

# INTRODUCCIÓN AL LENGUAJE C



**ACM Capítulo de Estudiantes**

Facultad de Informática, UPM

Marzo 2007

## **Introducción al lenguaje C**

©2003-2007 ACM Capítulo de Estudiantes - Facultad de Informática UPM

ACM Capítulo de Estudiantes

Facultad de Informática - Universidad Politécnica de Madrid

Campus de Montegancedo s/n

28660 Boadilla del Monte

MADRID (SPAIN)

Esta obra puede ser distribuida únicamente bajo los términos y condiciones expuestos en **Creative Commons Reconocimiento-CompartirIgual 2.0** o superior (puede consultarla en <http://creativecommons.org/licenses/by-sa/2.0/es/>).

# Índice general

<b>1. Introducción</b>	<b>2</b>
1.1. Un poco de historia...	2
1.2. Características	2
1.3. C frente a C++	2
<b>2. Herramientas y software para la programación en C (entorno GNU/Linux)</b>	<b>3</b>
2.1. Compilador: gcc	3
2.1.1. Manejo de gcc	3
2.1.2. Warnings y errores	5
2.1.3. Opciones más comunes	5
2.2. Depuradores: gdb y ddd	6
2.2.1. Depurando con gdb	6
2.2.2. Depurador gráfico: ddd	6
2.2.3. Manejo	7
2.3. Control de dependencias: make	7
2.4. Manuales: man	9
2.5. Control Concurrente de Versiones: cvs	10
2.5.1. Escenario de uso	10
2.5.2. Manejo	10
2.6. Herramientas de desarrollo C sobre Windows	10
2.6.1. GCC en Windows	11
<b>3. Introducción al lenguaje C. Sintaxis</b>	<b>13</b>
3.1. Estructura de un programa simple en C	13
3.2. Tipos básicos	16
3.2.1. Tipos de datos	16
3.2.2. Calificadores	16
3.2.3. Variables	18
3.3. Constantes	19
3.3.1. Constantes numéricas	19
3.3.2. Constantes de caracteres	19
3.3.3. Constantes enumeradas	20
3.3.4. Uso del preprocesador para definir constantes simbólicas	20
3.4. Operadores	21
3.4.1. Operadores aritméticos	21
3.4.2. Operadores relacionales	22
3.4.3. Operadores lógicos	23
3.4.4. Operadores a nivel de bit (bitwise operators)	23
3.4.5. Operadores especiales	25
3.4.6. Precedencia de operadores	27
3.5. Estructuras de control	27
3.5.1. Sentencia if	28
3.5.2. Sentencia if-else	28
3.5.3. Sentencia switch	29
3.5.4. Sentencia break	31
3.5.5. Sentencia for	31
3.5.6. Sentencia while	31
3.5.7. Sentencia do-while	32

3.6.	Funciones y subrutinas . . . . .	33
3.6.1.	Paso de parámetros a funciones. Llamadas por valor . . . . .	35
3.7.	Tipos de datos compuestos . . . . .	35
3.7.1.	<i>Arrays</i> . . . . .	36
3.7.2.	Estructuras . . . . .	37
3.7.3.	Uniones . . . . .	39
<b>4.</b>	<b>El modelo de compilación de C</b> . . . . .	<b>42</b>
4.1.	Introducción . . . . .	42
4.2.	El preprocesador . . . . .	42
4.2.1.	<code>#include</code> : inclusión de otros ficheros . . . . .	42
4.2.2.	<code>#define</code> : creación de macros . . . . .	43
4.2.3.	Compilación condicional . . . . .	44
4.3.	Compilación . . . . .	44
4.4.	Enlazado . . . . .	45
4.5.	Un ejemplo sencillo . . . . .	45
<b>5.</b>	<b>Punteros</b> . . . . .	<b>47</b>
5.1.	Introducción . . . . .	47
5.1.1.	¿Qué es un puntero? . . . . .	47
5.1.2.	¿Para qué necesito yo un puntero? . . . . .	48
5.2.	Sintaxis de punteros . . . . .	48
5.2.1.	Declaración de punteros . . . . .	48
5.2.2.	Punteros genéricos . . . . .	48
5.2.3.	Los operadores de contenido “*” y de indirección “&” . . . . .	48
5.3.	Strings . . . . .	49
5.4.	Aritmética de punteros . . . . .	50
5.4.1.	Contexto . . . . .	50
5.4.2.	Tipos de operaciones . . . . .	52
5.4.3.	Ejemplos de aritmética . . . . .	52
5.5.	Estructuras y punteros . . . . .	53
5.5.1.	El operador “->” . . . . .	53
5.6.	Memoria dinámica . . . . .	54
5.6.1.	¿Qué es la memoria dinámica? . . . . .	54
5.6.2.	El mapa de memoria en Unix . . . . .	54
5.6.3.	Primitivas básicas . . . . .	55
5.7.	Arrays y punteros . . . . .	57
5.7.1.	Repaso . . . . .	57
5.7.2.	Introducción . . . . .	57
5.8.	Indirecciones múltiples . . . . .	59
5.8.1.	Declaración . . . . .	59
5.8.2.	Utilización . . . . .	60
5.8.3.	Ejemplos de uso . . . . .	60
5.8.4.	Cadenas enlazadas . . . . .	63
5.9.	Paso por referencia vs. paso por valor . . . . .	65
5.9.1.	¿Qué es el paso por referencia? . . . . .	65
5.9.2.	¿Para qué necesito el paso por referencia? . . . . .	65
5.10.	Errores con punteros . . . . .	67
5.10.1.	Comparación de punteros a cadenas . . . . .	67
5.10.2.	Punteros “a Roma” ( <i>memory leaks</i> ) . . . . .	68
5.10.3.	Doble liberación . . . . .	69
5.10.4.	Usar . en lugar de -> . . . . .	69
5.10.5.	Operar con los punteros en lugar de con los contenidos . . . . .	71
5.10.6.	Finalización de cadenas . . . . .	73

<b>6. La librería estándar de C</b>	<b>75</b>
6.1. Introducción	75
6.2. Principales ficheros de cabecera	75
6.3. <code>stdio.h</code>	75
6.3.1. Funciones para el manejo de la Entrada/Salida	75
6.3.2. Funciones para el manejo de ficheros	76
6.4. <code>stdlib.h</code>	81
6.4.1. Funciones para la conversión de tipos	81
6.4.2. Funciones para el manejo de memoria	81
6.5. <code>string.h</code>	82
6.6. <code>math.h</code>	82
6.7. <code>ctype.h</code>	83
<b>7. Temas avanzados</b>	<b>86</b>
7.1. Entrada/Salida con archivos: Un pequeño tutorial	86
7.1.1. Variables de <i>flujo</i>	86
7.1.2. Abrir el fichero	86
7.1.3. Leer y escribir en un fichero	86
7.1.4. Cerrar el fichero	88
7.1.5. Ejemplo	88
7.2. Línea de comandos	89
7.2.1. Introducción	89
7.2.2. El prototipo de la función <code>main</code>	90
7.3. Punteros a funciones	91
7.3.1. El concepto	91
7.3.2. ¿Para qué sirven?	91
7.3.3. Ejemplo de orden superior	92
7.3.4. Ejemplo de código mutante	94
7.4. Gestión de bibliotecas de funciones	95
7.4.1. ¿Qué es una biblioteca?	95
7.4.2. ¿Para qué necesito una biblioteca?	95
7.4.3. Bibliotecas en Unix/Linux: <i>ar</i>	95
7.4.4. Ejemplo	96
7.5. Llamadas al sistema: POSIX	96
7.5.1. ¿Qué es POSIX?	96
7.5.2. Llamadas POSIX para gestión de procesos	96
7.5.3. Llamadas POSIX para la gestión de memoria	100
7.5.4. Llamadas POSIX para la entrada/salida y el sistema de ficheros	101
<b>A. Material de referencia</b>	<b>105</b>
A.1. Construcciones equivalentes ADA-C	105
A.1.1. Introducción	105
A.1.2. Sintaxis Ada vs. Sintaxis C	105

# Índice de figuras

2.1. DDD en acción, cadenas enlazadas . . . . .	7
3.1. Sentencia if . . . . .	28
3.2. Sentencia if-else . . . . .	29
3.3. Sentencia switch . . . . .	30
3.4. Sentencia for . . . . .	31
3.5. Sentencia while . . . . .	32
3.6. Sentencia do-while . . . . .	33
3.7. Un array de números enteros . . . . .	36
5.1. Punteros a variables char, int y double . . . . .	47
5.2. Inicializando punteros . . . . .	49
5.3. Aritmética sobre un puntero . . . . .	50
5.4. Suma sobre un puntero a integer . . . . .	50
5.5. Visión general del mapa de memoria . . . . .	55
5.6. Un array de números primos . . . . .	57
5.7. Avanzando sobre un array . . . . .	58
5.8. Más sobre arrays . . . . .	58
5.9. Indirección doble . . . . .	59
5.10. Indirección triple . . . . .	60
5.11. Ejemplo de una cadena enlazada . . . . .	63
5.12. Colocación de strings en memoria . . . . .	67
5.13. Punteros a roma . . . . .	69
5.14. Carácter de finalización . . . . .	74

# Índice de cuadros

3.1. Constantes de Cadena . . . . .	19
3.2. Operadores aritméticos . . . . .	22
3.3. Operadores relacionales. . . . .	23
3.4. Operadores lógicos. . . . .	23
3.5. Operadores a nivel de bit . . . . .	24
3.6. Precedencia de operadores . . . . .	27



# Capítulo 1

## Introducción

### 1.1. Un poco de historia...

El lenguaje de programación C fue ideado e implementado por Dennis Ritchie en 1972 en un DEC PDP-11<sup>1</sup> usando UNIX como sistema operativo. Inicialmente, el estándar de C fue realmente la versión proporcionada por la implementación de la versión V del sistema operativo UNIX. Su rápida expansión lleva a la aparición de varias variantes y problemas de compatibilidad, por lo que en verano del 1983 se estableció un comité para crear el estándar ANSI<sup>2</sup> para C. En 1989 se adoptó finalmente el estándar y poco después aparecieron los primeros compiladores conformes a este estándar. En 1995 se adoptó la 1ª enmienda con algunos cambios de la biblioteca de funciones, y fue la base para desarrollar C++. Finalmente, en 1999 se adoptó el estándar C99 con algunas mejoras e ideas prestadas de C++. Actualmente coexisten las dos versiones, mientras los programas migran a C99.

### 1.2. Características

- C es un lenguaje estructurado de nivel medio, ni de bajo nivel como ensamblador, ni de alto nivel como Ada o Haskell. Esto permite una mayor flexibilidad y potencia, a cambio de menor abstracción.
- No se trata de un lenguaje fuertemente tipado, lo que significa que se permite casi cualquier conversión de tipos. No es necesario que los tipos sean exactamente iguales para poder hacer conversiones, basta con que sean parecidos.
- No lleva a cabo comprobación de errores en tiempo de ejecución, por ejemplo no se comprueba que no se sobrepasen los límites de los *arrays*<sup>3</sup>. El programador es el único responsable de llevar a cabo esas comprobaciones.
- Tiene un reducido número de palabras clave, unas 32 en C89 y 37 en C99.
- C dispone de una *biblioteca estándar* que contiene numerosas funciones y que siempre está disponible, además de las extensiones que proporcione cada compilador o entorno de desarrollo.

En resumen, es un lenguaje muy flexible, muy potente, muy popular, pero que no protege al programador de sus errores.

### 1.3. C frente a C++

No podemos hablar de C sin mencionar a C++, dado que es habitual no tener muy claro cuales son sus diferencias. En pocas palabras, C++ es un lenguaje para programación orientada a objetos que toma como base el lenguaje C. La programación orientada a objetos es otra filosofía de programación distinta a la de C (programación estructurada)<sup>4</sup> aunque con C++ es posible (aunque no especialmente recomendable) mezclar ambos estilos de programación. En términos generales, podemos ver a C++ como una extensión de C. Sin embargo, como se definió a partir del estándar de C de 1989, y han evolucionado por separado, hay pequeñas divergencias entre ellos.

Por lo general, se puede utilizar un compilador de C++ para compilar un programa escrito en C, inversamente la gran mayoría de los programas escritos en C son válidos en C++. De hecho actualmente la mayoría de los compiladores admiten tanto código en C como en C++.

---

<sup>1</sup>Un ordenador "prehistórico", enorme, con cintas, similar a los que salen en las películas

<sup>2</sup>Instituto Nacional de Estándares Americano (American National Standards Institute)

<sup>3</sup>Agrupación de elementos de un mismo tipo de forma consecutiva en memoria. Volveremos sobre los *arrays* más tarde

<sup>4</sup>La programación orientada a objetos queda fuera del ámbito de este texto, para más información se puede consultar cualquier libro de C++

## Capítulo 2

# Herramientas y software para la programación en C (entorno GNU/Linux)

Aunque existen múltiples plataformas, mencionamos ahora sólo las más comunes: Microsoft Windows y GNU/Linux. En Windows disponemos de herramientas propietarias de desarrollo en C (Microsoft Visual Studio, Borland, etc) y herramientas de libre distribución (djgpp, gcc, etc). Los productos propietarios suelen integrar varias herramientas en un solo producto IDE<sup>1</sup> como Visual Studio, aunque también podemos encontrar algún IDE de libre distribución. En cualquier caso, todas las herramientas de libre distribución que podamos encontrar para GNU/Linux están disponibles para Windows (gcc, gdb, wincvs, etc).

También existen IDE's disponibles para GNU/Linux pero en estas secciones no hablaremos de ellos. Recomendamos en cambio el uso de las herramientas más extendidas y comunes en entornos UNIX. Y de ellas son de las que hablaremos en las siguientes secciones. Como es lógico, la herramienta principal a la hora de programar en cualquier lenguaje es el compilador (o el intérprete en su caso). Pero no es ni mucho menos la única herramienta de la que disponemos. También necesitaremos un editor de texto con el que editaremos el código fuente y, como veremos en este capítulo, también disponemos de depuradores, sistemas de control de versiones, etc, que facilitan enormemente la vida del programador.

### 2.1. Compilador: gcc

GCC<sup>2</sup> es un compilador rápido, muy flexible, y riguroso con el estándar de C ANSI. Como ejemplo de sus múltiples virtudes, diremos que gcc puede funcionar como *compilador cruzado*<sup>3</sup> para un gran número de arquitecturas distintas. gcc no proporciona un entorno IDEs, es solo una herramienta más a utilizar en el proceso. gcc se encarga de realizar (o encargar el trabajo a otras utilidades, como veremos) el preprocesado (ver 4.2) del código, la compilación, y el enlazado. Dicho de otra manera, nosotros proporcionamos a gcc nuestro código fuente en C, y él nos devuelve un archivo binario compilado para nuestra arquitectura.

Como curiosidad, mencionar que en realidad gcc no genera código binario alguno, sino código ensamblado. La fase de ensamblado a código binario la realiza el ensamblador de GNU (*gas*), y el enlazado de los objetos resultantes, el enlazador de GNU (*ld*). Este proceso es transparente para el usuario, ya que a no ser que se lo especifiquemos, gcc realiza el paso desde código en C a un binario ejecutable automáticamente.

#### 2.1.1. Manejo de gcc

Casi siempre, gcc es invocado desde la herramienta *make*, cuyo funcionamiento se explica más adelante. Pero obviamente, debemos saber manejar mínimamente gcc para compilar nuestros programas. La sintaxis de gcc es la siguiente:

```
Usage: gcc [options] file...
```

Vamos pues a compilar nuestro primer programa con gcc, que no podría ser de otra manera, será un *hola mundo*<sup>4</sup>:

---

<sup>1</sup>“suites” que incorporan el editor, manuales de referencia, compilador...

<sup>2</sup>Originalmente acrónimo de *GNU C Compiler*. Actualmente se refiere a *GNU Compiler Collection*, debido a la posibilidad de compilar otros lenguajes como Ada, Java o Fortran

<sup>3</sup>un compilador cruzado corre bajo una arquitectura, por ejemplo Intel x86, pero el código binario que genera está diseñado para correr en otra arquitectura distinta, por ejemplo SPARC

<sup>4</sup>el primer ejemplo de cualquier tutorial de un lenguaje

**Ejemplo**

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      printf("Hola mundo\n");
6      return 0;
7  }

```

y lo compilamos ejecutando:

```

mustang@amarok:~/seminarioc/documentacion/herramientas > gcc holamundo.c
mustang@amarok:~/seminarioc/documentacion/herramientas > ls
total 68
drwxr-xr-x   4 mustang  mustang    4096 2003-10-30 14:07 .
drwxr-xr-x  16 mustang  mustang    4096 2003-10-30 13:43 ..
-rwxr-xr-x   1 mustang  mustang   5159 2003-10-30 14:00 a.out
-rw-r--r--   1 mustang  mustang    77 2003-10-30 14:00 holamundo.c
mustang@amarok:~/seminarioc/documentacion/herramientas >

```

Nos muestra un archivo `a.out`, que es el archivo ejecutable resultado de la compilación. Lo ejecutamos:

```

mustang@amarok:~/seminarioc/documentacion/herramientas > ./a.out
Hola mundo
mustang@amarok:~/seminarioc/documentacion/herramientas >

```

Veamos como se comporta si introducimos un error en el fichero:

**Ejemplo**

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      printf("Hola mundo\n");
6      a
7      return 0;
8  }

```

```

mustang@amarok:~/seminarioc/documentacion/herramientas > gcc holamundo_error.c
holamundo_error.c: In function 'main':
holamundo_error.c:7: 'a' undeclared (first use in this function)
holamundo_error.c:7: (Each undeclared identifier is reported only once
holamundo_error.c:7: for each function it appears in.)
holamundo_error.c:7: parse error before 'return'
mustang@amarok:~/seminarioc/documentacion/herramientas >

```

Como vemos gcc nos proporciona el fichero y la línea en la que ha detectado el error. El formato de la salida de error es reconocido por la mayoría de los editores, que nos permiten visitar esa posición con atajos de teclado<sup>5</sup>. Obviamente, cuando gcc genera algún error, no se crea archivo ejecutable como resultado.

<sup>5</sup>en el editor Emacs, se puede hacer compilando mediante `M-x compile`, y usando el atajo `C-x `SPC`

### 2.1.2. Warnings y errores

**DEFINICIÓN: Error:** fallo al analizar el código C que impide la generación de un ejecutable final.

**DEFINICIÓN: Warning:** advertencia del compilador al analizar el código C que no impide la generación de un ejecutable final.

Vamos a provocar que gcc se queje con un *warning*. Para ello, utilizamos el siguiente código:



#### Ejemplo

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      printf("Hola mundo\n");
6  }
```

Y lo compilamos con:

```
mustang@amarok:~/documentacion/herramientas > gcc -Wall holamundo_warning.c
holamundo_warning.c: In function 'main':
holamundo_warning.c:6: warning: control reaches end of non-void function
mustang@amarok:~/documentacion/herramientas >
```

A pesar del *warning*, gcc ha compilado un fichero ejecutable. Más adelante veremos el significado de la opción `-Wall`.

### 2.1.3. Opciones más comunes

A continuación mostramos algunas de las opciones más habituales al usar gcc:

- `--help`  
Indica a gcc que muestre su salida de ayuda (muy reducida).
- `-o <file>`  
El archivo ejecutable generado por gcc es por defecto `a.out`. Mediante este modificador, le especificamos el nombre del ejecutable.
- `-Wall`  
No omite la detección de ningún *warning*. Por defecto, gcc omite una colección de *warnings* “poco importantes”.
- `-g`  
Incluye en el binario información necesaria para utilizar un depurador posteriormente.
- `-O <nivel>`  
Indica a gcc que utilice optimizaciones en el código. Los niveles posibles van desde 0 (no optimizar) hasta 3 (optimización máxima). Utilizar el optimizador aumenta el tiempo de compilación, pero suele generar ejecutables más rápidos.



**Consejo:** No utilices optimización cuando generes un ejecutable con información de depuración (opción `-g`). Las optimizaciones introducidas pueden confundir al depurador.

- `-E`  
Sólo realiza la fase del preprocesador, no compila, ni ensambla, ni enlaza.
- `-S`  
Preprocesa y compila, pero no ensambla ni enlaza.
- `-c`  
Preprocesa, compila y ensambla, pero no enlaza.

- `-I <dir>`

Especifica un directorio adicional donde gcc debe buscar los archivos de cabecera indicados en el código fuente (ver 4.2.1).

- `-L <dir>`

Especifica un directorio adicional donde gcc debe buscar las librerías necesarias en el proceso de enlazado (ver 4.4).

- `-l<library>`

Especifica el nombre de una librería adicional que deberá ser utilizada en el proceso de enlazado (ver 4.4).

La colección completa de modificadores a utilizar con gcc se encuentra en su página de manual, `man gcc`, cuyo manejo se explica un poco más adelante (ver 2.4).

## 2.2. Depuradores: gdb y ddd

**DEFINICIÓN: Depurador:** *Herramienta que permite al programador seguir la ejecución de su programa paso a paso, así como ver el contenido de variables.*

### 2.2.1. Depurando con gdb

Para poder ejecutar un depurador sobre nuestros programas en C, debemos especificar a gcc que incluya información de depuración en los binarios que genere. Como se vio en 2.1.3, en el caso de utilizar gcc compilamos el programa con el modificador `-g`.

### 2.2.2. Depurador gráfico: ddd

DDD es un interfaz gráfico para el depurador gdb. La principal ventaja de ddd es la facilidad para mostrar los contenidos de las posiciones de memoria durante la ejecución de nuestro programa, como puede verse en la siguiente captura:

The screenshot shows the DDD interface with the following components:

- Top Panel:** File Edit View Program Commands Status Source Data Help. The address bar shows the file path: `DDD: /home/mustang/facultad/tercero/MDP/t/sivba_entrega/sources/objetos/gestor_pasajeros.cpp`.
- Left Panel:** Memory dump showing addresses from `0x8058b40` to `0x8058fa0`.
- Main Display:** A diagram of a linked list. Three nodes are shown, each with fields: `anterior`, `vuelo`, and `siguiente`. Arrows labeled 'anterior' and 'siguiente' connect the nodes. Arrows labeled 'vuelo' point to detailed object views below.
- Object Views:** Three detailed views of flight objects, each with fields: `estado_Id`, `estado_fecha_s`, `estado_fecha_ll`, `estado_origen`, `estado_destino`, `estado_plazas`, `estado_importe`, and `estado_pasajeros`.
- Bottom Panel:** Source code for the constructor:
 

```
Gestor_Pasajeros::Gestor_Pasajeros(class Almacenpasajeros * pasajeros,
                                   class Almacenvuelos * vuelos)
{
  this-> a_pasajeros=pasajeros;
  this-> a_vuelos=vuelos;
  // cout <<"Constructor Gestor_Pasajeros\n";
}
```
- Debugger Output:** A list of gdb graph display commands showing dependencies:
 

```
(gdb) graph display *((this->a_vuelos->array[0]->siguiente)->siguiente) dependent on 9
(gdb) graph display *((this->a_vuelos->array[0]->siguiente->siguiente)->anterior) dependent on 11
(gdb) graph display *((this->a_vuelos->array[0]->siguiente)->vuelo) dependent on 9
(gdb) graph display *((this->a_vuelos->array[0]->vuelo)->estado_pasajeros) dependent on 8
(gdb) graph display *((this->a_vuelos->array[0]->siguiente->vuelo)->estado_pasajeros) dependent on 13
(gdb) graph display *((this->a_vuelos->array[0]->siguiente->siguiente->vuelo)->estado_pasajeros) dependent on 14
(gdb)
```
- Status Bar:** `In display 17: (this->a_vuelos->array[0]->siguiente->siguiente->vuelo->estado_pasajeros)->elem_inicial (double-click to deref`

Figura 2.1: DDD en acción, cadenas enlazadas

### 2.2.3. Manejo

Para obtener una descripción del manejo de gdb y ddd, os remitimos a la documentación de un seminario previo de ACM: [Adá02].

## 2.3. Control de dependencias: make

La mayoría de nuestros proyectos de programación no constarán de un solo fichero fuente sino de varios (puede incluso que de centenas de ellos). Cuando editamos un fichero de código fuente y lo modificamos pasamos a compilarlo (con GCC o cualquier otro compilador). Hay tener en cuenta que puede haber otros ficheros de código fuente que dependan del que acabamos de modificar. Por lo tanto, esos ficheros de código fuente que dependen del que acabamos de modificar también deben ser compilados. La herramienta make nos evita la tediosa tarea de compilar las dependencias, por no hablar de que nos evita la necesidad de tener presente en todo momento cuales son las dependencias entre ficheros de código fuente. Para ello nos valemos de un fichero (típicamente con nombre `Makefile`) en el que declaramos las dependencias entre ficheros de código fuente y las órdenes necesarias para actualizar cada fichero. Una vez escrito el fichero `Makefile`, cada vez que cambiemos algún fichero fuente, con la orden:

```
make
```

basta para que se realicen todas las recompilaciones necesarias. Veamos ahora el formato básico del fichero `Makefile`. El fichero `Makefile` más simple está compuesto por “reglas” de este aspecto.

```

objetivo ... : prerequisites ...
               comando
               ...
               ...

```

Un objetivo suele ser el nombre de un fichero generado por un programa; ejemplos de objetivos son ficheros ejecutables u objetos. Un objetivo puede ser también el nombre de una acción a llevar a cabo, como “clean”, que veremos más adelante en un ejemplo.

Un prerequisite es un fichero que se usa como entrada para crear un objetivo. Un objetivo con frecuencia depende de varios ficheros.

Un comando es una acción que `make` ejecuta. Una regla puede tener más de un comando, cada uno en su propia línea. **Atención:** ¡hay que poner un tabulador al principio de cada línea de comando!

Normalmente un comando está en una regla con prerequisites y contribuye a crear un fichero objetivo si alguno de los prerequisites cambia. Un ejemplo de excepción a esto es el objetivo “clean”, que no tiene prerequisites. Una regla, por tanto, explica como y cuando reconstruir ciertos ficheros que son objetivos de reglas.

A continuación tenemos un ejemplo de un Makefile que describe la manera en la que un fichero ejecutable llamado “edit” depende de ocho ficheros objeto que a su vez dependen de ocho ficheros de código fuente C y tres archivos de cabecera. Los ocho ficheros de código fuente C dependen del archivo de cabecera “defs.h” (como dijimos anteriormente, mediante la directiva `#include`). Sólo los ficheros de código que definen los comandos de edición dependen de “command.h”, y sólo los ficheros de operaciones de bajo nivel que cambian el buffer del editor dependen de “buffer.h”.

