CAPÍTULO 1

© F.J.Ceballos/RA-MA

FASES EN EL DESARROLLO DE UN PROGRAMA

En este capítulo aprenderá lo que es un programa, cómo escribirlo y qué hacer para que el ordenador lo ejecute y muestre los resultados perseguidos. También adquirirá conocimientos generales acerca de los lenguajes de programación utilizados para escribir programas. Después, nos centraremos en un lenguaje de programación específico y objetivo de este libro, *C/C++*, presentando sus antecedentes y marcando la pauta a seguir para realizar una programación estructurada.

QUÉ ES UN PROGRAMA

Probablemente alguna vez haya utilizado un ordenador para escribir un documento o para divertirse con algún juego. Recuerde que en el caso de escribir un documento, primero tuvo que poner en marcha un procesador de textos, y que si quiso divertirse con un juego, lo primero que tuvo que hacer fue poner en marcha el juego. Tanto el procesador de textos como el juego son *programas* de ordenador.

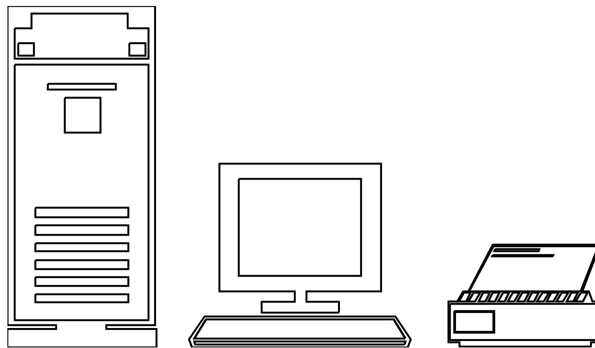
Poner un programa en marcha es sinónimo de ejecutarlo. Cuando ejecutamos un programa, nosotros solo vemos los resultados que produce (el procesador de textos muestra sobre la pantalla el texto que escribimos; el juego visualiza sobre la pantalla las imágenes que se van sucediendo) pero no vemos el guión seguido por el ordenador para conseguir esos resultados. Ese guión es el programa.

Ahora, si nosotros escribimos un programa, entonces sí que sabemos cómo trabaja y por qué trabaja de esa forma. Esto es una forma muy diferente y curiosa de ver un programa de ordenador, lo cual no tiene nada que ver con la experiencia adquirida en la ejecución de distintos programas.

Piense ahora en un juego cualquiera. La pregunta es: ¿qué hacemos si queremos enseñar a otra persona a jugar? Lógicamente le explicamos lo que debe hacer; esto es, los pasos que tiene que seguir. Dicho de otra forma, le damos instrucciones de cómo debe actuar. Esto es lo que hace un programa de ordenador. Un *programa* no es nada más que una serie de instrucciones dadas al ordenador en un lenguaje entendido por él, para decirle exactamente lo que queremos que haga. Si el ordenador no entiende alguna instrucción, lo comunicará generalmente mediante mensajes visualizados en la pantalla.

LENGUAJES DE PROGRAMACIÓN

Un programa tiene que escribirse en un lenguaje entendible por el ordenador. Desde el punto de vista físico, un ordenador es una máquina electrónica. Los elementos físicos (memoria, unidad central de proceso, etc.) de que dispone el ordenador para representar los datos son de tipo binario; esto es, cada elemento puede diferenciar dos estados (dos niveles de voltaje). Cada estado se denomina genéricamente *bit* y se simboliza por *0 o 1*. Por lo tanto, para representar y manipular información numérica, alfabética y alfanumérica se emplean cadenas de *bits*. Según esto, se denomina *byte* a la cantidad de información empleada por un ordenador para representar un carácter; generalmente un *byte* es una cadena de ocho *bits*.



Así, por ejemplo, cuando un programa le dice al ordenador que visualice un mensaje sobre el monitor, o que lo imprima sobre la impresora, las instrucciones correspondientes para llevar a cabo esta acción, para que puedan ser entendibles por el ordenador, tienen que estar almacenadas en la memoria como cadenas de *bits*. Esto hace pensar que escribir un programa utilizando ceros y unos (lenguaje máquina) llevaría mucho tiempo y con muchas posibilidades de cometer errores. Por este motivo, se desarrollaron los lenguajes *ensambladores*.

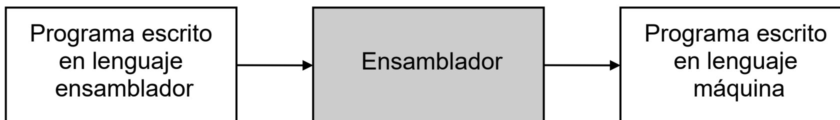
Un lenguaje *ensamblador* utiliza *códigos nemotécnicos* para indicarle al hardware (componentes físicos del ordenador) las operaciones que tiene que reali-

zar. Un código nemotécnico es una palabra o abreviatura fácil de recordar que representa una tarea que debe realizar el procesador del ordenador. Por ejemplo:

```
MOV AH, 4CH
```

El código *MOV* expresa una operación consistente en mover alguna información desde una posición de memoria a otra.

