

Programación





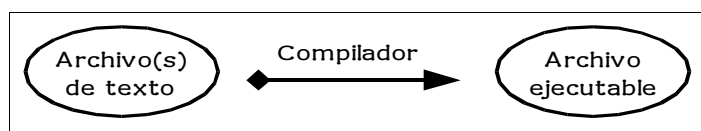
Conceptos generales

Creación de un programa

Simplificando un poco, se puede decir que los programas son **archivos ejecutables**. Internamente, están formados por órdenes para el microprocesador, que sólo éste entiende; las órdenes están en **código máquina**.

Crear programas directamente así es extremadamente difícil (aunque posible), por lo que se han inventado métodos que simplifican la tarea. Lo más habitual es escribir el programa usando un lenguaje que entiendan las personas y posteriormente usar otro programa para que haga la traducción a código máquina.

Los programadores escriben sus programas siguiendo unas determinadas reglas, y lo almacenan en uno o más archivos de texto, fácilmente manipulables. Estos archivos constituyen el **código fuente** del programa, o, sencillamente, *los fuentes*. Un programa llamado **compilador** examina el código fuente; si encuentra errores en la aplicación de las reglas, los indica; si no encuentra errores, produce el archivo ejecutable. Esquemáticamente, éste es el proceso:



Lenguajes de programación

Un lenguaje de programación es un conjunto de reglas que debe seguir el programador para escribir el código fuente de un programa. Existen miles de lenguajes distintos, aunque los más importantes no son más que unas decenas.

Nivel del lenguaje

Algunos lenguajes son más fáciles de entender por las personas y otros son más fáciles de entender por los compiladores. Atendiendo a esta característica se habla de lenguajes de programación de diferentes niveles:

- ◆ Lenguajes de **bajo nivel**. Difíciles de entender por los humanos, muy fáciles para los compiladores. El ejemplo más característico es el lenguaje **Ensamblador**. Estos lenguajes se utilizan en las partes más críticas de los programas, aquellas que deben ejecutarse con mucha rapidez y seguridad, ya que permiten manipular de modo muy cercano las características del microprocesador.
- ◆ Lenguajes de **medio nivel**. Permiten un acceso fácil a aspectos de bajo nivel y también crear estructuras de alto nivel. Estos lenguajes son muy versátiles y se utilizan muy ampliamente. Ejemplos típicos: **C** y **FORTH**. Con ellos se escriben sistemas operativos, software de comunicaciones, etc.
- ◆ Lenguajes de **alto nivel**. Permiten a los programadores concentrarse en aspectos muy generales y abstractos, despreocupándose de las características del microprocesador. Ejemplos: **C++**, **Pascal**, **BASIC**. Se utilizan para crear aplicaciones de tipo general, como procesadores de texto, hojas de cálculo, etc.

Lenguajes importantes

Además de los lenguajes citados anteriormente como ejemplos, que han sido elegidos por su relevancia, es conveniente conocer al menos los nombres de algunos otros lenguajes muy importantes:

- ◆ **FORTRAN**. Es un lenguaje científico, con el que hay escrito gran cantidad de software matemático crítico, usado en modelos de simulación, en control de sistemas, etc.
- ◆ **Cobol**. El típico lenguaje de programación usado en medios financieros; poco a poco está siendo desbancado por lenguajes más modernos, pero sigue funcionando gran cantidad de código escrito en este lenguaje.

Partes de un programa

Simplificando mucho, se puede pensar que un programa consta de dos partes: la interfaz de usuario y las manipulaciones de datos. Con el interfaz de usuario se pide a la persona que usa el programa los datos de partida y se le entrega el resultado de las operaciones. La parte interna del programa procesa los datos iniciales para obtener las soluciones. Es fácil separar las dos partes, y es una buena práctica de programación hacerlo así.

Métodos de programación

Además de un lenguaje de programación, se suele seguir un determinado método escribiendo los programas. Muchas veces el lenguaje determina el método, pero casi siempre son independientes. Los métodos y los lenguajes han ido evolucionando juntos en la breve historia de la informática (poco más de cincuenta años).

Primeros métodos

Los primeros lenguajes eran muy simples y no permitían mucha flexibilidad. Los programas eran muy propensos a errores y muy difíciles de modificar. El lenguaje Ensamblador, FORTH y los primeros BASIC, son buenos ejemplos de ello. Seguir el funcionamiento del código era tan difícil como seguir el rastro de un hilo de *spaghetti* en un plato lleno de ellos. Precisamente se denomina *código spaghetti* al código que se solía escribir así.

Programación estructurada

Para poner orden en el código se inventaron lenguajes como **ALGOL**, que permitieron escribir mejor código en menos tiempo. En la programación estructurada se prohíben los saltos de una parte a otra del código, y esto supone una gran ventaja. Prácticamente todos los lenguajes modernos son estructurados.

Programación orientada al objeto

Una abstracción muy importante fue unir en un todo, llamado objeto, un conjunto de datos y los métodos necesarios para manipularlos. Esta abstracción permitió la creación más sencilla de software reutilizable, es decir, que se escribe una vez para un programa y se puede usar posteriormente en otros programas. El primer lenguaje así fue **Smalltalk**, los lenguajes **Ada** y **Eiffel** aportaron buenas ideas, apareció el C++, que incorpora a la potencia del C la facilidad del nuevo método, y en estos momentos el lenguaje de moda es **Java**. (El simpático logotipo de Java es *Duke*).



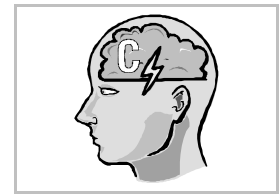
Programación visual

Crear buenas interfaces de usuario consume mucho tiempo y sin embargo es muy fácil de estandarizar. Existen muchas herramientas que permiten crear fácilmente estas interfaces; se suelen llamar herramientas **visuales**. De momento, son inútiles para crear la parte fundamental de los programas, la que transforma los datos. Son muy populares las herramientas Visual Basic, Delphi, Kylix y Visual C++.

Las librerías

Cuando un equipo de programación acomete un nuevo proyecto, no tiene por qué volver a programar todo por primera vez, sino que puede (y debe) utilizar partes que ya hayan sido creadas. En inglés se llaman *librarys* a las partes ya programadas que se pueden incorporar a un programa nuevo. Aunque la traducción correcta de este término inglés es “biblioteca”, lo cierto es que la mala traducción “librería” ha resultado ser la que se ha incorporado al lenguaje informático español.

Existen gran cantidad de librerías disponibles. Los lenguajes ya incorporan las suyas propias, imprescindibles para la creación de cualquier programa por sencillo que sea. Pero también existen enormes librerías que dan resueltos multitud de problemas. Un buen programador debe conocer tanto los lenguajes como las librerías disponibles.



El lenguaje C

Origen

El lenguaje C fue creado hacia 1972 por **Dennis M. Ritchie** en un ordenador PDP-11. El primer objetivo que perseguía era crear un lenguaje sencillo que produjera programas de rápida ejecución, una especie de ensamblador avanzado, que permitiera crear estructuras complejas sin perder de vista las características del ordenador. Su origen se encuentra muy ligado al nacimiento del sistema operativo UNIX, creado, entre otros, por **Brian W. Kernighan**. UNIX está escrito en C, así como la inmensa mayoría de las aplicaciones UNIX. El libro clásico en el que se expone el C es *The C Programming Language*, de Kernighan y Ritchie, publicado en 1978.

En diciembre de 1989 se produjo una importante estandarización del C, creándose lo que se conoce como **ANSI C**, ahora universalmente reconocido.

Aplicación

El C se utiliza en una gran variedad de situaciones; se escriben en C sistemas operativos, aplicaciones científicas y comerciales, juegos, emuladores, compiladores de otros lenguajes, etc.

Es un lenguaje extremadamente flexible, que otorga poder absoluto al programador. Con él se crean programas muy rápidos. Todo esto tiene su contrapartida: cuando se comete un error en un programa, sus efectos pueden ser desastrosos.

Disponibilidad

Este lenguaje está disponible prácticamente para todos los ordenadores del mundo, sean grandes o pequeños, caros o baratos, empresariales o personales. Además, no resulta muy difícil escribir programas en C en un ordenador que se puedan compilar perfectamente en otro; así se dispone de programas **multiplataforma**.

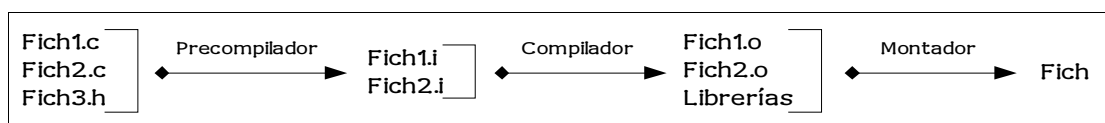
Fases de la compilación

La compilación de programas en C está compuesta en realidad de varias partes bien diferenciadas. Es muy importante conocer estas fases para entender bien el lenguaje y para poder corregir los errores, que pueden aparecer en cualquiera de ellas. Cuando se invoca el compilador, es él quien va ejecutando las diferentes fases, por lo que, en principio, el programador sólo debe conocer el proceso, no dirigirlo.