```

edit: main.o kbd.o command.o display.o \
      insert.o search.o files.o utils.o
      gcc -o edit main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

main.o : main.c defs.h
      gcc -c main.c

kbd.o : kbd.c defs.h command.h
      gcc -c kbd.c

command.o : command.c defs.h command.h
      gcc -c command.c

display.c : display.c defs.h buffer.h
      gcc -c display.c

insert.o : insert.c defs.h buffer.h
      gcc -c insert.c

search.o : search.c defs.h buffer.h
      gcc -c search.c

files.o : files.c defs.h buffer.h command.h
      gcc -c files.c

utils.o : utils.c defs.h
      gcc -c utils.c

clean :
      rm -f edit *.o

```

En el ejemplo dividimos cada línea larga en dos líneas usando la contrabarra<sup>6</sup>. Para crear el fichero ejecutable “edit”, escribimos:

---

<sup>6</sup>El carácter \

```
make
```

Para borrar el fichero ejecutable y todos los ficheros objeto del directorio, escribimos:

```
make clean
```

En el fichero Makefile del ejemplo, son objetivos el fichero ejecutable “edit” y los ficheros objeto “main.o” y “kbd.o”, entre otros. Son prerequisites ficheros como “main.c” y “defs.h”. De hecho, cada fichero “.o” es tanto objetivo como prerequisite. Son comandos “gcc -c main.c” y “gcc -c kbd.c”.

Cuando un objetivo es un fichero, necesita ser recompilado si cualquiera de los prerequisites cambia. Además, cualquier prerequisite que es generado automáticamente debería ser actualizado primero. En el ejemplo, “edit” depende de cada uno de los ocho ficheros objeto; el fichero objeto “main.o” depende a su vez del fichero de código fuente “main.c” y del fichero de cabecera “defs.h”.

Un comando de shell sigue a cada línea que contiene un objetivo y prerequisites. Estos comandos de shell indican como actualizar el fichero objetivo. Recuerda que hay que poner un tabulador al principio de cada línea de comando para distinguir líneas de comando de otras líneas en el Makefile. (Ten en cuenta que `make` no sabe nada sobre cómo funcionan los comandos. Depende de tí proporcionar los comandos que actualizaran los ficheros objetivos de manera apropiada).

El objetivo “clean” no es un fichero, es simplemente el nombre de una acción. Tampoco es un prerequisite de otra regla ni tiene prerequisites. Por tanto, `make` nunca hará nada con este objetivo a no se que se le pida específicamente.

Con esto hemos visto el funcionamiento más esencial de la herramienta `make`. Pero `make` es mucho más. En la bibliografía podemos encontrar referencias a documentación más extensa de `make`.

## 2.4. Manuales: man

`man`<sup>7</sup> es un mandato de Unix/Linux que nos informa sobre el funcionamiento de otros mandatos. El funcionamiento de `man` no sólo se limita a informarnos sobre el uso de mandatos, nos informa también de funciones de librería y llamadas a funciones del sistema. Los archivos de manual de `man` se encuentran organizados en 9 secciones, de las cuales, ahora sólo estamos interesados en las 3 primeras:

Sección	Descripción
1	Programas ejecutables y comandos de la shell
2	Llamadas al sistema
3	Llamadas a funciones de biblioteca

Antes de comenzar a utilizar `man` es recomendable que nos informemos más sobre su utilización (`man man`). Aquí encontraremos algunas opciones muy útiles:

Opción	Descripción
-a	Muestra de forma consecutiva las secciones en que existe manual del comando
-k	Muestra las paginas de manual y secciones en que se hace referencia a lo buscado

En el momento de buscar información debemos tener en cuenta que algunas funciones y mandatos se encuentran en varias secciones y, por lo tanto, deberemos indicárselo al `man` antes de su ejecución. Para especificar la sección sobre la que queremos consultar, lo haremos de la siguiente forma:

```
man [nº seccion] [mandato]
```

Como ejemplo de la utilización de `man`, consultemos el uso de `printf` como llamada a función de biblioteca, mediante la siguiente línea:

```
man 3 printf
```

Ahora veamos su uso como comando de la Shell:

```
man 1 printf
```

---

<sup>7</sup>El nombre del comando viene de `manual`

## 2.5. Control Concurrente de Versiones: *cv*s

CVS significa *Concurrent Version System*. Es una herramienta que nos permite mantener nuestro trabajo en un **repositorio**, al que múltiples usuarios pueden conectarse y realizar cambios. En nuestra opinión, es una de las mejores ayudas para el trabajo en equipo, e incluso, para el trabajo individual (control de cambios en las versiones).

### 2.5.1. Escenario de uso

El usuario crea un repositorio en una máquina, que almacenará el proyecto en el que se va a trabajar. Cada vez que se crea un archivo nuevo en el proyecto, o un subdirectorio, se introduce en el repositorio con un número de versión inicial. Una vez creado el repositorio, cada usuario puede conectarse con el servidor y descargarse una copia del mismo a su máquina, trabajar con los ficheros, y enviar los cambios al servidor.

Donde se nota realmente la potencia del sistema es cuando los usuarios trabajan sobre los mismos ficheros. Mientras los cambios sean en zonas diferentes de un mismo fichero, CVS se encarga de mezclar las versiones sin interacción del usuario. El único escenario que requiere intervención del usuario es el siguiente:

- el usuario A se descarga la versión 1.3 del fichero X
- el usuario B se descarga la versión 1.3 del fichero X
- el usuario A modifica el fichero X y lo sube, generando la versión 1.4
- el usuario B modifica el fichero X en el mismo sitio y lo sube, produciendo un conflicto con los cambios del usuario A

En este caso, CVS avisa al usuario B de que tiene que reconciliar a mano ambas versiones, y le proporciona las diferencias entre sus modificaciones y las de su compañero. Una vez que el usuario B corrige a mano esos cambios, sube el fichero, generando la versión 1.5.

CVS proporciona otras muchas ventajas, como poder generar ramas a partir de un punto del proyecto, permitiendo que los usuarios trabajen en ramas distintas, y muy importante, permite recuperar cualquier versión de cualquier archivo existente. Podemos, por ejemplo, recuperar un archivo tal y como estaba hace un mes.

A continuación se muestra un ejemplo de operación sobre un repositorio:

```
Borrado gdb.tex, ya no era necesario
Cambios en las tildes de cvs.
CVS: -----
CVS: Enter Log. Lines beginning with 'CVS:' are removed automatically
CVS:
CVS: Committing in .
CVS:
CVS: Modified Files:
CVS:   cvs.tex
CVS: Removed Files:
CVS:   gdb.tex
CVS: -----
```

### 2.5.2. Manejo

Os remitimos a la estupenda documentación realizada por un compañero de ACM en [Her01].

## 2.6. Herramientas de desarrollo C sobre Windows

Cuando utilizamos Windows no tenemos tan claro que compilador utilizar ya que en Windows disponemos de herramientas portadas de Linux (*gcc.exe*) o las herramientas que Microsoft nos proporciona para desarrollar programas (Visual Studio) que podemos obtener de Biblioteca con una licencia de estudiante de forma gratuita.

### 2.6.1. GCC en Windows

La compilación de un programa utilizando el GCC se realiza de la misma forma a la vista en Linux, pero debemos tener en cuenta que no todo programa que compile en Linux compila en Windows debido a las diferencias existentes en las llamadas al sistema. Esto es debido a que Windows no cumple completamente con el estándar POSIX (ver 7.5.1) de llamadas al sistema. Estas herramientas para Windows las podemos obtener por medio de diferentes programas, ya sea bajándonos el conjunto de herramientas UNIX para Windows *MinGW*, o descargándonos desde la facultad el compilador de Ada95 (*GNAT*), el cual viene con un el GCC, ya que la compilación llevada a cabo por *GNATMAKE* es una traducción del lenguaje Ada a C y una posterior compilación en C. Estas herramientas están disponibles desde las siguientes direcciones:

- <ftp://lml.lis.fi.upm.es/pub/lenguajes/ada/gnat/3.13p/winnt/>
- <http://www.mingw.org/download.shtml>

Y recordad, tanto si elegimos utilizar el paquete de utilidades Linux que encontramos en *MinGW*, como si elegimos el *GNAT*, deberemos incluir la ruta donde instalemos estos archivos en nuestro `PATH` para que funcione la compilación. Otra posibilidad es crear todos los archivos dentro de la carpeta donde se encuentre instalado *MinGW* o el archivo `gcc.exe` ya que en caso contrario, Windows no podrá localizar el compilador `gcc` y nos mostrará ese maravilloso mensaje de error:

```
"gcc" no se reconoce como un comando interno o externo, programa o
archivo por lotes ejecutable.
```



## Capítulo 3

# Introducción al lenguaje C. Sintaxis

### 3.1. Estructura de un programa simple en C

En esta sección vamos a ver, de forma muy general, los elementos más significativos de la sintaxis de C. No explicaremos en profundidad la función de cada uno de estos elementos, ya que cada uno de ellos tiene dedicada una sección más adelante.

Todos los programas escritos en C se componen de **una o más rutinas o funciones**, no teniendo por qué estar todas en un mismo fichero (esto es, pueden escribirse programas en C cuyo código esté repartido por varios ficheros). Una de estas funciones, `main()` es de obligatoria declaración en todos los programas C, ya que será en esa función donde comience la ejecución de nuestro programa. Es decir, todos nuestros programas comenzarán como si “alguien” hubiera llamado a la función `main` por nosotros. Nuestro código deberá elaborarse y estructurarse teniendo en cuenta este punto de entrada.

Como decíamos uno de los elementos principales de todo programa C son las funciones. Cada función está compuesta por los siguientes elementos:


- **Una cabecera de función:** la misión de la cabecera es identificar la función, así como declarar el tipo de datos que devuelve.
- **Declaración de argumentos:** la declaración de argumentos informa de los parámetros que recibe la función (en caso de recibir alguno).
- **Una sentencia compuesta:** que contiene el código de la función.

El concepto de sentencia merece una explicación más detallada. Podemos considerar una sentencia **simple** como una instrucción u orden del programa. C extiende esta noción a grupos de sentencias, llamadas sentencias **compuestas**. Este tipo de sentencias sirven para almacenar otras sentencias que se ejecutarán de forma secuencial.

En cuanto a la sintaxis, las sentencias simples se delimitan por el carácter `;`, mientras que las sentencias compuestas delimitan las sentencias que contienen entre un par de llaves (`{` y `}`).

Además de funciones en un programa C podemos encontrar los siguientes elementos:

- **Comentarios:** los comentarios no son más que pequeñas anotaciones de texto que clarifican alguna parte del código. En C cada comentario debe comenzar por la cadena `/*` y terminar por `*/`



**Nota:** Puede que veas otro tipo de formato para los comentarios en algún código fuente:

```
// Texto del comentario <fin de linea>
```

No debes utilizar este tipo de comentarios cuando programes en C ya que esta forma de comentar viene del lenguaje C++. Un compilador de C (y no de C++) **no** tiene por qué aceptar comentarios de este estilo (aunque muchos lo hacen)

- **Definiciones de tipos de datos y variables globales:** pasaremos a ver estos elementos más adelante.
- **Directivas del preprocesador:** una directiva del preprocesador es una instrucción que se le da al compilador para que se realice cierta acción antes de realizar la compilación del código. Veremos esto en más profundidad en el capítulo 4

Veamos algunos ejemplos de programas simples en C:



### Ejemplo

```

1  #include <stdio.h> /* Directiva del procesador */
2
3  /* Esto es un comentario */
4
5  int main (void) /* Cabecera de funcion y lista de argumentos */
6  { /* Comienzo de la instrucción compuesta */
7
8      printf( "Hola mundo!\n" ); /* Llamada a la funcion printf */
9
10     return 1; /* Devolvemos 1 */
11 }
12

```

Nº Línea	Línea	Significado
1	/* Esto es un comentario */	Comentario
3	int main (void)	Inicio de la función main
6	printf( "Hola mundo!\n" );	printf escribe texto por pantalla
8	return 1;	Salimos de la función main, devolviendo 1



**Nota:** Como puedes observar la sintaxis de C tiene un estilo *muy conciso*, y no precisamente agradable de leer para el principiante.

El anterior ejemplo introduce algunos conceptos importantes de la sintaxis de los programas C:

- Los comentarios pueden ir en cualquier parte del código. A efectos del compilador todo texto encerrado entre `/*` y `*/` no existe. Observarás que no hemos escrito ninguna tilde en el texto del comentario. Suele ser norma no escribir caracteres “raros” tales como tildes o cedillas en los comentarios de programas, ya que algunos compiladores pueden tener problemas ante estos caracteres.
- Como ya hemos comentado, las sentencias compuestas están delimitadas por un par de llaves. En el ejemplo anterior encontramos:
  - Una sentencia compuesta, delimitada por las llaves de las líneas 4 y 9. Esta sentencia compuesta forma el cuerpo de la función `main`.
  - Dos sentencias simples (líneas 6 y 8), componentes de la anterior sentencia compuesta. Observa como **todas** las sentencias simples están delimitadas por el caracter `;`.
- La función `main` no recibe ningún parámetro. Para indicar esto se escribe `void` (vacío en inglés) en la lista de parámetros de su declaración.

El siguiente ejemplo es un programa más elaborado que calcula el área de un círculo dado un radio. Recuerda que estamos echando un vistazo muy general a la sintaxis de los programas C. No te preocupes si no entiendes alguna sentencia o construcción del programa.



## Ejemplo

```

1  #include <stdio.h> /* Directiva del preprocesador */
2
3  #define PI 3.1415 /* Directiva del preprocesador */
4
5
6  /* Funcion secundaria */
7  float areaCirculo ( float radio ) {
8      return PI*radio*radio;
9  }
10
11 /* Funcion principal */
12 int main (void) /* Cabecera de funcion y lista de argumentos */
13 { /* Comienzo de la instrucción compuesta */
14
15     float radio;
16     float area;
17
18     /* Pedimos un numero */
19     printf( "Radio? " );
20     scanf( "%f", &radio );
21
22     if (radio < 0)
23     {
24         printf( "No me des un radio negativo!\n" );
25     }
26     else
27     {
28         /* Obtenemos el area */
29         area = areaCirculo( radio );
30
31         /* Mostramos el area */
32         printf( "Area: %f\n", area );
33
34         return 1; /* Devolvemos 1 */
35     }
36 }
37

```

Nº Línea	Línea	Significado
1	#include <stdio.h>	Directiva del preprocesador
3	#define PI 3.1415	Directiva del preprocesador
6-9		Declaración y definición de la función areaCirculo
12	int main (void)	Inicio de la función main
15	float radio;	Declaración de una variable real (tipo float)
20	scanf( "%f", &radio );	Llamada a la función scanf, que maneja la lectura desde el teclado
22	if (radio < 0)	Inicio de una sentencia condicional if-else

Como puedes observar este ejemplo es un poco más complicado que el anterior. La estructura sintáctica del programa no cambia, solo se añaden nuevos elementos:

- Funciones:** Se ha añadido la función `areaCirculo`, llamada desde `main`, que calcula el área de un círculo dado su radio. Además esta función recibe un parámetro (`radio`) y devuelve un número real. Observa como la estructura de declaración de funciones es consistente.
- Directivas del procesador:** Toda línea que empiece por el carácter `#` es una directiva del preprocesador. Como ya hemos dicho explicaremos estas directivas en el capítulo 4.

- **Nuevos tipos de sentencias:** Hasta ahora habíamos visto solo sentencias de llamada a función (a la función `printf` más concretamente) y de devolución de valores desde la función (`return`). Aquí se presentan sentencias de declaración de variables (líneas 15,16) y de ejecución condicional (sentencia `if-else`).

## 3.2. Tipos básicos

Como ya sabrás, el tipo de una variable determina el dominio de valores que dicha variable debe tomar. En este capítulo veremos los tipos básicos que ofrece C al programador. Más adelante (sección 3.7) veremos como construir, a partir de este pequeño conjunto, tipos de datos más complicados.

### 3.2.1. Tipos de datos

C ofrece una colección de tipos de datos bastante limitada, aunque no por ello poco funcional. Dicha colección se compone de los siguientes tipos:

- `char`:
  - **Contenido:** Puede contener un caracter del conjunto de caracteres locales
  - **Tamaño:** Normalmente<sup>1</sup> 1 byte.
- `int`:
  - **Contenido:** Un número entero
  - **Tamaño:** El determinado por la arquitectura para números enteros. En arquitecturas Intel/x86 es 4 bytes
- `float`:
  - **Contenido:** Un número en coma flotante
  - **Tamaño:** El determinado por la arquitectura para números en coma flotante. En arquitecturas Intel/x86 es 4 bytes
- `double`
  - **Contenido:** Un número en coma flotante de precisión doble
  - **Tamaño:** El determinado por la arquitectura para números en coma flotante de doble precisión. En arquitecturas Intel/x86 es 8 bytes

### 3.2.2. Calificadores

**DEFINICIÓN: Calificador:** *Es un atributo que indica al compilador una característica de la variable declarada.*

#### `short` y `long`

Normalmente el rango de representación ofrecido por un tipo es suficientemente grande para nuestras necesidades. Sin embargo en C es posible *sugerir* al compilador que utilice más memoria a la hora de almacenar la variable de ese tipo. Esto se hace con el calificador `long`

```
long int numero_entero_grande;
long double numero_real_grande;
```



**Nota:** El estándar que regula la implementación de compiladores de C **no especifica** cuanto más grande tiene que ser una variable calificada como `long` que otra que no lo está. Lo único que especifica es que el rango de una variable `long` **no puede ser menor** que el rango de una no calificada. Es por tanto posible que una variable `long int` y una variable `int` tengan el mismo tamaño.

El calificador `short` actúa de forma contraria a `long`. Si por ejemplo disponemos de una variable que sabemos que no va a usar todo el rango ofrecido por el tipo `int` podemos calificarla como `short int` ahorrando (posiblemente) memoria.



**Nota:** Los calificadores `short` y `long` normalmente sólo están disponibles para el tipo `int`. Algunos compiladores también permiten usar el calificador `long` con el tipo `double`

<sup>1</sup>Con la introducción del juego de caracteres *Unicode* esto está empezando a cambiar. *Unicode* permite representar un amplio juego de caracteres, que incluye varios alfabetos (occidental, cirílico, etc.). El precio que se debe pagar a cambio de esta deseable funcionalidad es el que los caracteres *unicode* ocupen 2 bytes

**signed y unsigned**

Estos calificadores se usan en los tipos `char` e `int` para indicar al compilador que la variable declarada tomará valores negativos y positivos (`signed`) o sólo positivos o cero (`unsigned`).



**Nota:** A no ser que se indique lo contrario las variables de tipo `int` y `char` se califican automáticamente con `signed`

**static(local)**

Si declaramos una variable local con el modificador `static`, conseguiremos que nuestra variable sea accesible solamente desde esa función y que conserve su valor a través de sucesivas llamadas.

**Ejemplo**

```

1  #include <stdio.h>
2  #define TOTAL 5
3
4  void contador(void) {
5      static int cuenta = 0;
6
7      cuenta++;
8      printf("Cuenta: %d\n", cuenta);
9  }
10
11 int main(void) {
12     int i=0;
13     for(; i<TOTAL; i++) contador();
14
15     return 0;
16 }
```

La salida mostrada por pantalla sería:

```

Cuenta: 1
Cuenta: 2
Cuenta: 3
Cuenta: 4
Cuenta: 5
```

**static(global)**

Las variables globales son por defecto accesibles desde cualquier fuente del programa, si declaramos una variable global con el modificador `static` impediremos que estas variables sean visibles desde otro fuente. De igual modo, las funciones definidas en un fuente son utilizables desde cualquier otro. Si queremos impedir que una función sea llamada desde fuera del fuente en la que está definida usaremos el modificador `static`.

**register**

Con este modificador indicamos al compilador que la variable debe ser almacenada en un registro de la CPU (no en memoria).



**Nota:** Este modificador es únicamente aplicable a variables locales. Además hay que tener en cuenta que no podríamos obtener su dirección, ya que está almacenada en un registro y no en memoria.

```
register int i;
```

**volatile**

Este modificador se usa con variables que puedan ser modificadas desde el exterior del programa por procesos externos. Obliga al compilador a leer de memoria el valor de la variable siempre que vaya a ser usada aunque ya tenga dicho valor en algún registro de una lectura anterior. Se usa con variables en las que no se sabe con certeza cuándo se va a modificar su contenido.

```
volatile int i;
```

**3.2.3. Variables**

**DEFINICIÓN: Variable:** *Espacio de memoria, referenciado por un identificador, en el que el programador puede almacenar datos de un determinado tipo*

Declarar una variable es indicar al compilador que debe reservar espacio para almacenar valores de un tipo determinado, que serán referenciados por un identificador determinado. En C debemos declarar **todas** las variables antes de usarlas, establecer el tipo que tienen y, en los casos que sea necesario, darles un valor inicial.

**Sintaxis de declaración de variables**

La declaración de una variable es una sentencia simple con la siguiente sintaxis:

```
tipoVariable nombreVariable;
```

De esta forma declaramos que la variable `nombreVariable` es de tipo `tipoVariable`. El siguiente ejemplo declara una variable de tipo `int` (entero):

```
int contador;
```

También es posible dar un valor inicial a la variable declarada:

```
int contador = 2;
```

Como habrás adivinado esta sentencia declara la variable `contador` y le da un valor inicial de 3.

A veces es necesario declarar varias variables de un mismo tipo. Existe una forma abreviada de declaración de variables que permite hacer esto en una sola sentencia:

```
int x, y, z;
```

El anterior ejemplo declara tres variables (`x`, `y`, `z`), todas de tipo entero (`int`). Usando esta forma abreviada también es posible dar un valor inicial a las variables:

```
int x=0, y=1, z=23;
```

**Declaraciones y nombres**

A la hora de declarar una variable debemos tener en cuenta diversas restricciones :

- Los nombres de variables se componen de letras, dígitos y el carácter de subrayado `_`.
- El primer carácter del nombre debe ser una letra o el carácter de subrayado.
- Las letras mayúsculas y minúsculas son distintas en C.
- Las palabras reservadas del lenguaje no se pueden usar como nombres de variable.



**Consejo:** Elige nombres de variable que sean significativos y estén relacionados con el propósito de la misma.

### 3.3. Constantes

Las constantes son valores fijos<sup>2</sup> que no pueden ser modificados por el programa. Pueden ser de cualquier tipo de datos básico (punteros incluidos). Para marcar que queremos que una variable sea constante utilizamos la palabra reservada `const` tal que:

```
const int dummy = 321;          /* declaramos que dummy vale y valdrá siempre 321 */
```

No tiene demasiado sentido declarar una variable de tipo `const` sin darle valor inicial, pero algunos compiladores permiten hacerlo.



**Nota:** Con algunos compiladores<sup>a</sup>, si se intenta cambiar de valor una constante se genera un “warning” al compilar, pero se cambia el valor. C, como lenguaje para programadores que es, asume que sabes lo que estás haciendo y por eso continúa. En cualquier caso, como la constante ocupa espacio en memoria, siempre se podrá modificar, si *realmente* se quiere ser un chico malo, es cuestión de acceder a ella directamente a base de punteros de memoria...

<sup>a</sup>gcc, el compilador estándar de Linux, sin ir mas lejos

#### 3.3.1. Constantes numéricas

Aparte de constantes enteras tipo `234` y en coma flotante de la forma `10.4`, a veces, sobre todo al trabajar a bajo nivel, resulta más cómodo poder introducir la constante en base 8 (octal) o 16 (hexadecimal) que en base 10. Dado que es corriente usar estos sistemas de numeración, C permite especificar constantes enteras en hexadecimal u octal. Una constante hexadecimal empieza por `0x` seguido del número esa base. Igualmente una constante octal comienza por `0`:

```
const int hex = 0x80A; /* 2058 en decimal */
const int oct = 012;  /* 10 en decimal */
```

#### 3.3.2. Constantes de caracteres

Las cadenas de caracteres se representan encerrando la cadena entre comillas dobles ("`hola caracola`" sería un ejemplo). Hay ciertos caracteres que no se puede (o no es cómodo) introducir de esta forma, como son los caracteres de control: tabulador, retorno de carro, etc... Para introducirlos hay que usar unos códigos que consisten en una barra invertida y una letra, estos son los principales:

Código	Significado
<code>\n</code>	Retorno de carro
<code>\t</code>	Tabulador
<code>\"</code>	Comillas dobles
<code>\'</code>	Comillas simples
<code>\\</code>	Barra invertida

Cuadro 3.1: Constantes de Cadena

Por ejemplo, este programa escribe "hola mundo" desplazado un tabulador a la izquierda y luego salta de línea:



#### Ejemplo

```
1 #include <stdio.h>
2 int main(void){
3     printf("\thola mundo\n");
4     return 0;
5 }
6
```

<sup>2</sup>podríamos decir que son “variables constantes” lo que es un oxímoron, es decir, una contradicción, igual que “inteligencia militar”...

### 3.3.3. Constantes enumeradas

Las constantes enumeradas permiten definir una lista de constantes representadas por identificadores. Estas constantes son, realmente, enteros. Si no se especifica una correspondencia entre nombre y número el compilador se encarga de asignarles números correlativos (empezando por 0). Se pueden usar como enteros que son, pero la idea es usarlos en comparaciones, haciendo así el código más legible.

Para definir las se usa la palabra reservada `enum` tal que:

```
enum color{rojo, amarillo, morado};
enum bool{false=0, true=1};
```



**Consejo:** Definir el tipo `bool` ayuda a conseguir código más claro...

A partir de entonces se puede poner por ejemplo:

```
if(color == rojo)...
```

en lugar de tener que poner:

```
if(color == 0)...
```

(¿es o no es más claro?)

### 3.3.4. Uso del preprocesador para definir constantes simbólicas

Con el preprocesador, del que hablaremos más adelante, también se pueden definir constantes, aunque en este caso son realmente *alias*. Al compilar se sustituye tal cual un valor por otro, tenga o no sentido, y no se reserva memoria para el valor. Por ejemplo, para definir que la longitud de algo es 10 pondríamos:

```
#define LONGITUD 10
```



**Nota:** Ojo, no ponemos punto y coma al final, ni signo de equivalencia. Hemos *definido* un *alias* para 10, que es "LONGITUD". Por costumbre se suelen poner las constantes así definidas en mayúsculas, para no confundirlas con constantes normales.

Ejemplo de uso de `const`:



### Ejemplo

```

1  #include <stdio.h>
2  #define TAM 5
3
4  void modificar(const int array[]) {
5      int i;
6      for(i=0; i<TAM; i++) array[i] = -1; /* Intentamos rellenar el
7                                          * array con -1, como ha sido declarado
8                                          * constante provocaría un warning a la
9                                          * hora de compilar el programa. */
10 }
11
12 int main(void) {
13
14     int i, array[TAM];
15     for(i=0; i<TAM; i++) array[i] = i%2; /* Rellenamos el array */
16     /* Mostramos el array */
17     for(i=0; i<TAM; i++) printf("array[%d] = %d\n",i,array[i]);
18
19     printf("Modificamos el array:\n");
20
21     modificar(array);
22     array[0] = 2; /* Aqui array no ha sido declarada constante y
23                 * por tanto no tendríamos ningun problema si
24                 * intentamos modificarla. */
25
26     /* Mostramos el array tras las modificaciones */
27     for(i=0; i<TAM; i++) printf("array[%d] = %d\n",i,array[i]);
28
29     return 0;
30 }

```

## 3.4. Operadores

**DEFINICIÓN: ¿qué es un operador?:** *Un operador es un símbolo (+, -, \*, /, etc) que tiene una función predefinida (suma, resta, multiplicación, etc) y que recibe sus argumentos de manera infija, en el caso de tener 2 argumentos (a operador b), o de manera prefija o postfija, en el caso de tener uno solo (operador a , o bien, a operador).*

En C existen una gran variedad de operadores, que se pueden agrupar de la siguiente manera:

- Operadores aritméticos
- Operadores relacionales
- Operadores lógicos
- Operadores a nivel de bit (bitwise operators)
- Operadores especiales

En el siguiente apartado veremos cuáles son estos operadores, sus funciones, y por último la precedencia u orden de evaluación entre éstos.

### 3.4.1. Operadores aritméticos

Los operadores aritméticos nos permiten, básicamente, hacer cualquier operación aritmética, que necesitemos (ejemplo: suma, resta, multiplicación, etc). En la siguiente tabla se muestran los operadores de los que disponemos en C y su función asociada.

Operador	Acción	Ejemplo
-	Resta	$x = 5 - 3$ ; // x vale 2
+	Suma	$x = 2 + 3$ ; // x vale 5
*	Multiplicación	$x = 2 * 3$ ; // x vale 6
/	División	$x = 6 / 2$ ; // x vale 3
%	Módulo	$x = 5 \% 2$ ; // x vale 1
--	Decremento	$x = 1$ ; $x--$ ; // x vale 0
++	Incremento	$x = 1$ ; $x++$ ; // x vale 2

Cuadro 3.2: Operadores aritméticos

**Incrementos y Decrementos** Como hemos visto los operadores de incremento y decremento, añaden o restan una unidad a su operando. Observar, que estos operadores modifican el valor del operando:

```
x = 4;
y = x++;
/* Después de esta instrucción x valdrá 5 e */
/* y valdrá 4 (como veremos a continuación). */
```

Los incrementos y decrementos, son operadores muy útiles, y en muchas ocasiones es bastante más clara su utilización que hacer una simple suma o resta:

```
/* Un pequeño for */
int i, j;

for ( i = 0; i < 10; i = i + 1 )
    /* no hace nada */;

for ( j = 0; j < 10; j++ )
    /* hace lo mismo que el anterior: nada */;

/* Cual es más clara? */
```

Sin embargo es necesario prestar atención a lo siguiente:

Los incrementos y decrementos se pueden poner de forma prefija y postfija; y según esto pueden significar:

**Forma prefija: preincremento y predecremento** Cuando un operador de incremento o decremento precede a su operando, se llevará a cabo la operación de incremento o de decremento antes de utilizar el valor del operando. Veámoslo con un ejemplo:

```
int x, y;

x = 2004;
y = ++x;
/* x e y valen 2005. */
```

**Forma postfija: postincremento y postdecremento** En el caso de los postincrementos y postdecrementos pasa lo contrario: se utilizará el valor actual del operando y luego se efectuará la operación de incremento o decremento.

