Introducción al lenguaje de programación C

Féliix García Carballeira
Grupo de Arquitectura de Computadores
Universidad Carlos III de Madrid
felix.garcia@uc3m.es

Presentación

Este pequeño manual de C pretende ser una guía rápida para aprender a programar en C. Está pensado para alumnos que ya poseen conocimientos básicos de programación, estructuras de datos, algoritmos y arquitectura de computadores. No pretende, por tanto, ser una guía detallada ni considerarse un libro de programación como tal.

Enero de 2020

Índice general

Capítulo 1

Introduccioin al lenguaje C

En este capítulo se realiza una pequeña introducción al lenguaje de programación C.

1.1. Historia de C

La historia de C puede resumirse en los siguientes puntos:

- Muchas ideas provienen de los lenguajes BCPL (*Martin Richards*, 1967) y B (*Ken Thompson*, 1970).
- C fue diseñdo originalmente en 1972 para el sistema operativo UNIX en el DEC PDP-11 por *Dennis Ritchie* en los Laboratorios Bell.
- El primer libro de referencia de C fue *The C Programming Language* (1978) de Brian Kernighan y Dennis Ritchie.
- En 1989 aparece el estándar ANSI C, que está soportado por la casi totalidad de los compiladores.
- En 1990 aparece el estándar ISO C. WG14 se convierte en el comité oficial del estándar ISO C.
- En 1994 WG14 crea las primeras extensiones a ISO C.
- En 1999 aparece el estándar C99.
- En 2011 se publica el estándar C11 (ISO/IEC 9899:2011).
- En 2018 se publica el último estándar del lenguaje C18 (ISO/IEC 9899:2018). El último borrador de este estándar se puede obtener en `http://www.iso-9899.info/wiki/The_Standard`.

La figura 1.1 muestra la historia de C y su relación con el sistema operativo UNIX.

1.2. Características de C

Las principales características del lenguaje C pueden resumirse en los siguientes puntos:

- C es un lenguaje de propósito general ampliamente utilizado.
- Presenta características de **bajo nivel**: C trabaja con la misma clase de objetos que la mayoría de los computadores (caracteres, números y direcciones). Esto permite la creación programas eficientes.

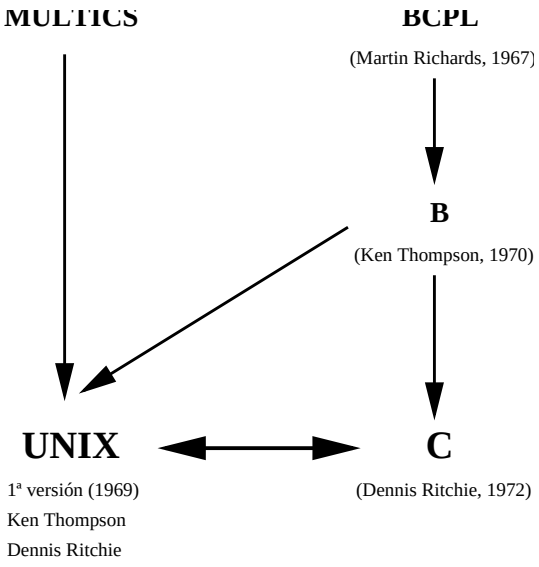

Figura 1.1: Historia de C

- Está estrechamente asociado con el sistema operativo UNIX. UNIX y su software fueron escritos en C.
- Es un lenguaje adecuado para *programación de sistemas* por su utilidad en la escritura de sistemas operativos.
- Es adecuado también para cualquier otro tipo de aplicación.
- Es un lenguaje relativamente pequeño, solo ofrece sentencias de control sencillas y funciones.
- No ofrece mecanismos de E/S (entrada/salida). Todos los mecanismos de alto nivel se encuentran fuera del lenguaje y se ofrecen como funciones de biblioteca.
- Permite la creación de programas portables, es decir, programas que pueden ejecutarse sin cambios en multitud de computadores.
- Permite el uso de de técnicas de programación estructurada y diseño modular.
- A pesar de su bajo nivel, es el lenguaje más portado a todo tipo de plataformas. Existen compiladores de C para casi todos los sistemas.

1.3. Inconvenientes de C

Aunque C ofrece innumerables ventajas, también presenta una serie de inconvenientes:

- No es un lenguaje *fuertemente tipado*.
- Es bastante permisivo con la conversión de datos.
- Sin una programación metódica y cuidadosa puede ser propenso a errores difíciles de encontrar.
- La versatilidad de C permite crear programas difíciles de leer, como el que se muestra a continuación.

```
#define _ -F<00||--F-00--;
int F=00,00=00;
main(){F_00();printf("%1.3f\n",4.*-F/00/00);}F_00()
{
                    _-_-_-_
              _-_-_-_-_-_-_-_-_-_
           _-_-_-_-_-_-_-_-_-_-_-_-_
         _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
        _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
        _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
       _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
       _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
       _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
       _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
        _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
        _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
         _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
           _-_-_-_-_-_-_-_-_-_-_-_-_
               _-_-_-_-_-_-_-_
                   _-_-_-_
    }
```

1.4. Entorno de desarrollo de C

Para poder realizar un programa en C, se necesita un entornos de desarrollo de C, que debe incluir, entre otros:

- Editor de texto.
- Compilador.
- Ficheros de cabecera.
- Ficheros de biblioteca.
- Enlazador.
- Depurador.

1.5. Primer programa en C

En esta sección se presenta un primer programa en C, que imprime por pantalla el mensaje: "Primer programa en C".

```c
#include <stdio.h>

int main(void)
{
      /* Primer programa en C. Esto es un comentario */
       // este es otro comentario

       printf("Primer programa en C. \n");

       return 0;
}
```

Todo programa debe contener una función denominada `main()` que es la función que se ejecuta en todo programa inicialmente. Esta función devuelve un valor de tipo entero (`int`). Esta función puede aceptar o no argumentos. En este primer ejemplo, la función no acepta ningún argumento, lo que se indica con la palabra reservada `void`. La función `printf` se utiliza para imprimir un texto en la pantalla, en este caso `Primer programa en C`. La función acaba cuando se ejecuta la sentencia `return` que se utiliza para finalizar la ejecución de las funciones devolviendo un valor. El programa también se puede terminar ejecutando la función de biblioteca `exit()`.

1.5.1. ¿Cómo crear el programa?

Para crear el programa anterior es necesario :

- Editarlo con un editor de texto (en Linux: *vi*, *pico*, *emacs*, etc.)
- Grabar el programa en un fichero, `ejemplo.c`.

1.5.2. ¿Cómo compilar y ejecutar el programa?

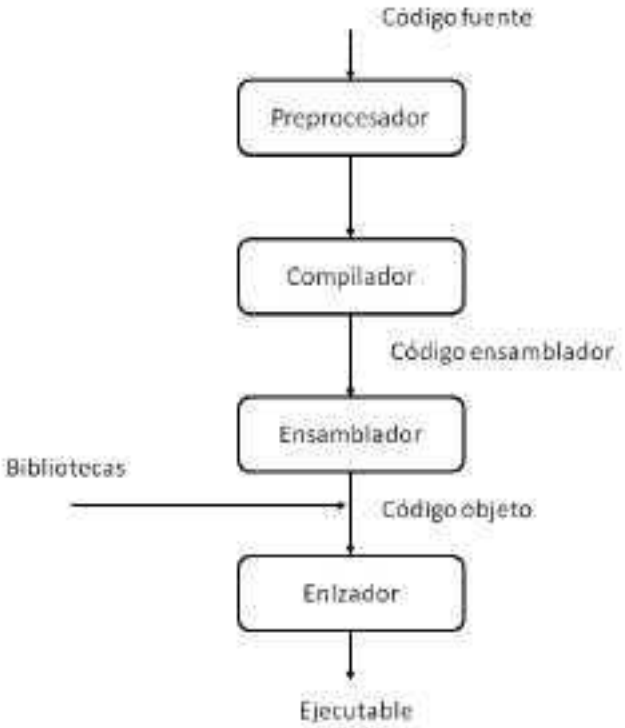

Figura 1.2: Modelo de compilacion de C

Un **compilador de C** permite:

- Analizar y detectar errores en el código fuente.
- Convertir un programa escrito en C en código ejecutable por el computador.

El mandato de compilación básico (utilizando la línea de mandatos) es el siguiente:

```
cc ejemplo.c
```

Si se utiliza el compilador gcc (compilador de libre distribución ampliamente utilizado), el mandato sería el siguiente:

```
gcc ejemplo.c
```

Este mandato permite crear los siguientes ficheros:

- Genera el código objeto `ejemplo.o`, que incluye el código que resulta de compilar un único fichero fuente.
- Genera el ejecutable `a.out`, que se obtiene a partir de uno o varios ficheros objeto.

El programa se ejecuta tecleando `a.out`.

El mandato `gcc -c ejemplo.c` genera el fichero objeto `ejemplo.o`. El mandato `gcc ejemplo.o -o ejemplo` genera el ejecutable `ejemplo`. En este caso el programa se ejecuta tecleando `ejemplo`.

El modelo de compilación de C se puede ver en la figura 1.2.

Capítulo 2

Fundamentos de C

Este capítulo presenta las principales características del lenguaje C.

2.1. Comentarios

Los comentarios comienzan con `/*` y finalizan con `*/`. Algo importante a tener en cuenta es que no se pueden anidar, es decir, dentro de un comentario no puede aparecer el símbolo `/*`.

También se pueden poner comentarios en una línea utilizando `//`. Todo lo que aparece a partir de estos símbolos se consideran comentarios hasta el final de la línea.

El siguiente programa ilustra la utilización de los comentarios en C en el primer programa `Hola Mundo`.

```
#include <stdio.h>

int main(void)
{
        /*********************************/
        /*      Esto es un comentario      */
        /*********************************/

        // esto es tambien otro comentario

        printf("Hola Mundo\n");
        return 0;
}
```

En este programa se puede apreciar:

- Se incluye el archivo de cabecera `stdio.h`. Con la directiva `include` se indica al preprocesador que se incluya todo el texto de este fichero en el fichero `HolaMundo.c`. Este archivo de cabecera incluye todas las declaraciones de funciones de la biblioteca estándar relacionada con la entrada-salida [1]

- La función `main`, que se corresponde con el punto de entrada al programa. Es la primera función en ser llamada. El prototipo de la función (nombre, argumentos, tipo de los argumentos y tipo de retorno de la llamada) indica que la función no acepta ningún argumento (`void`) y que la función devuelve un valor de tipo entero (`int`). Más adelante se indicará otra declaración posible para la función `main` que permite pasar argumentos a dicha función.

[1] Los archivos de cabecera incluyen declaración de tipos y de funciones, pero no el código de dichas funciones. El código asociado a estas funciones se encuentran en otros archivos fuente (.c) o en bibliotecas

- La sentencia `return` indica el punto de salida de la función. En este caso la función devuelve el valor 0. En cuanto una función ejecuta la sentencia `return` finaliza su ejecución. La finalización de un programa también se puede hacer utilizando la función de biblioteca `exit`. En este caso la sentencia `return 0` sería equivalente a la ejecución de la función `exit(0)`

- Las llaves `{ }` indican el comienzo y el fin de un bloque de sentencias. En el ejemplo anterior el comienzo y el fin de la función.

- Todas las sentencias finalizan con punto y coma.

2.1.1. Compilación y generación de ejecutables

Como se indicó anteriormente, existen cuatro fases en la generación de un ejecutable a partir de un fichero fuente escrito en C:

- **Preprocesado**: procesa todas las directivas del preprocesador. En el caso anterior `include`.

- **Compilación**: traduce el código fuente a código ensamblador.

- **Ensamblado**: ensambla a código objeto. Esta fase está normalmente integrada con la compilación.

- **Enlazado**: resuelve las referencias externas entre distintos módulos objetos y bibliotecas para generar el fichero ejecutable.

El preprocesador se invoca utilizando el compilador `gcc`:

```
gcc HolaMundo.c  -E  -o hola.i
```

Este mandato preprocesa el fichero `HolaMundo` y obtiene como resultado el fichero preprocesado `hola.i`, que será el punto de entrada al compilador. El fichero obtenido tiene 754 líneas.
El código ensamblador se puede obtener de la siguiente forma:

```
gcc HolaMundo.c  -S  -o hola.s
```

La compilación se puede realizar de la siguiente forma:

```
gcc HolaMundo.c  -c -o hola.o
```

De esta manera se obtiene el fichero objeto `hola.o`.
La generación del fichero ejecutable se puede realizar:

```
gcc HolaMundo.c  -o hola
```

El mandato anterior genera el ejecutable `hola`. Si no se especifica el nombre del ejecutable:

```
gcc HolaMundo.c
```

El nombre por defecto que tendrá el ejecutable será `a.out`.

2.2. Palabras reservadas

Las palabras reservadas que incluye el lenguaje C son las siguientes:

auto	if	unsigned
break	inline	void
case	int	volatile
char	long	while
const	register	_Alignas
continue	restrict	_Alignof
default	return	_Atomic
do	short	_Bool
double	signed	_Complex
else	sizeof	_Generic
enum	static	_Imaginary
extern	struct	_Noreturn
float	switch	_Static_assert
for	typedef	_Thread_local
goto	union	

Cuadro 2.1: Palabras reservadas en C

2.3. Tipos de datos elementales

Los principales tipos de datos de C son los siguientes:

- `_Bool`: tipo de datos utilizado para almacenar un valor lógico 0 (false) o 1 (true).
- `char`: tipo de datos básico para almacenar un objeto de tipo carácter. Su tamaño típico es el byte.
- Enteros con sigo: `signed char`, `short int`, `int`, `long int` y `long long int`. Un tipo de datos `signed char` ocupa el mismo tamaño que el tipo `char`. Un objeto de tipo `int` ocupa típicamente el tamaño que ocupa el ancho de palabra en un computador (aunque el tamaño real no está especificado en el estándar). Los valores mínimos y máximos que se pueden almacenar en variables de estos tipo se encuentran definidos en el archivo de cabecera `<limits.h>`. En la Tabla 2.2 se muestran los valores típicos de estas variables (obtenidas en un computador de 64 con sistema operativo Linux y compilador gcc). De forma abreviada se puede utilizar el tipo `short` por `short int` y `long` por `long int`.
- Enteros sin signo: `unsigned`. Se puede aplicar a cualquiera de los tipos anteriores. Así por ejemplo `unsigned long int`, hace referencia a un tipo sin signo que tiene el mismo tamaño básico que tipo `long int`.
- Números en coma flotante: `float` que permite definir números en coma flotante de simple precisión según el estándar IEEE 754, `double` para números en doble precisión (64 bits) y `long double` para el formato de doble precisión de 128 bits.
- Números complejos: `float _Complex`, `double _Complex` y `long double _Complex`.

El tipo `intN_t` designa a un entero con sigo de tamaño `N bits`. El tipo `unitN_t` designa a un entero sin sigo de tamaño `N bits`.

Además el tipo `void` comprende un conjunto vacío de valores. Cuando se declara una función de tipo `void`, lo que se está indicando es que la función no devuelve ningún valor.

Tipo	Significado	Tamaño en bytes
`char`	caracter	1
`short int`	entero corto	2
`int`	entero	4
`long int`	entero largo	8
`long long int`	entero largo	8
`unsigned char`	caracter sin signo	1
`unsigned`	entero sin signo	se aplica a cualquier entero
`float`	coma flotante de 32 bits	4
`double`	coma flotante de 64 bits	8
`long double`	coma flotante de 128 bits	16

Cuadro 2.2: Tamaños de los principales tipos de C en un computador de 64 bits utilizando el compilador gccf

En las primeras versiones de C, no existía el tipo booleano. Un valor igual a 0 se interpretaba como falso y un valor distinto de 0 como verdadero. En las últimas versiones de C se han añadido el tipo boolean `_Bool`.

En el siguiente programa se puede ver su uso. Si se desea utilizar las constantes `false` (valor 0) o `true` (valor distinto de 0) es necesario incluir el archivo de cabecera `stdbool.h`.

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

int main(void)
{
        _Bool a;

         a = true;

         if (a == true)
             printf("true\n");
        return 0;
}
```

En las últimas versiones de C también existe como tipo básico el número complejo `_Complex`. Se pueden definir variables de tipo:

- `_Complex`: parte real e imaginaria de tipo `float`.
- `double _Complex`: parte real e imaginaria de tipo `double`.
- `long double_Complex`: parte real e imaginaria de tipo `long double`.

En el siguiente programa se puede ver su uso:

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

int main(void)
{
    double _Complex c1, c2, c3;

    c1 = 1.0 + 3 * I;
```

```
    c2 = 1.0 + 3 * I;

    c3 = c1+c2;

    printf("La suma es:  %.2f %.2f\n", creal(c3), cimag(c3));
    return 0;
}
```

La función `creal` permite obtener la parte real del número y `cimag` la parte imaginaría.

El siguiente programa permite imprimir el tamaño que ocupa cada uno de los tipos básicos de C. Se emplea el operador `sizeof` de C que devuelve el número de bytes que ocupa un determinado tipo de datos o variable (más adelante se describirá el uso de la función `printf`).

```
#include <stdio.h>

int main(void)
{
        printf("Tamaño que opupa un char: %ld \n", sizeof(char));
        printf("Tamaño que opupa un short: %ld \n", sizeof(short int));
        printf("Tamaño que opupa un int: %ld \n", sizeof(int));
        printf("Tamaño que opupa un long int: %ld \n", sizeof(long int));
        printf("Tamaño que opupa un long long int: %ld \n", sizeof(long long int));
        printf("Tamaño que opupa un float: %ld \n", sizeof(float));
        printf("Tamaño que opupa un double: %ld \n", sizeof(double));
        printf("Tamaño que opupa un long double: %ld \n", sizeof(long double));

        return 0;
}
```

2.4. Constantes

Los distintos tipos de constantes en C son:

- Caracteres: **'a'**, **'b'**
- Cadenas de caracteres: **"hola"**.
- Valores enteros, que pueden expresarse de distintas formas:
  - Notación decimal: 987
  - Notación hexadecimal: **0x**25 o **0X**25
  - Notación octal: **0**34
  - Enteros sin signo: 485**U** o 485**u**
  - Enteros de tipo `long int`: 485**L** o 485**l**
  - Enteros con signo de tipo `long long int`: 485**LL** o 485**ll**
  - Enteros sin signo de tipo `long long int`: 485**ULL** o 485**ull**
  - Valores negativos (signo menos): -987
- Valores reales (coma flotante):
  - Ejemplos: **12**, **14**, **8.**, **.34**

Carácter	Argumentos	Resultado
`d, i`	entero	entero decimal con signo
`u`	entero	entero decimal sin signo
`o`	entero	entero octal sin signo
`x,X`	entero	entero hexadecimal sin signo
`f`	coma flotante	coma flotante (simple y doble) con punto y con signo
`e, E`	coma flotante	coma flotante (simple y doble) en formato exponencial con signo
`Lf`	coma flotante	coma flotante (doble precisión) con punto y con signo
`Le, LE`	coma flotante	coma flotante (doble precisión) en formato exponencial con signo
`c`	carácter	un carácter
`s`	cadena de caracteres	cadena de caracteres
`%`		imprime %
`p`	* void	El argumento `void *` se imprime en hexadecimal.
`ld,lu,lx,lo`	long int	long int
`lld,llu,llx,llo`	long long int	long int

Cuadro 2.3: Principales especificadores de formato

- Notación exponencial: **.2e+9**, **1.04E-12**
- Valores negativos (signo menos): **-12 -2e+9**
- Las constantes correspondientes a números en coma flotante de tipo `double` no llevan sufijo: 8.0.
- Una constante de tipo `float` se indica con el sufijo `f` o `F`: 8.0**f**.
- Una constante de tipo `long double` se indica con el sufijo `l` o `L`: 8.0**L**.