Para traducir un programa escrito en *ensamblador* a lenguaje máquina (código binario), se utiliza un programa llamado *ensamblador* que ejecutamos mediante el propio ordenador. Este programa tomará como datos nuestro programa escrito en lenguaje ensamblador y dará como resultado el mismo programa, pero escrito en lenguaje máquina, lenguaje que entiende el ordenador.



Cada modelo de ordenador, dependiendo del procesador que utilice, tiene su propio lenguaje ensamblador. Debido a esto decimos que estos lenguajes están orientados a la máquina.

Hoy en día son más utilizados los lenguajes orientados al problema o lenguajes de alto nivel. Estos lenguajes utilizan una terminología fácilmente comprensible que se aproxima más al lenguaje humano. En este caso la traducción es llevada a cabo por otro programa denominado *compilador*.

Cada sentencia de un programa escrita en un lenguaje de alto nivel se descompone en general en varias instrucciones en ensamblador. Por ejemplo:

```
printf("hola");
```

La función **printf** del lenguaje C le dice al ordenador que visualice en el monitor la cadena de caracteres especificada. Lo mismo podríamos decir del método **WriteLine** de C#. Este mismo proceso escrito en lenguaje ensamblador necesitará de varias instrucciones.

```
System.Console.WriteLine("hola");
```

A diferencia de los lenguajes ensambladores, la utilización de lenguajes de alto nivel no requiere en absoluto del conocimiento de la estructura del procesador que utiliza el ordenador, lo que facilita la escritura de un programa.

Compiladores

Para traducir un programa escrito en un lenguaje de alto nivel (programa fuente) a lenguaje máquina se utiliza un programa llamado *compilador*. Este programa tomará como datos nuestro programa escrito en lenguaje de alto nivel y dará como resultado el mismo programa, pero escrito en lenguaje máquina, programa que ya puede ejecutar directa o indirectamente el ordenador.

Por ejemplo, un programa escrito en el lenguaje C necesita del compilador C para poder ser traducido. Posteriormente, el programa traducido podrá ser ejecutado directamente por el ordenador. En cambio, para traducir un programa escrito en el lenguaje Java o C# necesita del compilador Java o C#, respectivamente.

Intérpretes

A diferencia de un compilador, un intérprete no genera un programa escrito en lenguaje máquina a partir del programa fuente, sino que efectúa la traducción y ejecución simultáneamente para cada una de las sentencias del programa. Por ejemplo, un programa escrito en el lenguaje *Basic* necesita el intérprete *Basic* para ser ejecutado. Durante la ejecución de cada una de las sentencias del programa, ocurre simultáneamente la traducción.

A diferencia de un compilador, un intérprete verifica cada línea del programa cuando se escribe, lo que facilita la puesta a punto del programa. En cambio, la ejecución resulta más lenta ya que acarrea una traducción simultánea.

¿QUÉ ES C?

C es un lenguaje de programación de alto nivel con el que se pueden escribir programas con fines muy diversos.

Una de las ventajas significativas de C sobre otros lenguajes de programación es que el código producido por el compilador C está muy optimizado en tamaño lo que redunda en una mayor velocidad de ejecución, y una desventaja es que C es independiente de la plataforma solo en código fuente, lo cual significa que cada plataforma diferente debe proporcionar el compilador adecuado para obtener el código máquina que tiene que ejecutarse.

¿Por qué no se diseñó C para que fuera un intérprete más entre los que hay en el mercado? La respuesta es porque la interpretación, si bien es cierto que proporciona independencia de la máquina (suponiendo que esta tiene instalado el intérprete), conlleva también un problema grave, que es la pérdida de velocidad en la ejecución del programa.

HISTORIA DEL LENGUAJE C

C es un lenguaje de programación de propósito general. Sus principales características son:

- Programación estructurada.
- Economía en las expresiones.
- Abundancia en operadores y tipos de datos.
- Codificación en alto y bajo nivel simultáneamente.
- Reemplaza ventajosamente la programación en ensamblador.
- Utilización natural de las funciones primitivas del sistema.
- No está orientado a ningún área en especial.
- Producción de código objeto altamente optimizado.
- Facilidad de aprendizaje.

El lenguaje C nació en los laboratorios Bell de AT&T y ha sido estrechamente asociado con el sistema operativo UNIX, ya que su desarrollo se realizó en este sistema y debido a que tanto UNIX como el propio compilador C y la casi totalidad de los programas y herramientas de UNIX fueron escritos en C. Su eficiencia y claridad han hecho que el lenguaje ensamblador apenas haya sido utilizado en UNIX.

Este lenguaje está inspirado en el lenguaje B escrito por *Ken Thompson* en 1970 con intención de recodificar UNIX, que en la fase de arranque estaba escrito en ensamblador, en vistas a su transportabilidad a otras máquinas. B era un lenguaje evolucionado e independiente de la máquina, inspirado en el lenguaje BCPL concebido por *Martin Richard* en 1967.

En 1972, *Dennis Ritchie* toma el relevo y modifica el lenguaje B, creando el lenguaje C y reescribiendo UNIX en dicho lenguaje. La novedad que proporcionó el lenguaje C sobre el B fue el diseño de tipos y estructuras de datos.