```
int x, y;

x = 2004;
y = x++;
/* y vale 2004 y x vale 2005 */
```

### 3.4.2. Operadores relacionales

Al igual que en matemáticas, estos operadores nos permitirán evaluar las relaciones (igualdad, mayor, menor, etc) entre un par de operandos (en principio, pensemos en números). Los operadores relacionales de los que disponemos en C son:

<i>Operador</i>	<i>Acción</i>
>	Mayor que
>=	Mayor o igual que
<	Menor que
<=	Menor o igual que
==	Igual
!=	Distinto

Cuadro 3.3: Operadores relacionales.

El resultado de cualquier evaluación de este tipo, es un valor “cierto” (*true*) o “falso” (*false*). La mayoría de lenguajes tienen algún tipo predefinido para representar estos valores (boolean, bool, etc); sin embargo en C, se utilizan valores enteros para representar esto:

falso (false)	0
cierto (true)	cualquier valor distinto de 0, aunque normalmente se usará el 1

Volviendo a los operadores relacionales, el resultado de una evaluación será un valor entre 0 y 1, que indicará como hemos dicho, la falsedad o certeza de esa relación.

### 3.4.3. Operadores lógicos

Como operadores lógicos designamos a aquellos operadores que nos permiten “conectar” un par de propiedades (al igual que en lógica):

```
numero = 2701;
if ( EsPrimo(numero) && (numero > 1000) ){
    /* Ejecutaremos este código si numero */
    /* es primo y numero es mayor que 100 */
}
```

Los operadores lógicos de los que disponemos en C son los siguientes:

<i>Operador</i>	<i>Acción</i>
&&	Conjunción (Y)
	Disyunción (O)
!	Negación

Cuadro 3.4: Operadores lógicos.

Al igual que con la igualdad hay que tener especial cuidado con los operadores && y ||, ya que si ponemos solamente un & o un |, nos estamos refiriendo a un “and” o un “or” a nivel de bit, por lo que el código puede que no haga lo que queremos (o que algunas veces lo haga y otras veces no).

### 3.4.4. Operadores a nivel de bit (bitwise operators)

En determinadas ocasiones nos puede interesar manipular datos a nivel de bit; por ejemplo activar o desactivar *flags*. Un *flag* es una “variable” que puede tomar 2 valores, por lo que se suele representar con un bit. Debido a que en C (y en la mayoría de lenguajes de programación) no existen tipos predefinidos de un bit, lo que se suele hacer es agrupar varios *flags* en una variable de tipo entero (“short int”, “int” o lo que queramos).

Para acceder a estos flags o simplemente para activarlos es necesario utilizar operadores a nivel de bit. Veámoslo en un ejemplo.

La llamada al sistema “*open*” (en *POSIX*) necesita que se le especifique que hacer con el fichero a abrir: crearlo si no existe, sobrescribirlo si existe, no sobrescribirlo, etc.

```

int open(const char *path, int flags);
//Donde:
//  path --> indica el path de donde se encuentra el fichero
//  flags --> indica lo que queremos hacer con el fichero ...
//
//Flags puede ser:
//  O_CREAT Si el fichero no existe, será creado.
//  O_EXCL Cuando se combina con O_CREAT, se considerará un
//          error que el fichero ya exista.
//  O_TRUNC Si el fichero ya existe, será truncado.
//  O_APPEND El fichero se abrirá en modo de sólo-añadir.
//  O_NONBLOCK El fichero se abre en modo no bloqueante.
//  ...

```

Si queremos abrir el fichero “/tmp/tutorial.c.txt” en modo lectura y que se pueda escribir al final del mismo pondremos:

```

int fd;

fd = open ("/tmp/tutorial_c.txt", O_CREAT | O_APPEND );
// también lo podríamos hacer "a pelo":
//  open ("/tmp/tutorial_c.txt", 0x440)

write(fd, "Hola mundo...\n", 14);

close (fd);

```

Aunque normalmente no se suelen utilizar, es bueno conocer como actúan estos operadores y de cuales disponemos.

<i>Operador</i>	<i>Acción</i>
&	AND a nivel de bit.
	OR a nivel de bit.
^	XOR a nivel de bit.
~	Complemento.
<<	Desplazamiento a la izquierda.
>>	Desplazamiento a la derecha.

Cuadro 3.5: Operadores a nivel de bit

A continuación describiremos cada uno de estos operadores brevemente.

**DEFINICIÓN: El operador AND (&):** *El operador AND compara dos bits; si los dos son 1 el resultado es 1, en otro caso el resultado será 0. Ejemplo:*

```

c1 = 0x45      --> 01000101
c2 = 0x71      --> 01110001
-----
c1 & c2 = 0x41 --> 01000001

```

**DEFINICIÓN: El operador OR (|):** *El operador OR compara dos bits; si cualquiera de los dos bits es 1, entonces el resultado es 1; en otro caso será 0. Ejemplo:*

```

i1 = 0x47      --> 01000111
i2 = 0x53      --> 01010011
-----
i1 | i2 = 0x57 --> 01010111

```

**DEFINICIÓN: El operador XOR (^):** *El operador OR exclusivo o XOR, dará como resultado un 1 si cualquiera de los dos operandos es 1, pero no los dos a la vez. Ejemplo:*

```
i1 = 0x47      --> 01000111
i2 = 0x53      --> 01010011
-----
i1 ^ i2 = 0x14 --> 00010100
```

**DEFINICIÓN: El operador de complemento (~):** Este operador devuelve como resultado el complemento a uno del operando:

```
c = 0x45  --> 01000101
-----
~c = 0xBA --> 10111010
```

**DEFINICIÓN: Los operadores de desplazamiento a nivel de bit (<< y >>):** Desplazan a la izquierda o a la derecha un número especificado de bits. En un desplazamiento a la izquierda los bits que sobran por el lado izquierdo se descartan y se rellenan los nuevos espacios con ceros. De manera análoga pasa con los desplazamientos a la derecha. Veamos un ejemplo:

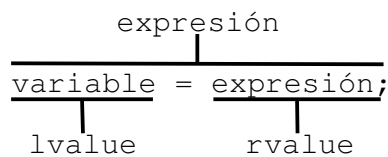
	c = 0x1C	00011100
c << 1	c = 0x38	00111000
c >> 2	c = 0x07	00000111

### 3.4.5. Operadores especiales

Por último describiremos los operadores que nos faltan:

- El operador de asignación
- El operador ternario ?
- Los operadores & y \* (unarios)
- El operador sizeof
- El operador cast
- El operador coma “;”
- Los operadores “.” y “->”
- El operador ( )
- El operador [ ]

**El operador de asignación** En otros lenguajes de programación se considera la asignación como una sentencia especial, en C, esto no es así. La asignación es un operador más. Este operador, tiene como lvalue (left value, el valor a la izquierda del operador) un variable y como rvalue una expresión válida. Puesto que es un operador, el todo es una expresión:



Es por ello que podemos hacer asignaciones múltiples:

```
int x, y, z;

x = y = z = 0;
```

Existen una serie de casos particulares de la asignación, que son las asignaciones compuestas. La asignación compuesta simplifica la escritura de operaciones en las que el lvalue aparece tanto en la derecha como en la izquierda; como en el siguiente ejemplo:

```
int x;

x = x + 2;
```

La forma abreviada de escribirlo será poniendo el operador “+” (en este caso) a continuación el igual y por último la expresión que estamos sumando al lvalue, que en este caso es 2.

```
int x;

x += 2;
```

**El operador ternario ? (operador condicional)** El operador ternario ? sirve para escribir expresiones condicionales. Su formato es el siguiente “*expresion1 ? expresion2 : expresion3*”, *expresion1* es evaluada primero, si es diferente de cero (verdadero) entonces se evalúa *expresion2* devolviéndose como resultado de la expresión condicional. Si *expresion1* es igual a cero (falso) se evalúa *expresion3* y se devuelve como resultado de la expresión condicional.

Operador ternario	Sentencia if
Cálculo de max(a,b)	
<code>z = (a &gt; b) ? a : b;</code>	<code>if (a &gt; b)</code> <code>  z=a;</code> <code>else</code> <code>  z=b;</code>

**Los operadores & y \* (unarios)** Se estudiarán en la parte de punteros (ver 5.2.3).

**El operador sizeof** El operador sizeof es un operador unario que devuelve la longitud, en bytes, de la variable o de un tipo especificado. Es muy útil para hacer un *malloc* u otras operaciones.

```
double un_double;

printf ("double: %d \n", sizeof( un_double )); //8
printf ("float : %d \n", sizeof( float )); //4
printf ("int : %d \n", sizeof( int )); //4
```

**El operador cast** El operador cast es un operador unario que hace una conversión explícita (la fuerza) del tipo de una expresión. Su formato es el siguiente “*(nombre\_de\_tipo) expresión*”, *expresión* es convertida al tipo nombrado, siempre que esa conversión sea posible.

```
int n; // n es un entero

sqrt((double) n); // se pasa n a double
//porque sqrt no acepta un int como parámetro
```

**El operador coma “,”** Encadena varias expresiones. En este caso lvalue y rvalue son expresiones válidas. El valor resultante de evaluar una expresión del tipo “*(lvalue , rvalue)*” es el resultante de evaluar el rvalue (esto no significa que el lvalue no se evalúe). Ejemplo:

```
x = ( y = 9, z = 0 ); // x valdrá 0.
```

Aunque su utilización no es precisamente la que se ha detallado en el ejemplo (puede dar lugar a códigos un tanto oscuros), puede servirnos para hacer cosas como el siguiente ejemplo, de una manera más o menos simple.

```
int i, j;
char *cadena = strdup ("hola mundo!");

printf("%s\n", cadena);
for ( i = 0, j = strlen(cadena) - 1; i > j ; i++, j--){
  cadena[i] = '*';
  cadena[j] = '*';
  printf("%s\n", cadena);
}
```

**Los operadores “.” y “->”** El operador “.” se estudiará en la parte de structs (ver 3.7.2), y el operador “->” en la parte de punteros (ver 5.5.1).

**Los operadores ( y )** Simplemente, alteran el orden de evaluación por defecto (utilizando las precedencias de los operadores); este concepto nos es bastante familiar, pues lo utilizamos también en matemáticas.

```
x = 5 * 2 + 1; // x == 11
y = 5 * (2 + 1); // x == 15
```

**Los operadores [ y ]** Se estudiarán en la parte de arrays (ver 3.7.1).

### 3.4.6. Precedencia de operadores

Una expresión está compuesta por operadores, variables y constantes. Para simplificar, podemos pensar que la forma en la que C evalúa esta expresión es dividiendo el *todo* en subexpresiones. Las reglas que definen que subexpresión evaluar primero, se denominan reglas de precedencia. Aunque siempre podemos alterar dichas reglas mediante la utilización de paréntesis. En la siguiente tabla detallamos la precedencia entre los operadores de C.

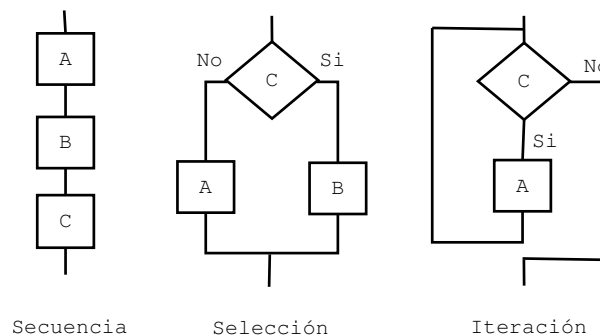
Mayor precedencia	( ) [ ] -> . ! ~ ++ * & sizeof (operadores unarios) * / % + - << >> < <= > >= == != & & &&    ?: = += -= *= /= %=
Menor precedencia	,

Cuadro 3.6: Precedencia de operadores

## 3.5. Estructuras de control

Podemos clasificar cada una de las estructuras de control más comunes en programación en uno de los siguientes tipos:

- **Secuencia:** Ejecución sucesiva de una o más operaciones.
- **Selección:** Se realiza una u otra operación, dependiendo de una condición.
- **Iteración:** Repetición de una o varias operaciones mientras se cumpla una condición.



### 3.5.1. Sentencia if

La forma general de esta sentencia es:

```
if (expresion)
    sentencia
```

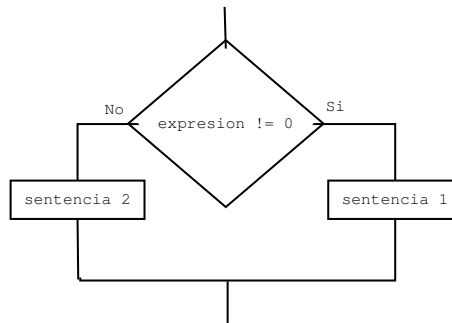


Figura 3.1: Sentencia if

- Si la **expresion** es verdadera (valor distinto de 0), entonces se ejecuta **sentencia**.
- La **expresion** debe estar entre paréntesis.
- Si **sentencia** es compuesta tenemos:

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

Un ejemplo de uso de esta sentencia es el siguiente fragmento de programa, que decide si un número es par:

```
int numero = 0, esPar= 0;
if ((numero % 2) == 0)
    esPar = 1;
```

### 3.5.2. Sentencia if-else

La forma general de esta sentencia es:

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

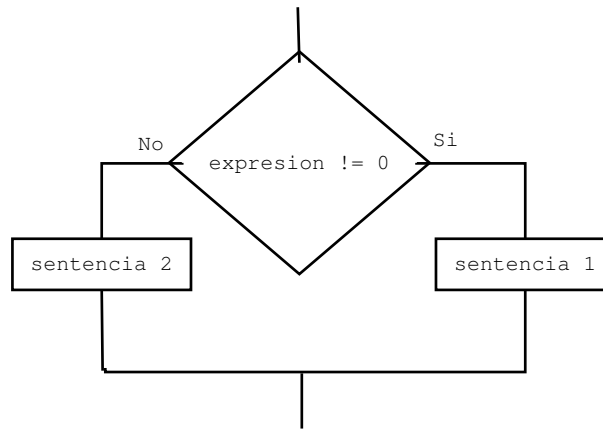


Figura 3.2: Sentencia if-else

- Si **expresion** es verdadera (valor distinto de 0), entonces se ejecuta **sentencia 1**; en caso contrario, se ejecuta **sentencia 2**.
- Si las sentencias son compuestas se cierran entre { }.
- Las sentencias pueden ser a su vez sentencias **if-else**.

```

if (expresion 1)
    if (expresion 2)
        S1
    else
        S2
else
    S3

```

Un ejemplo de uso de esta sentencia es el siguiente fragmento de programa, que elige el menor de tres números:

```

float a, b, c, menor;
a=2; b=4; c=1;

if (a < b) {
    if (a < c)
        menor = a;
    else
        menor = c;
} else {
    if (b < c)
        menor = b;
    else
        menor = c;
}

```

### 3.5.3. Sentencia switch

La forma general de esta sentencia es:

```

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

```

```

case exp 2:
case exp N:
    sentencia N;
    break;
default:
    sentencia D;
}

```

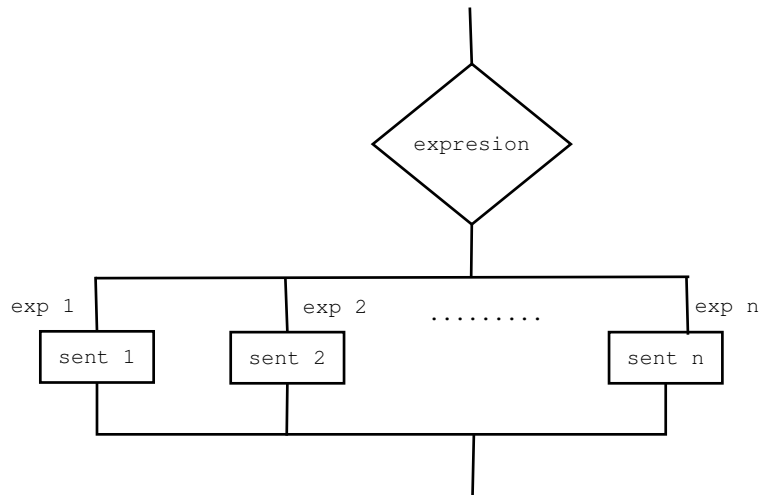


Figura 3.3: Sentencia switch

- **expresion** devuelve un valor entero, pero también puede ser de tipo char.
- **exp1, ..., exp N** representan expresiones constantes de valores enteros, aunque también pueden ser caracteres.

Un ejemplo de uso de esta sentencia es el siguiente fragmento de programa, que decide si imprime la vocal dada:

```

letra='e';
switch(letra);
{
    case 'a':
    case 'A':
        printf(`Es la vocal a\n`);
        break;
    case 'e':
    case 'E':
        printf(`Es la vocal e\n`);
        break;
    case 'i':
    case 'I':
        printf(`Es la vocal i\n`);
        break;
    case 'o':
    case 'O':
        printf(`Es la vocal o\n`);
        break;
    case 'u':
    case 'U':
        printf(`Es la vocal u\n`);
        break;
    default: printf(`Es una consonante\n`);
}

```

### 3.5.4. Sentencia break

La sentencia **break** 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.

### 3.5.5. Sentencia for

La forma general de esta sentencia es:

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

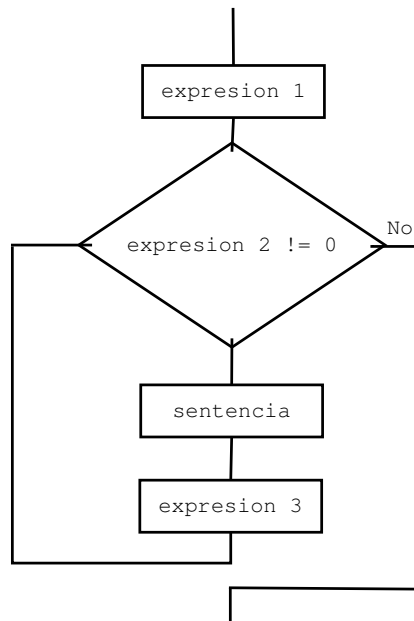


Figura 3.4: Sentencia for

- Inicialmente se ejecuta **expresion 1**, se hace para inicializar algún parámetro que controla la repetición del bucle.
- **expresion 2** es una condición que debe ser cierta para que se ejecute **sentencia**.
- **expresion 3** se utiliza para modificar el valor del parámetro.
- El bucle se repite mientras **expresion 2** sea cierto.
- Si **sentencia** es compuesta se encierra entre { }.
- Si se omite **expresion 2** se asumirá el valor permanente de 1 y el bucle se ejecutará de forma indefinida (bucle infinito).

Un ejemplo de uso de esta sentencia es el siguiente fragmento de programa, que calcula la suma de los números del 1 al 100:

```
int numero, suma;

suma=0;
for (numero=1; numero<=100; numero++)
    suma = suma + numero;
```

### 3.5.6. Sentencia while

La forma general de esta sentencia es:

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

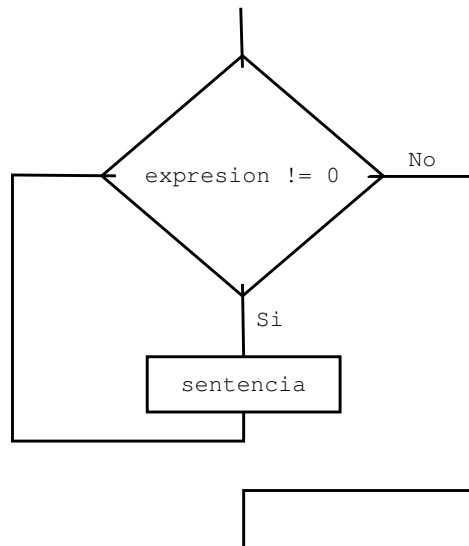


Figura 3.5: Sentencia while

- **sentencia** se ejecutará mientras el valor de **expresion** sea verdadero.
- 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.
- Si **sentencia** es compuesta se encierra entre { }.

Un ejemplo de uso de esta sentencia es el siguiente fragmento de programa, que calcula la suma de los numeros del 1 al 100:

```

int suma, limite;

suma=1; limite=100;
while(limite>0)
{
    suma=suma+limite;
    limite--;
}
  
```

### 3.5.7. Sentencia do-while

La forma general de esta sentencia es:

```

do
    sentencia;
while (expresion);
  
```

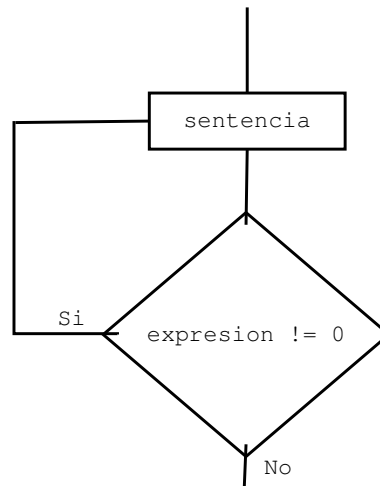


Figura 3.6: Sentencia do-while

- **sentencia** se ejecutará mientras el valor de **expresion** sea verdadero.
- **sentencia** siempre se ejecuta al menos una vez.
- Si **sentencia** es compuesta se encierra entre { }.



**Nota:** Lo normal es que **sentencia** incluya algún elemento que altere el valor de condición de salida del bucle.

Para la mayoría de las aplicaciones es mejor y más natural comprobar la condición antes de ejecutar el bucle, por ello se usa más la sentencia **while**.

Un ejemplo de uso de esta sentencia es el siguiente fragmento de programa, que pide un número igual a 0:

```

int numero = 0;
do
{
    printf("Introduce el número 0:\n");
    scanf("%d", &numero); /* Se lee el numero */
} while (numero != 0);
  
```

## 3.6. Funciones y subrutinas

Las funciones en C desempeñan el papel de las subrutinas o procedimientos en otros lenguajes, esto es, permiten agrupar una serie de operaciones de tal manera que se puedan utilizar más tarde sin tener que preocuparnos por cómo están implementadas, simplemente sabiendo lo que harán.

El uso de funciones es una práctica común y recomendable ya que permite modularizar nuestro código, simplificando así el desarrollo y la depuración del mismo. Para utilizar funciones en un programa es necesario declararlas previamente al igual que las variables (en este caso indicaremos los argumentos de entrada y su tipo, y el tipo del valor que devolverá) y definir las operaciones que contiene.

En C la declaración de una función tiene la siguiente estructura:

```
tipo_devuelto nombre_funcion (argumentos);
```

Y su definición:

```
tipo_devuelto nombre_funcion (argumentos)
{
    sentencias;
}
```



**Nota:** A la hora de definir una función que **no** acepte argumentos escribiremos `void` en vez de los argumentos.

Pongamos un ejemplo, un programa en el que queremos incluir una función que devuelva el factorial de un número:



### Ejemplo

```
1  #include <stdio.h>
2
3  int factorial(int a);
4
5  int main()
6  {
7      int f;
8
9      f=factorial(5);
10     printf("El factorial de 5 es: %d\n",f);
11     return 0;
12 }
13
14 int factorial (int a)
15 {
16     int i = 0, res = 1;
17
18     for (i=a; i>1; i--)
19         res=res*i;
20
21     return res;
22 }
```

La declaración

```
int factorial(int a);
```

debe coincidir con la definición de la función factorial que aparece posteriormente, si no coincide obtendremos un error en la compilación del programa. El valor que calcula la función `factorial()` se devuelve por medio de la sentencia `return`, ésta puede estar seguida de cualquier expresión o aparecer sola. En tal caso la función no devuelve ningún valor y al llegar a ese punto simplemente se devuelve el control a la función desde la que se invocó.

### 3.6.1. Paso de parámetros a funciones. Llamadas por valor

Es importante destacar que en C todos los argumentos de una función se pasan por valor. Esto es, las funciones trabajan sobre copias privadas y temporales de las variables que se le han pasado como argumentos, y no directamente sobre ellas. Lo que significa que no podemos modificar directamente desde una función las variables de la función que la ha llamado.

Veamos esto con un ejemplo:



#### Ejemplo

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int a=5;
6      int b;
7
8      int factorial(int n)
9          {
10         int res=1;
11
12         for (;n>1;n--)
13             {
14                 res=res*n;
15             }
16
17         return res;
18     }
19
20     b=factorial(a);
21
22     printf("a=%d\n",a);
23     printf("b=%d\n",b);
24
25     return 0;
26 }
```

Como podemos ver, en este caso no utilizamos una variable temporal en la función factorial para ir calculando la solución, sino que vamos disminuyendo el argumento `n` de entrada. Esto no influye en la variable `a` (que es la que se paso como argumento a la función factorial) ya que al pasarse los parámetros por valor es una copia de la variable `a` y no a directamente la que maneja la función factorial como argumento `n`.

Si quisieramos modificar una variable llamando a una función tendríamos que pasarle como argumento a dicha función la dirección en memoria de esa variable (un puntero a la variable). Esto lo veremos en la sección 5.9.

## 3.7. Tipos de datos compuestos

En esta subsección vamos a ver como crear y utilizar tipos compuestos. Un tipo compuesto no es más que un tipo de datos que es capaz de almacenar una información.

### 3.7.1. Arrays

Los *arrays*<sup>3</sup> quizás sean la forma más simple de tipos de datos compuestos.

**DEFINICIÓN: Array:** *Un array es una colección **ordenada** de elementos de un mismo tipo de datos, agrupados de forma consecutiva en memoria. Cada elemento del array tiene asociado un **índice**, que no es más que un número natural que lo identifica inequívocamente y permite al programador acceder a él.*

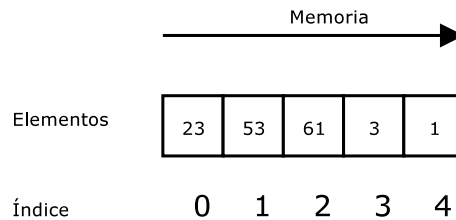


Figura 3.7: Un array de números enteros

La figura 3.7.1 muestra gráficamente un array de números enteros. Óbserve que el índice del primer elemento del *array* es el **número cero**.



**Consejo:** Piensa en los índices como desplazamientos desde el inicio del *array*. Así pues el primer elemento tendrá un índice (desplazamiento) **cero**, el segundo un índice (desplazamiento) **uno**, etc.

