Introducción al lenguaje C
2. Elementos del lenguaje

C es un lenguaje de programación de propósito general desarrollado en 1972 por Dennis Ritchie. Aunque inicialmente se diseño para programa el sistema operativo UNIX, es un lenguaje que se encuentra implementado en la gran mayoría de los sistemas operativos. Se trata de un lenguaje **imperativo**[1] que facilita la creación de programas siguiendo la **programación estructurada**.

Un programa C está formado por uno o más archivos de texto que contienen módulos de código fuente. Los compiladores serán capaces de compilar cada uno de esos módulos de código fuente por separada para formar un único programa ejecutable. Cada uno de los módulos está formado, a su vez, por funciones, pequeños fragmentos de código que realizan una función determinada. Existe una función especial, la función `main`, que será el punto de arranque del programa: al ejecutar un programa C, se comenzará por la primera instrucción de la función `main`.

2.1. Caracteres, palabras reservadas, comentarios e identificadores

Cada función en C está formada por una serie de sentencias o instrucciones. Esas instrucciones, a su vez están formadas por elementos o **componentes léxicos** (*tokens*) que constituyen el léxico (las palabras) del lenguaje. Esas palabras se pueden combinar con una serie de reglas (sintaxis del lenguaje) para formar sentencias que indicarán al procesador qué instrucciones debe ejecutar (semántica). Existen cinco categorías básicas de componentes léxicos:

- Palabras reservadas
- Identificadores
- Constantes
- Literales de cadena
- Operadores y signos de puntuación

2.1.1. Caracteres C

Las palabras C se escriben utilizando un conjunto limitado de caracteres. Estos caracteres serán:

[1] Un lenguaje imperativo está formado por una serie de sentencias u órdenes que cambian el estado del programa.

- Caracteres alfabéticos[2]
- Dígitos
- Caracteres especiales
- Espacios en blanco

Tabla 2.1. Caracteres C

Letras minúsculas	a b c d e f g h y j k l m n o p q r s t u v w x y z
Letras mayúsculas	A B C D E F G H I J K L M N O P Q R S T U W X Y Z
Dígitos	0 1 2 3 4 5 6 7 8 9
Caracteres especiales	, & . ^ * : - ? + ‘ < “ > ¡ (|) / [\ ] ~ { _ } $ # %
Espacios en blanco	Espacio en blanco, retorno de carro, tabulación, nueva línea, avance de página[3]

Los espacios en blanco se ignoran, a no ser que se utilicen para separa componentes.

2.1.2. Palabras reservadas

Todos los lenguajes de programación tienen una serie de palabras reservadas que tienen un significado especial dentro del lenguaje. Se trata de palabras incluidas dentro del propio lenguaje de programación y que no pueden usarse para otros fines que los dispuestos en la sintaxis del lenguaje.

Tabla 2.2. Palabras reservadas en C

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

2.1.3. Comentarios

Los comentarios permiten incluir explicaciones dentro del código fuente. El compilador ignora los comentarios, por lo que también se utilizan para anular una instrucción o un grupo de instrucciones durante la fase de depuración del problema.

Un comentario es un bloque de caracteres entre los símbolos /*, que inicia el comentario, y */ que lo cierra. Puede ocupar varias líneas. Por ejemplo:

[2] No siempre están disponibles todos los caracteres posibles, ya que en ocasiones se utilizan sólo los caracteres alfabéticos del código ASCII estándar, es decir los comprendidos entre el carácter 0 y el carácter 127. Esto implica que, por ejemplo, no se incluye la eñe ni las vocales acentuadas.

[3] A diferencia del carácter en blanco, que se trata de un carácter imprimible, concretamente el carácter 32 del código ASCII, el resto se trata de caracteres de control que envían códigos especiales a la impresora o a la pantalla. El retorno de carro (*carriege return* o CR) es el carácter 13 del código ASCII que se genera al pulsar la tecla ENTER. El carácter de tabulación es el carácter 9 del código ASCII que se genera al pulsar la tecla TAB. El avance de página (*line feed* o LF) produce el salto a la siguiente línea y es el carácter ASCII 10. El avance de página (*form feed* o FF) es el carácter 12 del código ASCII y provoca el salto a la siguiente página en la impresora.

```
/* Esto es un comentario
de dos líneas*/
```

El estándar ANSI C de 1999 (C99) incluyó también los caracteres // para comentarios de una sola línea. Por ejemplo:

```
//Esto es un comentario
//de dos líneas
```

2.1.4. Identificadores

Los identificadores son elementos del programa utilizados para nombrar los objetos que crea el programador (nombres de variables, constantes simbólicas, nombres de funciones, etc.). Son una secuencia de caracteres alfabéticos, dígitos y el carácter de subrayado.

A la hora de formar un identificador hay que tener en cuenta las siguientes restricciones:

- Deben comenzar por un carácter alfabético
- Su nombre no puede coincidir con el de una palabra reservada

Aunque el carácter de subrayado (_) se considera como una letra, no es conveniente comenzar un identificador por ese carácter, ya que algunas rutinas internas comienzan en ocasiones por dicho carácter. Su utilidad está en aumentar la legibilidad de las variables separando las palabras de aquellas que tienen un nombre compuesto por varias palabras (por ejemplo `total_producto`).

Otro aspecto a tener en cuenta es la longitud del identificador. Sólo son significativos los primeros 63 caracteres del identificador. En ocasiones puede haber problemas en los nombres de funciones y variables externas; en estos casos la longitud puede ser menor, ya que los ensambladores y cargadores que utilicen esos elementos pueden no reconocer tantos caracteres. Es estándar garantiza una longitud de 31 caracteres significativos para las funciones y variables externas.

Los identificadores en C son **sensibles a mayúsculas**. Esto quiere decir que se hace distinción entre mayúsculas y minúsculas, de forma que una variable `Total` será distinta de otra que se llame `total`. Esto también es aplicable a las palabras reservadas que deben escribirse todas en minúscula. Como estilo de programación, lo normal es utilizar letras minúsculas para los nombres de las variables y mayúsculas para las constantes.