2.5. La función `printf`

Esta función permite imprimir información por la salida estándar. El formato de dicha función es:

```
printf(formato, argumentos);
```

donde `formato` especifica entre otras cosas, los tipos de datos que se van a imprimir y `argumentos` los datos o valores a imprimir.

A continuación se presenta un ejemplo de utilización. Observe que por cada valor o expresión a imprimir existe un especificador de formato (utiliza `%`).

```
printf("Hola mundo\n");
printf("El numero 28 es %d\n", 28);
printf("Imprimir %c %d %f\n", 'a', 28, 3.0e+8f);
```

En la última llamada a `printf` se utiliza `%c` para imprimir el valor `'a'`, `%d` para imprimir el valor entero `28` y `%f` para imprimir el valor en coma flotante `3.0e+8f`.

Los especificadores de formato permite dar un determinado formato a la salida que se imprime con `printf`. Los principales especificadores de formato son:

2.5.1. Secuencias de escape

La secuencias de escape permite imprimir determinados valores especiales:

Secuencia	Significado
\n	nueva línea
\t	tabulador
\b	backspace
\r	retorno de carro
\"	comillas
\'	apóstrofo
\\	backslash
\?	signo de interrogación

Cuadro 2.4: Principales secuencias de escape

2.5.2. Especificadores de ancho de campo

Permite indicar el ancho que se dejará en la salida para imprimir determinada información. Así por ejemplo:

```
printf("Numero entero = %5d \n", 28);
```

produce la salida:

```
Numero entero =     28
```

y

```
printf("Numero real = %5.4f \n", 28.2);
```

produce la salida:

```
Numero entero =     28.2000
```

2.6. Variables

Una variable es un identificador utilizado para representar un cierto tipo de dato. Cada variable es de un tipo de datos determinado. Una variable puede almacenar diferentes valores en distintas partes del programa.

Toda variable debe comenzar con una letra o el carácter _. El resto solo puede contener letras, números o _

Ejemplos de variables válidas son:

```
numero
_color
identificador_1
```

Algo importante a tener en cuenta es que C es **sensible a mayúsculas y minúsculas**, lo que quiere decir que las siguientes variables, por tanto, son todas distintas:

```
pi   PI   Pi   pI
```

2.6.1. Declaración de variables

Una declaración asocia un tipo de datos determinado a una o más variables. El formato de una declaración es:

```
tipo_de_datos  var1, var2, ..., varN;
```

donde `var1` ... `varN` representan identificadores de variables válidos y `tipo_de_datos` el tipo de datos de las variables que se definen.

A continuación se presentan algunos ejemplos:

```
int a, b, c;
float numero_1, numero_2;
char letra;
unsigned long entero;
```

Es importante tener en cuenta que todas las variables deben declararse antes de su uso, y que deben asignarse a las variables nombres significativos. Así en el siguiente ejemplo:

```
int temperatura;
int k;
```

`temperatura` es una variable bastante descriptiva, mientas que `k` no.

2.7. Expresiones y sentencias

Una **expresión** representa una unidad de datos simple, tal como un número o carácter. También puede estar formados por identificadores y operadores. A continuación se presentan dos ejemplos de expresiones válidas en C.

```
a + b
num1 * num2
```

Una **sentencia** controla el flujo de ejecución de un programa. Existen dos tipos de sentencias

- Sentencia simple:

```
temperatura = 4 + 5;
```

- Sentencia compuesta, se encierran entre llaves:

```
{
   temperatura_1 = 4 + 5;
   temperatura_2 = 8 + 9;
}
```

2.8. Sentencia de asignación

El operador de asignación (=) asigna un valor a una variable. Puede asignarse valor inicial a una variable en su declaración. Considere el siguiente fragmento de programa:

```
#include <stdio.h>

int main(void)
{
   int a = 1;
   float b = 4.0;
   int c, d;
   char letra;

   c = 10;
   letra = 'a';
   d = a + c;

   printf("a = %d \n", a);
```

```
    printf("b = %f \n", b);
    printf("c = %d \n", c);
    printf("d = %d \n", d);
    printf("La letra es %c \n", letra);

    return 0;
}
```

El programa anterior declara las variables `a` y `b` y les asigna valores iniciales.

2.9. Función `scanf()`

Esta función permite leer datos del usuario. La función devuelve el número de datos que se han leído correctamente. El formato de dicha función es el siguiente:

```
scanf(formato, argumentos);
```

Al igual que la función `printf()`, en esta función pueden especificarse determinados especificadores de formato. Por ejemplo, `%f` para números reales, `%c` para caracteres y `%d` para números enteros.

```
scanf("%f", &numero);

scanf("%c", &letra);

scanf("%f %d %c", &real, &entero, &letra);

scanf("%ld",  &entero_largo);
```

La primera sentencia lee un número en coma flotante y lo almacena en la variable `numero`. La segunda lee un carácter y lo almacena en la variable `letra` de tipo `char`. La ultima sentencia lee tres datos y los almacena en tres variables. Es importante el uso del operador de dirección `&`, que se describirá más adelante.

Ejemplo El siguiente programa lee un número entero y lo eleva al cuadrado:

```
#include <stdio.h>

int main(void)
{
    int numero;
    int cuadrado;

    printf("Introduzca un numero:");
    scanf("%d", &numero);

    cuadrado = numero * numero;

    printf("El cuadrado de %d es %d\n", numero,  cuadrado);

    return 0;
}
```

2.10. Introducción a la directiva `#define`

Esta directiva ermite definir constantes simbólicas en el programa. Su formato es el siguiente:

```
#define   nombre   texto
```

donde `nombre` representa un nombre simbólico que suele escribirse en mayúsculas, y `texto` no acaba en `;`

El `nombre` es sustituido por `texto` en cualquier lugar del programa. Considere los siguientes ejemplos:

```
#define PI    3.141593

#define CIERTO   1
#define FALSO    0
#define AMIGA    "Marta"
```

Ejemplo El siguiente programa lee el radio de un circulo y calcula su área.

```
#include <stdio.h>
#define PI    3.141593

int main(void)
{
   float radio;
   float area;

   printf("Introduzca el radio: ");
   scanf("%f", &radio);

   area = PI * radio * radio;
   printf("El area del circulo es %5.4f \n", area);

   return 0;
}
```

2.11. Definición de constantes con `const`

En la sección anterior se ha visto el empleo de la directiva `#define` para la definición de valores constantes en un programa. El problema de este enfoque es que la constantes definidas de esta forma no llevan asociado tipo de datos. Una forma más correcta de definir y emplear constantes en un programa es utilizar `const` cuando se define una variable. El siguiente ejemplo es similar al anterior pero definiendo la constante `PI` en este caso como una variable constante de tipo `double`. Una variable definida como constante no puede cambiar su valor durante la ejecución del programa.

```
#include <stdio.h>

const double PI =   3.141593;

int main(void)
{
   float radio;
   float area;

   printf("Introduzca el radio: ");
   scanf("%f", &radio);

   area = PI * radio * radio;
   printf("El area del circulo es %5.4f \n", area);

   return 0;
}
```

2.12. Errores de programación comunes

A continuación se citan algunos de los errores de programación que aparecen más comúnmente cuando se empieza a utilizar el lenguaje de programación C.

- Problemas con las mayúsculas y minúsculas.
- Omisión del punto y coma.
- Comentarios incompletos.
- Comentarios anidados.
- Uso de variables no declaradas.

Ejercicio Indique los errores que aparecen en el siguiente programa.

```
#include <stdio.h>

#define PI    3.141593

      /* Programa que calcula el area de  un circulo */

int main(void)
{
   float radio;

   printf("Introduzca el radio: ")
   scanf("%f", &radio);

   area = PI * radio * Radio;
   printf("El area del circulo es %5.4f \n", area);

   return 0;
}
```

Capítulo 3

Operadores y expresiones

Este capítulo se dedica a presentar los distintos operadores del lenguaje C.

3.1. Operadores aritméticos

Los operadores aritméticos de C se muestran en la siguiente tabla:

Operador	**Función**
+	suma
−	resta
*	producto
/	división
%	operador módulo
	resto de la división entera

La división entera realiza la división de una cantidad entera por otra , se desprecia la parte decimal del cociente. El operador % requiere que los dos operandos sean enteros. La mayoría de las versiones de C asignan al resto el mismo signo del primer operando.

Ejemplo Si $a = 10$ y $b = 3$, entonces

Expresión	**Valor**
$a + b$	13
$a - b$	7
$a * b$	30
a/b	3
$a \ \% \ b$	1

Si $a = 11$ y $b = -3$, entonces

Expresión	**Valor**
$a + b$	8
$a - b$	14
$a * b$	-33
a/b	-3
$a \ \% \ b$	2

3.1.1. Operadores aritméticos. Conversión de tipos

En C un operador se puede aplicar a dos variables o expresiones distintas. Los operandos que difieren en tipo pueden sufrir una conversión de tipo. Como **norma general** el operando de menor precisión toma el tipo del operando de mayor precisión.

Las principales **reglas de conversión de tipos** son las siguientes:

1. Si un operando es `long double` el otro se convierte a `long double`.
2. En otro caso, si es `double` el otro se convierte a `double`.
3. En otro caso, si es `float` el otro se convierte a `float`.
4. En otro caso, si es `unsigned long long int` el otro se se convierte a `unsigned long int`.
5. Si un operando es `long int` y el otro es `unsigned int`, entonces:
   - Si el `unsigned int` puede convertirse a `long int` el operando `unsigned int` se convertirá en `long int`.
   - En otro caso, ambos operandos se convertirán a `unsigned long int`.
6. En otro caso, si un operando es `long int` el otro se convertirá a `long int`.
7. En otro caso, si un operando es `unsigned int` el otro se convertirá a `unsigned int`.
8. En otro caso, ambos operandos serán convertidos a tipo `int` si es necesario.

3.2. Conversión explícita de tipos o `cast`

Se puede convertir una expresión a otro tipo. Para ello se realiza lo siguiente:

```
(tipo datos) expresión
```

En este caso `expresión` se convierte al tipo `tipo datos`.
Así por ejemplo, en la siguiente expresión, `5.5` se convierte a entero (obteniéndose `5`).

```
( (int) 5.5 % 4)
```

3.2.1. Prioridad de los operadores aritméicos

La prioridad indica el orden en el que se realizan las operaciones aritméticas. Las operaciones con mayor precedencia se realiza antes. La prioridad de los operadores aritméticos se presenta en en la siguiente tabla:

Prioridad	Operación
Primero	( )
Segundo	Negación (signo menos)
Tercero	$*, /, \%$
Cuarto	$+, -$

Dentro de cada grupo las operaciones se realizan de izquierda a derecha. La expresión:

```
a - b / c * d
```

es equivalente a

```
a - ((b/c) * d)
```

De igual forma, las dos siguientes expresiones son equivalentes:

```
int a, b;
/* ... */
a = a + 32760 + b + 5;

a = (((a + 32760) + b) + 5);
```

Es importante tener cuidado cuando se realizan operaciones con números en coma flotante.

```
double x, y, z;
/* ... */
x=(x*y)*z; // no es equivalente a  x*=y*z;
z=(x-y)+y; // no es equivalente a  z=x;
z = x + x * y; // no es equivalente a  z = x * (1.0 + y);
y = x / 5.0; // no es equivalente a y = x * 0.2;
```

3.2.2. Operadores de incremento y decremento

En C existen dos tipos de operadores de incremento y decremento:

- Operador de **incremento** $++$ incrementa en uno el operando.
- Operador de **decremento** $--$ decrementa en uno el operando.

Estos presentan a su vez las siguientes variantes:

- Postincremento, $i++$
- Preincremento, $++i$
- Postdecremento, $i--$
- Predecremento, $--i$

Ejemplo

- La expresión $i++;$ es equivalente a $i = i + 1$
- La expresión $++i;$ es equivalente a $i = i + 1$
- La expresión $i--;$ es equivalente a $i = i - 1$
- La expresión $--i;$ es equivalente a $i = i - 1$

Hay que tener en cuenta que:

- Si el operador *precede* al operando el valor del operando se modificará **antes** de su utilización.
- Si el operador *sigue* al operando el valor del operando se modificará **después** de su utilización.

Así por ejemplo, si $a = 1$

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

entonces se imprime:

```
a = 1
a = 2
a = 2
a = 3
```

Estos operadores tienen mayor prioridad que los operadores aritméticos.

Ejemplo Indique el resultado de los siguientes fragmentos de código.

```
j = 2;              j = 2;
k = j++  + 4;       k = ++j  + 4;
```

En general, hay que evitar el uso de este tipo de sentencias.

3.3. Operadores relacionales y lógicos

Los operadores relaciones en C son:

Operador	Función
$<$	menor que
$>$	mayor que
$<=$	menor o igual que
$>=$	mayor o igual que
$==$	igual que
$!=$	distinto que

Estos operadores se utilizan para formar expresiones lógicas. El resultado de una expresión lógica es un valor entero que puede ser:

- *cierto* se representa con un 1.
- *falso* se representa con un 0

Ejemplo Si $a = 1$ y $b = 2$, entonces:

Expresión	Valor	Interpretación
$a < b$	1	cierto
$a > b$	0	falso
$(a + b) != 3$	0	falso
$a == b$	0	falso
$a == 1$	1	cierto

3.3.1. Prioridad de los operadores relacionales

La prioridad de este tipo de operadores se presenta a continuación:

Prioridad	Operación
Primero	( )
Segundo	> < >= <=
Tercero	== !=

Los operadores aritméticos tienen mayor prioridad que los operadores relacionales.
Dentro de cada grupo las operaciones se realizan de izquierda a derecha.

Ejemplo Si $a = 1$ y $b = 2$

Expresión	Valor	Interpretación
$a < b == a > b$	0	falso
$(a < b) == (a > b)$	0	falso
$1 > (a == 2)$	1	cierto
$a == (b == 2)$	1	cierto

3.4. Operadores lógicos

Los operadores lógicos en C son:

Operador	Función
&&	Y *lógica* (AND)
||	O *lógica* (OR)
!	NO *lógico* (NOT)

Estos operadores actúan sobre operandos que son a su vez expresiones lógicas que se interpretan como:

- *cierto*, cualquier valor distinto de 0
- *falso*, el valor 0

Se utilizan para formar expresiones lógicas y su resultado es un valor entero que puede ser:

- *cierto* se representa con un 1
- *falso* se representa con un 0

3.4.1. Tablas de verdad de los operadores lógicos

Las tablas de verdad indican el resultado de los operadores lógicos.

- Tabla de verdad del Y *lógico*:

Expresión	Valor	Interpretación
1 && 1	1	cierto
1 && 0	0	falso
0 && 1	0	falso
0 && 0	0	falso

- Tabla de verdad del O *lógico*:

Expresión	Valor	Interpretación
1 || 1	1	cierto
1 || 0	1	cierto
0 || 1	1	cierto
0 || 0	0	falso

- Tabla de verdad de la negación *lógica*:

Expresión	Valor	Interpretación
!1	0	falso
!0	1	cierto

3.4.2. Prioridad de los operadores lógicos

La prioridad de los distintos operadores lógicos se presenta a continuación:

Prioridad	Operación
Primero	( )
Segundo	!
Tercero	&&
Cuarto	||

&& y || se evalúan de izquierda a derecha, y ! se evalúa de derecha a izquierda.

Ejemplo Si $a = 7$ y $b = 3$

Expresión	Valor	Interpretación
$(a + b) < 10$	0	falso
$!((a + b) < 10)$	1	cierto
$(a\ != 2)\ ||\ ((a + b) <= 10)$	1	cierto
$(a > 4)\ \&\&\ (b < 5)$	1	cierto

Las expresiones lógicas con && o || se evalúan de izquierda a derecha, **solo** hasta que se ha establecido su valor cierto/falso.

Ejemplo Ssi $a = 8$ y $b = 3$, en la sentencia

```
a > 10  &&  b < 4
```

no se evaluará $b < 4$ puesto que $a > 10$ es falso, y por lo tanto el resultado final será falso.
Si $a = 8$ y $b = 3$, en la sentencia

```
a < 10  ||  b < 4
```

no se evaluará $b < 4$ puesto que $a < 10$ es verdadero, y por lo tanto el resultado final será verdadero.

3.5. Resumen de prioridades

A continuación se presenta la relación de las prioridades de los distintos operadores (vistos hasta ahora) de mayor a menor prioridad.

Operador	Evaluación
$++\quad --$ (*post*)	I → D
$++\quad --$ (*pre*)	D → I
$!\quad -$ (signo menos)	D → I
(*tipo*) (*cast*)	D → I
$*\quad /\quad \%$	I → D
$+\quad -$	I → D
$<\quad >\quad <=\quad >=$	I → D
$==\quad !=$	I → D
&&	I → D
||	I → D

Ejemplo Si $a = 7$ y $b = 3$

Expresión	Valor	Interpretación
$a + b < 10$	0	falso
$a\ != 2\ ||\ a + b <= 10$	1	cierto
$a > 4\ \&\&\ b < 5$	1	cierto

3.6. Operadores de asignación

La forma general del operador de asignación es la siguiente:

```
identificador = expresion
```

Por ejemplo:

```
a = 3;
area = lado * lado;
```

El operador de asignación = y el el igualdad == son **distintos**. Este es un error común que se comete cuando se empieza a programar en C.