Los tipos básicos de datos eran **char** (carácter), **int** (entero), **float** (reales en precisión simple) y **double** (reales en precisión doble). Posteriormente se añadieron los tipos **short** (enteros de longitud ≤ longitud de un **int**), **long** (enteros de longitud ≥ longitud de un **int**), **unsigned** (enteros sin signo) y *enumeraciones*. Los tipos estructurados básicos de C son las *estructuras*, las *uniones* y las *matrices*

(*arrays*). A partir de los tipos básicos es posible definir tipos derivados de mayor complejidad.

Las instrucciones para controlar el flujo de ejecución de un programa escrito en C son las habituales de la programación estructurada, **if**, **for**, **while**, **switch-case**, todas incluidas en su predecesor BCPL. Así mismo, C permite trabajar con direcciones de memoria, con funciones y soporta la recursividad.

Otra de las peculiaridades de C es su riqueza en operadores. Puede decirse que prácticamente dispone de un operador para cada una de las posibles operaciones en código máquina. Por otra parte, hay toda una serie de operaciones que pueden hacerse con el lenguaje C, que realmente no están incluidas en el compilador propiamente dicho, sino que las realiza un *preprocesador* justo antes de la compilación. Las dos más importantes son **#define** (directriz de sustitución simbólica o de definición) e **#include** (directriz de inclusión de un archivo fuente).

Finalmente, C, que ha sido pensado para ser altamente transportable a nivel de código fuente y para programar lo improgramable, igual que otros lenguajes, tiene sus inconvenientes. Por ejemplo, carece de instrucciones de entrada y salida, de instrucciones para manejo de cadenas de caracteres, etc., trabajo que queda para la biblioteca de funciones, lo que favorece la pérdida de transportabilidad. Además, la excesiva libertad en la escritura de los programas puede llevar a errores en la programación que, por ser correctos sintácticamente, no se detectan a simple vista. También, las precedencias de los operadores convierten a veces las expresiones en pequeños rompecabezas. A pesar de todo, C ha demostrado ser un lenguaje extremadamente eficaz y expresivo.

Lenguaje C++

C++ fue desarrollado a partir del lenguaje de programación C y, con pocas excepciones, incluye a C. Esta parte de C incluida en C++ es conocida como C–, y puede compilarse como C++ sin problemas.

En 1980, se añadieron al lenguaje C características como *clases* (concepto tomado de Simula 67), comprobación del tipo de los argumentos de una función y conversión, si es necesaria, de los mismos, así como otras características; el resultado fue el lenguaje denominado *C con Clases.*

En 1983/84, *C con Clases* fue rediseñado, extendido y nuevamente implementado. El resultado se denominó *Lenguaje C++*. Las extensiones principales fueron *funciones virtuales*, *funciones sobrecargadas* (un mismo identificador puede utilizarse para invocar a distintas formas de una función) y *operadores sobrecargados* (un mismo operador puede utilizarse en distintos contextos y con

distintos significados). Después de algún otro refinamiento más, C++ quedó disponible en 1985. Este lenguaje fue creado por *Bjarne Stroustrup* (AT&T Bell Laboratories) y documentado en varios libros suyos.

El nombre de C++ se debe a *Rick Mascitti*, significando *el carácter evolutivo de las transformaciones de C* ("++" es el operador de incremento de C).

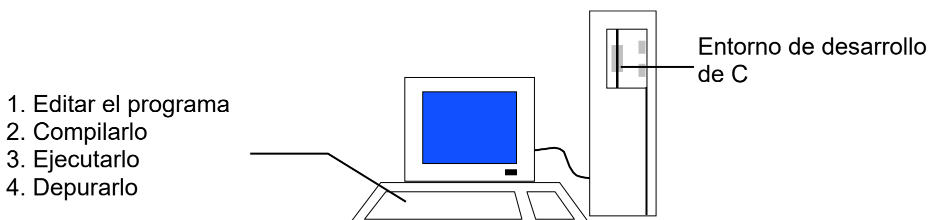
Posteriormente, C++ ha sido ampliamente revisado y refinado, lo que ha dado lugar a añadir nuevas características, como herencia múltiple, funciones miembro **static** y **const**, miembros **protected**, tipos genéricos de datos (también denominados plantillas) y manipulación de excepciones. Además de esto, también se han hecho pequeños cambios para incrementar la compatibilidad con C.

De lo expuesto se deduce que C++ es un lenguaje híbrido, que, por una parte, ha adoptado todas las características de la programación orientada a objetos que no perjudiquen su efectividad, y por otra, mejora sustancialmente las capacidades de C. Esto dota a C++ de una potencia, eficacia y flexibilidad que lo convierten en un estándar dentro de los lenguajes de programación orientados a objetos.

En este libro no abordaremos las nuevas aportaciones de C++ encaminadas a una programación orientada a objetos, sino que nos limitaremos a realizar programas estructurados utilizando lo que hemos denominado C– o simplemente C. Cuando haya aprendido a programar con C, puede dar un paso más e introducirse en la programación orientada a objetos para lo cual le recomiendo mi otro libro titulado *Programación orientada a objetos con C++* o bien, si prefiere un libro más completo, *Enciclopedia del lenguaje C++*.