2.2. Tipos de datos

Un tipo de dato es la forma que tiene el compilador de indicar a la máquina cómo entender una secuencia de bits almacenada en memoria. Por ejemplo si se almacena en memoria una variable de tipo `int` (entero), el compilador interpretará los 32 bits que la forman como un número natural. Si esos mismos 32 bits se almacenaran en memoria como una variable de tipo `float` (real de simple precisión), lo interpretará como un número real en coma flotante.

2.2.1. Tipos enteros

Un dato de tipo entero (`int`) se utiliza para almacenar datos numéricos enteros. El tamaño depende de la implementación del compilador y se encuentra especificado en el archivo de cabecera `limits.h`. El estándar fija su tamaño mínimo que debe ser de 16 bits, lo que permitiría representar valores entre -32.768 y 32.767.

Además se dispone de dos modificadores que se pueden aplicar a esos tipos enteros: `short` y `long`, es decir enteros cortos y enteros largos. De esta forma tendríamos los datos `short int` y `long int`. En la práctica, en estos casos se puede omitir la palabra `int`. El propósito de estos modificadores es poder aumentar o disminuir el tamaño natural de un entero para una máquina concreta. Lo habitual es que el `short` sea de 16 bits (valores entre -32.768 y 32.767), el `long` de 32 bits (valores entre -2.147.483.648 y 2.147.483.647) y que el `int` pueda tener 16 o 32 bits. De esta forma se puede adaptar el compilador a su uso en una máquina concreta. El requisito que pone el estándar es que los datos de tipo `short` e `int` son al menos de 16 bits, los de tipo `long` tienen al menos 32 bits y que el tamaño de un `long` debe ser mayor o igual que el de un `int` y éste, a su vez, mayor o igual que un `short`.

Los datos enteros también pueden tener los modificadores `signed` o `unsigned`. Con el primero el tipo podrá tener valores negativos y con el segundo su valor mínimo sería de 0. De esta forma con datos `short` de 16 bits, un dato `unsigned short` tomaría valores entre 0 y 65.535, con datos `int` de 32 bits, un dato `unsigned int` tomaría valores entre 0 y 4.294.967.295.

2.2.2. Datos reales

Los datos reales representan datos numéricos no enteros almacenados en coma flotante (en forma de mantisa y exponente). Existen dos tipos de datos reales los `float` y los `double`. Un dato `float` permite almacenar valores numéricos de precisión normal almacenados en coma flotante. Un dato de tipo `double` almacena valores numéricos de doble precisión almacenados también en coma flotante. Al igual que los tipos enteros, el tipo `double` admite el modificador `long` para la representación de números reales de precisión extendida.

La precisión de estos tipos de datos depende de la implementación del compilador y se encuentra especificada en el archivo de cabecera `float.h` dónde se indica la precisión, tamaño de la mantisa, del exponente, etc. de los tipos `float`, `double` y `long double`. En la implementación de lcc para win32 los tamaños establecidos son de 4 bytes para un `float`, 8 bytes para un `double` y 12 bytes para un `long double`.

2.2.3. Datos de tipo carácter

El otro tipo de dato básico de C es el tipo carácter (`char`). Un `char` es un número entero que se utiliza para almacenar un único carácter de juego de caracteres utilizado. Normalmente ocupa un único byte y admite los modificadores `signed` o `unsigned`. De esta forma, si cada carácter ocupa 8 bits, un `signed char` permitiría almacenar valores entre -128 y 127, y un `unsigned char` almacenaría valores entre 0 y 255.

Dependiendo de la implementación del compilador el dato `char` sin modificador podrá tener o no signo (en el caso de la implementación de lcc para win32 es con signo), pero los caracteres que se pueden imprimir serían siempre positivos (es decir entre 0 y 127).

2.2.4. Datos lógicos

Un dato lógico es el resultado de una expresión lógica y sólo puede almacenar los valores verdadero y falso. En C no existen un tipo de dato lógico específico ya que los resultados de las operaciones lógicas devuelven un dato de tipo entero: 0 si la expresión es falsa o 1 si la expresión es verdadera.

El estándar ANSI C99 incluyó entre los archivos de cabecera estándar el archivo `stdbool.h` que contiene el tipo de dato `bool` para almacenar datos lógicos y las constantes `true` y `false`. Si queremos utilizar este tipo de datos y estas constantes será necesario incluir ese archivo de cabecera al comienzo del programa fuente:

```
#include <stdbool.h>
```

2.2.5. Datos de tipo cadena

Los datos de tipo cadena no están implementados de forma directa en C. Un dato de tipo cadena es una secuencia de caracteres y se implementa mediante un array de caracteres cuyo último carácter significativo es el carácter nulo (`\0`).

2.3. Constantes

Los datos se utilizan en un programa en forma de constantes o de variables. Una constante es un dato que no puede modificar su valor a lo largo de la ejecución del programa. En un programa introduciremos una constante escribiendo directamente su valor dentro del programa (constante literal). También podemos asociar dicho valor a un identificador y utilizarlo posteriormente (constantes simbólicas).

2.3.1. Constantes enteras

Están formadas por una secuencia de dígitos que puede ir precedido por el signo – o + de signo. El tipo de dato de la constante dependerá del valor que represente, de forma que el dato 2345 será una constante de tipo `int`, el dato 1023564677 podría ser de tipo `long` y el dato 3223345567 de tipo `unsigned long`.

Una constante entera puede llevar un sufijo `L`, `U` o `UL` (en mayúscula o minúscula) para indicar que el dato es de tipo `long`, `unsigned` o `unsigned long`.

Las constantes enteras también se pueden expresar en notación octal o hexadecimal. Una constante que comience por un cero representará un dato expresado en base 8. Una constante que comience por la secuencia `0x` o `0X` representará un dato expresado en hexadecimal.

