

# 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 .....	19
Capítulo 4. Constantes (uso de #define). ....	23
Introducción .....	23
Tipos de datos en las constantes .....	23
Constantes con nombre .....	25
Capítulo 5. Manipulando datos (operadores) .....	28
¿Qué es un operador? .....	28
Operador de asignación .....	28
Operadores aritméticos .....	28
Operadores de comparación .....	33
Operadores lógicos .....	34
Introducción a los bits y bytes .....	36

Operadores de bits.....	36
Operador Sizeof .....	40
Otros operadores .....	41
Orden de evaluación de Operadores.....	41
Ejercicios .....	44

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

Variable local:

```
#include <stdio.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:

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 Modificador  
 int: entero decimal %i  
 int: entero decimal %d  
 unsigned int: entero decimal sin signo %u  
 int: entero octal %o  
 int: entero hexadecimal %x  
 Podemos verlos en acción con el siguiente ejemplo:

```
#include <stdio.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 0;
}
```

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

Cuya salida será:

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



¡Importante! Si imprimimos una variable a la que no hemos dado ningún valor no obtendremos ningún error al compilar pero la variable tendrá un valor cualquiera. Prueba el ejemplo anterior quitando: `int x = 10;`. Puede que te imprima el valor 10 o puede que no.

## 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 usamos. 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>  
  
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 0;  
}
```

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

### 1.5 El modificador Unsigned

Este modificador (que significa sin signo) modifica el rango de valores que puede contener una variable. Sólo 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 excepción 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 explícitamente utilizando ya sea signed char o unsigned char.

### 1.6 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  $3,4 \times 10^{38}$ , que es el número: 340.000.000.000.000.000.000.000.000.000.000.000.000.000. El número 3,4E-38 es equivalente a  $3,4 \times 10^{-38}$ , 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

### 1.7 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.

#### 1.7.1 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>

```
int main() { printf( "%f\n", 10.30 ); return 0; }
```

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:

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

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; }
```

### 1.8 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>\nint main() { printf( "El tipo int ocupa %i bytes\\n", sizeof(int) ); return 0; }
```

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 1byte = 8 bits. Por lo tanto el tipo int ocupa  $4 \times 8 = 32$  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 está 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.

### 1.9 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. 1.10 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. 1.10.1 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> int main() { int num1; num1 = 2147483648; printf( "El valor de num1 es: %i\n", num1 ); }` 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. 1.10.2 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). 1.11 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. 1.12 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	16 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> int main() { printf( "Tamaño (en bits) de int = %i\n", sizeof( int ) * 8 ); return 0; }` 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. 1.13 Ejercicios Ejercicio 1: Busca los errores: `#include <stdio.h> int main() { int número; número = 2; return 0; }` Solución: Los nombres de variables no pueden llevar acentos, luego al compilar número dará error. `#include <stdio.h> int main() { int número; número = 2; printf( "El valor es %i" número ); return 0; }` 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 número de bits es usualmente ocho no tiene por qué ser así y puede ser mayor. Dependerá del compilador.

Para estar seguros del número 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 → 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 → 10101010

155 → 10011011 &

138 → 10001010

## Operador ^ (XOR)

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

235 → 11101011

143 → 10001111 ^

100 → 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.