REALIZACIÓN DE UN PROGRAMA EN C

En este apartado se van a exponer los pasos a seguir en la realización de un programa, por medio de un ejemplo. La siguiente figura muestra de forma esquemática lo que un usuario de C necesita y debe hacer para desarrollar un programa.



Evidentemente, para poder escribir programas se necesita un entorno de desarrollo C/C++; esto es: un editor de texto, el compilador C/C++ (incluyendo el enlazador del que hablaremos más adelante) y un depurador. Por lo tanto, en la

unidad de disco de nuestro ordenador tienen que estar almacenadas las herramientas necesarias para editar, compilar y depurar nuestros programas. Por ejemplo, supongamos que queremos escribir un programa denominado *saludo.c* (o bien *saludo.cpp*). Las herramientas (programas) que tenemos que utilizar y los archivos que producen son:

Programa	**Produce el archivo**
Editor	*saludo.c* (o bien *saludo.cpp*)
Compilador C/C++	*saludo.obj* o *saludo.o*, dependiendo del compilador
Enlazador	*saludo.exe* o *a.out* por omisión, dependiendo del compilador
Depurador	ejecuta paso a paso el programa ejecutable

La tabla anterior indica que una vez editado el programa *saludo.c* o *saludo.cpp*, se compila obteniéndose el archivo objeto *saludo.obj* o *saludo.o*, el cual es enlazado con las funciones necesarias de la biblioteca de C dando lugar a un único archivo ejecutable *saludo.exe* o *a.out*.

Edición de un programa

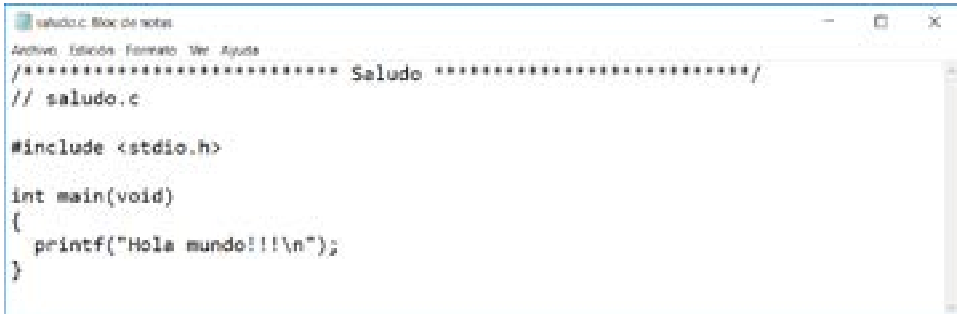
Para *editar* un programa, primeramente, pondremos en marcha el editor o procesador de textos que vayamos a utilizar. Si disponemos de un entorno integrado (incluye el editor, el compilador y el depurador; vea el apéndice C) podemos utilizar el procesador de textos suministrado con él y si no, utilizaremos nuestro propio procesador, por ejemplo, uno de los proporcionados con el sistema operativo (vea la figura mostrada un poco más adelante; se trata del bloc de notas). El nombre del archivo que se elija para guardar el programa en el disco debe tener como extensión *.c* o *.cpp* (*cpp* es la extensión utilizada por *C++*).

El paso siguiente es escribir el texto correspondiente al programa fuente. Cada *sentencia* del lenguaje C finaliza con un *punto y coma* y cada *línea del programa* se finaliza pulsando la tecla *Entrar* (*Enter* o ↵).

Como ejercicio para practicar lo expuesto hasta ahora, empecemos con la creación de un programa sencillo: el clásico ejemplo de mostrar un mensaje de saludo.

Empecemos por editar el archivo fuente C correspondiente al programa. El nombre del archivo elegido para guardar el programa en el disco debe tener como extensión *c*, o bien *cpp* si el compilador soporta C++; por ejemplo, *saludo.c*.

Una vez visualizado el editor, escribiremos el texto correspondiente al programa fuente. Escríbalo tal como se muestra a continuación:



¿Qué hace este programa?

Comentamos brevemente cada línea de este programa. No hay que apurarse si algunos de los términos no quedan muy claros ya que todos ellos se verán con detalle en capítulos posteriores.

Las dos primeras líneas son simplemente comentarios: empiezan con /* y terminan con */ o, simplemente, empiezan con //. Los comentarios no son tenidos en cuenta por el compilador.

La tercera línea incluye el archivo de cabecera *stdio.h* que contiene las declaraciones necesarias para las funciones de entrada o salida (E/S) que aparecen en el programa; en nuestro caso para **printf**. Esto significa que, como regla general, antes de invocar a una función hay que declararla. Las palabras reservadas de C que empiezan con el símbolo # reciben el nombre de *directrices* del compilador y son procesadas por el *preprocesador* de C cuando se invoca al compilador, pero antes de iniciarse la compilación.

A continuación, se escribe la función principal **main**. Todo programa escrito en C tiene una función **main**. Observe que una función se distingue por el modificador () que aparece después de su nombre y que el cuerpo de la misma empieza con el carácter { y finaliza con el carácter }.