Tabla 2.3. Constantes enteras (se considera que un int tiene 16 bits y un long 32)

Constante	Tipo de dato
1234	Constante entera representada en formato decimal
23456L	Constante entera larga representada en formato decimal
23456UL	Constante entera larga sin signo representada en formato decimal
40234	Constante entera sin signo representada en formato decimal
109345	Constante entera larga representada en formato decimal
3456345678	Constante entera larga sin signo representada en formato decimal
045	Constante octal con el valor decimal 37
0xA2E	Constante hexadecimal con el valor decimal 2606

2.3.2. Constante en coma flotante

Están formadas por una serie de dígitos con un punto decimal (234.54) o en formato exponencial (2.345E-12, es decir 2.345 x 10^{12}). Si no contiene ningún sufijo, las constantes serán de tipo `dobule`.

Al igual que las constantes enteras, se pueden acompañar por un sufijo. `F` (o `f`) indica que es de tipo `float`; `L` (o `l`) indica que la constante será de tipo `long double`.

2.3.3. Constantes de carácter

Una constante de carácter está formada por un carácter encerrado entre comillas simples. El valor de la constante será el valor numérico de la misma según el juego de caracteres que se esté utilizando[4].

Algunos caracteres no pueden representarse directamente mediante un carácter imprimible como el carácter de nueva línea o la comilla simple. Para representarlos se utilizan las **secuencias de escape**, uno o más caracteres precedidos por la barra invertida (\). La siguiente tabla muestra las distintas secuencias de escape disponibles.

Tabla 2.4. Secuencias de escape

Secuencia	Significado	Valor
\n	Nueva línea	10
\r	Retorno de carro	13
\b	Retroceso	8
\v	Tabulador vertical	11
\t	Tabulador horizontal	9
\f	Avance de página	12
\e	Espape	27
\a	Beep	7
\\	Barra invertida	\
\”	Comilla doble	“
\’	Comilla simple	‘
\?	Interrogación	?
\x<dígito hex.>	Inserta el carácter correspondiente al valor entero del dígito hexadecimal	Depende del dígito hexadecimal, \x41 (valor decimal 65) devolvería una A
\<número octal>	Inserta el carácter correspondiente al	Depende del número octal, \101 (valor

[4] El juego de caracteres utilizado por lcc para Win32 es el Unicode ISO-8859-1 o Latin-1 que permite representar los caracteres de los lenguajes de Europa occidental y que en sus primeros 127 caracteres coincide con el código ASCII estándar.

Secuencia	Significado	Valor
	valor entero del número octal	decimal 65) devolvería una A

2.3.4. Constantes de cadena

Una constante o literal de cadena se representa como una secuencia de caracteres encerrada entre comillas dobles. Las comillas no forman parte de la cadena y dentro de los caracteres se pueden incluir cualquiera de las secuencias de escape de la tabla 2.4.

Las constantes se pueden concatenar (unir) en tiempo de compilación, de forma que las cadenas

```
"Fundamentos" " de Programación"
```

Son equivalentes a la cadena `"Fundamentos de Programación"`.

Internamente las cadenas son un array de caracteres (un conjunto de caracteres que ocupan posiciones contiguas de memoria) y que acaban en un carácter nulo. Es importante distinguir entre una constante de carácter y una de cadena. Hay que tener en cuenta que `'x'` es un dato distinto que `"x"`; mientras que el primero es un carácter, ocupa un byte y por lo tanto se puede representar como un número entero, el segundo es un conjunto de caracteres que acaba con la secuencia `\0`.

2.3.5. Constantes simbólicas

Es posible asignar un nombre a una constante de forma que en un programa hagamos referencia a la constante mediante el nombre y no mediante su valor.

Se pueden definir esas constantes de dos formas:

- Al principio del programa fuente y fuera de cualquier función con la directiva del pre-procesador `#define`.

  ```
  #define IVA      0.18
  ```

  Al compilar el programa todas las apariciones de la palabra IVA serán sustituidas por su valor (0.16) en el programa objeto.

- Mediante la declaración `const`. Una variable declarada con `const` indica que su valor no cambiará a lo largo de la ejecución del programa.

  ```
  const float iva = 0.18
  ```

2.3.6. Constantes enumeradas

Otra forma de declarar constantes es mediante enumeraciones. Una enumeración permite asociar valores constantes enteros a una serie de identificadores.

```
enum  nombreEnumeracion {enumerado1, enumerado2,…,enumeradon}
```

de forma que al identificador `enumerado1` se le asocia el valor 0, al identificador `enumeradoo2` el valor 1, etc.

```
enum diaSemana {domingo, lunes, martes, miercoles,
                jueves, viernes, sabado};
```

Con esta declaración se crean 7 constantes enteras de forma que a la constante `domingo` se le asocia el valor entero 0 y a la constante `sabado` el valor entero 6.

Es posible asignar un valor entero a alguno de los identificadores. A partir de ese valor se irán generando los valores de los identificadores siguientes incrementándose en 1.

```
enum meses {ene=1,feb,mar,may,abr,jun,jul,ago,sep,oct,nov,dic}
```

La constante `ene` tendrá valor 1, `feb` 2, `mar` 3, etc.

2.3. Variables

Desde el punto de vista de un programa, una variable es un dato cuyo valor puede cambiar a lo largo de la ejecución del programa. Para el compilador una variable es una zona de la memoria a la que se hace referencia mediante un identificador. El compilador necesita saber tres cosas de esa variable:

- Como se va a hacer referencia a esa zona de memoria
- Cuánto va a ocupar
- Cómo se va a interpretar el contenido de esa variable

El tipo de dato de la declaración de la variable le indica al compilador el tamaño del área de memoria dónde se va a almacenar la variable y qué significan la secuencia de bits que hay en dicho área. En nombre de la variable (un identificador) será la forma en que el programa haga referencia a esa zona de la memoria.

Además, la declaración puede incluir una expresión de inicialización con la que se da el valor inicial a esa zona de memoria.

La declaración sigue el siguiente formato:

`tipoDeDato identificador [=expresióInicialización]…`[5]

El *`tipoDeDato`* puede ser cualquier especificador de tipo de dato válido. La tabla 2.5 indica los distintos especificadores de tipo disponibles.

