

Programación en C para principiantes

Gorka Urrutia

Table of Contents

Capítulo 1. Introducción.....	1
Sobre el libro	1
Cómo resolver tus dudas	1
El lenguaje C	1
Peculiaridades de C	1
Compiladores de C	1
El editor de código fuente	2
IDE: Entorno de desarrollo integrado	2
El primer programa: Hola Mundo	2
¿Cómo se hace?	5
Nota adicional sobre los comentarios	6
¿Qué sabemos hacer?	6
Ejercicios	6
¿Dudas?	7
¿Has visto un error?	7
Capítulo 2. Mostrando Información por pantalla.	8
Printf: Imprimir en pantalla	8
Gotoxy: Posicionando el cursor (requiere conio.h)	10
Clrscr: Borrar la pantalla (requiere conio.h)	11
Borrar la pantalla (otros métodos)	11
¿Qué sabemos hacer?	11
Ejercicios	12
Capítulo 3. Tipos de Datos.	14
Introducción	14
Notas sobre los nombres de las variables	14
El tipo Int	15
El tipo Char	20
El modificador Unsigned	21
El tipo Float	22
El tipo Double	23
Cómo calcular el máximo valor que admite un tipo de datos.	24
El fichero <limits.h>	25
Overflow: Qué pasa cuando nos saltamos el rango	25
Los tipos short int, long int y long double	26
Resumen de los tipos de datos en C	26
Ejercicios	27
Capítulo 4. Constantes (uso de #define).	28
Introducción	28

Tipos de datos en las constantes	28
Constantes con nombre	30
Capítulo 5. Manipulando datos (operadores)	33
¿Qué es un operador?	33
Operador de asignación	33
Operadores aritméticos	33
Operadores de comparación	38
Operadores lógicos	39
Introducción a los bits y bytes	41
Operadores de bits	42
Operador Sizeof	45
Otros operadores	46
Orden de evaluación de Operadores	46
Ejercicios	49
¿Dudas? ¿Errores?	51
Capítulo 6. Introducir datos por teclado	52
Introducción	52
Scanf	52
Modificadores	55
Ejercicios	55
¿Dudas? ¿Errores?	57
Capítulo 7. Sentencias de control de flujo	58
Introducción	58
Bucles	58
Sentencias de condición	65
Sentencias de salto: Goto	73
Notas sobre las condiciones	73
Ejercicios	75
Capítulo 8. Introducción a las funciones	78
Introducción	78
Definición de una función	78
Dónde se definen las funciones	81
Vida de una variable	87
Ejercicios	87
Capítulo 9. Punteros	90
Introducción	90
La memoria del ordenador	90
Direcciones de variables	90
Qué son los punteros	92
Para qué sirve un puntero y cómo se usa	96
Usando punteros en una comparación	98

Punteros como argumentos de funciones	100
Ejercicios	104
Capítulo 10. Arrays.....	106
¿Qué es un array?	106
Declaración de un Array	110
Sobre la dimensión de un Array	110
Inicializar un array	111
Punteros a arrays.....	120
Paso de un array a una función	126

Capítulo 1. Introducción.

Sobre el libro

Este es un curso para principiantes así que intentaré que no haga falta ningún conocimiento anterior para seguirlo. Muchos otros cursos suponen conocimientos previos pero voy a intentar que eso no suceda aquí.

NOTA IMPORTANTE: Si te pierdes no te desanimes, ponte en contacto conmigo y consúltame (al final del libro tienes varias formas para contactarme). Puede que alguna sección esté mal explicada. De esta forma estarás colaborando a mejorar el libro.

Cómo resolver tus dudas

En la última sección del libro podrás encontrar varias formas de contactar conmigo (email, Twitter, mi blog, etc).

El lenguaje C

El lenguaje C es uno de los más rápidos y potentes que hay hoy en día. Hay quien dice que está desfasado. No se si tendrá futuro pero está claro que presente si tiene. No hay más que decir que el sistema operativo Linux está desarrollado en C en su práctica totalidad. Así que creo que no sólo no perdemos nada aprendiéndolo sino que ganamos mucho. Para empezar nos servirá como base para aprender C++ e introducirnos en el mundo de la programación Windows. Si optamos por Linux existe una biblioteca llamada gtk (o librería, como prefieras) que permite desarrollar aplicaciones estilo Windows con C.

No debemos confundir C con C++, que no son lo mismo. Se podría decir que C++ es una extensión de C. Para empezar en C++ conviene tener una sólida base de C. Existen otros lenguajes como Visual Basic que son muy sencillos de aprender y de utilizar. Nos dan casi todo hecho. Pero cuando queremos hacer algo complicado o que sea rápido debemos recurrir a otros lenguajes (C++, Delphi,...).

Peculiaridades de C

Una de las cosas importantes de C que debes recordar es que es Case Sensitive (sensible a las mayúsculas o algo así). Es decir que para C no es lo mismo escribir Printf que printf. Conviene indicar también que las instrucciones se separan por ";".

Compiladores de C

Un compilador es un programa que convierte nuestro código fuente en un programa ejecutable (me imagino que la mayoría ya lo sabéis pero más vale asegurar). El ordenador trabaja con 0 y 1. Si escribiéramos un programa en el lenguaje del ordenador nos volveríamos locos. Para eso están lenguajes como el C. Nos permiten escribir un programa de manera que sea fácil entenderlo por una persona (el código fuente). Luego es el compilador el que se encarga de convertirlo al

complicado idioma de un ordenador.

En la practica a la hora de crear un programa nosotros escribimos el código fuente, en nuestro caso en C, que normalmente será un fichero de texto normal y corriente que contiene las instrucciones de nuestro programa. Luego se lo pasamos al compilador y este se encarga de convertirlo en un programa.

Si tenemos el código fuente podemos modificar el programa tantas veces como queramos (sólo tenemos que volver a compilarlo), pero si tenemos el ejecutable final no podremos cambiar nada (realmente sí se puede pero es mucho más complicado y requiere más conocimientos).

Existen multitud de compiladores. Yo suelo recomendar el Geany y Code::Blocks, que tiene versiones tanto para Linux como para Windows. Estos programas usan el compilador GNU GCC (<http://gcc.gnu.org>) y se pueden descargar aquí:

- Geany - <http://www.geany.org/>
- Code::Blocks - <http://www.codeblocks.org/>

Nota: Cuando comencé a escribir el curso solía usar el DJGPP en Windows, sin embargo, ahora me decanto más bien por el Geany por la comodidad y facilidad que supone para los principiantes.

El editor de código fuente

El compilador en sí mismo sólo es un programa que traduce nuestro código fuente y lo convierte en un ejecutable. Para escribir nuestros programas necesitamos un editor. La mayoría de los compiladores al instalarse incorporan ya un editor; es el caso de los conocidos Turbo C, Borland C, Code::Blocks, Visual C++,... Pero otros no lo traen por defecto. No debemos confundir por tanto el editor con el compilador. Estos editores suelen tener unas características que nos facilitan mucho el trabajo: permiten compilar y ejecutar el programa directamente, depurarlo (corregir errores), gestionar complejos proyectos, etc. Si nuestro compilador no trae editor la solución más simple usar un editor de texto plano (sin formato).

IDE: Entorno de desarrollo integrado

Para la comodidad de los desarrolladores se crearon lo que se llaman Entornos de Desarrollo Integrado (en inglés IDE). Un IDE es un software que incluye todo lo necesario para la programación: un compilador (con todos sus programas accesorios), un editor con herramientas que ayudan en la creación de programas, un depurador para buscar errores, etc... Es la solución más completa y recomendada.

Existen multitud de IDE que puedes utilizar. Geany y Code::Blocks anteriormente mencionados son muy recomendables en entornos MS Windows, para Linux tenemos montones de opciones, como el Geany, Anjuta o el Kdevelop.

El primer programa: Hola Mundo

En un alarde de originalidad vamos a hacer nuestro primer programa: hola mundo. Nadie puede llegar muy lejos en el mundo de la programación sin haber empezado su carrera con este original y

funcional programa. Allá va:

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

int main()
{
    /* Aquí va el cuerpo del programa */
    printf("Hola mundo\n");
    return EXIT_SUCCESS;
}
```



Si trabajas en Windows

Hay mucha gente que programa en Windows que se queja de que cuando ejecuta el programa no puede ver el resultado. Para evitarlo se puede añadir antes de return 0; la siguiente línea:

```
system("PAUSE");
```

Si esto no funciona prueba a añadir getch();

Otra nota: En compiladores MS Windows, para poder usar la función `system()` debes añadir al principio del fichero la línea:

```
#include <windows.h>
```

Este programa lo único que hace es sacar por pantalla el mensaje:

```
Hola mundo
```

Vamos ahora a comentar el programa línea por línea (Esto no va a ser más que una primera aproximación).

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

`#include` es lo que se llama una directiva. Sirve para indicar al compilador que incluya otro archivo. Cuando en compilador se encuentra con esta directiva la sustituye por el archivo indicado. En este caso son los archivos `stdio.h` (que es donde está definida la función `printf`, que veremos luego) y `stdlib.h`.

```
int main()
```

Es la **función** principal del programa. Todos los programas de C deben tener una función llamada *main*, que es la que primero se ejecuta.

El *int* (viene de Integer=Entero) que tiene al principio significa que cuando la función *main* acabe devolverá un número entero. Este valor se suele usar para saber cómo ha terminado el programa. Normalmente este valor será 0 si todo ha ido bien, o un valor distinto si se ha producido algún error (pero esto lo decidimos nosotros, ya lo veremos). De esta forma si nuestro programa se ejecuta desde otro el programa *padre* sabe como ha finalizado, si ha habido errores o no.

Se puede usar la definición:

```
void main()
```

que no necesita devolver ningún valor, pero se recomienda la forma con *int* que es más correcta. Es posible que veas muchos ejemplos que uso *void main* y en los que falta el `return 0;` del final; el código funciona correctamente pero puede dar un *warning* (un aviso) al compilar dado que no es una práctica correcta.

```
{}
```

Las *llaves* indican, entre otras cosas, el comienzo y el final de una función; en este caso la función *main*.

```
/* Aquí va el cuerpo del programa */
```

Esto es un comentario, el compilador lo ignorará. Sirve para describir el programa a otros desarrolladores o a nosotros mismos para cuando volvamos a ver el código fuente dentro de un tiempo. Conviene acostumbrarse a comentar los programas pero sin abusar de ellos (ya hablaremos sobre esto más adelante).

Los comentarios van encerrados entre */** y **/*.

Un comentario puede ocupar más de una línea. Por ejemplo el comentario:

```
/* Este es un comentario  
que ocupa dos filas */
```

es perfectamente válido.

```
printf( "Hola mundo\n" );
```

Aquí es donde por fin el programa hace algo que podemos ver al ejecutarlo. La función ***printf*** muestra un mensaje por la pantalla.

Al final del mensaje "Hola mundo" puedes ver el símbolo `|n`. Éste hace que después de imprimir el

mensaje se pase a la línea siguiente. Por ejemplo:

```
printf( "Hola mundo\nAdiós mundo" );
```

mostrará:

```
Hola mundo
Adiós mundo
```

Fíjate en el ";" del final. Es la forma que se usa en C para separar una instrucción de otra. Se pueden poner varias en la misma línea siempre que se separen por el punto y coma.

```
return EXIT_SUCCESS;
```

Como he indicado antes el programa al finalizar devuelve un valor entero. Este valor se devuelve usando **return**. El valor devuelto será el que pongamos después de *return*.

En este caso el valor que devolvemos es *EXIT_SUCCESS*, que es una **constante** (un valor predefinido que no cambia) que tiene el valor 0.



¿De dónde sale EXIT_SUCCESS?

El valor de *EXIT_SUCCESS* está definido en el fichero *stdlib.h*.

¿Y por qué no poner `return 0` en lugar de `return EXIT_SUCCESS`?

Sería perfectamente válido usar un 0 en lugar de *EXIT_SUCCESS*, pero el programa se entiende mejor usando esta última. Cuando lees esa línea ya entiendes, sin ninguna duda, que el programa está terminando con éxito.

...y por fin cerramos llaves con lo que termina el programa. Todos los programas finalizan cuando se llega al final de la función *main*.

¿Cómo se hace?

Primero debemos crear el código fuente del programa. Para nuestro primer programa el código fuente es el del listado anterior.

Si estás usando un IDE busca una opción llamada "compile", o make, build o algo así.

Si estamos usando el compilador **gcc** sin IDE tenemos que llamarlo desde la línea de comando:

```
gcc primero.c -o primero
```

Nota adicional sobre los comentarios

Los comentarios se pueden poner casi en cualquier parte. Excepto en medio de una instrucción. Por ejemplo lo siguiente no es válido:

```
pri/* Esto es un comentario */ntf( "Hola mundo" );
```

No podemos cortar a printf por en medio, tendríamos un error al compilar. Lo siguiente puede no dar un error, pero es una fea costumbre:

```
printf( /* Esto es un comentario */ "Hola mundo" );
```

Y por último tenemos:

```
printf( "Hola/* Esto es un comentario */ mundo" );
```

Que no daría error, pero al ejecutar tendríamos:

```
Hola /* Esto es un comentario */ mundo
```

porque /* Esto es un comentario */ queda dentro de las comillas y C lo interpreta como texto, no como un comentario.

¿Qué sabemos hacer?

Pues la verdad es que todavía no hemos aprendido mucho. Lo único que podemos hacer es compilar nuestros programas. Pero paciencia, en seguida avanzaremos.

Ejercicios

Busca los errores en este programa:

```
int main()
{
    /* Aquí va el cuerpo del programa */
    Printf( "Hola mundo\n" );
    return 0;
}
```

Solución:

Si lo compilamos obtendremos un error que nos indicará que no hemos definido la función *Printf*. Esto es porque no hemos incluido la dichosa directiva *#include <stdio.h>*.



En algunos compiladores no es necesario incluir esta directiva, pero es una buena costumbre hacerlo.

Si lo corregimos y volvemos a compilar obtendremos un nuevo error. Otra vez nos dice que desconoce *Printf*. Esta vez el problema es el de las mayúsculas que hemos indicado antes. Lo correcto es poner *printf* con minúsculas.

Te habrás fijado que en esta ocasión hemos usado *return 0* en lugar de *return EXIT_SUCCESS*. En realidad ambas son equivalentes puesto que *EXIT_SUCCESS* tiene el valor 0.

¿Dudas?

Si tienes dudas sobre este capítulo plantéalas en el foro:

<https://elrincondelc.com/foros/viewforum.php?f=47>

¿Has visto un error?

Si has encontrado algún error o quieres sugerir cambios entra aquí:

<https://github.com/gorkau/Libro-Programacion-en-C/blob/master/capitulo1.adoc>

Capítulo 2. Mostrando Información por pantalla.

Printf: Imprimir en pantalla

Siempre he creído que cuando empiezas con un nuevo lenguaje suele gustar el ver los resultados, ver que nuestro programa hace *algo*. Por eso creo que el curso debe comenzar con la función *printf*, que sirve para sacar información por pantalla. Para utilizar la función *printf* en nuestros programas debemos incluir la directiva:

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

al principio de programa. Como hemos visto en el programa hola mundo. Si sólo queremos imprimir una cadena basta con hacer (no olvides el ";" al final):

```
printf( "Cadena" );
```

Esto resultará por pantalla:

```
Cadena
```

Lo que pongamos entre las comillas es lo que vamos a sacar por pantalla. Si volvemos a usar otro *printf*, por ejemplo:

```
#include <stdio.h>
int main() {
    printf( "Cadena" );
    printf( "Segunda" );
    return 0;
}
```

Obtendremos:

```
CadenaSegunda
```

Este ejemplo nos muestra cómo funciona printf. Para escribir en la pantalla se usa un cursor que no vemos. Cuando escribimos algo el cursor va al final del texto. Cuando el texto llega al final de la fila, lo siguiente que pongamos irá a la fila siguiente. Si lo que queremos es sacar cada una en una línea deberemos usar "\n". Es el indicador de retorno de carro. Lo que hace es saltar el cursor de escritura a la línea siguiente:

```
#include <stdio.h>

int main()
{
    printf( "Cadena\n" );
    printf( "Segunda" );
    return 0;
}
```

y tendremos:

```
Cadena
Segunda
```

También podemos poner más de una cadena dentro del printf:

```
printf( "Primera cadena" "Segunda cadena" );
```

Lo que no podemos hacer es meter cosas entre las cadenas:

```
printf( "Primera cadena" texto en medio "Segunda cadena" );
```

esto no es válido. Cuando el compilador intenta interpretar esta sentencia se encuentra *"Primera cadena"* y luego texto en medio, no sabe qué hacer con ello y da un error. Pero ¿qué pasa si queremos imprimir el símbolo " en pantalla? Por ejemplo imaginemos que queremos escribir:

```
Esto es "raro"
```

Si hacemos:

```
printf( "Esto es "raro" );
```

obtendremos unos cuantos errores. El problema es que el símbolo " se usa para indicar al compilador el comienzo o el final de una cadena. Así que en realidad le estaríamos dando la cadena "Esto es", luego extraño y luego otra cadena vacía "". Pues resulta que *printf* no admite esto y de nuevo tenemos errores.

La solución es usar \". Veamos:

```
printf( "Esto es \"extraño\"" );
```

Esta vez todo irá como la seda. Como vemos la contrabarra | sirve para indicarle al compilador que

escriba caracteres que de otra forma no podríamos. Esta contrabarra se usa en C para indicar al compilador que queremos meter símbolos especiales. Pero ¿Y si lo que queremos es usar | como un carácter normal y poner por ejemplo Hola\Adiós? Pues muy fácil, volvemos a usar |:

```
printf( "Hola\\Adiós" );
```

y esta doble | indica a C que lo que queremos es mostrar una |. He aquí un breve listado de códigos que se pueden imprimir:

Código Nombre Significado \a alert Hace sonar un pitido \b backspace Retroceso \n newline Salta a la línea siguiente (salto de línea) \r carriage return Retorno de carro (similar al anterior) \t horizontal tab Tabulador horizontal \v vertical tab Tabulador vertical \\ backslash Barra invertida \? question mark Signo de interrogación \' single quote Comilla sencilla \" double quote Comilla doble

Es recomendable probarlas para ver realmente lo que significa cada una.

Esto no ha sido mas que una introducción a printf. Luego volveremos sobre ella.

Gotoxy: Posicionando el cursor (requiere conio.h)

Esta función sólo está disponible en compiladores de C que dispongan de la biblioteca <conio.h>, de hecho, en la mayoría de compiladores para Linux no viene instalada por defecto. No debería usarse aunque se menciona aquí porque en muchos cursos de formación profesional y en universidades aún se usa. Hemos visto que cuando usamos printf se escribe en la posición actual del cursor y se mueve el cursor al final de la cadena que hemos escrito.

Vale, pero ¿qué pasa cuando queremos escribir en una posición determinada de la pantalla? La solución está en la función gotoxy. Supongamos que queremos escribir *Hola* en la fila 10, columna 20 de la pantalla:

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

int main()
{
    gotoxy( 20, 10 );
    printf( "Hola" );
    return 0;
}
```



para usar gotoxy hay que incluir la biblioteca conio.h).

Fíjate que primero se pone la columna (x) y luego la fila (y). La esquina superior izquierda es la posición (1, 1).

Clrscr: Borrar la pantalla (requiere conio.h)

Ahora ya sólo nos falta saber cómo se borra la pantalla. Pues es tan fácil como usar:

```
clrscr();
```

(clear screen, borrar pantalla).

Esta función no sólo borra la pantalla, sino que además sitúa el cursor en la posición (1, 1), en la esquina superior izquierda.

