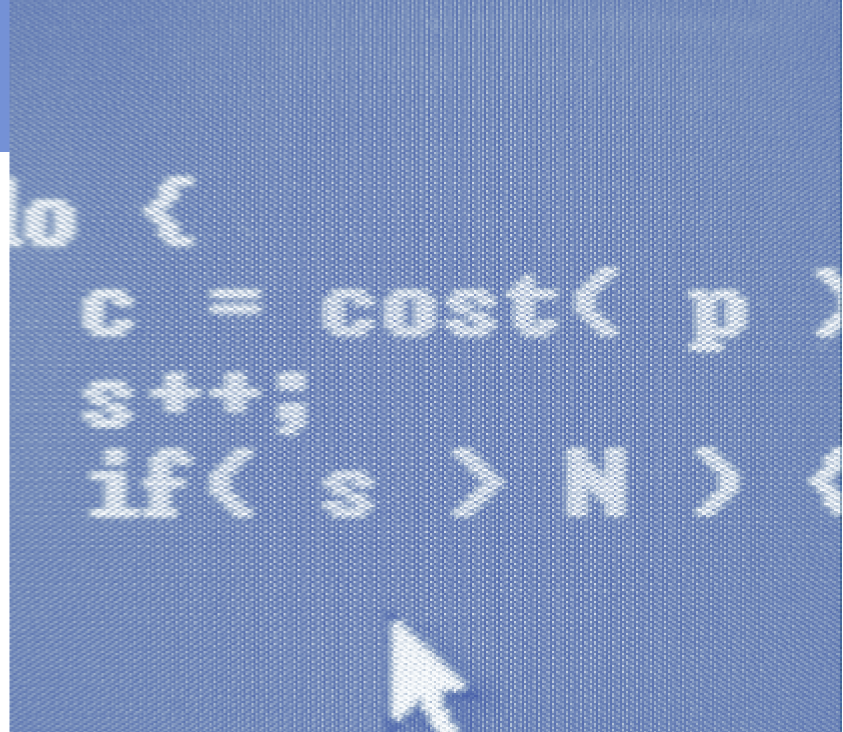
Software libre

Josep Anton Pérez López
Lluís Ribas i Xirgo

XP04/90793/00018



Introducción al desarrollo del software

UOC
Formación de Posgrado

David Megías Jiménez

Coordinador

Ingeniero en Informática por la UAB.

Magíster en Técnicas Avanzadas de Automatización de Procesos por la UAB.

Doctor en Informática por la UAB.

Profesor de los Estudios de Informática y Multimedia de la UOC.

Jordi Mas

Coordinador

Ingeniero de software en la empresa de código abierto Ximian, donde trabaja en la implementación del proyecto libre Mono. Como voluntario, colabora en el desarrollo del procesador de textos Abiword y en la ingeniería de las versiones en catalán del proyecto Mozilla y Gnome. Es también coordinador general de Softcatalà. Como consultor ha trabajado para empresas como Menta, Telépolis, Vodafone, Lotus, eresMas, Amena y Terra España.

Josep Anton Pérez López

Autor

Licenciado en Informática por la Universidad Autónoma de Barcelona.

Magíster en el programa Gráficos, Tratamiento de Imágenes y de Inteligencia Artificial, por la UAB.

Actualmente trabaja como profesor en un centro de educación secundaria.

Lluís Ribas i Xirgo

Autor

Licenciado en Ciencias-Informática y doctorado en Informática por la Universidad Autònoma de Barcelona (UAB).

Profesor titular del Departamento de Informática de la UAB. Consultor de los Estudios de Informática y Multimedia en la Universitat Oberta de Catalunya (UOC).

Primera edición: marzo 2004
© Fundació per a la Universitat Oberta de Catalunya
Av. Tibidabo, 39-43, 08035 Barcelona
Material realizado por Eureca Media, SL
© Autores: Josep Antoni Pérez López y Lluís Ribas i Xirgo
Depósito legal: B-7.600-2004
ISBN: 84-9788-119-2

Se garantiza permiso para copiar, distribuir y modificar este documento según los términos de la *GNU Free Documentation License, Version 1.2*o cualquiera posterior publicada por la Free Software Foundation, sin secciones invariantes ni textos de cubierta delantera o trasera. Se dispone de una copia de la licencia en el apartado "GNU Free Documentation License" de este documento.

