionadas en este enunciado con el cual se inicia la implementación de una función, son OBLIGATORIOS.

Ejemplos:

```
int potencia (int n, int k)
        {.....
          .....
          return
valor;}
```

```
int funcx (int n)
{
  if (n<1)
        return (n*n);
   else
   return (1);
    }
```

Cuando se requiera que la función no devuelva valor alguno el tipo de retorno especificado será *void.*

- Llamadas: implica ser invocada su ejecución desde alguna sentencia de otra función. Se especificarán aquí los verdaderos valores (*parámetro actual*) o sea los argumentos con los que se resolverá la función.
  Los parámetros actuales pueden presentarse bajo la forma de una:
  - constante
  - expresión
  - variable

  Ejemplos:

```
…….
n = calculo1 (-735.8, angulo, ka );
.....…
.....
if (calculo2( marc ))
        a = z * 2;
      else
        argv = marc;
.....…
.....
calculo3 ( min, b + c*2, "---RESTAR----")
.....…
.....
```

- Declaradas: para indicarle al compilador el tipo de valor de retorno y la cantidad y tipo de argumentos que necesita la función. De esta forma el compilador informará sobre cualquier diferencia entre el **tipo** del argumento y el parámetro correspondiente y en cuanto a la **cantidad** de argumentos y parámetros. La manera de declarar una función es mediante su PROTOTIPO; que no es más que un enunciado con el formato:

tipo de retorno NOMBRE DE LA FUNCION (tipo nombre del parámetro formal);

deben existir siempre

Ejemplos:

```
double media (double x, double y);
double media (double, double );
float func (void);
void func_calc (int);
```

El prototipo es igual a la declaración de la función o casi igual a ella (lo de casi es por que puede prescindir del nombre de los parámetros formales) sólo que termina con un punto y coma..

Pasaje de parámetros

- Por valor: el **valor** del argumento en la llamada *se copia* en el parámetro formal de la función y ésta sólo opera con esa copia temporal.
  Cuando el argumento es una variable:
  En el código:

Llamada:	Definición:
Nombre de la función (*a*);	***tipo de retorno* Nombre de la función (*tipo x*)**

  En la memoria:

  Existen dos bloques con el mismo contenido, por lo tanto todos los cambios que, desde la función se produzcan en **x** NO tienen efecto sobre **a**.

Ejemplo:

```
int cambia (int, int)
main()
{
    int z;
    int n1= 11;
    int n2 = 84;

    z = cambia (n1, n2);
    printf(" %d  ", z);
       }
```

```
int cambia (int  a, int  b)
{
   int x;

   x = a;
   a = b;
   b = x;
   return (a)
      }
```

- Por referencia: la **dirección** del argumento en la llamada es lo que *se copia* en el parámetro formal de la función.
  Para provocar esto, en C hay que indicar que el parámetro actual sea una dirección y por lo tanto el parámetro formal deberá ser un puntero ya que es la única clase de variable que puede recepcionarla.

  En el código:

Llamada:	Definición:
Nombre de la función (&*a*);	***tipo de retorno* Nombre de la función (*tipo *x*)**

  En la memoria:

Existen dos bloques con distinto contenido, uno es la dirección del otro. Por lo tanto, desde la función se puede afectar el contenido de **a** indicándose modificaciones en ***x** (o sea en el contenido de la variable cuya dirección está en **x)**

Ejemplo:

```
void cambia (int *, int *)
main()
{
    int z;
    int n1= 11;
    int n2 = 84;

    cambia (&n1, &n2);
    z = n1;
    printf(" %d ",z);
          }
```

```
void cambia (int * a, int * b)
   {
     int x;

     x = *a;
     *a = *b;
     *b = x;
       }
```

En C, las funciones NO tienen acceso a los elementos propios de la función invocante salvo que se usen argumentos de tipo puntero.

En el caso especial de que el parámetro actual fuera un arreglo, el pasaje es *indefectiblemente* por puntero (referencia), ya que lo que se transfiere a la función es la dirección del comienzo del arreglo, pues el nombre de un array es en sí, un puntero a su primer elemento. Por lo tanto, la función trabaja con el original y no con una copia del arreglo. Desde ella se tiene acceso y puede alterarse cualquier valor del arreglo.

En el código:

Llamada:

Nombre de la función (*a*);

Definición:

***tipo de retorno* Nombre de la función (*tipo *x*)**

***tipo de retorno* Nombre de la función (*tipo x[ ]*)**

***tipo de retorno* Nombre de la función (*tipo x[tamaño]*)**

cualquiera de estas definiciones son válidas

Según la primer forma de definición, planteando en el código de la función ***(x + 3)** se representaría al 4to. elemento del arreglo, en las otras dos formas, **x[3]** representaría al 4to. elemento.

En C, el arreglo NO se transfiere por valor, es decir, no puede trabajarse desde una función con una copia del arreglo.

Si fuera necesario salvar esta característica de funcionamiento, se podría transferir el arreglo *adentro* de una estructura (como un campo de ella). Así, sería la estructura el

argumento en la llamada y por lo tanto, al transferirla, se generaría en memoria una copia de ella y la función podría trabajar sobre la “copia” del arreglo sin afectar el arreglo original.

Ejemplos:

```
void cambia_1 (int [ ], int )
main()
{
    int ar [5] = {11, 23, 4, 76, 82};

    cambia_1 (ar, 25);

    for (i=0; i<5; i++)
            printf(“%d  ”, ar[i]);

    cambia_1 (&ar[1], 13);

    printf(“\n”);
    for (I=0; I<5, I++)
            printf(“%d  ”,ar[I]);
            }
```

```
// esta función, suma el entero que recibe
// como parámetro a tres elementos
// consecutivos del arreglo transferido
void cambia_1 (int *x, int m)
{
   *x +=m;
   *(x+1) += m;
   *(x+2) += m;
              }
```

```
struct arreglo {
  int a[5];   };

void ceros (struct arreglo);
main()
{
   int i;
   struct arreglo z = {11,23,4,76,82};

   ceros (z);
   printf("\n");
   for (i=0; i<5; i++)
         printf("%d\n",z.a[i]);
   getch();
   return (0);    }
```

```
// esta función, llena el arreglo con ceros
void ceros (struct arreglo x)
 {
   int i;

   for (i=0; i<5; i++)
          x.a[i] = 0;
                   }
```

INDICE

Bibliografía:

“El lenguaje de programación C” de Brian W. Kernighan y Dennis M. Ritchie
“Turbo C (manual de referencia)” de Herbert Schildt

Material elaborado por:

Ing. Gracia María Gagliano