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

int main()
{
    clrscr();
    printf( "Hola" );
    return 0;
}
```

Este método sólo vale para compiladores que incluyan el fichero conio.h. Si tu sistema no lo tiene puedes consultar la sección siguiente.

Borrar la pantalla (otros métodos)

Existen otras formas de borrar la pantalla aparte de usar conio.h.

Si usas DOS:

```
system("cls"); //Para DOS
```

Si usas Linux:

```
system("clear"); // Para Linux
```

Otra forma válida para ambos sistemas:

```
char a[5]={27,[,2,J,0}; /* Para ambos (en DOS cargando antes ansi.sys) */ printf("%s",a);
```

¿Qué sabemos hacer?

Bueno, ya hemos aprendido a sacar información por pantalla. Si quieres puedes practicar con las instrucciones printf, gotoxy y clrscr. Lo que hemos visto hasta ahora no tiene mucho secreto, pero ya veremos cómo la función printf tiene mayor complejidad.

Ejercicios

Ejercicio 1: Busca los errores en el programa (este programa usa conio.h, pero aunque tu compilador no la incluya aprenderás algo con este ejercicio).

```
#include <stdio.h>
int main()
{
    ClrScr();
    gotoxy( 10, 10 )
    printf( Estoy en la fila 10 columna 10 );
    return 0;
}
```

Solución:

ClrScr está mal escrito, debe ponerse todo en minúsculas, recordemos una vez más que el C diferencia las mayúsculas de las minúsculas. Además no hemos incluido la directiva #include <conio.h>, que necesitamos para usar clrscr() y gotoxy(). Tampoco hemos puesto el punto y coma (;) después del gotoxy(10, 10). Después de cada instrucción debe ir un punto y coma. El último fallo es que el texto del printf no lo hemos puesto entre comillas. Lo correcto sería: printf("Estoy en la fila 10 columna 10");

Ejercicio 2: Escribe un programa que borre la pantalla y escriba en la primera línea tu nombre y en la segunda tu apellido:

Solución:

```
#include <stdio.h>
#include <conio.h>
int main()
{
    clrscr();
    printf( "Gorka\n" );
    printf( "Urrutia" );
    return 0;
}
```

También se podía haber hecho todo de golpe:


```
#include <stdio.h>
#include <conio.h>
int main()
{
    clrscr();
    printf( "Gorka\nUrrutia" );
    return 0;
}
```

Ejercicio 3: Escribe un programa que borre la pantalla y muestre el texto "estoy aqui" en la fila 10, columna 20 de la pantalla.

Solución:

```
#include <stdio.h>
#include <conio.h>
int main() {
    clrscr();
    gotoxy( 20, 10 );
    printf( "Estoy aqui" );
    return 0;
}
```

Capítulo 3. Tipos de Datos.

Introducción

Cuando usamos un programa es muy importante manejar datos. En C podemos almacenar los datos en variables. Una variable es una porción de la memoria del ordenador que queda asignada para que nuestro programa pueda almacenar datos. El contenido de las variables se puede ver o cambiar en cualquier momento. Estas variables pueden ser de distintos tipos dependiendo del tipo de dato que queramos meter. No es lo mismo guardar un nombre que un número.

Hay que recordar también que la memoria del ordenador es limitada, así que cuando guardamos un dato, debemos usar sólo la memoria necesaria. Por ejemplo si queremos almacenar el número 400 usaremos una variable tipo *int* (la estudiamos más abajo) que ocupa menos memoria que una variable de tipo *float*. Si tenemos un ordenador con 32Mb de RAM parece una tontería ponernos a ahorrar bits (1Mb=1024Kb, 1Kb=1024bytes, 1byte=8bits), pero si tenemos un programa que maneja muchos datos puede no ser una cantidad despreciable. Además ahorrar memoria es una buena costumbre.



Por si alguno tiene dudas: No hay que confundir la memoria con el espacio en el disco duro. Son dos cosas distintas. La capacidad de ambos se mide en bytes, y la del disco duro suele ser mayor que la de la memoria RAM. La información en la RAM se pierde al apagar el ordenador, la del disco duro permanece. Cuando queremos guardar un fichero lo que necesitamos es espacio en el disco duro. Cuando queremos ejecutar un programa lo que necesitamos es memoria RAM. La mayoría me imagino que ya lo sabéis, pero me he encontrado muchas veces con gente que los confunde).

Notas sobre los nombres de las variables

A las variables no se les puede dar cualquier nombre pero siguiendo unas sencillas normas:

- No se pueden poner más que letras de la *a* a la *z* (la *ñ* no vale), números y el símbolo *_*.
- No se pueden poner signos de admiración, ni de interrogación...
- El nombre de una variable puede contener números, pero su primer carácter no puede ser un número.

Ejemplos de nombres válidos:

- camiones
- numero
- buffer
- a1
- j10hola29
- num_alumnos

Ejemplos de nombres no válidos:

- 1abc
- nombre?
- número
- num/alumnos

Tampoco valen como nombres de variable las palabras reservadas que usa el compilador. Por ejemplo: for, main, do, while. Lista de palabras reservadas según el estándar ISO-C90:

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Por último es interesante señalar que el C distingue entre mayúsculas y minúsculas. Por lo tanto:

- Nombre
- nombre
- NOMBRE

serían tres variables distintas.

El tipo Int

En una variable de este tipo se almacenan números enteros (sin decimales). El rango de valores que admite es -32.768 a 32.767.



Nota importante: el rango indicado (de -32.768 a 32.767) puede variar de un compilador a otro, en este caso sería un compilador donde el tipo int es de 16 bits.

¿Por qué estos números tan extraños? Esto se debe a los 16 bits mencionados. $2^{16} = 65.536$, que dividido por dos nos da 32.768. Por lo tanto, en una variable de este tipo podemos almacenar números negativos desde el -32.768 hasta el -1 y números desde el 0 hasta el 32.767.

Cuando definimos una variable lo que estamos haciendo es decirle al compilador que nos reserve una zona de la memoria para almacenar datos de tipo *int*. Para guardarla necesitaremos por tanto 16 bits de la memoria del ordenador.

Las variables de tipo int se definen así:

```
int número;
```

Esto hace que declaremos una variable llamada número que va a contener un número entero.

¿Pero dónde se declaran las variables?

Tenemos dos posibilidades, una es declararla como global y otra como local. Por ahora vamos a decir que global es aquella variable que se declara fuera de la función main y local la que se declara dentro. Variable global:

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

int x;
int main()
{
}
```

Variable local:

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

int main()
{
    int x;
}
```

La diferencia práctica es que las variables globales se pueden usar en cualquier función (o procedimiento). Las variables locales sólo pueden usarse en el procedimiento en el que se declaran. Como por ahora sólo tenemos el procedimiento (o función, o rutina, o subrutina, como prefieras) main esto no debe preocuparnos mucho por ahora. Cuando estudiemos cómo hacer un programa con más funciones aparte de main volveremos sobre el tema. Sin embargo debes saber que es buena costumbre usar variables locales que globales. Ya veremos por qué.

Podemos declarar más de una variable en una sola línea:

```
int x, y;
```

Mostrar variables por pantalla

Vamos a ir un poco más allá con la función *printf*. Supongamos que queremos mostrar el contenido de la variable x por pantalla:

```
printf( "%i", x );
```

Suponiendo que x valga 10 (x=10) en la pantalla tendríamos:

```
10
```

Empieza a complicarse un poco ¿no? Vamos poco a poco. ¿Recuerdas el símbolo "\" que usábamos para sacar ciertos caracteres? Bueno, pues el uso del "%" es parecido. "%i" no se muestra por pantalla, se sustituye por el valor de la variable que va detrás de las comillas (%i, de integer=entero en inglés). Para ver el contenido de dos variables, por ejemplo x e y, podemos hacer:

```
printf( "%i ", x );  
printf( "%i", y );
```

resultado (suponiendo x=10, y=20):

```
10 20
```

Pero hay otra forma mejor:

```
printf( "%i %i", x, y );
```

...y así podemos poner el número de variables que queramos. Obtenemos el mismo resultado con menos trabajo. No olvidemos que por cada variable hay que poner un %i dentro de las comillas.

También podemos mezclar texto con enteros:

```
printf( "El valor de x es %i, ¡que bien!\n", x );
```

que quedará como:

```
El valor de x es 10, ¡que bien!
```

Como vemos %i al imprimir se sustituye por el valor de la variable.

A veces %d, a veces %i

Seguramente habrás visto que en ocasiones se usa el modificador %i y otras %d ¿cuál es la diferencia entre ambos? ¿cuál debe usarse? En realidad, cuando los usamos en un *printf* no hay ninguna diferencia, se pueden usar indistintamente. La diferencia está cuando se usa con otras funciones como *scanf* (esta función la estudiaremos más adelante).

Hay varios modificadores para los números enteros:

Tipo de variable	Descripción	Modificador
<i>int</i>	Entero decimal	%i
<i>int</i>	Entero decimal	%i
<i>unsigned int</i>	Entero decimal sin signo	%u
<i>int</i>	Entero octal	%o
<i>int</i>	Entero hexadecimal	%x

Podemos verlos en acción con el siguiente ejemplo:

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

int main()
{
    int numero = 13051;

    printf("Decimal usando 'i': %i\n", numero);
    printf("Decimal usando 'd': %d\n", numero);
    printf("Hexadecimal: %x\n", numero);
    printf("Octal: %o\n", numero);
    return EXIT_SUCCESS;
}
```

Este ejemplo mostraría:

```
Decimal usando 'i': 13051
Decimal usando 'd': 13051
Hexadecimal: 32fb
Octal: 31373
```

Asignar valores a variables de tipo int

La asignación de valores es tan sencilla como:

```
x = 10;
```

También se puede dar un valor inicial a la variable cuando se define:

```
int x = 15;
```

También se pueden dar valores iniciales a varias variables en una sola línea:

```
int x = 15, y = 20;
```

Hay que tener cuidado con lo siguiente:

```
int x, y = 20;
```

Podríamos pensar que *x* e *y* son igual a 20, pero no es así. La variable *x* está sin valor inicial y la variable *_y*: tiene el valor 20.

Veamos un ejemplo para resumir todo:

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

int main()
{
    int x = 10;
    printf( "El valor inicial de x es %i.\n", x );
    x = 50;
    printf( "Ahora el valor de x es %i.\n", x );

    return EXIT_SUCCESS;
}
```

Cuya salida será:

```
El valor inicial de x es 10
Ahora el valor es 50
```

Valores iniciales

¡Importante! Antes de usar una variable debemos darle un valor inicial. En compiladores nuevos se nos avisa si intentamos usar una variable antes de asignarle un valor inicial. En los más viejos no se nos avisa y esto puede dar lugar a errores "lógicos". Prueba a cambiar:



```
int x = 10;
```

por:

```
int x;
```

El tipo Char

Las variables de tipo char se puede usar para almacenar caracteres. Los caracteres se almacenan en realidad como números del 0 al 255. Los 128 primeros (0 a 127) son el ASCII estándar. El resto es el ASCII extendido y depende del idioma y del ordenador. Consulta la tabla ASCII en el anexo (más información sobre los caracteres ASCII: <http://es.wikipedia.org/wiki/Ascii>).

Para declarar una variable de tipo char hacemos:

```
char letra;
```

En una variable char sólo podemos almacenar solo una letra, no podemos almacenar ni frases ni palabras. Eso lo veremos más adelante (strings, cadenas). Para almacenar un dato en una variable *char* tenemos dos posibilidades:

```
letra = 'A';
```

o:

```
letra = 65;
```

En ambos casos se almacena la letra A en la variable. Esto es así porque el código ASCII de la letra A es el 65.

Para imprimir un *char* usamos el símbolo `%c` (c de character=carácter en inglés):

```
letra = 'A';  
printf( "La letra es: %c.", letra );
```

resultado:

```
La letra es A.
```

También podemos imprimir el valor ASCII de la variable usando `%i` en vez de `%c`:

```
letra = 'A';  
printf( "El número ASCII de la letra %c es: %i.", letra, letra );
```

resultado:

```
El código ASCII de la letra A es 65.
```


Como vemos la única diferencia para obtener uno u otro es el modificador (%c ó %i) que usemos. Las variables tipo char se pueden usar (y de hecho se usan mucho) para almacenar enteros. Si necesitamos un número pequeño (entre -128 y 127) podemos usar una variable char (8bits) en vez de una int (16bits), con el consiguiente ahorro de memoria. Todo lo demás dicho para los datos de tipo “int” se aplica también a los de tipo “char”.

Una curiosidad:

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

int main()
{
    char letra = 'A';
    printf( "La letra es: %c y su valor ASCII es: %i.\n", letra,
    letra );
    letra = letra + 1;
    printf( "Ahora es: %c y su valor ASCII es: %i.\n", letra, letra );

    return EXIT_SUCCESS;
}
```

En este ejemplo letra comienza con el valor A, que es el código ASCII 65. Al sumarle 1 pasa a tener el valor 66, que equivale a la letra B (código ASCII 66). La salida de este ejemplo sería:

```
La letra es A y su valor ASCII es 65
Ahora es B y su valor ASCII es 66
```

El modificador Unsigned

Este modificador (que significa sin signo) modifica el rango de valores que puede contener una variable. Solo admite valores positivos. Si hacemos:

```
unsigned char variable;
```

Esta variable en vez de tener un rango de -128 a 127 pasa a tener un rango de 0 a 255. Los indicadores de signo *signed* y *unsigned* solo pueden aplicarse a los tipos enteros. El primero indica que el tipo puede almacenar tanto valores positivos como negativos y el segundo indica que solo se admiten valores no negativos, esto es, solo se admite el cero y valores positivos. Si se declara una variable de tipo short, int o long sin utilizar un indicador de signo esto es equivalente a utilizar el indicador de signo signed. Por ejemplo:

```
signed int i;
int j;
```

Declara dos variables de tipo *signed int*.

La excepcion es el tipo **char**. Cuando se declara una variable de tipo char sin utilizar un indicador de signo si esta variable es equivalente a signed char o a unsigned char depende del compilador que estemos utilizando. Por lo mismo si debemos tener total certeza de que nuestras variables de tipo char puedan almacenar (o no) valores negativos es mejor indicarlo explicitamente utilizando ya sea signed char o unsigned char.

El tipo Float

En este tipo de variable podemos almacenar números decimales, no sólo enteros como en los anteriores. El mayor número que podemos almacenar en un float es 3,4E38 y el más pequeño 3,4E-38. ¿Qué significa 3,4E38? Esto es equivalente a $3,4 * 10^{38}$, que es el número: 340.000.000.000.000.000.000.000.000.000.000.000.000.000.000.000 El número 3,4E-38 es equivalente a $3,4 * 10^{-38}$, vamos un número muy, muy pequeño.

Declaración de una variable de tipo float:

```
float número;
```

Para imprimir valores tipo float Usamos %f.

```
int main()
{
    float num=4060.80;
    printf( "El valor de num es : %f", num );
}
```

Resultado:

```
El valor de num es: 4060.80
```

Si queremos escribirlo en notación exponencial usamos %e:

```
float num = 4060.80;
printf( "El valor de num es: %e", num );
```

Que da como resultado:

```
El valor de num es: 4.06080e003
```

El tipo Double

En las variables tipo double se almacenan números reales. El mayor número que se pueda almacenar es el 1,7E308 y el más pequeño del 1,7E-307. Se declaran como double:

```
double número;
```

Para imprimir se usan los mismos modificadores que en float.

Números decimales ¿float o double?

Cuando escribimos un número decimal en nuestro programa, por ejemplo 10.30, ¿de qué tipo es? ¿float o double?

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

int main()
{
    printf( "%f\n", 10.30 );
    return EXIT_SUCCESS;
}
```

Por defecto, si no se especifica nada, las constantes son de tipo double. Para especificar que queremos que la constante sea float debemos especificar el sufijo “f” o “F”. Si queremos que la constante sea de tipo long double usamos el sufijo “l” o “L”. Veamos el siguiente programa:

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

int main() {
    float num;

    num = 10.20 * 20.30;

    return EXIT_SUCCESS;
}
```

En este caso, ya que no hemos especificado nada, tanto 10.20 como 20.30 son de tipo double. La operación se hace con valores de tipo double y luego se almacena en un float. Al hacer una operación con double tenemos mayor precisión que con floats, sin embargo es innecesario, ya que en este caso al final el resultado de la operación se almacena en un float, de menor precisión. El programa sería más correcto así:

```
int main()
{
    float num;
    num = 10.20f * 20.30f;
}
```

Cómo calcular el máximo valor que admite un tipo de datos

Lo primero que tenemos que conocer es el tamaño en bytes de ese tipo de dato. Vamos a ver un ejemplo con el tipo INT. Hagamos el siguiente programa:

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

int main()
{
    printf( "El tipo int ocupa %lu bytes\n", sizeof(int) );
    return EXIT_SUCCESS;
}
```



En este caso usamos el modificador `%lu` porque `sizeof()` devuelve un valor del tipo *long unsigned int*.

La función `sizeof()` calcula el tamaño en bytes de una variable o un tipo de datos. En mi ordenador el resultado era (en tu ordenador podría ser diferente):

El tipo int ocupa 4 bytes.

Como sabemos $1 \text{ byte} = 8 \text{ bits}$. Por lo tanto el tipo int ocupa $4 \times 8 = 32 \text{ bits}$. Ahora para calcular el máximo número debemos elevar 2 al número de bits obtenido. En nuestro ejemplo: $2^{32} = 4.294.967.296$. Es decir en un int se podrían almacenar 4.294.967.296 números diferentes.

El número de valores posibles y únicos que pueden almacenarse en un tipo entero depende del número de bits que lo componen y esta dado por la expresión 2^N donde N es el número de bits. Si usamos un tipo unsigned (sin signo, se hace añadiendo la palabra unsigned antes de int) tenemos que almacenar números positivos y negativos.

Así que de los 4.294.967.296 posibles números la mitad serán positivos y la mitad negativos. Por lo tanto tenemos que dividir el número anterior entre 2 = 2.147.483.648. Como el 0 se considera positivo el rango de números posibles que se pueden almacenar en un int sería: -2.147.483.648 a 2.147.483.647.

El fichero <limits.h>

Existe un fichero llamado limits.h en el directorio includes de nuestro compilador (sea cual sea) en el que se almacena la información correspondiente a los tamaños y máximos rangos de los tipos de datos char, short, int y long (signed y unsigned) de nuestro compilador. Se recomienda como curiosidad examinar este fichero.

Overflow: Qué pasa cuando nos saltamos el rango

El overflow es lo que se produce cuando intentamos almacenar en una variable un número mayor del máximo permitido. El comportamiento es distinto para variables de números enteros y para variables de números en coma flotante.

Con números enteros

Supongamos que en nuestro ordenador el tipo int es de 32 bits. El número máximo que se puede almacenar en una variable tipo int es por tanto 2.147.483.647 (ver apartado anterior). Si nos pasamos de este número el que se guardará será el siguiente pero empezando desde el otro extremo, es decir, el -2.147.483.648. El compilador seguramente nos dará un aviso (warning) de que nos hemos pasado.

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

int main() {
    int num1;

    num1 = 2147483648;
    printf( "El valor de num1 es: %i\n", num1 );

    return EXIT_SUCCESS;
}
```

El resultado que obtenemos es:

```
El valor de num1 es: -2147483648
```

Comprueba si quieres que con el número anterior (2.147.483.647) no pasa nada.

Con números en coma flotante

El comportamiento con números en coma flotante es distinto. Dependiendo del ordenador si nos pasamos del rango al ejecutar un programa se puede producir un error y detenerse la ejecución. Con estos números también existe otro error que es el underflow. Este error se produce cuando almacenamos un número demasiado pequeño (3,4E-38 en float).

Los tipos short int, long int y long double

Existen otros tipos de datos que son variaciones de los anteriores que son: short int, long int, long long y long double.

En realidad, dado que el tamaño de los tipos depende del compilador, lo único que nos garantiza es que:

- El tipo long long no es menor que el tipo int.
- El tipo long no es menor que el tipo int.
- El tipo int no es menor que el tipo short.

Resumen de los tipos de datos en C

Los números en C se almacenan en variables llamadas “de tipo aritmético”. Estas variables a su vez se dividen en variables de tipos enteros y de tipos en coma flotante. Los tipos enteros son char, short int, int y long int. Los tipos short int y long int se pueden abreviar a solo short y long. Esto es algo orientativo, depende del sistema.