Tabla 2.5. Tipos de datos

Especificador de tipo
`char`
`signed char`
`unsigned char`
`short, signed short, short int,` o `signed short int`
`unsigned short, o unsigned short int`

[5] La declaración puede llevar también otros elementos como cualificadores de tipo, especificaciones sobre cómo almacenar la variable, etc. que se verán más adelante.

Especificador de tipo
`int, signed, or signed int`
`unsigned, or unsigned int`
`long, signed long, long int,` o `signed long int`
`unsigned long, or unsigned long int`
`long long, signed long long, long long int,` o `signed long long int`
`unsigned long long,` o `unsigned long long int`
`float`
`double`
`long double`

Además podemos utilizar como especificador de tipo:

- El tipo `void`, *un tipo incompleto* que se puede utilizar en aquellos casos dónde no sea requerido un valor, por ejemplo en funciones que no retornan nada y en funciones sin argumentos.
- El tipo `_Bool` que es un entero que se utiliza para almacenar los valores lógicos 1 (verdadero) o 0 (falso). También se puede utilizar el tipo `bool` si se incluye el archivo de cabecera `stdbool.h`.
- También se pueden incluir declaraciones de estructuras, uniones, enumeraciones o definiciones de tipo.
- ANSI C99 incluye también declaraciones para números complejos con `float _Complex`, `double _Complex` y long `double _Complex`.

El *`identificador`* será la forma de referenciar a la variable y deberá seguir las normas expuestas más arriba.

Además, se puede incluir una expresión de inicialización para indicar el valor inicial de la variable. Mientras no se dé un valor explícito a la variable (en la expresión de inicialización o en una operación de asignación) tendrá un valor indeterminado.

Ejemplos de declaraciones de variable:

```
int a;
double b,c,d;
long e;
char f = 'x';
char g = 45;
unsigned short h,i=230,k=4,l;
unsigned char m = i+5;
char* nombre ="Juan Ruiz"
char población[50] = "Madrid"
```

En los ejemplos anteriores, las variables `nombre` y `población` se han declarado como un dato de tipo cadena que se tratará en profundidad más adelante.

Como se vio más arriba, también es posible declarar variables cuyo valor no de modifica utilizando el cualificador de tipo `const`.

2.4. Operadores y expresiones

Una expresión es una secuencia de operadores y operandos que devuelve siempre un valor. Los operandos podrán ser variables, constantes u otras expresiones.

2.4.1. Operadores aritméticos

Los operadores **aritméticos binarios** (necesitan dos operadores) son la suma (+), la resta (-), la multiplicación (*), la división (/) y el módulo o resto de la división entera (%).

El operador % no puede aplicarse a datos reales. En la división, si los operandos son enteros, se hará la división entera, truncando la parte decimal. Si los operandos son reales se hará la división real. Por ejemplo

```
int a = 5,b=2;
float c,d=5,e=2;
c = a/b //División entera de 5 entre 2; c toma el valor 2
c = d/e //División real de 5 entre 2; c toma el valor 2.5
```

También se están disponibles los operadores **aritméticos unarios** (sólo necesitan un operador) de incremento (++) o decremento (--) que incrementan o decrementan en 1 a su operando. Pueden utilizarse como prefijos (el operador antes de la variable) o postfijos (el operador después de la variable). En el primer caso primero incrementa el valor de la variable y luego devuelve su valor, mientras que en el segundo primero devuelve su valor y luego incrementa.

```
int x,n=5;
//Cuando se  utiliza solo el operador el resultado del
//operador prefijo o postfijo es el mismo
++n; //Equivale a n = n + 1, n valdría 6
n--; //Equivale a n = n -1, n valdría 5 otra vez

//En otros contextos el resultado puede variar
n = 1;
x = ++n; //Incrementa n y luego devuelve su valor. x = 2
n = 1
x = n++; //Devuelve el valor de n y luego lo incrementa. x = 1
```

2.4.2. Operadores de relación y lógicos

Los **operadores lógicos** evalúan la relación entre dos datos del mismo tipo y devuelven un dato de tipo lógico:

- >, mayor qué
- <, menor qué
- >=, mayor o igual qué
- <=, menor o igual qué
- ==, igual a
- ¡=, distinto de

El dato que devuelven será 0 si la relación es falsa o 1 si es verdadera. Se trata de un tipo entero sin signo.

Los operadores lógicos evalúan la relación entre dos datos de tipo lógico y devuelven también un dato de tipo lógico.

- &&, devuelve verdadero si los dos operandos son verdaderos
- ||, devuelve verdadero si uno de los operandos es verdadero
- !, niega la expresión; devuelve falso si la expresión es verdadera y verdadero si la expresión es falsa

Hay que tener en cuenta que cualquier número entero se puede considerar un dato lógico, de forma que el valor 0 se considera falso y cualquier valor distinto de 0 verdadero. De esta forma,

- La expresión `3==3 && 3<4`, es verdadera puesto que ambas expresiones lógicas son verdaderas
- La expresión `1 && 3<4`, es verdadera puesto que ambas expresiones lógicas son verdaderas
- La expresión `0==3 && 3<4`, es falsa puesto que la primera expresión es falsa y la segunda verdadera
- La expresión `5 && 3<4`, es verdadera puesto que ambas expresiones lógicas son verdaderas
- La expresión `0 && 3 < 4`, es falsa puesto que la primera es false y la segunda verdadera

Expresión condicional

Existe una expresión que también utiliza expresiones lógicas, pero que devuelve un dato no necesariamente lógico, la expresión condicional. Su formato es:

expresiónLógica ? *expresión1* : *expresión2*

Si la expresión lógica es verdadera, se devuelve el valor de la *expresión1*, en caso contrario devuelve la *expresión2*.

```
//Si a>b, devuelve la cadena si,
//en caso contrario devuelve no
a>b?"si":"no"
```

2.4.3. Operadores de asignación

Se utilizan para la instrucción de asignación y su función es asignar a una variable el valor de una expresión.

variable = expresión

Se evalúa primero el valor de *expresión* y su resultado se asigna a la *variable*.