Índice

Agradecimientos 9

1. Introducción a la programación 11
- 1.1. Introducción 11
- 1.2. Un poco de historia de C........ 12
- 1.3. Entorno de programación en GNU/C 16
  - 1.3.1. Un primer programa en C 17
  - 1.3.2. Estructura de un programa simple 21
- 1.4. La programación imperativa 23
  - 1.4.1. Tipos de datos básicos 24
  - 1.4.2. La asignación y la evaluación de expresiones 28
  - 1.4.3. Instrucciones de selección 33
  - 1.4.4. Funciones estándar de entrada y de salida 35
- 1.5. Resumen 40
- 1.6. Ejercicios de autoevaluación 41
  - 1.6.1. Solucionario 42

2. La programación estructurada 45
- 2.1. Introducción 45
- 2.2. Principios de la programación estructurada 48
- 2.3. Instrucciones iterativas 49
- 2.4. Procesamiento de secuencias de datos 52
  - 2.4.1. Esquemas algorítmicos: recorrido y búsqueda 53
  - 2.4.2. Filtros y tuberías 59
- 2.5. Depurado de programas 62
- 2.6. Estructuras de datos 66
- 2.7. Matrices 67
  - 2.7.1. Declaración 68
  - 2.7.2. Referencia 70
  - 2.7.3. Ejemplos 71
- 2.8. Estructuras heterogéneas 74
  - 2.8.1. Tuplas 74
  - 2.8.2. Variables de tipo múltiple 77

2.9. Tipos de datos abstractos 79
2.9.1. Definición de tipos de datos abstractos 80
2.9.2. Tipos enumerados 81
2.9.3. Ejemplo 82
2.10. Ficheros 84
2.10.1. Ficheros de flujo de bytes 84
2.10.2. Funciones estándar para ficheros 85
2.10.3. Ejemplo 90
2.11. Principios de la programación modular 92
2.12. Funciones 92
2.12.1. Declaración y definición 92
2.12.2. Ámbito de las variables 96
2.12.3. Parámetros por valor y por referencia 99
2.12.4. Ejemplo 101
2.13. Macros del preprocesador de C 103
2.14. Resumen 104
2.15. Ejercicios de autoevaluación 106
2.15.1. Solucionario 110

3. Programación avanzada en C. Desarrollo eficiente de aplicaciones 125
3.1. Introducción 125
3.2. Las variables dinámicas 127
3.3. Los apuntadores 129
3.3.1. Relación entre apuntadores y vectores 132
3.3.2. Referencias de funciones 135
3.4. Creación y destrucción de variables dinámicas 137
3.5. Tipos de datos dinámicos 139
3.5.1. Cadenas de caracteres 140
3.5.2. Listas y colas 145
3.6. Diseño descendente de programas 156
3.6.1. Descripción 157
3.6.2. Ejemplo 158
3.7. Tipos de datos abstractos y funciones asociadas 159
3.8. Ficheros de cabecera 166
3.8.1. Estructura 166
3.8.2. Ejemplo 169
3.9. Bibliotecas 172
3.9.1. Creación 172
3.9.2. Uso 173
3.9.3. Ejemplo 174
3.10. Herramienta *make* 175
3.10.1. Fichero *makefile* 176

3.11. Relación con el sistema operativo. Paso de parámetros a programas 179
3.12. Ejecución de funciones del sistema operativo 181
3.13. Gestión de procesos 183
3.13.1. Definición de proceso 184
3.13.2. Procesos permanentes 185
3.13.3. Procesos concurrentes 188
3.14. Hilos 189
3.14.1. Ejemplo 190
3.15. Procesos 193
3.15.1. Comunicación entre procesos 198
3.16. Resumen 202
3.17. Ejercicios de autoevaluación 204
3.17.1. Solucionario 211