Por ejemplo en un sistema de 16 bits podría ser algo así:

Tipo	Datos almacenados	Nº de Bits	Valores posibles (Rango)	Rango usando unsigned
char	Caracteres y enteros pequeños	8	-128 a 127	0 a 255
int	Enteros	16	-32.768 a 32.767	0 a 65.535
long	Enteros largos	32	-2.147.483.648 a 2.147.483.647	0 a 4.294.967.295
float	Números reales (coma flotante)	32	3,4E-38 a 3,4E38	No se aplica
double	Números reales (coma flotante doble)	64	1,7E-307 a 1,7E308	No se aplica

Como hemos mencionado antes esto no siempre es cierto, depende del ordenador y del compilador. Para saber en nuestro caso qué tamaño tienen nuestros tipos de datos debemos hacer lo siguiente.

Ejemplo para int:

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

int main() {
    printf( "Tamaño (en bits) de int = %i\n", sizeof( int )*8 );

    return EXIT_SUCCESS;
}
```

Ya veremos más tarde lo que significa sizeof. Por ahora basta con saber que nos dice cual es el tamaño de una variable o un tipo de dato.

Ejercicios

Ejercicio 1: Busca los errores:

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

int main()
{
    int número;
    número = 2;
    return EXIT_SUCCESS;
}
```

Solución: Los nombres de variables no pueden llevar acentos, luego al compilar número dará error.

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

int main()
{
    int número;
    número = 2;
    printf( "El valor es %i" número );
    return EXIT_SUCCESS;
}
```

Solución: Falta la coma después de "El valor es %i". Además la segunda vez número está escrito con mayúsculas.

Capítulo 4. Constantes (uso de #define).

Introducción

Las constantes son aquellos datos que no pueden cambiar a lo largo de la ejecución de un programa.

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

int main()
{

    double radio, perimetro;

    radio = 20;
    perimetro = 2 * 3.1416 * radio;
    printf( "El perimetro es: %f", perimetro );
    return EXIT_SUCCESS;

}
```

radio y *perimetro* son variables, su valor puede cambiar a lo largo del programa. Sin embargo 20, 2 y 3.1416 son constantes, no hay manera de cambiarlas. El valor 3.1416 no cambia a lo largo del programa, ni entre ejecución y ejecución. Solo cambiará cuando edites el programa y lo cambies tu mismo. En resumen, cuando escribimos directamente un número se le llama una constante.

Tipos de datos en las constantes

En el capítulo anterior vimos que las existen diferentes tipos de datos para las variables. Las constantes también tienen tipos de datos. Recordemos que especificábamos el tipo de dato de la variable usando int, float, double y otros. Con las constantes indicamos el tipo dependiendo del sufijo que empleemos después de la constante. Veamos unos ejemplos:

```
a = 100; /* 100 es de tipo signed int */
b = 200U; /* 200U es de tipo unsigned int */
c = 300L; /* 300L es de tipo signed long */
d = 400UL; /* 400UL es de tipo unsigned long */
```

Pero ¿para qué queremos indicar el tipo de dato de una constante? Al fin y al cabo son todos números. Veremos más adelante que es muy importante, sobre todo a la hora de hacer ciertas operaciones matemáticas.

Constantes en base 10 sin parte fraccionaria



NOTA

Los números en base 10 son los que llamamos decimales. Se llaman así porque los números se pueden representar usando como base el 10:

$$3.284 = 3 \times 1000 + 2 \times 100 + 8 \times 10 + 4 = 3 \times 10^3 + 2 \times 10^2 + 8 \times 10^1 + 4 \times 10^0$$

Recordemos que también hay números binarios (en base 2), hexadecimales y octales.

Las constantes en base 10 y sin fracción ni exponente son de tipo *signed int*.

¿Y que pasa si una constante "no cabe" en el tipo indicado?

Supongamos un ordenador de 16 bits donde el valor máximo que se puede almacenar en el tipo `int` es 32.767 y (por poner un ejemplo) en nuestro programa tenemos:

```
int a = 32768; /* recordemos 32768 "no cabe" en un int de 16 bits. */
```

¿Que es lo que sucede?

Cuando el número no cabe en el tipo que se está indicando (en este caso no se indica nada así que se considera como un tipo `int`) se comprueba si cabe en el siguiente tipo de dato. Si tampoco cabe se prueba con el siguiente. El orden que se sigue es:

1. `int`
2. `long`
3. `unsigned long`

Debido a que en nuestro ejemplo 32.768 no cabe en un `int` se comprueba con el tipo *signed long*. Si en éste tampoco cabe se considera que el tipo de la constante es *unsigned long*.

Si la constante en cuestión tiene uno de los dos sufijos *U* o *L* el tipo a utilizar se restringe (limita) y selecciona en este orden:

A) En el caso de utilizar *U*:

1. `unsigned int`
2. `unsigned long`

B) En el caso de utilizar *L*:

1. `signed long`
2. `unsigned long`

Constantes en base 10 y con decimales

Las constantes en base 10 y con un punto decimal y/o exponente son de tipo double.

Algunos ejemplos:

```
a = 100.0; /* 100.0 es de tipo 'double' */
b = 10E2; /* 10E2 es de tipo 'double' */
```

Nota técnica:

Las constantes de punto flotante son de tipo double a menos que se utilice uno de estos sufijos ya sea en minúsculas o mayúsculas:

- El sufijo *F* indica que la constante es de tipo float.
- El sufijo *L* indica que la constante es de tipo long double.

Solo se puede utilizar uno de estos sufijos pero no ambos.

Algunos ejemplos:

```
a = 100.0F /* 100.0F es de tipo float */
b = 200.0 /* 200.0 es de tipo double */
c = 300.0L /* 300.0L es de tipo long double */
```

Constantes con nombre

Imagina el siguiente programa:

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

int main()
{
    float precio;

    precio = ( 4 * 25 * 100 ) * ( 1.16 );

    printf( "El precio total es: %f", precio );

    return EXIT_SUCCESS;
}
```

Es un programa sencillo y que funciona bien. Sin embargo ¿qué sentido tienen los números 4, 25, 100 y 1,16? Es difícil saberlo. Es bastante habitual escribir un programa así, volver a echarle un

vistazo unos meses más tarde y no recordar qué eran esos números.

Ahora mira este otro programa:

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

#define CAJAS 4
#define UNIDADES_POR_CAJA 25
#define PRECIO_POR_UNIDAD 100
#define IMPUESTOS 1.16

int main()
{
    float precio;

    precio =
    ( CAJAS * UNIDADES_POR_CAJA * PRECIO_POR_UNIDAD ) *
    ( IMPUESTOS );

    printf( "El precio total es: %f", precio );
    return EXIT_SUCCESS;
}
```

Ahora todos los números tienen un significado claro. Es porque esta vez estamos usando *constantes con nombre*.

`#define` es lo que se llama una *directiva*. Estas directivas se utilizan, entre otras cosas, para definir constantes. Los usos de `#define` y de otras directivas los veremos en el capítulo de directivas.

Las constantes, una vez definidas, no pueden cambiar su valor. No son como las variables. Cuando hacemos:

```
#define CAJAS 4
```

estamos diciendo que, dentro de nuestro programa, donde aparezca la palabra `CAJAS` hay que sustituirlo por el valor 4.

Para definir constantes hay que seguir unas sencillas normas:

- Sólo se puede definir una constante por línea.
- No llevan ; al final.
- Se suelen escribir en mayúsculas aunque no es obligatorio.

También podemos definir una constante usando el valor de otras. Por supuesto las otras tienen que estar definidas antes:

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

#define CAJAS 4
#define UNIDADES_POR_CAJA 25
#define PRECIO_POR_UNIDAD 100
#define PRECIO_POR_CAJA UNIDADES_POR_CAJA * PRECIO_POR_UNIDAD
#define IMPUESTOS 1.16

int main()

{

    float precio;

    precio = ( CAJAS * PRECIO_POR_CAJA ) * ( IMPUESTOS );

    printf( "El precio total es: %f", precio );

    return EXIT_SUCCESS;

}
```

Capítulo 5. Manipulando datos (operadores)

¿Qué es un operador?

Un operador sirve para manipular datos. Los hay de varios tipos: de asignación, de relación, lógicos, aritméticos y de manipulación de bits. En realidad los nombres tampoco importan mucho; aquí lo que queremos es aprender a programar, no aprender un montón de nombres.

Operador de asignación

Este es un operador que ya hemos visto en el capítulo de Tipos de Datos. Sirve para dar un valor a una variable. Este valor puede ser un número que tecleamos directamente u otra variable:

```
a = 3; /* Metemos un valor directamente */
```

o

```
a = b; /* Le damos el valor de una variable */
```

Podemos dar valores a varias variables a la vez:

```
a = b = c = 10; /* Damos a las variables a,b,c el valor 10 */
```

También podemos asignar a varias variables el valor de otra de un solo golpe:

```
a = b = c = d; /* a,b,c toman el valor de d */
```

Operadores aritméticos

Los operadores aritméticos son aquellos que sirven para realizar operaciones tales como suma, resta, división, multiplicación y módulo (o resto o residuo).

Operador (+) : Suma

Este operador permite sumar variables:

```

#include <stdio.h>
#include <stdlib.h>

int main()
{

    int a = 2;
    int b = 3;
    int c;

    c = a + b;

    printf ( "Resultado = %i\n", c );

    return EXIT_SUCCESS;

}

```

El resultado será 5 obviamente.

Por supuesto se pueden sumar varias variables o variables más constantes:

```

#include <stdio.h>
#include <stdlib.h>

int main()
{

    int a = 2;
    int b = 3;
    int c = 1;
    int d;

    d = a + b + c + 4;

    printf ( "Resultado = %i\n", d );

    return EXIT_SUCCESS;

}

```

El resultado es 10.

Podemos utilizar este operador para incrementar el valor de una variable:

```
x = x + 5;
```

Pero existe una forma abreviada:

```
x += 5;
```

Esto suma el valor 5 al valor que tenía la variable x. Veamos un ejemplo:

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

int main()
{
    int x, y;

    x = 3;
    y = 5;
    x += 2;

    printf( "x = %i\n", x );

    x += y; /* esto equivale a x = x + y */

    printf( "x = %i\n", x );

    return EXIT_SUCCESS;
}
```

Resultado:

```
x = 5
x = 10
```

Operador (++) : Incremento

Este operador equivale a sumar uno a la variable:

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

int main()
{
    int x = 5;

    printf ( "Valor de x = %i\n", x );

    x++;

    printf ( "Valor de x = %i\n", x );

    return EXIT_SUCCESS;
}
```

Resultado:

```
Valor de x = 5
Valor de x = 6
```

Se puede poner antes o después de la variable.

Operador (-) : Resta/Negativo

Este operador tiene dos usos, uno es la resta que funciona como el operador suma y el otro es cambiar de signo.

Resta:

```
x = x - 5;
```

Para la operación resta se aplica todo lo dicho para la suma. Se puede usar también como:

```
x -= 5;
```

Pero también tiene el uso de cambiar de signo. Poniéndolo delante de una variable o constante equivale a multiplicarla por -1.


```

#include <stdio.h>
#include <stdlib.h>

int main()
{
    int a, b;

    a = 1;
    b = -a;

    printf( "a = %i, b = %i\n", a, b );

    return EXIT_SUCCESS;
}

```

Resultado: a = 1, b = -1. No tiene mucho misterio.

Operador (--): Decremento

Es equivalente a ++ pero en vez de incrementar disminuye el valor de la variable. Equivale a restar uno a la variable.

Operador (*): Multiplicación y punteros

Este operador sirve para multiplicar y funciona de manera parecida a los anteriores.

También sirve para definir y utilizar punteros, pero eso lo veremos más tarde.

Operador (/): División

Este funciona también como los anteriores pero hay que tener dos cosas en cuenta:

División de enteros

Si dividimos dos números en coma flotante (tipo *float*) tenemos la división con sus correspondientes decimales. Pero si dividimos dos enteros obtenemos un número entero. Es decir que si dividimos 4/3 tenemos como resultado 1. Se hace un redondeo por truncamiento y se eliminan los decimales.

Para conseguir el resultado correcto debemos usar 4.0/3.0, dado que 4 se considera como *int* y 4.0 como *float*.

Al dividir dos enteros el resultado es siempre un número entero, aunque luego lo saquemos por pantalla usando %f no obtendremos la parte decimal.

Si queremos saber cuál es el resto (o módulo) usamos el operador %, que vemos más abajo.

División por cero

En C no podemos dividir un número por cero, es una operación ilegal. Hay que evitar esto pues se producirá un error en nuestro programa. Los operadores división y módulo no aceptan como segundo parámetro el cero. No se puede usar:

- El valor 0 con los operadores de división y módulo.
- El valor 0.0 con el operador de división.

Operador (%) : Módulo o Resto

Si con el anterior operador obteníamos el módulo o cociente de una división entera con éste podemos tener el resto. **Sólo funciona con enteros**, no vale para números *float* o *double*.

Cómo se usa:

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

int main()
{
    int a, b;

    a = 18;

    b = 5;

    printf( "Resto de la división: %d \n", a % b );

    return EXIT_SUCCESS;
}
```

Operadores de comparación

Los operadores de condición se utilizan para comprobar las condiciones de las sentencias de control de flujo (las estudiaremos en el capítulo sentencias).

Cuando se evalúa una condición el resultado que se obtiene es 0 si no se cumple y un número distinto de 0 si se cumple. Normalmente cuando se cumplen devuelven un 1.

Los operadores de comparación son:

==	igual que	se cumple si son iguales
!=	distinto que	se cumple si son diferentes
>	mayor que	se cumple si el primero es mayor que el segundo

==	igual que	se cumple si son iguales
<	menor que	se cumple si el primero es menor que el segundo
>=	mayor o igual que	se cumple si el primero es mayor o igual que el segundo
≤	menor o igual que	se cumple si el primero es menor o igual que el segundo

Veremos la aplicación de estos operadores en el capítulo Sentencias. Pero ahora vamos a ver un ejemplo:

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

int main()
{
    printf( "10 > 5 da como resultado %i\n", 10>5 );
    printf( "10 < 5 da como resultado %i\n", 10<5 );
    printf( "5 == 5 da como resultado %i\n", 5==5 );
    printf( "10 == 5 da como resultado %i\n", 10==5 );

    return EXIT_SUCCESS;
}
```

Como se puede ver al ejecutar este programa: * cuando la condición se cumple el resultado es un 1 (true) * cuando no se cumple es un 0 (false).

No sólo se pueden comparar constantes, también se pueden comparar variables.

Operadores lógicos

Estos son los que nos permiten unir varias comparaciones, por ejemplo: $10 > 5$ y $6 == 6$. Los operadores lógicos son: **AND (&&)**, **OR (|)**, **NOT(!)**.

Operador && (AND, en castellano Y): Devuelve un 1 si se cumplen dos condiciones.

```
printf( "Resultado: %i", (10==10 && 5>2 ); /* Resultado: 1 */
printf( "Resultado: %i", (10==10 && 5<2 ); /* Resultado: 0 */
```

Operador || (OR, en castellano O): Devuelve un 1 si se cumple una de las dos condiciones.

```
printf( "Resultado: %i", (10==10 || 5<2 ); /* Resultado: 1 */
```

Operador ! (NOT, negación): Si la condición se cumple NOT hace que no se cumpla y viceversa.

```
printf( "Resultado: %i", !10==10 ); /* Resultado: 0 */
```

```
printf( "Resultado: %i", !(5<2) ); /* Resultado: 1 */
```

En los operadores && y || primero se evalúa la condición de la izquierda y si es necesario se evalúa la de la derecha. Por ejemplo:

```
(10>5 && 6==6)
```

Se evalúa $10 > 5 \rightarrow$ verdadera. A continuación se evalúa $6 == 6 \rightarrow$ verdadera. Resultado: verdadera.

```
(10<5 && 6==6)
```

Se evalúa la de la izquierda \rightarrow falso. Dado que el operador && requiere que ambas condiciones sean ciertas no es necesario evaluar la segunda ya que aunque sea cierta el resultado será falso. Es decir:

- En el caso del operador AND si la primera expresión es falsa (igual a 0) el resultado final va a ser falso así que la segunda expresión no se evalúa.
- En el caso del operador OR si la primera expresión es verdadera (diferente de 0) el resultado final va a ser verdadero así que la segunda expresión no se evalúa.

Por esta forma de funcionamiento se les llama operadores shortcircuit operators (u operadores cortocircuito).

Estos dos operadores son particularmente útiles cuando se debe evaluar (o no) una expresión dependiendo de la evaluación de una expresión anterior.

Por ejemplo supongamos que tenemos dos números enteros (a y b) y tenemos que verificar si el primero (a) es un múltiplo del segundo (b). Podemos hacer:

```
if ((a % b == 0))  
    printf("%d es divisible por %d", a, b);
```

Pero si b es cero tendremos un error de división por cero. Para evitarlo podemos usar la siguiente expresión:

```
if ((b != 0) && (a % b == 0))  
    /* b es múltiplo de a */
```



NOTA

el funcionamiento del *if* lo estudiaremos en un capítulo posterior. Por ahora es suficiente con saber que permite controlar el flujo de un programa dependiendo de la condición que le sigue.

Aquí el operador AND primero evalúa la expresión a su izquierda y solo si esta es verdadera (¿*b* es diferente de cero?) se evalúa la expresión a su derecha (¿el residuo de *a* entre *b* es cero?).

Ver el capítulo Sentencias, sección Notas sobre las condiciones para más información.

Introducción a los bits y bytes

Supongo que todo el mundo sabe lo que son los bytes y los bits, pero por si acaso allá va.

Los bits son la unidad de información más pequeña, digamos que son la base para almacenar la información. Son como los átomos a las moléculas. Los valores que puede tomar un bit son 0 ó 1. Si juntamos ocho bits tenemos un byte.

Un byte puede tomar 256 valores diferentes (de 0 a 255).

¿Cómo se consigue esto? Imaginemos nuestro flamante byte con sus ocho bits. Supongamos que los ocho bits valen cero. Ya tenemos el valor 0 en el byte. Ahora vamos a darle al último byte el valor 1.

Cambiando los 1 y 0 podemos conseguir los 256 valores:

00000000 → 0

00000001 → 1

00000010 → 2

00000011 → 3

...

11111110 → 254

11111111 → 255

Como vemos con ocho bits podemos tener 256 valores diferentes, que en byte corresponden a los valores entre 0 y 255.

En C en lugar de utilizarse el byte la unidad “básica” es el **unsigned char**. Aunque su numero de bits es usualmente ocho no tiene por qué ser así y puede ser mayor. Dependerá del compilador.

Para estar seguros del numero de bits por carácter lo mejor es verificar el valor de la macro `CHAR_BIT`, esta se define en el header `limits.h`.

Operadores de bits

Ya hemos visto que las variables unsigned char están compuestas de bits. Pues bien, con los operadores de bits podemos manipular las variables por dentro. Los diferentes operadores de bits son:

| OR (O)

& AND (Y)

^ XOR (O exclusivo)

~ Complemento a uno o negación

>> Desplazamiento a la derecha

<< Desplazamiento a la izquierda

Operador | (OR)

Toma dos valores y hace con ellos la operación OR. Vamos a ver un ejemplo:

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

int main()
{
    printf( "El resultado de la operación 235 | 143 es: %i\n", 235 |
143 );

    return EXIT_SUCCESS;
}
```

Se obtiene:

```
El resultado de la operación 235 | 143 es: 239
```

Veamos la operación a nivel de bits:

235 → 11101011

143 → 10001111 |

239 → 11101111

La operación OR funciona de la siguiente manera: Tomamos los bits de cada uno de los valores y los comparamos si alguno de los bits es 1, se obtiene un uno. Si ambos bits son cero el resultado es cero. Primero se compara los dos primeros (el primero de cada uno de los números, 1 y 1 → 1),

luego la segunda pareja (1 y 0 \rightarrow 1) y así sucesivamente.