La función **printf** pertenece a la biblioteca de C y su cometido es escribir en el monitor la expresión que aparece especificada entre comillas. La secuencia de escape *\n* que aparece a continuación de la cadena de caracteres “Hola mundo!!!” indica al ordenador que después de escribir ese mensaje, avance el cursor de la pantalla al principio de la línea siguiente. Observe que la sentencia finaliza con punto y coma.

Guardar el programa escrito en el disco

El programa editado está ahora en la memoria. Para que este trabajo pueda tener continuidad, el programa escrito se debe grabar en el disco utilizando la orden correspondiente del editor. Muy importante: el nombre del programa fuente debe añadir la extensión *c*, o bien *cpp* si el compilador soporta C++.

Compilar y ejecutar el programa

El siguiente paso es *compilar* el programa; esto es, traducir el programa fuente a lenguaje máquina para posteriormente enlazarlo con las funciones necesarias de la biblioteca de C, proceso que generalmente se realiza automáticamente, y obtener así un programa ejecutable. Dependiendo del fabricante del compilador, la orden correspondiente para compilar y enlazar el programa *saludo.c* podría ser alguna de las siguientes:

- En un sistema Windows con un compilador de Microsoft, la orden *cl* del ejemplo siguiente invoca al compilador C (incluye el preprocesador, el compilador propiamente dicho y el enlazador) para producir el archivo ejecutable *saludo.exe* (véase *Interfaz de línea de órdenes en Windows* en el apéndice C).

```
cl saludo.c
```

- En un sistema UNIX, la orden *cc* del ejemplo siguiente invoca al compilador C (incluye el preprocesador, el compilador propiamente dicho y el enlazador) para producir el archivo ejecutable *saludo*. Si no hubiéramos añadido la opción *-o saludo*, el archivo ejecutable se denominaría, por omisión, *a.out* (véase *Interfaz de línea de órdenes en Unix/Linux* en el apéndice C).

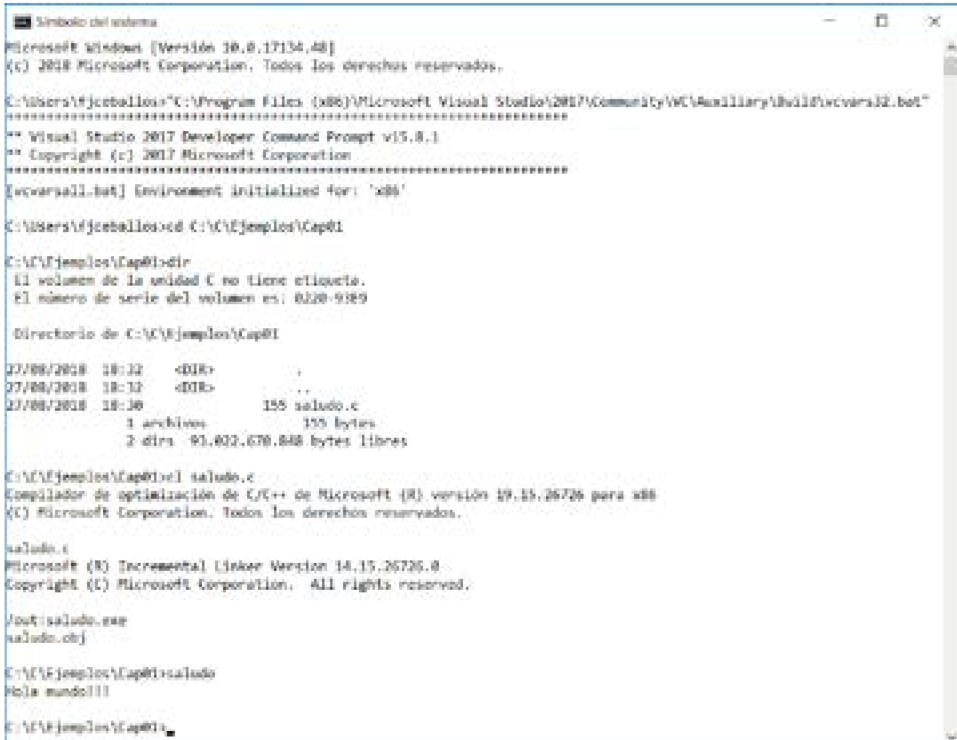
```
cc saludo.c -o saludo
```

Al compilar un programa, se pueden presentar *errores de compilación* debidos a que el programa escrito no se adapta a la sintaxis y reglas del compilador. Estos errores tendremos que corregirlos hasta obtener una compilación sin errores.

Para ejecutar el archivo resultante, escriba el nombre de dicho archivo a continuación del símbolo del sistema, en nuestro caso *saludo*, y pulse *Entrar*. Para el ejemplo que nos ocupa, el resultado será que se visualizará sobre la pantalla el mensaje:

```
Hola mundo!!!
```

La siguiente figura muestra la consola de un sistema Windows desde la cual se ha realizado este proceso:



En esta figura puede observar que, después de establecer las variables de entorno del sistema operativo, nos hemos dirigido a la carpeta donde guardamos el archivo fuente *saludo.c* para compilarlo: *cl saludo.c*. El resultado es el archivo intermedio *saludo.obj* y el archivo ejecutable *saludo.exe*. Finalmente ejecutamos el programa: *saludo* (no hace falta especificar la extensión *.exe*).