Tabla 2.6. Operadores de asignación

Operador	Explicación	Ejemplo
=	Asigna la expresión a la variable	`a = 5+6`, asigna a la variable a el valor 11

Operador	Explicación	Ejemplo
+=	Suma la expresión la variable y lo asigna a la misma variable	`a +=2`, equivale a `a = a +2`
-=	Resta la expresión a la variable y asigna el resultado a la misma variable	`a -=b*c`, equivale a `a= a - (b*c)`
*=	Multiplica la expresión a la variable y asigna el resultado a la misma variable	`a *= d` equivale a `a= a * d`
/=	Divide la expresión a la variable y asigna el resultado a la misma variable	`a /=b*c`, equivale a `a= a / (b*c)`
%=	Obtiene el resto de la variable entre la expresión y asigna el resultado a la misma variable	`a %=b`, equivale a `a= a % b`

2.4.4. Conversiones de tipo

Conversiones en expresiones aritméticas

Si una expresión tiene distintos tipos de operandos, éstos se convierten a un tipo determinado. En general, el dato de tipo *menor* se promueve al de tipo *mayor* antes de la operación y el tipo del resultado será el del tipo mayor. De esta forma, con las siguientes declaraciones:

```
float menor= 5;
double mayor = 3;
```

la operación `menor+mayor`, primero convertiría la variable `menor` a `double` y, a continuación realizaría la operación que devolvería un dato de tipo también `double`.

Si se trata de datos `unsigned` se seguirían básicamente estas reglas:

1. Si cualquier operando es `long double`, el otro se convierte también al mismo tipo
2. Si no, si cualquier operando es `double`, el otro se convierte también a `double`
3. Si no, si cualquier operando es `float`, el otro se convierte a `float`
4. Si no, se convierten los tipos `char` y `short` a `int`
5. Por último, si cualquier operando es `long`, se convierte el otro a `long`

Cuando hay operandos con signo y sin signo en una misma expresión, las reglas de conversión son más complejas ya que las comparaciones entre datos con signo y sin signo dependen de la máquina ya que están relacionadas con el tamaño de los tipos enteros.

Conversiones en expresiones de asignación

En una asignación el valor de la expresión (el operando situado a la derecha) se convierte al tipo de la variable (el operando situado a la derecha).

Si la expresión es de tipo entero, y la variable es de un tipo entero mayor (por ejemplo, si la expresión es `short` y la variable es `long`), la conversión se hace de forma automática, desechando los bits más significativos. Pero si ocurre lo contrario (la variable de un tipo entero más pequeño que la expresión) el resultado queda indefinido.

Si la expresión es de un tipo entero y la variable es de tipo real, el resultado se convierte a real de forma automática.

Si la expresión es de tipo real y la variable es de tipo entero ocurren varias cosas. Primero se trunca la parte decimal de la expresión. Si la parte entera se puede representar con el tipo de la variable se produce la conversión de forma automática. En caso contrario el resultado queda indeterminado.

Si la expresión de tipo `float` y la variable de tipo `double` se produce una conversión automática. En cambio si la expresión es de tipo `double` y la variable de tipo `float` el resultado se puede redondear o truncar dependiendo de la implementación.

El siguiente fragmento de código muestra el resultado de distintos tipos de conversión en asignaciones:

```
short s;
int i;
long l;
float f;
double d;

s = 12345;
l = s;              //l toma el valor 12345

l=1234567;
s=l;                //s toma el valor -10617 (u otro cualquiera)

f=l;                //f toma el valor 1234567.000000

f = 12345.456;
s = f;              //s toma el valor 12345
f = 1234567.456;
s = f;              //s toma el valor -10617 (u otro cualquiera)
```

Cast (moldes)

Los moldes o *cast* realizan la conversión forzosa de un tipo de dato en otro. El operador de cast es un operador unario que cuyo formato es el siguiente:

`(`*`tipoDeDato`*`)`*`expresión`*

Devuelve el valor de la *`expresión`* convertido al *`tipoDeDato`* especificado según las reglas anteriores. En *`tipoDeDato`* puede ir cualquiera de los especificadores de tipo de la tabla 2.5.

Se pueden utilizar, por ejemplo, para forzar a realizar una división real:

```
int i,a=5,b=2;
float f;

i = a/b;          //Realiza la división entera. Devuelve 2

f = (float)a/b;  //Realiza la división real. Devuelve 2.500000
```

2.4.5. Prioridad de los operadores

Una expresión puede estar compuesta por varios operadores incluso de distinto tipo. A la hora de evaluarla se sigue un orden de prioridades que aparece en la tabla 2.7 (en la tabla faltan

algunos operadores que todavía no se han visto). A igual prioridad las expresiones se evalúan de izquierda a derecha. Si algunas operaciones están entre paréntesis, estas se harán primero; si hay varios paréntesis anidados se realizarán primero las operaciones de los paréntesis internos.

Tabla 2.7. Prioridad de los operadores

Prioridad	Operador	Explicación
Más prioridad	! ++ -- + - (*cast*) sizeof	Operadores unarios. No lógico, incremento, decremento, más y menos unario (signo), moldes. El operador `sizeof` devuelve el tamaño en bites de un dato.
	* / %	Operadores multiplicativos
	+ -	Operadores aditivos
	< <= > >=	Operadores de relación
	== ¡=	Igualdad y desigualdad
	&&	Y lógico
	||	O lógico
Menos prioridad	? :	Expresión condicional
	= += -= *= /= %=	Operadores de asignación

2.5. Entrada y salida

Las operaciones de entrada y salida no forman parte intrínseca de C sino que se encuentran en una de las bibliotecas estándar de funciones que sí están definidas por el estándar ANSI. Las funciones para entrada y salida se encuentran en el archivo de cabecera `stdio.h`, por lo que, si se utilizan, habría que establecer la directiva `#include` del preprocesador:

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

Las funciones de entrada y salida gestionan los flujos de datos (*stream*). Un flujo una secuencia de datos de entrada o salida que puede estar asociada a un archivo u a otro periférico de entrada y salida. El archivo de entrada estándar (`stdin`) es el teclado y el de salida estándar (`stdout`) el monitor.