Operador & (AND)

Este operador compara los bits también dos a dos. Si ambos son 1 el resultado es 1. Si no, el resultado es cero.

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

int main()
{
    printf( "El resultado de la operación 170 & 155 es: %i\n", 170 & 155 );

    return EXIT_SUCCESS;
}
```

Tenemos:

```
El resultado de la operación 170 & 155 es: 138
```

A nivel de bits:

170 \rightarrow 10101010

155 \rightarrow 10011011 &

138 \rightarrow 10001010

Operador ^ (XOR)

Compara los bits y los pone a unos si son distintos. Si son iguales el bit resultante es un cero.

235 \rightarrow 11101011

143 \rightarrow 10001111 ^

100 \rightarrow 01100100

Operador ~ (Complemento a uno)

Este operador acepta un sólo dato (operando) y pone a 0 los 1 y a 1 los 0, es decir los invierte. Se pone delante del operando.

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

int main()
{
    printf( "El resultado de la operación ~152 es: %i\n", ~152 & 0xFF );

    return EXIT_SUCCESS;
}
```

El resultado de la operación ~152 es: 103

Y a nivel de bits: name: value

152 → 10011000 ~

103 → 01100111

Operador >> (Desplazamiento a la derecha)

Este operador mueve cada bit a la derecha. El bit de la izquierda se pone a cero, el de la derecha se pierde. Si después de usar este operador realizamos la operación inversa no recuperamos el número original. El formato es:

variable o dato >> número de posiciones a desplazar

El *número de posiciones a desplazar* indica cuantas veces hay que mover los bits hacia la derecha. Ejemplo:

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

int main()
{
    printf("El resultado de 150U >> 2 es %u\n", 150U >> 2);

    return EXIT_SUCCESS;
}
```

Salida:

El resultado de la operación 150 >> 2 es: 37

Veamos la operación paso a paso. Esta operación equivale a hacer dos desplazamientos a la derecha:

150 → 10010110 Número original

75 → 01001011 Primer desplazamiento. Entra un cero por la izquierda. El bit de la derecha se pierde.

37 → 00100101 Segundo desplazamiento.



NOTA

Un desplazamiento a la derecha equivale a dividir por dos. Esto es muy interesante porque el desplazamiento es más rápido que la división.

Si queremos optimizar un programa esta es una buena idea. Sólo sirve para dividir entre dos. Si hacemos dos desplazamientos sería dividir por dos dos veces, no por tres.

Los "bits de relleno", los que se añaden por la izquierda, son siempre ceros cuando el número al que hacemos la operación es un entero sin signo.

En caso de que el desplazamiento se haga sobre un valor entero con signo hay un pequeño problema; los bits de relleno será uno o cero dependiendo del compilador. Por ejemplo:

-1 >> 4 /* No se puede predecir el resultado */

El rango válido para el desplazamiento va desde 0 hasta (sizeof(int) * CHAR_BIT) - 1.

Operador << (Desplazamiento a la izquierda)

Funciona igual que la anterior pero los bits se desplazan a la izquierda. Esta operación equivale a multiplicar por 2.

Operador sizeof

Este es un operador muy útil. Nos permite conocer el tamaño en bytes de una variable. De esta manera no tenemos que preocuparnos en recordar o calcular cuanto ocupa. Además el tamaño de una variable cambia de un compilador a otro, es la mejor forma de asegurarse. Se usa poniendo el nombre de la variable después de sizeof y separado de un espacio:

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

int main()
{
    int variable;

    printf( "Tamaño de la variable: %lu\n",
        (unsigned long) sizeof (variable) );

    return EXIT_SUCCESS;
}
```

NOTA: Como se puede apreciar, para mostrar el tamaño de la variable hemos usado %lu en lugar de %i. Esto es así porque *sizeof* devuelve un valor del tipo *size_t* y el estándar ISO-C90 sólo especifica que *size_t* debe ser un entero sin signo (puede ser un *int*, *short int* o *long int*). Para asegurarnos que mostramos correctamente su valor debemos usar %lu en lugar de %i.

También se puede usar con los especificadores de tipos de datos (char, int, float, double...) para averiguar su tamaño:

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

int main()
{

    printf( "Las variables tipo int ocupan: %lu\n",

        (unsigned long) sizeof(**int**) );

    return EXIT_SUCCESS;
}
```

Otros operadores

Existen además de los que hemos visto otros operadores. Sin embargo ya veremos en sucesivos capítulos lo que significa cada uno.

Orden de evaluación de Operadores

Debemos tener cuidado al usar operadores pues a veces podemos tener resultados no esperados si no tenemos en cuenta su orden de evaluación. Vamos a ver la lista de precedencias, cuanto más arriba se evalúa antes:

Precedencia

```
() [] -> .  
! ~ ++ -- (molde) * & sizeof (El * es el de puntero)  
* / % (El * de aquí es el de multiplicación)  
+ -  
<< >>  
< <= > >=  
== !=  
&  
^  
|  
&&  
||  
?:  
= += -= *= /=  
,
```

Por ejemplo imaginemos que tenemos la siguiente operación:

$10 * 2 + 5$

Si vamos a la tabla de precedencias vemos que el $*$ tiene un orden superior al $+$, por lo tanto primero se hace el producto $10*2=20$ y luego la suma $20+5=25$.

Veamos otra:

$10 * (2 + 5)$

Ahora con el paréntesis cambia el orden de evaluación. El que tiene mayor precedencia ahora es el paréntesis, se ejecuta primero. Como dentro del paréntesis sólo hay una suma se evalúa sin más, $2+5=7$. Ya solo queda la multiplicación $10*7=70$.

Otro caso:

$10 * (5 * 2 + 3)$

Como antes, el que mayor precedencia tiene es el paréntesis, se evalúa primero. Dentro del paréntesis tenemos producto y suma. Como sabemos ya se evalúa primero el producto, $5*2=10$. Seguimos en el paréntesis, nos queda la suma $10+3=13$. Hemos acabado con el paréntesis, ahora al resto de la expresión. Cogemos la multiplicación que queda:

$10*13=130$

Otro detalle que debemos cuidar son los operadores `y --`. Es mejor no usar los operadores `y` —mezclados con otros, pues puede ser confuso y a veces obtenemos resultados inesperados. Por ejemplo:

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

int main()
{
    int a;

    a = 5;

    printf( "a = %i\n", a++ );

    return EXIT_SUCCESS;
}
```

El resultado sería:

```
a = 5
```

Para evitar confusiones lo mejor sería separar la línea donde se usa el `++`:

```

#include <stdio.h>
#include <stdlib.h>

int main()
{

    int a;

    a = 5;
    a++;

    printf( "a = %i\n", a );

    return EXIT_SUCCESS;

}

```

Ejercicios

Ejercicio 1: En este programa hay un fallo muy gordo y muy habitual en programación. A ver si lo encuentras:

```

#include <stdio.h>
#include <stdlib.h>

int main()
{

    int a, c;

    a = 5;

    c += a +5;

    return EXIT_SUCCESS;

}

```

Solución:

Cuando calculamos el valor de *c* sumamos *a*+5 (=10) al valor de *c*. Pero resulta que *c* no tenía ningún valor indicado por nosotros. Estamos usando la variable *c* sin haberle dado valor. En algunos compiladores el resultado será inesperado. Este es un fallo bastante habitual, usar variables a las que no hemos dado ningún valor.

Ejercicio 2: ¿Cual será el resultado del siguiente programa?

```

#include <stdio.h>
#include <stdlib.h>

int main()
{
    int a, b, c;

    a = 5;
    b = ++a;
    c = ( a + 5 * 2 ) * ( b + 6 / 2 ) + ( a * 2 );

    printf( "%i, %i, %i", a, b, c );

    return EXIT_SUCCESS;
}

```

Solución:

El resultado es 156. En la primera a vale 5. Pero en la segunda se ejecuta $b = a = 5 = 6$. Tenemos $a = b = 6$.

Ejercicio 3: Escribir un programa que compruebe si un número es par o impar.

Solución:

```

#include <stdio.h>
#include <stdlib.h>

int main() \{
    int a;

    a = 124;

    if ( a % 2 == 0 ) {
        printf( "%d es par\n", a );
    }
    else {
        printf( "%d es impar\n", a );
    }

    printf( "\n" );

    return EXIT_SUCCESS;
}

```

Para comprobar si un número es par o impar podemos usar el operador %. Si al calcular el resto de dividir un número por 2 el resultado es cero eso indica que el número es par. Si el resto es distinto de cero el número es impar.

¿Dudas? ¿Errores?

Si tienes dudas sobre este capítulo plantéalas en el foro:

<https://elrincondelc.com/foros/viewforum.php?f=51>

Si has encontrado algún error o quieres sugerir cambios entra aquí:

<https://github.com/gorkau/Libro-Programacion-en-C/blob/master/capitulo5.adoc>

Capítulo 6. Introducir datos por teclado

Introducción

Algo muy usual en un programa es esperar que el usuario introduzca datos por el teclado. Para ello contamos con varias posibilidades: Usar las funciones de la biblioteca estándar, crear nuestras propias interrupciones de teclado o usar funciones de alguna biblioteca diferente (como por ejemplo Allegro).

Nosotros en este capítulo vamos a estudiar la primera opción (biblioteca estándar) mediante el uso de la función *scanf* y también la tercera opción (bibliotecas de terceros) mediante el uso de las funciones *getch* y *getche*. Estas ultimas solo funcionan con los compiladores que soporten la biblioteca *conio* de Borland. Pero antes veamos por encima las otras posibilidades.

Las funciones estándar están bien para un programa sencillito. Pero cuando queremos hacer juegos por ejemplo, no suelen ser suficiente. Demasiado lentas o no nos dan todas las posibilidades que buscamos, como comprobar si hay varias teclas pulsadas. Para solucionar esto tenemos dos posibilidades:

La más complicada es crear nuestras propias interrupciones de teclado. ¿Qué es una interrupción de teclado? Es un pequeño programa en memoria que se ejecuta continuamente y comprueba el estado del teclado. Podemos crear uno nuestro y hacer que el ordenador use el que hemos creado en vez del suyo. Este tema no lo vamos a tratar ahora, quizás en algún capítulo posterior.

Otra posibilidad más sencilla es usar una biblioteca que tenga funciones para controlar el teclado. Por ejemplo si usamos la biblioteca Allegro, ella misma hace todo el trabajo y nosotros no tenemos más que recoger sus frutos con un par de sencillas instrucciones. Esto soluciona mucho el trabajo y nos libra de tener que aprender cómo funcionan los aspectos más oscuros del control del teclado.

Vamos ahora con las funciones de la biblioteca estándar.

Scanf

El uso de *scanf* es muy similar al de *printf* con una diferencia, nos da la posibilidad de que el usuario introduzca datos en vez de mostrarlos. No nos permite mostrar texto en la pantalla, por eso si queremos mostrar un mensaje usamos un *printf* delante.

El formato de *scanf* es:

```
scanf( "%d", &var );
```

Donde:

- *%d* puede ser un modificador de los que ya habíamos visto en el *printf*. En la sección “[Modificadores](#)” hay una lista de modificadores que se pueden usar.
- *var* es la variable donde se va a almacenar el valor que teclee el usuario.

Un ejemplo:

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

int main()
{
    int num;

    printf( "Introduce un número: " );

    fflush(stdout);

    scanf( "%d", &num );

    printf( "Has tecleado el número %d\n", num );

    return EXIT_SUCCESS;
}
```

NOTA 1: En este ejemplo hemos usado el modificador `%d` en lugar de `%i`. Si usamos el modificador `%i` puede llevar a resultados inesperados debido a que `%i` acepta tanto números en base 10 como en base 8 (octal) y 16 (hexadecimal).

Por ejemplo si se introduce el número 0123 éste no se interpreta (como es de esperarse) como un número decimal sino como un número octal. El resultado del programa sería:

Introduce un numero: 0123

Has tecleado el numero 83



Quizás te estés preguntando qué es eso de `fflush(stdout)`. Cuando usamos la función `printf`, no escribimos directamente en la pantalla, sino en una memoria intermedia (lo que llaman un *buffer*). Cuando este *buffer* se llena o cuando metemos un carácter `\n` es cuando se envía el texto a la pantalla. En algunos compiladores es posible que el texto “Introduce una letra:” no se muestre sin el `fflush` (pruébalo en el tuyo).

Primero vamos a ver una nota de estética, para hacer los programas un poco más elegantes. Parece una tontería, pero los pequeños detalles hacen que un programa gane mucho. El `scanf` no mueve el cursor de su posición actual, así que en nuestro ejemplo queda:

Introduce un número _



La barra horizontal indica dónde está el cursor.

Esto es porque en el `printf` no hemos puesto al final el símbolo de salto de línea `\n`. Además hemos dejado un espacio al final de *Introduce un número*: para que así cuando tecleemos el número no salga pegado al mensaje. Si no hubiésemos dejado el espacio quedaría así al introducir el número 120 (es un ejemplo):

```
Introduce un número120
```

Bueno, esto es muy interesante pero vamos a dejarlo y vamos al grano. Veamos cómo funciona el `scanf`. Lo primero nos fijamos que hay una cadena entre comillas. Esta es similar a la de `printf`, nos sirve para indicarle al programa qué tipo de datos puede aceptar.

En el ejemplo, al usar `%d`, estábamos diciendo al programa que acepte únicamente números enteros decimales (en base 10).

Después de la coma tenemos la variable donde almacenamos el dato, en este caso `num`.

Fíjate que en el `scanf` la variable `num` lleva delante el símbolo `&`. Éste es muy importante; sirve para indicar al compilador cuál es la dirección (o posición en la memoria) de la variable. Por ahora no te preocupes por eso, ya volveremos más adelante sobre el tema.

Podemos pedir al usuario más de un dato a la vez en un sólo `scanf`. Para eso hay que poner un modificador por cada variable:

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

int main()
{
    int a, b, c;

    printf( "Introduce tres números: " );

    fflush(stdout);

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

    printf( "Has tecleado los números %d %d %d\n", a, b, c );

    return EXIT_SUCCESS;
}
```

De esta forma cuando el usuario ejecuta el programa debe introducir los tres datos separados por un espacio.

También podemos pedir en un mismo `scanf` variables de distinto tipo:

```

#include <stdio.h>
#include <stdlib.h>

int main()
{
    int a;
    float b;

    printf( "Introduce dos numeros: " );

    fflush(stdout);

    scanf( "%d %f", &a, &b );

    printf( "Has tecleado los numeros %d %f\n", a, b );

    return EXIT_SUCCESS;
}

```

A cada modificador (%d, %f) le debe corresponder una variable de su mismo tipo. Es decir, al poner un %d el compilador espera que su variable correspondiente sea de tipo int. Si ponemos %f espera una variable tipo float.

Modificadores

Hemos visto que cuando el dato introducido lo queremos almacenar en una variable tipo int usamos el modificador %d. Cada variable usa un modificador diferente.

int: entero. Puede ser decimal, octal o hexadecimal	%i
int: entero decimal	%d
unsigned int: entero decimal sin signo	%u
int: entero octal	%o
int: entero hexadecimal	%x
float	%f
double	%lf
char	%c
cadena de caracteres	%s

Ejercicios

Ejercicio 1: Busca el error en el siguiente programa:

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

int main()
{
    int numero;

    printf( "Introduce un numero: " );

    scanf( "%d", numero );

    printf( "\nHas introducido el número %d.\n", numero );

    return EXIT_SUCCESS;
}
```

Solución:

A la variable número le falta el & con lo que no estamos indicando al programa la dirección de la variable y no obtendremos el resultado deseado. Haz la prueba y verás que el mensaje "Has introducido el número X" no muestra el número que habías introducido.

Ejercicio 2: Escribe un programa que pida 3 números: un float, un double y un int y a continuación los muestre por pantalla.

Solución:

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

int main()
{
    float num1;
    double num2;
    int num3;

    printf( "Introduce 3 numeros: " );

    scanf( "%f %lf %i", &num1, &num2, &num3 );

    printf ( "\nNumeros introducidos: %f %f %i\n",
        num1, num2, num3 );

    return EXIT_SUCCESS;
}
```

¿Dudas? ¿Errores?

Si tienes dudas sobre este capítulo plantéalas en el foro:

<https://elrincondelc.com/foros/viewforum.php?f=52>

Si has encontrado algún error o quieres sugerir cambios entra aquí:

<https://github.com/gorkau/Libro-Programacion-en-C/blob/master/capitulo6.adoc>

Capítulo 7. Sentencias de control de flujo

Introducción

Hasta ahora los programas que hemos visto eran lineales. Comenzaban por la primera instrucción y acababan por la última, ejecutándose todas una sola vez. Lógico ¿no?. Pero resulta que muchas veces no es esto lo que queremos que ocurra. Lo que nos suele interesar es que dependiendo de los valores de los datos se ejecuten unas instrucciones y no otras. O también puede que queramos repetir unas instrucciones un número determinado de veces. Para esto están las sentencias de control de flujo.

Bucles

Los bucles nos ofrecen la solución cuando queremos repetir una tarea un número determinado de veces. Supongamos que queremos escribir 100 veces la palabra hola. Con lo que sabemos hasta ahora haríamos:

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

int main()
{
    printf( "Hola\n");
    printf( "Hola\n");
    printf( "Hola\n");
    printf( "Hola\n");
    printf( "Hola\n");
    ... (y así hasta 100 veces)

    return EXIT_SUCCESS;
}
```

¡Menuda locura! Y si queremos repetirlo más veces nos quedaría un programa de lo más largo.

Sin embargo usando un **bucle *for*** (ya lo veremos con calma en un rato) el programa quedaría:

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

int main()
{
    int i;

    for ( i=0 ; i<100 ; i++ )
    {
        printf( "Hola\n" );
    }

    return EXIT_SUCCESS;
}
```

Con lo que tenemos un programa más corto.

El bucle For

El formato del bucle for es el siguiente:

```
for( valores iniciales; condiciones; incrementos/cambios )
{
    /* conjunto de instrucciones a ejecutar en el bucle */
}
```

El bucle for tiene tres partes:

- **Valores iniciales:** En esta parte damos los valores iniciales a nuestro bucle, para empezar el bucle como más nos convenga.
- **Condiciones:** antes de comenzar cada ciclo del bucle comprobamos si se cumplen ciertas condiciones. Si se cumplen se ejecuta el conjunto de instrucciones del bucle.
- **Incrementos/cambios:** esta parte se ejecuta después del conjunto de instrucciones. Hacemos algún cambio en alguna de las variables que hemos usado en la parte de condiciones y pasamos al siguiente ciclo del *for*.

Vamos a verlo con el ejemplo anterior:

```
for ( i=0 ; i<100 ; i++ )
```

Las tres partes antes mencionadas serían:

- En este caso usamos la variable *i* para controlar nuestro bucle. Asignamos a esta variable el valor inicial 0. Esa es la parte de *valores iniciales*.
- Luego tenemos *i<100*. Esa es la parte *condiciones*. En este caso la condición es que *i* sea menor que 100, de modo que el bucle continuará mientras *i* sea menor que 100. Es decir, mientras se

cumpla la condición. Si se cumple se ejecuta el bloque de instrucciones del *for*. Si no se cumple la condición damos el bucle por terminado y el programa continúa.

- Luego tenemos la parte de *incrementos*, donde indicamos cuánto se incrementa la variable. En el ejemplo le sumamos uno a la variable *i*.

¿Cómo funciona el *for*?

- La primera vez que el programa llega al *for* se asigna un valor inicial a *i*.
- A continuación se comprueba la condición.
- Como *i* tiene el valor 0 se cumple.
- Se ejecuta el bloque de instrucciones que está justo después del *for*.
- El programa va a la sección de incrementos y ve que tiene que incrementar el valor de *i* en 1.
- Se comprueba la condición otra vez.
- ...
- Repetimos hasta que ya no se cumpla la condición (*i* coge el valor 100).