En esquema, éste es el proceso que se sigue para crear un programa en C:

1. El programador escribe uno o más ficheros fuente, con extensión **c**, y, opcionalmente, uno o más ficheros de cabecera, con extensión **h** (del inglés *header*). En los ficheros de cabecera suele haber definiciones generales y en los fuente, el código.
2. El precompilador examina el código fuente de cada archivo **c**, realiza sustituciones de símbolos e incluye en el código el contenido de los ficheros **h**.
3. El compilador recoge la salida del precompilador y la convierte en código máquina, dejando el resultado en ficheros llamados **ficheros objeto**, que pueden tener extensión **o** o bien **obj**.
4. El montador (llamado en inglés *linker*) une los archivos objeto que provienen de los archivos del programador con otros archivos objeto imprescindibles y con las partes necesarias de los archivos de biblioteca (de extensión **a** o bien **lib**) y forma el archivo ejecutable, que en Windows tiene extensión **exe** y en GNU/Linux no tiene extensión, sino permiso de ejecución.

Y gráficamente se puede ver así:



Entornos integrados de desarrollo

El proceso de creación de un programa, en cualquier lenguaje, exige repetir bastantes veces el proceso de compilación, para corregir los errores sintácticos que van apareciendo y repetir la ejecución del programa para corregir los errores lógicos. Se llama entorno de desarrollo a un pro-

grama que permite editar el código fuente, lanzar el compilador, recoger los errores que éste genere y ejecutar el programa de forma dirigida para poder controlarlo. Estos entornos se denominan en inglés *Integrated Development Environment*, y se conocen con sus siglas: IDE.

Casi todos los compiladores ofrecen dos modos de trabajo: mediante un conjunto de programas independientes o mediante un IDE. Suele ser cuestión de gusto usar uno u otro modo.

Compiladores

Existe una gran cantidad de ellos, tanto libres como en propiedad. Destacaremos algunos:

Para GNU/Linux

En el mundo del software libre tiene una enorme importancia el compilador de C auspiciado por GNU, el `gcc` y el de C++, `gpp`. De hecho, es prácticamente el único que se utiliza en GNU/Linux. El diseño de este compilador permite que sea utilizado fácilmente en muchos otros sistemas operativos. El núcleo Linux se compila con `gcc`. Su mera existencia fue fundamental para que Linus Torvalds comenzara a escribir Linux.

Es el que se utilizará en todos los ejemplos de este curso, aunque éstos se podrán usar con cualquier otro compilador.



Para MS-DOS

- ♦ **DJGPP.** DJ Delorie llevó el `gcc` y el `gpp` a MS-DOS, creando el compilador DJGPP. Se puede obtener en <http://www.delorie.com>. Es un compilador extremadamente potente, con buena documentación, y una gran cantidad de herramientas traídas del mundo UNIX. Una gran cantidad de programas importantes (sobre todo libres) para MS-DOS se compilan con él.
- ♦ **Turbo C++.** Aunque fue comercializado hace muchos años por la empresa Borland, la versión 1.0 se ofreció gratuitamente a los lectores de la revista PC Magazine. Se puede descargar gratuitamente del “museo” de la empresa. Consúltase <http://www.borland.com>. Permite compilar programas en C y en C++, incluye muchas herramientas y un IDE; ocupa muy poco espacio en disco duro y funciona ágilmente en ordenadores de poca potencia.

Para Windows

- ♦ **MinGW32.** Este críptico nombre esconde el compilador “Minimalist GNU for Win32”, es decir, la versión de `gcc` para Windows. Por tanto, es software libre. Si no se dispone de una instalación GNU/Linux para seguir estas hojas, sino una instalación Windows, éste es el compilador recomendado. Su web es <http://www.mingw.org>.
- ♦ **C++ Builder Compiler.** El compilador más avanzado de Borland. Aunque es un programa comercial, se puede descargar gratuitamente de la página web de la compañía la versión más sencilla, que sólo incluye el compilador, sin IDE.
- ♦ **Microsoft Visual C++.** El producto más usado por los desarrolladores profesionales en la plataforma Windows, ya que es un producto de Microsoft. Lleva licencia en propiedad.

GNU+Win32



C en GNU/Linux

Buena pareja

El lenguaje C se concibió, entre otras cosas, para realizar el sistema operativo UNIX; la mayoría de los programas del proyecto GNU se escriben en C; el núcleo Linux está escrito en C. Como se ve, el lenguaje C y el sistema operativo GNU/Linux se complementan perfectamente. Por eso, resulta muy adecuado desarrollar programas en C y aprender el lenguaje en este sistema operativo.

Editores

Para escribir el código es imprescindible un editor de textos, bien independiente o bien el editor del IDE. Ayuda mucho para escribir que el editor coloree de distinta manera cada parte del programa, lo que en inglés se llama *syntax highlighting*; muchos editores disponen de esta característica, como *emacs*, *kwrite* y *zed*.

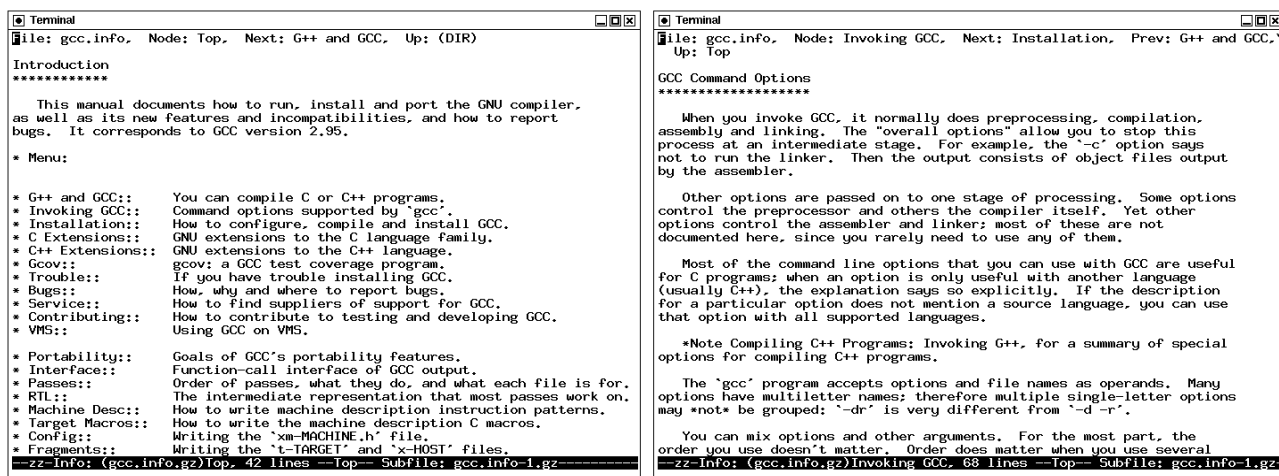
El compilador gcc

Es casi el único que se usa en GNU/Linux, ya que es el compilador del proyecto GNU. Es habitual que la instalación del sistema deje el compilador preparado para trabajar, pero si no es así, hay que instalarlo.



Documentación

La documentación de *gcc* se encuentra en formato *info*, por lo que se puede leer con varios programas diferentes: *info*, *GNOME Help Browser*, *Konqueror*. Véanse dos muestras:



Opciones básicas

Todos los compiladores de C admiten multitud de opciones, aunque para compilar programas sencillos no hacen falta más que unas pocas. Veamos un par de ejemplos, que serán suficientes para seguir el curso:

1. Para compilar los archivos **Fichero1.c** y **Fichero2.c** junto con la librería matemática y crear el ejecutable **Fichero**, se usa:

```
gcc -o fichero fichero1.c fichero2.c -lm
```

Como se ve, la opción **-o** sirve para indicar que el siguiente parámetro es el nombre que se desea dar al ejecutable; si no se usara, el ejecutable creado se llamaría **a.out**.

2. Para compilar los archivos **Fichero1.c** y **Fichero2.c** y crear únicamente sus archivos objeto se usa:

```
gcc -c fichero1.c fichero2.c
```

Queda de manifiesto que la opción **-c** es la que indica que sólo hay que realizar la compilación, pero no el montaje.

3. Para montar los archivos **Fichero1.o** y **Fichero2.o** y crear el ejecutable **Fichero**, se usa:

```
gcc -o fichero fichero1.o fichero2.o
```

Por tanto, es fácil separar las etapas de compilación y montaje, algo que para programas pequeños no es necesario, pero resulta imprescindible para programas grandes compuestos de muchos archivos.