C también permite la realización de asignaciones múlltiples como la siguiente:

```
id_1 = id_2 = ... = expresion
```

Las asignaciones se realizan de derecha a izquierda. Así, en `i = j = 5`

1. A j se le asigna 5
2. A i se le asigna el valor de j

3.7. Reglas de asignación

La reglas de asignación a tener en cuenta son las siguientes:

- Si los dos operandos en una sentencia de asignación son de tipos distintos, entonces el valor del operando de la derecha será automáticamente convertido al tipo del operando de la izquierda. Además:

  1. Un valor en coma flotante se puede truncar si se asigna a una variable de tipo entero.
  2. Un valor de doble precisión puede redondearse si se asigna a una variable de coma flotante de simple precisión.
  3. Una cantidad entera puede alterarse si se asigna a una variable de tipo entero más corto o a una variable de tipo carácter.

- Es **importante** en C utilizar de forma correcta la conversión de tipos.

3.7.1. Operadores de asignación compuestos

C permite la utilización de los siguientes operadores de asignación compuestos, cuyo significado se muestra a continuación.

`+= -= *= /= %=`

Expresión	Expresión equivalente
`j += 5`	`j = j + 5`
`j -= 5`	`j = j - 5`
`j *= 5`	`j = j * 5`
`j /= 5`	`j = j / 5`
`j %= 5`	`j = j % 5`

Los operadores de asignación tienen menor prioridad que el resto.

3.7.2. Ejemplo

Programa que convierte grados Fahrenheit a grados centígrados. Para ello se utiliza la siguiente expresión:

$$C = (5/9) * (F - 32)$$

```
#include <stdio.h>

int main(void)
{
  float centigrados;
  float fahrenheit;

  printf("Introduzca una temperatura en grados
          fahrenheit: ");
  scanf("%f", &fahrenheit);

  centigrados = 5.0/9 * (fahrenheit - 32);

  printf("%f grados fahrenheit = %f grados
      centigrados \n", fahrenheit, centigrados);

        return 0;
}
```

¿Qué ocurriría si se hubiera utilizado la siguiente sentencia?

```
centigrados = 5/9 * (fahrenheit - 32);
```

3.8. Operador condicional

Su forma general es:

```
expresion_1 ? expresion_2 : expresion_3
```

- Si `expresion_1` es *verdadero* devuelve `expresion_2`
- Si `expresion_1` es *falso* devuelve `expresion_3`
- Su prioridad es justamente superior a los operadores de asignación.
- Se evalúa de derecha a izquierda.

Así por ejemplo, si $a = 1$, en la sentencia:

```
k = (a < 0) ? 0 : 100;
```

- Primero se evalúa `(a < 0)`
- Como es *falso* el operador condicional devuelve `100`
- Este valor se asigna a `k`. Es decir `k` toma el valor `100`

3.9. Operadores de bits

Estos operadores permiten manejar los bits individuales en una palabra de memoria. Hay varias categorías:

- Operador de complemento a uno.
- Operadores lógicos binarios.
- Operadores de desplazamiento.

3.9.1. Operador de complemento a uno

El Operador de complemento a uno (~), invierte los bits de su operando, los unos se transforman en ceros y los ceros en unos.

```
#include <stdio.h>

int main(void)
{
    unsigned int n = 0x4325;

    printf("%x ; %x\n", n, ~n);

   return 0;
}
```

Si la variable `n` ocupa 32 bits, entonces:

- `n = 0x4325`
- `~n = 0xffffbcda`

3.9.2. Operadores lógicos binarios

Los operadores lógicos binarios de C son los siguientes:

Operador	Función
&	AND binario
|	OR binario
^	OR exclusivo binario

Las operaciones se realizan de forma independiente en cada par de bits que corresponden a cada operando.

a	b	a & b	a | b	a ^ b
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

A continuación se muestra un ejemplo de uso

```
#include <stdio.h>
int main(void)
{
    int a = 8;
    int b = 5;
    int c = 3;


    printf("%d\n", a & b);
    printf("%d\n", b & c);
    printf("%d\n", a | c);
    printf("%d\n", b ^ c);

    return 0;

}
```

Este programa imprime: 0, 1, 11, 6

Máscaras

El *enmascaramiento* es un proceso en el que un patrón dado de bits se convierte en otro patrón por medio de una operación lógica a nivel de bits. El segundo operando se llama **máscara**.

El siguiente programa obtiene los cuatros bits menos significativos de un número dado.

```
#include <stdio.h>

#define MASCARA 0xF

int main(void)
{
   int n;

   printf("Introduzca n: ");
   scanf("%d", &n);
   printf("bits menos signif. = %d\n", n & MASCARA );

   return 0;
}
```

3.9.3. Operadores de desplazamiento

Los operadores de desplazamiento son:

- Desplazamiento a la izquierda: <<

- Desplazamiento a la derecha: `>>`

Requieren dos operandos, el primero es un operando de tipo entero que representa el patrón de bits a desplazar. El segundo es un entero sin signo que indica el número de desplazamientos.

Por ejemplo, si $a = 8$

Expresión	**Valor**
$a << 1$	16
$a << 2$	32
$a << 3$	64
$a >> 3$	1

Los operadores de asignación binarios son:

```
&=      ^=      |=      <<=      >>=
```

Capítulo 4

Sentencias de control

Este capítulo se dedica a describir las principales estructuras de control de C.

4.1. Introducción

Los principales tipos de estructuras de programación (ver figura 4.1) son:

- **Secuencia**: ejecución sucesiva de dos o más operaciones.
- **Selección**: se realiza una u otra operación, dependiendo de una condición.
- **Iteración**: repetición de una operación mientras se cumpla una condición.

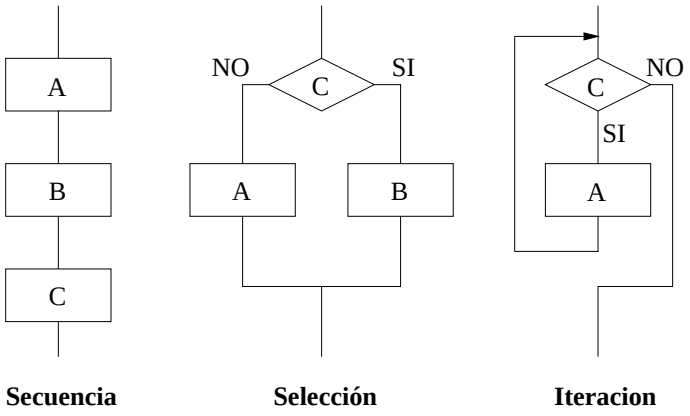

Figura 4.1: Estructuras de programación

4.2. Sentencia `if`

Su forma general es:

```
if (expresion)
   sentencia
```

Si `expresion` es verdadera (valor distinto de 0) se ejecuta `sentencia`. La `expresion` debe estar entre paréntesis.

En el caso de que `sentencia` sea compuesta, la forma general debería ser la siguiente:

```
if (expresion)  {
   sentencia 1
   sentencia 2
        .
        .
        .
   sentencia N
}
```

El diagrama de flujo de esta sentencia es el que se muestra en la figura 4.2.

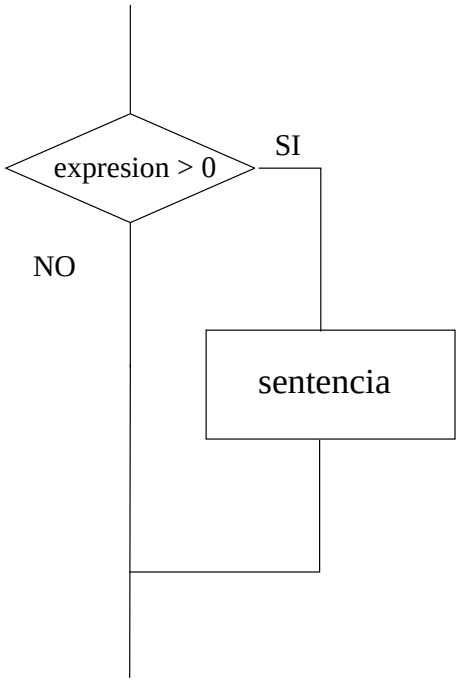

Figura 4.2: Diagrama de flujo de la sentencia `if`

Ejemplo El siguiente programa lee un número e indica si es par.

```
#include <stdio.h>

int main(void)
{
   int numero;

   /* leer el numero */
   printf("Introduzca un numero: ");
   scanf("%d", &numero);

   if ((numero % 2) == 0)
       printf("El numero %d es par.\n", numero);

   return 0;
}
```

Ejemplo El siguiente programa lee un número y lo eleva al cuadrado si es par.

```c
#include <stdio.h>

int main(void)
{
   int numero;
   int cuadrado;

   /* leer el numero */
   printf("Introduzca un numero: ");
   scanf("%d", &numero);

   if ((numero % 2) == 0) {
       cuadrado = numero * numero;
       printf("El cuadrado de %d es %d.\n", numero,
                  cuadrado);
   }

   return 0;
}
```

4.3. Sentencia `if-else`

Su forma general es:

```
if (expresion)
   sentencia 1
else
   sentencia 2
```

Si `expresion` es:

- verdadera (valor distinto de 0) se ejecuta `sentencia 1`.
- falsa (valor igual a 0) se ejecuta `sentencia 2`.

El diagrama de flujo de esta sentencia es el que se muestra en la figura 4.3. Si las sentencias son compuestas se encierran entre `{ }`.

Las sentencias pueden ser a su vez sentencias `if-else`. Por ejemplo:

```
if (e1)
   if (e2)
      S1
   else
      S2
else
   S3
```

Aunque las sentencias incluidas en las ramas del `if` y del `else` sean simples, es conveniente para evitar errores hacer uso siempre de las llaves:

```
if (e1){
   if (e2){
      S1
   }
   else{
      S2
  }
}
```

```
else{
   S3
}
```

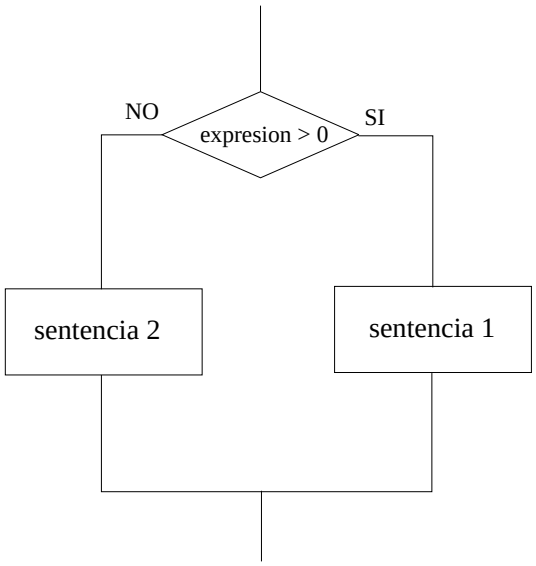

Figura 4.3: Diagrama de flujo de la sentencia `if-else`

Ejemplo El siguiente programa lee un número y dice si es par o impar.

```
#include <stdio.h>

int main(void)
{
    int numero;

   /* Leer el numero */
   printf("Introduzca un numero: ");
   scanf("%d", &numero);

   if ((numero % 2) == 0)
      printf("El numero %d es par.\n", numero);
   else
      printf("El numero %d es impar.\n", numero);

   return 0;

}
```

4.4. Sentencia `for`

Su forma general es:

```
for (expresion 1; expresion 2; expresion 3)
   sentencia
```

Inicialmente se ejecuta `expresion 1`. Esta expresión se inicializa con algún parámetro que controla la repetición del bucle. La expresión `expresion 2` es una condición que debe ser cierta para que se ejecute `sentencia`. La expresión `expresion 3` se utiliza para modificar el valor del parámetro.

El bucle se repite mientras `expresion 2` no sea cero (falso). Si `sentencia` es compuesta se encierra entre `{ }`. `expresion 1` y `expresion 3` se pueden omitir. Si se omite `expresion 2` se asumirá el valor permanente de 1 (cierto) y el bucle se ejecutará de forma indefinida.

El diagrama de flujo de esta sentencia es el que se muestra en la figura 4.4.

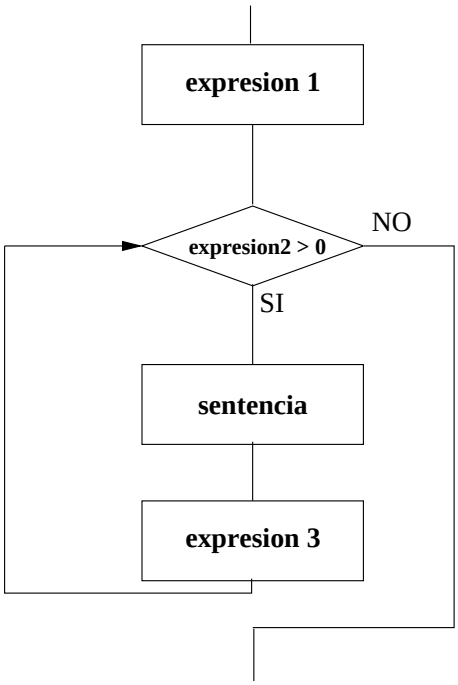

Figura 4.4: Diagrama de flujo de la sentencia for

Ejemplo El siguiente programa imprime los 100 primeros números.

```
#include <stdio.h>

int main(void)
{
    int numero;

    for (numero=0; numero <100; numero++)
       printf("%d\n", numero);

   return 0;
}
```

La figura 4.5 muestra el diagrama de flujo del programa anterior.

4.5. Sentencia `while`

Su forma general es la siguiente:

```
while (expresion)
   sentencia
```

La sentencia `sentencia` se ejecutará mientras el valor de `expresion` sea verdadero (distinto de 0). En la sentencia anterior primero se evalúa `expresion`. Lo normal es que `sentencia` incluya algún elemento que altere el valor de `expresion`, proporcionando así la condición de salida del bucle.

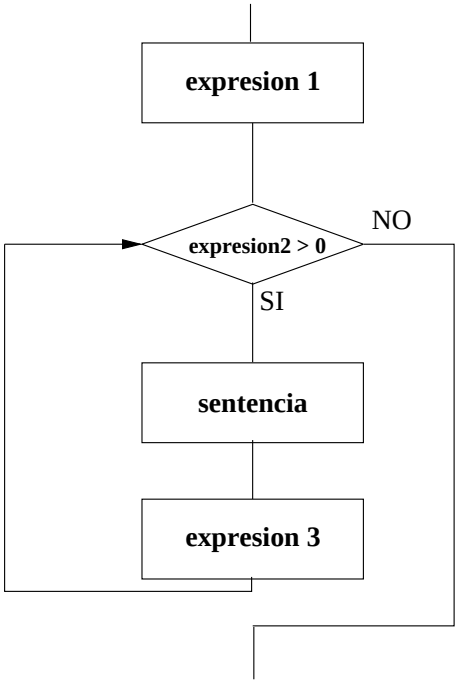

Figura 4.5: Diagrama de flujo para el programa que imprime los 100 primeros números naturales

Si la sentencia es compuesta se encierra entre `{ }`, de la siguiente forma:

```
while (expresion) {
   sentencia 1
   sentencia 2
        .
        .
   sentencia N
}
```

El diagrama de flujo de la sentencia `while` es el se muestra en la figura 4.6.

Ejemplo El siguiente programa lee un número N y calcula $1 + 2 + 3 + \ldots + N$

```
#include <stdio.h>

int main(void)
{
   int N;
   int suma = 0;

   /* leer el numero N */
   printf("N: ");
   scanf("%d", &N);

   while (N > 0)  {
      suma = suma + N;
      N = N - 1;       /* equivalente a N-- */
   }

   printf("1 + 2 +...+ N = %d\n", suma);

   return 0;
```

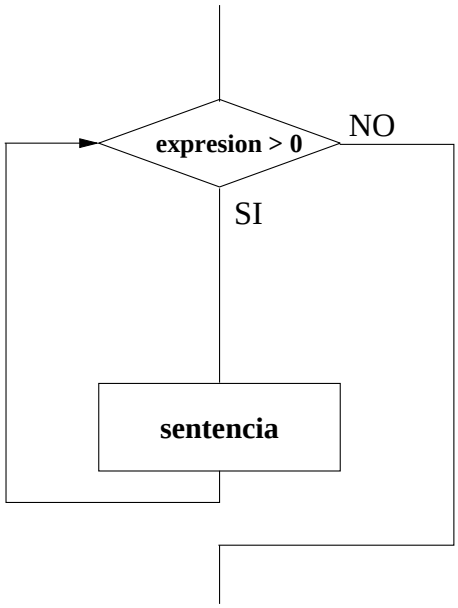

Figura 4.6: Diagrama de flujo de la sentencia `while`

```
}
```

4.6. Sentencia `do-while`

Su forma general es la siguiente:

```
do
   sentencia
while (expresion);
```

La sentencia `sentencia` se ejecutará mientras el valor de `expresion` sea verdadero (distinto de 0). En este tipo de bucles `sentencia` siempre se ejecuta al menos una vez (diferente a `while`). Lo normal es que `sentencia` incluya algún elemento que altere el valor de `expresion`, proporcionando así la condición de salida del bucle.

Si la sentencia es compuesta se encierra entre `{ }`, de la siguiente manera:

```
do {
   sentencia 1
   sentencia 2
        .
        .
   sentencia N
}while (expresion);
```

Para la mayoría de las aplicaciones es mejor y más natural comprobar la condición antes de ejecutar el bucle (bucle `while`).

El diagrama de flujo de esta sentencia es el que se muestra en la figura 4.7.

Ejemplo El siguiente programa lee de forma repetida un número e indica si es par o impar. El programa se repite mientras el numero sea distinto de cero.