4. Programación orientada a objetos en C++ 219
4.1. Introducción 219
4.2. De C a C++ 221
4.2.1. El primer programa en C++ 221
4.2.2. Entrada y salida de datos 223
4.2.3. Utilizando C++ como C 226
4.2.4. Las instrucciones básicas 227
4.2.5. Los tipos de datos 227
4.2.6. La declaración de variables y constantes 230
4.2.7. La gestión de variables dinámicas 231
4.2.8. Las funciones y sus parámetros 235
4.3. El paradigma de la programación orientada a objetos 240
4.3.1. Clases y objetos 242
4.3.2. Acceso a objetos 246
4.3.3. Constructores y destructores de objetos 250
4.3.4. Organización y uso de bibliotecas en C ++ 257
4.4. Diseño de programas orientados a objetos 261
4.4.1. La homonimia 262
4.4.2. La herencia simple 267
4.4.3. El polimorfismo 273
4.4.4. Operaciones avanzadas con herencia 279
4.4.5. Orientaciones para el análisis y diseño de programas 281
4.5. Resumen 285
4.6. Ejercicios de autoevaluación 286
4.6.1. Solucionario 287

ANOTACIONES

5. Programación en Java 309
5.1. Introducción 309
5.2. Origen de Java 312
5.3. Características generales de Java 313
5.4. El entorno de desarrollo de Java 317
5.4.1. La plataforma Java 318
5.4.2. Mi primer programa en Java 319
5.4.3. Las instrucciones básicas y los comentarios 320
5.5. Diferencias entre C++ y Java 321
5.5.1. Entrada/salida 321
5.5.2. El preprocesador 324
5.5.3. La declaración de variables y constantes . 325
5.5.4. Los tipos de datos 325
5.5.5. La gestión de variables dinámicas 326
5.5.6. Las funciones y el paso de parámetros 328
5.6. Las clases en Java 329
5.6.1. Declaración de objetos 330
5.6.2. Acceso a los objetos 331
5.6.3. Destrucción de objetos 332
5.6.4. Constructores de copia 333
5.6.5. Herencia simple y herencia múltiple 333
5.7. Herencia y polimorfismo 334
5.7.1. Las referencias `this` y `super` 334
5.7.2. La clase Object 334
5.7.3. Polimorfismo 335
5.7.4. Clases y métodos abstractos 335
5.7.5. Clases y métodos finales 336
5.7.6. Interfaces 337
5.7.7. Paquetes 339
5.7.8. El API (*applications programming interface*) de Java 340
5.8. El paradigma de la programación orientada a eventos 341
5.8.1. Los eventos en Java 342
5.9. Hilos de ejecución (*threads*) 344
5.9.1. Creación de hilos de ejecución 345
5.9.2. Ciclo de vida de los hilos de ejecución 348
5.10. Los *applets* 349
5.10.1. Ciclo de vida de los *applets* 350
5.10.2. Manera de incluir *applets* en una página HTML 351
5.10.3. Mi primer *applet* en Java 352
5.11. Programación de interfaces gráficas en Java 353
5.11.1. Las interfaces de usuario en Java 354

ANOTACIONES

5.11.2. Ejemplo de *applet* de Swing 355
5.12. Introducción a la información visual 356
5.13. Resumen 357
5.14. Ejercicios de autoevaluación 358
5.14.1. Solucionario 359

Glosario 373

Bibliografia 381

ANOTACIONES

Agradecimientos

Los autores agradecen a la Fundación para la Universitat Oberta de Catalunya (http://www.uoc.edu) la financiación de la primera edición de esta obra, enmarcada en el Máster Internacional en Software Libre ofrecido por la citada institución.

ANOTACIONES

1. Introducción a la programación

1.1. Introducción

En esta unidad se revisan los fundamentos de la programación y los conceptos básicos del lenguaje C. Se supone que el lector ya tiene conocimientos previos de programación bien en el lenguaje C, bien en algún otro lenguaje.

Por este motivo, se incide especialmente en la metodología de la programación y en los aspectos críticos del lenguaje C en lugar de hacer hincapié en los elementos más básicos de ambos aspectos.

El conocimiento profundo de un lenguaje de programación parte no sólo del entendimiento de su léxico, de su sintaxis y de su semántica, sino que además requiere la comprensión de los objetivos que motivaron su desarrollo. Así pues, en esta unidad se repasa la historia del lenguaje de programación C desde el prisma de la programación de los computadores.