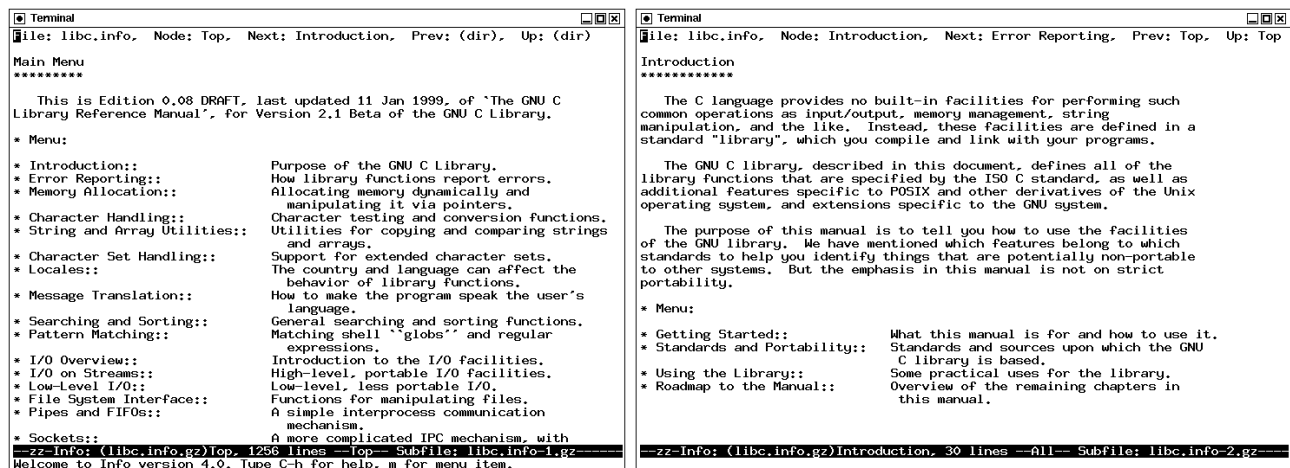
La librería glibc

El lenguaje C no incluye en sí mismo capacidades que en otros lenguajes se dan por supuestas, como imprimir datos en pantalla, por ejemplo. En vez de eso, existen funciones encargadas de realizar gran cantidad de tareas. La biblioteca que reúne las funciones fundamentales de C se llama `libc`, y `glibc` es la versión GNU de `libc`; `glibc` incorpora todas las características de `libc` y algunas más, propias y exclusivas.

En general, cuando se desea realizar alguna acción en la que intervenga el sistema operativo, hay que buscar la función correspondiente consultando la documentación de `libc`, que no es la misma que la de `gcc`. Una vez que se conocen las funciones que más se necesitan, la documentación sólo se consulta de vez en cuando.

Documentación

Se encuentra en formato `info`. Véase una muestra:



```
Terminal
File: libc.info, Node: Top, Next: Introduction, Prev: (dir), Up: (dir)

Main Menu
*****

This is Edition 0.08 DRAFT, last updated 11 Jan 1999, of "The GNU C
Library Reference Manual", for Version 2.1 Beta of the GNU C Library.

* Menu:

* Introduction::          Purpose of the GNU C Library.
* Error Reporting::       How library functions report errors.
* Memory Allocation::     Allocating memory dynamically and
                          manipulating it via pointers.
* Character Handling::     Character testing and conversion functions.
* String and Array Utilities:: Utilities for copying and comparing strings
                          and arrays.
* Character Set Handling:: Support for extended character sets.
                          The country and language can affect the
                          behavior of library functions.
* Locales::               How to make the program speak the user's
                          language.
* Message Translation::   General searching and sorting functions.
                          Matching shell "globs" and regular
                          expressions.
* Searching and Sorting::
* Pattern Matching::
* I/O Overview::          Introduction to the I/O facilities.
* I/O on Streams::        High-level, portable I/O facilities.
* Low-Level I/O::         Low-level, less portable I/O.
* File System Interface:: Functions for manipulating files.
* Pipes and FIFOs::       A simple interprocess communication
                          mechanism.
* Sockets::               A more complicated IPC mechanism, with

--zz-Info: (libc.info.gz)Top, 1256 lines --Top-- Subfile: libc.info-1.gz--
Welcome to Info version 4.0. Type C-h for help. m for menu item.

Terminal
File: libc.info, Node: Introduction, Next: Error Reporting, Prev: Top, Up: Top

Introduction
*****

The C language provides no built-in facilities for performing such
common operations as input/output, memory management, string
manipulation, and the like. Instead, these facilities are defined in a
standard "library", which you compile and link with your programs.

The GNU C Library, described in this document, defines all of the
library functions that are specified by the ISO C standard, as well as
additional features specific to POSIX and other derivatives of the Unix
operating system, and extensions specific to the GNU system.

The purpose of this manual is to tell you how to use the facilities
of the GNU library. We have mentioned which features belong to which
standards to help you identify things that are potentially non-portable
to other systems. But the emphasis in this manual is not on strict
portability.

* Menu:

* Getting Started::       What this manual is for and how to use it.
* Standards and Portability:: Standards and sources upon which the GNU
                          C library is based.
* Using the Library::     Some practical uses for the library.
* Roadmap to the Manual:: Overview of the remaining chapters in
                          this manual.

--zz-Info: (libc.info.gz)Introduction, 30 lines --All-- Subfile: libc.info-2.gz--
```

Programar para GNOME y KDE

La programación para entornos gráficos tiene algunas características que la hacen en principio diferente de la creación de programas para consola. Para crear un programa que tenga un interfaz GNOME o KDE, será necesario seguir ciertas reglas, utilizar algunas funciones específicas y montar el programa con librerías adicionales.

Entornos integrados

Existen en GNU/Linux varios entornos de integrados de programación. El más potente es *KDevelop*, que aunque funciona bajo KDE, permite crear programas para consola, para GNOME y para KDE. Un entorno menos potente es *gIDE*, que funciona bajo GNOME.



Primeros programas

Dificultad del comienzo

En general, en cualquier materia resulta difícil el comienzo porque falta visión de conjunto. Como es necesario presentar los contenidos linealmente, hasta que no se ha desarrollado una buena cantidad de ellos no se puede ver la relación global que liga todas las partes.

Esto es especialmente cierto en programación. Para crear un programa hay que manejar simultáneamente muchos conceptos distintos. Por ello, antes de comenzar con el estudio detallado de las partes principales de la programación en C, se van a presentar en esta hoja unos pequeños programas, muy sencillos, sobre los que se explicará el proceso global de compilación. Evidentemente, según se está comentando, no se podrá comprender completamente el proceso, eso se hará más adelante, pero el objetivo es comenzar por obtener la visión global antes de estudiar los detalles.

Hola, mundo

Éste es el nombre del primer programa que aparece en el libro clásico de Kernighan y Ritchie. A partir de su publicación, es costumbre comenzar por este programa: se trata sencillamente de mostrar por pantalla un mensaje de saludo a todo el mundo. Éste es el proceso:

1. Con cualquier editor de texto se escribe el código. Se almacena en el fichero **holamundo.c**.
2. Se invoca el compilador con la orden `gcc -o hol amundo hol amundo. c`. Esto genera el archivo **holamundo**, que es ejecutable.
3. Se ejecuta el programa con la orden `./hol amundo`.
4. Si en algún momento hay algún error, bien de compilación o de ejecución, se retoca el código fuente y se repite el proceso.

El código

A continuación se presenta el código del programa, junto con una explicación de cada línea. Es importante hacer notar que C es un lenguaje de formato libre: se pueden colocar los distintos elementos como se desee, aunque es habitual seguir consistentemente alguno de los estilos clásicos.

<code>/* Programa Hola mundo */</code>	En C se indican los comentarios con los caracteres <code>/*</code> al principio y <code>*/</code> al final. El compilador los ignora.
<code>#include <stdio.h></code>	Todas las líneas que comienzan con el carácter <code>#</code> son órdenes para el preprocesador. En ésta se indica que incluya el fichero de cabecera stdio.h , en el se encuentra la definición de la función printf() .
<code>int main (void)</code>	Los programas en C comienzan con la ejecución de la función main() . En esta línea se especifica que la función main() no va a tomar ningún parámetro (<i>void</i>) y que va a devolver un número entero (<i>int</i>).
<code>{</code>	Las instrucciones que forman la función main() deben estar encerradas entre llaves. Ésta es la de comienzo. Se ha metido dos espacios hacia la derecha para mejorar la legibilidad del código.
<code>printf ("Hola, mundo\n");</code>	Se invoca la función printf() para que envíe a la pantalla un texto. La combinación <code>\n</code> se interpreta como un salto de línea. El punto y coma especifica el fin de la sentencia.
<code>return 0;</code>	La función main() devuelve 0 al sistema operativo indicando que todo ha sido correcto.
<code>}</code>	Con esta llave se concluye la función main() y también el programa.

Raíces cuadradas

Ahora se presenta un programa más complejo, pero más útil. Se trata de invocar el nombre del programa con un parámetro. Deberán aparecer los números naturales y sus raíces cuadradas comenzando por el 1 y terminando por el parámetro. Por ejemplo, con la orden `./raices 4` se obtendrá el resultado de la derecha.

1	->	1.0000
2	->	1.4142
3	->	1.7321
4	->	2.0000

Código

A continuación se ve el código del programa, acompañado de unos comentarios:

```
/* Cálculo de raíces cuadradas */
#include <stdio.h> /* printf() */
#include <stdlib.h> /* atoi() */
#include <math.h> /* sqrt() */

int main (int argc, char *argv[])
{
    int i, Max;

    if ( argc != 2 )
        printf ("Uso: raices Número\n");

    else
    {
        Max = atoi (argv[1]);
        for ( i=1 ; i<=Max ; i++ )
            printf ("%d -> %.4f\n", i,
                    sqrt(i));
    }

    return 0;
}
```

- ◆ Las tres primeras líneas son órdenes al precompilador para incluir los ficheros de cabecera necesarios para definir las funciones que se indica en los comentarios.
- ◆ Se han dejado líneas en blanco para mejorar la legibilidad y la línea que incluye la llamada a **printf()** está dividida en dos, para acomodar el código al cuadro.
- ◆ La función **main()** admite dos parámetros: **argc** es el número de parámetros con que se invoca el programa y **argv** son sus textos.
- ◆ Además de las variables **argc** y **argv**, el programa declara las variables **i** y **Max**.
- ◆ Si el número de parámetros no es exactamente dos (el nombre del programa y el número), el programa emite un sencillo mensaje de ayuda al usuario.
- ◆ Si el número de parámetros es dos (numerados 0 y 1), se convierte el 1 en un entero usando la función **atoi()** y mediante un bucle **for** se van imprimiendo en pantalla los números y sus raíces cuadradas.