El *for* va delante del grupo de instrucciones a ejecutar. Si la condición es falsa desde el principio, esas instrucciones no se ejecutan ni una sola vez. Por ejemplo:

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

int main()
{
    char i;

    for (i=0; i<0; i++) {
        printf("Me siento tan invisible \n");
    }

    return EXIT_SUCCESS;
}
```

Este programa no va a mostrar nada porque el *printf* no llega a ejecutarse nunca.

Cuidado: No se debe poner un ";" justo después de la sentencia *for*. Si lo hacemos entonces sería un bucle vacío y las instrucciones siguientes sólo se ejecutarían una vez. Veámoslo con un ejemplo:


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

int main()
{
    int i;

    for ( i=0 ; i<100 ; i++ ); /* Cuidado con este punto y coma */
    {
        printf( "Hola\n" );
    }

    return EXIT_SUCCESS;
}
```

Este programa sólo escribirá en pantalla:

Hola

una sola vez. El bucle se ejecuta pero el ; hace que el bloque que viene a continuación no se considere parte del *for*.

También puede suceder que quieras ejecutar un cierto número de veces una sola instrucción (como sucede en nuestro ejemplo). Entonces no necesitas las llaves "{}":

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

int main()
{
    int i;

    for ( i=0 ; i<100 ; i++ ) printf( "Hola\n" );

    return EXIT_SUCCESS;
}
```

o también:

```
for ( i=0 ; i<100 ; i++ )
    printf( "Hola\n" );
```

Sin embargo, yo me he encontrado muchas veces que es mejor poner las llaves aunque sólo haya una instrucción; a veces al añadir una segunda instrucción más tarde se te olvidan las comillas y no te das cuenta. Parece una tontería, pero muchas veces, cuando programas, son estos los pequeños fallos los que te vuelven loco.

En otros lenguajes, como Basic, la sentencia for es muy rígida. En cambio en C es muy flexible. Se puede omitir cualquiera de las secciones (inicialización, condiciones o incrementos). También se pueden poner más de una variable a inicializar, más de una condición y más de un incremento. Por ejemplo, el siguiente programa sería perfectamente correcto:

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

int main()
{
    int i, j;

    for( i=0, j=5 ; i<10 ; i++, j=j+5 )
    {
        printf( "Hola " );
        printf( "Esta es la línea %i", i );
        printf( "j vale = %i\n", j );
    }

    return EXIT_SUCCESS;
}
```

Como vemos en el ejemplo tenemos más de una variable en la sección de inicialización y en la de incrementos. También podíamos haber puesto más de una condición. Los elementos de cada sección se separan por comas. Cada sección se separa por punto y coma.

Bucles infinitos

Entramos en un bucle infinito cuando nuestro for nunca termina. Esta es una situación a evitar y puede ocurrir si no tenemos cuidado. Con un *for* un bucle infinito puede ocurrir cuando:

1) No usamos la condición:

En caso de omitirse la condición el bucle se ejecuta continuamente sin detenerse. A este tipo de bucle se le conoce como *endless loop* o *bucle infinito*. Por ejemplo:

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

int main()
{
    int i, j;

    for( i=0; ; i++ ) {
        printf( "Este bucle no terminará nunca." );
    }

    return EXIT_SUCCESS;
}
```

Este ejemplo estará ejecutándose indefinidamente porque el bucle *for* no tiene una condición de finalización. Lo mismo ocurriría con, por ejemplo, este otro:

```
for(;;)
```

2) No usamos incrementos:

Si no hay nada que cambie en cada ciclo, el *for* no puede “avanzar”:

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

int main()
{
    int i;

    for (i=0; i<10; ) {
        printf("Soy un bucle infinito\n");
    }

    return EXIT_SUCCESS;
}
```

En este ejemplo *i* nunca será mayor que 10 puesto que su valor no cambia nunca.

3) La condición se cumple siempre:

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

int main()
{
    char i;

    for (i=0; i==i; i++) {
        printf("Soy un bucle infinito\n");
    }

    return EXIT_SUCCESS;
}
```

En este caso siempre se va a cumplir que `i==i` por lo tanto el bucle no terminará nunca.

While

El formato del bucle while es es siguiente:

```
while ( condición )
{
    /* bloque de instrucciones a ejecutar */
}
```

While quiere decir *mientras*. Aquí se ejecuta el bloque de instrucciones mientras se cumpla la condición impuesta en while.

Vamos a ver un ejemplo:

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

int main()
{
    int contador = 0;

    while ( contador<100 ) {
        contador++;
        printf( "Ya voy por el %i, pararé enseguida.\n", contador );
    }

    return EXIT_SUCCESS;
}
```

Este programa imprime en pantalla los valores del 1 al 100. Cuando `i=100` ya no se cumple la condición. Un detalle importante: si hubiésemos cambiado el orden de las instrucciones a ejecutar

tendríamos un resultado diferente:

```
printf( "Ya voy por el %i, pararé enseguida.\n", contador );
contador++;
```

En esta ocasión se imprimen los valores del 0 al 99. Cuidado con ésto, que a veces produce errores difíciles de encontrar.

Do While

El formato del bucle do-while es:

```
do
{
    /* instrucciones a ejecutar */
} while ( condición );
```

La diferencia entre *while* y *do-while* es que en este último, la condición va después del conjunto de instrucciones a ejecutar. De esta forma, esas instrucciones se ejecutan al menos una vez.

Su uso es similar al de *while*.

Sentencias de condición

Hasta aquí hemos visto cómo podemos repetir un conjunto de instrucciones las veces que deseemos. Pero ahora vamos a ver cómo podemos controlar totalmente el flujo de un programa. Dependiendo de los valores de alguna variable se tomarán unas acciones u otras. Empecemos con la sentencia *if*.

If

La palabra *if* significa *si* (condicional), pero supongo que esto ya lo sabías. Su formato es el siguiente:

```
if ( condición )
{
    /* instrucciones a ejecutar */
}
```

Cuando se cumple la condición entre paréntesis se ejecuta el bloque inmediatamente siguiente al *if* (bloque *instrucciones a ejecutar*).

En el siguiente ejemplo tenemos un programa que nos pide un número, si ese número es 10 se muestra un mensaje. Si no es 10 no se muestra ningún mensaje:

```

#include <stdio.h>
#include <stdlib.h>

int main()
{
    int num;

    printf( "Introduce un numero: " );
    fflush(stdout);
    scanf( "%i", &num );

    if (num==10) {
        printf( "El numero es igual a 10.\n" );
    }

    return EXIT_SUCCESS;
}

```

Como siempre, la condición es falsa si es igual a cero. Si es distinta de cero será verdadera.

If - Else

El formato es el siguiente:

```

if ( condición )
{
    /* bloque que se ejecuta si se cumple la condición */
}
else
{
    /* bloque que se ejecuta si no se cumple la condición */
}

```

En el *if* del apartado anterior si no se cumplía la condición no se ejecutaba el bloque siguiente y el programa seguía su curso normal. Con el *if-else* tenemos un bloque adicional que sólo se ejecuta si no se cumple la condición. Veamos un ejemplo:

```

#include <stdio.h>
#include <stdlib.h>

int main()
{
    int a;

    printf( "Introduce un numero: " );
    fflush(stdout);
    scanf( "%i", &a );

    if ( a==8 ) {
        printf ( "El numero introducido era un ocho.\n" );
    }
    else {
        printf ( "Pero si no has escrito un ocho!!!\n" );
    }

    return EXIT_SUCCESS;
}

```

Al ejecutar el programa si introducimos un 8 se ejecuta el bloque siguiente al *if* y se muestra el mensaje:

El numero introducido era un ocho.

Si escribimos cualquier otro número se ejecuta el bloque siguiente al *else* mostrándose el mensaje:

Pero si no has escrito un ocho!!!

If else if

Se pueden poner if else anidados si se desea:

```

#include <stdio.h>
#include <stdlib.h>

int main()
{
    int a;

    printf( "Introduce un numero: " );
    fflush(stdout);
    scanf( "%i", &a );

    if ( a<10 ) {
        printf ( "El numero introducido era menor de 10.\n" );
    }
    else if ( a>=10 && a<=100 ) {
        printf ( "El numero esta entre 10 y 100\n" );
    }
    else if ( a>100 ) {
        printf( "El numero es mayor que 100\n" );
    }

    printf( "Fin del programa.\n" );

    return EXIT_SUCCESS;
}

```

NOTA: El símbolo && de la condición del segundo if es un AND (Y). De esta forma la condición queda: Si a es mayor que 10 Y a es menor que 100. Consulta la sección [Notas sobre las condiciones](#) para saber más.

Podemos poner todos los if else que queramos. Si la condición del primer if es verdadera se muestra el mensaje "El número introducido era menor de 10" y se saltan todos los if-else siguientes (se muestra el mensaje "Fin del programa"). Si la condición es falsa se ejecuta el siguiente else-if y se comprueba si a está entre 10 y 100. Si es cierto se muestra "El número está entre 10 y 100". Si no es cierto se evalúa el último else-if.

? (el otro if-else)

El uso de la interrogación es una forma de condensar un if-else. Su formato es el siguiente:

```
( condicion ) ? ( instrucción 1 ) : ( instrucción 2 )
```

Si se cumple la condición se ejecuta la *instrucción 1* y si no se ejecuta la *instrucción 2*. Veamos un ejemplo con el if-else y luego lo reescribimos con "?":


```

#include <stdio.h>
#include <stdlib.h>

int main()
{
    int a;
    int b;

    printf( "Introduce un número " );
    fflush(stdout);
    scanf( "%i", &a );

    if ( a<10 ) {
        b = 1;
    }

    else {
        b = 4;
    }
    printf ( "La variable 'b' toma el valor: %i\n", b );

    return EXIT_SUCCESS;
}

```

Si el valor que tecleamos al ejecutar es menor que 10 entonces la variable b toma el valor 1. En cambio si tecleamos un número mayor o igual que 10 b será igual a 4. Ahora vamos a reescribir el programa usando ?:

```

#include <stdio.h>
#include <stdlib.h>

int main()
{
    int a;
    int b;

    printf( "Introduce un número " );
    fflush(stdout);
    scanf( "%i", &a );

    b = ( a<10 ) ? 1 : 4 ;

    printf ( "La variable 'b' toma el valor: %i\n", b );

    return EXIT_SUCCESS;
}

```

¿Qué es lo que sucede ahora? Se evalúa la condición a<10. Si es verdadera (a menor que 10) se

ejecuta la instrucción 1, es decir, que b toma el valor 1. Si es falsa se ejecuta la instrucción 2, es decir, b toma el valor 4.

Switch

El formato de la sentencia *switch* es:

```
switch ( valor ) { case opción 1: /* código a ejecutar si el valor es el de "opción 1" / break; case opción 1: / código a ejecutar si el valor es el de "opción 1" / break; default: / Código a ejecutar si el valor no es ninguno de los anteriores */ break; }
```

Vamos a ver cómo funciona. La sentencia *switch* sirve para elegir una opción entre varias disponibles. Dependiendo del valor se cumplirá un caso u otro.

Por ejemplo si la opción elegida fuera la dos se ejecutaría el código que está justo después de:

```
case _opción 2_:
```

hasta el primer *break* que encontremos.

Vamos a ver un ejemplo de múltiples casos con if-else y luego con switch:

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

int main()
{
    int num;

    printf( "Introduce un número: " );
    fflush(stdout);
    scanf( "%d", &num );

    if ( num==1 ) {
        printf ( "Es un 1\n" );
    }
    else if ( num==2 ) {
        printf ( "Es un 2\n" );
    }
    else if ( num==3 ) {
        printf ( "Es un 3\n" );
    }
    else {
        printf ( "No era ni 1, ni 2, ni 3\n" );
    }

    return EXIT_SUCCESS;
}
```

Ahora con *switch*:

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

int main()
{
    int num;

    printf( "Introduce un número: " );
    fflush(stdout);
    scanf( "%d", &num );

    switch( num ) {

        case 1:
            printf( "Es un 1\n" );
            break;

        case 2:
            printf( "Es un 2\n" );
            break;

        case 3:
            printf( "Es un 3\n" );
            break;

        default:
            printf( "No es ni 1, ni 2, ni 3\n" );
    }

    return EXIT_SUCCESS;
}
```

Como vemos el código con *switch* es más cómodo de leer.

Vamos a ver qué pasa si nos olvidamos algún *break*:

```

#include <stdio.h>
#include <stdlib.h>

int main()
{
    int num;

    printf( "Introduce un número: " );
    fflush(stdout);
    scanf( "%d", &num );

    switch( num ) {

        case 1:
            printf( "Es un 1\n" );
            /* Nos olvidamos el break que debería haber aquí */

        case 2:
            printf( "Es un 2\n" );
            break;

        default:

            printf( "No es ni 1, ni 2, ni 3\n" );
    }

    return EXIT_SUCCESS;
}

```

Si al ejecutar el programa escribimos un 2 tenemos el mensaje “*Es un dos*”. Todo correcto. Pero si escribimos un 1 lo que nos sale en pantalla es:

```

Es un 1
Es un 2

```

¿Por qué? Pues porque cada caso empieza con un *case* y acaba donde hay un *break*. Si no ponemos *break* aunque haya otro *case* más adelante el programa sigue hacia adelante. Por eso se ejecuta el código del *case 1* y del *case 2*.

Puede parecer una desventaja pero a veces es conveniente. Por ejemplo cuando dos *case* deben tener el mismo código. Si no tuviéramos esta posibilidad tendríamos que escribir dos veces el mismo código.



Vale, vale, también podríamos usar funciones, pero si el código es corto puede ser más conveniente no usar funciones. Ya hablaremos de eso más tarde.

Sin embargo *switch* tiene algunas limitaciones. Por ejemplo no podemos usar condiciones en los *case*. El ejemplo que hemos visto en el apartado *if-else-if* no podríamos hacerlo con *switch*.

Sentencias de salto: Goto

La sentencia **goto** (ir a) nos permite hacer un salto a la parte del programa que deseemos. En el programa podemos poner etiquetas, estas etiquetas no se ejecutan. Es como poner un nombre a una parte del programa. Estas etiquetas son las que nos sirven para indicar a la sentencia *goto* dónde tiene que saltar.

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

int main()
{
    printf( "Línea 1\n" );

    goto linea3; /* Le decimos al goto que busque la etiqueta _linea3_ */

    printf( "Línea 2\n" );

    linea3:/* Esta es la etiqueta */

    printf( "Línea 3\n" );

    return EXIT_SUCCESS;
}
```

Resultado:

```
Línea 1
Línea 3
```

Como vemos no se ejecuta el printf de Línea 2 porque nos lo hemos saltado con el goto.

El goto sólo se puede usar dentro de funciones, y no se puede saltar desde una función a otra (Las funciones las estudiamos en el siguiente capítulo).

Un apunte adicional del goto: Cuando yo comencé a programar siempre oía que no era correcto usar el goto, que era una mala costumbre de programación. Decían que hacía los programas ilegibles, difíciles de entender. Ahora en cambio hay quien dice que no está tan mal. En Internet se pueden encontrar páginas que discuten sobre el tema. Pero como conclusión yo diría que cada uno la use si quiere, el caso es no abusar de ella y tener cuidado.

Notas sobre las condiciones

Las condiciones de las sentencias (por ejemplo del *if*) se evalúan al ejecutarse. De esta evaluación obtenemos un número. **Las condiciones son falsas si este número es igual a cero.** Son **verdaderas si es distinto de cero** (los números negativos son verdaderos).

Ahí van unos ejemplos:

```
a = 2;  
b = 3;  
if ( a == b ) ...
```

Aquí `a==b` sería igual a 0, por lo tanto *falso*.

```
if ( 0 ) ...
```

Como la condición es igual a cero, es falsa.

```
if ( 1 ) ...
```

Como la condición es distinta de cero, es verdadera.

```
if ( -100 ) ...
```

Como la condición es distinta de cero, es verdadera.

Supongamos que queremos mostrar un mensaje si una variable es distinta de cero:

```
if ( a!=0 ) printf( "Hola\n" );
```

Esto sería redundante, bastaría con poner:

```
if ( a ) printf( "Hola\n" );
```

Esto sólo vale si queremos comprobar que es distinto de cero. Si queremos comprobar que es igual a 3:

```
if ( a == 3 ) printf( "Es tres\n" );
```

Como vemos las condiciones no sólo están limitadas a comparaciones, se puede poner cualquier expresión que devuelva un valor. Dependiendo de si este valor es cero o no, la condición será falsa o verdadera.

También podemos evaluar varias condiciones en una sola usando `&&` (AND), `||` (OR).

Ejemplos de `&&` (AND)

```
if ( a==3 && b==2 ) printf( "Hola\n" );/* Se cumple si a es 3 Y b es dos */
```

```
if ( a>10 && a<100 ) printf( "Hola\n" );/* Se cumple si a es mayor que 10 Y menor que 100 */
```

```
if ( a==10 && b<300 ) printf( "Hola\n" );/* Se cumple si a es igual a 10 Y b es menor que 300 */
```

Ejemplos de || (OR):

```
if ( a<100 || b>200 ) printf( "Hola\n" ); /* Se cumple si a menor que 100 ó b mayor que 200 */
```

```
if ( a<10 || a>100 ) printf( "Hola\n" ); /* Se cumple si a menor que 10 ó a mayor que 100 */
```

Se pueden poner más de dos condiciones:

```
if ( a>10 && a<100 && b>200 && b<500 )/* Se deben cumplir las cuatro condiciones */
```

Esto se cumple si a es mayor que 10 y menor que 100 y b es mayor que 200 y menor que 500.

También se pueden agrupar mediante paréntesis varias condiciones:

```
if ( ( a>10 && a<100 ) || ( b>200 && b<500 ) )
```

Esta condición se leería como sigue:

- si a es mayor que 10 y menor que 100 o
- si b es mayor que 200 y menor que 500

Es decir que si se cumple el primer paréntesis o si se cumple el segundo la condición es cierta.

Ejercicios

Ejercicio 1: ¿Cuántas veces nos pide el siguiente programa un número y por qué?

```

#include <stdio.h>
#include <stdlib.h>

int main()
{
    int i;
    int numero, suma = 0;

    for ( i=0; i<4; i++ ); {

        printf( "\nIntroduce un número: " );
        scanf( "%d", &numero );
        suma += numero;
    }

    printf ( "\nTotal: %d\n", suma );

    return EXIT_SUCCESS;
}

```

Solución: El programa pedirá un número una única vez puesto que al final de la sentencia *for* hay un punto y coma. Como sabemos, el bucle *for* hace que se ejecuten las veces necesarias la sentencia siguiente (o el siguiente bloque entre `{}`).

Para que el programa funcione correctamente habría que eliminar el punto y coma.

Ejercicio 2: Una vez eliminado el punto y coma ¿cuántas veces nos pide el programa anterior un número?

Solución: Se ejecuta cuatro veces. Desde `i=0` mientras la segunda condición sea verdadera, es decir, desde `i=0` hasta `i=3`. Cuando `i` vale 4 la condición del *for* no es cierta y no se ejecuta más su código.

Ejercicio 3: Escribe un programa que muestre en pantalla lo siguiente:

```

*
**
***
****
*****

```

Solución:


```

#include <stdio.h>
#include <stdlib.h>

int main()
{
    int i, j;

    for( i=0; i<6; i++ ) {
        for( j=0; j<i; j++ )
            printf( "*" );
        printf( "\n" );
    }

    return EXIT_SUCCESS;
}

```

Ejercicio 4: Escribe un programa que pida un número a un usuario hasta que el usuario introduzca "-1". Usar un bucle while o do while.