Los programas descritos en un lenguaje de programación como C no pueden ser ejecutados directamente por ninguna máquina. Por tanto, es necesario disponer de herramientas (es decir, programas) que permitan obtener otros programas que estén descritos como una secuencia de órdenes que sí que pueda ejecutar directamente algún computador.

En este sentido, se describirá un entorno de desarrollo de software de libre acceso disponible tanto en plataformas Microsoft como GNU/Linux. Dado que las primeras requieren de un sistema operativo que no se basa en el software libre, la explicación se centrará en las segundas.

Nota

Una plataforma es, en este contexto, el conjunto formado por un tipo de ordenador y un sistema operativo.

El resto de la unidad se ocupará del lenguaje de programación C en lo que queda afectado por el paradigma de la programación imperativa y su modelo de ejecución. El modelo de ejecución trata de la

ANOTACIONES

forma como se llevan a cabo las instrucciones indicadas en un programa. En el paradigma de la programación imperativa, las instrucciones son órdenes que se llevan a cabo de forma inmediata para conseguir algún cambio en el estado del procesador y, en particular, para el almacenamiento de los resultados de los cálculos realizados en la ejecución de las instrucciones. Por este motivo, en los últimos apartados se incide en todo aquello que afecta a la evaluación de expresiones (es decir, al cálculo de los resultados de fórmulas), a la selección de la instrucción que hay que llevar a cabo y a la obtención de datos o a la producción de resultados.

En esta unidad, pues, se pretende que el lector alcance los objetivos siguientes:

1. Repasar los conceptos básicos de la programación y el modelo de ejecución de los programas.
2. Entender el paradigma fundamental de la programación imperativa.
3. Adquirir las nociones de C necesarias para el seguimiento del curso.
4. Saber utilizar un entorno de desarrollo de software libre para la programación en C (GNU/Linux y útiles GNU/C).

1.2. Un poco de historia de C

El lenguaje de programación C fue diseñado por Dennis Ritchie en los laboratorios Bell para desarrollar nuevas versiones del sistema operativo Unix, allá por el año 1972. De ahí, la fuerte relación entre el C y la máquina.

> **Nota**
>
> Como curiosidad cabe decir que bastaron unas trece mil líneas de código C (y un millar escaso en ensamblador) para programar el sistema operativo Unix del computador PDP-11.

El lenguaje ensamblador es aquel que tiene una correspondencia directa con el lenguaje máquina que entiende el procesador. En otras

ANOTACIONES

palabras, cada instrucción del lenguaje máquina se corresponde con una instrucción del lenguaje ensamblador.

Por el contrario, las instrucciones del lenguaje C pueden equivaler a pequeños programas en el lenguaje máquina, pero de frecuente uso en los algoritmos que se programan en los computadores. Es decir, se trata de un lenguaje en el que se emplean instrucciones que sólo puede procesar una máquina abstracta, que no existe en la realidad (el procesador real sólo entiende el lenguaje máquina). Por ello, se habla del C como lenguaje de alto nivel de abstracción y del ensamblador como lenguaje de bajo nivel.

Esta máquina abstracta se construye parcialmente mediante un conjunto de programas que se ocupan de gestionar la máquina real: el ***sistema operativo***. La otra parte se construye empleando un programa de traducción del lenguaje de alto nivel a lenguaje máquina. Estos programas se llaman ***compiladores*** si generan un código que puede ejecutarse directamente por el computador, o ***intérpretes*** si es necesaria su ejecución para llevar a cabo el programa descrito en un lenguaje de alto nivel.

Para el caso que nos ocupa, resulta de mucho interés que el código de los programas que constituyen el sistema operativo sea lo más independiente de la máquina posible. Únicamente de esta manera será viable adaptar un sistema operativo a cualquier computador de forma rápida y fiable.

Por otra parte, es necesario que el compilador del lenguaje de alto nivel sea extremadamente eficiente. Para ello, y dado los escasos recursos computacionales (fundamentalmente, capacidad de memoria y velocidad de ejecución) de los ordenadores de aquellos tiempos, se requiere que el lenguaje sea simple y permita una traducción muy ajustada a las características de los procesadores.