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

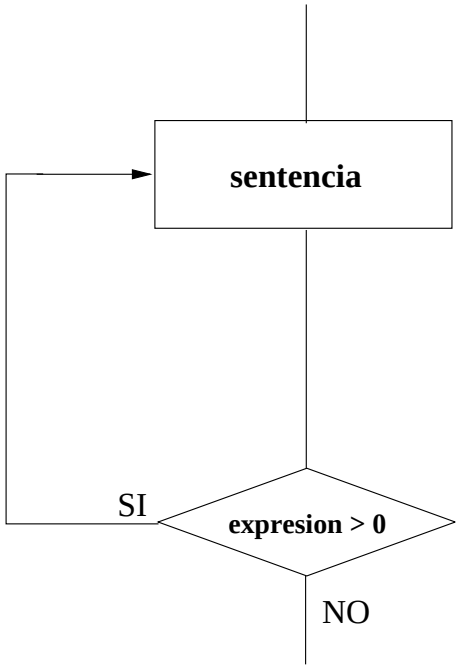

Figura 4.7: Diagrama de flujo de la sentencia `do`

```
int main(void)
{
   int numero;

   do {
      /* se lee el numero */
      printf("Introduzca un numero: ");
      scanf("%d", &numero);

      if ((numero % 2) == 0)
         printf("El numero %d es par.\n", numero);
      else
         printf("El numero %d es par.\n", numero);

   } while (numero != 0)

   return 0;
}
```

4.7. Sentencia `switch`

Su forma general es la siguiente:

```
switch (expresion) {
   case exp 1:
      sentencia 1;
      sentencia 2;
         .
         .
      break;
```

```
        case exp N:
        case exp M:
           sentencia N1;
           sentencia N2;
              .
              .
           break;

        default:
           sentencia D;
              .
    }
```

La expresión `expresion` devuelve un valor entero (también puede ser de tipo `char`).
Las expresiones `exp 1`, ..., `exp` representan valores constantes de tipo entero (también caracteres).
El diagrama de flujo de este tipo de sentencias se muestra en la figura 4.8.

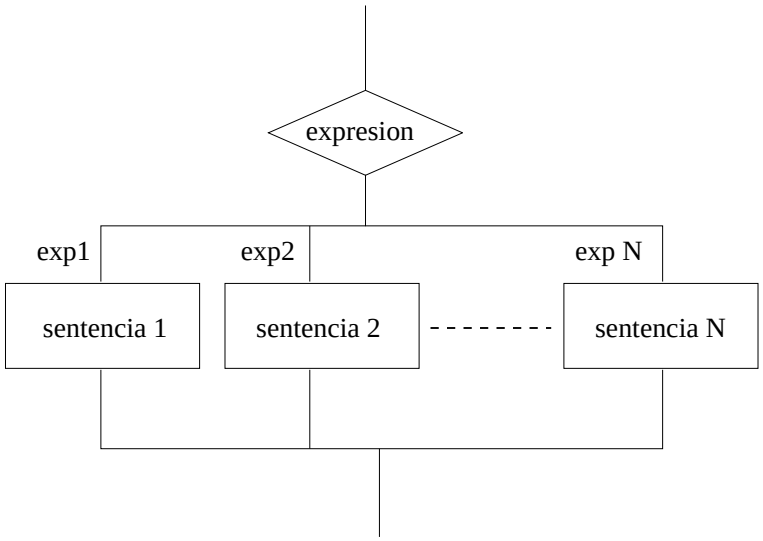

Figura 4.8: Diagrama de flujo de la sentencia `switch`

```
#include <stdio.h>

int main(void)
{
   char letra;

   printf("Introduzca una letra: ");
   scanf("%d", &letra);

   switch(letra) {
      case 'A':
      case 'a':
         printf("Vocal %c\n", letra);
         break;
      case 'E':
      case 'e':
         printf("Vocal %c\n", letra);
         break;
      case 'I':
```

```
        case 'i':
            printf("Vocal %c\n", letra);
            break;
        case 'O':
        case 'o':
            printf("Vocal %c\n", letra);
            break;
        case 'U':
        case 'u':
            printf("Vocal %c\n", letra);
            break;
        default:
            printf("Consonante %c\n", letra);
    }

    return 0;
}
```

4.8. Bucles anidados

Los bucles se pueden *anidar* unos en otros. Se pueden anidar diferentes tipos de bucles. Es importante estructurarlos de forma correcta.

Ejemplo El siguiente programa calcula $1 + 2 + \ldots N$ mientras N sea distinto de 0.

```
#include <stdio.h>
int main(void)
{
    int N;
    int suma;
    int j;

    do {
        /* leer el numero N */
        printf("Introduzca N: ");
        scanf("%d", &N);

        suma = 0;
        for (j = 0; j <= N; j++) /* bucle anidado */
            suma = suma + j;
        printf("1 + 2 + ... + N = %d\n", suma);
    } while (N > 0);      /* fin del bucle do */

    return 0;
}
```

4.9. Sentencia `break`

Esta sentencia se utiliza para terminar la ejecución de bucles o salir de una sentencia `switch`. Es necesaria en la sentencia `switch` para transferir el control fuera de la misma.

En caso de bucles anidados, el control se transfiere fuera de la sentencia más interna en la que se encuentre, pero no fuera de las externas.

Esta sentencia **no** es aconsejable el uso de esta sentencia en bucles contrario a la programación estructurada. Sin embargo, puede ser útil cuando se detectan errores o condiciones anormales.

Ejemplo El siguiente programa calcula la media de un conjunto de datos.

- Primera versión: utiliza la sentencia `break`.
- Segunda versión: estructurada no utiliza la sentencia `break`.

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

int main(void)
{
   float valor;
   float suma = 0.0;
   float media;
   int cont;
   int total;
   int error = false;
   int leidos;

   printf("Cuantos datos: ");
   scanf("%d", &total);

   for(cont = 0; cont < total; cont++) {
      printf("Introduzca valor: ");
      leidos = scanf("%f", &valor);
      if (leidos == 0) {
         error = true;
         break;
      }

      suma = suma + valor;
   }

   if (error)
      printf("Error en la lectura de los datos\n");
   else  {
      if (total > 0) {
         media = suma / total;
         printf("Media = %f\n", media);
      }
      else
         printf("El numero de datos es 0\n");
   }

   return 0;
}
```

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


int main(void)
```

```
{
   float valor;
   float suma = 0.0;
   float media;
   int cont = 0;
   int total;
   int error = false;
   int leidos;

   printf("Cuantos datos: ");
   scanf("%d", &total);

   while ((cont < total) && (!error)) {
      printf("Introduzca valor: ");
      leidos = scanf("%f", &valor);
      if (leidos == 0)
         error = true;
      else {
         suma = suma + valor;
         cont = cont + 1;
      }
   }
   if (error)
      printf("Error en la lectura de los datos\n");
   else {
      if (total > 0){
         media = suma / total;
         printf("Media = %f\n", media);
      }
      else
         printf("El numero de datos es 0\n");
   }

   return 0;
}
```

4.10. Sentencia `continue`

Cuando se ejecuta esta sentencia dentro de un bucle se salta inmediatamente a evaluar la condición asociada al bucle. El siguiente programa solicita de forma repetida la introducción de un número. El programa finaliza cuando se introduce un 0. Este programa es similar al mostrado anteriormente, pero en este caso la suma solo se realiza en caso de que el número introducido sea par. Si es impar se ejecuta la sentencia `continue` y el flujo se transfiere inmediatamente a evaluar la expresión `N > 0)`.

```
#include <stdio.h>
int main(void)
{
   int N;
   int suma;
   int j;

   do {
      /* leer el numero N */
      printf("Introduzca N: ");
      scanf("%d", &N);
```

```
        if (N%2 != 0)
            continue;     // si es impar salta a evaluar la condición de salida

        suma = 0;
        for (j = 0; j <= N; j++) /* bucle anidado */
           suma = suma + j;
        printf("1 + 2 + ... + N = %d\n", suma);
    } while (N > 0);      /* fin del bucle do */

    return 0;
}
```

Capítulo 5

Funciones

En este capítulo se describe el uso de las funciones en los programas escritos en C, y se presentan las bases de la programación estructurada en C.

5.1. Introducción

Una función es un segmento de programa que realiza una determinada tarea. Todo programa C consta de una o más funciones. Una de estas funciones se debe llamar `main()`. Todo programa comienza su ejecución en la función `main()`.

El uso de funciones permite la descomposición y desarrollo modular. Permite dividir un programa en componentes más pequeños: funciones.

Ejemplo El siguiente programa calcula el máximo de dos números utilizando la función `maximo`.

```
#include <stdio.h>

int maximo(int a, int b);    /* Declaración de función (prototipo) */

int maximo(int a, int b) /* definicion de la funcion */
{
     int max;

     if (a > b)
          max = a;
     else
          max = b;
     return(max); /* devuelve el valor maximo */
}

int main(void)
{
     int x, y;
     int max;

     printf("Introduzca dos numeros: ");
     scanf("%d %d", &x, &y);

     max = maximo(x,y);     /* llamada a la funcion */
     printf("El maximo es %d\n", max);

     return 0;
}
```

Ejemplo El siguiente programa determina si un número es un cuadrado un perfecto.

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

void explicacion(void);

_Bool cuadrado_perfecto(int x);

void explicacion(void)
{
   printf("Este programa dice si un numero ");
   printf("es cuadrado perfecto \n");
   printf("Introduzca un numero: " );

   return;
}

_Bool cuadrado_perfecto(int x)
{
   int raiz;
   int perfecto;

   raiz = (int) sqrt(x);
   if (x == raiz * raiz)
      perfecto = true;    /* cuadrado perfecto */
   else
      perfecto = false; /* no es cuadrado perfecto */

   return(perfecto);
}

int main(void)
{
   int n;
   _Bool perfecto;

   explicacion();

   scanf("%d", &n);

   perfecto = cuadrado_perfecto(n);

   if (perfecto)
      printf("%d es cuadrado perfecto.\n", n);
   else
      printf("%d no es cuadrado perfecto.\n", n);

   return 0;
}
```

Este programa hace uso de la función de biblioteca `sqrt` que permite calcular la raíz cuadrada. Esta función está declarada en el archivo de cabecera `<math.h>`. Es necesario incluir este archivo para que el compilador conozca el tipo de argumentos y resultado que devuelve esta función. El código de la función se encuentra definido en la biblioteca matemática (`libm`). Para poder generar correctamente el ejecutable, es necesario compilar el programa de esta forma:

```
gcc -Wall -o perfecto perfecto.c -lm
```

Observer que se utiliza `-lm`. De esta forma se indica al compilador que busque el código de la función `sqrt` en el archivo de biblioteca `libm`. Esto no es necesario para otras funciones como las que se han utilizado hasta ahora, por ejemplo `printf`. El código de estas funciones se define en la biblioteca estándar de C (`libc`), y esta se incluye siempre por defecto cuando se compila un programa.

5.2. Definición de una función

El formato general de definición de una función en C es el siguiente:

```
tipo nombre(tipo1 arg1, ..., tipoN argN)
{
    /* CUERPO DE LA FUNCION */
}
```

Los argumentos se denominan **parámetros formales**. Como puede apreciarse la función devuelve un valor de tipo `tipo`. Si se omite `tipo` se considera que devuelve un `int`. Cuando la función no devuelve ningún tipo se indica con `void`. De igual manera, si la función no tiene argumentos, se colocará `void` entre los paréntesis, tal y como se muestra en siguiente ejemplo.

```
void explicacion(void)
```

Entre llaves se encuentra el *cuerpo* de la función (igual que `main()`), que incluye la sentencias de la función. La sentencia `return` finaliza la ejecución y devuelve un valor a la función que realiza la llamada.

```
return(expresion);
```

En C hay que tener en cuenta dos cosas relacionadas con las funciones:

- Una función solo puede devolver un valor.
- En C no se pueden anidar funciones.

5.3. Declaración de funciones: prototipos

La forma general de declarar un función es la siguiente:

```
tipo nombre(tipo1 arg1, ..., tipoN argN);
```

La declaración o prototipo de una función especifica el nombre de la función, el valor que devuelve y los tipos de parámetros que acepta. Esta declaración finaliza con `;` y no incluye el cuerpo con el código de la función.

No es obligatorio declarar una función pero si aconsejable, ya que permite la comprobación de errores entre las llamadas a una función y la definición de la función correspondiente. Si el compilador no conoce el tipo de una función cuando se la invoca asumirá que la función devuelve un `int` y que los parámetros que se le pasan a la función son también de tipo `int`, lo cual puede dar lugar a errores en tiempo de ejecución.

Ejemplo la siguiente función calcula x elevado a y (con y entero).

```
float potencia (float x, int y); /* prototipo */

float potencia (float x, int y)  /* definición */
{
   int i;
   float prod = 1;
```

```
    for (i = 0; i < y; i++)
        prod = prod * x;

    return(prod);
}
```

5.4. Llamadas a funciones

Para llamar a una función se especifica su nombre y la lista de argumentos. Por ejemplo:

```
maximo(2, 3);
```

Se denominan **parámetros formales** a los que aparecen en la definición de la función, y **parámetros reales** a los que se pasan en la llamada a la función. En una llamada a una función habrá un argumento real por cada argumento formal.

Los parámetros reales pueden ser:

- Constantes.
- Variables simples.
- Expresiones complejas, que pueden incluir llamadas a otras funciones.

Todos estos parámetros deben ser del mismo tipo de datos que el argumento formal correspondiente. Cuando se pasa un valor a una función se copia el argumento real en el argumento formal. Se puede modificar el argumento formal dentro de la función, pero el valor del argumento real no cambia: **paso de argumentos por valor**. En C todos los parámetros se pasan por **valor**.

El proceso de llamada a una función se puede apreciar en la figura 5.1.

Ejemplo El siguiente programa lee el número de caracteres introducidos hasta fin de fichero.

```
#include <stdio.h>

int cuenta_caracteres(void);

int cuenta_caracteres(void)
{
    char c;
    int cont = 0;

    c = getchar();
    while (c != EOF) {
        cont = cont + 1;
        c = getchar();
    }
    return(cont);
}

int main(void)
{
    int num_car;

    num_car = cuenta_caracteres();
    printf("Hay %d caracteres\n", num_car);

    return 0;
}
```

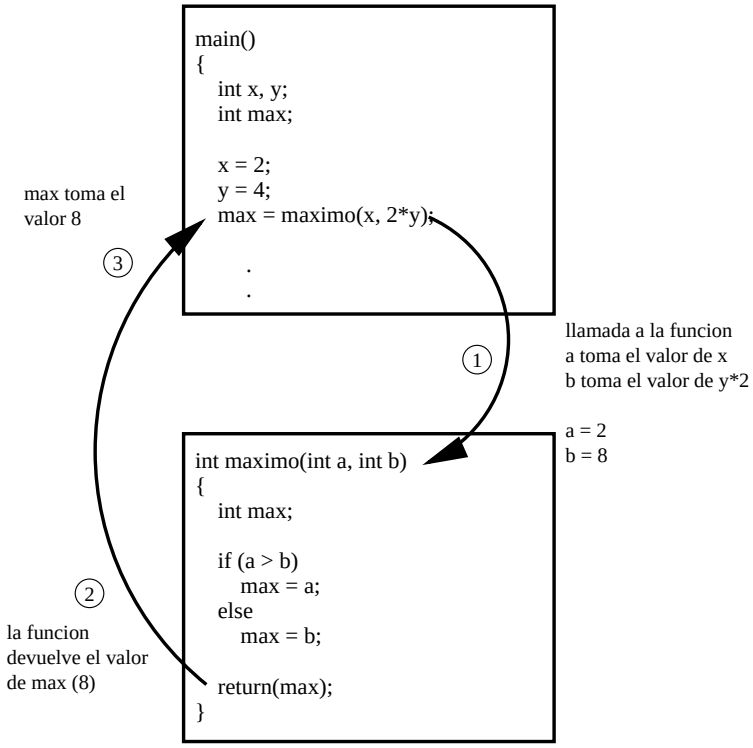

Figura 5.1: Proceso de llamada a una función

5.5. Recursividad

Una función **recursiva** es aquella que se llama así misma de forma repetida hasta que se cumpla alguna condición.

Un ejemplo de función recursiva es el factorial de un número, cuya definición se muestra a continuación.

$$f(n) = \begin{cases} 1 & \text{si } n = 0 \\ n * f(n-1) & \text{si } n > 0 \end{cases}$$

El siguiente fragmento de c´cdigo incluye una función en C que calcula el factorial de forma recursiva.

```
long int factorial(int n)
{
   if (n <= 1)
      return(1);
   else
      return(n * factorial(n-1));
}
```

5.6. Macros

La sentencia `#define` se puede utilizar para definir macros. Una **macro** es un identificador equivalente a una expresión, sentencia o grupo de sentencias.

Ejemplo el siguiente programa calcula el máximo de dos números utilizando una macro denominada `maximo`.

```
#include <stdio.h>

#define maximo(a,b)      ((a > b) ? a : b)     // macro

int main(void)
{
     int x, y;
     int max;

     printf("Introduzca dos numeros: ");
     scanf("%d %d", &x, &y);

     max = maximo(x,y);     /* uso de la macro */
     printf("El maximo es %d\n", max);

    return 0;
}
```

El preprocesador sustituye todas las referencias a la macro que aparezcan dentro de un programa antes de realizar la compilación, por lo que es importante que no haya blancos entre el identificador y el paréntesis izquierdo. Así, el código anterior se sustituye por el siguiente antes de realizar la compilación:

```
int main(void)
{
     int x, y;
     int max;

     printf("Introduzca dos numeros: ");
     scanf("%d %d", &x, &y);

     max = ((x > y) ? x : y);
     printf("El maximo es %d\n", max);

     return 0;
}
```

Utilizando macros no se produce llamada a función por lo tanto el programa ejecuta con mayor velocidad. Sin embargo, se repite el código en cada uso de la macro con lo que se obtiene un código objeto más grande. Hay que tener siempre en mente que una macro no es una llamada a función. El código se expande en el lugar en el que se utiliza.

Ejemplo Dada la siguiente definición de macro:

```
#define maximo (x,y,z)  if (x > y)      \
                           z = x;   \
                        else            \
                           z = y;
```

Cuando el preprocesador encuentra:

```
maximo(a, b, max);
```

Lo sustituirá por:

```
if ( a  >   b ) max =  a ; else  max =  b ; ;
```

Esta sentencia es equivalente a:

```
    if (a > b)
       max = a;
    else
       max = b;
```

5.7. Funciones `inline`

Una función **`inline`** es como cualquier otra función sólo que al realizar la llamado a la misma, si su cuerpo lo permite, se reemplaza la llamada por el cuerpo de la función al igual que ocurre con las macros. En caso de que el cuerpo de la función **`inline`** no lo permita, el compilador la tomará como cualquier otra función haciendo una llamada a la misma. Es preferible hacer uso de este tipo de funciones puesto que admiten la declaración de variables locales y asegura un código más limpio. El siguiente programa es similar al mostrado anteriormente utilizando en este caso una función **`inline`**.