Compilación

Como en el programa se usa la función **sqrt()**, es necesario compilar añadiendo la librería matemática; por tanto, la orden correcta para compilar el programa es

```
gcc -o raices raices.c -lm
```

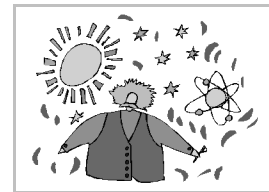
Ejecución

Es posible ejecutar el programa variando el parámetro que se le pase, para ir viendo cómo cambia la salida del programa; por ejemplo, `./raices 100`, `./raices 200`, etc.

Algunas ideas importantes

Después de estos dos ejemplos se pueden resaltar algunas ideas básicas:

- ◆ Las líneas que debe atender el precompilador comienzan con el carácter **#**.
- ◆ Las sentencias deben finalizar con punto y coma, pero se pueden escribir con formato libre.
- ◆ Los bloques de sentencias se encierran entre llaves.
- ◆ La ejecución del programa siempre comienza en la función **main()**, que puede recibir parámetros desde la línea de órdenes del sistema operativo.
- ◆ Existen multitud de funciones disponibles que realizan tareas útiles.
- ◆ Para usar una variable, antes hay que declararla.



Elementos fundamentales

Palabras reservadas

En un lenguaje de programación se llaman palabras reservadas a aquellas que no puede usar libremente el programador, puesto que tienen significados específicos en el lenguaje. C es uno de los lenguajes que tiene menos palabras reservadas. Son éstas (siempre en minúsculas):

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

Hay que añadir también las palabras reservadas por las librerías, así como las que incorporan algunos compiladores para extender las capacidades del lenguaje.

Comentarios

En C se introducen comentarios en el código comenzando con `/*` y terminando por `*/`. Los comentarios pueden ocupar varias líneas y situarse en cualquier lugar respecto al resto del código. No está permitido *anidar* comentarios, es decir, escribir comentarios dentro de otros comentarios.

Sentencias

Son la materia prima de un programa. En principio, cada instrucción que se escribe es una sentencia. Todas las sentencias simples deben acabar en punto y coma. Se admite la sentencia vacía: sólo el punto y coma. Hay sentencias compuestas, que consisten en varias sentencias simples encerradas entre llaves.

Identificadores

Los identificadores son las palabras que introduce el programador para referirse a los distintos elementos que necesita: variables, funciones, estructuras, etc. Los identificadores pueden contener letras (del inglés), dígitos y caracteres de subrayado, pero el primer carácter debe ser una letra. Serán tenidos en cuenta los 31 primeros caracteres, aunque se pueden usar más. El C distingue las mayúsculas de las minúsculas, de modo que los identificadores **Fecha**, **fecha** y **FECHA** son diferentes. Un identificador nunca puede coincidir con una palabra reservada.

Tipos de datos

Hay cuatro tipos básicos de datos en C, aunque cada uno de ellos admite variantes.

- ◆ Tipo **carácter**. Sólo admite la variante **char**. Se utiliza para designar caracteres, aunque internamente se consideran números enteros de 8 bits. Los caracteres se deben escribir entre comillas simples. Ejemplos: `'A'`, `'c'`.
- ◆ Tipo **entero**, denominado **int**, pero que admite cuatro variantes. Es útil para describir números enteros de 16 ó 32 bits. Ejemplos: `23`, `-14`.
- ◆ Tipo **coma flotante**, con las variantes **float** y **double**. Se usa para representar números con decimales. Es obligatorio escribirlos con un punto decimal. Ejemplos: `0.12`, `3.0`, `-0.003`.
- ◆ Tipo **vacío**, llamado **void**. es útil tanto para representar la ausencia de tipo como para hacer manipulaciones avanzadas de tipos.

Constantes

Las constantes se representan cada una según su tipo de dato, tal como se ha explicado más arriba; pero, a efectos prácticos, para representar constantes se suele utilizar el precompilador. Por ejemplo, para designar con el identificador **PULGADA** el número en coma flotante `2.54` se escribe:

```
#define PULGADA 2.54
```

Así, el precompilador sustituirá las referencias a **PULGADA** por el número `2.54`. Esto recibe el nombre de **macro constante**. Es costumbre escribir los macros constantes en mayúsculas, aunque no es obligatorio.

Variables

Son zonas de memoria RAM reservadas por el compilador para almacenar un valor. Las variables se nombran con un identificador. Siempre se deben declarar antes de usar. Su valor puede cambiar en cualquier momento que decida el programador.

Tipos de variables

Según dónde se declaren, existen dos tipos de variables:

- ◆ Variables **locales**. Se declaran dentro de una función, antes de cualquier otro tipo de operación. Las variables locales sólo se pueden utilizar dentro de la función en que se definen, y desaparecen cuando el flujo del programa sale de ella.
- ◆ Variables **globales**. Se declaran fuera de todas las funciones del programa, y son accesibles desde todas ellas. En general, se desaconseja su uso.

Declaración de variables

En cada sentencia se escribe el tipo de las variables y a continuación la lista de variables de ese tipo, separadas por comas. Ejemplo:

```
int    i, j, k;        /* Tres índices          */
float  Inicio, Fin;    /* Valores inicial y final */
char   Respuesta;     /* La letra que se contesta */
```

Operadores y expresiones

Los operadores son símbolos que representan operaciones matemáticas. Por ser éste un curso elemental, de simple iniciación, sólo se explicarán los operadores más sencillos, en C existen más. Las expresiones son combinaciones de constantes, variables y operadores.

Operadores aritméticos

A la derecha se ve una tabla con siete operadores aritméticos y sus significados. Se usan según las reglas habituales del álgebra, y por supuesto se pueden añadir paréntesis donde sea necesario.

Operador	Signo
Suma	+
Diferencia	−
Producto	*
Cociente	/
Módulo	%
Incremento	++
Decremento	--

Operador de asignación

Se representa con el signo =. Permite asignar a una variable, que se pondrá a la izquierda, el valor de una expresión, que estará a la derecha. Ejemplos:

```
Inicio = 8 * PULGADA + 0.34; /* PULGADA es un macro          */
Fin = (i+j) / 2.0;           /* Fin es la media de i y j      */
k = j++;                     /* Se asigna j a k y luego se incrementa */
```

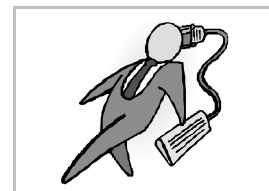
Operadores lógicos

Una expresión con estos operadores podrá ser verdadera (valor 1) o falsa (valor 0). Por tanto, se suelen utilizar para escribir condiciones y a partir de ellas tomar decisiones. Se ven aquí:

Operador	Significado	Operador	Significado	Operador	Significado
<	Menor	>=	Mayor o igual	&&	Y
<=	Menor o igual	==	Igual		O
>	Mayor	!=	Distinto	!	No

Ejemplos:

```
i<j || k>0 /* i es menor que j o k es positivo */
a!=2 && a!=7 /* a es distinto de 2 y distinto de 7 */
```



Entrada y salida

La interfaz de usuario

La creación de interfaces de usuario es una materia compleja, sobre la que existe mucha literatura técnica. Sin embargo, en estas hojas no se hará hincapié en este aspecto de la programación, para concentrarse en el proceso de transformación de datos.

Ahora se muestran las tres maneras más sencillas de relacionarse con el usuario: recibir parámetros por la línea de órdenes, escribir todo tipo de datos en la pantalla y recibir datos del teclado.

Parámetros

Cuando en un sistema operativo se ejecuta un programa, siempre se puede acompañar su nombre de más texto. El nombre del programa y el texto adicional reciben el nombre de parámetros.

En C los parámetros se numeran a partir de 0 y se reciben como cadenas de texto, pero se pueden convertir en números usando las funciones `atoi()`, `atof()` y similares.

Ejemplo

Si se invoca un programa con la línea `Mi Prog -p -a Mi Fi ch. c -m 4`, los seis parámetros que recibe el programa en C son:

Parámetro 0: "MiProg"; parámetro 1: "-p"; parámetro 2: "-a"; parámetro 3: "MiFich.c"; parámetro 4: "-m"; parámetro 5: "4".

Definición de `main()`

Para poder usar los parámetros dentro del programa hay que declarar la función `main()` de modo que su primer parámetro sea el número de parámetros y su segundo parámetro sea el conjunto de cadenas de texto de los parámetros. Es costumbre llamar `argc` (del inglés *argument counter*) al primer argumento de `main()` y `argv` (del inglés *argument values*) al segundo, pero se les puede poner otros nombres.

Un programa

El pequeño programa que se ve a la derecha escribe en pantalla todos los parámetros que recibe.

```
/* Impresión de parámetros */
#include <stdio.h> /* printf() */
int main (int argc, char *argv[])
{
    int i;

    for ( i=0 ; i<argc ; i++ )
        printf ("Parámetro %d: %s\n",
                i, argv[i]);

    return 0;
}
```

La función "`printf()`"

Es una función fundamental para presentar información en pantalla, ya que es muy versátil. Permite representar todo de tipo de datos, con muchas opciones.