#### Definición y declaración de arrays en C

La sintaxis para declarar un array es simple: `tipo nombre[tamaño];` siendo:

- `tipo`: El tipo de los elementos que compondrán el *array*
- `nombre`: El nombre de nuestro *array*
- `tamaño`: Un número entero mayor que cero que indica el tamaño del array

Veamos algunos ejemplos:

```
int array1[100]; /* Declara un array de 100 elementos enteros */
char array2[15]; /* Declara un array de 15 caracteres */
```

#### Acceso a elementos de un array

Los *arrays* no nos servirían de nada si no pudiéramos manipular la información que contienen. Para ello se utiliza el operador de acceso a *arrays* `[]`. Este operador accede al elemento del array indicado entre los dos corchetes, como se puede observar en los siguientes ejemplos:

```
/* Declaramos un array de 10 elementos */
int array[10];
int tmp;

/* Asignamos al tercer elemento (índice 2) el valor 23 */
array[2] = 23;

/* Guardamos en tmp el valor del tercer elemento */
tmp = array[2];

/* El quinto elemento (índice 4) tiene el mismo valor */
array[4] = array[2];
```

<sup>3</sup>En español la palabra *array* no tiene una traducción clara. Algunos autores utilizan *agrupación*, aunque en este manual utilizaremos simplemente *array*

Como hemos visto, se puede utilizar el operador [ ] tanto para acceder a elementos de un array como para asignarles algún valor.

Llegados a este punto puedes estar preguntándote: ¿qué valor tienen los elementos de un array recién declarado? La respuesta es tan simple como decepcionante: **no lo sabes**. Los elementos de un *array* recién declarado tomarán valores no determinados.



**Consejo:** Siempre que declares un array asegúrate que inicializas los elementos a un valor conocido antes de operar con ellos.

### 3.7.2. Estructuras

**DEFINICIÓN: Estructura:** Una estructura (*struct*) es un tipo de datos compuesto que agrupa un conjunto de tipos de datos (no necesariamente iguales) en un único tipo.

La anterior definición, que en un principio se nos puede antojar oscura, oculta un concepto realmente simple, que vamos a explicar mediante el siguiente ejemplo: supongamos que estamos escribiendo un *software* de gestión de miembros de asociaciones universitarias. Obviamente necesitaremos almacenar y manipular datos sobre personas, tales como nombre, DNI, edad, etc. Podríamos aproximarnos al problema declarando estas variables por separado en nuestro programa:

```
/* Edad de la persona */
int edad;
```

```
/* DNI */
char DNI[20];
```

```
/* Nombre de la persona */
char nombre[50];
```

Sin embargo hay algo que no “encaja” demasiado bien dentro de esta aproximación. Esta claro que, aunque tengamos tres variables independientes, las tres se refieren a distintos aspectos de información acerca de una misma persona, lo cual no queda reflejado en el código del programa, pudiendo llevar a errores o malentendidos<sup>4</sup>. Adoptamos los siguientes cambios:

```
/* Edad de la persona */
int persona_edad;
```

```
/* DNI */
char persona_DNI[20];
```

```
/* Nombre de la persona */
char persona_nombre[50];
```

Claro está, los cambios realizados no influyen para nada en el funcionamiento del programa. Sin embargo mejoran la legibilidad del código, algo **muy importante**, sobre todo cuando estamos tratando con programas de más de unas pocas líneas de código. Aún así no estamos totalmente convencidos del resultado; esta claro que ahora queda patente que las tres variables se refieren a distintos aspectos de una misma unidad de información. Sin embargo todavía sigue siendo “engorroso” manipular los datos de una persona, sobre todo cuando hay que pasar dichos datos a una función, ya que tendremos que pasar tres nombres de variable a la función, aún tratándose de una misma persona. Nos gustaría “agrupar” estos datos, de igual forma que agrupamos todos los apuntes de una asignatura en una misma carpeta. Las estructuras nos permiten hacer precisamente esto.

#### Definición de una estructura en un programa C

La sintaxis para definir una estructura es la siguiente:

```
struct nombreEstructura {
    declaración de variable1;
    declaración de variable2;
    .
    .
    declaración de variableN;
};
```

Dicho de forma informal lo que estamos haciendo es englobar un conjunto de variables dentro de un nuevo entorno.

<sup>4</sup>Recordemos que la mayoría de los programas que escribamos serán leídos por al menos otra persona, la cual no tiene por qué estar familiarizada con nuestro estilo de programación

### Declaración de variables de esa estructura

Una vez que tengamos definida la estructura podremos declarar variables de ese tipo, de la siguiente forma:

```
struct nombreEstructura nombreVariable;
```

A partir de este momento la variable `nombreVariable` contendrá información acerca de cada uno de los campos definidos en la estructura.

### Accediendo a los campos de una estructura

Supongamos que tenemos declarada la variable `var_struct`, perteneciente al tipo de la estructura `struct1`, la cual tiene declarada los campos `campo1` (entero) y `campo2` (real). Para acceder a estos campos utilizamos el operador `.` (punto), de la siguiente forma:

```
var_struct.campo1 = 10;
var_struct.campo2 = var_struct.campo1 * 3.1415;
```

Al igual que en el caso de los *arrays*, este operador se puede utilizar tanto para acceder a la información de los campos como para modificarlos.

### Ejemplo

Retomando el ejemplo de nuestro gestor de miembros de asociaciones universitarias, podemos reescribir el código de la siguiente forma:

```
struct Persona {
    /* Edad de la persona */
    int edad;

    /* DNI */
    char DNI[20];

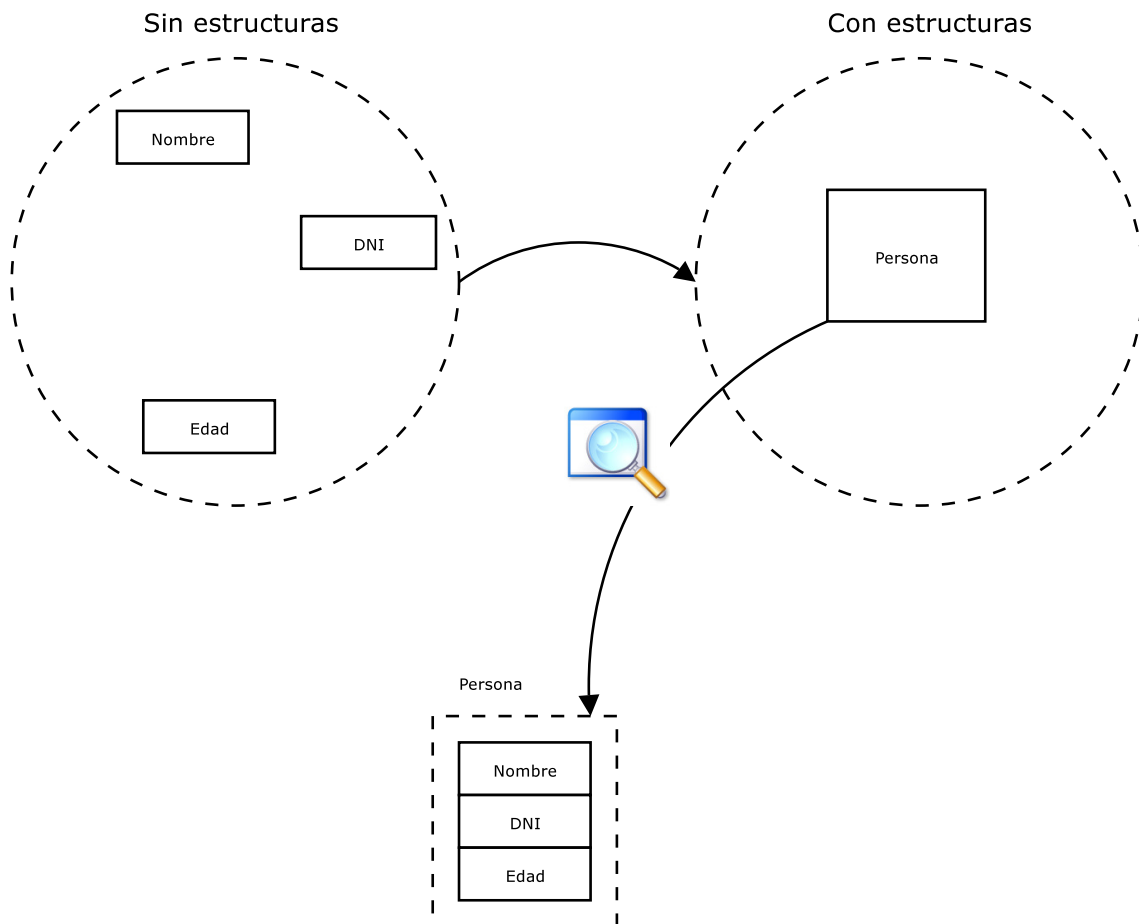
    /* Nombre */
    char nombre[20];
};

/* Declaramos dos variables del tipo Persona */
struct Persona miembro_acm1;
struct Persona miembro_acm2;

/* Asignamos algunos valores */
miembro_acm1.edad = 21;
miembro_acm2.edad = 23;

/* Los demás campos quedan indefinidos!! */
```

Gráficamente lo que hemos hecho es lo siguiente:



### 3.7.3. Uniones

Las uniones (*unions*) tienen un cometido parecido a las estructuras, esto es, agrupar en una sola variable varios valores. Sin embargo, al contrario que las estructuras, las uniones sólo pueden guardar un valor a la vez. Esto significa que si asignamos valor a uno de los componentes de una variable unión, los demás componentes dejarán de tener un valor asignado.

**DEFINICIÓN: Unión:** *Una unión es un tipo de datos compuesto que puede almacenar valores de diferentes tipos, aunque no a la vez.*



**Consejo:** Puedes pensar en las uniones como tipos “mutantes”, es decir, como variables que pueden almacenar valores de distintos tipos. Al igual que pasa con las variables ordinarias el hecho de asignar a una unión un valor destruye automáticamente los valores anteriores (sean del tipo que sean). Al fin y al cabo, lo que el compilador hace es reservar espacio para una variable de tamaño el elemento de mayor tamaño de la unión.

#### Definición de una unión en un programa C

La definición de una unión es muy similar a la de una estructura:

```
union nombreUnion {
    declaración de variable1;
    declaración de variable2;
    .
    .
    declaración de variableN;
};
```

#### Declaración de variables tipo union

```
union nombreUnion variableUnion1;
```

Al igual que en el caso de las estructuras, para acceder a un componente de una unión se utiliza el operador . (punto).

**Ejemplo**

```
union Salario {
    int enPesetas;
    float enEuros;
};

union Salario SalarioJuan;

/* En principio Juan cobra en pesetas */
SalarioJuan.enPesetas = 250000; /* ahora SalarioJuan.enEuros no tiene valor */
.
.
/* Pero ahora vamos a reconvertir el salario de Juan a euros */
SalarioJuan.enEuros = SalarioJuan.enPesetas / 166.386;

/* A partir de este momento SalarioJuan.enPesetas no tiene valor válido */
```



# Capítulo 4

## El modelo de compilación de C

### 4.1. Introducción

En este capítulo vamos a ver cómo es el modelo de compilación en C, es decir, por qué fases pasa nuestro código desde que lo editamos hasta que obtenemos un fichero ejecutable. En principio uno podría pensar que conocer lo que ocurre “por debajo” no es necesario a la hora de programar C, y de hecho no estaría muy equivocado. Sin embargo hay ciertos procesos que ocurren cuando compilamos nuestro código C que como programadores debemos conocer.

El proceso de compilación de código C consta de tres fases principales:

1. **Preprocesado**
2. **Compilado**
3. **Enlazado**

Primera sorpresa: La compilación es solo una fase del proceso de compilación. Esta aparente contradicción se explica fácilmente: cuando hablamos del proceso de compilación nos estamos refiriendo al proceso mediante el cual transformamos nuestro código fuente en un fichero ejecutable. Sin embargo cuando nos referimos a la fase de compilación nos estamos refiriendo a la parte del proceso que se encarga de traducir código fuente en instrucciones en formato binario.

### 4.2. El preprocesador

El preprocesador es algo característico de C/C++, que no se suele encontrar en otros lenguajes de programación. El preprocesador actúa sobre el programa fuente, antes de que empiece la compilación propiamente dicha, para realizar ciertas operaciones.

Una de estas operaciones es, por ejemplo, la sustitución de constantes simbólicas, como vimos en el capítulo de constantes. También permite definir macros, que son parecidas a funciones, *pero no iguales*, y realizar compilación condicional. En general se encarga de modificar el código fuente, según una serie de directivas. Estas se reconocen puesto que empiezan por # y no tienen que terminar en ;, a diferencia de las instrucciones de C.

#### 4.2.1. #include: inclusión de otros ficheros

Cuando en un archivo de código fuente se encuentra una línea con un `#include` seguido de un nombre de archivo, el preprocesador *incluye* el archivo en el punto donde se encuentra la directiva, exactamente igual que si hiciéramos copiar/pegar. La sintaxis de este comando es la siguiente:

```
#include "nombre_del_archivo"  
  
o  
  
#include <nombre_del_archivo>
```

La diferencia entre la primera forma (con comillas "...") y la segunda forma (con los símbolos <...>) está en el directorio de búsqueda de los archivos a incluir. En la forma con comillas se busca el archivo en el directorio actual y posteriormente en el (o los) directorio(s) estándar de librerías<sup>1</sup>. En la forma que utiliza los símbolos <...> se busca directamente en el directorio estándar de librerías, sin buscar en el directorio actual.

<sup>1</sup>los directorios de librerías dependen del sistema operativo y compilador que se use



**Consejo:** Los archivos de la librería estándar (`stdio.h`, `math.h`, etc...) se suelen incluir con `<...>` mientras que los archivos hechos por el propio programador se ponen entre comillas. Es una cuestión de costumbre más que otra cosa.

Los archivos incluidos pueden contener a su vez directivas `#include`, esto se conoce como inclusión anidada. El número de niveles de anidamiento depende del compilador, pero ha de ser al menos 8 en C89 y 15 en C99 (en cualquier caso es más que de sobra).

Este comando del preprocesador se utiliza normalmente para incluir las cabeceras (*headers* en inglés). Estos son archivos con los prototipos de las funciones de librería o con las funciones definidas para el programa en cuestión<sup>2</sup>.

#### 4.2.2. #define: creación de macros

Como vimos, con la directiva `#define` se define un *alias*, es decir una sustitución de texto. Esto, que usábamos para definir constantes, se puede utilizar de la misma manera para definir macros. En efecto, podemos poner parámetros a esas sustituciones, que se comportan entonces como si de una pseudo-función se tratara.

En una macro con argumentos, los argumentos se sustituyen en el texto de reemplazo, y a continuación la macro se *expande*, es decir, en el programa el texto de reemplazo reemplaza al identificador y a la lista de argumentos.

Veamos por ejemplo:

```
#define PI 3.14
#define AREA_CIRCULO(x) PI * (x) * (x)
```

Ahora podemos usar la macro como si fuera función normal:

```
void main() {
    int a;
    a = AREA_CIRCULO(3);
}
```

Durante la compilación la macro se expande a:

```
a = 3.14 * (3) * (3)
```

y obtenemos el resultado esperado.

Las macros nos permiten insertar código en el programa directamente, evitando la sobrecarga de invocar a una función (pasar parámetros a la pila, realizar un salto, recibir parámetros ...) <sup>3</sup> pero conservando la legibilidad del programa. Por otra parte permite realizar cálculos durante la compilación, en lugar de realizarlos durante la ejecución. Así en el ejemplo que nos ocupa el compilador le da directamente el valor adecuado a la variable “a”, en lugar de insertar instrucciones para que se evalúe cada vez que se use.

Es importante no olvidar los PARÉNTESIS alrededor de los parámetros en la definición de la macro, de lo contrario la expansión de parámetros puede ser incorrecta, por ejemplo:

```
#define AREA_CIRCULO(x) PI * x * x
void main() {
    int a,b;
    a = AREA_CIRCULO(c + 3);
}
```

expande a:

```
a = 3.14 * c + 3 * c + 3
```

que, por culpa de la precedencia de operadores, es equivalente a

```
a = (3.14 * c) + (3 * c) + 3
```

en lugar de expandir a:

```
a = 3.14 * (c + 3) * (c + 3)
```

que es lo que queríamos.

<sup>2</sup>las cabeceras son el equivalente en C/C++ de los archivos “.ads” de ADA

<sup>3</sup>En C++ se puede hacer lo mismo usando la directiva *inline*.

### 4.2.3. Compilación condicional

Hay varias directivas que permiten compilar selectivamente partes del código fuente del programa. Este proceso se llama *compilación condicional* y se utiliza mucho cuando se quiere mantener versiones diferentes de un programa. Por ejemplo se puede mantener una versión demo o recortada, gratuita y otra mas potente de pago, y las partes de código que difieran entre las dos se compilan según sea la plataforma de destino.

#### uso de #if, #else, #elif y #endif

Estas directivas permiten incluir parte del código según el valor que tome una constante de preprocesador. Veamos un ejemplo:

```
#define MAX 10

int main(void){
#if MAX > 99
    printf("versión PRO, compilada para arrays mayores de 99.\n");
    ...
#else
    printf("versión DEMO, compilada para arrays menores de 99.\n");
    ...
#endif
    return 0;
}
```

Aquí, al ser MAX menor de 99 el bloque que sigue al #if no se compila, en su lugar se compila la alternativa del bloque #else.

Se pueden hacer selecciones múltiples mediante #elif que equivale a “else if”

#### uso de #ifdef, #ifndef y #undef

Estas directivas permiten compilación condicional basándose en si esta definida una macro o no, independientemente del valor que tenga. Se usan a menudo para tener una versión de prueba con chequeos adicionales (de rangos, de corrección de parámetros...), y otra final, sin ellos (y por tanto más rápida).



**Nota:** No es mala idea definir una macro DEBUG, por ejemplo, que permita alternar entre compilación con chequeos extra y versión final

Con \#undef podemos “desdefinir” una definición previamente realizada.

## 4.3. Compilación

La compilación es la fase más costosa del proceso de compilación y esto es debido a que el compilador debe analizar el texto de nuestro programa fuente, comprobar que no contiene errores y producir como salida un fichero con la traducción de nuestro código a conjunto de instrucciones de nuestro procesador.

Ningún proyecto de programación serio está compuesto hoy en día por un solo archivo fuente, sino más bien todo lo contrario. Es conveniente que cualquier programa que pase de unos cientos de líneas sea dividido en una serie de módulos que faciliten la legibilidad y el mantenimiento del código.

A la hora de compilar nuestro proyecto lo que haremos será *procesar* cada uno de estos módulos por separado, diciéndole al compilador que tenga en cuenta que ninguno de estos módulos es un programa por si mismo, sino una parte del mismo. Lo que hará el compilador será producir como salida una serie de **ficheros objeto**<sup>4</sup>. Estos ficheros son la traducción a binario de cada uno de nuestros módulos. Sin embargo ninguno de ellos conforma un ejecutable por sí mismo, ya que ninguno contiene el código completo de nuestro programa.

<sup>4</sup>La extensión habitual de estos ficheros es .o (Unix/Linux) o .obj (Windows)

## 4.4. Enlazado

La fase de enlazado consiste simplemente en “reunir” cada uno de los ficheros objeto producidos en la fase anterior, resultando de este proceso un fichero ejecutable. Si nuestro programa hace uso de librerías externas (la mayoría lo hacen), el código de las funciones utilizadas será añadido también al fichero ejecutable<sup>5</sup>.

## 4.5. Un ejemplo sencillo

A continuación vamos a ver de qué nos sirven todos estos conceptos con un ejemplo sencillo.



**Nota:** Como entorno de programación vamos a utilizar el compilador GCC en un PC normal con Linux, aunque todo lo que veamos será fácilmente extrapolable a cualquier otro entorno.

Queremos compilar el siguiente programa, que está compuesto de dos módulos:

principal.c

```

1  #include <stdio.h>
2
3  /* Prototipo */
4  int obten_numero( void );
5
6  /* Código */
7  int main(void) {
8      int n;
9
10     n = obten_numero();
11
12     printf( "Has introducido el %d\n", n );
13 }
```

entrada\_teclado.c

```

1  #include <stdio.h>
2
3  int obten_numero(void) {
4      int n;
5
6      printf( "Introduce un número: " );
7      scanf( "%d", &n );
8
9      return n;
10 }
```

Lo primero que vamos a hacer es compilar los dos módulos, por separado:

```

$ gcc -c principal.c
$ gcc -c entrada_teclado.c
$
```



**Nota:** El flag `-c` le dice al compilador que **no queremos** enlazar el código proporcionado, sólo preprocesarlo y compilarlo.

Una vez compilados los módulos sólo nos queda enlazarlos:

```
$ gcc -o programa.exe principal.o entrada_teclado.o
```

De esta forma hemos generado un fichero ejecutable (`programa.exe` en este caso) a partir de los dos módulos.

<sup>5</sup>Realmente esta forma de enlazado con librerías es solo una modalidad de las dos existentes, llamada enlazado estático. Si en vez de añadir el código de la función de librería utilizada se deja una referencia a esta (el equivalente a decir “Esta función no está aquí, la puedes encontrar en la librería X”), estamos hablando de enlazado *dinámico*.



# Capítulo 5

## Punteros

### 5.1. Introducción

#### 5.1.1. ¿Qué es un puntero?

Podríamos imaginar un puntero como una *flecha* que apunta a otra variable, pero esto es sólo para aclararnos a la hora de hacer esquemas. En realidad, dentro del ordenador, una variable puntero no es más que una dirección de memoria, un puntero nos dice **dónde** se encuentra almacenado el valor de la variable a la que apunta. Si nos quedamos con esta última definición (la de dirección de memoria), nos será más fácil aclararnos cuando trabajemos con punteros a gran escala.

Como ya sabemos, al declarar una variable lo que estamos haciendo es *reservar* espacio en memoria para esa variable (el tamaño del espacio reservado dependerá del *tipo* de la variable). Pues bien, al declarar un puntero lo que hacemos es reservar espacio para almacenar *una dirección de memoria*, no el contenido de la variable.

La siguiente figura muestra tres variables, de tipos `char`, `int` y `double`, y un puntero a cada una de ellas:

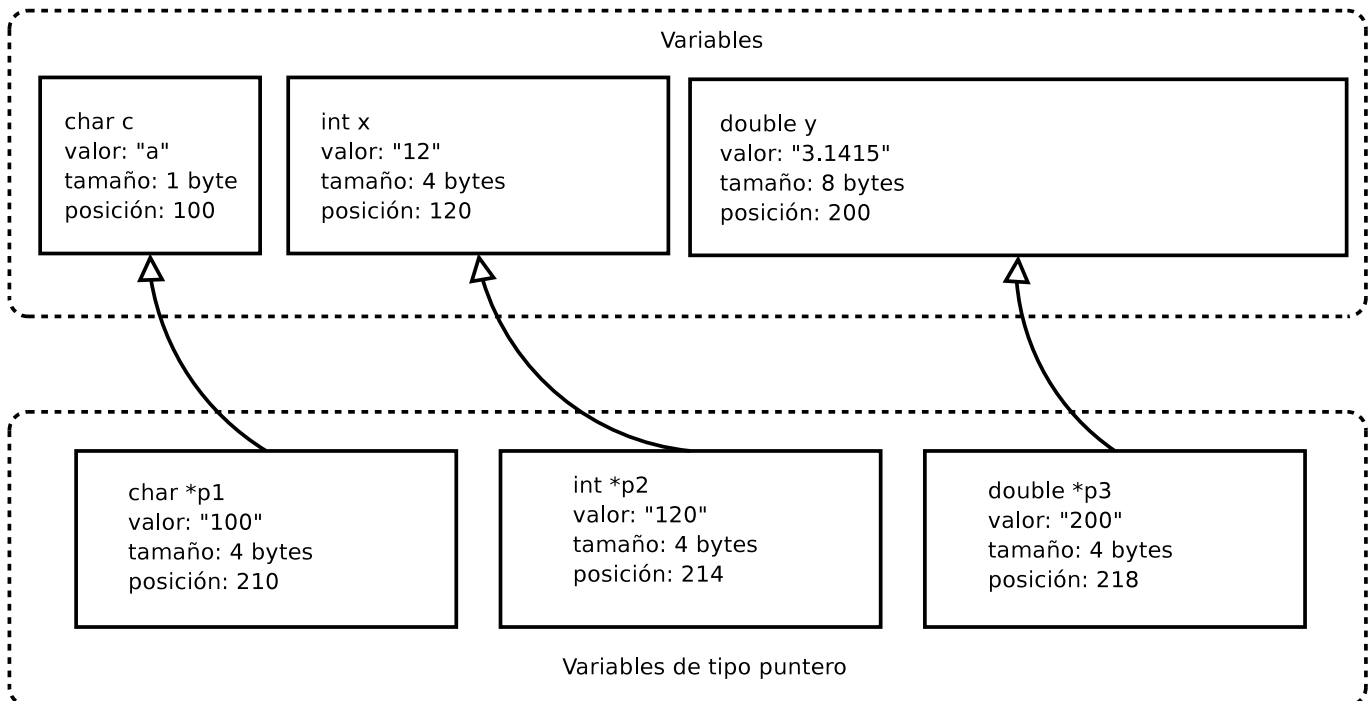


Figura 5.1: Punteros a variables `char`, `int` y `double`

Tomemos como ejemplo la variable de tipo `char`. Está almacenada en la dirección de memoria (ficticia) 100. El contenido de dicha posición (esto es, el contenido de la variable) es el carácter "a". Por otro lado, el puntero `p1` se encuentra en la dirección 210, y su valor es 100. Como vemos, el contenido de la variable `p1` (puntero) es la dirección de la variable `char`.

Como se ve en la figura, mientras que el tamaño que ocupan en memoria las variables es distinto (un `char` ocupa un byte, un `double` ocupa 8 bytes), el tamaño que ocupan las variables tipo puntero es siempre el mismo: 4 bytes. Esto es lógico: las direcciones de memoria siempre ocupan el mismo tamaño.<sup>1</sup>

### 5.1.2. ¿Para qué necesito yo un puntero?

Es posible que al abordar un concepto algo más abstracto que los que surgen al aprender un lenguaje de programación, sea difícil imaginar para qué sirve. Al plantearnos esa pregunta, surgen dos respuestas inmediatas:

- para poder usar “memoria dinámica”.
- para poder usar “argumentos por referencia”.

Ambos conceptos van a ser tratados con más profundidad en las secciones 5.6 y 5.9.

## 5.2. Sintaxis de punteros

### 5.2.1. Declaración de punteros

Ante todo, *un puntero es una variable*. Al dedicar un interés especial en los punteros, puede dar la impresión de que son elementos separados de las variables. No es así. Son variables, y como veremos, podemos asignarles nuevos valores, e incluso realizar algunas operaciones aritméticas útiles con ellos.

*Una variable de tipo puntero está íntimamente ligada con el tipo al que apunta*. Por ello, en la declaración, escribimos el nombre del tipo al que apuntará nuestro puntero, seguido de asterisco, y por último, el nombre del puntero. Ejemplo:

```
int *p_entero;
float *p_real;
char *caracter;
```

Serán declaraciones de punteros que contendrán la dirección de memoria de un entero, un real y un carácter<sup>2</sup> respectivamente.

### 5.2.2. Punteros genéricos

Un caso especial, es cuando queremos declarar un puntero “genérico”, esto es, queremos que el puntero no esté ligado a ningún tipo concreto. ¿Qué sentido tiene esto?, como se ha visto previamente, en C podemos realizar un *cast* (ver 3.4.5) sobre una variable, y forzar su conversión a otro tipo. De esta manera, por ejemplo, podríamos declarar un puntero genérico, apuntar a una zona de memoria, y realizar un cast sobre el puntero genérico.