```
#include <stdio.h>

inline int maximo(int a, int b)
{
        if (a > b )
                return a;
        else
                return b;
}

int main(void)
{
    int x, y;
    int max;

    printf("Introduzca dos numeros: ");
    scanf("%d %d", &x, &y);

    max = maximo(x,y);     /* llamada a la función */
    printf("El maximo es %d\n", max);

   return 0;
}
```

5.8. Programación estructurada

Las técnicas de programación estructurada tiende a construir programas fácilmente comprensibles. Se basa en la técnica de diseño mediante refinamiento progresivo, en el que las operaciones se van descomponiendo poco a poco hasta llegar a operaciones básicas.

La figura 5.2 presenta las construcciones básicas de la programación estructurada:

Cuando se utilizan técnicas de programación estructurada:

- Todos los bloques y funciones tienen un único punto de entrada.
- Todos los bloques y funciones tienen un único punto de salida.

Ejemplo El siguiente programa calcula la hipotenusa de un triángulo rectángulo. utilizando la siguiente expresión:

$$h = \sqrt{(a^2 + b^2)}$$

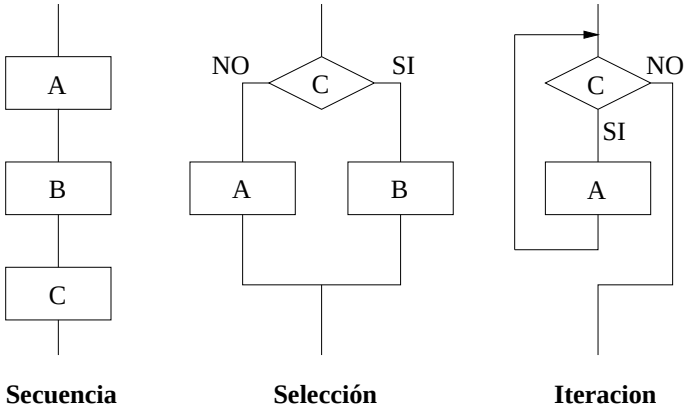

Figura 5.2: Construcciones básicas de la programación estructurada

los pasos a seguir para desarrollar el programa son:

1. Leer a y b función `scanf()`
2. Calcular h según la fórmula dada definimos una función `hipotenusa()`.
3. Imprimir el valor de h función `printf()`.

A continuación se presenta un programa que resuelve el problema anterior.

```
#include <stdio.h>
#include <math.h>
float hipotenusa(float a, float b);  /* prototipo */

int main(void)
{
    float a, b;
    float h;
    int error;

    printf("Introduzca a y b: ");
    error = scanf("%f %f", &a, &b);
    if (error !=2)
       printf("Error al leer a y b\n");
    else
    {
        h = hipotenusa(a,b);
        printf("La hipotenusa es %f\n", h);
    }

    return 0;
}
float hipotenusa(float a, float b)
{
   float h;

   h = sqrt(pow(a,2) + pow(b, 2));
   return(h);
}
```

Capítulo 6

Punteros y ámbito de las variables

6.1. Organización de la memoria

Por norma general la memoria de un computador se direcciona por bytes, es decir, en cada posición de memoria se almacena un byte. La memoria del computador se encuentra organizada en grupos de **bytes** que se denominan **palabras** (ver figura 6.1). Dentro de la memoria cada dato ocupa un número determinado de bytes:

- Un `char` 1 byte.
- Un `int` normalmente 4 bytes.
- Un `double` 4 bytes.

A cada byte o palabra se accede por su dirección. Si `x` es una variable que representa un determinado dato el compilador reservará los bytes necesarios para representar `x` (4 bytes si es de tipo `int`).

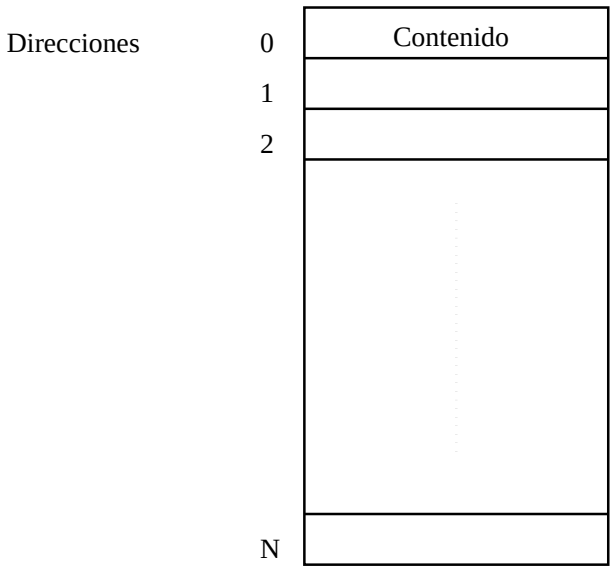

Figura 6.1: Organización de la memoria del computador

6.2. Punteros

Si `x` es una variable, `&x` representa la dirección de memoria de `x`, es decir, la dirección de memoria donde se almacena el valor de la variable `x`. Al operador `&` se le denomina **operador de dirección**.

Un **puntero** es una variable que almacena la dirección de otro objeto (variable, función, ...).

Ejemplo El siguiente ejemplo presenta el uso de punteros.

```
#include <stdio.h>

int main(void)
{
   int x;     // variable de tipo entero
   int y;     // variable de tipo entero
   int *px;   // variable de tipo  puntero a entero

   x = 5;
   px = &x; // asigna a px la dirección de x
   y = *px; // asigna a y el contenido de la dirección almacenada en px

   printf("x = %d\n", x);
   printf("y = %d\n", y);
   printf("*px = %d\n", *px);

   return 0;
}
```

La expresión `*px` representa el contenido almacenado en la dirección a la que apunta `px`. El operador `*` se denomina **operador de indirección** y opera sobre una variable de tipo puntero (ver figura 6.2).

Es importante recordar que un puntero representa la dirección de memoria del objeto al que apunta, **NO** su valor.

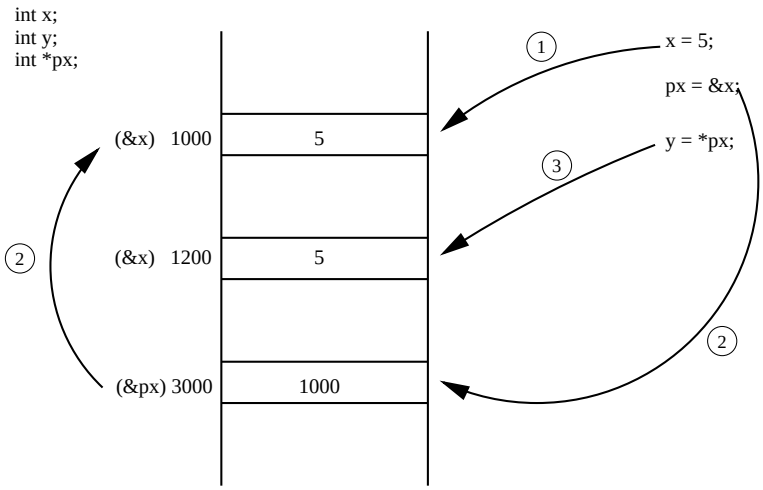

Figura 6.2: Uso de punteros

6.3. Declaración de punteros

Los punteros al igual que el resto de variables deben declararse antes de usarlos. La declaración de una variable de tipo puntero se realiza de la siguiente forma:

```
tipo_dato  *variable_ptr;
```

El identificador `variable_ptr` es el nombre de la variable puntero, y `tipo_dato` se refiere al tipo de dato apuntado por el puntero.

Ejemplos

- `int *numero;`
- `float *p;`
- `char *letra;`

La variable `variable_ptr` solo puede almacenar la dirección de variables de tipo `tipo_dato`. Para usar un puntero se debe estar seguro de que apunta a una dirección de memoria correcta.

Es importante recordar que un puntero no reserva memoria. El siguiente fragmento de programa es incorrecto:

```
int *p;

*p = 5;
```

¿A qué dirección de memoria apunta `p`? ¿Dónde se almacena el 5?

El siguiente fragmento sí es correcto:

```
char *p;
char letra;

letra = 'a';
p = &letra;
```

En este caso, `p` almacena la dirección de `letra`, es decir, apunta a una dirección conocida. La variable `*p` representa el valor almacenado en `letra` (`'a'`).

Ejemplo El siguiente ejemplo muestra cómo pueden asignarse punteros del mismo tipo entre sí.

```
float n1;
float n2;
float *p1;
float *p2;

n1 = 4.0;
p1 = &n1;

p2 = p1;

n2 = *p2;

n1 = *p1 + *p2;
```

Dado el fragmento de código anterior, ¿cuánto vale `n1` y `n2`?

6.4. Paso de punteros a una función

Cuando se pasa un puntero a una función, lo que se está pasando (por valor) a la función es una dirección de memoria, **no** se pasa una copia del dato al que apunta, sino la dirección del dato al que apunta (se pasa por valor una dirección de memoria). El uso de punteros permite simular en C el

paso de argumentos por **referencia**. Recuerde, que en el capítulo anterior se dijo que en C todos los argumentos se pasan por valor.

Cuando un argumento se pasa por valor, el dato (el valor) se *copia* a la función. Esto quiere decir que un argumento pasado por valor no se puede modificar dentro de la función.

Cuando se pasa un argumento por *referencia* (cuando un puntero se pasa a una función), se pasa la *dirección* del dato, lo que implica que el contenido de la dirección se puede modificar en la función. Por lo tanto los argumentos pasados por referencia, si se pueden modificar.

El uso de punteros como argumentos de funciones permite que el dato sea alterado globalmente dentro de la función, y por tanto, el comportamiento es similar a si se estuviera pasando el argumento por referencia.

Ejemplo El siguiente ejemplo ilustra el paso de parámetros por valor.

```
#include <stdio.h>

void funcion(int a, int b); /* prototipo */

int main(void)
{
   int x = 2;
   int y = 5;

   printf("Antes x = %d, y = %d\n", x, y);

   funcion(x, y);

   printf("Despues x = %d, y = %d\n", x, y);

   return 0;
}

void funcion(int a, int b)
{
   a = 0;
   b = 0;

   // los cambios en estos parámetros no afectan a los valores que
   // se pasaron a la llamada

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

   return;
}
```

La figura 6.3 ilustra el paso de parámetros por valor.

Ejemplo El siguiente ejemplo muestra cómo se puede simular el paso de parámetros por referencia en C.

```
#include <stdio.h>

void funcion(int *a, int *b); /* prototipo */

int main(void)
{
   int x = 2;
   int y = 5;
```

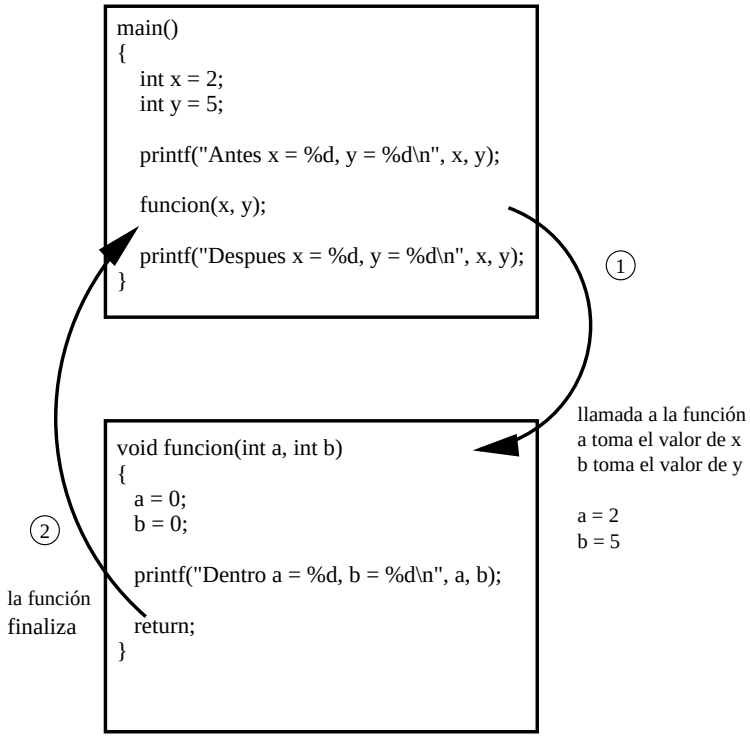

Figura 6.3: Paso de parámetros por valor

```
    printf("Antes x = %d, y = %d\n", x, y);

    funcion(&x, &y);

    printf("Despues x = %d, y = %d\n", x, y);

    return 0;
}

void funcion(int *a, int *b)
{
    *a = 0;
    *b = 0;

    printf("Dentro *a = %d, *b = %d\n", *a, *b);

    return;
}
```

En la figura 6.4 se ilustra el paso de parámetros por referencia.

Ejemplo La siguiente función intercambia el valor de dos variables.

```
#include <stdio.h>

void swap(int *a, int *b); /* prototipo */

int ain(void)
{
    int x = 2;
```

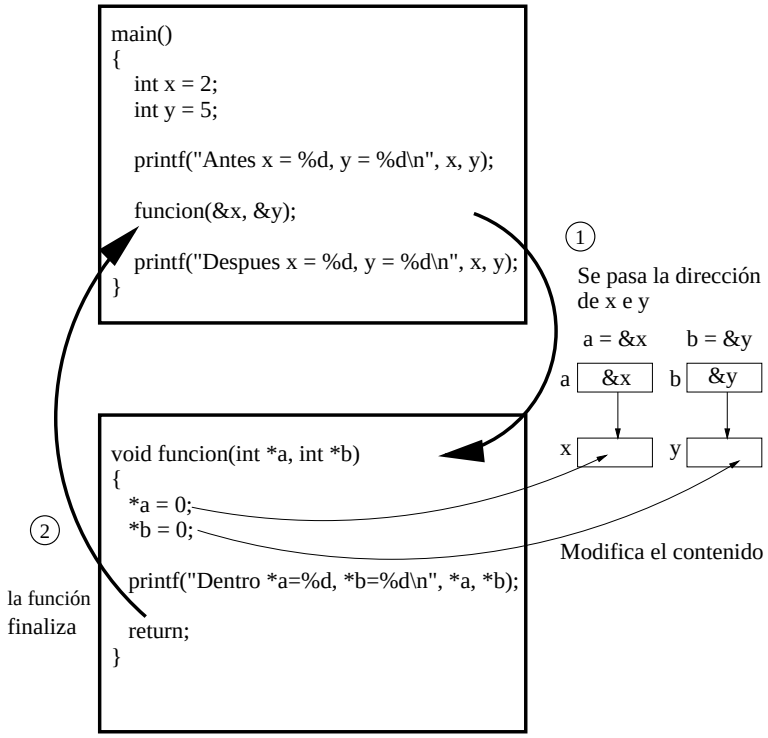

Figura 6.4: Paso de parámetros por referencia

```
    int y = 5;

    printf("Antes x = %d, y = %d\n", x, y);
    swap(&x, &y);
    printf("Despues x = %d, y = %d\n", x, y);

    return 0;
}

void swap(int *a, int *b)
{
    int temp;

    temp = *b;
    *b = *a;
    *a = temp;

    return;
}
```

Como norma general si se quiere modificar el contenido de una variable dentro de una función, es necesario pasar la dirección de esa variable a la función. Si se ha definido una variable:

```
int *p;
```

y se quiere que una función pueda modificar el valor de esta variable, es decir, variables de tipo puntero a entero, el prototipo de la función tendrá que ser el siguiente:

```
void F(int **x);
```

Es decir, habrá que invocar a la función de esta manera:

```
void F(&p);
```

A la función se le pasa la dirección de `p`. Como `p` es de tipo `int *`, el parámetro de la función debe ser `int **`.

6.4.1. Puntero NULL

Cuando se asigna **0** a un puntero, este no apunta a ningún objeto o función. La constante simbólica **NULL** definida en `stdio.h` tiene el valor **0** y representa el puntero nulo. Es una buena técnica de programación asegurarse de que todos los punteros toman el valor **NULL** cuando no apuntan a ningún objeto o función.

```
int *p = NULL;
```

Para ver si un puntero no apunta a ningún objeto o función, se puede utilizar el siguiente fragmento de código:

```
if (p == NULL)
   printf("El puntero es nulo\n");
else
   printf("El contenido de *p es\n", *p);
```

6.5. El operador `sizeof`

Este operador devuelve el tamaño en bytes que ocupa un tipo o variable en memoria. El siguiente fragmento de cedido imprime el número de bytes que ocupan distintos tipos de datos. Este operador se puede aplicar a tipos de datos y a variables.

```
#include <stdio.h>

int main(void)
{
  float num = 8.0f;

  printf("Un int ocupa %d bytes\n", sizeof(int));
  printf("Un char ocupa %d bytes\n", sizeof(char));
  printf("Un float ocupa %d bytes\n", sizeof(float));
  printf("Un double ocupa %d bytes\n", sizeof(double));
  printf("La variable num ocupa %d bytes\n", sizeof(num));

  return 0;
}
```

6.6. Ámbito de las variables y tipos de almacenamiento

Existen dos formas de caracterizar una variable:

- Por su *tipo de datos*.
- Por su *tipo de almacenamiento*.

El tipo de dato se refiere al tipo de información que representa la variable (`int`, `char`, ...). El tipo de almacenamiento se refiere a su permanencia y a su *ámbito*.

El **ámbito** de una variable es la porción del programa en la cual se reconoce la variable. Según el ámbito, las variables pueden ser:

- Variables locales.
- Variables globales.

Según el tipo, las variables pueden ser:

- Variables automáticas.
- Variables estáticas.
- Variables externas.
- Variables de tipo registro.

6.6.1. Variables locales

Las variables locales solo se reconocen dentro de la función donde se definen. Son invisibles al resto. Una variable local normalmente no conserva su valor una vez que el control del programa se transfiere fuera de la función. Así, en el siguiente ejemplo, la variable local `a` de la función `main` es distinta a la variable `a` de la función `funcion1`.

```
#include <stdio.h>
void funcion1(void);

int main(void)
{
   int a = 1;     /* variable local */
   int b = 2;     /* variable local */

   funcion1();
   printf("a = %d, b = %d \n", a, b);

   return 0;
}
void funcion1(void)
{
   int a = 3;     /* variable local */
   int c = 4;     /* variable local */

   printf("a = %d, c = %d \n", a, c);
   return;
}
```

6.6.2. Variables globales

Una variable global se declara fuera de las funciones y antes de su uso, y pueden ser accedidas desde cualquier función. Así, en el siguiente ejemplo `a` es una variable global y puede accederse desde la función `main` y la función `funcion1`.