```

Introduce un número: 4
Introduce un número: 2
Introduce un número: 7
Introduce un número: -1
Fin de programa

```

Solución:

```

#include <stdio.h>
#include <stdlib.h>

int main()
{
    int numero;

    do {
        printf("Introduce un numero: ");
        fflush(stdin);
        scanf( "%d", & numero );
    } while ( numero != -1 );

    printf( "Fin de programa\n" );

    return EXIT_SUCCESS;
}

```

Capítulo 8. Introducción a las funciones

Introducción

Vamos a dar un paso más en la complejidad de nuestros programas. Vamos a empezar a usar funciones creadas por nosotros. Las funciones son de una gran utilidad en los programas. Nos ayudan a que sean más legibles y más cortos. Con ellos estructuramos mejor los programas.

Una función sirve para realizar tareas concretas y simplificar el programa. Nos sirve para evitar tener que escribir el mismo código varias veces.

Los bucles que hemos estudiado son útiles cuando hay que repetir una parte del código varias veces. Las funciones son útiles cuando tenemos que repetir el mismo código en diferentes partes del programa.

Ya hemos visto en el curso algunas funciones como `printf` y `scanf`. Algunas de éstas están definidas en una biblioteca (la biblioteca estándar de C) que el compilador carga automáticamente en cada programa.

Sin embargo nosotros también podemos definir nuestras propias funciones. Pocas veces se ve un programa un poco complejo que no use funciones. Una de ellas, que usamos siempre, es la función `main`.

Definición de una función

Una función tiene el siguiente formato:

```
tipo_de_dato nombre_de_la_función( argumentos )
{
    definición de variables;

    cuerpo de la función;

    return valor;
}
```

A continuación vamos a ver en detalle cada una de las partes de una función:

El nombre de la función

El nombre de la función se usa para llamarla dentro del programa. Nombres de funciones que ya conocemos serían *printf* o *scanf*.

El nombre de una función debe cumplir los siguientes requisitos:

- Sólo puede contener letras, números y el símbolo `_`.
- No se pueden usar tildes ni espacios.

- El nombre de una función debe empezar por una letra, nunca por un número.
- No podemos usar *palabras reservadas*. Las palabras reservadas son aquellas que se usan en C, por ejemplo *if*, *for*, *while*. Un nombre de función puede contener una palabra reservada: por ejemplo *while* no se puede usar como nombre de función, pero *while_1* sí.

Tipo de dato

Cuando una función se ejecuta y termina puede devolver un valor. Este valor puede ser cualquiera de los tipos de variables que hemos visto en el capítulo de Tipos de datos (int, char, float, double). También puede ser un tipo de dato definido por nosotros (esto lo veremos más tarde). El valor que devuelve la función suele ser el resultado de las operaciones que se realizan en la función, o si han tenido éxito o no.

El valor devuelto debe ser del tipo indicado en *tipo_de_dato*.

También podemos usar el tipo **void**. Este nos permite indicar que la función no devuelve ningún valor. Cuando este sea el caso la palabra reservada *return* se utiliza sola, sin ningún valor a continuación de ésta. O también podemos emitir poner *return*.

En este ejemplo usamos una (absurda) función que devuelve un número entero:

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

int dame_un_diez()
{
    return 10;
}

int main()
{
    printf( "%d\n", dame_un_diez() );

    return EXIT_SUCCESS;
}
```

El tipo de dato que devuelve es *int* y lo devuelve a través del *return*.

Definición de variables

Dentro de la función podemos definir variables que sólo tendrán validez dentro de la propia función. Si declaramos una variable en una función no podemos usarla en otra. A estas variables que solo existen dentro de una función las llamamos **variables locales**.

Ver “[Vida de una variable](#)” para más información sobre el tema.

Otro (absurdo) ejemplo de una función que usa una *variable local*:

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

int dame_un_diez()
{
    int diez = 10;

    return diez;
}

int main()
{
    printf( "%d\n", dame_un_diez() );

    return EXIT_SUCCESS;
}
```

En este ejemplo la variable *diez* no está accesible desde *main*.

Cuerpo de la función

Aquí es donde va el código de la función.

Argumentos o parámetros

Antes hemos visto que una variable definida en una función no se puede usar dentro de otra. Entonces ¿cómo le pasamos valores a una función para que trabaje con ellos? Para esto es para lo que se usan los argumentos.

Estos son variables que se pasan como datos a una función. Una función puede tener un argumento, muchos o ninguno. Cuando tiene más de uno éstos deben ir separados por una coma. Cada variable debe ir con su tipo de variable.

Un ejemplo de una función con parámetros sería:

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

int suma(int numero1, int numero2)
{
    return numero1 + numero2;
}

int main()
{
    printf( "%d\n", suma(10, 5) );

    return EXIT_SUCCESS;
}
```

Dónde se definen las funciones

Las funciones deben definirse siempre antes de donde se usan. Lo habitual en un programa es:

*Sección *	*Descripción *
Includes	Aquí se indican qué ficheros externos se usan.
Definiciones de constantes	Aquí se definen las constantes que se usan en el programa.
Definición de variables	Aquí se definen las variables globales (las que se pueden usar en TODAS las funciones).
Prototipos de funciones	Aquí es donde se definen las cabeceras de las funciones.
Definición de las funciones	Aquí se “desarrollan” las funciones. Pueden colocarse en cualquier orden. Por costumbre, yo suelo poner main en primer lugar.

Esta es una forma muy habitual de estructurar un programa. Sin embargo esto no es algo rígido, no tiene por qué hacerse así, pero es recomendable.

Los prototipos de las funciones consisten en definir solo las cabeceras de las funciones, sin escribir su código. Esto nos permite luego poner las funciones en cualquier orden. El estándar ANSI C dice que no son obligatorios los prototipos de las funciones pero es recomendable usarlos.

Ejemplos:

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

```
int compara( int a, int b ); /* Definimos la cabecera de la función */
```

```
int main()
```

```
{
```

```
int num1, num2;
```

```
int resultado;
```

```
printf( "Introduzca dos numeros: " );
```

```
scanf( "%d %d", &num1, &num2 );
```

```
resultado = compara( num1, num2 );
```

```
printf( "El mayor de los dos es %d\n", resultado );
```

```
return 0;
```

```
}
```

int compara(int a, int b) /* Ahora podemos poner el cuerpo de la función donde queramos. Incluso después de donde la llamamos (main) */

```
{
```

```
int mayor;
```

```
if ( a>b )
```

```
mayor = a;
```

```
else
```

```
mayor = b;
```

```
return mayor;
```

```
}
```

NOTA: Por simplicidad este ejemplo no tiene en cuenta el caso de que los números sean iguales. Se deja al alumno como ejercicio modificar el programa para que tenga en cuenta la posibilidad de que los números sean iguales.

Cuando se define la cabecera de la función sin su cuerpo (o código) debemos poner un ; al final. Cuando definamos el cuerpo más tarde no debemos poner el ;, se hace como una función normal.

La definición debe ser igual cuando definimos sólo la cabecera y cuando definimos el cuerpo. Mismo nombre, mismo número y tipo de parámetros y mismo tipo de valor devuelto.

Las funciones deben definirse antes de ser llamadas. En los ejemplos a continuación se llama a la función desde main, así que tenemos que definirlas antes que main. Lo habitual es definir primero la “cabecera” o prototipos de la función, que no es más que la definición de la función si su “cuerpo” y desarrollar después la función completa.

Ejemplo 1. Función sin argumentos que no devuelve nada:

Este programa llama a la función prepara pantalla que borra la pantalla y muestra el mensaje "la pantalla está limpia". Por supuesto es de nula utilidad pero nos sirve para empezar.

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

```
void mostrar_mensaje(); /* Prototipo de la función */
```

```
int main()
```

```
{
```

```
printf( "Esta es la función main\n" );
```

```
    prepara_pantalla(); /* Llamamos a la función */
```

```
    return 0;
```

```
}
```

```
/* Desarrollo de la función */
```

```
void mostrar_mensaje() /* No se debe poner punto y coma aquí */
```

```
{
```

```
    printf( "Esta es la funcion mostrar_mensaje\n" );
```

```
return; /* No hace falta devolver ningún valor,
```

```
este return no es necesario */
```

```
}
```

Ejemplo 2. Función con argumentos, no devuelve ningún valor:

En este ejemplo la función *compara* toma dos números, los compara y nos dice cual es mayor.

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

```
void compara( int a, int b );
```

```
int main()
```

```
{
```

```
int num1, num2;
```

```
printf( "Introduzca dos numeros: " );
```

```
fflush(stdout);
```

```
scanf( "%d %d", &num1, &num2 );
```

```
/* Llamamos a la función con sus dos argumentos */
```

```
compara( num1, num2 );
```

```
return 0;
```

```
}
```

```
void compara( int a, int b ) /* Pasamos los parámetros a y b a la función */
```

```
{
```

```
if ( a>b )
```



```
printf( "%d es mayor que %d\n" , a, b );
```

```
else if ( a<b )
```

```
printf( "%d es mayor que %d\n", b, a );
```

```
else +  
printf("%d es igual que %d\n", a, b);
```

```
}
```

Ejemplo 3. Función con argumentos que devuelve un valor.

Este ejemplo es como el anterior pero devuelve como resultado el mayor de los dos números.

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

```
int compara( int a, int b );
```

```
int main()
```

```
{
```

```
int num1, num2;
```

```
int resultado;
```

```
printf( "Introduzca dos numeros: " );
```

```
fflush(stdout);
```

```
scanf( "%d %d", &num1, &num2 );
```

```
/* Recogemos el valor que devuelve la función en la variable  
*resultado* */
```

```
resultado = compara( num1, num2 );
```

```
printf( "El mayor de los dos es %d\n", resultado );
```

```
return 0;
```

```
}
```

/* Metemos los parámetros a y b a la función */

```
int compara( int a, int b )
```

```
{
```

```
/* Esta función define su propia variable,
```

```
esta variable sólo se puede usar aquí */
```

```
int mayor;
```

```
if ( a>b )
```

```
    mayor = a;
```

```
else
```

```
    mayor = b;
```

```
return mayor;
```

```
}
```

En este ejemplo podíamos haber hecho también:

```
printf( "El mayor de los dos es %i\n", compara( num1, num2 ) );
```

De esta forma nos ahorramos tener que definir la variable *resultado*.

Vida de una variable

Cuando definimos una variable dentro de una función, esa variable sólo es válida dentro de la función. Este tipo de variables se denominan **variables locales**. Si definimos una variable dentro de main sólo podremos usarla dentro de main, será por tanto una variable local de la función main.

Si por el contrario la definimos fuera de las funciones se trataría de una **variable global** y se podría usar en cualquier función.

Podemos crear una variable global y en una función una variable local con el mismo nombre. Dentro de la función estaremos trabajando con la variable local, no con la global. Esto no da errores pero puede crear confusión al programar y al analizar el código. No es nada recomendable seguir esta práctica.

Por norma general **es aconsejable usar siempre variables locales** frente a las globales ya que será más sencilla la localización de errores y ayuda a la reutilización de código (podemos copiar/pegar las funciones a otro programa). Por sencillez, en muchos ejemplos usaremos variables globales, pero el alumno debería acostumbrarse a usar variables locales.

Ejercicios

Ejercicio 1: Descubre los errores:

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

```
int main()
```

```
{
```

```
int num1, num2;
```

```
int resultado,
```

```
printf( "Introduzca dos números: " );
```

```
fflush(stdout);
```

```
scanf( "%d %d", &num1, &num2 );
```

```
resultado = compara( num1, num2 );
```

```
printf( "El mayor de los dos es %d\n", resultado );
```

```
return 0;
```

```
}
```

```
int compara( int a, int b );
```

```
{
```

```
int mayor;
```

```
if ( a>b ) mayor = a;
```

```
else mayor = b;
```

```
return mayor;
```

```
}
```

Solución:

- Hay una coma después de *int resultado* en vez de un punto y coma.
- Llamamos a la función *compara* dentro de *main* antes de definirla. Si hubiésemos puesto el prototipo de la función al principio del código no hubiera sido problema.
- Cuando definimos la función *compara* hemos puesto un punto y coma al final, eso es un error. El código que hay justo detrás no pertenece a ninguna función.

Ejercicio 2: Busca los errores.

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

```
int resultado( int parametro )
```

```
int main()
```

```
{
```

```
int a, b;
```

```
a = 2; b = 3;
```

```
printf( "%i", resultado( a );
```

```
return 0;
```

```
}
```

```
char resultado( int parametro )
```

```
{
```

```
return parametro + b;
```

```
}
```

Solución:

- Hemos definido el prototipo de *resultado* sin punto y coma.
- Cuando definimos el cuerpo de *resultado* en su cabecera hemos puesto *char*, que no coincide con el prototipo.
- En la función *resultado* estamos usando la variable *b* que está definida sólo en main. No es una variable global y por lo tanto es como si no existiera para *resultado*.
- En printf nos hemos dejado un paréntesis al final.

Capítulo 9. Punteros

Introducción

Este capítulo puede resultar problemático a aquellos que no han visto nunca lo que es un puntero. Por lo tanto si tienes alguna duda o te parece que alguna parte está poco clara ponte en contacto conmigo.

¡¡¡Punteros!!! uff. Este es uno de los temas que más suele costar a la gente al aprender C. Los punteros son una de las más potentes características de C, pero a la vez uno de sus mayores peligros. Si no se manejan con cuidado pueden ser una fuente ilimitada de errores. Un error usando un puntero puede bloquear el sistema y a veces puede ser difícil detectarlo.

Otros lenguajes no nos dejan usar punteros para evitar estos problemas, pero a la vez nos quitan parte del control que tenemos en C.

A pesar de todo esto no hay que tenerles miedo. Casi todos los programas C usan punteros. Si aprendemos a usarlos bien no tendremos más que algún problema esporádico. Así que atención, valor y al toro.

La memoria del ordenador

Si tienes bien claro lo que es la memoria del ordenador puedes saltarte esta sección. Pero si confundes la memoria con el disco duro o no tienes claro lo que es no te la pierdas.

A lo largo de mi experiencia con ordenadores me he encontrado con mucha gente que no tiene claro cómo funciona un ordenador. Cuando hablamos de memoria nos estamos refiriendo a la memoria RAM del ordenador. Son unas *pastillas* que se conectan a la placa base y nada tienen que ver con el disco duro. El disco duro guarda los datos permanentemente (hasta que se rompe) y la información se almacena como ficheros. Nosotros podemos decirle al ordenador cuándo grabar, borrar, abrir un documento, etc. La memoria RAM en cambio, se borra al apagar el ordenador. La memoria RAM la usan los programas sin que el usuario de éstos se de cuenta.

Hay otras memorias en el ordenador aparte de la mencionada. Por ejemplo la memoria de vídeo (que está en la tarjeta gráfica), las memorias caché (del procesador, de la placa...).

Direcciones de variables

Vamos a ir como siempre por partes. Primero vamos a ver qué pasa cuando declaramos una variable.

Al declarar una variable estamos diciendo al ordenador que nos reserve una parte de la memoria para almacenarla. Cada vez que ejecutemos el programa la variable se almacenará en un sitio diferente. Eso no lo podemos controlar, depende de la memoria disponible y otros factores "misteriosos". Puede que se almacene en el mismo sitio, pero es mejor no fiarse.

Dependiendo del tipo de variable que declaremos el ordenador nos reservará más o menos memoria. Como vimos en el capítulo de tipos de datos cada tipo de variable ocupa más o menos

bytes. Por ejemplo si declaramos un *char*, el ordenador nos reserva 1 byte (usualmente 8 bits). Una variable de tipo *int* ocupará más espacio (depende del compilador y el sistema en el que trabajemos).

Cuando finaliza el programa todo el espacio reservado para las variables queda libre para ser usado por otros programas.



Si bien usualmente un carácter (*char*) esta constituido por ocho bits esto no lo garantiza el estándar. El numero exacto esta dado por la macro `CHAR_BIT`).

Saber cuánto ocupa un carácter en tu sistema es importante. Para saberlo puedes usar este sencillo programa:

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

int main()
{
    printf("Bits que ocupa un carácter: %d", CHAR_BIT);

    return EXIT_SUCCESS;
}
```

Existe una forma de saber qué direcciones nos ha reservado el ordenador para nuestras variables. Se trata de usar el operador `&` (operador de dirección). Ya lo habíamos visto en el *scanf* pero no sabíamos bien para qué era. Vamos a ver un ejemplo: Declaramos la variable *a* y obtenemos su valor y dirección.

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

int main()
{
    char a;

    a = 10;

    printf( "La variable 'a' se almacena en la posición de memoria %p ", (void *) &a );
    printf( "y su valor es %d\n.", a );

    return EXIT_SUCCESS;
}
```



Para mostrar la dirección de la variable usamos `%p` (en lugar de `%d`). Éste *modificador de formato* sirve para escribir direcciones de memoria. El formato que se usa para mostrar las direcciones de memoria es el hexadecimal.

El resultado sería algo como:

La variable `a` se almacena en la posición de memoria `0x7fff13be7627` y su valor es `10`.

Si ejecutamos el programa varias veces seguidas veremos algo curioso:

- El valor de `a` siempre es el mismo.
- Pero la posición de la memoria donde se almacena la variable cambia cada vez.

La segunda vez pude mostrar algo como ésto:

La variable `a` se almacena en la posición de memoria `0x7fff8e74c037` y su valor es `10`.

Y la tercera:

La variable `a` se almacena en la posición de memoria `0x7ffc79a200a7` y su valor es `10`.

Como podemos ver, cada vez que ejecutamos el programa la dirección de la memoria donde se almacena `a` es diferente. Pero una vez el sistema nos ha reservado esa dirección ya no cambia durante la ejecución del programa.

El valor de `a` lo podemos controlar nosotros. El valor de la memoria que nos asigna el sistema podemos conocerlo pero no controlarlo.

Es importante tener clara la diferencia entre el valor de `a` y el valor de la memoria donde se almacena `a`.

El nombre de la variable es equivalente a poner un nombre a una zona de la memoria. Cuando en el programa escribimos `a`, en realidad estamos diciendo, "el valor que está almacenado en la dirección de memoria a la que llamamos `a`".

Qué son los punteros

Ahora ya estamos en condiciones de ver lo que es un puntero. Un puntero es una variable un tanto especial. Con un puntero podemos almacenar direcciones de memoria. En un puntero podemos tener guardada la dirección de una variable.

Vamos a ver si captamos bien el concepto de puntero y la diferencia entre éstos y las variables *normales*.

00001	
00002	
00003	
00004	
00005	
00006	

En el dibujo anterior tenemos una representación de lo que sería la memoria del ordenador. Cada casilla representa un byte de la memoria. Y cada número es su dirección de memoria. La primera casilla es la posición 00001 de la memoria. La segunda casilla la posición 00002 y así sucesivamente.

Supongamos que ahora declaramos una variable *char*:

```
char numero = 43.
```

El ordenador nos guardaría por ejemplo la posición 00003 para esta variable. Esta posición de la memoria queda reservada y ya no la puede usar nadie más. Además esta posición a partir de ahora se le llama *numero*. Como le hemos dado el valor 43 a *numero*, el valor 43 se almacena en *numero*, es decir, en la posición 00003.