Éste fue el motivo de que en el origen de C estuviera el lenguaje denominado B, desarrollado por Ken Thompson para programar Unix para el PDP-7 en 1970. Evidentemente, esta versión del sistema operativo también incluyó una parte programada en ensamblador, pues había operaciones que no podían realizarse sino con la máquina real.

ANOTACIONES

Queda claro que se puede ver el lenguaje C como una versión posterior (y mejorada con inclusión de tipos de datos) del B. Más aún, el lenguaje B estaba basado en el BCPL. Éste fue un lenguaje desarrollado por Martín Richards en 1967, cuyo tipo de datos básico era la palabra *memoria*; es decir, la unidad de información en la que se divide la memoria de los computadores. De hecho, era un lenguaje ensamblador mejorado que requería de un compilador muy simple. A cambio, el programador tenía más control de la máquina real.

A pesar de su nacimiento como lenguaje de programación de sistemas operativos, y, por tanto, con la capacidad de expresar operaciones de bajo nivel, C es un lenguaje de programación de **propósito general**. Esto es, con él es posible programar algoritmos de aplicaciones (conjuntos de programas) de muy distintas características como, por ejemplo, software de contabilidad de empresas, manejo de bases de datos de reservas de aviones, gestión de flotas de transporte de mercancías, cálculos científicos, etcétera.

Bibliografía

La definición de las reglas sintácticas y semánticas del C aparece en la obra siguiente:

B.W. Kernighan; D.M. Ritchie (1978). *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall. Concretamente en el apéndice “C Reference Manual”.

La relativa simplicidad del lenguaje C por su escaso número de instrucciones permite que sus compiladores generen un código en lenguaje máquina muy eficiente y, además, lo convierten en un lenguaje fácilmente transportable de una máquina a otra.

Por otra parte, el repertorio de instrucciones de C posibilita realizar una programación estructurada de alto nivel de abstracción. Cosa que redunda en la programación sistemática, legible y de fácil mantenimiento.

ANOTACIONES

Esta simplicidad, no obstante, ha supuesto la necesidad de disponer para C de un conjunto de funciones muy completas que le confieren una gran potencia para desarrollar aplicaciones de todo tipo. Muchas de estas funciones son estándar y se encuentran disponibles en todos los compiladores de C.

Nota

Una función es una secuencia de instrucciones que se ejecuta de forma unitaria para llevar a cabo alguna tarea concreta. Por lo tanto, a mayor número de funciones ya programadas, menos código habrá que programar.

Las funciones estándar de C se recogen en una biblioteca: la biblioteca estándar de funciones. Así pues, cualquier programa puede emplear todas las funciones que requiera de la misma, puesto que todos los compiladores disponen de ella.

Finalmente, dada su dependencia de las funciones estándar, C promueve una programación modular en la que los propios programadores también pueden preparar funciones específicas en sus programas.

Todas estas características permitieron una gran difusión de C que se normalizó por la ANSI (American National Standards Association) en 1989 a partir de los trabajos de un comité creado en 1983 para "proporcionar una definición no ambigua e independiente de máquina del lenguaje". La segunda edición de Kernighan y Ritchie, que se publicó en 1988, refleja esta versión que se conoce como ANSI-C.

Nota

La versión original de C se conocería, a partir de entonces, como K&R C, es decir, como elC *de Kernighan y Ritchie*. De esta manera, se distinguen la versión original y la estandarizada por la ANSI.

El resto de la unidad se dedicará a explicar un entorno de desarrollo de programas en C y a repasar la sintaxis y la semántica de este lenguaje.

ANOTACIONES

1.3. Entorno de programación en GNU/C

En el desarrollo de software de libre acceso es importante emplear herramientas (programas) que también lo sean, pues uno de los principios de los programas de libre acceso es que su código pueda ser modificado por los usuarios.

El uso de código que pueda depender de software privado implica que esta posibilidad no exista y, por tanto, que no sea posible considerarlo como libre. Para evitar este problema, es necesario programar mediante software de libre acceso.