El primer parámetro de la función es la cadena de formato. En ella se escribe cómo se desea que aparezcan en pantalla el resto de los parámetros. Todo el texto de la cadena de formato se escribirá en pantalla tal cual aparezca, salvo los códigos reservados, que se usan para indicar el tipo de las variables y sus modos de representación. Los restantes parámetros, tantos como sea necesario, son las variables que se imprimirán en el lugar de los códigos en la cadena de formato.

```
/* Ejemplo de printf() */
#include <stdio.h> /* printf() */
int main (void)
{
    float a;
    int b;
    char c;

    a = 1.0 / 4.0;
    b = 3 * 4;
    c = 'G';

    printf ("%s a = %f. b = %d. c = %c\n",
            "Resultado:", a, b, c);

    return 0;
}
```

Los códigos

Existen muchos códigos disponibles, y con muchas variantes, que permiten un ajuste muy fino. Éstos son los códigos más importantes:

- ♦ **%d** Sirve para imprimir un número entero.
- ♦ **%f** Para imprimir un número en coma flotante.
- ♦ **%c** Para imprimir un carácter.
- ♦ **%s** Para imprimir una cadena.

Ejemplo

Cuando se ejecuta el programa de más arriba se obtiene esta salida por pantalla:

Resultado: a = 0.250000. b = 12. c = H

La función “scanf()”

Esta función lee desde el teclado las entradas del usuario y las asigna a las variables que se indique. Se suele usar para leer una variable cada vez.

El primer parámetro de **scanf()** es una cadena de formato, en la que especifica el tipo de variable que se va a leer usando los mismos códigos explicados para **printf()**. El segundo parámetro es la dirección de memoria donde se está almacenando la variable. Para obtener la dirección de memoria de una variable se antepone a su nombre el carácter **&**.

Un ejemplo

El siguiente programa pide al usuario un número entero y luego escribe su cuadrado:

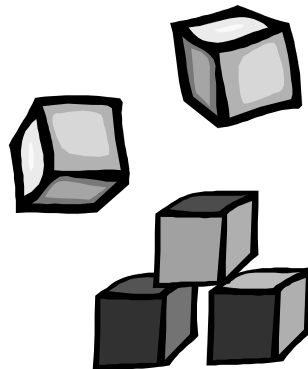
```
#include <stdio.h> /* printf() scanf() */

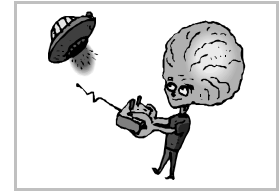
int main (void)
{
    int Numero, Cuadrado;

    printf ("Escribe un número: ");
    scanf ("%d", &Numero);

    Cuadrado = Numero * Numero;
    printf ("Su cuadrado es %d\n", Cuadrado);

    return 0;
}
```





Sentencias de control (1)

Para qué sirven

Según se ha visto hasta el momento, los programas en C se ejecutan comenzando por la primera instrucción, siempre de la función **main()**, pasan a la siguiente y así sucesivamente hasta la última. Pero un programa real no se puede escribir sólo con eso. Es necesario tomar decisiones que lleven la ejecución por uno u otro camino, y también es necesario repetir una secuencia de instrucciones gran número de veces. Éste es el cometido de las sentencias de control. Las tienen todos los lenguajes de programación, aunque siempre con variantes de unos a otros.

La sentencia if

La estructura general de la sentencia **if** es ésta:

```
if ( Condición )
    Sentencia1
else
    Sentencia2
```

Condición es una condición lógica, que, por tanto, puede ser cierta o falsa. **Sentencia1** es la sentencia que se ejecutará si **Condición** es cierta. **else** indica que si **Condición** es falsa, habrá que ejecutar **Sentencia2**. La parte else es optativa. Tanto **Sentencia1** como **Sentencia2** pueden ser sentencias simples o compuestas.

Ejemplo

El siguiente programa pide un número, que puede tener decimales, al usuario; si el número es positivo o cero, escribe su raíz cuadrada y si es negativo escribe un mensaje.

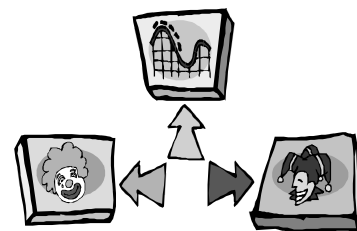
```
#include <stdio.h>    /* printf() scanf() */
#include <math.h>      /* sqrt() */

int main (void)
{
    float Numero, Raiz;

    printf ("Escribe un número: ");
    scanf ("%f", &Numero);

    if ( Numero >= 0 )
    {
        Raiz = sqrt (Numero);
        printf ("Su raíz es %f\n", Raiz);
    }
    else
        printf ("No tiene raíz cuadrada\n");

    return 0;
}
```



Sentencias if anidadas

Es perfectamente posible, y habitual, incluir sentencias **if** dentro de otras sentencias **if**. En C hay un modo peculiar de hacerlo, pero no se va a explicar por no ser un concepto general.

La sentencia switch

Se suele ver esta sentencia como una especie de **if** ampliado. Permite ejecutar diferentes sentencias según el valor que tome una variable. Su estructura general es ésta:

```
switch ( Variable )
{
    case Valor1: Sentencias1; break;
    case Valor2: Sentencias2; break;
    ...
    default: Sentencias;
}
```

Variable es el nombre de una variable, que casi siempre es de tipo entero o carácter. **Valor1**, **Valor2**, etc. son valores constantes, llamados etiquetas, del mismo tipo que **Variable**. **Sentencias1** son las sentencias que se deben ejecutar si **Variable** presenta el **Valor1**. **default** indica que si **Variable** no toma ninguno de los valores indicados, se deberán ejecutar **Sentencias**. La palabra **break** es simple opcional, pero si en un caso no se pone, la ejecución continuará en la siguiente etiqueta.

Ejemplo

El siguiente programa pide una vocal al usuario, e imprime una palabra que comienza por ella; si el usuario no introduce una vocal, el programa imprime un mensaje.

```
#include <stdio.h>    /* printf() scanf() */

int main (void)
{
    char  Caracter;

    printf ("Escribe una vocal: ");
    scanf ("%c", &Caracter);

    switch ( Caracter )
    {
        case 'a': printf ("Alemania"); break;
        case 'e': printf ("Estonia");  break;

        case 'i': printf ("Indonesia"); break;
        case 'o': printf ("Omán");      break;
        case 'u': printf ("Uganda");    break;
        default: printf ("No es una vocal");
    }

    printf ("\n");

    return 0;
}
```





Sentencias de control (2)

La sentencia while

Esta sentencia permite la repetición de una sentencia mientras una condición sea verdadera. Así que ésta es su estructura:

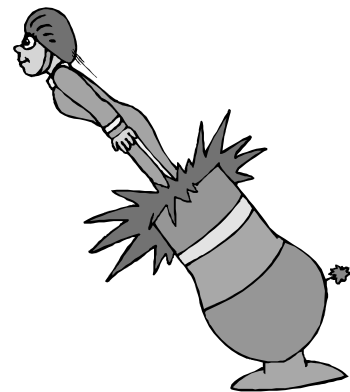
```
while ( Condición )  
    Sentencia
```

Condición es una condición lógica y **Sentencia** puede ser una sentencia simple, compuesta o vacía. Primeramente se evalúa **Condición**; si es cierta, se ejecuta **Sentencia** y se vuelve a evaluar **Condición**, siguiendo de ese modo hasta que **Condición** sea falsa. Pero si **Condición** es falsa la primera vez, **Sentencia** no se ejecuta nunca.

Ejemplo

El siguiente programa pide un número al usuario y le va calculando la raíz cuadrada mientras su diferencia con 1 sea mayor que una milésima.

```
#include <stdio.h>    /* printf() scanf() */  
#include <math.h>     /* sqrt() */  
  
int main (void)  
{  
  
    int  Numero;  
    float Raiz;  
  
    printf ("Escribe un número: ");  
    scanf ("%d", &Numero);  
    Raiz = Numero;  
  
    while ( Raiz-1 >= 0.001 )  
    {  
        Raiz = sqrt (Raiz);  
        printf ("Raíz: %f\n", Raiz);  
    }  
  
    return 0;  
}
```



La sentencia do

Esta sentencia es muy similar a la sentencia **while**. La diferencia es que la condición se evalúa después de ejecutar la sentencia, con lo que se garantiza que ésta se ejecutará al menos una vez, mientras que en una sentencia **while** es perfectamente posible que la sentencia no se ejecute ni una sola vez. Ésta es la estructura de la sentencia **do**:

```
do Sentencia  
while ( Condición );
```

Ejemplo

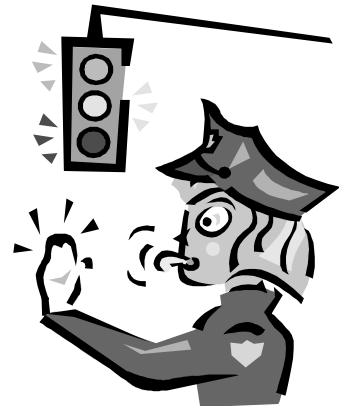
El programa que se muestra a continuación pide números al usuario y va mostrando sus cuadrados hasta que el usuario introduce el número 0.

```
#include <stdio.h>  /* printf() scanf() */

int main (void)
{
    int Numero, Cuadrado;

    do
    {
        printf ("Escribe un número: ");
        scanf ("%d", &Numero);
        Cuadrado = Numero * Numero;
        printf ("Su cuadrado es %d\n", Cuadrado);
    } while ( Numero != 0 );

    return 0;
}
```



La sentencia for

Esta es la sentencia que en los lenguajes de programación permite repetir un número determinado de veces una porción del código. En C es especialmente potente, y permite gran expresividad. Su formato general es éste:

```
for ( SentenciaInicial ; Condición ; SentenciaIncremento )
    Sentencia
```

Y se ejecuta de esta manera:

1. Se ejecuta **SentenciaInicial**.
2. Se comprueba **Condición**.
3. Si **Condición** es cierta, se ejecutan **Sentencia** y **SentenciaIncremento** (por ese orden) y se vuelve al paso 2.
4. Si **Condición** es falsa, concluye la ejecución de la sentencia **for**.