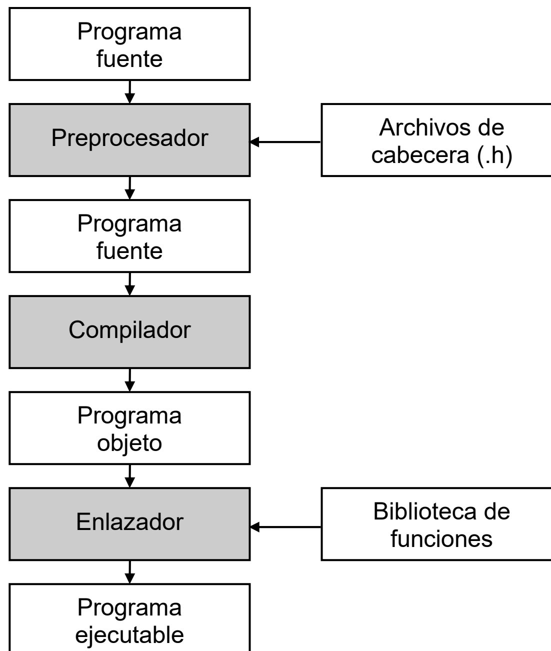
Biblioteca de funciones

Como ya dijimos anteriormente, C carece de instrucciones de E/S, de instrucciones para manejo de cadenas de caracteres, etc., con lo que este trabajo queda para la biblioteca de funciones provista con el compilador. Una función es un conjunto de instrucciones que realizan una tarea específica. Una biblioteca es un archivo separado en el disco (generalmente con extensión *.lib* en Windows o con extensión *.a* en UNIX) que contiene las funciones que realizan las tareas más comunes, para que nosotros no tengamos que escribirlas. Como ejemplo, hemos visto anteriormente la función **printf**. Si esta función no existiera, sería labor nuestra escribir el código necesario para visualizar los resultados sobre la pantalla.

Para utilizar una función de la biblioteca simplemente hay que invocarla utilizando su nombre y pasar los argumentos necesarios entre paréntesis. Por ejemplo:

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

La figura siguiente muestra cómo el código correspondiente a las funciones de biblioteca invocadas en nuestro programa es añadido por el *enlazador* cuando se está creando el programa ejecutable.



Guardar el programa ejecutable en el disco

Como hemos visto, cada vez que se realiza el proceso de *compilación* y *enlace* del programa actual, C genera automáticamente sobre el disco un archivo ejecutable. Este archivo puede ser ejecutado directamente desde el sistema operativo sin el soporte de C, escribiendo el nombre del archivo a continuación del símbolo del sistema (*prompt* del sistema) y pulsando la tecla *Entrar*.

Cuando se crea un archivo ejecutable, primero se utiliza el compilador C para compilar el programa fuente, dando lugar a un archivo intermedio conocido como archivo objeto (con extensión *.obj* o *.o* según el compilador). A continuación, se utiliza el programa *enlazador* (*linker*) para unir, en un único archivo ejecutable, el módulo o los módulos que forman el programa compilados separadamente y las funciones de la biblioteca del compilador C que el programa utilice.

Al ejecutar el programa, pueden ocurrir *errores durante la ejecución*. Por ejemplo, puede darse una división por 0. Estos errores solamente pueden ser detectados por C cuando se ejecuta el programa y serán notificados con el correspondiente mensaje de error.

Hay *otro tipo de errores* que no dan lugar a mensaje alguno. Por ejemplo, un programa que no termine nunca de ejecutarse, debido a que presenta un bucle o lazo donde no se llega a dar la condición de terminación. Para detener la ejecución se tienen que pulsar las teclas *Ctrl+C*.

Depurar un programa

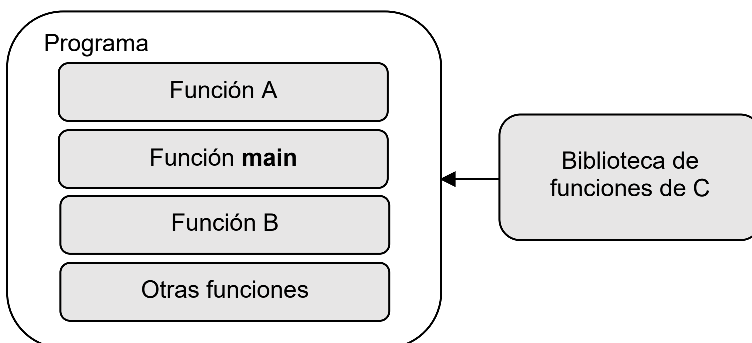
Una vez ejecutado el programa, la solución puede ser incorrecta. Este caso exige un análisis minucioso de cómo se comporta el programa a lo largo de su ejecución; esto es, hay que entrar en la fase de *depuración* del programa.