**Nota:** La única diferencia existente entre un puntero genérico, y un puntero ligado a un tipo de datos concreto, se produce en las operaciones de aritmética de punteros (ver 5.4).

Para declarar un puntero de tipo genérico, utilizamos:

```
void * pointer;
```

Como veremos, muchas funciones de biblioteca de POSIX manejan punteros genéricos en sus argumentos.

### 5.2.3. Los operadores de contenido “\*” y de indirección “&”

El operador de indirección `&` se utiliza para referirse a la dirección de una variable, así el código siguiente:

```
/* Reservamos 4 bytes para almacenar una dirección de memoria */
int *p_entero;

/* Reservamos 4 bytes para almacenar un entero */
int entero;

/* Escribimos en la variable p_entero la dirección de entero */
p_entero = &entero;
```

<sup>1</sup>Es importante recordar que los tamaños de las variables `char`, `int` y `double` aquí especificados son dependientes de la plataforma.

<sup>2</sup>Como veremos, la declaración `char *` es usada también para la declaración de cadenas de caracteres

Es totalmente correcto ya que, aunque la variable `entero` no tenga ningún valor asignado todavía, lo que estamos haciendo es escribir en la variable `p_entero` la dirección de memoria donde se almacena la variable `entero`.

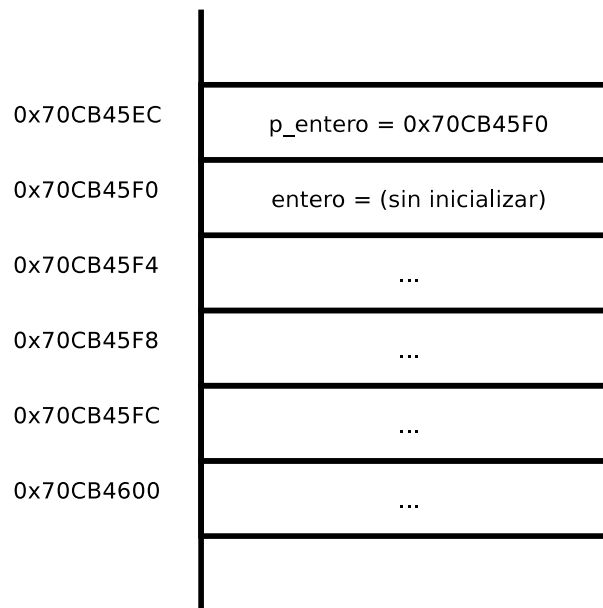


Figura 5.2: Inicializando punteros

El operador `*` se utilizar para manejar la dirección a la que apunta un puntero. Podríamos llamarlo el operador de *acceso*. Tanto el operador de acceso, como el de indirección, funcionan en **notación prefija**. Veamos un ejemplo que combina ambos operadores:

```
1 int *p_entero;
2 int entero1, entero2;
3 entero1 = 2;
4 entero2 = 5;
5 p_entero = &entero1;
6 entero1 = *p_entero + entero2;
```

Del mismo modo podemos asignar un valor a la variable a la que apunta el puntero, de la forma:

```
1 int *p_entero;
2 int entero1, entero2;
3 entero1 = 2;
4 entero2 = 5;
5 p_entero = &entero1;
6 *p_entero = entero1 + entero2;
```

Este último código ejecutaría **exactamente** lo mismo que el anterior. Debemos notar que no tendría sentido prescindir de la línea 5, puesto que estaríamos intentando introducir el valor `entero1 + entero2` en una dirección de memoria que no conocemos, puesto que no le habríamos dado valor a `p_entero` (ver 5.10.2).

### 5.3. Strings

Los arrays de caracteres se llaman **strings**. Funcionan exactamente igual que cualquier otro array. En los **strings** no hace falta una variable que guarde el tamaño del array, puesto que se marca el final del array con el carácter “\0”. La ventaja de los **strings** es que podemos rellenar y consultar varios elementos del array a la vez. En lugar de:

```
cadena[0] = 'h';
cadena[1] = 'o';
cadena[2] = 'l';
cadena[3] = 'a';
cadena[4] = '\0';
```

Podremos hacer esto:

```
cadena = "hola";
/* cuando metemos un texto entre comillas dobles, al copiarlo al array
   el carácter '\0' se incluye solo. */
print ("%s", cadena);
```

Una serie de fallos comunes al trabajar con strings se exponen en 5.10.4.

Existen funciones específicas para trabajar con strings (`#include <string.h>`) para realizar ese tipo de tareas. Recomendamos al lector que consulte la documentación de funciones como *strlen*, *strcpy*, *strncpy*, *strdup*, *strcat*.

## 5.4. Aritmética de punteros

Es posible realizar operaciones aritméticas sobre las variables de tipo puntero para conseguir que apunten a una posición diferente. Por ejemplo:

```
char cadena[5];
char *puntero;

puntero = &cadena[0]; /* puntero apunta a cadena[0] */
*puntero = 'h';       /* cadena[0] = 'h' */
*(puntero+1) = 'o';   /* cadena[1] = 'o' */
*(puntero+2) = 'l';   /* cadena[2] = 'l' */
*(puntero+3) = 'a';   /* cadena[3] = 'a' */
*(puntero+4) = '\0';  /* cadena[4] = '\0' */
```

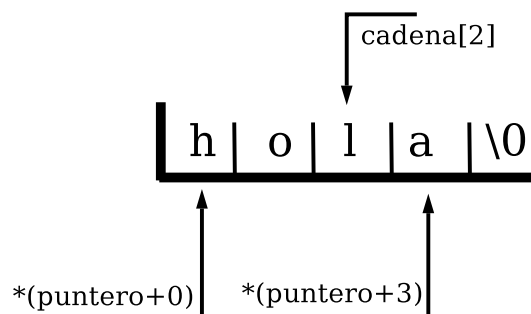


Figura 5.3: Aritmética sobre un puntero

### 5.4.1. Contexto

Se debe tener en cuenta que `puntero+x` apuntará a la dirección de `puntero` sumándole  $x$  veces el espacio ocupado por un elemento del tipo al que apunta, no el número de bytes.

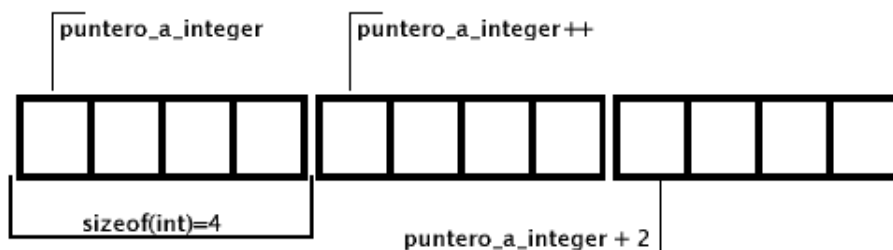


Figura 5.4: Suma sobre un puntero a integer

El programa mostrado a continuación nos muestra la diferencia entre considerar un puntero a integer y un puntero a char, en lo que se refiere a la suma:



### Ejemplo

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      int arraynum[2];
6      char arraychar[2];
7
8      int *pnum;
9      char *pchar;
10     unsigned int pnum1, pnum2, pchar1, pchar2;
11
12     /* Mostramos los tamaños de int y char */
13
14     printf("Tamaño de int: %d\n", sizeof(int));
15     printf("Tamaño de char: %d\n", sizeof(char));
16
17     /* Apuntamos con los punteros al principio de los arrays */
18
19     pnum = &arraynum[0];
20     pchar = &arraychar[0];
21
22     /* Calculamos el valor de los punteros a un elemento
23        y el siguiente */
24
25     pnum1 = (unsigned int) pnum;
26     pnum2 = (unsigned int) (pnum+1);
27
28     pchar1 = (unsigned int) pchar;
29     pchar2 = (unsigned int) (pchar+1);
30
31     /* Mostramos la resta entre un puntero al elemento n+1
32        y uno al elemento n */
33
34     printf("Distancia entre punteros sucesivos a int : %u\n", pnum2 - pnum1);
35     printf("Distancia entre punteros sucesivos a char: %u\n", pchar2 - pchar1);
36
37     return 0;
38 }

```

El resultado de ejecutar <sup>3</sup> el código anterior es:

```

Tamaño de int: 4
Tamaño de char: 1
Distancia entre punteros sucesivos a int : 4
Distancia entre punteros sucesivos a char: 1

```

Queda clara la importancia entre declarar un puntero de un tipo o otro. Ambos punteros del ejemplo ocupan lo mismo, ambos apuntan a direcciones de memoria del sistema, pero cuando el compilador tiene que generar código para realizar operaciones aritméticas, lo hace de manera distinta en función del tipo de puntero.

<sup>3</sup>Una vez más, recordamos que el tamaño de las variables en C es dependiente de la plataforma sobre la que compilamos/ejecutemos

### 5.4.2. Tipos de operaciones

Las operaciones soportadas sobre punteros son:

- Suma y resta de valores enteros (+, -, ++ y --)
- Comparación y relación (<, >, <=, >=, == y !=)
- Valor booleano (comparación con NULL)

### 5.4.3. Ejemplos de aritmética

A continuación mostramos un ejemplo de una función que recibe dos strings (ver 5.3), y copia uno sobre otro. En la segunda versión, las operaciones de aritmética de punteros se han agrupado, para escribir menos código. La finalidad de la segunda versión es acostumbrar al lector a la complejidad de algunas operaciones en C (función *copiar*).

Primera versión:



#### Ejemplo

```

1  #include <stdio.h>
2
3  void copiar (char* dest, const char* orig) {
4      if(orig != NULL && dest != NULL) {
5          while(*orig != '\0') {
6              *dest = *orig;
7              dest++;
8              orig++;
9          }
10         *dest = '\0'; /* terminamos la cadena */
11     }
12 }
13
14 int main(void)
15 {
16     char * origen = "Datos";
17     char destino[100]; /* usar "char * destino" sin inicializarlo
18                        * a una direccion de memoria valida sería
19                        * un error */
20
21     copiar(destino, origen);
22
23     printf("Origen: %s\n", origen);
24     printf("Destino: %s\n", destino);
25
26     return 0;
27 }
```

Segunda versión:



### Ejemplo

```

1  #include <stdio.h>
2
3  void copiar (char* dest, char* orig) {
4      if(orig && dest) /* por si acaso orig fuera NULL */
5          while(*dest++ = *orig++); /* waw */
6  }
7
8  int main(void)
9  {
10     char * origen = "Datos";
11     char destino[100]; /* usar "char * destino" sin inicializarlo
12                        * a una direccion de memoria valida sería
13                        * un error */
14
15     copiar(destino, origen);
16
17     printf("Origen: %s\n", origen);
18     printf("Destino: %s\n", destino);
19
20     return 0;
21 }

```

Comentaremos la siguiente sentencia:

```
while(*dest++ = *orig++);
```

Para entender la sentencia, la observaremos desde el punto de vista de la precedencia entre operadores. En primer lugar, se ejecutarían los post-incrementos, pero su efecto sólo tendría lugar al acabar la sentencia (la expresión parentizada). Por tanto, lo siguiente en ejecutarse sería el operador de acceso (“\*”). Eso accedería a los caracteres apuntados por `dest` y por `orig`. Después se ejecutaría la asignación, (copia de un carácter de la cadena). Como se vió en 3.4.5, la asignación “devuelve” el valor asignado, por lo que la expresión parentizada equivale al valor que se asigna. Cuando se asigna el último carácter de la cadena (`\0`), el valor de la expresión es falso (`\0` equivale a 0, esto es, falso), y el `while` saldría. Antes de terminar de procesar, se incrementarían ambos punteros (post-incrementos), haciendo que apunten al siguiente carácter de la cadena.



**Consejo:** Normalmente el uso de la aritmética de punteros se centra en operaciones sencillas de incremento o decremento. Operaciones más complejas son potencialmente peligrosas, además operaciones como la multiplicación o división de dos apuntadores no están permitidas lo cual es bastante lógico debido a la mínima utilidad práctica de estos operadores en punteros. A la hora de programar se debe recordar que un mal uso de la aritmética de punteros puede dejar poco portable nuestro código.

## 5.5. Estructuras y punteros

### 5.5.1. El operador “->”

En programas un poco más elaborados, usaremos a menudo los punteros para apuntar a estructuras (`struct`). Nada nos impide referirnos a los campos de la estructura a la que apunta el puntero de la forma que hemos visto hasta ahora, es decir, llamando al campo que deseamos del contenido del puntero dado. Con este ejemplo entenderemos mejor a qué nos referimos:

```

struct coordenada {
    int x, y;
} coord;
struct coordenada *p_coord;
p_coord = &coord;
(*p_coord).x = 4;
(*p_coord).y = 3;

```

Aun así esto puede resultar algo engorroso, por eso C nos da la posibilidad de usar el operador `->` que nos facilita las cosas:

```

struct coordenada {
    int x, y;
} coord;
struct coordenada *p_coord;
p_coord = &coord;
p_coord->x = 4;
p_coord->y = 3;

```

Los dos códigos anteriores ejecutan lo mismo pero en el segundo utilizamos el operador `->`, que nos hace el código más legible.



**Nota:** Es un error frecuente utilizar el operador punto sobre un puntero a estructura.

## 5.6. Memoria dinámica

### 5.6.1. ¿Qué es la memoria dinámica?

Supongamos que nuestro programa debe manipular estructuras de datos de longitud desconocida. Un ejemplo simple podría ser el de un programa que lee las líneas de un archivo y las ordena. Por tanto, deberemos leer un número indeterminado de líneas, y tras leer la última, ordenarlas. Una manera de manejar ese “número indeterminado”, sería declarar una constante `MAX_LINEAS`, darle un valor vergonzosamente grande, y declarar un array de tamaño `MAX_LINEAS`. Esto, obviamente, es muy ineficiente (y feo). Nuestro programa no sólo quedaría limitado por ese valor máximo, sino que además gastaría esa enorme cantidad de memoria para procesar hasta el más pequeño de los ficheros.

La solución consiste en utilizar memoria dinámica. La memoria dinámica es un espacio de almacenamiento que se solicita *en tiempo de ejecución*<sup>4</sup>. De esa manera, a medida que el proceso va necesitando espacio para más líneas, va solicitando más memoria al sistema operativo para guardarlas. El medio para manejar la memoria que otorga el sistema operativo, es el puntero, puesto que no podemos saber *en tiempo de compilación*<sup>5</sup> dónde nos dará huecos el sistema operativo (en la memoria de nuestro PC).

### 5.6.2. El mapa de memoria en Unix

Antes de profundizar en el manejo de memoria dinámica, vamos a dar una breve visión del mapa de memoria en Unix, esto es, cómo se organiza la memoria de un proceso. Ante todo, esto es una simplificación, para poder situar mejor los punteros en su contexto.

De entre las varias regiones de memoria que existen, hay dos que nos interesan al hablar de punteros (y sobre todo, al depurar): la pila y el heap.

#### La pila

En inglés, *stack*. Su contenido se estudia en profundidad en las asignaturas de *Laboratorio de Estructura de Computadores* y *Compiladores*. Aquí solo diremos que cada vez que se realiza una llamada a una función, se introduce en la pila una estructura que almacena los parámetros pasados a la función, y las variables declaradas dentro de ella. Cuando la función retorna, dicha estructura es destruida.

#### El heap

Esta región queda disponible para las solicitudes de memoria dinámica al sistema operativo. Su crecimiento va ligado a la disminución de la pila, y viceversa.

<sup>4</sup>cuando el programa llega al punto en el que necesita espacio para una línea más

<sup>5</sup>es decir, al programar

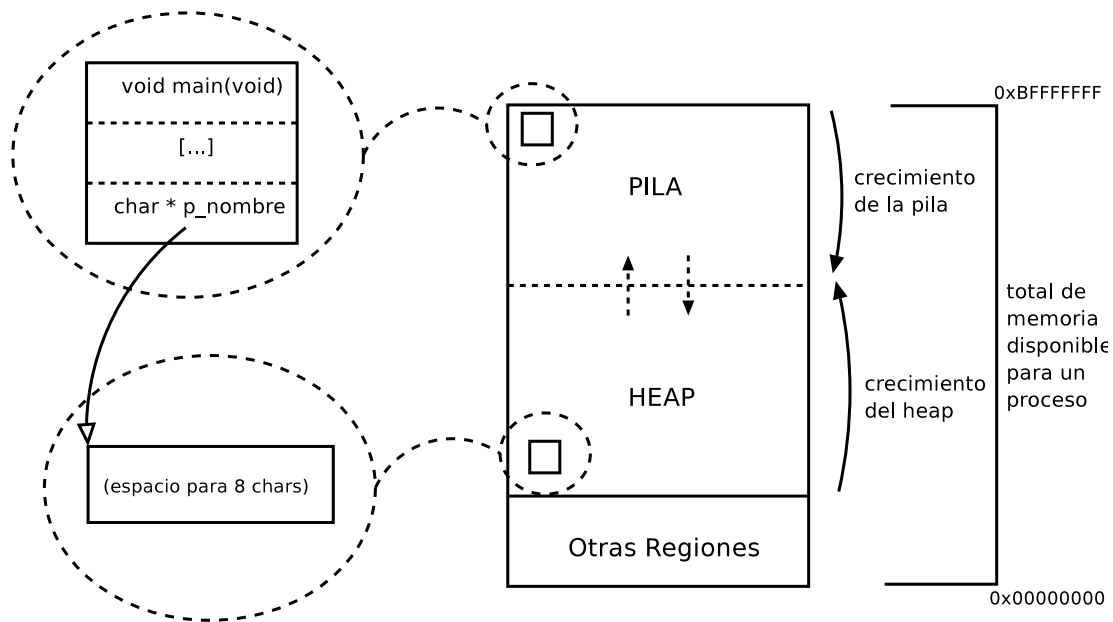


Figura 5.5: Visión general del mapa de memoria

La figura anterior, muestra el resultado de solicitar al sistema operativo espacio en memoria dinámica para 8 chars. Podemos quedarnos con la idea de que las variables locales de una función, y los argumentos de la misma van en la pila, mientras que la memoria dinámica va en el heap.

### 5.6.3. Primitivas básicas

Las primitivas básicas para el manejo de memoria dinámica son:

- `malloc`
- `realloc`
- `free`

#### **malloc**

Solicita memoria dinámica al sistema operativo. Su prototipo es:

```
void *malloc(size_t size);
```

Devuelve un puntero tipo `void`, que apunta a la zona solicitada, o `NULL`, en caso de no poderse cumplir la solicitud (probablemente por falta de memoria libre). El tipo `size_t`, tiene conversión directa desde los `int`.

Por ejemplo, si quisiéramos solicitar memoria dinámica para almacenar 8 chars:

```
char *p_char;
p_char = malloc(8*sizeof(char));
```

No obstante, el autor prefiere especificar a mano los cast (ver 3.4.5) que se deben realizar para que encajen los argumentos:

```
char *p_char;
p_char = (char *) malloc( (size_t) 8*sizeof(char));
```



**Nota:** La zona de memoria devuelta por `malloc` no se inicializa a ningún valor concreto.

**realloc**

Cambia el tamaño de una zona de memoria dinámica, pedida al sistema operativo previamente mediante la orden `malloc`. Su prototipo es:

```
void *realloc(void *ptr, size_t size);
```

Un ejemplo de uso de `realloc`:

```
char *p_char;
int size;

size = 8 * sizeof(char);

/* pedimos memoria en p_char */
p_char = (char *) malloc( (size_t) size);

[.....]

/* necesitamos más memoria en p_char */
size *= 2;
p_char = (char *) realloc(p_char, (size_t) size);
```

Un ejemplo habitual de uso de `realloc` está en 5.8.3.



**Nota:** `realloc` puede devolvernos la zona de memoria solicitada en otra posición. Esto es, independientemente de encargarse de reservar el nuevo espacio solicitado, el puntero que retorna `realloc` puede ser distinto al devuelto originalmente por `malloc`. Aun así, `realloc` se encarga de que el contenido apuntado por el puntero sea el mismo. Dicho de otra manera, si pedimos memoria para  $x$  caracteres, y luego hacemos `realloc` sobre esa zona pidiendo  $x+n$  caracteres, `realloc` se encargará de que los  $x$  primeros caracteres de la zona devuelta (sea la misma zona pero más grande, o sea otra zona distinta) sean idénticos. Como es habitual, si solicitamos más espacio, ese espacio extra no será inicializado.

**free**

Libera una zona de memoria dinámica, solicitada previamente mediante `malloc`. Su prototipo es:

```
void free(void *ptr);
```

Un ejemplo de uso de `free`:

```
char *p_char;

/* pedimos memoria en p_char */
p_char = (char *) malloc( (size_t) 8*sizeof(char));

[...]

free(p_char);
```



**Nota:** Liberar una zona de memoria una segunda vez es ilegal (ver 5.10.3).



**Nota:** Es habitual al empezar a manejar memoria dinámica, dejar la tarea de liberar la memoria solicitada para el final. Esto es una mala política de trabajo. Por cada `malloc` que utilizamos, debemos pensar donde se va a hacer su `free`, y colocar ambos en el código. En cuanto los programas crecen, es habitual olvidarse de liberar memoria, y el consumo de nuestros programas pueden crecer de manera desorbitada.

**strdup**

Es una primitiva incluida en *string.h*. La hemos incluido en el manual, porque se usa frecuentemente, aunque al igual que `strdup`, hay muchas llamadas al sistema similares (piden memoria dinámica por nosotros). Su prototipo es:

```
char *strdup(const char *s);
```

Es una función bastante cómoda, le suministramos un puntero a un string, y nos devuelve un puntero a una zona de memoria dinámica, que es una copia de la cadena que le hemos pasado. Dicho de otra manera, `strdup` equivale a hacer un `malloc` de la longitud de la cadena argumento, y a copiarla sobre la zona devuelta. Un ejemplo de uso sería:

```
char * pointer;
char * data = "Hello world\n";

pointer = strdup(data);

printf("%s", pointer);

free(pointer);
```

Este caso es un candidato ideal para mostrar un error frecuente al trabajar con strings y punteros (ver 5.10.1).

**5.7. Arrays y punteros**

Hasta ahora para nosotros un array era un conjunto ordenado de elementos del mismo tipo.

**5.7.1. Repaso**

La sintaxis en C para declarar un array es:

```
tipo_elementos nombre_array[numero_elementos];
```

La sintaxis para acceder a sus elementos es:

```
nombre_array[indice]
```

Ejemplo: un array de cinco elementos que contenga números primos sería así:

```
int numeros_primos[5];

numeros_primos[0] = 2;
numeros_primos[1] = 3;
numeros_primos[2] = 5;
numeros_primos[3] = 7;
numeros_primos[4] = 11;
```

**5.7.2. Introducción**

Ahora vamos a ver qué es para el compilador un array, y así aprenderemos a usarlos de manera más eficiente. Un array es un conjunto de elementos del mismo tipo. Para que sea conjunto “ordenado”, lo que el compilador hace es juntar todos los elementos en la misma zona de memoria. Almacena la dirección inicial en nuestra variable para saber dónde está el primer elemento, que correspondería al índice “0”. A partir de ahí, al incrementar la dirección de memoria en el tamaño de los elementos, va accediendo a `array[1]`, `array[2]`, etc.

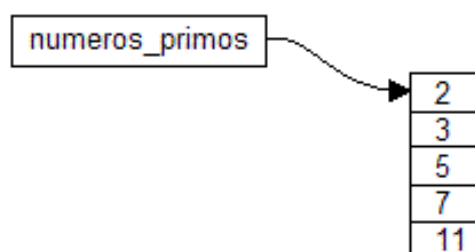


Figura 5.6: Un array de números primos

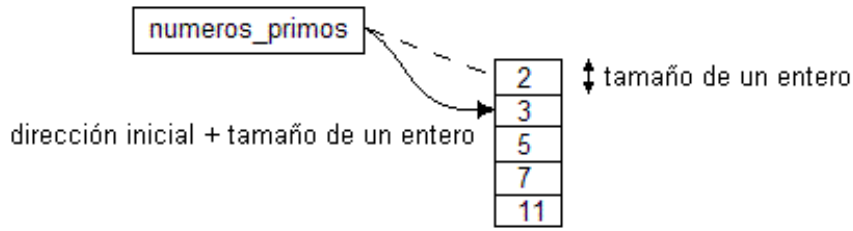


Figura 5.7: Avanzando sobre un array

Así podemos deducir la siguiente fórmula:  $dirección\_elemento = dirección\_inicial + (índice * tamaño\_elementos)$ .

Esta fórmula es la que aplica el compilador para calcular la dirección del elemento al que nos referimos al hacer un acceso al array del tipo `array[índice]`, como por ejemplo `numeros_primos[3]`. Claramente, se dibuja la idea del puntero en el concepto de array:

**DEFINICIÓN: Variable tipo array:** *Un puntero al primer elemento del array.*

¿Cuál es la ventaja de trabajar de punteros, con los posibles problemas que eso trae, en vez de con arrays sin más? La respuesta surge enseguida: un array tiene un tamaño fijo desde su declaración. Sin embargo, trabajando con punteros, nuestro array podrá tener el tamaño que nosotros queramos durante el programa, y podemos incluso variarlo en función de otros datos del programa.

Por supuesto, esta ventaja tiene un precio (aunque muy bajo) que no debemos olvidar. Debemos apuntar en algún sitio (variable, constante) cuánto espacio hemos pedido y en otro cuánto de ese espacio hemos aprovechado. Como al programar no conoceremos el espacio aprovechado del array, deberemos hacer una de estas dos cosas:

- apuntar en otra variable el tamaño ocupado del array.
- hacer que el último elemento del array sea diferente, un dato que no nos puedan introducir, por ejemplo, un número negativo o una letra cuando hablamos de números de teléfono.

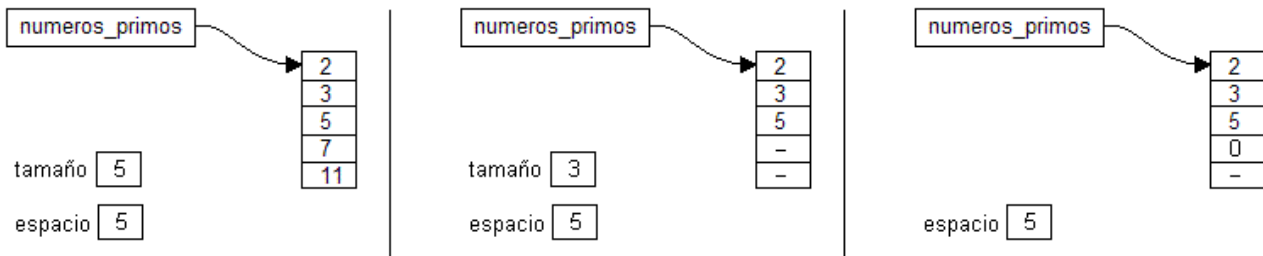


Figura 5.8: Más sobre arrays

**Ejemplo** Queremos que el usuario introduzca varios números de teléfono y los vamos a almacenar en el array `telefonos`. No sabemos cuántos números va a introducir el usuario. Si lo tuviéramos que implementar con un array definido desde su declaración, tendríamos que poner un tope a los números de teléfono que se pueden introducir, por ejemplo 10:

```
int i;
int telefonos[10];

i=0;
while (usuario_introduce && (i<10))
{
    telefonos[i] = numero_introducido;
    i++;
}
numeros_introducidos = i;
```

Sin embargo, trabajando con punteros podemos preguntar al usuario cuántos números quiere introducir:

```
int i, tamaño;
int *telefonos; // o bien: int telefonos[]

telefonos = NULL;
/* es muy recomendable inicializar el valor de un puntero a NULL.
 * Así, si se nos olvida pedir memoria para él, el programa fallará siempre.
 * Por el contrario, si no lo inicializamos, el programa fallará
 * algunas veces sí y otras no. */

tamaño = preguntar_tamaño();
telefonos = malloc (tamaño);

for (i=0;i<tamaño;i++)
    telefonos[i] = numero_introducido;
```

O también variar el tamaño del array de forma dinámica:

```
int i, tamaño, nuevo_tamaño;
int *telefonos; // o bien: int telefonos[]

telefonos = NULL;
i=0;
while (usuario_introduce)
{
    nuevo_tamaño = sizeof (int) * (i+1);
    telefonos = realloc (telefonos,nuevo_tamaño);
    telefonos[i] = numero_introducido;
    i++;
}
tamaño=i;
```



**Nota:** Si el tamaño del array es "X", los índices (que empiezan por cero) irán de "0" a "X-1".

## 5.8. Indirecciones múltiples

Una herramienta muy útil que proporciona C en su manejo de punteros son las indirecciones múltiples, es decir, punteros que apuntan a punteros. Las indirecciones pueden ser dobles, triples, cuádruples ....

### 5.8.1. Declaración

Ejemplo:

```
char **ind_doble;
char ***ind_triple;
```

La primera declaración declararía un puntero de indirección doble de tipo char (léase: puntero que apunta a puntero), mientras que la segunda declararía un puntero de indirección triple a char (puntero que apunta un puntero que apunta a otro puntero [es decir, un lío]). Usar más de indirección triple es abiertamente desaconsejable.

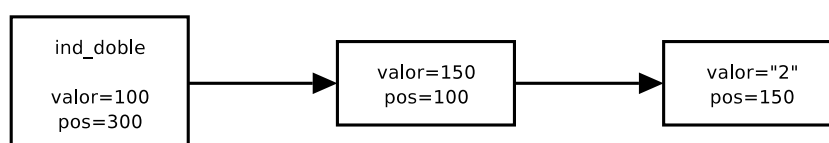


Figura 5.9: Indirección doble

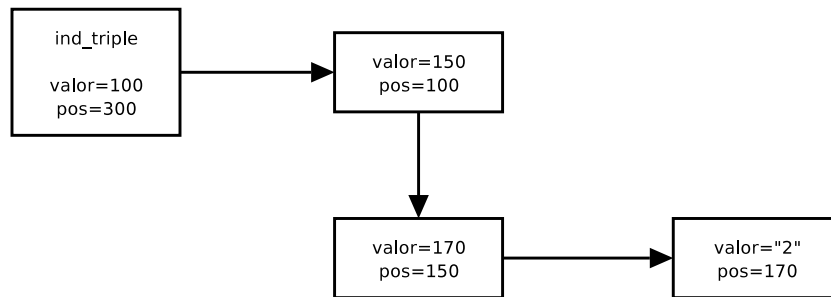


Figura 5.10: Indirección triple

### 5.8.2. Utilización

```

char **ind_doble;
char *puntero_normal;

/* Este acceso nos daría la dirección de la primera indirección */
puntero_normal = *ind_doble;

/* Estos dos accesos serían equivalentes (después de la asignación de
 * arriba) */

*puntero_normal = 'A';
**ind_doble = 'A';

```

Esto es así porque `*ind_doble`, es un puntero a char, es decir, como regla cada `*` que pongamos en un puntero con indirección múltiple quitamos un nivel de indirección.

Una vez más, recordar la dualidad entre índices de array y punteros. Por ejemplo, cuando tratamos con un puntero de doble indirección, como puede ser `char **data`, las expresiones

- `data[1][2]` y
- `*(*(data+1)+2)`

son equivalentes.

#### Ejemplo

- `*ind_doble` sería un puntero simple (de los vistos anteriormente).
- `**ind_doble` sería el valor apuntado, es decir el carácter de los ejemplos anteriores.
- `*ind_triple` sería un puntero de indirección doble.

### 5.8.3. Ejemplos de uso

Uno de los usos de punteros de indirección múltiple, es, por ejemplo, leer un texto completo y almacenarlo por líneas.