00001		
00002		
00003	43	← numero
00004		
00005		
00006		

Veamos cómo hubiera sido el resultado del programa anterior con esta situación:

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

int main()
{
    char numero;

    numero = 43;

    printf( "La variable numero " );
    printf( "se almacena en la posición de memoria %p,", (void *) &numero );
    printf( "y su valor es %d\n", numero );

    return EXIT_SUCCESS;
}
```

El resultado sería:

La variable numero se almacena en la posición de memoria *00003*, y su valor es *43*

Creo que así ya está clara la diferencia entre el valor de una variable (43) y su dirección (00003).

Ahora vamos un poco más allá, vamos a declarar un puntero. Hemos dicho que un puntero sirve para almacenar la direcciones de memoria. Muchas veces los punteros se usan para guardar las direcciones de variables. Vimos en el capítulo Tipos de Datos que cada tipo de variable ocupa un espacio distinto en la memoria. Por eso cuando declaramos un puntero debemos especificar el tipo de datos cuya dirección almacenará. En nuestro ejemplo queremos que almacene la dirección de una variable char. Así que para declarar el puntero **punt** debemos hacer:

```
char *punt;
```

El * (asterisco) sirve para indicar que se trata de un puntero, debe ir antes del nombre de la variable.

NOTA: El lenguaje C es un lenguaje de "formato libre" y la declaración de la variable "punt" podría realizarse en cualquiera de estas formas, todas ellas validas:

```
char*punt; +
char* punt; +
char * punt; +
char *punt; +
char +
* +
punt;
```

En la variable punt sólo se pueden guardar direcciones de memoria, no se pueden guardar datos. Vamos a volver sobre el ejemplo anterior un poco ampliado para ver cómo funciona un puntero:

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

```
int main()
```

```
{
```

```
char numero;
```

```
char *punt;
```

```
numero = 43;
```

```
punt = &numero;
```

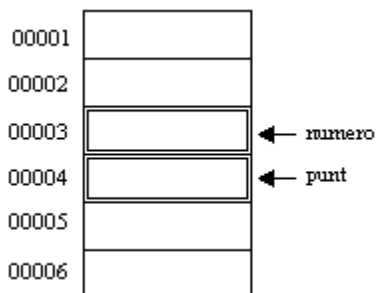
```
printf( "La variable numero se almacena en la posición de memoria %p, y  
su valor es %d\n", (void *) &numero, numero );
```

```
return EXIT_SUCCESS;
```

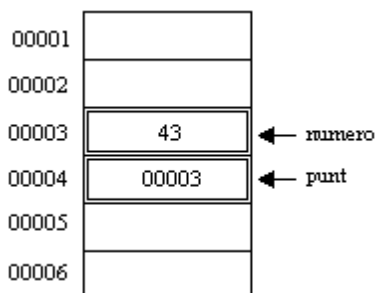
```
}
```

Vamos a ir línea a línea:

- En el primer *int numero* reservamos memoria para *numero* (supongamos que queda como antes, posición 00003). Por ahora *numero* no tiene ningún valor.
- Siguiendo línea: *int *punt;*. Reservamos una posición de memoria para almacenar el puntero, por ejemplo en la posición 00004. Por ahora *punt* no tiene ningún valor, es decir, no apunta a ninguna variable. Esto es lo que tenemos por ahora:



- Tercera línea: *numero = 43;*. Aquí ya estamos dando el valor 43 a *numero*. Se almacena 43 en la dirección 00003, que es la de *numero*.
- Cuarta línea: *punt = №*. Por fin damos un valor a *punt*. El valor que le damos es la dirección de *numero* (ya hemos visto que *&* devuelve la dirección de una variable). Así que *punt* tendrá como valor la dirección de *numero*, 00003. Por lo tanto ya tenemos:



Cuando un puntero tiene la dirección de una variable se dice que ese puntero **apunta** a esa variable.

**NOTA:* La declaración de un puntero depende del tipo de dato al que queramos apuntar. En general la declaración es:

```
tipo_de_dato *nombre_del_puntero;
```

Si en vez de querer apuntar a una variable tipo *char* como en el ejemplo hubiese sido de tipo *int*:

```
int *punt;
```

Para qué sirve un puntero y cómo se usa

Los punteros tienen muchas utilidades, por ejemplo nos permiten pasar variables a una función y modificarlos. También permiten el manejo de cadenas de texto de arrays, de ficheros y de listas enlazadas (ya veremos todo esto más adelante). Otro uso es que nos permiten acceder directamente a la pantalla, al teclado y a todos los componentes del ordenador (sólo en determinados sistemas operativos).

Pero si sólo sirvieran para almacenar direcciones de memoria no servirían para mucho. Nos deben dejar también la posibilidad de acceder a esas posiciones de memoria. Para acceder a ellas se usa el operador ***, que no hay que confundir con el de la multiplicación.

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

```
int main()
```

```
{
```

```
    char numero;
```

```
    char *punt;
```

```
    numero = 43;
```

```
    punt = &numero;
```

```
    printf( "La variable numero se almacena en la posición de memoria %p, y  
    su valor es %d.\n",
```

```
    (void *) &numero, **punt* );
```

```
    return EXIT_SUCCESS;
```

```
}
```

Si nos fijamos en lo que ha cambiado con respecto al ejemplo anterior, vemos que para acceder al valor de número usamos **punt* en vez de *numero*. Esto es así porque *punt* apunta a *numero* y **punt* nos permite acceder al valor al que apunta *punt*.

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

```
int main()
```

```
{
```

```
    char numero;
```

```
    char *punt;
```

```
    numero = 43;
```

```
    punt = &numero;
```

```
    **punt = 30;*
```

```
    printf( "La variable numero se almacena en la posición de memoria %p, y  
    su valor es %d.\n", (void *) &numero, numero );
```

```
    return EXIT_SUCCESS;
```

```
}
```

Ahora hemos cambiado el valor de *numero* a través de **punt*.

En resumen, usando *punt* podemos apuntar a una variable y con **punt* vemos o cambiamos el contenido de esa variable.

Un puntero no sólo sirve para apuntar a una variable, también sirve para apuntar una dirección de memoria determinada. Esto tiene muchas aplicaciones, por ejemplo nos permite controlar el hardware directamente (en MS-Dos y Windows, no en Linux). Podemos escribir directamente sobre la memoria de vídeo y así escribir directamente en la pantalla sin usar `printf`.

Usando punteros en una comparación

Veamos el siguiente ejemplo. Queremos comprobar si dos variables son iguales usando punteros:

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

```
int main()
```

```
{
```

```
    int a, b;
```

```
    int *punt1, *punt2;
```

```
    a = 5; b = 5;
```

```
    punt1 = &a; punt2 = &b;
```

```
    if ( punt1 == punt2 )
```

```
        printf( "Son iguales.\n" );
```

```
    return EXIT_SUCCESS;
```

```
}
```

Alguien podría pensar que el *if* se cumple y se mostraría el mensaje *Son iguales* en pantalla. Pues no es así, el programa es erróneo. Es cierto que *a* y *b* son iguales. También es cierto que *punt1* apunta a *a* y *punt2* a *b*. Lo que queríamos comprobar era si *a* y *b* son iguales. Sin embargo con la condición estamos comprobando si *punt1* apunta al mismo sitio que *punt2*, estamos comparando las direcciones donde apuntan. Por supuesto *a* y *b* están en distinto sitio en la memoria así que la condición es falsa. Para que el programa funcionara deberíamos usar los asteriscos:

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

```
int main()
```

```
{
```

```
    int a, b;
```

```
int *punt1, *punt2;
```

```
a = 5; b = 5;
```

```
punt1 = &a; punt2 = &b;
```

```
if ( **punt1 == *punt2* )
```

```
printf( "Son iguales.\n" );
```

```
return EXIT_SUCCESS;
```

```
}
```

Ahora sí. Estamos comparando el contenido de las variables a las que apuntan punt1 y punt2. Debemos tener mucho cuidado con esto porque es un error que se nos puede escapar con mucha facilidad.

Vamos a cambiar un poco el ejemplo. Ahora *b* no existe y punt1 y punt2 apuntan a *a*. La condición se cumplirá porque apuntan al mismo sitio.

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

```
int main()
```

```
{
```

```
int a;
```

```
int *punt1, *punt2;
```

```
a = 5;
```

```
*punt1 = &a; punt2 = &a;*
```

```
if ( punt1 == punt2 )
```

```
printf( "punt1 y punt2 apuntan al mismo sitio.\n" );
```

```
return EXIT_SUCCESS;
```

```
}
```

Punteros como argumentos de funciones

Hemos visto en el capítulo de funciones cómo pasar parámetros y cómo obtener resultados de las funciones (con los valores devueltos con return). Pero tiene un inconveniente, sólo podemos tener un valor devuelto. Ahora vamos a ver cómo los punteros nos permiten modificar varias variables en una función.

Hasta ahora para pasar una variable a una función hacíamos lo siguiente:

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

```
int suma( int a, int b )
```

```
{
```

```
    return a+b;
```

```
}
```

```
int main()
```

```
{
```

```
    int var1, var2, resultado;
```

```
    var1 = 5; var2 = 8;
```

```
    resultado = suma(var1, var2);
```

```
    printf( "La suma es : %i\n", resultado );
```

```
    return EXIT_SUCCESS;
```

```
}
```


Aquí hemos pasado a la función los parámetros *a* y *b* (que no podemos modificar) y nos devuelve la suma de ambos.

Vamos a modificar el ejemplo para que use punteros:

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

```
void suma( int a, int b, int *total )
```

```
{
```

```
    *total = a + b;
```

```
}
```

```
int main()
```

```
{
```

```
    int var1, var2, resultado;
```

```
    var1 = 5; var2 = 8;
```

```
    suma(var1, var2, &resultado);
```

```
    printf( "La suma es: %d.\n", resultado );
```

```
    return EXIT_SUCCESS;
```

```
}
```

Como podemos ver la función ya no devuelve un valor, pero le hemos añadido un tercer parámetro *int *total*. Este parámetro es un puntero que va a recibir la dirección donde se almacena *resultado* (*&resultado*) y va a guardar ahí el resultado de la suma. Cuando finalice la función y volvamos a la función *main* la variable *resultado* se encontrará con que tiene como valor la suma de los dos números.

Supongamos ahora que queremos tener la suma pero además queremos que *var1* se haga cero dentro de la función. Para eso haríamos lo siguiente:

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

```
int suma_y_cambia( int *a, int b )
```

```
\{
    int c;

    c = *a + b;

    *a = 0;

    return c;
}
```

```
int main()
```

```
\{
    int var1, var2, resultado;

    var1 = 5; var2 = 8;

    resultado = suma_y_cambia(&var1, var2);

    printf( "La suma es: %d y var1 vale: %d.\n", resultado , var1 );

    return EXIT_SUCCESS;
}
```

Fijémonos en lo que ha cambiado (con letra en **negrita**): En la función `suma` hemos declarado *a* como puntero. En la llamada a la función (dentro de `main`) hemos puesto `&` para pasar la dirección de la variable `var1`. Ya sólo queda hacer cero a `var1` a través de `*a=0`.

También usamos una variable *c* que nos servirá para almacenar la suma de *a* y *b*.

Es importante no olvidar el operador `&` en la llamada a la función ya que sin el no estaríamos pasando la dirección de la variable sino el valor de *var1*.

Podemos usar tantos punteros como queramos en la definición de la función.

NOTA IMPORTANTE: Existe la posibilidad de hacer el ejercicio de esta otra manera, sin usar la

variable *resultado*:

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

```
int suma( int *a, int b )
```

```
{
```

```
    int c;
```

```
    c = *a + b;
```

```
    *a = 0;
```

```
    return c;
```

```
}
```

```
int main()
```

```
{
```

```
    int var1, var2;
```

```
    var1 = 5; var2 = 8;
```

```
    printf( "La suma es: %d y var1 vale: %d\n", suma(&var1, var2) , var1 );
```

```
    return EXIT_SUCCESS;
```

```
}
```

Sin embargo, esto puede dar problemas, ya que no podemos asegurar de cómo va a evaluar el compilador los argumentos de `printf`. Es posible que primero almacene el valor de `var1` antes de evaluar *suma*. Si ocurriese así el resultado del programa sería: *La suma es 13 y a vale 5*, en lugar de *La suma es 13 y a vale 0*.

Ejercicios

**Ejercicio 1:* Encuentra un fallo muy grave:

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

```
int main()
```

```
{
```

```
    int *a;
```

```
    *a = 5;
```

```
    return EXIT_SUCCESS;
```

```
}
```

Solución: No hemos dado ninguna dirección al puntero. No sabemos a dónde apunta. Puede apuntar a cualquier sitio, al darle un valor estamos escribiendo en un lugar desconocido de la memoria. Esto puede dar problemas e incluso bloquear el ordenador. Recordemos que al ejecutar un programa éste se copia en la memoria, al escribir en cualquier parte puede que estemos cambiando el programa (en la memoria, no en el disco duro).

**Ejercicio 2:* Escribe un programa que asigne un valor a una variable de tipo *int*. Hacer un puntero que apunte a ella y sumarle 3 usando el puntero. Luego mostrar el resultado.

Solución: Esta es una posible solución:

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

```
int main()
```

```
{
```

```
    int a;
```

```
    int *b;
```

```
    a = 5;
```

```
    b = &a;
```

```
*b += 3;
```

```
printf( "El valor de a es = %d.\n", a );
```

```
return EXIT_SUCCESS;
```

```
}
```

También se podía haber hecho:

```
printf( "El valor de a es = %d\n", *b );
```

Capítulo 10. Arrays

¿Qué es un array?

Nota: algunas personas conocen a los *arrays* como *arreglos*, *matrices* o *vectores*. Sin embargo, en este curso, vamos a usar el término array ya que es, según creo, el más extendido en la bibliografía sobre el tema.

La definición sería algo así:

Un array es un conjunto de variables del mismo tipo que tienen el mismo nombre y se diferencian en el índice.

Pero ¿qué quiere decir esto y para qué lo queremos? Pues bien, supongamos que somos un meteorólogo y queremos guardar en el ordenador la temperatura que ha hecho cada hora del día. Para darle cierta utilidad al final calcularemos la media de las temperaturas. Con lo que sabemos hasta ahora sería algo así (que nadie se moleste ni en probarlo):

```

#include <stdio.h>
#include <stdlib.h>

int main()
{
    /* Declaramos 24 variables, una para cada hora del día */
    int temp1, temp2, temp3, temp4, temp5, temp6, temp7, temp8;
    int temp9, temp10, temp11, temp12, temp13, temp14, temp15, temp16;
    int temp17, temp18, temp19, temp20, temp21, temp22, temp23, temp0;

    float media;

    /* Ahora tenemos que dar el valor de cada una */
    printf( "Temperatura de las 0: " );
    scanf( "%d", &temp0 );

    printf( "Temperatura de las 1: " );
    scanf( "%d", &temp1 );

    printf( "Temperatura de las 2: " );
    scanf( "%d", &temp2 );

    ...

    printf( "Temperatura de las 23: " );
    scanf( "%d", &temp23 );

    media = ( temp0 + temp1 + temp2 + temp3 + temp4 + ... + temp23 ) / 24;
    printf( "\nLa temperatura media es %f\n", media );

    return EXIT_SUCCESS;
}

```



Los puntos suspensivos los he puesto para no tener que escribir todo y que no ocupe tanto, no se pueden usar en un programa.

Para acortar un poco el programa podríamos hacer algo así:

```

#include <stdio.h>
#include <stdlib.h>

int main()
{
    /* Declaramos 24 variables, una para cada hora del día */
    int temp1, temp2, temp3, temp4, temp5, temp6, temp7, temp8;
    int temp9, temp10, temp11, temp12, temp13, temp14, temp15, temp16;
    int temp17, temp18, temp19, temp20, temp21, temp22, temp23, temp0;

    float media;

    /* Ahora tenemos que dar el valor de cada una */
    printf( "Introduzca las temperaturas desde las 0 hasta las 23 separadas
por un espacio: " );
    scanf( "%d %d %d ... %d", &temp0, &temp1, &temp2, ... &temp23 );

    media = ( temp0 + temp1 + temp2 + temp3 + temp4 + ... + temp23 ) / 24;

    printf( "\nLa temperatura media es %f\n", media );

    return EXIT_SUCCESS;
}

```

Lo que no deja de ser un muy laborioso además de confuso para el usuario. Y esto con un ejemplo que tiene tan sólo 24 variables, ¡imagínate si son más!

Y precisamente aquí es donde nos vienen de perlas los arrays. Vamos a hacer el programa con un array. Usaremos nuestros conocimientos de bucles *for* y de *scanf*.


```

#include <stdio.h>
#include <stdlib.h>

int main()
{
    int temp[24]; /* Con esto ya tenemos declaradas las 24 variables */
    float media = 0;
    int hora;

    /* Ahora tenemos que dar el valor de cada una */
    for( hora=0; hora<24; hora++ ) {
        printf( "Temperatura de las %i: ", hora );
        scanf( "%d", &temp[hora] );

        media += temp[hora];
    }

    media = media / 24;
    printf( "\nLa temperatura media es %f\n", media );

    return EXIT_SUCCESS;
}

```

Como ves es un programa más corto que los anteriores (recuerda que hemos usado puntos suspensivos en los ejemplos anteriores, sin ellos el código hubiera sido mucho más largo).

Como ya hemos comentado cuando declaramos una variable lo que estamos haciendo es reservar una zona de la memoria para ella. Cuando declaramos un array lo que hacemos (en este ejemplo) es reservar espacio en memoria para 24 variables de tipo *int*. El tamaño del array (24) lo indicamos entre corchetes al definirlo.

En este ejemplo recorreremos la matriz mediante un bucle *for* y vamos dando valores a los distintos elementos de la matriz. Para indicar a qué elemento nos referimos usamos un número entre corchetes (en este caso la variable *hora*), este número es lo que se llama **índice**. El primer elemento de la matriz tiene el índice 0, el segundo tiene el 1 y así sucesivamente.

El índice de un elemento es siempre la posición del elemento menos uno. De modo que el cuarto elemento tendrá el índice $4-1 = 3$. Y podemos asignarle un valor haciendo:

```
temp[ 3 ] = 20;
```



No hay que confundirse; En la declaración del array el número entre corchetes es el número de elementos, en cambio cuando ya usamos la matriz el número entre corchetes es el índice.

Declaración de un Array

La forma general de declarar un array es la siguiente:

tipo_de_dato nombre_del_array[*dimensión*];

El *tipo_de_dato* es uno de los tipos de datos conocidos (*int*, *char*, *float*...) o de los definidos por nosotros mismos con *typedef* (lo estudiaremos más adelante). En el ejemplo el tipo de dato que habíamos usado era *int*.

El *nombre_del_array* es el nombre que damos al array, en el ejemplo era *temp*. El nombre de un array tiene las mismas limitaciones que vimos para un nombre de variable.

La *dimensión* es el número de elementos que tiene el array.

Como he indicado antes, al declarar un array reservamos en memoria tantas variables del *tipo_de_dato* como las indicada en *dimensión*.

Sobre la dimensión de un Array

Hemos visto en el ejemplo que tenemos que indicar en varios sitios el tamaño del array: en la declaración, en el bucle for y al calcular la media. Este es un programa pequeño, en un programa mayor probablemente habrá que escribirlo muchas más veces. Si en un momento dado queremos cambiar la dimensión del array tendremos que cambiar todos. Si nos equivocamos al escribir el tamaño (ponemos 25 en vez de 24) cometeremos un error y puede que no nos demos cuenta. Por eso es mejor usar una constante con nombre, por ejemplo ELEMENTOS. Además, nuestro código será más legible.

Este sería el ejemplo anterior usando una constante para el tamaño del array:

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

```
#define ELEMENTOS 24
```

```
int main()
```

```
{
```

```
    int temp[*ELEMENTOS*]; /* Con esto ya tenemos declaradas las 24
    variables */
```

```
    float media = 0;
```

```
    int hora;
```

```
/* Ahora tenemos que dar el valor de cada una */
```

```
for( hora=0; hora<**ELEMENTOS**; hora++ )
```

```
\{
```

```
printf( "Temperatura de las %i: ", hora );
```

```
scanf( "%d", &temp[hora] );
```

```
media += temp[hora];
```

```
}
```

```
media = media / *ELEMENTOS*;
```

```
printf( "\nLa temperatura media es %f\n", media );
```

```
return 0;
```

```
}
```

Ahora con sólo cambiar el valor de elementos una vez lo estaremos haciendo en todo el programa.

Inicializar un array

En **C** se pueden inicializar los arrays al declararlos igual que hacíamos con las variables. Recordemos que se podía hacer:

```
int numero = 34;
```

Con arrays se puede hacer:

```
int temperaturas[24] = \{
```

```
15, 18, 20, 23, 22, 24, 22, 25,
```

```
26, 25, 24, 22, 21, 20, 18, 17,
```

```
16, 17, 15, 14, 14, 14, 13, 12
```

```
};
```

Así el primer elemento del array (que tiene índice 0), es decir temperaturas[0] valdrá 15. El segundo elemento (temperaturas[1]) valdrá 18 y así con todos. Vamos a ver un ejemplo:

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