GNU (www.gnu.org) es un proyecto de desarrollo de software libre iniciado por Richard Stallman en el año 1984, que cuenta con el apoyo de la Fundación para el Software Libre (FSF).

GNU es un acrónimo recursivo (significa *GNU No es Unix*) para indicar que se trataba del software de acceso libre desarrollado sobre este sistema operativo, pero que no consistía en el sistema operativo. Aunque inicialmente el software desarrollado utilizara una plataforma Unix, que es un sistema operativo de propiedad, no se tardó en incorporar el *kernel* Linux como base de un sistema operativo independiente y completo: el GNU/Linux.

Nota

El kernel de un sistema operativo es el software que constituye su núcleo fundamental y, de ahí, su denominación (*kernel* significa, entre otras cosas, 'la parte esencial'). Este núcleo se ocupa, básicamente, de gestionar los recursos de la máquina para los programas que se ejecutan en ella.

El kernel Linux, compatible con el de Unix, fue desarrollado por Linus Torvalds en 1991 e incorporado como kernel del sistema operativo de GNU un año más tarde.

En todo caso, cabe hacer notar que todas las llamadas *distribuciones de Linux* son, de hecho, versiones del sistema operativo GNU/Linux que, por tanto, cuentan con software de GNU (editor de textos

ANOTACIONES

Emacs, compilador de C, etc.) y también con otro software libre como puede ser el procesador de textos TeX.

Para el desarrollo de programas libres se empleará, pues, un ordenador con el sistema operativo Linux y el compilador de C de GNU (gcc). Aunque se puede emplear cualquier editor de textos ASCII, parece lógico emplear Emacs. Un editor de textos ASCII es aquél que sólo emplea los caracteres definidos en la tabla ASCII para almacenar los textos. (En el Emacs, la representación de los textos se puede ajustar para distinguir fácilmente los comentarios del código del programa en C, por ejemplo.)

Aunque las explicaciones sobre el entorno de desarrollo de software que se realizarán a lo largo de esta unidad y las siguientes hacen referencia al software de GNU, es conveniente remarcar que también es posible emplear herramientas de software libre para Windows, por ejemplo.

En los apartados siguientes se verá cómo emplear el compilador `gcc` y se revisará muy sucintamente la organización del código de los programas en C.

1.3.1. Un primer programa en C

Un programa es un texto escrito en un lenguaje simple que permite expresar una serie de acciones sobre objetos (instrucciones) de forma no ambigua.

Antes de elaborar un programa, como en todo texto escrito, habremos de conocer las reglas del lenguaje para que aquello que se exprese sea correcto tanto léxica como sintácticamente.

Las normas del lenguaje de programación C se verán progresivamente a lo largo de las unidades correspondientes (las tres primeras).

Además, deberemos procurar que "eso" tenga sentido y exprese exactamente aquello que se desea que haga el programa. Por si no

ANOTACIONES

Nota

El símbolo del dólar ($) se emplea para indicar que el intérprete de comandos del sistema operativo del ordenador puede aceptar una nueva orden.

Nota

El nombre del fichero que contiene el programa en C tiene extensión ".c" para que sea fácil identificarlo como tal.

Ejemplo

No es lo mismo `printf` que `PrintF`.

fuera poco, habremos de cuidar el aspecto del texto de manera que sea posible captar su significado de forma rápida y clara. Aunque algunas veces se indicará alguna norma de estilo, normalmente éste se describirá implícitamente en los ejemplos.

Para escribir un programa en C, es suficiente con ejecutar el `emacs`:

```
$ emacs hola.c &
```

Ahora ya podemos escribir el siguiente programa en C:

```
#include <stdio.h>
main( )
{
 printf( ""Hola a todos! \n" );
} /* main */
```

Es muy importante tener presente que, en C (y también en C++ y en Java), **se distinguen mayúsculas de minúsculas**. Por lo tanto, el texto del programa tiene que ser exactamente como se ha mostrado a excepción del texto entre comillas y el que está entre los símbolos `/*` y `*/`.

El editor de textos `emacs` dispone de menús desplegables en la parte superior para la mayoría de operaciones y, por otra parte, acepta comandos introducidos mediante el teclado. A tal efecto, es necesario teclear también la tecla de control ("CTRL" o "C-") o la de carácter alternativo junto con la del carácter que indica la orden.