```
char **texto;
```

Cada `*texto` sería una línea, y cada `**texto` una letra de la línea seleccionada. Veámoslo con un programa:



### Ejemplo

```
1  int main(){
2
3     char **texto;
4     int i=0;
5
6     /* Se obvia el uso de memoria dinámica. este programa no ejecutaría, está
7     * puesto para fin didáctico */
8
9     /* Leemos una línea de la entrada estándar (máximo de caracteres 1024, y
10    * la almacenamos en texto[0], es decir, la primera línea */
11
12    fgets(texto[0], 1024, stdin);
13
14    /* Leemos una segunda línea */
15    fgets(texto[1], 1024, stdin);
16
17    /* Este bucle imprimiría la primera línea entera */
18    while (texto[0][i] != '\n' && texto[0][i] != '\0')
19        printf("%c", texto[0][i++]);
20
21    /* Se podría reescribir este bucle, siendo equivalente como: */
22    while ( *((*texto)+i) != '\n' && *((*texto)+i) != '\0')
23        printf("%c", *((*texto)+(i++)));
24
25    return 0;
26 }
```

El siguiente ejemplo es especialmente interesante, combina el uso de `malloc` y `realloc` con direcciones múltiples. El programa utiliza un doble puntero, sobre el que ejecuta un `malloc`, y posteriormente, un `realloc` por cada nueva línea que vayamos leyendo. En cada nuevo espacio que devuelve `realloc`, se ejecuta un `malloc` para almacenar una nueva línea de texto.



### Ejemplo

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define MAX_LINEA 1024
5
6  int main(){
7
8      char **texto;
9      int i;
10
11     texto = (char **) malloc((size_t) sizeof(char *));
12
13     for(i=0; i<10; i++) {
14
15         *(texto+i) = (char *) malloc((size_t) sizeof(char) * MAX_LINEA);
16         fgets(*(texto+i), MAX_LINEA, stdin);
17         texto = (char **) realloc(texto,
18                                 (size_t) sizeof(char *) * (i+2));
19     }
20
21     /* mostramos las lineas */
22
23     for (i=0; i<10; i++) {
24         printf("%s", texto[i]);
25     }
26
27     /* liberación de la memoria pedida
28      * el orden es importante, primero liberamos
29      * los punteros del array, luego el array */
30
31     for (i=0; i<10; i++)
32         free(texto[i]);
33
34     free(texto);
35
36     return 0;
37 }

```

Debemos tener en cuenta que el orden de los `free` es importante: primero liberamos cada uno de los `malloc` que se ejecutaron dentro del bucle (zonas sueltas de memoria, que almacenan una línea cada una). Al final, liberamos el doble puntero, sobre el que hemos ejecutado un `malloc`, y varios `realloc`. Si hiciéramos la liberación al revés, estaríamos pasándole como argumento a `free`, punteros situados en una zona de memoria que ha sido liberada (ya no es nuestra). El comportamiento sería impredecible.

Como se vió en la sección 5.7, es indiferente acceder a un puntero múltiple usando la notación de los arrays o el operador de acceso junto con la aritmética de punteros:

```

double ** pointer;
double * p1;

p1 = (double *) malloc((size_t) 8 * sizeof(double));
pointer = &p1; /* pointer apunta a p1 */

pointer[0][0]=3.141592;
pointer[0][1]=6.022E23;

```

```
printf("%g %g\n",pointer[0][0], pointer[0][1]);
printf("%g %g\n",**pointer, *(*pointer+1));
```

#### 5.8.4. Cadenas enlazadas

Uno de los usos más útiles de punteros es la implementación de las cadenas enlazadas, que se estudian en la asignatura de *Estructura de Datos I*.

El tipo `cadena_enlazada` es un puntero a un nodo, consistente de un puntero al siguiente nodo (una `cadena_enlazada` en si misma) y un dato de un tipo cualquiera.

La dificultad de implementar es qué tipo declarar primero, `nodo` o `cadena_enlazada`. Para ello, usaremos una técnica conocida como *forward declaration*. La idea consiste en avisar al compilador de que estamos declarando un tipo, que es un puntero a una estructura, pero que dicha estructura aún no la hemos declarado, la declaramos después.

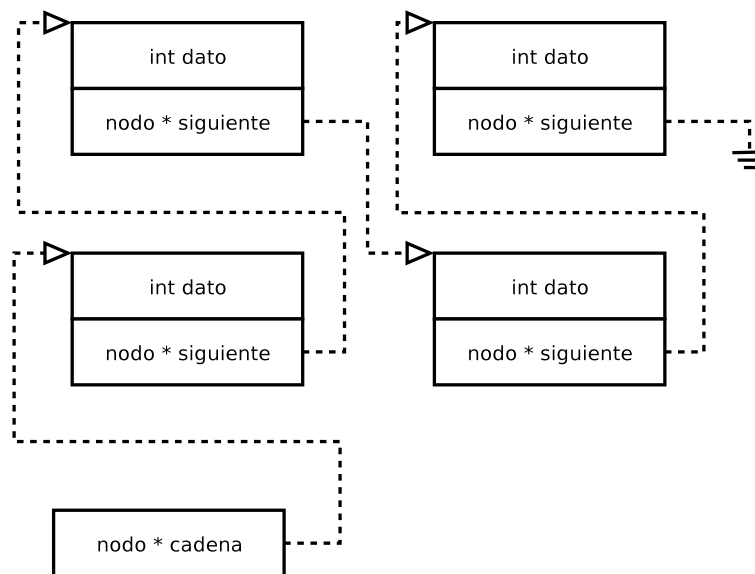


Figura 5.11: Ejemplo de una cadena enlazada

Una manera incorrecta de implementarlas sería:

```
typedef nodo * cadena_enlazada;

struct nodo{
    cadena_enlazada siguiente;
    int dato;
}
```

debido a que el compilador no tiene definido el tipo `nodo` en el momento de definir `cadena_enlazada`. La forma correcta sería:

```
typedef struct nodo *cadena_enlazada;

struct nodo{
    cadena_enlazada siguiente;
    int dato;
};
```

o también, si quisieramos realizar el `typedef` sobre ambos tipos:

```
typedef struct nodo *cadena_enlazada;

typedef struct nodo{
```

```
cadena_enlazada siguiente;
int dato;
};
```

Un ejemplo de uso de las cadenas enlazadas es:



### Ejemplo

```
1
2 #include <stdlib.h>
3 #include <stdio.h>
4
5 typedef struct nodo *cadena_enlazada;
6
7 struct nodo{
8     cadena_enlazada siguiente;
9     int dato;
10 };
11
12 int main(){
13     cadena_enlazada cad;
14     cadena_enlazada actual; /* Apunta al nodo con el que trabajamos */
15
16     cad = (cadena_enlazada) malloc((size_t) sizeof(struct nodo));
17
18     cad->dato=2;
19     cad->siguiente = (cadena_enlazada) malloc((size_t) sizeof(struct nodo));
20     actual = cad->siguiente;
21     actual->dato = 3; /* seria equivalente a cad->siguiente->dato = 3; */
22     actual->siguiente = NULL;
23
24     printf("%d\n", cad->dato);
25     printf("%d\n", actual->dato);
26
27     /* Devolvemos la memoria */
28     free(actual);
29     free(cad);
30     return 0;
31 }
```

## 5.9. Paso por referencia vs. paso por valor

### 5.9.1. ¿Qué es el paso por referencia?

A la hora de pasar una variable como argumento a una función, el paso por referencia consiste en entregar como argumento un puntero a la variable, y no el contenido de la variable.

### 5.9.2. ¿Para qué necesito el paso por referencia?

Si tratamos de modificar los valores de los argumentos de una función, estos cambios no serán vistos desde fuera de la misma. Esto es debido a que los argumentos de la función son copias de las variables reales y son almacenadas como variables locales a dicha función, desapareciendo por tanto al regresar a la llamante. Se incluye un ejemplo a continuación:



#### Ejemplo

```
1  #include <stdio.h>
2  /*
3   * Función que modifica un argumento pasado por valor
4   */
5  int no_mod_args (int arg)
6  {
7   /*
8   * Modificamos el valor del argumento
9   * (no tendrá efecto fuera de la función)
10  */
11  arg = 2;
12  return 0;
13  }
14  int main ()
15  {
16  /*
17   * Declaramos una variable para comprobar que no
18   * es modificada al llamar a la función pasándola
19   * por valor
20   */
21  int var1;
22  var1 = 1;
23  /*
24   * Imprimimos el valor de la variable antes de la
25   * llamada a la función
26   */
27  printf ("Variable var1 = %d\n", var1);
28  /* Variable var1 = 1 */
29
30  /*
31   * Llamada a la función que modifica el argumento
32   */
33  no_mod_args (var1);
34  /*
35   * Imprimimos el valor de la variable después de la
36   * llamada a la función
37   */
38  printf ("Variable var1 = %d\n", var1);
39  /* Variable var1 = 1 */
40
41  return 0;
42  }
```

Si compilamos y ejecutamos el código anterior, obtenemos el siguiente resultado:

```
Variable var1 = 1
Variable var1 = 1
```

Como era de esperar, no se ha modificado el valor de la variable `var1` fuera de la función.

Por esto, si queremos modificar el valor de un argumento en una función, debemos pasar este parámetro *por referencia*, como se muestra en el siguiente ejemplo:



### Ejemplo

```
1  #include <stdio.h>
2
3  /*
4   * Función que modifica un argumento pasado por referencia
5   */
6  int mod_args (int * arg)
7  {
8   /*
9   * Modificamos el valor del argumento
10  */
11  (*arg) = 2;
12
13  return 0;
14 }
15
16 int main ()
17 {
18  /*
19   * Declaramos una variable para comprobar que
20   * es modificada al llamar a la función pasándola
21   * por referencia
22   */
23
24  int var1;
25
26  var1 = 1;
27  /*
28   * Imprimimos el valor de la variable antes de la
29   * llamada a la función
30   */
31  printf ("Variable var1 = %d\n", var1);
32  /* Variable var1 = 1 */
33
34  /*
35   * Llamada a la función que modifica el argumento
36   * pasando como parámetro la dirección de la variable
37   */
38  mod_args (&var1);
39  /*
40   * Imprimimos el valor de la variable después de la
41   * llamada a la función
42   */
43  printf ("Variable var1 = %d\n", var1);
44  /* Variable var1 = 2 */
45
46  return 0;
47 }
```

Como se puede comprobar, la salida de este programa es correcta:

```
Variable var1 = 1
Variable var1 = 2
```

Si en una función quisiéramos modificar un parámetro que fuera un puntero, sería necesario pasar como argumento la dirección del mismo, es decir, *doble indirección*. Y así sucesivamente si el argumento fuera doble puntero, triple puntero, etc.

También es muy aconsejable el paso por referencia en el caso en que una función reciba como argumento una gran estructura de datos (arrays, matrices, ...), puesto que el paso por valor implicaría una copia completa de la estructura en el espacio de memoria de la función.

## 5.10. Errores con punteros

### 5.10.1. Comparación de punteros a cadenas

Un error que ocurre frecuentemente al empezar a programar en C es intentar comparar dos cadenas de caracteres mediante sus punteros. Veámoslo en el siguiente ejemplo:



#### Ejemplo

```
1 char * string1 = "Cadena 1";
2 char * string2 = "Cadena 2";
3
4 int main (void)
5 {
6     if (string1 == string2)
7         printf ("Cadenas iguales\n");
8     else
9         printf ("Cadenas distintas\n");
10    return 0;
11 }
```

En la ejecución de este programa, tendríamos la siguiente asignación de memoria:

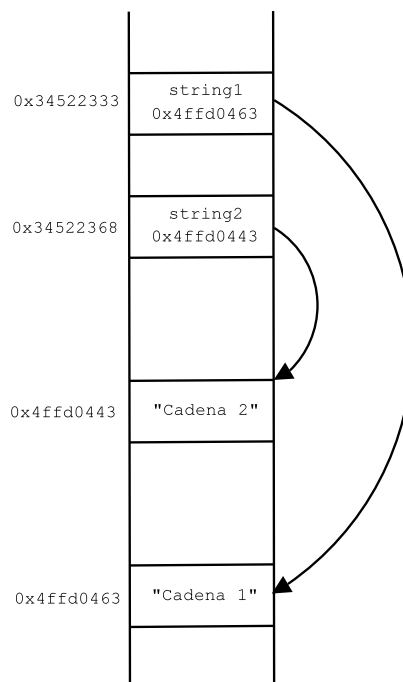


Figura 5.12: Colocación de strings en memoria

Al observar el diagrama, se entiende más fácilmente que el código anterior no es correcto, esto es, únicamente compara el valor de las variables `string1` y `string2`, que son las direcciones de memoria de las cadenas de caracteres, y no las cadenas en sí.

Por este motivo, si queremos comparar dos cadenas de caracteres correctamente, deberemos usar las funciones de la librería estándar de C que operan sobre strings. En concreto, la función a utilizar es `strcmp`<sup>6</sup>, que compara dos cadenas carácter a carácter.

De esta manera, el código corregido quedaría:



### Ejemplo

```

1 char * string1 = "Cadena 1";
2 char * string2 = "Cadena 2";
3
4 int main (void)
5 {
6     if (strcmp(string1, string2) == 0)
7         printf ("Cadenas iguales\n");
8     else
9         printf ("Cadenas distintas\n");
10    return 0;
11 }

```

## 5.10.2. Punteros “a Roma” (*memory leaks*)

Observemos lo que ocurre en el siguiente ejemplo:



### Ejemplo

```

1 int a;
2 int * ptr;
3
4 int main ( void )
5 {
6     /* Solicitamos memoria para ptr */
7     ptr = (int *) malloc (sizeof (int));
8
9     /* Asignamos el valor 5 a la posición de
10    * memoria apuntada por ptr y 3 al entero a
11    */
12    (*ptr) = 5;
13
14    a = 3;
15
16    /* Asignamos al puntero ptr la dirección
17    * de la variable a
18    */
19    ptr = &a;
20
21    return 0;
22 }

```

Al ejecutar el programa anterior, obtendríamos el siguiente diagrama:

<sup>6</sup>Explicada más detalladamente en la sección 6.5

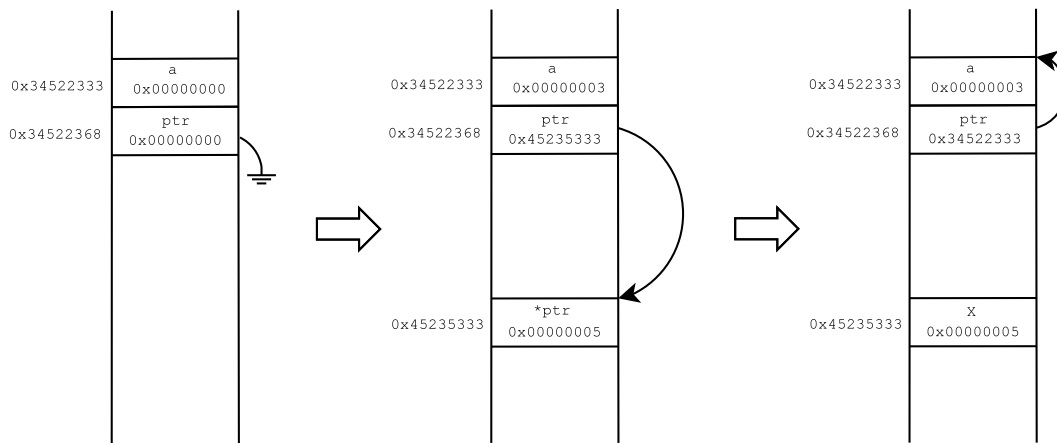


Figura 5.13: Punteros a roma

Como se puede apreciar, en la finalización del programa, ha quedado una zona de memoria (0x45235333) ocupada, que no está apuntada por ninguna variable. Esto implica no poder acceder al contenido de dicha zona y por tanto, no poder liberarla, con el consiguiente gasto innecesario de memoria.

### 5.10.3. Doble liberación

Cuando se programa utilizando memoria dinámica, es necesario liberar toda la memoria solicitada, pero únicamente es necesario realizarlo una vez. En ocasiones, se intenta liberar varias veces la misma zona de memoria, consiguiendo un `Segmentation Fault`, pues esta memoria ya estaba marcada como libre. Se puede comprobar con un ejemplo muy simple como el siguiente:



#### Ejemplo

```

1  #include <stdlib.h>
2
3  char * string;
4
5  int main ( void )
6  {
7      /* Solicitamos memoria para el puntero */
8      string = (char *) malloc (200 * sizeof (char));
9
10     /* Primera liberación */
11     free (string);
12
13     /*
14      * Segunda liberación:
15      * (al llegar a este punto, el programa dara un Segfault)
16      */
17     free (string);
18
19     return 0;
20 }
```

### 5.10.4. Usar . en lugar de ->

Al utilizar estructuras apuntadas por punteros, hemos de tener precaución y utilizar correctamente los operadores `.` y `->`. Vemos un mal uso de ellos en el siguiente ejemplo:



### Ejemplo

```
1  /* Definimos una estructura de ejemplo */
2  struct estructura {
3      char caracter;
4      int  entero;
5  };
6
7  int main ( void )
8  {
9      /* Declaramos una variable que apunte a dicha estructura */
10     struct estructura * s_ptr;
11
12     /* Solicitamos memoria para dicha variable */
13     s_ptr = (struct estructura *) malloc (sizeof (struct estructura));
14
15     /* Rellenamos los campos */
16     s_ptr.caracter = 'A';
17     s_ptr.entero   = 2345;
18     /*
19      * ERROR: estamos intentando acceder a los campos caracter
20      *           y entero de un puntero, el compilador dará un error
21      */
22
23     /* Liberamos la memoria solicitada */
24     free (s_ptr);
25
26     return 0;
27 }
28
```

Y cual es su uso correcto:



### Ejemplo

```
1  /* Definimos una estructura de ejemplo */
2  struct estructura {
3      char caracter;
4      int  entero;
5  };
6
7  /* Definimos una variable que apunte a dicha estructura */
8  struct estructura * s_ptr;
9
10 int main ( void )
11 {
12
13     /* Declaramos una variable que apunte a dicha estructura */
14     struct estructura * s_ptr;
15
16     /* Solicitamos memoria para dicha variable */
17     s_ptr = (struct estructura *) malloc (sizeof (struct estructura));
18
19     /* Rellenamos los campos */
20     (*s_ptr).caracter = 'A';
21     (*s_ptr).entero   = 2345;
22
23     /* Rellenamos los campos (más elegante)*/
24     s_ptr -> caracter = 'A';
25     s_ptr -> entero   = 2345;
26
27     /* Liberamos la memoria solicitada */
28     free (s_ptr);
29
30     return 0;
31 }
32
```

#### 5.10.5. Operar con los punteros en lugar de con los contenidos

En los programas que utilizan punteros (memoria dinámica), es imprescindible diferenciar perfectamente entre el puntero en sí y su contenido, pues en otro caso, realizaremos algunas operaciones sobre el puntero cuando en realidad queremos hacerlas sobre el contenido <sup>7</sup>. Veamos algunos errores más frecuentes:

<sup>7</sup>Un caso de este tipo es comentado más en detalle en la sección 5.10.1



### Ejemplo

```
1  int * var1;
2  int * var2;
3
4
5  int main ( void )
6  {
7      /* Solicitamos memoria para cada puntero */
8      var1 = (int *) malloc (sizeof (int));
9      var2 = (int *) malloc (sizeof (int));
10
11
12     /* Asignamos un valor a cada variable */
13     var1 = 3;
14     var2 = 4;
15     /*
16      * ERROR: las variables no son enteros, sino punteros
17      *         estamos cambiando la dirección y no el contenido
18      */
19
20     /* Comparamos las variables */
21     if (var1 == var2)
22         printf ("Iguales\n");
23     else
24         printf ("Diferentes\n");
25     /*
26      * ERROR: en este caso estamos comparando las direcciones
27      *         de las variables (punteros) y no su contenido
28      */
29
30     /* Liberamos la memoria solicitada */
31     free (var1);
32     free (var2);
33
34     return 0;
35 }
```

Ahora la versión correcta:



### Ejemplo

```
1  int * var1;
2  int * var2;
3
4
5  int main ( void )
6  {
7      /* Solicitamos memoria para cada puntero */
8      var1 = (int *) malloc (sizeof (int));
9      var2 = (int *) malloc (sizeof (int));
10
11     /* Asignamos un valor a cada variable */
12     (*var1) = 3;
13     (*var2) = 4;
14     /*
15      * OK: ahora lo estamos asignando correctamente
16      */
17
18     /* Comparamos las variables */
19     if ((*var1) == (*var2))
20         printf ("Iguales\n");
21     else
22         printf ("Diferentes\n");
23     /*
24      * OK: ahora comparamos correctamente los valores
25      */
26
27     /* Liberamos la memoria solicitada */
28     free (var1);
29     free (var2);
30
31     return 0;
32 }
```

#### 5.10.6. Finalización de cadenas

En el lenguaje C, las cadenas de caracteres deben tener un carácter de finalización, `\0`. Cuando asignamos o inicializamos una variable con una cadena entre comillas dobles, el carácter de finalización es puesto automáticamente por el compilador, por ejemplo:

```
char * string1 = "Cadena";
```

El código anterior inicializará la variable de la siguiente manera:

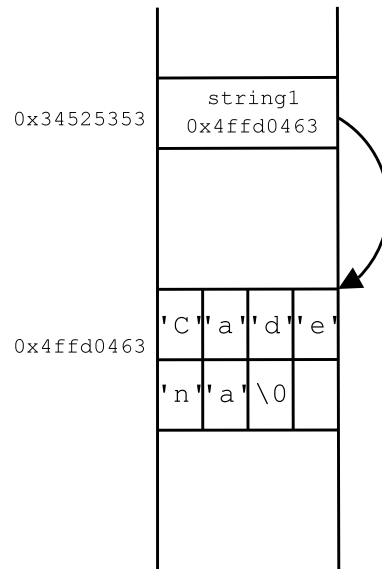


Figura 5.14: Carácter de finalización

De esta forma, en tiempo de ejecución, al operar sobre una cadena se busca el carácter `\0`, que indique el fin de la misma. Es por esto que si se olvida poner dicho indicador cuando se está rellenando una cadena carácter a carácter (en un bucle, por ejemplo), al intentar acceder a ella, el programa seguirá leyendo la información que exista a continuación de la cadena hasta que muy probablemente intente acceder a una zona en la que no tenga permisos y termine con el típico `Segmentation Fault`.

# Capítulo 6

## La librería estándar de C

### 6.1. Introducción

El estándar de C define sólo unas pocas palabras reservadas. Sólo con ellas no puede hacerse un programa “normal” en la vida real. El programador necesita una serie de funciones y herramientas **estándar**, que deben estar disponibles en cualquier entorno de programación de C / C++. A este conjunto de funciones se le llama **librería estándar**. Las funciones se declaran en ficheros de cabecera o **.h**. Las cabeceras contienen única y exclusivamente los prototipos de las funciones. El código o cuerpo de las funciones se incluye en **ficheros objeto** que son realmente la librería.

### 6.2. Principales ficheros de cabecera

Los principales ficheros de cabecera de C “suelen ser” los siguientes:

- `ctype.h`: Funciones útiles para la clasificación y el mapeado de códigos.
- `errno.h`: Funciones que permiten comprobar el valor almacenado en `errno` por algunas funciones de librerías.
- `float.h`: Funciones que establecen algunas propiedades de las representaciones de tipos real.
- `limits.h`: Funciones que establecen algunas propiedades de las representaciones de tipos enteros.
- `math.h`: Funciones que sirven para realizar operaciones matemáticas comunes sobre valores de tipo `double`.
- `stdarg.h`: Son declaraciones que permiten acceder a los argumentos adicionales sin nombre en una función que acepta un número variable de argumentos.
- `stdio.h`: Macros y funciones para realizar operaciones de entrada y salida sobre ficheros y flujos de datos.
- `stdlib.h` y a veces `unistd.h`: Declaraciones de una colección de funciones útiles y la definición de tipos y macros para usarlas. Entre ellas suele estar la función **malloc** que permite hacer peticiones de memoria dinámica al sistema.
- `string.h`: Declaración de una colección de funciones útiles para manejar cadenas y otros arrays de caracteres.
- `time.h`: Declaración de funciones para el manejo de fechas.