```
#include <stdio.h>

void funcion1(void);

int a = 1000;     /* variable global */

int main(void)
{
```

```
    int b = 2;     /* variable local */

    funcion1();
    printf("a = %d, b = %d \n", a, b);

    return 0;
}
void funcion1(void)
{
    int c = 4;     /* variable local */

    printf("a = %d, c = %d \n", a, c);
    return;
}
```

Las variables globales mantienen los valores que se les asignan en las funciones. Es mejor hacer uso de variables locales para evitar efectos secundarios o laterales, como el que se muestra en el siguiente ejemplo.

```
#include <stdio.h>
void funcion1(void);
int a;     /* variable global */

int main(void)
{

    printf("Antes a = %d\n", a);
    funcion1();
    printf("Despues a = %d\n", a);

    return 0;
}
void funcion1(void)
{
    a = 1000;
    return;
}
```

Las variables globales se almacenan en el segmento de datos asociado al proceso.

6.6.3. Variables automáticas

Las variables automáticas son las variables locales que se definen en las funciones. Su ámbito es local y su *vida* se restringe al tiempo en el que está activa la función. Los parámetros formales se tratan como variables automáticas. Este tipo de variables se pueden especificar con la palabra reservada `auto` aunque no es necesario.

Una variable automática sin inicializar puede tomar cualquier valor, no se puede asumir que tendrá valor 0. Este tipo de variables se almacenan en la pila de ejecución asociada al proceso.

```
#include <stdio.h>

int main(void)
{
    auto int valor; /* equivalente a int valor */

    valor = 5;
    printf("El valor es %d\n", valor);

    return 0;
```

```
}
```

6.6.4. Variables estÃ¡ticas

El ámbito de una variable estática es local a la función en la que se define, sin embargo, su vida coincide con la del programa por lo que retienen sus valores durante toda la vida del programa. Estas variables se almacenan en el segmento de datos del proceso.

Se especifican con la palabra reservada `static`.

Ejemplo El siguiente ejemplo hace uso de una variable de tipo `static` para contabilizar el número de veces que se invoca a una función.

```
#include <stdio.h>
void funcion(void);

int main(void)
{
    funcion();
    funcion();
    funcion();

   return 0;
}

void funcion(void)
{
    static int veces = 0;

    veces = veces + 1;
    printf("Se ha llamado %d veces a funcion\n",     veces);

    return;
}
```

6.6.5. Variables de tipo registro

Este tipo de variables informan al compilador que el programador desea que la variable se almacene en un lugar de rápido acceso, generalmente en los registros del computador. Si no existen registros disponibles se almacenará en memoria.

Este tipo de variables se especifican con la palabra reservada `register` como se muestra en el siguiente ejemplo. Esta palabra reservada no tiene mucho uso hoy en día, puesto que los compiladores ya intentan hacer esto en la medida de lo posible.

```
#include <stdio.h>

int main(void)
{
    register int j;

    for (j = 0; j < 10; j++)
        printf("Contador = %d\n", j);

    return 0;
}
```

6.6.6. Variables externas

Una variable externa es una variable global. Hay que distinguir entre *definición* y *declaración* de una variable externa.

- Una **definición** se escribe de la misma forma que las variables normales y reserva espacio para la misma en memoria.
- Una **declaración** no reserva espacio de almacenamiento. Se especifica con `extern`.

Este tipo de variables se emplean cuando un programa consta de varios módulos (archivos `.c`) de tal manera que en uno en uno de ellos se define la variable y en los demás se declara con la palabra reservada (`extern`). Si se define en todos los módulos, el enlazador generará un error al obtener el ejecutable indicando que se está redefiniendo múltiples veces una misma variable. Si se declara en todos, pero no se define en ninguno, el enlazador también generará un error, al indicar que la variable no se define en ningún sitio.

Ejemplo Considere un programa compuesto por dos módulos. El Módulo principal que contiene la función `main` (almacenado en el fichero `main.c`) es el siguiente:

```
#include <stdio.h>

extern int valor;   /* se declara */

void funcion(void);

int main(void)
{
   funcion();
   printf("Valor = %d\n", valor);

   return 0;
}
```

El módulo auxiliar (`aux.c`) es el sigiuente:

```
int valor;      /* se define la variable */

void funcion(void)
{
   valor = 10;
   return;
}
```

Como puede verse ambos archivos utilizan la variable `valor` de tipo `int`. La variable se define en el módulo auxiliar (se reserva memoria para ella) y se declara (se utiliza) en los dos.

Para obtener el ejecutable, se compilan los módulos por separado de la siguiente forma:

```
gcc -c -Wall main.c
gcc -c -Wall aux.c
```

Estos dos mandatos obtienen dos módulos objetos: `main.o` y `aux.o`. El ejecutable (`prog`) se genera de la siguiente forma:

```
gcc main.o aux.o -o prog
```

Capítulo 7

Cadenas de caracteres

7.1. Introducción

Una **cadena de caracteres** es un conjunto o vector de caracteres.

- `'¡'` representa un carácter individual.
- `"Hola"` representa una cadena de caracteres.
- `"a"` representa una cadena de caracteres compuesta por un único carácter.
- Todas las cadenas de caracteres en C finalizan con el **carácter nulo** de C (`'\0'`). Este carácter indica el fin de una cadena de caracteres.

La cadena `"hola"` se almacena en memoria utilizando 5 bytes (véase la Figura 7.1) y su longitud es 4 (no se incluye el carácter nulo).

7.2. Declaración de cadenas de caracteres

La definición:

```
char cadena[] = "Hola";
```

Declara una cadena denominada `cadena` y reserva espacio para almacenar los siguientes caracteres:

```
'H'  'O' 'L' 'A' '\0'
```

Por ejemplo:

```
#include <stdio.h>

int main(void)
{
   char cadena[] = "hola";
```

'H'	'o'	'l'	'a'	'\0'

Figura 7.1: Ejemplo de cadena de caracteres

```
    printf("La cadena es %s  \n", cadena);
    printf("Los caracteres son: \n");
    printf("%c \n", cadena[0]);
    printf("%c \n", cadena[1]);
    printf("%c \n", cadena[2]);
    printf("%c \n", cadena[3]);
    printf("%c \n", cadena[4]);

    return 0;
}
```

En el programa anterior, `cadena[i]` representa el iésimo carácter de la cadena.
La declaración

```
char cadena[80];
```

Declara una cadena de caracteres denominada `cadena` compuesta por 80 caracteres incluido el carácter nulo.
La declaración

```
char cadena[4] = "Hola";
```

Declara una cadena de exactamente 4 caracteres que no incluye el carácter nulo por lo que no se tratará de forma correcta como una cadena de caracteres.

7.3. Asignación de valores a cadenas de caracteres

La asignación de valores iniciales a una cadena se puede realizar en su declaración:

```
char cadena[5] = "Hola";
char cadena[10] = "Hola";
```

NO se puede asignar valores de la siguiente forma:

```
cadena = "Hola";
```

Una forma de asignar un valor a una cadena es mediante la función de biblioteca `strcpy` declarada en el archivo de cabecera `string.h`.

```
strcpy(cadena, "Hola");
```

`cadena` debe tener suficiente espacio reservado.
La función `strcpy(cadena1, cadena2)` copia `cadena1` en `cadena2` incluyendo el carácter nulo.
La función `strcpy` se encuentra declarada en el archivo de cabecera `string.h`

7.4. Lectura y escritura de cadenas de caracteres

El siguiente programa ilustra cómo leer y escribir una cadena de caracteres.

```
#include <stdio.h>

#define TAM_CADENA    80

int main(void)
{
    char cadena[TAM_CADENA];

    printf("Introduzca una cadena: ");
```

```
    scanf("%s", cadena);

    printf("La cadena es %s\n", cadena);

    return 0;
}
```

La función de biblioteca `scanf` deja de buscar cuando encuentra un blanco. Por lo tanto si se introduce

```
Hola a todos
```

solo se leerá `Hola`. No es necesario el operador de dirección (`&`) ya que `cadena` representa de forma automática la dirección de comienzo. Esto mismo pasa también con el resto de vectores o arays.

7.5. Paso de cadenas de caracteres a funciones

Cuando se pasa una cadena a una función se pasa la dirección de comienzo de la misma (el nombre de la cadena representa la dirección de memoria donde se almacena el primer carácter de la cadena). Por lo tanto, la cadena se puede modificar en la función.

Por ejemplo:

```
#include <stdio.h>

const int TAM_LINEA =  80;

void leer_linea(char linea[]);

int main(void)
{
    char linea[TAM_LINEA];

    leer_linea(linea);
    printf("La linea es: ");
    printf(linea);
    printf("\n");

    return 0;
}

void leer_linea(char linea[])
{
    gets(linea);
    return;
}
```

La función de biblioteca `gets()` fue sustituido en el estándar de C (C11) por la función de biblioteca `gets_s(s, n)`, que lee como mucho `n` caracteres hasta llegar al fin de línea que es descartado.

7.6. Ejemplos

El siguiente programa lee líneas hasta fin de fichero y cuenta el número de caracteres de cada línea:

```
#include <stdio.h>

const in TAM_LINEA =   80;
```

```
int longitud(char cadena[]);

int main(void)
{
   char linea[TAM_LINEA];
   int num_car;

   while (gets_s(linea, TAM_LINEA) != NULL)  {
      num_car = longitud(linea);
      printf("Esta linea tiene %d caracteres\n", num_car);
   }

   return 0;
}

int longitud(char cadena[])
{
   int j = 0;

   while (cadena[j] != '\0')
      j++;

   return(j);
}
```

El siguiente programa lee una línea en minúsculas y la convierte a mayúsculas.

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

const int TAM_LINEA =  80;

void Convertir_may(char min[], char may[]);

int main(void)
{
   char linea_min[TAM_LINEA];
   char linea_may[TAM_LINEA];

   while (gets_s(linea_min, TAM_LINEA) != NULL) {
      Convertir_may(linea_min, linea_may);
      puts(linea_may);
   }

   return 0;
}


void Convertir_may(char min[], char may[])
{
   int j = 0;

   while (min[j] != '\0') {
      may[j] = toupper(min[j]);
      j++;
   }
  may[j] = '\0';
  return;
}
```

7.7. Principales funciones de biblioteca para manejar cadenas

En `<string.h>` se declaran las funciones de biblioteca que permiten trabajar con cadenas de caracteres.

Función	Significado
`strcpy`	Copia una cadena en otra
`strlen`	Longitud de la cadena
`strcat`	Concatenación de cadenas
`strcmp`	Comparación de dos cadenas
`strchr`	Buscar un carácter dentro de una cadena
`strstr`	Buscar una cadena dentro de otra

El siguiente programa imprime la longitud de una cadena de caracteres.

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

int main(void)
{
     char cadena = "Hola";

    printf("La longitud de %s es %d\n", cadena, strlen(cadena));

   return 0;
}
```

En `<stdlib.h>` se declaran:

Función	Significado
`atoi`	Convierte una cadena a un entero (`int`)
`atol`	Convierte una cadena a un entero largo (`long`)
`atof`	Convierte una cadena a un real (`double`)

El siguiente ejemplo ilustra el uso de la función `atoi`.

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

int main(void)
{
     char cadena = "128";
     int    numero;

     mumero = atoi(cadena);

    printf("El valor de %s es numericamente %d\n", cadena, numero);
   return 0;
}
```

Capítulo 8

Vectores y matrices

8.1. Introducción

Un vector o *array* es un conjunto de valores, todos del mismo tipo, a los que se da un nombre común, distinguiendo cada uno de ellos por su índice.

$$V = (V_0, V_1, \ldots V_n)$$

El número de índices determina la dimensión del vector.

En C los datos individuales de un vector pueden ser de cualquier tipo (`int`, `char`, `float`, etc.). El índice inferior siempre es 0 y el superior siempre es $N - 1$.

8.2. Definición de un vector

Un vector unidimensional se declara:

```
tipo_dato vector[expresion];
```

donde

- `tipo_dato` es el tipo de datos de cada elemento.
- `vector` es el nombre del vector (*array*).
- `expresion` indica el número de elementos del vector.

Por ejemplo:

```
int v_numeros[20];    // vector de 20 enteros
float n[12];  // vector de 12 floats
char vector_letras[5]; // vector de 5 caracteres o cadena de caracteres
double x[5*40];  // vector de 200 elementos de tipo double
int *v_punteros[10];  // define un vector de 10 punteros a entero
```

En C el primer índice del vector es **0**. Se puede asignar valores iniciales a un vector de esta forma:

```
int n[5] = {1, 2, 18, 24, 3};
```

C también permite asignar valores iniciales a un vector usando expresiones. En C no se realizan comprobaciones para determinar si el índice que se utiliza para el acceso a un elemento se encuentra dentro del rango correcto.

8.3. Procesamiento de un vector

En C no se permiten operaciones que impliquen vectores completos. Esto quiere decir que:

- No se pueden asignar vectores completos.
- No se pueden comparar vectores completos.

El procesamiento debe realizarse elemento a elemento como se muestra en el siguiente ejemplo.

```
#include <stdio.h>

#define TAM_VECTOR   10

int main(void)
{
   int vector_a[TAM_VECTOR];
   int vector_b[TAM_VECTOR];
   int j; /* variable utilizada como indice */

   /* leer el vector a */
   for (j = 0; j < TAM_VECTOR; j++) {
      printf("Elemento %d: ", j);
      scanf("%d", &vector_a[j]);
   }

   /* copiar el vector */
   for (j = 0; j < TAM_VECTOR; j++)
      vector_b[j] = vector_a[j];

   /* escribir el vector b */
   for (j = 0; j < TAM_VECTOR; j++)
      printf("El elemento %d es %d \n", j, vector_b[j]);

   return 0;
}
```

8.4. Paso de vectores a funciones

Un vector se pasa a una función especificando su nombre sin corchetes. El nombre representa la dirección del primer elemento del vector. Cuando se pasa un vector a una función, se pasa la dirección del primer elemento (se simula el paso de parámetros por referencia) y, por tanto, se pueden modificar en las funciones.

El argumento formal correspondiente al vector se escribe con un par de corchetes cuadrados vacíos. El tamaño no se especifica.

Ejemplo El siguiente programa calcula la media de los componentes de un vector.

```
#include <stdio.h>

#define MAX_TAM 4

void leer_vector(int vector[], int N);
float media_vector(int vector[], int N);

int main(void)
```

```
{
   int v_numeros[MAX_TAM];
   float media;

   leer_vector(v_numeros, MAX_TAM);
   float = media_vector(v_numeros, MAX_TAM);

   printf("La media es %f\n", media);

   return 0;
}

void leer_vector(int vector[], int n)
{
   int j;

   for(j=0; j < n ; j++) {
      printf("Elemento %d: ", j);
      scanf("%d", &vector[j]);
   }
   return;
}


float media_vector(int vector[], int n)
{
   int j;
   float media = 0.0;

   for(j=0; j<nM; j++)
      media = media + vector[j];

   return(media / (float)n);
}
```

8.5. Punteros y vectores

El nombre del vector representa la dirección del primer elemento del vector.

```
float vector[MAX_TAM];

vector == &vector[0]
```

El nombre del vector es realmente un puntero al primer elemento del vector.

`&x[0]` $\Longleftrightarrow$ `x`
`&x[1]` $\Longleftrightarrow$ `(x+1)`
`&x[2]` $\Longleftrightarrow$ `(x+2)`
`&x[i]` $\Longleftrightarrow$ `(x+i)`

Es decir, `&x[i]` y `(x+i)` representan la dirección del iésimo elemento del vector `x`. Por lo tanto: `x[i]` y `*(x+1)` representan el contenido del iésimo elemento del vector `x`.

Cuando un vector se define como un puntero no se le pueden asignar valores ya que un puntero no reserva espacio en memoria.

`float x[10]` define un vector compuesto por 10 números reales. Reserva espacio para los elementos.

`float *x` declara un puntero a float. Si se quiere que `float x` se comporte como un vector entonces habrá que reservar memoria para los 10 elementos:

```
x = (float *) malloc(10 * sizeof(float));
```

La función de biblioteca `malloc(nb)` (`stdlib.h`) reserva un bloque de memoria de `nb` bytes. C a diferencia de lenguajes como Java no tiene recolección de basura automática, el programador es responsable de liberar aquella memoria que ya no se vaya a utilizar.

Para liberar la memoria asignada se utiliza `free()` (`stdlib.h`)

```
free(x);
```

El uso de punteros permite definir vectores de forma dinámica.

Ejemplo El siguiente programa calcula la media de un vector de tamaño especificado de forma dinámica.

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

void leer_vector(int vector[], int dim);
float media_vector(int vector[], int dim);

int main(void)
{
   int *v_numeros;
   int dimension;
   float media;

   printf("Dimension del vector: ");
   scanf("%d", &dimension);

   v_numeros = (int *) malloc(dimension*sizeof(int));

   leer_vector(v_numeros, dimension);
   float = media_vector(v_numeros, dimension);

   printf("La media es %f\n", media);
   free(v_numeros);

   return 0;
}

void leer_vector(int vector[], int dim)
{
   int j;

   for(j=0; j<dim; j++) {
      printf("Elemento %d: ", j);
      scanf("%d", &vector[j]);
   }
   return;
}


float media_vector(int vector[], int dim)
{
   int j;
   float media = 0.0;

   for(j=0; j<dim; j++)
      media = media + vector[j];

   return(media / (float) dim);
}
```

El programa anterior es equivalente al siguiente:

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

int *crear_vector(int dim);
int destruir_vector(int *vector):
void leer_vector(int *vector, int dim);
float media_vector(int *vector, int dim);

int main(void)
{
   int *v_numeros;
   int dimension;
   float media;

   printf("Dimension del vector: ");
   scanf("%d", &dimension);

   v_numeros = crear_vector(dimension);
   leer_vector(v_numeros, dimension);
   media = media_vector(v_numeros, dimension);

   printf("La media es %f\n", media);
   destruir_vector(v_numeros);

   return 0;
}

int *crear_vector(int dim)
{
   int *vec;

   vec = (int *) malloc(dim*sizeof(int));
   return(vec);
}

int destruir_vector(int *vector)
{
    return(free(vector));
}

void leer_vector(int *vector, int dim)
{
   int j;

   for(j=0; j<dim; j++) {
      printf("Elemento %d: ", j);
      scanf("%d", (vector + j));
   }
   return;
}
int media_vector(int *vector, int dim)
{
   int j;
   float media = 0.0;

   for(j=0; j<dim; j++)
      media = media + *(vector + j);
```

```
    return(media / (float)dim);
}
```

8.6. Vectores y cadenas de caracteres

Una cadena de caracteres es un vector de caracteres. Cada elemento del vector almacena un carácter. La siguiente función copia una cadena en otra:

```
void copiar(char *destino, char *fuente)
{
    while (*fuente != '\0') {
        *destino = *fuente;
        destino ++;
        fuente++ ;
    }
    *destino = '\0';
    return;
}
```

8.7. Vectores multidimensionales

Un vector multidimensional se declara de la siguiente forma:

```
tipo_dato vector[exp1] [exp2] ... [expN];
```

En el caso de una matriz o vector de 2 dimensiones:
`int matriz[20][30];`
define una matriz de 20 filas por 30 columnas.
El elemento de la fila `i` columna `j` es `matriz[i][j]`.