A modo de breve resumen, se describen algunos de los comandos más empleados en la siguiente tabla:

Tabla 1.

Comando	Secuencia	Explicación
Files → Find file	C-x, C-f	Abre un fichero. El fichero se copia en un *buffer* o área temporal para su edición.
Files → Save buffer	C-x, C-s	Guarda el contenido del *buffer* en el fichero asociado.

ANOTACIONES

Comando	Secuencia	Explicación
Files → Save buffer as	C-x, C-w	Escribe el contenido del *buffer* en el fichero que se le indique.
Files → Insert file	C-x, C-i	Inserta el contenido del fichero indicado en la posición del texto en el que se encuentre el cursor.
Files → Exit	C-x, C-c	Finaliza la ejecución de `emacs`.
(cursor movement)	C-a	Sitúa el cursor al principio de la línea.
(cursor movement)	C-e	Sitúa el cursor al final de la línea.
(line killing)	C-k	Elimina la línea (primero el contenido y luego el salto de línea).
Edit → Paste	C-y	Pega el último texto eliminado o copiado.
Edit → Copy	C-y, ..., C-y	Para copiar un trozo de texto, puede eliminarse primero y recuperarlo en la misma posición y, finalmente, en la posición de destino.
Edit → Cut	C-w	Elimina el texto desde la última marca hasta el cursor.
Edit → Undo	C-u	Deshace el último comando.

Nota

Para una mejor comprensión de los comandos se recomienda leer el manual de **emacs** o, en su caso, del editor que se haya elegido para escribir los programas.

Una vez editado y guardado el programa en C, hay que compilarlo para obtener un fichero binario (con ceros y unos) que contenga una versión del programa traducido a lenguaje máquina. Para ello, hay que emplear el compilador `gcc`:

```
$ gcc –c hola.c
```

Nota

En un tiempo `gcc` significó compilador de C de GNU, pero, dado que el compilador entiende también otros lenguajes, ha pasado a ser la colección de compiladores de GNU. Por este motivo, es necesario indicar en qué lenguaje se han escrito los programas mediante la extensión del nombre de los ficheros correspondientes. En este caso, con `hola.c`, empleará el compilador de C.

Con ello, se obtendrá un fichero (`hola.o`), denominado fichero objeto. Este archivo contiene ya el programa en lenguaje máquina derivado del programa con el código C, llamado también *código fuente*. Pero desgraciadamente aún no es posible ejecutar este programa, ya que requiere de una función (`printf`) que se encuentra en la biblioteca de funciones estándar de C.

Para obtener el código ejecutable del programa, será necesario enlazar (en inglés: *link*):

```
$ gcc hola.o –o hola
```

Como la biblioteca de funciones estándar está siempre en un lugar conocido por el compilador, no es necesario indicarlo en la línea de comandos. Eso sí, será necesario indicar en qué fichero se quiere el resultado (el fichero ejecutable) con la opción –o seguida del nombre deseado. En caso contrario, el resultado se obtiene en un fichero llamado "a.out".

Habitualmente, el proceso de compilación y enlazado se hacen directamente mediante:

```
$ gcc hola.c –o hola
```

Si el fichero fuente contuviese errores sintácticos, el compilador mostrará los mensajes de error correspondientes y deberemos corregirlos antes de repetir el procedimiento de compilación.

En caso de que todo haya ido bien, dispondremos de un programa ejecutable en un fichero llamado hola que nos saludará vehementemente cada vez que lo ejecutemos:

```
$ ./hola
Hola a todos!
$
```

Nota

Se debe indicar el camino de acceso al fichero ejecutable para que el intérprete de órdenes pueda localizarlo. Por motivos de seguridad, el directorio de trabajo no se incluye por omisión en el conjunto de caminos de búsqueda de ejecutables del intérprete de comandos.

Este procedimiento se repetirá para cada programa que realicemos en C en un entorno GNU.