La forma más sencilla y eficaz para realizar este proceso es utilizar un programa *depurador*. En el apéndice C se explica cómo utilizar el depurador del entorno de desarrollo integrado (EDI) Microsoft Visual Studio y el depurador *gdb* de un sistema UNIX.

UN AVANCE SOBRE LA PROGRAMACIÓN CON C

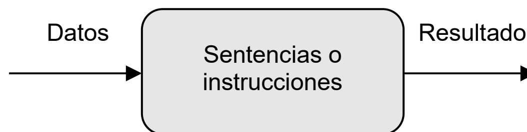
En este apartado vamos a exponer un pequeño avance sobre conceptos que se estudiarán con detenimiento en los capítulos siguientes. Este avance le proporcionará un conocimiento básico sobre aspectos que están presentes en todos los programas como son la definición de variables, la entrada/salida, las sentencias de control y las funciones.

Un programa C es un conjunto de funciones y una función es la unidad básica de un programa C.



En un programa C tiene que haber obligatoriamente, al menos, una función llamada **main**, ya que es esta función la que define el compilador como punto por el cual empieza y finaliza la ejecución de todo programa. Un ejemplo puede verlo en el programa *saludo* que acabamos de exponer.

Una función incluye un conjunto de sentencias o instrucciones que al ejecutarse producen un resultado obtenido, generalmente, en base a unos datos. Puede pensar en una función como en una caja negra a la que le son entregados unos datos para que los procese y genere un resultado:



Un ejemplo: *x* = *log(y)*; en este caso, el dato es *y* y el resultado generado se guarda en *x*. Si tuviéramos que escribir esa función *log*, tendría la siguiente forma:

```
double log( double y )
{
  double resultado;
  // Sentencias que calculan el resultado (el logaritmo)
  // en función del parámetro y. Por ejemplo:
  resultado = log10(y);
  return resultado; // devolver el valor calculado
}
```

En este ejemplo, la función *log* recibe un dato en su parámetro *y* (el dato es de tipo **double**: valor real con decimales), ejecuta las sentencias que incluye el cuerpo de la misma (la caja) y devuelve un resultado que almacenamos en algún sitio (en *x*, por ejemplo). Una vez que la función existe, otra función del programa puede llamarla cuando requiera el cálculo del logaritmo de un valor. Por ejemplo:

```
int main(void)
{
  double x;
  x = log(2);
  printf("%f\n", x);
  return 0; // escribir esta línea y no hacerlo, es equivalente
}
```

Los datos pueden ser también definidos dentro de la propia función (en la caja negra) en lugar de enviárselos a la misma, y la función puede generar un resultado que ella misma controla sin necesidad de tener que devolverlo. Por ejemplo, en el código anterior podemos observar que la función *log* indica que devuelve un valor de tipo **double**; si no devolviera nada hubiéramos escrito: **void** *log*(**double** *y*).

Entrada y salida

Cuando resolvemos un determinado problema (por ejemplo, una ecuación de segundo grado) lo hacemos siguiendo una serie de operaciones (algoritmo) que nos conducen a la solución (la salida), lógicamente partiendo de unos datos de entrada. Los datos de entrada a los que nos referimos, los proporcionaremos por medio de variables (si fueran constantes, como en la función **main** anterior, el resultado sería siempre el mismo), datos que leeremos al inicio de la ejecución.

Ya hemos visto que la función **printf** de C sirve para mostrar resultados. Pues bien, para leer datos introducidos por el teclado, la biblioteca de C proporciona la función **scanf**. Como ejemplo, vamos a modificar la función **main** anterior para que permita mostrar el logaritmo de un valor *n* introducido por el teclado, para lo cual, utilizaremos la función **scanf**. La solución sería la siguiente:

```
int main(void)
{
  double x, n;
  scanf("%lf", &n);
  x = log(n);
  printf("%f\n", x);
}
```

Observemos los parámetros de la función **scanf**: el primero es una cadena de caracteres *%lf* que indica que el valor introducido será almacenado como un valor de tipo **double** (números decimales: poseen una parte decimal, en oposición a los números enteros), y el segundo parámetro es la dirección (posición en la memoria) de la variable donde queremos almacenar el resultado (*&n* se lee "dirección de *n*"). Para leer un decimal de tipo **float** emplearíamos el formato *%f*, para leer un entero (**int**) emplearíamos el formato *%d*, para leer un carácter (**char**) emplearíamos el formato *%c* y para leer una cadena de caracteres (por ejemplo, *Isabella*) emplearíamos el formato *%s*. Estos formatos también los utiliza **printf** para mostrar los valores de las variables o expresiones. En los siguientes capítulos veremos todo esto con detalle.

Sentencias de control

En ocasiones será necesario ejecutar unas sentencias u otras en función de otros resultados. Por ejemplo, cuando se introduzca el valor de *n* solicitado por una sentencia como *scanf("%lf", &n)*, puede suceder que nos equivoquemos y escribamos, por ejemplo, *x2* en lugar de *2*. Evidentemente *x2* no es un valor real de tipo **double**, por lo que **scanf** no podrá leerlo para almacenarlo en *n*. El programa continuaría, pero el resultado ya no sería el esperado, ya que *n* puede contener cualquier cosa (basura).