La forma más simple de entrada y salida de datos permite leer por teclado o escribir por pantalla un único carácter. La función `getchar()` devuelve un dato de tipo `int` que es el carácter leído por teclado. La función `putchar(int)` permite sacar por pantalla el carácter que se pasa como argumento.

Estas dos funciones son especialmente útiles para la lectura y escritura de archivos de texto. Sin embargo, para la entrada y salida por pantalla se suelen utilizar las funciones `scanf` y `printf` que permiten la entrada y salida con formato.

2.5.1. Salida con formato: printf

La función `printf` se encarga de traducir secuencias de bits a caracteres y sacarlos por la salida estándar. El formato de la función es el siguiente:

`int printf(char *formato, argumento1, argumento2,…)`[6]

La función convierte y da formato a los argumentos según las especificaciones de la cadena de formato y los saca por la salida estándar. Devuelve un entero con el número de caracteres impresos.

formato es una cadena de caracteres que contiene dos tipos de caracteres:

- Caracteres ordinarios que se copiarán directamente a la salida. Por ejemplo, `printf("Hola, mundo!\n")`, sacará por pantalla la cadena seguida de un salto de línea.
- Especificaciones de conversión, que comienzan por el carácter % y termina con un carácter de conversión.

Existe una correspondencia entre el número de especificaciones de conversión y el número de argumentos, de forma que la primera especificación convertirá y dará el formato al primer argumento, la segunda al segundo, etc. Por ejemplo:

```
printf("La suma de %i + %i es igual a %i", 3,5,3+5);
```

sacaría por pantalla…

```
La suma de 3 + 5 es igual a 8
```

Los grupos de caracteres que no van precedidos por el símbolo % se sacan a la pantalla sin modificar; las especificaciones de conversión de la cadena de formato(en este caso tres grupos `%i` que convierten un número entero) son reemplazadas por los argumentos en el orden en que aparecen.

Además del símbolo de %, las especificaciones de conversión pueden tener en este orden:

- Cero o más caracteres indicadores que modifican la especificación de conversión. Puede ser entre otros:
  - un signo −, que especifica que el dato se debe ajustar a la izquierda
  - un signo +, que fuerza a que salga el signo + o - en los datos numéricos, según sea positivo o negativo. Si no aparece este carácter sólo se indican los números negativos
  - Un espacio en blanco. Si el número es negativo saca el es signo menos, en caso contrario saca un espacio en blanco
- Un número que indica el ancho mínimo del campo. Si el dato que se saca es menor que ese ancho, se rellena con espacios en blanco a la izquierda o a la derecha, según sea el carácter indicador

[6] En la documentación que existe sobre C y otros lenguajes al indicar el formato de las funciones se especifica el tipo de dato que devuelve, el nombre de la función y el número tipo y orden de los argumento. En el caso de `printf` se señala que la función devuelve un dato de tipo entero (el primer `int`) y que recibe una cadena de caracteres con el formato (`char *` indica que es una cadena de caracteres) y uno o más argumentos. Al no indicar nada podrán ser de cualquier tipo y los puntos suspensivos se utilizan para decir que los argumentos se pueden repetir.

- Un número, precedido por el carácter punto (.) que indica la precisión, número mínimo de dígitos para valores enteros, el número de decimales para datos reales o el número máximo de caracteres para cadenas
- Un carácter *modificador de longitud.* Podrá ser, entre otros:
    - `h`, en datos enteros se convierten a un `short` o `unsigned short` antes de la salida
    - `l`, en datos enteros se convierten en `long` o `unsigned long` antes de la salida
    - `L`, en datos reales se convierte a `long double` antes de la salida.
- Un *carácter de conversión*. Se trata del único carácter obligatorio en las especificaciones de conversión. La tabla 2.8 muestra los distintos caracteres de conversión.

Table 2.8 Caracteres de conversión

Carácter	Significado
`d i`	El argumento entero se convierte a un número decimal con signo
`o`	El argumento entero sin signo se convierte a un número octal
`X x`	El argumento entero sin signo se convierte a un número hexadecimal utilizando la letras mayúsculas (X) o minúsculas (x)
`u`	El argumento entero sin signo se convierte a un número decimal sin signo
`c`	El argumento entero se escribe como un carácter
`s`	Saca los caracteres de una cadena hasta el carácter nulo (\0)
`f`	El argumento de tipo `double` se convierte a un número real con el número de decimales indicado en la precisión (por defecto 6)
`E e`	El argumento de tipo `double` se convierte a notación exponencial utilizando una `E` o una `e` para el exponente. El número de decimales viene dado por la precisión. Por omisión es 6
`G g`	Es equivalente a `E` o `e` si el exponente del argumento es menor que -4 o mayor o igual que la precisión, en caso contrario es equivalente a `f`. Los ceros no significativos de la parte decimal se eliminan.

2.5.2. Entrada de datos con formato: scanf

La función `scanf` se encarga de recoger caracteres por la entrada estándar (el teclado) los transforma en datos y los almacena en una o varias variables. Su formato es el siguiente:

`int scanf(*char` *`formato`*`, `*`puntero1`*`, `*`puntero2`*`, …)`

`scanf` lee caracteres de la entrada estándar, los interpreta según las especificaciones de la cadena formato y almacena los resultados en los siguientes argumentos. La función devuelve el número de elementos que ha logrado convertir.

Los argumentos que hay después de la cadena de formato deben ser **punteros**[7] o **referencias** a las variables dónde se desea almacenar el valor leído por teclado. Para datos numéricos (incluidos los `char`), para utilizar un puntero a la variable se utiliza el carácter & seguido del nombre de variable. Las cadenas ya son por sí mismas punteros a caracteres, por lo que no es necesario incluir el símbolo &.

[7] Un puntero es un tipo de dato especial que contiene la dirección de memoria dónde se almacena una variable.

formato es una **cadena de conversión** que contiene las especificaciones de conversión utilizadas para convertir la secuencia de caracteres de entrada al tipo de dato de la variable a la que apuntan los punteros. De esta forma,

```
int a;
scanf(“%i”,&a)
```

leería caracteres de la entrada hasta el siguiente espacio en blanco (se considera espacio en blanco tanto el carácter de espacio en blanco como los caracteres de tabulación, nueva línea, retorno de carro, tabulador vertical o avance de página), los convertiría en un dato de tipo entero y haría la asignación de ese dato a la variable `a`.