Ejemplo La siguiente función calcula el producto de dos matrices cuadradas. En el caso de las vectores de varias dimensiones, en el paso de parámetros es necesario especificar la dimensión que tiene todas las dimensiones a excepción de la primera.

```
void multiplicar(float a[][DIMENSION],
                 float b[][DIMENSION],
                 float c[][DIMENSION])
{
    int i, j, k;

    for(i = 0; i < DIMENSION; i++)
        for(j = 0; j < DIMENSION; j++)
        {
            c[i][j] = 0.0;
            for(k = 0; k < DIMENSION; k++)
                c[i][j] += a[i][k] * b[k][j];
        }
    return;
}
```

8.8. Punteros y vectores multidimensionales

El siguiente programa que define de forma dinámica una matriz (mediante punteros).

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

float **crear_matriz(int fil, int col);
void destruir_matriz(float **mat, int fil);
void leer_matriz(float **mat, int fil, int col);
void imprimir_matriz(float **mat, int fil, int col);

int main(void)
{
   float **matriz;
   int fil, col;

   printf("Numero de filas: ");
   scanf("%d", &fil);

   printf("Numero de columnas: ");
   scanf("%d", &col);

   matriz = crear_matriz(fil, col);
   leer_matriz(matriz, fil, col);
   imprimir_matriz(matriz, fil, col);
   destruir_matriz(matriz, fil);

   return 0;
}

float **crear_matriz(int fil, int col)
{
   int j;
   float **mat;

   mat = (float **) malloc(fil * sizeof(float *));
   for (j = 0; j < fil; j++)
      mat[j] = (float *) malloc(col * sizeof(float));

   return(mat);
}

void destruir_matriz(float **mat, int fil)
{
   int j;

   for (j = 0; j < fil; j++)
      free(mat[j]);
   free(mat);
   return;
}

void leer_matriz(float **mat, int fil, int col)
{
   int i, j;
   float dato;

   for (i = 0; i < fil; i++)
      for (j = 0; j < col; j++)
      {
         printf("Elemento %d %d: ", i,j);
         scanf("%f", &dato);
         mat[i][j] = dato;    // es equivalente a     *(*(mat + i) + j) = dato;
```

```
        }
    return;
}

void imprimir_matriz(float **mat, int fil, int col)
{
    int i, j;
    for (i = 0; i < fil; i++)
        for (j = 0; j < col; j++)
            printf("Elemento %d %d = %f\n", i,j,mat[i][j]);
    return;
}
```

Capítulo 9

Estructuras de datos

9.1. Estructuras (*struct*)

Una estructura de datos en C está compuesta de elementos individuales que pueden ser de distinto tipo. Cada uno de los elementos de una estructura se denomina **miembro**.

A continuación se muestra la definición de una estructura:

```
struct nombre_estructura {
   tipo1 miembro_1;
   tipo2 miembro_2;
   .
   .
   tipoN miembro_N;
};
```

Los miembros pueden ser de cualquier tipo excepto `void`.

9.1.1. Ejemplos

```
struct fecha {
   int mes;
   int dia;
   int anno;
};
```

declara una estructura denominada fecha.

```
struct fecha fecha_de_hoy;
```

Define una variable denominada `fecha_de_hoy` de tipo `struct fecha`.

```
struct cuenta {
   int    cuenta_no;
   int    cuenta_tipo;
   char   nombre[80];
   float  saldo;
   struct fecha ultimopago;
};
```

Define una estructura de datos denominada `cuenta`.

9.2. Procesamiento de una estructura

Los miembros de una estructura se procesan individualmente. Para hacer referencia a un miembro determinado se utiliza:

```
variable_estructura.miembro
```

El **.** se denomina **operador de miembro**.
Por ejemplo, para imprimir la fecha de hoy:

```
struct fecha hoy;

printf("%d: %d: %d\n", hoy.dia, hoy.mes, hoy.anno);
```

Se pueden copiar estructuras:

```
struct fecha hoy, ayer;

ayer = hoy;
```

Pero no se pueden comparar estructuras directamente.

```
#include <stdio.h>

struct fecha{
   int dia;
   int mes;
   int anno;
};

struct cuenta{
   int cuenta_no;
   char nombre[80];
   float saldo;
   struct fecha ultimo_pago;
};

int main(void)
{
   struct cuenta c1, c2;

   /* rellena la estructura c1 */
   c1.cuenta_no = 2;
   strcpy(c1.nombre, "Pepe");
   c1.saldo = 100000;
   c1.ultimo_pago.dia = 12;
   c1.ultimo_pago.mes = 5;
   c1.ultimo_pago.anno = 1997;

   /* asignacion de estructuras */
   c2 = c1;

   printf("No. Cuenta %d \n", c2.cuenta_no);
   printf("Nombre %s \n", c2.nombre);
   printf("Saldo %f \n", c2.saldo);
   printf("Fecha de ultimo pago: %d:%d:%d \n",
            c2.ultimo_pago.dia, c2.ultimo_pago.mes,
            c2.ultimo_pago.anno);

   return 0;
}
```

9.3. Paso de estructuras a funciones

Se pueden pasar miembros individuales y estructuras completas. Las estructuras, como todos los argumentos en C, se pasan por valor. Una función puede devolver una estructura así como punteros a estructuras. La siguiente función permite imprimir una fecha:

```
void imprimir_fecha(struct fecha f)
{
   printf("Dia: %d\n", f.dia);
   printf("Mes: %d\n", f.mes);
   printf("Anno: %d\n", f.anno);
   return;
}
```

La siguiente función lee una fecha:

```
struct fecha leer_fecha(void)
{
   struct fecha f;

   printf("Dia: ");
   scanf("%d", &(f.dia));

   printf("Mes: ");
   scanf("%d", &(f.mes));

   printf("Anno: ");
   scanf("%d", &(f.anno));

   return(f);
}
```

En el programa principal:

```
int main(void)
{
   struct fecha fecha_de_hoy;

   fecha_de_hoy = leer_fecha();
   imprimir_fecha(fecha_de_hoy);

   return 0;
}
```

9.4. Punteros a estructuras

Se utilizan gual que con el resto de variables.

```
struct punto {
   float x;
   float y;
};

int main(void)
{
    struct punto punto_1;
    struct punto *punto_2;
```

```
    punto_1.x = 2.0;
    punto_1.y = 4.0;

    punto_2 = &punto_1;

    printf("x = %f \n", punto_2->x);
    printf("y = %f \n", punto_2->y);

   return 0;
}
```

En una variable de tipo puntero a estructura los miembros se acceden con ->

Se pueden pasar punteros a funciones. En este caso los miembros de la estructura se pueden modificar dentro de la función (se simula el paso por referencia).

Un puntero a una estructura no reserva memoria, para los miembros de la estructura. Solo se reserva espacio para el propio puntero, que solo permite almacenar una dirección de memoria.

```
void leer_punto(struct punto *p);
void imprimir_punto(struct punto p);

int main(void)
{
   struct punto *p1;

   p1 = (struct punto *)malloc(sizeof(struct punto));

   leer_punto(p1);
   imprimir_punto(*p1);
   free(p1);

   return 0;
}

void leer_punto(struct punto *p)
{
   printf("x = ");
   scanf("%f", &(p->x));
   printf("y = ");
   scanf("%f", &(p->y));

   return;
}

void imprimir_punto(struct punto p)
{
   printf("x = %f\n", p.x);
   printf("y = %f\n", p.y);

   return;
}
```

9.5. Vectores de estructuras

Se pueden definir vectores cuyos elementos sean estructuras:

```
int main(void)
{
```

```
    struct punto vector_puntos[10];
    int j;

    for(j = 0; j < 10; j++)  {
       printf("x_%d = %f\n", j, vector_puntos[j].x);
       printf("y_%d = %f\n", j, vector_puntos[j].y);
    }

    return 0;
}
```

9.6. Uniones

Una `unión` contiene miembros cuyos tipos de datos pueden ser diferentes (igual que las estructuras). Su declaración es similar a las estructuras:

```
union nombre_estructura {
   tipo1 miembro_1;
   tipo2 miembro_2;
   .
   .
   tipoN miembro_N;
};
```

Todos los miembros que componen la unión comparten la misma zona de memoria Una variable de tipo `unión` solo almacena el valor de uno de sus miembros.

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

union numero{
   int entero;
   float real;
};

int main(void)
{
   union numero num;

   /* leer un entero e imprimirlo */
   printf("Entero: ");
   scanf("%d", &(num.entero));
   printf("El entero es %d\n", num.entero);

   /* leer un real e imprimirlo */
   printf("Real: ");
   scanf("%f", &(num.real));
   printf("El entero es %f\n", num.real);

   return 0;
}
```

9.7. Tipos enumerados

Un tipo enumerado permite definir un tipo de datos formado por un conjunto de valores de tipo entero . Sus miembros son constantes de tipo `int`. Su definición se hace de la siguiente forma:

```
enum nombre {m1, m2, ..., mN};
```

Por ejemplo:

```
enum color {negro, blanco, rojo};
```

```
#include <stdio.h>
enum diasemana {lunes,martes,miercoles,jueves,
           viernes,sabado,domingo};
int main(void)
{
   enum diasemana dia;

   for(dia=lunes; dia<=domingo; dia++)
      switch(dia) {
         case lunes:
                printf("Lunes es laboral\n");
                break;
         case martes:
                printf("Martes es laboral\n");
                break;
         case miercoles:
                printf("Miercoles es laboral\n");
                break;
         case jueves:
                printf("Jueves es laboral\n");
                break;
         case viernes:
                printf("Viernes es laboral\n");
                break;
         case sabado:
                printf("Sabado no es laboral\n");
                break;
         case domingo:
                printf("Domingo no es laboral\n");
                break;
      }

   return 0;
}
```

9.8. Definición de tipos de datos (`typedef`)

Un nuevo tipo se define como:

```
typedef tipo nuevo_tipo;
```

Por ejemplo:

```
typedef char letra;
typedef struct punto tipo_punto;
```

```
letra c;
tipo_punto p;
```

9.9. Estructuras de datos autorreferenciadas

Una estructura de datos autorreferenciada es una estructura que contiene un puntero a una estructura del mismo tipo.

```
struct nombre_estructura {
   tipo1 miembro_1;
   tipo2 miembro_2;
   .
   .
   tipoN miembro_N;
   struct nombre_estructura *puntero;
};
```

9.10. Listas enlazadas

Es una estructura de datos compuesta por un conjunto de elementos enlazados.
Para definir una lista de numero enteros hay que utilizar una estructura de datos autorreferenciada:

```
struct elemento{
   int num;
   struct elemento *enlace;
};

typedef struct elemento lista_num;

lista_num *lista;
```

Una lista es una estructura de datos dinámica que puede crecer según las necesidades del programa (a diferencia de los vectores).

9.10.1. Ejemplo de lista enlazada

A continuación se muestra un código que define y hace uso de una lista enlazada de números enteros.

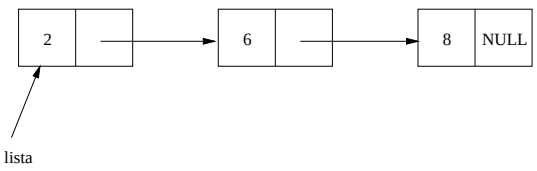

Figura 9.1: Lista enlazada

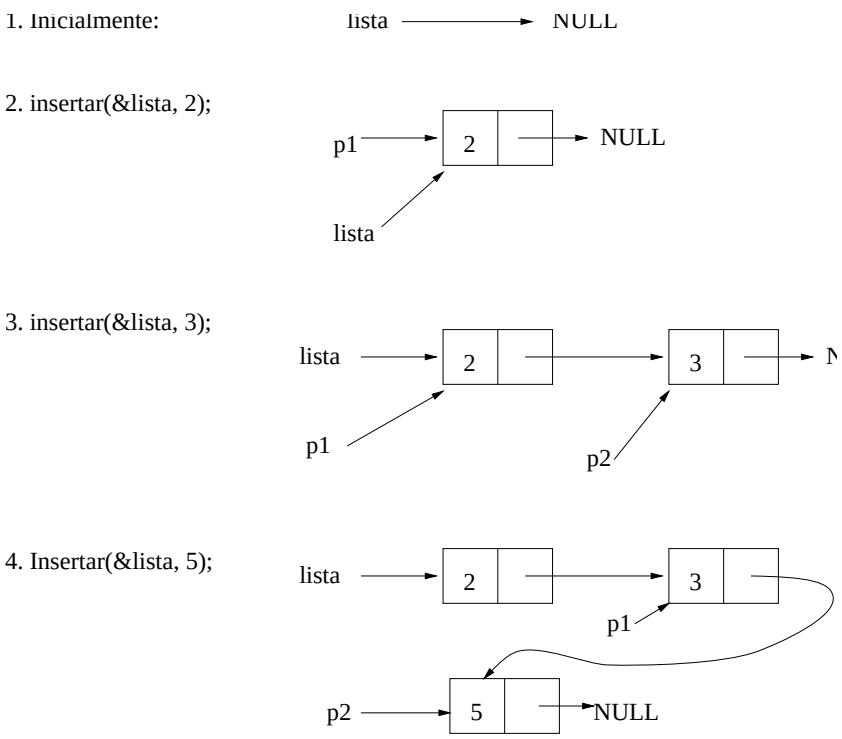

Figura 9.2: Operaciones con listas enlazadas

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

struct elemento{
   int num;
   struct elemento *enlace;
};
typedef struct elemento lista_num;


void insertar(lista_num **ptr, int num);
void mostrar(lista_num *ptr);
void borrar(lista_num *ptr);

int main(void)
{
   lista_num *lista = NULL;  // lista inicial vacía
   int numero;

   scanf("%d", &numero);
   while(numero != 0) {
      insertar(&lista, numero);
      scanf("%d", &numero);
   }
   mostrar(lista);
   borrar(lista);

   return 0;
}

void insertar(lista_num **ptr, int num)
```

```
{
   lista_num *p1, *p2;

   p1 = *ptr;
   if (p1 == NULL) {
      p1 = (lista_num *)malloc(sizeof(lista_num));
      p1->num = num;
      p1->enlace = NULL;
      *ptr = p1;
   }
   else {
      while (p1->enlace != NULL)    // inserta por el final
         p1 = p1->enlace;

      p2 = (lista_num *)malloc(sizeof(lista_num));
      p2->num = num;
      p2->enlace = NULL;
      p1->enlace = p2;
   }

   return;
}

void mostrar(lista_num *ptr)
{
   while (ptr != NULL) {
      printf("%d \n", ptr->num);
      ptr = ptr->enlace;
   }

   return;
}

void borrar(lista_num *ptr)
{
   lista_numeros *aux;

   while (ptr != NULL) {
      aux = ptr->enlace;
      free(ptr);
      ptr = aux;
   }

   return;
}
```

9.11. Diferencia entre vectores y listas

Un vector definido de la siguiente forma:

```
int vector[10];
```

reserva un número fijo de elementos antes de la compilación. Un vector creado de la siguiente forma:

```
int *vector;
vector = (int *)malloc(DIM*sizeof(int));
```

reserva un número fijo de elementos que puede variar de una ejecución a otra (en función de DIM).

Una lista de enteros puede almacenar un número que puede variar durante la ejecución (no es fijo).

Capítulo 10

Entrada/salida

10.1. Apertura y cierre de un archivo

Para abrir un archivo:

```
desc = fopen(nombre_archivo, modo);
```

donde `desc` se declara como:

```
FILE *desc;
```

y `modo` especifica la forma de apertura del archivo. Si `fopen` devuelve `NULL` el fichero no se pudo abrir.

Modo	Significado
`"r"`	Abre un archivo existente para lectura.
`"w"`	Abre un nuevo archivo para escritura. Si existe el archivo se borra su contenido. Si no existe se crea.
`"a"`	Abre un archivo existente para añadir datos al final. Si no existe se crea.
`"r+"`	Abre un archivo existente para lectura y escritura.
`"w+"`	Abre un archivo nuevo para escritura y lectura. Si existe lo borra. Si no existe lo crea.
`"a+"`	Abre un archivo para leer y añadir.

Para cerrar el fichero:

```
fclose(desc);
```

El siguiente ejemplo ilustra el código necesario para abrir un fichero para escritura.

```
#include <stdio.h>

int main(void)
{
   FILE *desc;

   desc = fopen("ejemplo.txt", "w");
   if (desc == NULL)
   {
      printf("Error, no se puede abrir el archivo\n");
   }
```

```
    else
    {
        /* se procesa el archivo */

        /* al final se cierra */
        fclose(desc);
    }
    return 0;
}
```

10.2. Lectura y escritura

Para leer:

```
fscanf(desc, formato, ...);
```

Para escribir:

```
fprintf(desc, formato, ...);
```

Ejemplo:

```
fscanf(desc, "%d %f", &num1, &num2);

fprintf(desc, "%d\n", num1);
```

Estas funciones son similares a las funciones `scanf` y `pritntf`.

Ejemplo El siguiente programa copia un archivo en otro.

```
#include <stdio.h>

int main(void)
{
    FILE *fent;
    FILE *fsal;
    char car;

    fent = fopen("entrada.txt", "r");
    if (fent == NULL){
        printf("Error abriendo entrada.txt \n");
        return -1;
    }

    fsal = fopen("salida.txt", "w");
    if (fsal == NULL){
        printf("Error creando entrada.txt \n");
        close(fent);
        return -1;
    }

    while (fscanf(fent, "%c", &car) != EOF)
        fprintf(fsal, "%c", car);

    fclose(fent);
    fclose(fsal);
```

```
    return 0;
}
```

La lectura del archivo se realiza de forma secuencial, cuando se llega al fin de fichero, la función `fscanf` devuelve `EOF` (*end of file*).

10.3. Argumentos en la línea de mandatos

Cuando se ejecuta un programa se pueden pasar argumentos a la función `main` desde la línea de mandatos.