Para dar solución al problema expuesto, tenemos que saber que **scanf** devuelve un valor entero que coincide con el número de variables a las que ha podido asignar un valor (en el supuesto de haber especificado varias variables separadas por comas, cuando se produzca un fallo para una, se abandona la lectura y el programa continúa). En nuestro ejemplo, como solo estamos leyendo una variable, *n*, **scanf** devolverá *1* si el valor tecleado fue correcto y *0* si no fue correcto. Según esto vamos a modificar la función **main** del apartado anterior así:

```
int main(void)
{
  int r;
  double x, n;
  printf("Dato: ");
  r = scanf("%lf", &n);
  if (r == 1)
  {
    x = log(n);
    printf("El logaritmo de %g es %f\n", n, x);
  }
  else
  {
    printf("El dato introducido no es correcto.\n");
  }
}
```

Observemos el código anterior. Según lo explicado anteriormente, cuando se llame a la función **scanf**, ésta devolverá un valor *0* o *1* que se guardará en *r*. Entonces, si *r* es igual a *1* (*if (r == 1)*) se calcula e imprime el logaritmo de *n*, y si no (*else*), se muestra un mensaje indicando que el dato introducido no es correcto. La cláusula **else** es opcional.

En otras ocasiones necesitaremos repetir la ejecución de unas determinadas sentencias un número de veces. Por ejemplo, la función **main** anterior podría requerir repetir la entrada de datos siempre que el dato tecleado sea incorrecto, esto es, mientras el valor de *r* sea *0*. Esto lo podemos hacer así:

```
int main(void)
{
  int r = 0;
  double x, n;
  while (r == 0)
  {
    printf("Dato: ");
    r = scanf("%lf", &n);
    if (r == 0)
    {
      printf("El dato introducido no es correcto.\n");
    }
```

```
    while (getchar() != '\n'); // eliminar el dato incorrecto
  }
  x = log(n);
  printf("El logaritmo de %g es %f\n", n, x);
}
```

```
Ejecución del programa:

Dato: x2
El dato introducido no es correcto.
Dato: hola
El dato introducido no es correcto.
Dato: 2
El logaritmo de 2 es 0.301030
```

Ahora, en el código anterior, observamos que mientras el valor de *r* sea *0* (*while (r == 0) { ... }*) repetimos la ejecución de las sentencias que hay dentro del bloque (*{ ... }*) de la sentencia repetitiva **while**.

Funciones

El concepto básico de lo que es una función ya está explicado, pero tenemos que decir también que los compiladores C/C++, cuando una función llama (invoca) a otra para que se ejecute, requieren que el código de la función llamada (la definición de la función) esté escrito explícitamente antes de la sentencia de llamada a la misma; si está después, es necesario escribir antes la declaración o prototipo de dicha función (la cabecera empleada en la definición de la función). Esto, entre otras cosas, es lo que hacen los archivos de cabecera con respecto a las funciones de la biblioteca de C, como **scanf** y **printf**, que utilizamos. Por ejemplo, la operación de leer un dato de tipo **double** y verificar que el dato fue válido podríamos realizarla en una función, así podríamos utilizarla siempre que necesitemos leer un dato del tipo citado, lo que evita tener que repetir código. Esto lo haríamos así:

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

double leerDato(); // declaración de la función

double log(double y)
{
  double resultado;
  // Sentencias que calculan el resultado
  // en función del parámetro y. Por ejemplo:
  resultado = log10(y);
  return resultado;
}
```

```
int main(void)
{
  double x, n;
  printf("Dato: ");
  n = leerDato(); // llamada a la función
  x = log(n);
  printf("El logaritmo de %g es %f\n", n, x);
}

double leerDato() // definición de la función
{
  int r = 0;
  double dato;
  while (r == 0)
  {
    r = scanf("%lf", &dato);
    if (r == 0)
    {
      printf("Dato incorrecto. Introduzca otro: ");
    }
    while (getchar() != '\n'); // eliminar el dato incorrecto
  }
  return dato;
}
```

```
Ejecución del programa:

Dato: x2
Dato incorrecto. Introduzca otro: hola
Dato incorrecto. Introduzca otro: 2
El logaritmo de 2 es 0.301030
```

Observemos en el código anterior la declaración, la llamada y la definición de la función *leerDato*; como la llamada está antes que la definición, ha sido necesario añadir la declaración antes de la llamada. Lo mismo ocurre con las funciones de la biblioteca de C **scanf** y **printf**; sus prototipos son incluidos por la directriz *#include <stdio.h>*. En cambio, la función *log* está definida antes de ser invocada por **main**, por lo que no se requiere escribir su declaración.

Matrices

Una matriz de una dimensión (*array* en inglés) se define como una variable que contiene una serie de elementos del mismo tipo. Cada elemento es referenciado por la posición que ocupa dentro de la matriz; esa posición recibe el nombre de índice y los índices son correlativos, siendo el primero, generalmente, 0 o 1. Por ejemplo, la siguiente línea define una matriz m de n componentes m_0, m_1,…, m_{n-1}.