La cadena de conversión podrá contener:

- Caracteres en blanco o tabuladores que se ignoran
- Caracteres ordinarios (distintos de blanco o del carácter %) que debería coincidir con el siguiente carácter distinto de blanco del flujo de entrada.
- Opcionalmente, especificaciones de conversión, formadas por:
    - El carácter %
    - Un carácter optativo (*) de supresión de asignación que indica que el dato correspondiente a esa cadena de conversión se leerá pero no hará la asignación a la variable correspondiente
    - Un número optativo que indica el ancho del campo
    - Opcionalmente, un carácter *modificador de longitud*: h para datos de tipo `short`, `l` para datos `long` y `L` para datos `double`
    - Un *carácter de conversión* que indica el tipo de dato que se leerá y que deberá coincidir con el tipo de variable del argumento según la tabla 2.9.

Tabla 2.9. Caracteres de conversión para scanf

Tipo de dato	Especificación de conversión
`char`	`%c`
`short`	`%hd`
`int`	`%d or %i`
`long`	`%ld`
`long long`	`%lld`
`float`	`%f or %e`
`double`	`%lf or %le`
`long double`	`%Lf or %Le`
`string`	`%s`

2.10. Ejemplos de scanf

Código	Entrada	Resultado
`int a,b,c;` `scanf("%i %i %i",&a, &b ,&c);`	1 2 3(INTRO)	a =1, b=2, c=3
`int n,i; float f; char *s="";` `n=scanf(“%i %f %s”,&i,&f,s);`	3 3.45 Adios	n=3,i = 3, f=3.45, s=”Adios”
`int dia, mes, año;` `scanf("%i/%i/%i",&dia,&mes,&año);`	3/12/2010	dia=3, mes=12, año=2010
`int n,i,j; float x;`	56789 0123 56a72	n=3,i=56,x=789.0,i=56,

Código	Entrada	Resultado
`n=scanf("%2d%f%*d %i", &i, &x, &j);` `char c = getchar();`		c='a'

En el último ejemplo, `n=3`, ya que se han leído 3 datos correctamente; `i=56` ya que en la cadena de conversión del primer entero se indica que la longitud del dato es dos caracteres; `x=789.0` ya que lee los siguientes números y hasta el blanco y los mete en un dato de tipo real, el siguiente dato `0123` lo ignora por el carácter de supresión (`*`), `j=56`, ya que lee los caracteres legales de la siguiente cadena. En el flujo de datos, el siguiente carácter es una `a`, que se asigna con `getchar()` a la variable `c`.

Errores comunes en scanf

- Cada variable leída con `scanf` debe tener su correspondiente cadena de especificación de conversión
- Las variables leídas con `scanf` deben ser punteros. Esto significa que deberán aparecer precedidas con el carácter `&`, que indica que lo que devuelve es la dirección de memoria dónde se almacena la variable. Esto es cierto excepto para variables de cadena ya que éstas se tratan de un puntero a un array de caracteres
- Los espacios en blanco se ignoran y se consideran como un separador entre variables
- El formato `%f` indica un dato de tipo `float`. Para datos de tipo `double` hay que utilizar `%lf`
- Cuando falla una entrada de datos, `scanf` intenta asignar a la variable los datos válidos hasta encontrar un dato erróneo. En ese caso en el buffer de entrada seguirá almacenado el siguiente carácter que saldrá en la siguiente operación de entrada (`scanf`, `getchar`, etc.). El buffer de entrada se puede vaciar llamando a la función `fflush(stdin)`

2.6. Estructura de un programa C

Un archivo de código fuente de C tiene la siguiente estructura

```
Directivas del preprocesador

Prototipos de funciones

Declaración de variables externas

Declaración de funciones

int main(void){
   sentencias…
}

Declaración de funciones
```

2.6.1. Directivas del preprocesador

Al compilar un programa C, lo primero que hace es traducir una serie de especificaciones que indican al compilador algunos de los pasos previos que debe dar. Las directivas más frecuentes son `#include` y `#define`.

Inclusión de archivos

La primera permite incluir el contenido de un archivo en el programa fuente antes de la compilación. Puede tomar el formato:

`#include <`*`nombreArchivo`*`>`

O

`#include "`*`nombreArchivo`*`"`

Ambas insertan el archivo *`nombreArchivo`* en ese punto del programa fuente. Si el nombre del archivo se encierra entre comillas, se buscará en el mismo lugar dónde se encuentra el programa fuente. De no encontrarse o si el nombre del archivo se encierra entre los símbolos < y > la búsqueda se realiza en una ruta predefinida por la instalación (en el caso de lcc, el archivo se buscará en la carpeta `include` dentro de la carpeta de instalación de lcc).

La utilidad principal de `#include` es informar al compilador que se van a utilizar una serie de bibliotecas de funciones, los llamados archivos de cabecera (*headers*) como `<stdio.h>`, `<stdbool.h>`, etc.

Sustitución de macros

La directiva `#define` sustituirá en el programa fuente todas las apariciones de una cadena por otra.

`#define` *`identificador textoDeRemplazo`*

Sustituirá, dentro del programa fuente, todas las apariciones de *`identificador`* por *`textoDeRemplazo`*.

```
#define IVA 0.18
```

```
#define leerFlotante scanf("%f",f)
#define maximo(A,B) ((A) > (B) ? (A) : (B))
```

El último ejemplo incluye dos argumentos, `A` y `B`. Por ejemplo,

```
mayor = maximo(numero1,numero2);
```

Si `numero1` es mayor que `numero2`, `mayor` toma el valor de `numero1`, en caso contrario toma el valor de `numero2`.

2.6.2. Declaración de funciones y variables externas.

Un programa C va a estar formado por varias funciones que incluyen todas las instrucciones del programa. La declaración de funciones se verá más adelante.