Ejemplo

El programa que aparece ahora pide al usuario dos números y muestra los cuadrados de todos los números que se encuentran entre ellos, ambos incluidos.

```
#include <stdio.h>  /* printf() scanf() */

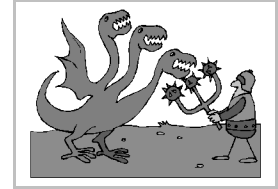
int main (void)
{
    int i, NumeroInicial, NumeroFinal, Cuadrado;

    printf ("Escribe el primer número: ");
    scanf ("%d", &NumeroInicial);
    printf ("Escribe el segundo número: ");
    scanf ("%d", &NumeroFinal);

    for ( i = NumeroInicial ; i <= NumeroFinal ; i++ )
    {
        Cuadrado = i * i;
        printf ("El cuadrado de %d es %d\n", i, Cuadrado);
    }

    return 0;
}
```





Arrays

Qué son los arrays

Con esta palabra inglesa se designa a un conjunto de variables que tienen el mismo nombre. Para acceder a cada variable individual se utiliza uno más números llamados **índices**. El número de índices necesario se llama **dimensión** del array.

Son muy útiles en gran variedad de problemas, ya que es muy habitual tener que describir una situación para muchos individuos simultáneamente.

La palabra *array* se puede traducir, en este contexto, por *vector* o por *matriz*, pero cualquiera de las dos traducciones resulta confusa. Se ha impuesto en la literatura técnica en castellano el uso directo de la palabra inglesa.

Declaración y uso

Los arrays se declaran como las demás variables, simplemente hay que poner tras el nombre de la variable el máximo número de elementos que tendrá cada dimensión.

El ejemplo más sencillo es la declaración `int Coordenada[3];`, que declara un array llamado **Coordenada**, que contendrá tres variables enteras. En C los índices siempre comienzan en 0, de modo que las tres variables son **Coordenada[0]**, **Coordenada[1]** y **Coordenada[2]**.

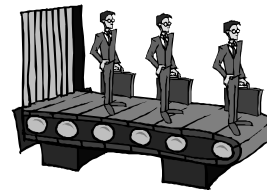
Para declarar y usar variables con más de una dimensión hay que usar corchetes adicionales para cada dimensión. Por ejemplo, la declaración `float Punto[4][3];` permite usar doce variables llamadas **Punto**, a las que se accede desde **Punto[0][0]** a **Punto[3][2]**.

Lo más habitual cuando se usan arrays es hacer cálculos con ellos dentro de sentencias `for`. Por ejemplo, para multiplicar por 2 todas las variables **Coordenada** definidas antes se puede usar este fragmento de código:

```
int i;
for ( i=0 ; i<3 ; i++)
    Coordenada[i] = 2 * Coordenada[i];
```

Y para multiplicar por 2 todas las variables **Punto**, hay que usar dos sentencias `for` anidadas, de este modo:

```
int i, j;
for ( i=0 ; i<4 ; i++)
{
    for ( j=0 ; j<3 ; j++)
        Punto[i][j] = 2 * Punto[i][j];
}
```



Ejemplo

¿Cómo se puede representar en C el tablero del juego de los barquitos? Mediante un array de dos dimensiones con números enteros. Cada variable tendrá un número que indique agua o el tipo de barco, por ejemplo, mediante los números 0, 1, 2, 3 y 4. Un tablero de 8×8 se representaría así:

```
#define DIMENSION 8
int Tablero[DIMENSION][DIMENSION];
```

Y para imprimir por pantalla el tablero se puede usar este código:

```
int i, j;
for ( i=0 ; i<DIMENSION ; i++)
{
    for ( j=0 ; j<DIMENSION ; j++) printf ("%d ", Tablero[i][j]);
    printf ("\n");
}
```



Cadenas

Qué son

Una cadena es un conjunto ordenado de caracteres, el modo de manejar texto en programación. Muchos lenguajes contemplan las cadenas como un tipo de datos básico, y ofrecen métodos específicos para manejarlas.

Sin embargo, en C las cadenas son simplemente arrays de tipo **char**. Eso no quiere decir que manejar cadenas en C sea más difícil que con otros lenguajes. Simplemente, es distinto.

Disposición en memoria

En C las cadenas siempre tienen un carácter más de los que se ven: el carácter que tiene el número 0 en la tabla de caracteres, llamado **carácter nulo** o **NULL**, que se representa `'\0'` (no hay que confundirlo con el carácter que representa al numeral 0, que es el número 48). Así pues, la cadena "Hola" realmente se representa en memoria como `H o l a \0`, ocupando cinco octetos, y no cuatro como podría parecer más normal.

Tipos de cadenas

En C se manejan cadenas de dos tipos diferentes: **constantes** y **variables**. Las primeras no se pueden modificar, y el compilador reserva automáticamente memoria para ellas. Las segundas son modificables, pero el programador debe reservar memoria suficiente para almacenarlas.

Los dos tipos se pueden manejar simultáneamente utilizando **punteros**, pero esa técnica no se va a explicar en este curso, por ser una característica avanzada de C y no ser suficientemente general.

Definición de cadenas constantes

Basta escribir el texto entre comillas dobles. Por ejemplo: `"Esto es una cadena"`.

Definición de cadenas variables

El modo más sencillo es declarar una variable como array de **char**, recordando que hay que reservar un lugar extra para el carácter nulo. Ejemplo: `char Cadena[11]`

Funciones que manejan cadenas

Existen muchas funciones de librería que ayudan en el manejo de cadenas, la mayor parte se declaran en el fichero de cabecera **string.h**. Por ejemplo, la función **strlen()** da la longitud de la cadena sin contar el carácter nulo, la función **strcpy()** copia una cadena en otra, etc.

Ejemplo

El siguiente programa pide una cadena al usuario y la muestra invertida:

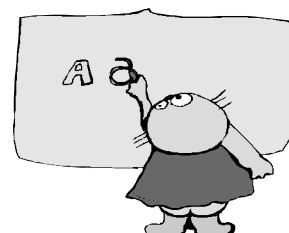
```
#include <stdio.h> /* printf() gets() */
#include <string.h> /* strlen() */

int main (void)
{
    char Original[11], Invertida[11];
    int Longitud, i;

    printf ("Escribe una cadena (máximo 10 caracteres): ");
    gets (Original); /* Recibe la cadena desde el teclado */

    Longitud = strlen (Original);
    for ( i=0 ; i<Longitud ; i++ )
        Invertida[i] = Original[Longitud-1-i];
    Invertida[Longitud] = '\0';

    printf ("La cadena invertida es %s\n", Invertida);
    return 0;
}
```





Funciones

Para qué sirven

Hasta el momento todos los programas que se han presentado constan de una única función, la función `main()`. En principio, sería posible escribir todo el código del programa dentro de ella, pero en la práctica pronto se descubre que hay fragmentos complicados de código que hay que repetir muy a menudo. Por tanto, es mejor escribir el fragmento que se repite una sola vez, e invocar su funcionamiento donde sea necesario. Ese fragmento, ahora independiente del flujo general del programa, es lo que se llama **función**.

Las funciones pueden devolver un resultado o no. En otros lenguajes de programación se llama **procedimientos** a las funciones que no devuelven valores. En C no hay tal distinción.

Manejo general

Se distinguen tres fases en la utilización de una función:

1. **Declaración de la función.** Se especifica qué tipos de datos recibirá la función y qué tipo devolverá. Las funciones se declaran escribiendo su **prototipo** antes de usarla por primera vez.
2. **Definición de la función.** En cualquier parte del código, siempre después de la declaración, se define la función escribiendo las sentencias que la forman.
3. **Invocación de la función.** Cuando se quiere activar la función basta escribir su nombre y especificar los datos con los que debe trabajar.

Declaración

Los valores que reciben las funciones para comenzar su trabajo se llaman **parámetros**. Pueden ser desde ninguno hasta una cantidad variable. En el prototipo de la función hay que escribir el tipo de dato que devolverá, el nombre de la función y la lista de los tipos de datos de todos los parámetros, separados por comas; opcionalmente, se pueden escribir los nombres de los parámetros. Si la función no devuelve ningún valor, se especifica el tipo **void**; y si no recibe ningún parámetro, la lista se sustituye por **void**.

Ejemplo

El prototipo de una función llamada **Lineal** que reciba tres parámetros **float** y devuelva un resultado **float** es éste:

```
float Lineal (float, float, float);
```

Definición

La definición de la función se realiza escribiendo el tipo de dato que devolverá, el nombre de la función y la lista de los tipos de datos y nombres de todos los parámetros, separados por comas; tras esto, entre llaves, se escribe el código de la función.