```
int main()
```

```
{
```

```
int hora;
```

```
int temperaturas[24] = { 15, 18, 20, 23, 22, 24, 22, 25, 26, 25, 24,
```

```
22, 21, 20, 18, 17, 16, 17, 15, 14, 14, 14,
```

```
13, 12 };
```

```
for (hora=0 ; hora<24 ; hora++ )
```

```
{
```

```
printf( "La temperatura a las %i era
```

```
de %i grados.\n", hora,
```

```
temperaturas[hora] );
```

```
}
```

```
return 0;
```

```
}
```

Nota: en la inicialización de arrays sólo pueden usarse numeros y constantes. No se pueden usar variables. Por ejemplo:

```
#define ELEMENTOS 24
```

```
i. + int array[3] = \{1, ELEMENTOS, 3};
```

Pero a ver quién es el habilidoso que no se equivoca al meter los datos, no es difícil olvidarse alguno. Hemos indicado al compilador que nos reserve memoria para un array de 24 elementos de tipo int. ¿Qué ocurre si metemos menos de los reservados? Pues no pasa nada, sólo que los elementos que falten valdrán cero.

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

```
int main()
```

```
\{
```

```
int hora;
```

```
/* Faltan los tres últimos elementos */
```

```
int temperaturas[24] = \{ 15, 18, 20, 23, 22, 24, 22, 25, 26, 25,
```

```
24, 22, 21, 20, 18, 17, 16, 17, 15, 14, 14 };
```

```
for (hora=0 ; hora<24 ; hora++ )
```

```
\{
```

```
printf( "La temperatura a las %i era de %i grados.\n", hora,  
temperaturas[hora] );
```

```
}
```

```
return 0;
```

```
}
```

El resultado será:

La temperatura a las 0 era de 15 grados.

La temperatura a las 1 era de 18 grados.

La temperatura a las 2 era de 20 grados.

La temperatura a las 3 era de 23 grados.

...

La temperatura a las 17 era de 17 grados.

La temperatura a las 18 era de 15 grados.

La temperatura a las 19 era de 14 grados.

La temperatura a las 20 era de 14 grados.

La temperatura a las 21 era de 0 grados.

La temperatura a las 22 era de 0 grados.

La temperatura a las 23 era de 0 grados.

Vemos que los últimos 3 elementos son nulos, que son aquellos a los que no hemos dado valores. El compilador no nos avisa que hemos metido menos datos de los reservados.

NOTA: Fíjate que para recorrer del elemento 0 al 23 (24 elementos) hacemos:

```
for(hora=0; hora<24; hora++)
```

La condición es que *hora* sea menor que 24. También podíamos haber hecho que *hora!=24*, pero es menos correcto.

Ahora vamos a ver el caso contrario, metemos más datos de los reservados. Vamos a meter 25 en vez de 24. Si hacemos esto dependiendo del compilador obtendremos un error o al menos un warning (aviso). En unos compiladores el programa se creará y en otros no, pero aún así nos avisa del fallo.

Si la matriz debe tener una longitud determinada usamos el método de indicar el número de elementos al declarar el array. En nuestro caso era conveniente, porque los días siempre tienen 24 horas. Es importante definir el tamaño de la matriz para que nos avise si metemos más elementos de los necesarios.

Hay casos en los que podemos usar un método alternativo, dejar al ordenador que cuente los

elementos que hemos metido y nos reserve espacio para ellos:

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

```
int main()
```

```
{
```

```
    int hora;
```

```
    /* Faltan los tres últimos elementos */
```

```
    int **temperaturas**__*[]*__ = {\
```

```
    15, 18, 20, 23, 22,
```

```
    24, 22, 25, 26, 25,
```

```
    24, 22, 21, 20, 18,
```

```
    17, 16, 17, 15, 14,
```

```
    14 };
```

```
    for ( hora=0 ; hora<24 ; hora++ )
```

```
    {\
```

```
        printf( "La temperatura a las %i era de %i grados.\n", hora,  
        temperaturas[hora] );
```

```
    }
```

```
    return 0;
```

```
}
```

Vemos que no hemos especificado la dimensión del array *temperaturas*. Hemos dejado los corchetes en blanco. El ordenador contará los elementos que hemos puesto entre llaves y reservará espacio para ellos. De esta forma siempre habrá el espacio necesario, ni más ni menos. La pega es que si ponemos más de los que queríamos no nos daremos cuenta, como en el ejemplo.

Este es el resultado que obtendríamos:

La temperatura a las 0 era de 15 grados.

La temperatura a las 1 era de 18 grados.

La temperatura a las 2 era de 20 grados.

...

La temperatura a las 20 era de 14 grados.

La temperatura a las 21 era de -1216612880 grados.

La temperatura a las 22 era de 0 grados.

La temperatura a las 23 era de 134513819 grados.

Vemos que las últimas tres líneas dan un resultado extraño. ¿De dónde salen esos números? Se debe a que el array sólo tenía 21 elementos pero hemos leído 24 valores. Los tres últimos son datos que había en la memoria y que no tenían relación con el array. Se debe tener mucho cuidado con esto.

Para saber en este caso cuantos elementos tiene la matriz podemos usar el operador sizeof. Dividimos el tamaño de la matriz entre el tamaño de sus elementos y tenemos el número de elementos.

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

```
int main()
```

```
{
```

```
    int hora;
```

```
    int elementos;
```

```
    int temperaturas[] = \{ 15, 18, 20, 23, 22, 24, 22, 25, 26, 25,
```

```
    24, 22, 21, 20, 18, 17, 16, 17, 15, 14, 14 };
```

```
    *elementos = sizeof temperaturas / sizeof(int);*
```

```
    for ( hora=0 ; hora<elementos ; hora++ )
```



```
\{
```

```
printf( "La temperatura a las %i era de %i grados.\n", hora,  
temperaturas[hora] );
```

```
}
```

```
printf( "Han sido %i elementos.\n" , elementos );
```

```
return 0;
```

```
}
```

Ahora el resultado será correcto (sólo mostrará los 21 valores que hay en el array):

La temperatura a las 0 era de 15 grados.

La temperatura a las 1 era de 18 grados.

La temperatura a las 2 era de 20 grados.

...

La temperatura a las 19 era de 14 grados.

La temperatura a las 20 era de 14 grados.

Veamos qué pasa si ahora intentamos mostrar más elementos de los que hay en la matriz, en este caso intentamos imprimir 28 elementos cuando sólo hay 24:

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

```
int main()
```

```
\{
```

```
int hora;
```

```
int temperaturas[24] = \{
```

```
15, 18, 20, 23, 22,
```

```
24, 22, 25, 26, 25,
```

```
24, 22, 21, 20, 18,
```

```
17, 16, 17, 15, 14,
```

```
14, 13, 13, 12 };
```

```
for (hora=0 ; *hora<28* ; hora++ )
```

```
\{
```

```
printf( "La temperatura a las %i era de %i grados.\n", hora,  
temperaturas[hora] );
```

```
}
```

```
return 0;
```

```
}
```

Lo que se obtiene es algo similar a esto:

La temperatura a las 22 era de 15 grados.

...

La temperatura a las 23 era de 12 grados.

La temperatura a las 24 era de 24 grados.

La temperatura a las 25 era de 3424248 grados.

La temperatura a las 26 era de 7042 grados.

La temperatura a las 27 era de 1 grados.

Vemos que a partir del elemento 24 (incluido) tenemos resultados extraños. Esto es porque nos hemos salido de los límites del array e intenta acceder al elemento `temperaturas[25]` y sucesivos que no existen. Así que nos muestra el contenido de la memoria que está justo detrás de

temperaturas[23] que puede ser cualquiera. Al contrario que otros lenguajes C no comprueba los límites de los array, nos deja saltárnoslos a la torera. Este programa no da error al compilar ni al ejecutar, tan sólo devuelve resultados extraños. Tampoco bloqueará el sistema porque no estamos escribiendo en la memoria sino leyendo de ella.

Otra cosa muy diferente es meter datos en elementos que no existen. Veamos un ejemplo **(ni se te ocurra ejecutarlo)**:

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

```
int main()
```

```
{
```

```
    int temp[24];
```

```
    float media = 0;
```

```
    int hora;
```

```
    for( hora=0; *hora<28*; hora++ )
```

```
    {
```

```
        printf( "Temperatura de las %d: ", hora );
```

```
        scanf( "%d", &temp[hora] );
```

```
        media += temp[hora];
```

```
    }
```

```
    media = media / 24;
```

```
    printf( "\nLa temperatura media es %f\n", media );
```

```
return 0;
```

```
}
```

En muchos ordenadores seguramente el programa se cerrará por un error o puede que incluso quede bloqueado el ordenador. Es probable que incluso haya que apagarlo. El problema ahora es que estamos intentando escribir en el elemento *temp[24]* que no existe y puede ser un lugar cualquiera de la memoria. Como consecuencia de esto podemos estar cambiando algún programa o dato de la memoria que no debemos y el sistema hace pluf. Así que mucho cuidado con esto.

Punteros a arrays

Aquí tenemos otro de los importantes usos de los punteros, los punteros a arrays. Estos están íntimamente relacionados.

Para que un puntero apunte a un array se puede hacer de dos formas, una es apuntando al primer elemento del array:

```
int *puntero;
```

```
int temperaturas[24];
```

```
puntero = &temperaturas[0];
```

El puntero apunta a la dirección del primer elemento. Otra forma equivalente, pero mucho más usada es:

```
puntero = temperaturas;
```

Con esto también apuntamos al primer elemento del array. Fijaos que el puntero tiene que ser del mismo tipo que el array (en este caso int).

Ahora vamos a ver cómo acceder al resto de los elementos. Para ello empezamos por cómo funciona un array: Un array se guarda en posiciones consecutivas en memoria, de tal forma que el segundo elemento va inmediatamente después del primero en la memoria. En un ordenador en el que el tamaño del tipo int es de 32 bits (4 bytes) cada elemento del array ocupará 4 bytes. Veamos un ejemplo:

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

```
int main()
```

```
{
```

```
int i;
```

```
int temp[24];
```

```
for( i=0; i<24; i++ )
```

```
\{
```

```
printf( "La dirección del elemento %i es %p.\n",
```

```
i, (void *)&temp[i] );
```

```
}
```

```
return 0;
```

```
}
```

NOTA: Recuerda que %p sirve para mostrar en pantalla una posición de memoria en hexadecimal.

El resultado es (en mi ordenador):

La dirección del elemento 0 es 4c430.

La dirección del elemento 1 es 4c434.

La dirección del elemento 2 es 4c438.

La dirección del elemento 3 es 4c43c.

...

La dirección del elemento 21 es 4c484.

La dirección del elemento 22 es 4c488.

La dirección del elemento 23 es 4c48c.

(Las direcciones están en hexadecimal). Vemos aquí que efectivamente ocupan posiciones consecutivas y que cada una ocupa 4 bytes. Si lo representamos en una tabla:

4C430	4C434	4C438	4C43C
temp[0]	temp[1]	temp[2]	temp[3]

Ya hemos visto cómo funcionan los arrays por dentro, ahora vamos a verlo con punteros. Voy a poner un ejemplo:

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

```
int main()
```

```
{
```

```
int i;
```

```
int temp[24];
```

```
int *punt;
```

```
*punt = temp;
```

```
for( i=0; i<24; i++ )
```

```
{
```

```
printf( "La dirección de temp[%i] es %p y la de punt es %p.\n",
```

```
i, (void *) &temp[i], (void *) *punt* );
```

```
*punt++;
```

```
}
```

```
return 0;
```

```
}
```

Cuyo resultado es:

La dirección de temp[0] es 4c430 y la de punt es 4c430.

La dirección de temp[1] es 4c434 y la de punt es 4c434.

La dirección de temp[2] es 4c438 y la de punt es 4c438.

...

La dirección de temp[21] es 4c484 y la de punt es 4c484.

La dirección de temp[22] es 4c488 y la de punt es 4c488.

La dirección de temp[23] es 4c48c y la de punt es 4c48c.

En este ejemplo hay dos líneas importantes (en negrita). La primera es *punt = temp*. Con esta hacemos que el punt apunte al primer elemento de la matriz. Si no hacemos esto punt apunta a un sitio cualquiera de la memoria y debemos recordar que no es conveniente dejar los punteros así, puede ser desastroso.

La segunda línea importante es *punt*. Con esto incrementamos el valor de *punt*, pero curiosamente aunque incrementamos una unidad (*punt* equivale a *punt=punt+1*) el valor aumenta en 4. Aquí se muestra una de las características especiales de los punteros. Recordemos que en un puntero se guarda una dirección. También sabemos que un puntero apunta a un tipo de datos determinado (en este caso int). Cuando sumamos 1 a un puntero sumamos el tamaño del tipo al que apunta. En el ejemplo el puntero apunta a una variable de tipo int que es de 4 bytes, entonces al sumar 1 lo que hacemos es sumar 4 bytes. Con esto lo que se consigue es apuntar a la siguiente posición int de la memoria, en este caso es el siguiente elemento de la matriz.

Esta tabla describe el bucle programa paso a pos:

Operación	Equivalente	Valor de <i>punt</i>
<i>punt = temp</i> ;	<i>punt = &temp[0]</i> ;	4c430
<i>punt++</i> ; (en el primer ciclo del for)	sumar 4 al contenido de <i>punt</i> (4c430 + 4)	4c434
<i>punt++</i> ; (en el segundo ciclo del for)	sumar 4 al contenido de <i>punt</i> (4c434 + 4)	4c438

Cuando hemos acabado estamos en temp[24] que no existe. Si queremos hacer que *punt* vuelva al elemento 1 podemos hacer *punt = temp* otra vez o restar 24 a *punt*:

punt -= 24;

con esto hemos restado 24 posiciones a *punt* (24 posiciones int*4 bytes por cada int= 96 posiciones).

Al final del programa *punt* apunta a la dirección de memoria 4C490. Para volver a la primera posición hemos dicho que restamos 24, que es equivalente a hacer:

$$4C490 - 18 * 4 = 4c430$$

donde:

- 4C490 es la posición de *punt*.
- 18 es el número de posiciones que queremos restar (¡ojo! 24 en hexadecimal es 18).
- 4 es el tamaño de un *int* (en el sistema donde se ha probado el ejemplo anterior).

Si coges una calculadora científica podrás ver que los números coinciden.

Vamos a ver ahora un ejemplo de cómo recorrer la matriz entera con punteros y cómo mostrarla en pantalla:

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

```
int main()
```

```
{
```

```
    int temperaturas[24] = {\
```

```
        15, 18, 20, 23, 22,
```

```
        24, 22, 25, 26, 25,
```

```
        24, 22, 21, 20, 18,
```

```
        17, 16, 17, 15, 14,
```

```
        14, 13, 12, 12 };
```

```
    int *punt;
```

```
    int i;
```

```
    punt = temperaturas;
```

```
    for( i=0 ; i<24; i++ )
```

```
    {\
```

```
        printf( "Elemento %i: %i\n", i, *punt );
```

```
        punt++;
```



```
}
```

```
return 0;
```

```
}
```

Cuando termina el bucle *for* el puntero *punt* apunta a *temperaturas[24]*, y no al primer elemento, si queremos volver a recorrer la matriz debemos volver como antes al comienzo. Para evitar perder la referencia al primer elemento de la matriz (*temperaturas[0]*) se puede usar otra forma de recorrer la matriz con punteros:

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

```
int main()
```

```
{
```

```
int temperaturas[24] = {
```

```
15, 18, 20, 23, 22,
```

```
24, 22, 25, 26, 25,
```

```
24, 22, 21, 20, 18,
```

```
17, 16, 17, 15, 14,
```

```
14, 13, 12, 12 };
```

```
int *punt;
```

```
int i;
```

```
punt = temperaturas;
```

```
for( i=0 ; i<24; i++ )
```

```
\{
```

```
printf( "Elemento %i: %i\n", i, _** (punt+i)*_ );
```

```
}
```

```
return 0;
```

```
}
```

Con **(punt+i)* lo que hacemos es tomar la dirección a la que apunta *punt* (la dirección del primer elemento de la matriz) y le sumamos *i* posiciones. De esta forma tenemos la dirección del elemento *i*. No estamos sumando un valor a *punt*, para sumarle un valor habría que hacer *punt+* o *punt=algo*, así que *punt* siempre apunta al principio de la matriz.

Se podría hacer este programa sin usar *punt*. Sustituyéndolo por *temperaturas* y dejar **(temperaturas+i)*. Lo que no se puede hacer es: *temperaturas++*;

Importante: Como final debo comentar que el uso de índices es una forma de maquillar el uso de punteros. El ordenador convierte los índices a punteros. Cuando al ordenador le decimos *temp[5]* en realidad le estamos diciendo **(temp+5)*. Así que usar índices es casi equivalente a usar punteros de una forma más cómoda (en la sección siguiente vamos a ver una diferencia).

Las que sí son equivalentes son estas dos definiciones:

```
int temp[];
```

```
int *temp;
```

Paso de un array a una función

En C se suele usar un puntero cuando se quiere pasar un parámetro a una función:

```
int sumar( int *m )
```

Otras declaraciones equivalentes serían:

```
int sumar( int m[] )
```

o

```
int sumar( int m[10] )
```

En realidad esta última no se suele usar, porque el número de elementos es ignorado por el compilador.

Con el puntero que hemos usado en la definición de la función podemos recorrer el array:

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

```
int sumar( int *m )
```

```
{
```

```
    int suma, i;
```

```
    suma = 0;
```

```
    for( i=0; i<10; i++ )
```

```
    {
```

```
        suma += m[i];
```

```
    }
```

```
    return suma;
```

```
}
```

```
int main()
```

```
{
```

```
    int contador;
```

```
    int matriz[10] = { 10, 11, 13, 10, 14, 9, 10, 18, 10, 10 };
```

```
    /* Mostramos el array */
```

```
for( contador=0; contador<10; contador++ )
```

```
printf( " %3i\n", matriz[contador] );
```

```
/* Calculamos la suma de los elementos y la mostramos */
```

```
printf( "+ -----\n" );
```

```
printf( " %3i", sumar( matriz ) );
```

```
return 0;
```

```
}
```

NOTA: Este programa tiene un detalle adicional que es que muestra toda la matriz en una columna. Además se usa para imprimir los números el modificador **%3i**. El 3 indica que se tienen que alinear los números a la derecha, así queda más elegante.

Como he indicado no se pasa el array, sino un puntero a ese array. Antes hemos usado el truco del *sizeof* para calcular el número de elementos de un array. Si lo probamos aquí no funcionará. Vamos a verlo con un ejemplo:

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

```
void calcular_tamano( int *m )
```

```
{
```

```
printf( "Tamaño del array (m, dentro de la función): %i Kb\n", sizeof m );
```

```
}
```

```
int main()
```

```
{
```

```
int matriz[10] = { 10, 11, 13, 10, 14, 9, 10, 18, 10, 10 };
```

```
int *pmatriz;
```

```
pmatriz = matriz;
```

```
printf( "Tamaño del array (matriz): %i Kb\n", sizeof matriz );
```

```
printf( "Tamaño del array (pmatriz): %i Kb\n", sizeof pmatriz );
```

```
calcular_tamano( matriz );
```

```
return 0;
```

```
}
```

El resultado será:

Tamaño del array (matriz): 40 Kb

Tamaño del array (pmatriz): 4 Kb

Tamaño del array (m, dentro de la función): 4 Kb

¿Por qué dice *sizeof* que el tamaño es 4 Kb cuando usamos un puntero? Porque nos calcula el tamaño del tipo de dato al que apunta el puntero. ¿Cómo sabemos entonces cual es el tamaño del array dentro de la función? En este caso lo hemos puesto nosotros mismos, 10. Pero se pueden utilizar constantes como en el apartado “[Sobre la dimensión de un Array](#)”, o se puede pasar el tamaño del array como parámetro a la función.

En el ejemplo usamos un puntero pero vemos que luego estamos usando $m[i]$. Esto lo podemos hacer porque, como se ha mencionado antes, el uso de índices en una forma que nos ofrece C de manejar punteros con matrices. Ya hemos visto que $m[i]$ es equivalente a $*(m+i)$.