Cada función puede tener un **prototipo de la función** que indica al compilador las funciones que se van a utilizar, el tipo de dato que devuelven y el número, orden y tipo de los argumentos que va a utilizar.

Como se ha dicho antes existe una función especial que se llama `main`. La función `main` será el punto de entrada del programa. Un programa ejecutable tendrá una función `main` en alguno de los módulos de código fuente que contiene. Si un programa tiene más de una función `main`, el compilador lo detectará y asumirá que una de ellas es el punto de entrada.

Fuera de la declaración de funciones, se podrán declarar lo que se llama **variables externas**, variables que podrán ser utilizadas por todas las funciones y cuya utilidad se verá también más adelante.

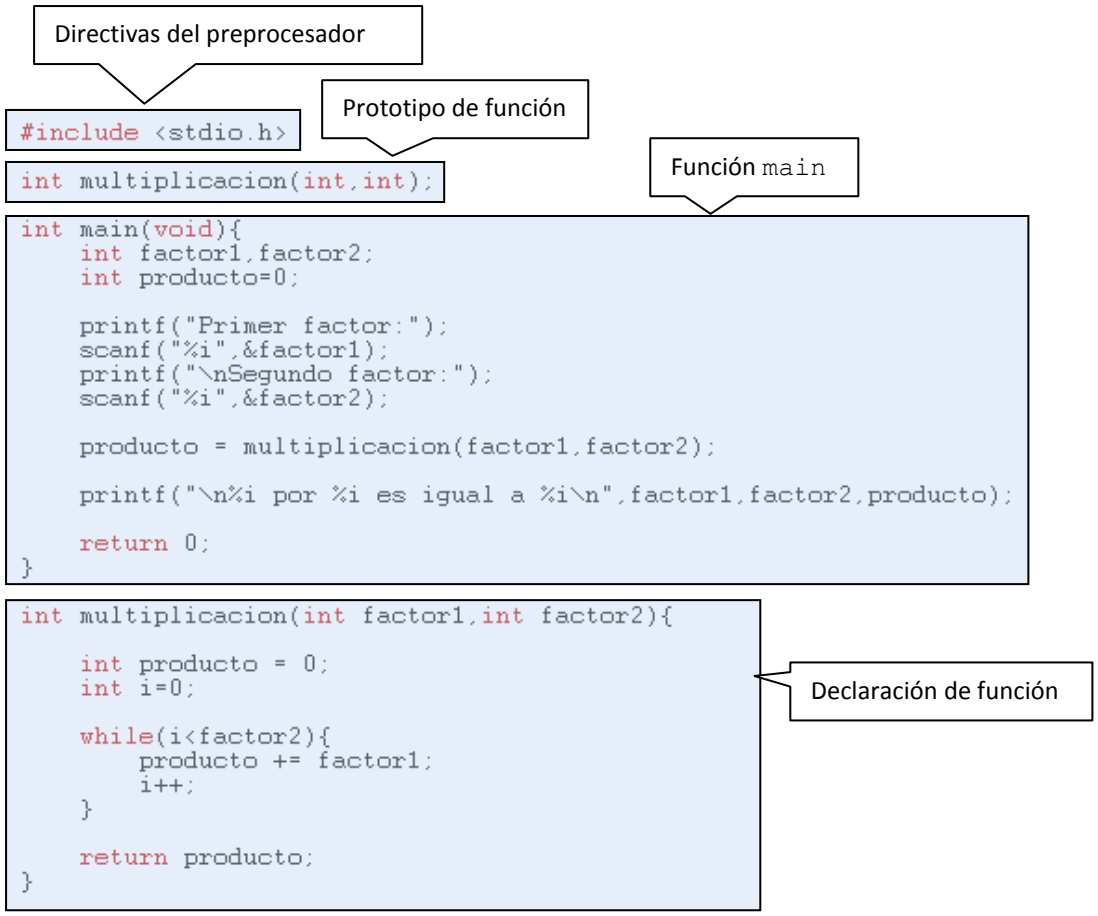

Figura 2.1. Partes de un programa en C

Cada una de las declaraciones de funciones va a contener una serie de sentencias o proposiciones,

2.6.3. Sentencias o proposiciones

Las sentencias o proposiciones de un programa son las instrucciones que el programa dará al ordenador. Una expresión (hay que recordar que una expresión puede ser una expresión aritmética, lógica, de asignación, una llamada a una función, declaraciones de variables, constantes o enumerados, etc.) se convierte en una sentencia o proposición se va seguida de un punto y coma. En C el punto y coma indica el final de una sentencia.

Son sentencias:

```
int  producto=0;
producto += factor1;
printf("\nSegundo factor:");
factor1+factor2;
```

La última sentencia (`factor1+factor2;`) no tiene ningún efecto. En algunos lenguajes, este tipo de proposiciones son erróneas. En C es posible incluirlas dentro del código fuente aunque no tengan efecto. El compilador avisará de esta circunstancia.

Es posible agrupar varias proposiciones encerrándolas entre llaves. Un grupo de sentencias encerradas entre llaves forma una **proposición compuesta** o **bloque**. Un bloque no llevará el punto y coma después de la última llave. En la figura 2.1 se observa que la propia función

`main` está formada por un bloque de sentencias. Todas las instrucciones de control de flujo (por ejemplo la sentencia `while` del ejemplo) también incluyen un bloque se sentencias.

2.7. Ejercicios

1. Calcular la longitud y el área de una circunferencia a partir del radio de la misma
2. Convertir una cantidad de pesetas a euros.
3. Convertir una cantidad en cualquier divisa a euros.
4. Se desea calcular el capital final de una cantidad de euros colocada en un banco a un interés compuesto determinado durante un periodo de años. Tanto el capital inicial, como el tipo de interés y el número de años se introducirán mediante teclado.
   Para resolver el problema habrá que aplicar la fórmula:

   $$CapitalFinal = CapitalInicial(1 + interés)^{años}$$

   Nota: la función `pow(x,y)` incluida en el archivo de cabecera `math.h` permite obtener el valor de x^y.