### 6.3. `stdio.h`

#### 6.3.1. Funciones para el manejo de la Entrada/Salida

##### `printf`

```
int printf (const char *formato, ...);
```

Escribe texto formateado por el flujo `stdout`, según las especificaciones de “formato” y la lista de expresiones. Devuelve el número de caracteres escritos o un valor negativo en caso de error.

**scanf**

```
int scanf (const char *formato, ...);
```

Lee texto por el flujo `stdin` y lo almacena según las especificaciones de “formato”. Devuelve el número de valores asignados o EOF si se produce error o se alcanza fin de fichero sin producirse lectura.

**puts**

```
int puts (const char *s);
```

Escribe los caracteres de la cadena “s” por el flujo `stdout`. Escribe un carácter “NL” en lugar del nulo de terminación. Devuelve un valor no negativo. En caso de error devuelve EOF.

**6.3.2. Funciones para el manejo de ficheros****fopen**

```
FILE *fopen(const char *nombre_fichero, const char *modo);
```

Abre el fichero de nombre “nombre\_fichero”, lo asocia con un flujo de datos y devuelve un puntero al mismo. Si falla la llamada, devuelve un puntero nulo. Algunos de los caracteres iniciales de “modo” son:

“r”, para abrir un fichero de texto existente para su lectura

“w”, para crear un fichero de texto o abrir y truncar uno existente, para su escritura

“a”, para crear un fichero de texto o abrir uno existente, para su escritura. El indicador de posición se coloca al final del fichero antes de cada escritura

“r+”, para abrir un fichero de texto existente para su lectura y escritura

**Ejemplo**

```
1 #include <stdio.h>
2
3 int main(void) {
4     FILE *flujo;
5     flujo = fopen ("datos.txt", "w");
6     if (flujo == NULL)
7     {
8         fprintf(stderr, "Error: el fichero no se ha podido abrir\n");
9     }
10 }
```

**fclose**

```
int fclose(FILE *flujo);
```

Cierra el fichero asociado con “flujo”. Devuelve 0 en caso de éxito y EOF (end of file) en caso contrario.

**Ejemplo**

```

1  #include <stdio.h>
2
3  int main(void) {
4      FILE *flujo;
5      /* abre el fichero */
6      flujo = fopen ("datos.txt","w");
7      if (flujo == NULL)
8          {
9              fprintf(stderr,"Error: el fichero no se ha podido abrir\n");
10         }
11         /* lo cierra */
12         fclose(flujo);
13     }

```

**fwrite**

```
size_t fwrite(const void *buffer, size_t n, size_t c, FILE *flujo);
```

La rutina `fwrite` permite escribir `c` elementos de longitud `n` bytes almacenados en el buffer apuntado por “flujo”.

**Ejemplo**

```

1  #include <stdio.h>
2
3  int main(void) {
4      FILE *flujo;
5      char *buffer = "Hola, esto es un ejemplo muy sencillo.";
6      int nescritos;
7
8      /* abre el fichero */
9      flujo = fopen ("datos.txt","w");
10     if (flujo == NULL)
11         {
12             fprintf(stderr,"Error: el fichero no se ha podido abrir\n");
13         }
14
15     /*Escribe en el fichero*/
16     nescritos = fwrite(buffer,sizeof(char),strlen(buffer),flujo);
17
18     printf("Se han escrito:%d caracteres\n",nescritos);
19
20     /* lo cierra */
21     fclose(flujo);
22 }

```

**fread**

```
size_t fread(const void *buffer, size_t n, size_t c, FILE *flujo);
```

La rutina `fread` permite leer `c` elementos de longitud `n` bytes del fichero apuntado por “flujo” y los almacena en el buffer especificado.



### Ejemplo

```

1  #include <stdio.h>
2
3  int main(void) {
4      FILE *flujo;
5      char buffer[38];
6      int nleidos;
7
8      /* abre el fichero */
9      flujo = fopen ("datos.txt","r");
10     if (flujo == NULL)
11     {
12         fprintf(stderr,"Error: el fichero no se ha podido abrir\n");
13         exit(1);
14     }
15
16     /* Lee del fichero*/
17     nleidos = fread(buffer,sizeof(char), 37, flujo);
18
19     /* Acotamos el buffer leído */
20     buffer[nleidos] = '\0';
21
22     printf( "Ha leído %d caracteres: %s\n", nleidos, buffer );
23
24     /* lo cierra */
25     fclose(flujo);
26
27 }

```

### fgetc

```
int fgetc(FILE *flujo);
```

Lee el siguiente carácter por “flujo”, avanza el indicador de posición y devuelve `int`. Devuelve `EOF` si pone a 1 el indicador de fin de fichero o el de error.

### fgets

```
char *fgets(char *s, int n, FILE *flujo);
```

Lee caracteres por “flujo” y los almacena en elementos sucesivos del “array” que comienza en “s”, continuando hasta que almacene “n-1” caracteres, almacene un carácter del nueva línea o ponga a 1 los indicadores de error o de fin de fichero. Si almacena un carácter, concluye almacenando un carácter nulo en el siguiente elemento del “array”. Devuelve “s” si almacena algún carácter y no ha puesto a 1 el indicador de error; en caso contrario devuelve un puntero nulo.

### fputc

```
int fputc(int c, FILE *flujo);
```

Escribe el carácter `c` por “flujo”, avanza el indicador de posición del fichero y devuelve `int c`. En caso de error devuelve `EOF`.

### fputs

```
int fputs(const char *s, FILE *flujo);
```

Escribe los caracteres de la cadena `s` por “flujo”. No escribe el carácter nulo de terminación. En caso de éxito, devuelve un valor no negativo; en caso de error devuelve `EOF`.

**fscanf**

```
int fscanf(FILE *flujo, const char *formato, ...);
```

Lee texto y convierte a la representación interna según el formato especificado en `formato`. Devuelve el número de entradas emparejadas y asignadas, o EOF si no se almacenan valores antes de que se active el indicador de error o de fin de fichero.

**fprintf**

```
int fprintf(FILE *flujo, const char *formato, ...);
```

Genera texto formateado, bajo el control de formato `formato` y escribe los caracteres generados por `flujo`. Devuelve el número de caracteres generados o un valor negativo en caso de error.

A modo de resumen estas son las especificaciones de formato más comunes:

Formato	Descripción
%d	Entero con signo
%u	Entero sin signo
%c	Caracter
%s	Puntero a cadena de caracteres

**Ejemplo**

```

1  #include <stdio.h>
2
3  int main(void) {
4      FILE *flujo ;
5      char *buffer = " Esto es un ejemplo sencillo.";
6      int nescritos;
7
8      /* abre el fichero */
9      flujo = fopen ("datos.txt","a");
10     if (flujo == NULL)
11     {
12         fprintf( stderr,"Error: el fichero no se ha podido abrir\n");
13     }
14
15     nescritos =  fprintf(flujo,
16         "El contenido de la cadena buffer es: %s \n",buffer);
17
18     if (nescritos < 0)
19     {
20         fprintf (stderr,
21             "Error al escribir %d caracteres en el fichero %s \n",
22             strlen(buffer),flujo);
23     }
24
25     /* Nota:
26     stdout es la salida estándar, si no está redirigida
27     suele ser la pantalla
28
29     stderr es la salida de error estándar, si no está redirigida
30     suele ser la pantalla
31     */
32
33     /* lo cierra */
34     fclose(flujo);
35 }

```

**fseek**

```
int fseek( FILE *flujo, long desplazamiento, int origen);
```

La función `fseek` mueve el puntero de posición del fichero correspondiente al flujo de datos apuntado por “flujo”. La nueva posición, medida en bytes, se obtiene añadiendo el número indicado por desplazamiento a la posición especificada por origen. La variable origen puede tomar tres valores:

- `SEEK_SET`: El puntero de posición apuntará al inicio del fichero más el desplazamiento
- `SEEK_CUR`: El puntero de posición apuntará a la posición actual del puntero de posición del fichero más el desplazamiento.
- `SEEK_END`: El puntero de posición apuntará al fin del fichero más el desplazamiento (deberá ser menor o igual que cero).



**Nota:** Al abrir un fichero el puntero de posición apunta al principio del mismo. Toda lectura o escritura hecha en el fichero modifica el puntero de posición.

**Ejemplo**

```
1  #include <stdio.h>
2
3  int main(void) {
4      FILE *flujo;
5      char *buffer = " Este texto quedara al final del fichero.";
6      int nescritos;
7
8      /* abre el fichero */
9      flujo = fopen ("datos.txt", "a");
10     if (flujo == NULL)
11     {
12         fprintf(stderr, "Error: el fichero no se ha podido abrir\n");
13     }
14
15     /* Mueve el puntero de posicion para escribir al final,
16      * aunque no haria falta porque en la llamada a fopen se
17      * abrio el fichero con el modo "a" que situa el puntero de
18      * posicion al final del fichero
19      */
20     fseek(flujo, 0, SEEK_END);
21
22     /* Escribe en el fichero */
23     nescritos = fprintf (flujo, "%s", buffer);
24
25     /* La siguiente llamada es equivalente, solo que usa fwrite:
26     nescritos = fwrite(buffer, sizeof(char), strlen(buffer), flujo);
27     */
28
29     fprintf (stdout, "Ha escrito %d caracteres:\n", nescritos);
30
31     /* lo cierra */
32     fclose(flujo);
33 }
```

## 6.4. *stdlib.h*

### 6.4.1. Funciones para la conversión de tipos

#### **abs**

```
int abs(int i);
```

Devuelve el valor absoluto de *i*.

#### **atof**

```
double atof(const char *s);
```

Convierte los caracteres de la cadena *s* a la representación interna de tipo `double` y devuelve ese valor.

#### **atoi**

```
int atoi(const char *s);
```

Convierte los caracteres de la cadena *s* a la representación interna de tipo `int` y devuelve ese valor.

#### **atol**

```
long atol(const char *s);
```

Convierte los caracteres de la cadena *s* a la representación interna de tipo `long` y devuelve ese valor.

#### **strtod**

```
double strtod(const char *s, char **fptr);
```

Convierte los caracteres iniciales de la cadena *s* en la correspondiente representación interna de tipo `double` y devuelve ese valor. Si *fptr* no es un puntero nulo, la función almacena en él un puntero al resto de la cadena que no se ha convertido.

#### **strtol**

```
long strtol(const char *s, char **fptr);
```

Convierte los caracteres iniciales de la cadena *s* en la correspondiente representación interna de tipo `long` y devuelve ese valor. Si *fptr* no es un puntero nulo, la función almacena en él un puntero al resto de la cadena que no se ha convertido.

### 6.4.2. Funciones para el manejo de memoria

#### **malloc**

```
void *malloc(size_t longitud);
```

Asigna una dirección de memoria para un objeto de datos de tamaño *longitud* y devuelve esa dirección.

#### **calloc**

```
void *calloc(size_t nelem, size_t longitud);
```

Asigna una localización en memoria a un objeto de datos *array* que contiene *nelem* elementos de tamaño *longitud*, asignando ceros a todos los bytes del *array* y devuelve la dirección del primer elemento en caso de éxito; en caso contrario, devuelve un puntero nulo.

#### **realloc**

```
void *realloc(void *p, size_t longitud);
```

Cambia el tamaño de la memoria apuntada por *p* al que se indica con *longitud*. Asigna una dirección de memoria para un objeto de datos de tamaño *longitud*, copiando los valores almacenados en *p*. Devuelve la nueva dirección de memoria asignada.

**free**

```
void free(void *p);
```

Si *p* no es un puntero nulo, la función libera la memoria asignada al objeto de datos cuya dirección es *p*, en caso contrario, no hace nada. Se puede liberar la memoria asignada con `calloc`, `malloc`, `realloc`.

**6.5. string.h****strcmp**

```
int strcmp(const char *s1, const char *s2);
```

Compara los elementos de dos cadenas *s1* y *s2* hasta que encuentra elementos diferentes. Si todos son iguales, devuelve 0. Si el elemento diferente de *s1* es mayor que el de *s2*, devuelve un valor mayor que cero; en caso contrario, devuelve un valor menor que cero.

**strcpy**

```
char *strcpy(char *s1, const char *s2);
```

Copia la cadena *s2*, incluyendo el nulo, en el *array* de elementos `char` que comienza en *s1*. Devuelve *s1*.

**strdup**

```
char *strdup(const char *s);
```

Devuelve un puntero a una nueva cadena de caracteres que es un duplicado de la cadena *s*. La memoria para esta cadena de caracteres se obtiene con la función `malloc` y se libera con la función `free`.

**strlen**

```
size_t strlen (const char *s);
```

Devuelve el número de caracteres de la cadena *s*, sin incluir el nulo de terminación.

**strncmp**

```
int strncmp(const char *s1, const char *s2, size_t n);
```

Compara los elementos de las cadenas *s1* y *s2* hasta que encuentra alguno diferente o hasta que se han comparado *n* elementos. Si todos los elementos son iguales, devuelve 0. Si el elemento diferente de *s1* es mayor que el de *s2*, devuelve un número positivo. En caso contrario, devuelve un número negativo.

**strncpy**

```
char *strncpy(char *s1, const char *s2, size_t n);
```

Copia la cadena *s2*, sin incluir el nulo, en la cadena *s1*. Copia no más de *n* caracteres de *s2*. Entonces almacena, cero o más caracteres nulos si son necesarios para completar un total de *n* caracteres. Devuelve *s1*.

**strndup**

```
char *strndup(const char *s, size_t n);
```

Devuelve un puntero a una nueva cadena de caracteres que es un duplicado de la cadena *s*, solo copia los primeros *n* caracteres, incluyendo el nulo. La memoria para esta cadena de caracteres se obtiene con la función `malloc` y se libera con la función `free`.

**6.6. math.h****ceil**

```
double ceil(double x);
```

Valor entero más pequeño no menor que *x*.

**cos**

```
double cos(double x);
```

Coseno de  $x$  en radianes.

**exp**

```
double exp(double x);
```

Exponencial de  $x$ ,  $e^x$ .

**fabs**

```
double fabs(double x);
```

Valor absoluto de  $x$ ,  $|x|$ .

**floor**

```
double floor(double x);
```

Mayor valor entero menor que  $x$ .

**log**

```
double log(double x);
```

Devuelve el logaritmo natural de  $x$ .

**log10**

```
double log10(double x);
```

Devuelve el logaritmo en base 10 de  $x$ .

**pow**

```
double pow(double x, double y);
```

Devuelve  $x$  elevado a la potencia  $y$ ,  $x^y$ .

**sin**

```
double sin(double x);
```

Devuelve el seno de  $x$  (en radianes).

**sqrt**

```
double sqrt(double x);
```

Devuelve la raíz cuadrada de  $x$ .

**tan**

```
double tan(double x);
```

Devuelve la tangente de  $x$  (en radianes).

## 6.7. ctype.h

**islower**

```
int islower(int c);
```

Devuelve un valor distinto de cero si  $c$  es cualquiera de las letras minúsculas  $[a-z]$  u otra minúscula local.

**isupper**

```
int isupper (int c);
```

Devuelve un valor distinto de cero si `c` es cualquiera de las letras mayúsculas `[A-Z]` u otra mayúscula local.

**tolower**

```
tolower (int c);
```

Devuelve la correspondiente letra minúscula si existe y si `isupper(c)` es distinto de cero; en caso contrario, devuelve `c`.

**toupper**

```
toupper (int c);
```

Devuelve la correspondiente letra mayúscula si existe y si `islower(c)` es distinto de cero; en caso contrario, devuelve `c`.



# Capítulo 7

## Temas avanzados

### 7.1. Entrada/Salida con archivos: Un pequeño tutorial

En el capítulo 6 hemos visto, de forma general, algunas de las funciones más importantes proporcionadas por la librería estándar de C. Entre otras funciones se describieron las rutinas para realizar entrada/salida con archivos. En esta sección vamos a ver cómo utilizar ese conjunto de funciones para leer y escribir en archivos. Dado que ya hemos explicado el funcionamiento de cada una de las funciones necesarias nos centraremos en describir el procedimiento.

Los pasos fundamentales a la hora de operar con ficheros son los siguientes:

1. Declarar una variable tipo *flujo*, que representará el fichero.
2. Abrir el fichero y asociar la variable con ese fichero.
3. Leer/Escribir en el fichero.
4. Cerrar el fichero.

Pasaremos a describir cada uno de los pasos más detalladamente

#### 7.1.1. Variables de *flujo*

La librería estándar de C tiene definido un tipo de datos, `FILE *` que representa un *flujo* de bytes. Asociado a este flujo puede estar un archivo, una posición de memoria, el teclado, etc... La declaración:

```
FILE *fich;
```

Declara que la variable `fich` representará un flujo de datos, que luego asociaremos.

#### 7.1.2. Abrir el fichero

Una vez que tenemos declarada una variable de tipo `FILE *` tenemos que asociarla con el fichero que queremos abrir. Esto se hace mediante la llamada `fopen`.

Como se comenta en 6.3.2, `fopen` admite varios *modos* de apertura de ficheros. Si quisiéramos abrir el fichero para lectura (esto es, leer los datos que contiene y no modificarlo), utilizaríamos `fopen` de la siguiente manera:

```
fich = fopen( "fichero.txt", "r" );
```

Si en cambio quisiéramos crear un nuevo fichero haríamos lo siguiente:

```
fich = fopen( "fichero.txt", "w" );
```

Por último es posible que necesitemos añadir datos al final de un fichero ya existente:

```
fich = fopen( "fichero.txt", "a" );
```

#### 7.1.3. Leer y escribir en un fichero

Ya vimos en el capítulo 6 algunas primitivas de entrada/salida que ofrece la librería estándar. Veamos como se usan en la práctica.

## Lectura

Leer un archivo que tenga un formato determinado es una tarea fácil utilizando la rutina `fscanf`, que funciona de forma análoga a `scanf`.

Supongamos que queremos leer una línea del fichero `notas.txt` que contiene un listado de notas de alumnos con el siguiente formato:

```
Nombre Apellido1 Apellido2 notaParcial1 NotaParcial2
```

El fragmento de código que realizaría esta lectura sería el siguiente:

```
FILE *fich;
char nombre[10], apellido1[10], apellido2[10];
float nota1, nota2;

fich = fopen( "notas.txt", "r" );
fscanf( fich, "%s %s %s %f %f\n", nombre, apellido1, apellido2, &nota1, &nota2 );
```



**Nota:** Tienes que tener en cuenta que la variable `fich` funciona como un *apuntador* al archivo. Cuando se realiza una lectura este apuntador se desplaza de forma que los datos leídos quedan *por detrás de él*. En la práctica esto quiere decir que para volver a leer unos datos que ya has leído previamente tienes que recolocar este puntero, utilizando la rutina `fseek()`

Una necesidad común a la hora de leer de un archivo consiste en saber cuando hemos llegado al final del archivo. Esto se realiza con la rutina `feof()`, que devuelve un valor distinto de cero cuando hemos llegado al final del archivo. El siguiente ejemplo lee todas las líneas del archivo `notas.txt`, imprimiendo por pantalla los datos:



### Ejemplo

```
1  #include <stdio.h>
2
3
4  int main (void) {
5      FILE *fich;
6      char nombre[10], apellido1[10], apellido2[10];
7      float nota1, nota2;
8
9      /* Abrimos el fichero */
10     fich = fopen( "notas.txt", "r" );
11
12
13     /* Leemos mientras halla datos */
14     do {
15         fscanf( fich, "%s %s %s %f %f\n", nombre,
16                 apellido1, apellido2, &nota1, &nota2 );
17         printf( "Alumno: %s %s %s. Notas: %f %f\n", nombre,
18                 apellido1, apellido2, nota1, nota2 );
19
20     } while ( !feof( fich ) );
21
22     fclose( fich );
23 }
24
```

Otra opción a la hora de leer de un archivo es leer un número determinado de caracteres y almacenarlos en un *buffer* para posterior proceso. Esto se puede realizar con la rutina `fgets`.

## Escritura

Para escribir sobre un archivo tenemos disponibles las siguientes primitivas:

- `fprintf`: Escritura con formato. Funcionamiento similar a `printf`
- `fputs`: Escribe un *buffer* de caracteres en archivo especificado



**Nota:** Obviamente para poder escribir sobre un archivo tenemos que abrir el mismo en modo escritura

### 7.1.4. Cerrar el fichero

Una vez que hemos terminado de operar con el fichero hay que realizar una operación de **cierre** sobre la variable asociada, utilizando la rutina `fclose()`. Una vez que hemos cerrado un fichero no podremos realizar ninguna operación de lectura/escritura sin antes volver a abrirlo.

### 7.1.5. Ejemplo

El siguiente ejemplo utiliza todas las operaciones vistas hasta ahora para analizar los datos del fichero `notas.txt` (que contiene notas de alumnos, en el formato especificado anteriormente), escribiendo los resultados en el archivo `notas_finales.txt`



## Ejemplo

```

1  #include <stdio.h>
2
3
4  int main (void) {
5      FILE *fich_lect;
6      FILE *fich_escr;
7      char nombre[10], apellido1[10], apellido2[10];
8      float nota1, nota2, nota_media;
9
10     /* Abrimos los ficheros */
11     fich_lect = fopen( "notas.txt", "r" ); /* Lectura */
12     fich_escr = fopen( "notas_finales.txt", "w" ); /* Escritura */
13
14     /* Una bonita cabecera */
15     fputs( "Notas Finales\n-----\n", fich_escr );
16
17     /* Leemos mientras halla datos */
18     do {
19         fscanf( fich_lect, "%s %s %s %f %f\n", nombre,
20             apellido1, apellido2, &nota1, &nota2 );
21
22         /* Escribimos las notas del alumno */
23         fprintf( fich_escr, "Alumno: %s %s, %s\n",
24             apellido1, apellido2, nombre );
25         fprintf( fich_escr, "Nota primer parcial: %f\n", nota1 );
26         fprintf( fich_escr, "Nota segundo parcial: %f\n", nota2 );
27         nota_media = (nota1+nota2)/2;
28         fprintf( fich_escr, "Nota media: %f\n", nota_media );
29         fprintf( fich_escr, "Aprueba?: " );
30         if (nota_media >= 5.0)
31             fprintf( fich_escr, "SI" );
32         else
33             fprintf( fich_escr, "NO" );
34         fprintf( fich_escr, "\n\n" );
35
36     } while (!feof( fich_lect ) );
37
38     fclose( fich_lect );
39     fclose( fich_escr );
40 }
41

```

## 7.2. Línea de comandos

### 7.2.1. Introducción

Seguramente habrás tenido que ejecutar alguna vez un programa de consola que requiera *argumentos en la línea de comandos*, como por ejemplo `ls -l /` (linux) o `dir /p` (DOS/Windows).

Los argumentos de la línea de comandos no son más que información adicional que se le pasa al programa a la hora de ejecutarse. En los ejemplos anteriores esta información adicional serían las cadenas `-l` y `/p`. En esta sección vamos a ver cómo acceder a esta información desde nuestros programas en C.

### 7.2.2. El prototipo de la función `main`

Hasta ahora el prototipo de la función `main` que hemos utilizado es el siguiente:

```
int main( void );
```

Es decir, nuestra rutina `main` no admitía ningún parámetro, lo cual parece lógico, ya que no somos nosotros quienes llamamos a esta rutina (lo hace el sistema operativo cuando ejecutamos el programa). Sin embargo existe otro posible prototipo para esta función:

```
int main( int argc, char **argv );
```

Como habrás observado hemos cambiado los argumentos que recibe la función (y por tanto el programa). Estos dos nuevos parámetros no son más que información acerca de la línea de comandos con la que se ha ejecutado el programa. Más concretamente:

- `int argc`: Este parámetro contiene el número de argumentos con los que el programa ha sido llamado. El propio programa se considera también un argumento el primero, como veremos más adelante.
- `char **argv`: Array de punteros a punteros de cadenas de caracteres. Esto quiere decir que `argv[0]` será un puntero a una cadena de caracteres que conformará el primer argumento del programa.

Antes de seguir profundizando en el tema veamos algunos ejemplos:

```
ls -l --color /
```

Índice de <code>argv</code>	Contenido de <code>argv</code>
0	<code>ls</code>
1	<code>-l</code>
2	<code>--color</code>
3	<code>/</code>

Cuando hablábamos del parámetro `argc` decíamos que contenía el número de argumentos que se pasan a `main`. Hemos de tener en cuenta que todo programa C recibe al menos un argumento, y este es el nombre del ejecutable que contiene al programa, como se puede comprobar fácilmente mediante el siguiente programa:



#### Ejemplo

```
1  #include <stdio.h>
2
3
4  int main (int argc, char **argv) {
5
6      printf( "argc vale: %d\nargv[0] es: %s\n", argc, argv[0] );
7
8      return 0;
9  }
```

Si ejecutamos el programa sin argumentos obtenemos la siguiente salida:

```
argc vale: 1
argv[0] es: ejemplo_argv0.exe
```

Por lo tanto al manejar los argumentos de la línea tienes que tener en cuenta que `argc` siempre será **mayor o igual que uno**, y `argv[0]` contendrá siempre la ruta al fichero ejecutable que contiene tu programa.

Veamos por último un ejemplo de un programa que actúa de forma diferente según los argumentos de línea de comandos que le pasemos:



### Ejemplo

```

1  #include <stdio.h>
2
3
4  int main (int argc, char **argv) {
5      int i;
6
7      if (argc == 1){
8          /* Si no nos han llamado con argumentos salimos
9           * dando un mensaje de error
10         */
11         printf( "Error: Número insuficiente de argumentos\n" );
12         printf( "Uso: %s argumentos\n", argv[0] );
13
14         return 1; /* Devolvemos código de error 1 */
15     }
16
17     for (i=1; i < argc; i++) {
18         printf( "Argumento número %d: %s\n", i, argv[i] );
19
20         /* Si nos dan la opcion --saluda, entonces saludamos */
21         if (strcmp( argv[i], "--saluda" )==0) {
22             printf( "Hola!\n" );
23         }
24
25     }
26
27     return 0;
28 }

```

¿Serías capaz de saber como se comporta este programa sin ejecutarlo previamente?

## 7.3. Punteros a funciones

### 7.3.1. El concepto

Esta es quizá una de las funcionalidades que menos conoce el programador que viene de otros lenguajes, pero que puede resultar muy útil para resolver problemas de “orden superior” con una simpleza y elegancia no muy corrientes en el paradigma imperativo.

Un puntero a función es una variable que guarda la posición de memoria de una función. Por ejemplo:

```
void (*funcion) (void)
```

Es un puntero a una función sin argumentos, ni valor de retorno.

### 7.3.2. ¿Para qué sirven?

Como ya vimos anteriormente, un puntero “corriente” nos permitía entre otras cosas: utilizar estructuras de datos dinámicas, pasar variables por referencia en vez de por valor.

Un puntero a función nos permite hacer cosas más “divertidas”. Podemos cargar librerías de forma dinámica, hacer que nuestro código se automodifique en tiempo de ejecución, programar rutinas de orden superior, e implementar otra serie de funcionalidades que amenizan nuestra tarea como desarrolladores.

### 7.3.3. Ejemplo de orden superior



**Nota:** Cuando hablamos de orden superior nos referimos a la capacidad que tiene un lenguaje para operar con funciones como si variables comunes se tratara. Con orden superior se puede, por ejemplo, pasar una función como argumento a otra función.