Si la función devuelve un valor, se usa una sentencia con **return** y el valor devuelto. Es posible usar varios **return** en una misma función, aunque se desaconseja hacerlo.

Ejemplo

Para definir una función que devuelva el resultado de una función lineal $y=ax+b$, para cada valor que se le dé de a , b y x , se escribe esto:

```
float Lineal (float a, float b, float x);  
{  
    return a * x + b;  
}
```

Invocación

Esta es la parte más sencilla: basta escribir el nombre de la función y los valores de los parámetros (si los hay), separados por comas. Si se desea usar el valor devuelto por la función, puede asignarse a una variable o usarlo en otra función; no es obligatorio usar el valor devuelto.

Ejemplo

Para asignar a la variable `y` el valor calculado por la función `Lineal` para $a=2$, $b=1.5$ y $x=0.3$ se escribe esta sentencia:

```
y = Lineal (2.0, 1.5, 0.3);
```

Un ejemplo completo

Para ver la utilidad de las funciones ahora se va a desarrollar un pequeño programa de ejemplo. El programa permite pasar de grados Celsius a grados Fahrenheit y viceversa. El programa se llamará `ConvCF` y se invocará `ConvCF x c` para convertir x grados Celsius a Fahrenheit y `ConvCF x f` para convertir x grados Fahrenheit a Celsius.

```
#include <stdio.h> /* printf() */
#include <stdlib.h> /* atof() */

float Lineal (float, float, float);

int main (int argc, char *argv[])
{
    float Dato, Resultado;
    char Orden;

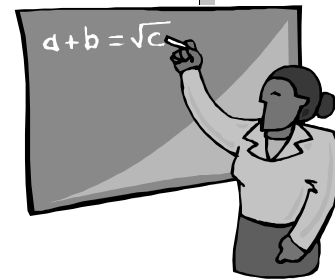
    Dato = atof (argv[1]);
    Orden = argv[2][0];

    if ( Orden == 'c' )
    {
        Resultado = Lineal (1.8, 32.0, Dato);
        printf (".2f grados Celsius son %.2f grados Fahrenheit.\n",
            Dato, Resultado);
    }

    if ( Orden == 'f' )
    {
        Resultado = Lineal (0.555556, -17.7778, Dato);
        printf (".2f grados Fahrenheit son %.2f grados Celsius.\n",
            Dato, Resultado);
    }

    return 0;
}

float Lineal (float a, float b, float x)
{
    return a * x + b;
}
```





Estilo de programación

Necesidad de un buen estilo

Conocer los detalles técnicos de uno o más lenguajes de programación no es suficiente garantía de crear buen código. El código de un programa no debe verse sólo como un conjunto de instrucciones, sino casi como literatura: está hecho por humanos y lo van a mantener (modificar cuando se precise) humanos. Por tanto, hay que escribirlo pensando en las personas y no en los compiladores. Éstos harán su trabajo para convertir el código fuente en ejecutable sin importarles cómo está escrito, bastará que sea correcto técnicamente; pero los humanos tendrán que leer el código y entender qué hace (y por qué) en el menor tiempo posible.



Así pues, es muy importante conocer unas pocas normas que orienten en la escritura de buen código. Para un principiante puede ser sorprendente que se le dé tanta importancia a un aspecto que luego no tendrá repercusión en el funcionamiento del programa, pero la experiencia de más de cincuenta años escribiendo programas ha llevado a los técnicos en la materia a reconocer que es un aspecto fundamental: a lo largo de la vida de un programa, la labor que más tiempo consume es su modificación, por eso es fundamental que esté bien escrito.



Escritura del código

Hay que intentar escribir el código de modo que se entienda qué hace en el menor tiempo posible; debe tener físicamente buen aspecto, para facilitar la lectura: ni todo muy junto ni todo muy separado. Algunos consejos más concretos:

- ♦ **Líneas en blanco.** Hay que usar líneas en blanco para separar en el código las partes que sean lógicamente diferentes. Por ejemplo, separar la declaración de variables de una función y el código con instrucciones de ejecución.
- ♦ **Longitud de las líneas.** Nunca superior a 80 caracteres, para poder trabajar el código con cualquier editor e imprimirlo en cualquier impresora.
- ♦ **Indentación coherente.** Cuando se utilizan sentencias de control anidadas es fácil perderse y no saber a qué nivel pertenece cada porción de código. Eso se puede solventar indentando (es decir, metiendo un poco hacia la derecha) un poco más cada nivel respecto al anterior. Pero hay que hacerlo de forma coherente, siempre igual. Las llaves tan utilizadas en C deben colocarse siempre de la misma forma. Hay varios estilos de hacerlo: cada programador debe elegir el suyo y seguirlo fielmente.
- ♦ **Nombres de las variables.** Deben ser cortos pero descriptivos. Conviene pensarlos bien y no poner el primero que aparezca en la imaginación. Como se pueden mezclar mayúsculas y minúsculas en los nombres, hay que hacerlo coherentemente.



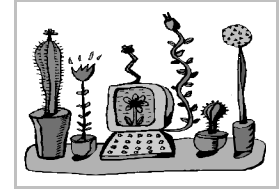
Introducción de comentarios

En muchas ocasiones el código solo es difícil de entender, aunque esté bien escrito. Entonces hay que añadir comentarios que faciliten la comprensión. Otras veces se utilizan comentarios simplemente para añadir información extra al código, como las partes de que se compone el fichero, quién lo ha escrito, etc. Consejos más precisos:

- ♦ En los comentarios debe escribirse qué hace el código, no cómo lo hace, puesto que eso es justo lo que ya se ve.
- ♦ Aunque se escriban comentarios, hay que seguir procurando que el código sea legible en sí mismo.
- ♦ No deben entorpecer la lectura del código, sino mejorarla.

- ◆ Deben escribirse con coherencia, no unas partes con muchos comentarios y otras sin ninguno.
- ◆ Sólo hay que escribir comentarios que sean realmente útiles y correctos: el que no es útil, molesta; el que no es correcto, equivoca.
- ◆ Conviene añadir comentarios físicamente evidentes para distinguir con más facilidad las distintas partes de un programa: ficheros de cabecera, declaración de funciones, etc.
- ◆ Es imprescindible que haya comentarios con los datos básicos del programa: autor, fecha de creación, explicaciones generales, etc.
- ◆ Cada función debe llevar un comentario que la anteceda, especificando sus entradas, salidas, y cualquier otro dato que sea pertinente.





Ciclo de vida

Ciclo de vida de un programa

Se llama así al conjunto de fases por las que pasa un programa, desde que no existe hasta que deja de usarse. Para crear un buen programa es imprescindible conocer estas fases, y respetarlas. Por supuesto, para hacer un programa muy pequeño no es necesario ser tan estricto, puesto que los programas pequeños apenas requieren esfuerzo, pero cuando los proyectos de programación van siendo mayores, enseguida se nota la necesidad de una buena planificación.

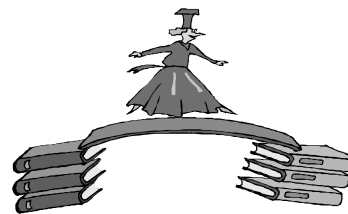
Nunca se debe pasar a la siguiente fase sin haber concluido la anterior. Se ahorra mucho tiempo trabajando así, porque, aunque el software es muy flexible, modificarlo cuando está escrito es muchísimo más complicado que hacerlo cuando todavía no se ha hecho.

No todas las fases se realizan en el ordenador: de hecho, las partes más importantes se llevan a cabo con ordenadores apagados. Normalmente se piensa que en informática se comienza cualquier tarea encendiendo el ordenador, sin embargo en programación se comienza apagándolo.



Fases del ciclo

- ◆ Análisis de requerimientos.
- ◆ Diseño.
- ◆ Codificación.
- ◆ Depuración.
- ◆ Explotación.



Análisis de requerimientos

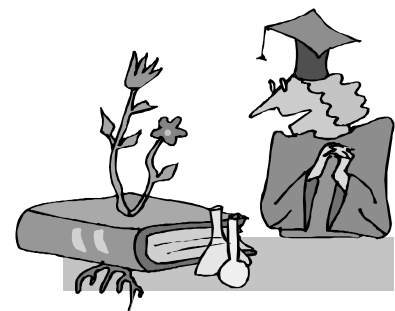
Para poder escribir un programa antes hay que saber qué se espera de él, cuál será su cometido. Por tanto, hay que comenzar por estudiar qué se desea exactamente que haga el programa. Esto se llama analizar los requerimientos, y en casos reales lo realizan los analistas de sistemas.

Hay que llegar a definir con la mayor exactitud posible qué entradas recibirá el programa, y qué salidas se espera de él.

Diseño

A partir del análisis de requerimientos se trabaja en el diseño del programa, pensando en qué partes habrá que dividirlo, así como qué estructuras serán necesarias para poder llevar a cabo todo lo que se pide.

Hay que procurar que haya partes claramente distinguibles, con responsabilidades bien definidas; se implementarán en funciones independientes. Se procura usar pocas variables globales, porque suelen ser fuente de errores.



Codificación

Si el programa está bien diseñado, codificarlo es bastante sencillo; pero si el diseño es pobre y se han pasado por alto muchos detalles, la codificación puede consumir mucho tiempo.