```
nombre_programa arg1 arg2 arg3 ... argn
```

El prototipo de la función `main` en este caso es el siguiente:

```
int main(int argc, char *argv[])
```

donde:

- `argc` indica el número de argumentos que se pasa incluido el nombre del programa.
- `argv[0]` almacena el nombre del programa (el nombre del ejecutable).
- `argv[1]` almacena el primer argumento que se pasa.
- `argv[i]` almacena el iésimo argumento que se pasa.

Ejemplo El siguiente programa recibe como argumento el nombre de dos archivos compuestos de números enteros:

```
copiar_enteros archivo_entrada archivo_salida
```

el programa copia `archivo_entrada` en `archivo_salida`.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
   FILE *fent;
   FILE *fsal;
   int num;

   if (argc != 3){
      printf("Uso: %s fich_entrada fich_salida\n", argv[0]);
      return -1;
   }

   fent = fopen(argv[1], "r");
   if (fent == NULL) {
      printf("Error abriendo entrada.txt \n");
      return -1;
   }


   fsal = fopen(argv[2], "w");
   if (fsal == NULL)   {
      printf("Error creando entrada.txt \n");
      close(fent);
      return -1;
```

```
    }

    while (fscanf(fent, "%d", &num) != EOF)
       fprintf(fsal, "%d\n", num);

    fclose(fent);
    fclose(fsal);

    return 0;
}
```

Capítulo 11

Aspectos avanzados

11.1. Campos de bits

C permite descomponer una palabra en distintos campos de bits que pueden manipularse de forma independiente.

La descomposición puede realizarse:

```
struct marca {
   miembro_1 : i;
   miembro_2 : j;
   .
   .
   miembro_N : k;
};
```

Cada declaración de miembro puede incluir una especificación indicando el tamaño del campo de bits correspondiente (i, j, k).

C permite que los campos de bits sean de tipo *unsigned int*, *signed int* o *int*.

La ordenación de los campos de bits puede variar de un compilador a otro.

```
#include <stdio.h>

struct registro {
   unsigned a : 2;
   unsigned b : 4;
   unsigned c : 1;
   unsigned d : 1;
};

int main(void)
{
  struct registro r;

  r.a = 3; r.b = 2; r.c = 1; r.d = 0;
  printf("%d %d %d %d\n", r.a, r.b, r.c, r.c);
  printf("r requiere %d\n", sizeof(r));

  return 0;
}
```

11.2. Punteros a funciones

C permite definir punteros a funciones. La forma de declarar un puntero a una función es:

```
tipo_dato (*nombre_funcion)(argumentos);
```

La forma de llamar a la función a partir de su puntero es:

```
(*nombre_funcion)(argumentos);
```

A continuación se muestra un ejemplo.

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

int sumar(int a, int b)
{
    return(a+b);
}

int restar(int a, int b)
{
    return(a-b);
}

int main(void)
{
   int (*operacion)(int a, int b);
   int a, b, oper, error, res;

   printf("a, b: ");
   scanf("%d %d", &a, &b);

   printf("Operacion (0=suma, 1=resta): ");
   scanf("%d", &oper);

   error = false;
   switch(oper)  {
      case 0: operacion = sumar;
              break;
      case 1: operacion = restar;
              break;
      default: error = true;
   }

   if (!error){
      res = (*operacion)(a,b);
      printf("Resultado = %d\n", res);
   }

   return 0;
}
```

11.3. Funciones como argumentos

En C se pueden pasar funciones como argumentos a otras funciones. Cuando una función acepta el nombre de otra como argumento, la declaración formal debe identificar este argumento como un puntero a otra función. En el siguiente ejemplo la función `ejecutar` acepta como primer argumento un

puntero a una función que acepta dos enteros y devuelve un entero (el nombre de la función se puede considerar como la dirección de la propia función).

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

 int sumar(int a, int b)
 {
      return(a+b);
 }

 int restar(int a, int b)
 {
      return(a-b);
 }

 int ejecutar(int (*f)(int a, int b), int a, int b)
 {
   int res;
   res = (*f)(a,b);
   return(res);
 }

 int main(void)
 {
    int a, b, oper, error, res;

    printf("a, b: ");
    scanf("%d %d", &a, &b);
    printf("Operacion (0=suma, 1=resta): ");
    scanf("%d", &oper);
    error = false;

    switch(oper) {
      case 0: res = ejecutar(sumar, a, b); break;
      case 1: res = ejecutar(restar, a, b); break;
      default: error = true;
    }
    if (!error)
      printf("Resultado = %d\n", res);

   return 0;
}
```

11.4. Funciones con un número variable de argumentos

C permite la declaración de funciones con un número de argumentos variable, por ejemplo: `printf()`, `scanf()`.
El prototipo de estas funciones es:

```
int sumar(int contador, ...);
```

El primer argumento es fijo y obligatorio.
Hay que utilizar las funciones y macros definidas en `<stdarg.h>`

- `va_list`, variable local utilizada para acceder a los parÃ¡metros.

- `va_start`, inicializa los argumentos.
- `va_arg`, obtiene el valor del siguiente parámetro.
- `va_end`, deberá ser llamada una vez procesados todos los argumentos.

Ejemplo Función que suma un número variable de enteros.

```
int sumar(int contador, ...)
{
   va_list ap;
   int arg;
   int total = 0;
   int i = 0;

   va_start(ap, contador);
   for (i = 0; i < contador; i++) {
      arg = va_arg(ap, int);
      total = total + arg;
   }
   va_end(ap);
   return(total);
}
```

Ejemplo Función que imprime un conjunto indeterminado de cadenas de caracteres:

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

void imprimir(char  *s1, ...)
{
   va_list ap;
   char *arg;

   va_start(ap, s1);
   printf("%s\n", s1);
   while ((arg = va_arg(ap, char *)) != NULL)
      printf("%s\n", arg);
   va_end(ap);
   return;
}

int main(void)
{

   imprimir("1", "2", "3", NULL);
   return 0;
}
```

Ejemplo Función para imprimir argumentos.

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

enum tipos_arg {INTARG, FLOATARG, NULO};
void miprint(enum tipos_arg *args, ...)
```

```
{
  va_list ap;
  int tipo;

  va_start(ap, args);
  while((tipo = *args++) != NULO){
     switch(tipo) {
          case INTARG:
            printf("int %d\n", va_arg(ap,int));
            break;
          case FLOATARG:
            printf("float %f\n", va_arg(ap,double));
            break;
          default:
            printf("Tipo desconocido\n");
     }
  }
  va_end(ap);
  return;
}

int main(void)
{
  enum tipos_arg args[6] = {INTARG,
                            INTARG,
                            FLOATARG,
                            INTARG,
                            NULO};

  miprint(&args[0], 5, 2, 3.4, 6);

  return 0;
}
```

11.5. Compilación condicional

Las siguientes líneas:

```
#ifdef      identificador
#ifndef     identificador
```

indican la compilación condicional del texto que las sigue hasta encontrar la línea:

```
#endif
```

Para compilar con este tipo de directivas hay que invocar al compilador de la siguiente forma:

```
cc -Didentificador programa.c
```

La compilación condicional es útil en la fase de depuración y prueba de un programa. Así, en el siguiente fragmento de programa se el código incluido entre

```
#ifdef DEBUG

#endif
```

Solo se compilará y se generará código en caso de estar definida la constante `DEBUG`. Así, mientras se está depurando el programa se puede compilar de la siguiente forma:

```
cc -DDEBUG programa.c
```

Una vez que se haya comprobado el funcionamiento del programa se puede volver a generar el ejecutable definitivo sin incluir ya código de depuración. Basta con compilar de esta forma:

```
cc - programa.c
```

En este caso, ya no se define `DEBUG` y por tanto el código encerrado entre

```
#ifdef DEBUG

#endif
```

ya no se compilará.

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

int sumar(int contador, ...)
{
   va_list ap;
   int arg;
   int total = 0;
   int i = 0;

   va_start(ap, contador);
   for (i = 0; i < contador; i++){
      arg = va_arg(ap, int);

#ifdef DEBUG
      printf("total = %d\n", total);
      printf("arg = %d\n", arg);
#endif

      total = total + arg;
   }
   va_end(ap);
   return(total);
}


int main(void)
{
   int suma;

   suma = sumar(4, 1, 2, 3, 4);
   printf("suma = %d\n", suma);

   return 0;
}
```

Capítulo 12

Herramientas para el desarrollo de programas en Linux

12.1. ar: Gestor de bibliotecas

Utilidad para la creación y mantenimiento de bibliotecas estáticas. Su principal uso es crear bibliotecas de objetos: Agrupar un conjunto de objetos relacionados.

En la línea de compilación se especifica la biblioteca (por convención `libnombre.a`) en vez de los objetos. El enlazador extraerá de la biblioteca los objetos que contienen las variables y funciones requeridas.

El formato del mandato es:

```
ar opciones biblioteca ficheros...
```

A continuación se indican algunas opciones:

- -d: Elimina de la biblioteca los ficheros especificados
- -r: Añade (o reemplaza si existe) a la biblioteca los ficheros especificados. Si no existe la biblioteca se crea
- -ru: Igual que -r pero solo reemplaza si el fichero es más nuevo
- -t: Muestra una lista de los ficheros contenidos en la biblioteca
- -v: *Verbose*
- -x: Extrae de la biblioteca los ficheros especificados

Como ejemplo:

- Obtención de la lista de objetos contenidos en la biblioteca estándar de C

```
ar -tv /usr/lib/libc.a
```

- Creación de una biblioteca con objetos que manejan distintas estructuras de datos

```
ar -rv $HOME/lib/libest.a pila.o lista.o
ar -rv $HOME/lib/libest.a arbol.o hash.o
```

- Dos formas de compilar un programa que usa la biblioteca matemática y la creada

```
gcc -o pr pr.c -lm $HOME/lib/libest.a
gcc -o pr pr.c -L$HOME/lib -lm -lest
```

Cuando se utiliza una biblioteca estática el ejecutable generado incluye el código de las funciones que se encuentran dentro de la biblioteca. Todas las referencias, por tanto, han sido resueltas.

12.2. Bibliotecas dinámicas

Cuando se usa una biblioteca dinámica el ejecutable generado no incluye el código de las funciones de bibliotecas que se usan, sino un enlace a al archivo de biblioteca donde se encuentra su definición (su código). En tiempo de ejecución, se resuelve la referencia y se produce el montaje definitivo. El código generado con este tipo de bibliotecas da lugar a ficheros ejecutables más pequeños. Además, permite realizar cambios en las bibliotecas sin necesidad de volver a general los ejecutables de las aplicaciones.

La creación de una biblioteca dinámica a partir de las funciones definidas en un archivo fuente `ejemplo.c` se puede realizar de la siguiente forma:

```
gcc  -fPIC -c -o  ejemplo.o ejemplo.c
gcc -shared -fPIC -o libejemplo.so   ejemplo.o
```

12.3. gdb: Depurador de programas

Permite que el usuario pueda controlar la ejecución de un programa y observar su comportamiento interno mientras ejecuta. El programa debe compilarse con la opción -g. Algunas de sus funciones son:

- Establecer puntos de parada en la ejecución del programa (*breakpoints*)
- Examinar el valor de variables
- Ejecutar el programa línea a línea

El formato del mandato es:

```
gdb programa
```

Algunos mandatos del depurador `gdb` son los siguientes:

- `run`: Arranca la ejecución del programa
- `break`: Establece un *breakpoint* (un número de línea o el nombre de una función)
- `list`: Imprime las líneas de código especificadas
- `print`: Imprime el valor de una variable
- `continue`: Continúa la ejecución del programa después de un *breakpoint*
- `next`: Ejecuta la siguiente línea. Si se trata de una llamada a función, la ejecuta completa
- `step`: Ejecuta la siguiente línea. Si se trata de una llamada a función, ejecuta solo la llamada y se para al principio de la misma.
- `quit`: Termina la ejecución del depurador

12.4. make: Herramienta para el mantenimiento de programas

Facilita el proceso de generación y actualización de un programa. Determina automáticamente qué partes de un programa deben recompilarse ante una actualización de algunos módulos y las recompila.

Para realizar este proceso, `make` debe conocer las dependencias entre los ficheros: Un fichero debe actualizarse si alguno de los que depende es más nuevo. `make` consulta un fichero (`Makefile`) que contiene las reglas que especifican las dependencias de cada fichero objetivo y los mandatos para actualizarlo.

El formato de una regla es:

```
objetivo: dep1 dep2 ...
TABmandato1
TABmandato2
TABmandato3
...
```

El formato del mandato es:

```
make objetivo
```

Que implica:

1. Encontrar la regla correspondiente a objetivo
2. Tratar sus dependencias como objetivos y actualizarlas recursivamente.
3. Actualizar objetivo si alguna de las dependencias es más actual.

Sin argumentos `make` activa la primera regla. Se pueden definir también macros:

```
NOMBRE=VALOR
```

Para acceder al valor de una macro:

```
$(NOMBRE) o ${NOMBRE}
```

Existen macros especiales: $@ corresponde con el nombre del objetivo.

Se pueden especificar reglas basadas en la extensión de un fichero. Algunas de ellas están predefinidas (p.ej. la que genera el .o a partir del .c).

Ejemplo

```
CC=gcc
CFLAGS=-g
OBJS2=prac2.o aux2.o

all: prac1 prac2

prac1: prac1.o aux1.o
       gcc -g -o prac1 prac1.o aux1.o

prac2: $(OBJS2)
       ${CC} ${CFLAGS} -o  \@+ ${OBJS2}

prac1.o prac2.o: prac.h

cleano:
       rm -f prac1.o aux1.o ${OBJS2}
```

Capítulo 13

Principales bibliotecas estándar

A continuación se enumeran las principales bibliotecas de C:

- `<complex.h>`: define macros y declara funciones para trabajar con números complejos.
- `<ctype.h>`: declara funciones para clasificar y trabajar con caracteres. Por ejemplo: saber si un carácter es un número, un blanco, etc.
- `<errno.h>`: define macros relacionadas con la gestión de errores.
- `<limits.h>`: define macros con valores límites de los tipos enteros.
- `<math.h>`: declara diversas funciones matemáticas.
- `<signal.h>`: declara tipos y funciones para tratar señales.
- `<stdargs.h>`: Permite trabajar con listas de argumentos cuyo número y tipo se desconoce en la llamada a la función.
- `<stdbool.h>`: permite trabajar con valores booleanos (`true` y `false`).
- `<stdio.h>`: declara tipos y funciones para realizar operaciones de entrada-salida.
- `<string.h>`: declara funciones para trabajar con cadenas de caracteres.
- `<threads.h>`: declara tipos y funciones para trabajar con *threads*.
- `<time.h>`: declara tipos y funciones para manipular fechas.
- `<uchar.h>`: declara tipos y funciones para trabajar con caracteres Unicode.
- `<stdint.h>`: define diversos tipos para trabajar con números enteros.
- `<stdlib.h>`: declara funciones de utilidad general, que permite realizar, por ejemplo, conversiones entre números y caracteres, números psuedoaleatorios, procesar cadenas de caracteres, funciones de gestión de memoria (como `malloc()` y `free()`), etc.

13.1. Archivos de cabecera y bibliotecas

No hay que confundir el concepto de archivo de cabecera (como `<stdio.h>`) con el concepto de biblioteca. Un archivo de cabecera **No** es una biblioteca.

Un archivo de cabecera es un archivo de texto que incluye definiciones de tipos y declaraciones de funciones (prototipos). Estas funciones no están definidas en los archivos de cabecera. La definición (el código) de la función se encuentra en las diferentes bibliotecas.

Una biblioteca incluye el código objeto de de una serie de funciones relacionadas entre sí.

Así por ejemplo, la función `printf()` se encuentra declarada en el archivo de cabecera `<stdio.h>` y se encuentra definida en la biblioteca `libc`. No es necesario incluir de forma explícita esta biblioteca en el momento de compilación puesto que siempre se incluye por defecto. Para compilar un programa (`programa.c`) que utilice esta función basta con indicar:

```
gcc programa.c  -o ejecutable
```

Sin embargo, cuando un programa utiliza la función de biblioteca `cos()` para calcular el coseno de un ángulo, es necesario incluir en el programa el archivo de cabecera `<math.h>` que incluye la declaración de esta función y otras funciones matemáticas. Sin embargo, en este caso es necesario incluir la biblioteca (`libm`) donde se encuentra definida la función (no se encuentra definida en `libc`).

Para ello hay que compilar el programa de esta forma:

```
gcc programa.c  -o ejecutable  -lm
```

Con la opción `-lm`, se indica al enlazador que utilice la biblioteca `libm` para enlazar (todas las bibliotecas empiezan por `lib`).

Si no se hubiera indicado la opción `-lm` en la compilación anterior, se habrá obtenido un error como el siguiente:

```
/tmp/ccaQ4NDw.o: In function ‘main’:
programa.c:(.text+0x1c): undefined reference to ‘cos’
collect2: ld returned 1 exit status
```

que indica que la función `cos()` no se encuentra definida.

Cuando se tenga duda sobre qué archivo de cabecera incluir y qué biblioteca utilizar, conviene utilizar el manual (mandato `man`) disponible en Linux. Cuando se quiere obtener ayuda de la función `cos()` se puede ejecutar en la línea de mandatos:

```
man cos
```

Aparecerá por pantalla:

```
NAME
       cos, cosf, cosl - cosine function

SYNOPSIS
       #include <math.h>

       double cos(double x);
       float cosf(float x);
       long double cosl(long double x);

       Link with -lm.
```

```
   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       cosf(), cosl(): _BSD_SOURCE || _SVID_SOURCE || _XOPEN_SOURCE >= 600
         || _ISOC99_SOURCE;
       or cc -std=c99

DESCRIPTION
       The cos() function returns the cosine of x, where x is given in radians.

RETURN VALUE
       On success, these functions return the cosine of x.

       If x is a NaN, a NaN is returned.

       If x is positive infinity or negative infinity, a domain error
 occurs,  and  a  NaN  is   returned.

ERRORS
       See  math_error(7)  for  information  on how to determine
 whether an error has occurred when calling these functions.

       The following errors can occur:

       Domain error: x is an infinity
              An invalid floating-point exception (FE_INVALID) is raised.

       These functions do not set errno.
```

En esta salida se indica el archivo de cabecera a incluir (`<math.h>`) y cómo debe realizarse el enlace (`Link with -lm`). Además se muestra la descripción de la función, el valor que devuelve y los posibles errores.