El siguiente ejemplo hace uso de la función `qsort()`. Dicha función reordena listas de elementos de cualquier tipo. La lista debe ser un vector de elementos de un tamaño fijo. Por ejemplo podemos reordenar una lista de estructuras de datos, o una simple lista de enteros. El algoritmo quicksort es siempre el mismo. Lo único que diferencia un caso de otro es el “criterio” de ordenación de la lista. En un caso hay que comparar dos enteros, y en el otro hay que utilizar un criterio de comparación adaptado a la estructura de datos. Si la estructura representa la lista de empleados de una empresa, habrá que preguntarse si queremos ordenar por nombre de empleado, por salario o por cualquier otro criterio variopinto.

“El criterio” de ordenación es una función que debe utilizar `qsort()` para saber cómo ordenar las listas. Por ese motivo, `qsort()` debe recibir un puntero a la función que compare los elementos de la lista.



### Ejemplo

```
1
2 #include <stdlib.h>
3 #include <stdio.h>
4
5 #define MAXELEM 5
6
7 // Función que compara dos enteros.
8 int compara_enteros(long *a, long *b)
9 {
10
11     if (*a > *b)
12         return -1;
13     else if (*a == *b)
14         return 0;
15     else
16         return 1;
17 }
18
19 //Función que imprime los elementos de un vector.
20 //Para ello necesita saber cuántos y de qué tamaño, así como cómo imprimirlos.
21 void showlist(void *lista, int nelem, int size, void (*PRINT)(void *) )
22 {
23     int i;
24     for (i=0; i < nelem; i++)
25         PRINT(lista+i*size);
26     printf("\n");
27 }
28
29 //Sin comentarios
30 void imprime_int(int *a)
31 {
32     printf("%d\t", *a);
33 }
34
35 int main(){
36     int i;
37     unsigned long lista_num[MAXELEM]; //Declaro la lista.
38     for (i=0; i<MAXELEM;i++) //Relleno la lista de manera ``aleatoria``
39         lista_num[i] = random() % 400;
40
41     printf("--Lista de enteros DESORDENADA-----\n");
42     showlist(lista_num,MAXELEM,sizeof(long),imprime_int);
43
44     // Da comienzo la fiesta de orden superior
45     qsort(lista_num, MAXELEM, 4, (int (*)(const void *,
46                                     const void *))compara_enteros);
47
48     printf("--Lista de enteros ORDENADA con qsort---\n");
49     showlist(lista_num,MAXELEM,sizeof(int),imprime_int);
50
51     return 0;
52 }
```

### 7.3.4. Ejemplo de código mutante

El siguiente programa muestra un programa que se reescribe en tiempo de ejecución. Para ello primero pide 100 Bytes de memoria, después escribe en esa memoria las instrucciones correspondientes a una función, y finalmente ejecuta la función con distintas variantes.



#### Ejemplo

```

1  /* Ejemplo de código automodificable */
2
3  #include <stdio.h>
4  #include <unistd.h>
5  #include <stdlib.h>
6
7  int main(){
8      char *memoria;
9      int (*pfuncion) ();
10
11     memoria = (char*)malloc(100); /* Pedimos memoria */
12     memoria[0] = 0x55;
13     memoria[1] = 0x89;
14     memoria[2] = 0xE5;
15     memoria[3] = 0xB8; // mov eax <= 0x10 valor de retorno
16     memoria[4] = 0x10; // 0x10
17     memoria[5] = 0x00;
18     memoria[6] = 0x00;
19     memoria[7] = 0x00;
20     memoria[8] = 0x5D; // ret
21     memoria[9] = 0xC3;
22
23     pfuncion=(int (*)())memoria; // un cast feliz!
24     printf("La funcion devuelve: %X\n",pfuncion());
25
26     memoria[4] = 0x20;
27     printf("La funcion devuelve: %X\n",pfuncion());
28
29     printf("Ahora un segfault para divertirnos un rato!Dadme un segundo\n");
30     sleep (1);
31     memoria[0]= 0x00;
32     memoria[1]= 0x00;
33     memoria[2]= 0x00;
34     memoria[3]= 0x00;
35
36     printf("La funcion devuelve: %X\n",pfuncion());
37 }

```

El anterior ejemplo rellena unas posiciones de memoria con código máquina. En concreto, el código máquina es equivalente a:

```

int imprime()
{
    return 0x10;
}

```

Que en ensamblador de Intel x86 corresponde a:

```

imprime:
    pushl    %ebp
    movl    %esp, %ebp

```

```

movl    $$10, %eax
popl    %ebp
ret

```

y en binario Intel x86

```

memoria[0] = 0x55;
memoria[1] = 0x89;
memoria[2] = 0xE5;
memoria[3] = 0xB8; // mov eax <= 0x10 valor de retorno
memoria[4] = 0x10; // 0x10
memoria[5] = 0x00;
memoria[6] = 0x00;
memoria[7] = 0x00;
memoria[8] = 0x5D; // ret
memoria[9] = 0xC3;

```

Para obtener el código equivalente a un programa de C en ensamblador se ha empleado:

```
gcc -S fichero.c -o fichero.s # generar ensamblador
```

Para obtener el código máquina se ha utilizado el ensamblador:

```
as fichero.s -o fichero.o # generar binario
```

Y las herramientas de visualización hexadecimal

```
hexedit fichero.o
hexdump -C fichero.o
```

## 7.4. Gestión de bibliotecas de funciones

### 7.4.1. ¿Qué es una biblioteca?

**DEFINICIÓN: Biblioteca:** *Consiste en un archivo binario que almacena el código compilado de funciones.*

El programa básico en Unix/Linux para gestionar bibliotecas es *ar* (también *ranlib*) y sus funciones básicas son crear, modificar y extraer funciones. Las bibliotecas más importantes que se utilizan en el desarrollo de aplicaciones C son *libc.a*, *libm.a*, etc...

### 7.4.2. ¿Para qué necesito una biblioteca?

Las bibliotecas permiten almacenar en un mismo archivo el código de muchas funciones. Las ventajas de esta alternativa saltan a la vista:

- Cuando queramos utilizar una función de la biblioteca no tendremos que buscar el código y compilarlo, sino simplemente decirle al compilador dónde puede encontrar la biblioteca
- Ayudan a la reutilización de código, permitiendo que varios programas compartan porciones de código.
- Promueven una programación más modular.

### 7.4.3. Bibliotecas en Unix/Linux: *ar*

La sintaxis de *ar* es la siguiente:

```
ar [-] [miembro] biblioteca [ficheros]
```

Comentaremos las opciones más importantes del *ar*:

- *d*: Borrar miembros de la biblioteca.
- *m*: Mover un miembro de la biblioteca.

- p: Imprimir un miembro de la biblioteca en el fichero estándar de salida.
- q: Añadido rápido.
- r: Reemplazar ficheros en la biblioteca.
- t: Mostrar una tabla con el contenido de la biblioteca.
- x: Extraer miembros de la biblioteca.

#### 7.4.4. Ejemplo

Imaginemos que tenemos una aplicación para gestionar matrices, y que tenemos una biblioteca en donde están incluidos ya varios ficheros objeto llamada *matriz.a*. Si ahora queremos darle mayor funcionalidad a nuestra biblioteca de matrices y queremos añadir una función que por ejemplo haga la inversa de una matriz, implementamos el fichero *inver.c*, lo compilamos con:

```
$ gcc -c inver.c
```

Esto nos crea el código objeto, y ahora para añadir esto a la biblioteca hacemos:

```
$ ar r matriz.a inver.o
```

Situándonos en el path donde se encuentra la biblioteca. Si el fichero *inver.o* se encuentra en otro directorio basta con hacer:

```
$ ar r matriz.a /home/micuenta/inver.o (por ejemplo)
```

Para visualizar el contenido de una biblioteca:

```
$ ar t matriz.a
inver.o
multi.o
sum.o
```

Si queremos más información:

```
$ ar tv matriz.a
rw-r--r-- 402/ 6 875 Oct 26 14:43 2003 inver.o
etc...
```

Para extraer un módulo de la biblioteca:

```
$ ar xv matriz.a inver.o
```

## 7.5. Llamadas al sistema: POSIX

### 7.5.1. ¿Qué es POSIX?

POSIX<sup>1</sup> define un estándar de llamadas al sistema operativo. La librería estándar de C define unas funciones que deben estar en cualquier entorno de desarrollo de C. POSIX Define un estándar IEEE de funciones que requieren “ayuda” del sistema operativo. En la siguiente sección mostramos algunas de ellas.

### 7.5.2. Llamadas POSIX para gestión de procesos

#### getpid

```
pid_t getpid(void);
```

Función que devuelve el identificador del proceso.

---

<sup>1</sup>Portable Operating System Interface

**getuid**

```
uid_t getuid(void);
```

Función que devuelve el identificador de usuario real.

**getgid**

```
gid_t getgid(void);
```

Función que devuelve el identificador de grupo real.

**fork**

```
pid_t fork();
```

Función que crea un proceso hijo. Devuelve 0 al proceso hijo y el pid del hijo al proceso padre. El proceso hijo creado es una copia exacta del padre, excepto que recibe un valor diferente de la llamada fork, 0 en el hijo, el pid del hijo en el padre. Devuelve un valor -1 en caso de no poder crearse el proceso hijo.

**Ejemplo**

```

1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main(){
5
6      pid_t pid_hijo; /* Identificador del
7                      * proceso que crearemos */
8
9      pid_hijo = fork(); /* Creamos un proceso hijo,
10                        * que es copia del padre */
11
12     switch(pid_hijo){
13         case -1: printf("Error en fork :(\n");
14                 break; /* Si no ponemos el break no sale del switch
15                        * y ejecutaria el codigo de abajo */
16
17         case 0: printf("Soy el proceso hijo. !HOLA!\n");
18                 break;
19
20         default: printf("Soy el proceso padre, el hijo que he
21                       creado es %d\n",pid_hijo);
22                  break;
23     }
24
25     return 0; /* Tanto el padre como el hijo
26              * devuelven 0 como valor de retorno */
27 }
```

**execvp**

```
int execvp(const char *fichero, const char *argv[]);
```

execvp es en realidad solo una función de la familia de funciones exec. Existen las funciones execl, execlp, execl, execlp, execl, execlp, pero la más sencilla y útil es execvp. El primer parámetro, \*fichero, es un puntero que apunta a la ruta del programa que queremos ejecutar, el segundo es un array de punteros a char que contiene los parámetros que queremos pasar a dicho programa (como si lo hicieramos desde la línea de comandos), siendo el primer parámetro el nombre del mismo programa, y el último NULL, con lo que decimos que no quedan más parámetros.

Estas funciones lo que hacen en realidad es cambiar completamente el programa que la ejecuta, el cual solo conserva su pid y su tabla de ficheros abiertos. Por lo tanto, si quisieramos que un programa nuestro ejecute un programa externo (un ls por ejemplo), podríamos crear un proceso hijo con `fork()` y hacer que el proceso hijo ejecute un `exec`, convirtiéndose así en otro programa.



**Nota:** Como `exec` reemplaza el código del programa que lo ejecuta, `exec` nunca retorna, por lo cual cualquier código siguiente a la llamada `exec` no ejecutara, a no ser que la llamada `exec` falle.

Veamos un ejemplo:



### Ejemplo

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(){
5
6      pid_t pid_hijo;
7      char *nombre = "ls"; /* Nombre del programa a ejecutar por el hijo */
8      char *mod = "-la"; /* Preparamos los modificadores a usar */
9      char *params[3]; /* y el "contenedor" de los parametros */
10
11     pid_hijo = fork(); /* Creamos un hijo */
12
13     switch(pid_hijo){
14         case -1: printf("Error en fork :(\n");
15                 break;
16         case 0: params[0]=nombre;
17                 params[1]=mod;
18                 params[2]=NULL;
19                 execvp (nombre,params); /* ahora el hijo es un ls -la :) */
20                 exit(-1); /* exec solo llega aqui si se produce un error,
21                             * por lo que devolvemos un codigo de que algo
22                             * ha ido mal */
23         default: waitpid(pid_hijo,NULL); /* esperamos a que el hijo acabe */
24                 break;
25     }
26     return 0;
27 }
28

```

### wait

```
pid_t wait(int *status);
```

Función que permite a un proceso padre esperar hasta que termine un proceso hijo cualquiera. Devuelve el identificador del proceso hijo y el estado de información del mismo.

### waitpid

```
pid_t waitpid(pid_t pid, int *status, int options);
```

Función que espera hasta termina el proceso pid.

**exit**

```
void exit(int status);
```

Función que finaliza la ejecución de un proceso indicando el estado de terminación del mismo.

**Ejemplo**

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void comprobar_paridad(){
5
6      int useless;
7
8      useless = 1; /* Por ejemplo */
9
10     if (useless % 2)
11         exit(-1); /* Podemos salir desde cualquier
12                    * sitio con exit, incluso desde una
13                    * funcion */
14
15     }
16 int main(){
17
18     comprobar_paridad();
19     return 0; /* Podemos salir con un return desde main también */
20     exit(0); /* Esto seria equivalente :) */
21 }
```

**kill**

```
int kill(pid_t pid, int sig);
```

Función que envía al proceso pid la señal sig.

**pause**

```
int pause(void);
```

Función que bloquea al proceso hasta la recepción de una señal.

**sleep**

```
int sleep(unsigned int seconds);
```

Función que hace que el proceso despierte cuando ha transcurrido el tiempo establecido o cuando se recibe una señal.

Un ejemplo que usa varias llamadas POSIX:



### Ejemplo

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/types.h>
4  #include <signal.h>
5
6  int main(){
7
8      pid_t pid_hijo;
9      pid_t pid_devuelto;
10     char *nombre = "sleep"; /* Nombre del programa a ejecutar por el hijo */
11     char *mod = "10"; /* Preparamos los modificadores a usar */
12     char *params[3]; /* y el "contenedor" de los parametros */
13
14     pid_hijo = fork(); /* Creamos un hijo */
15
16     switch(pid_hijo){
17         case -1: printf("Error en fork :(\n");
18                 break;
19         case 0: params[0]=nombre;
20                 params[1]=mod;
21                 params[2]=NULL;
22                 execvp (nombre,params); /* el hijo duerme 10 segundos
23                                     * (ejecuta sleep 10) */
24                 exit(-1); /* exec solo llega aqui si se produce un error,
25                             * por lo que devolvemos un codigo de que algo
26                             * ha ido mal */
27         default:pid_devuelto = wait(); /* Esperamos un poco */
28                 if (pid_devuelto != pid_hijo)
29                     kill(pid_hijo, SIGKILL); /* Mandamos al hijo una
30                                             * señal de que muera
31                                             * (tarda demasiado ;) */
32                 break;
33     }
34     return 0;
35 }
36

```

### 7.5.3. Llamadas POSIX para la gestión de memoria

#### brk

```
size = brk (addr);
```

Función que aumenta el tamaño de una región de memoria. Coloca, si es posible, el fin del heap en la dirección `addr`. Supone que aumenta o disminuye el heap del segmento de datos.

#### dlopen

```
void *dlopen (const char *filename, int flag);
```

Función que monta una biblioteca cuyo nombre se especifica en la cadena `filename`. El parámetro `flag` puede tener los valores `RTLD_NOW` o `RTLD_LAZY`.

#### dlclose

```
int dlclose (void *handle);
```

Función que elimina de memoria una biblioteca. Solamente la elimina cuando no queda ningún proceso usando la biblioteca.

### 7.5.4. Llamadas POSIX para la entrada/salida y el sistema de ficheros

#### open

```
int open(char *name, int flags, mode_t mode);
```

Función que abre un fichero, devolviendo un descriptor de fichero o -1 si se produce un error. Las opciones de apertura son:

O\_RDONLY: Sólo lectura.

O\_WRONLY: Sólo escritura.

O\_RDWR: Lectura y escritura.

O\_APPEND: EL puntero de acceso se desplaza al final del fichero abierto.

O\_CREAT: Si existe no tiene efecto. Si no existe lo crea.

O\_TRUNC: Trunca si se abre para escritura. Los permisos del fichero vienen indicados por mode.

#### creat

```
int creat(char *name, mode_t mode);
```

Función que crea un fichero, devuelve un descriptor de fichero o -1 si se produce un error. El fichero se abre sólo para lectura. Si no existe se crea un fichero vacío.

#### read

```
ssize_t read(int fd, void *buf, size_t n_bytes);
```

Función que lee de un fichero (descriptor de fichero), devuelve el número de bytes leídos o -1 si se produce un error. Transfiere n\_bytes como máximo. Puede leer menos datos de los solicitados si se rebasa el fin de fichero o se interrumpe por una señal.

#### write

```
ssize_t write(int fd, void *buf, size_t n_bytes);
```

Función que escribe en un fichero (descriptor de fichero), devuelve el número de bytes escritos o -1 si se produce un error. Transfiere n\_bytes como máximo. Puede escribir menos datos de los solicitados si se rebasa el tamaño máximo de un fichero o se interrumpe por una señal.

#### close

```
int close(int fd);
```

Función que cierra un descriptor de fichero, devuelve 0 o -1 si se produce un error. El proceso pierde la asociación entre el descriptor y el fichero.

#### dup

```
int dup(int fd);
```

Función que duplica un descriptor de fichero, devuelve un descriptor de fichero que comparte todas las propiedades del fd o -1 si se produce un error.

#### opendir

```
DIR *opendir(char *dirname);
```

Función que abre un directorio, devuelve un puntero para utilizarse en `readdir()` o `closedir()`. Si se produce un error devuelve NULL.

**readdir**

```
struct dirent *readdir(DIR *dirp);
```

Función que realiza la lectura de entradas de un directorio, devuelve un puntero a un objeto del tipo struct dirent que representa una entrada de directorio o NULL si se produce un error.

**closedir**

```
int closedir(DIR *dirp);
```

Función que cierra un directorio, devuelve un cero o un -1 si se produce un error.

**mkdir**

```
int mkdir(char *name, mode_t mode);
```

Función que crea un directorio de nombre name, devuelve un cero o un -1 si se produce un error.

**rmdir**

```
int rmdir(char *name);
```

Función que borra un directorio de nombre name si está vacío. Si no está vacío no se borra. Devuelve un cero o un -1 si se produce un error.

**link**

```
int link(char *existing, char *new);
```

Función que crea una entrada de directorio. Crea un nuevo enlace físico para un fichero existente. Devuelve cero o un -1 si se produce un error. *existing* no debe ser el nombre de un directorio salvo que se tenga privilegio suficiente y la implementación soporte el enlace de directorios.

**symlink**

```
int symlink(char *oldpath, char *newpath);
```

Función que crea una entrada de directorio. Crea un nuevo enlace simbólico para un fichero existente. Devuelve cero o un -1 si se produce un error.

**unlink**

```
int unlink(char *name);
```

Función que elimina la entrada de directorio name. Devuelve cero o un -1 si se produce un error.

**chdir**

```
int chdir(char *name);
```

Función que modifica el directorio actual, aquel a partir del cual se forman los nombres relativos. Devuelve cero o un -1 si se produce un error.

**rename**

```
int rename(char *old, char *new);
```

Función que cambia el nombre del fichero *old* por *new*, devuelve cero o un -1 si se produce un error.

**umask**

```
mode_t umask(mode_t cmask);
```

Función que asigna la máscara de creación de ficheros del proceso que la invoca. Devuelve un cero o un -1 si se produce un error.

**chmod**

```
int chmod(char *name, mode_t mode);
```

Función que modifica los bits de permiso y los bits SETUID y SETGID del fichero. Devuelve un cero o un -1 si se produce un error. Sólo el propietario del fichero puede cambiar estos bits.

**chown**

```
int chown(char *name, uid_t owner, gid_t group);
```

Función que modifica el identificador de usuario y del grupo de un fichero. Devuelve un cero o un -1 si se produce un error.

**utime**

```
int utime(char *name, struct utimbuf *times);
```

Función que cambia las fechas de último acceso y última modificación según los valores de la estructura `struct utimbuf`. Devuelve cero o un -1 si se produce un error.



# Apéndice A

## Material de referencia

### A.1. Construcciones equivalentes ADA-C

#### A.1.1. Introducción

En esta sección se pretende poner de manifiesto las diferencias y similitudes entre los lenguajes Ada y C. Estas diferencias están motivadas por la orientación radicalmente diferente de estos lenguajes. Ada un lenguaje es orientado a objetos y fue diseñado para sistemas empotrados y críticos, donde es muy importante la corrección del código (vidas humanas pueden depender de ello). Es por ello que es muy "verboso", es decir, las instrucciones son muy largas y pesadas<sup>1</sup>. El lenguaje C sin embargo fue diseñado inicialmente para programar sistemas operativos, es decir, programación cerca de la máquina (aunque dada su enorme flexibilidad se use para todo tipo de tareas), por lo que su sintaxis es muy escueta y expresiva.

#### A.1.2. Sintaxis Ada vs. Sintaxis C

Al llegar a este punto, todo lo aquí expuesto resultará conocido, pero no está de más recapitular un poco. Para seguir un orden comenzaremos con la declaración de variables, tipos y las cabeceras "headers" que se incluyen en los programas. Luego se verán las sentencias de selección (`if`) y de iteración (`for` y `while`). Finalmente se abordará uno de los temas más candentes de C pero no tanto en Ada, los punteros. Finalizando con funciones y procedimientos.

#### Tipos, Variables y Headers

Como hemos visto, en C existen casi los mismos tipos básicos que en Ada. Decimos casi, porque hay que excluir el tipo boolean que en C no está predefinido<sup>2</sup>. Para "arreglarlo" C toma el valor 0 como True y los demás como False. Por otra parte Ada incluye muchos tipos avanzados o de propósito específico (números complejos, duraciones, números en coma fija...). No olvidemos que Ada es un lenguaje enorme, con muchísimos identificadores, mientras que C es todo lo contrario.

Otra diferencia está en la forma en que se identifican los tipos. Así pues tenemos:

Tipo en Ada	Tipo en C
Integer	int
Character	char
Natural	—
Positive	—

A la hora de declarar variables, en C se indica primero de qué tipo es (`int`, `char`, `float`...) y luego se escribe el nombre de la variable. En cambio en Ada primero se indica el nombre de la variable, seguido de un espacio, dos puntos y por último el tipo:

Variable en Ada	Variable en C
<code>numero : Integer;</code>	<code>int numero;</code>
<code>letra : Character;</code>	<code>char letra;</code>

Para las tuplas, ternas,... en Ada se usa la palabra reservada `record` mientras que en C se denominan estructuras y se definen con la palabra reservada `struct`. Tomemos por ejemplo la definición de un punto (en dos dimensiones) en los dos

<sup>1</sup>Una de las ideas en su desarrollo fue la de que el código se escribe una vez, pero se lee muchas

<sup>2</sup>Aunque siempre se puede emular mediante una directiva `define`

lenguajes:

Punto en Ada	Punto en C
<pre>type Tipo_Punto is record   x : Integer;   y : Integer; end record;</pre>	<pre>struct Tipo_Punto {   int x, y; }</pre>

En cuanto a los arrays en Ada primero se indica el nombre y luego se dice que es un array **is array**, después se indica la longitud y finalmente se indica el tipo de datos que conforman el array. En C es más corto, se indica el tipo de datos, luego el nombre y por último la longitud:

Array en Ada	Array en C
Nombre is array (desde..hasta) of Tipo_Dato;	tipo_dato nombre [longitud];
Whatever is array (1..25) of Integer;	int whatever[25];

Hay que tener en cuenta que en C el primer elemento del array es el que ocupa la posición 0 (array[0]) y para Ada en este caso es el elemento que ocupa la posición 1 (array(1)), hay que notar que a diferencia de C, Ada no obliga a que los arrays empiecen en 0, pueden empezar en cualquier entero, incluso negativo (se pueden incluso declarar con el rango de otra variable).

Para incluir las cabeceras:

Headers en Ada	Headers en C
with fichero; use fichero;	#include "fichero"
with Ada.Text_IO; use Ada.Text_IO;	#include <stdio.h>

## Sentencias de selección y de iteración

### Sentencia if

if en Ada	if en C
<pre>if condicion then   ejecutar_uno; else   ejecutar_dos; end if;</pre>	<pre>if (condicion) {   ejecutar_uno; } else {   ejecutar_dos; }</pre>

En C es necesario que la condición vaya siempre entre paréntesis y no se usa `then` ni `end if` sino llaves `{ }`.

### Bucle while

while en Ada	while en C
<pre>while condicion loop   ejecutar_sentencias; end loop;</pre>	<pre>while (condicion) {   ejecutar_sentencias; }</pre>

Al igual que en el **if**, en C los paréntesis en la condición son necesarios.

### Bucle for

for en Ada	for en C
<pre>for I in 1 .. n loop   ejecutar_sentencias; end loop;</pre>	<pre>for (i=0; i&lt;n; i++) {   ejecutar_sentencias; }</pre>

La principal diferencia, es que el iterador `i` en C tiene que estar definido con anterioridad. Por otra parte, en C para hacer que el índice decremente habría que poner `i--` y en Ada añadiríamos la palabra `reverse`.

### Punteros

Los punteros son muy importantes en C, y son la base de su increíble flexibilidad (y también de gran parte de los quebraderos de cabeza a la hora de depurar). En Ada se les da un uso mucho menos marcado. Por ejemplo, en C se usan para el paso de parámetros por referencia en la invocación de funciones mientras que en Ada se usa la palabra reservada `inout`. Los punteros se definen con el término `access` y en C usando `*`. Para acceder a la información apuntada en C hay que desreferenciar el puntero y en Ada acceder a `nombrepuntero.all`.

En Ada los punteros no apuntan a un tipo básico tal como `character` o `integer`, sino a un `record`, formado por (contenido, siguiente elemento). Aquí radica otra de las diferencias entre ambos lenguajes:

Ada	C
<code>nombre.all.campo</code>	<code>nombre-&gt;campo</code>

Algo importante que tiene C y que no tiene equivalente en Ada es la aritmética de punteros.

### Funciones

En C sólo hay funciones, no hay procedimientos propiamente dichos. Lo más parecido son las funciones que no devuelven nada (`void`). La diferencia es que los datos no se pueden pasar como `in`, `out` o `inout`, sino como variables normales (que “mueren” al final de la ejecución de la función). Si no queremos que esto ocurra, los parámetros se han de pasar por referencia (usando punteros)<sup>3</sup>.

Función en Ada	Función en C
<pre>Function nombre (parametros) RETURN tipo_devuelto IS ... end nombre;</pre>	<pre>tipo_devuelto nombre (parametros) { ... }</pre>

De nuevo, se ve el uso de `{ }` frente a `end`.

<sup>3</sup>En C++ se puede hacer de manera algo más cómoda.



# Bibliografía

- [Adá02] Yari Adán. *Introducción a los depuradores*. <http://acm.asoc.fi.upm.es/documentacion/seminario2002/linux/depuradores.pdf>, 2002.
- [BWK88] Dennis M. Ritchie Brian W. Kernighan. *The C Programming Language*. Prentice Hall, 1988.
- [Got98] Byron Gottfried. *Programación en C*. Mc Graw Hill, 1998.
- [Her01] Carlos Hernando. *Uso práctico de CVS*. <http://acm.asoc.fi.upm.es/~chernando/doc/cvs/>, 2001.
- [Mar94] David Marshall. *Programming in C*. <http://laurel.datsi.fi.upm.es/~ssoo/C/CE.html/>, 1994.
- [Sch01] Herbert Schildt. *C Manual de referencia*. Mc Graw Hill, 2001.
- [Sta] Richard Stallman. *GNU Make*. [http://www.gnu.org/manual/make/html\\_node/make.html](http://www.gnu.org/manual/make/html_node/make.html).