Hay que dividir el código en funciones pequeñas, para que sea fácil de entender y modificar. Cada función debe hacer una sola cosa.

Depuración

Mientras se va escribiendo el código, hay que ir comprobando si funciona bien. No hay que esperar a que esté todo escrito, porque entonces habrá demasiados errores. Hay que ir detectando errores y corrigiéndolos.

Esta etapa se llama en inglés *debug*, “eliminar bichos”, ya que en uno de los primeros ordenadores un error estaba causado por un insecto que se había quedado enredado en los cables.

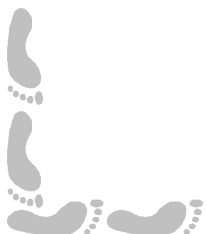
Explotación

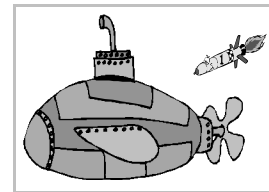
Esta es la fase en la que se usa el programa. Es habitual que durante esta fase aún se detecten más errores y haya que arreglarlos o bien tomar nota de ellos para corregirlos en la siguiente versión del programa.

Nuevas versiones

Cuando se decide que aparezca la nueva versión de un programa, el ciclo de vida comienza otra vez, pero con una particularidad: la mayor parte del código anterior se conserva o se adapta.

Se calcula que la mayoría del tiempo de programación se pasa retocando código, bien para arreglarlo o para mejorarlo; esto demuestra la importancia de prestar mucha atención a una codificación con buen estilo.





Un ejemplo completo (1)

Objetivo

En esta última hoja se presenta la creación de un programa completo. Aunque el programa es sencillo, lo que se pretende es que se produzca una toma de conciencia de la necesidad de seguir las fases del ciclo de vida del programa.

Caza submarina

Es un juego muy sencillo que consiste en disparar a un submarino que se desplaza por un mar representado por una cuadrícula. El jugador disparará diciendo las coordenadas del punto de impacto, el programa responderá a qué distancia del submarino ha caído el disparo, y el submarino a continuación se moverá un cuadro. El juego termina cuando se impacta sobre el submarino o cuando el jugador se queda sin disparos.

En la figura de la derecha se ve la cuadrícula propuesta. El submarino está en la casilla **D2** y si se dispara a **B3** la distancia será 2.

	A	B	C	D	E
1					
2				✳	
3		🌀			
4					
5					

Análisis de requerimientos

El programa deberá colocar el submarino aleatoriamente en la cuadrícula y moverlo, también al azar, tras cada disparo. Deberá ir pidiendo al jugador las coordenadas de cada disparo y contestar con la distancia al submarino. Por último, debe decir el resultado del juego.

Diseño

Es necesario disponer de dos variables globales para mantener la posición del submarino. Harán falta dos funciones para colocar y mover el submarino, otra para ir dirigiendo el juego y una que calcule la distancia entre el disparo y el submarino. Las dimensiones del mar y el número máximo de disparos se definirán como constantes, para que sea sencillo cambiarlas.

Codificación y depuración

Se eligen los nombres `FilaSubmarino`, `ColumnaSubmarino`, `ColocaSubmarino()`, `MueveSubmarino()`, `Juega()` y `CalculaDistancia()` a las variables y funciones obtenidos en la fase de diseño. Para ir codificando el programa es muy buena idea declarar y definir las funciones necesarias aunque no se disponga todavía de su implementación completa, para poder ir compilando incrementalmente el programa y así poder ir probándolo.

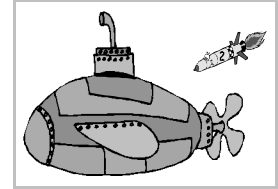
En otra parte de esta hoja se presenta el código resultante de esta fase.

Explotación

Es el momento de usar el programa, en este caso jugando. Fácilmente se puede apreciar que el programa admite muchas mejoras. La más evidente es que el programa no comprueba que el disparo sea válido. Un error sutil es el uso de la función `gets()`, que supone un grave fallo de seguridad. Se propone como ejercicio hacer las modificaciones oportunas para realizar ésta y cualquier otra mejora que resulte de interés.

Código: cazasubmarina.c

```
/*-----  
* Fichero: cazasubmarina.c  
* Objetivo: Jugar a "Caza submarina"  
* Autor: Pedro Reina  
* Fecha: J.20.6.2002  
*-----*/  
  
/*-----  
* Ficheros de cabecera  
*-----*/  
  
#include <stdio.h> /* printf() gets() */  
#include <stdlib.h> /* srand() rand() abs() */  
#include <ctype.h> /* toupper() */  
  
/*-----  
* Definición de macros constantes  
*-----*/  
  
#define DIMENSION 5 /* Filas y columnas del mar */  
#define MAXDISPARO 5 /* Máximo número de disparos */  
  
/*-----  
* Definición de variables globales  
*-----*/  
  
/* Posición del submarino, siempre entre 0 y DIMENSION-1 */  
int FilaSubmarino, ColumnaSubmarino;  
  
/*-----  
* Declaración de funciones  
*-----*/  
  
void ColocaSubmarino (void);  
int Juega (void);  
int CalculaDistancia (int, int);  
void MueveSubmarino (void);  
  
/*-----  
* Programa principal  
*-----*/  
  
/*-----  
* Función: main()  
* Objetivo: Presentar el programa, lanzar el juego y decir el resultado  
* Entradas: Ninguna  
* Salidas: 0, que indica que no hay errores  
*-----*/  
int main (void)  
{  
    int Resultado;  
  
    printf ( "Caza submarina\n" );  
    printf ( "=====\n" );  
  
    ColocaSubmarino();  
    Resultado = Juega();  
  
    if ( Resultado == 0 )  
        printf ("¡Blanco!\n");  
    else  
        printf ("Has perdido.\n");  
  
    return 0;  
}
```



Un ejemplo completo (2)

Código: cazasubmarina.c (continuación)

```
/*-----  
 * Definición de funciones  
 *-----*/  
  
/*-----  
 * Función:  Juega()  
 * Objetivo: Dirigir el juego  
 * Entradas: Ninguna  
 * Salidas:  La distancia del último disparo al submarino  
 *-----*/  
int Juega (void)  
{  
    int  Distancia, Fila, Columna, Disparo;  
    char Coordenadas[3];  
  
    Disparo = 1;  
    do  
    {  
        /* Pedimos las coordenadas del disparo */  
        printf ("Disparo %d: ", Disparo);  
        gets (Coordenadas);  
        Disparo++;  
  
        /* Calculamos el disparo con números entre 0 y DIMENSION-1 */  
        Fila = toupper (Coordenadas[0]) - 'A';  
        Columna = Coordenadas[1] - '1';  
  
        /* Informamos de la distancia */  
        Distancia = CalculaDistancia (Fila, Columna);  
        printf ("Distancia: %d\n", Distancia);  
  
        MueveSubmarino();  
    } while ( Distancia > 0 && Disparo <= MAXDISPARO );  
  
    return Distancia;  
}  
  
/*-----  
 * Función:  ColocaSubmarino()  
 * Objetivo: Colocar el principio del juego el submarino  
 * Entradas: Ninguna, se usan números aleatorios  
 * Salidas:  Cambian las variables globales del submarino  
 *-----*/  
void ColocaSubmarino (void)  
{  
    srand(0); /* Iniciamos el generador de números aleatorios */  
  
    FilaSubmarino = rand() % DIMENSION;  
    ColumnaSubmarino = rand() % DIMENSION;  
}
```

```

/*-----
* Función: MueveSubmarino()
* Objetivo: Mover aleatoriamente el submarino una casilla
* Entradas: Las variables globales
* Salidas: Las variables globales
*-----*/
void MueveSubmarino (void)
{
    int Aleatorio;

    /* Se cambia la fila al azar, abajo, arriba o sin cambio */
    Aleatorio = rand() % 3;
    switch ( Aleatorio )
    {
        case 0: FilaSubmarino++; break;
        case 1: FilaSubmarino--; break;
    }

    /* Se cambia la columna al azar, derecha, izquierda o sin cambio */
    Aleatorio = rand() % 3;
    switch ( Aleatorio )
    {
        case 0: ColumnaSubmarino++; break;
        case 1: ColumnaSubmarino--; break;
    }

    /* Controlamos que el submarino no se salga de límites */
    if ( FilaSubmarino < 0 )
        FilaSubmarino = 0;
    if ( FilaSubmarino >= DIMENSION )
        FilaSubmarino = DIMENSION-1;
    if ( ColumnaSubmarino < 0 )
        ColumnaSubmarino = 0;
    if ( ColumnaSubmarino >= DIMENSION )
        ColumnaSubmarino = DIMENSION-1;
    }

/*-----
* Función: CalculaDistancia()
* Objetivo: Calcular la distancia entre el disparo y el submarino.
*           la distancia es el mayor número entre la diferencia de
*           filas y la diferencia de columnas
* Entradas: La fila y columna del disparo
* Salidas: La distancia
*-----*/
int CalculaDistancia (int Fila, int Columna)
{
    int Distancia, DistanciaHorizontal, DistanciaVertical;

    DistanciaHorizontal = abs (Columna-ColumnaSubmarino);
    DistanciaVertical = abs (Fila-FilaSubmarino);

    if ( DistanciaHorizontal > DistanciaVertical )
        Distancia = DistanciaHorizontal;
    else
        Distancia = DistanciaVertical;

    return Distancia;
}

```