ANOTACIONES

1.3.2. Estructura de un programa simple

En general, un programa en C debería estar organizado como sigue:

```
/* Fichero: nombre.c                                          */
/* Contenido: ejemplo de estructura de un programa en C       */
/* Autor: nombre del autor                                    */
/* Revisión: preliminar                                       */

/* COMANDOS DEL PREPROCESADOR                                 */
/* -inclusión de ficheros de cabeceras                        */
#include <stdio.h>
/* -definición de constantes simbólicas                       */
#define FALSO 0

/* FUNCIONES DEL PROGRAMADOR                                  */

main( ) /* Función principal:                                 */
{       /* El programa se empieza a ejecutar aquí             */
  ...   /* Cuerpo de la función principal                     */
} /* main */
```

En esta organización, las primeras líneas del fichero son comentarios que identifican su contenido, autor y versión. Esto es importante, pues hay que tener presente que el código fuente que creemos debe ser fácilmente utilizable y modificable por otras personas... ¡ y por nosotros mismos!

Nota

Los comentarios son textos libres que se escriben entre los dígrafos `/*` y `*/`.

Dada la simplicidad del C, muchas de las operaciones que se realizan en un programa son, en realidad, llamadas a funciones estándar. Así pues, para que el compilador conozca qué parámetros tienen y qué valores devuelven, es necesario incluir en el código de nuestro programa las declaraciones de las funciones que se emplearán. Para ello, se utiliza el comando `#include` del llamado *preprocesador de C*, que se ocupa de montar un fichero único de entrada para el compilador de C.

Los ficheros que contienen declaraciones de funciones externas a un determinado archivo fuente son llamados *ficheros de cabeceras*

ANOTACIONES

(*header files*). De ahí que se utilice la extensión ".h" para indicar su contenido.

Las cabeceras, en C, son la parte en la que se declara el nombre de una función, los parámetros que recibe y el tipo de dato que devuelve.

Tanto estos ficheros como el del código fuente de un programa pueden contener definiciones de constantes simbólicas. A continuación presentamos una muestra de estas definiciones:

```
#define VACIO          '\0'      /* El carácter ASCII NUL   */
#define NUMERO_OCTAL   0173      /* Un valor octal          */
#define MAX_USUARIOS   20
#define CODIGO_HEXA    0xf39b    /* Un valor hexadecimal    */
#define PI             3.1416    /* Un número real          */
#define PRECISION      1e-10     /* Otro número real        */
#define CADENA         "cadena de caracteres"
```

Estas constantes simbólicas son reemplazadas por su valor en el fichero final que se suministra al compilador de C. Es importante recalcar que su uso debe redundar en una mayor legibilidad del código y, por otra parte, en una facilidad de cambio del programa, cuando fuese necesario.

Cabe tener presente que las constantes numéricas enteras descritas en base 8, u octal, deben estar prefijadas por un 0 y las expresadas en base 16, o hexadecimal, por "0x"

Ejemplo

020 no es igual a 20, puesto que este último coincide con el valor veinte en decimal y el primero es un número expresado en base 8, cuya representación binaria es 010000, que equivale a 16, en base 10.

Finalmente, se incluirá el programa en el cuerpo de la función principal. Esta función debe estar presente en todo programa, pues la

ANOTACIONES

primera instrucción que contenga es la que se toma como punto inicial del programa y, por tanto, será la primera en ser ejecutada.

1.4. La programación imperativa

La programación consiste en la traducción de algoritmos a versiones en lenguajes de programación que puedan ser ejecutados directa o indirectamente por un ordenador.

La mayoría de algoritmos consisten en una secuencia de pasos que indican lo que hay que hacer. Estas instrucciones suelen ser de carácter imperativo, es decir, indican lo que hay que hacer de forma incondicional.

La programación de los algoritmos expresados en estos términos se denomina *programación imperativa*. Así pues, en este tipo de programas, cada instrucción implica realizar una determinada acción sobre su entorno, en este caso, en el computador en el que se ejecuta.

Para entender cómo se ejecuta una instrucción, es necesario ver cómo es el entorno en el que se lleva a cabo. La mayoría de los procesadores se organizan de manera que los datos y las instrucciones se encuentran en la memoria principal y la unidad central de procesamiento (CPU, de las siglas en inglés) es la que realiza el siguiente algoritmo para poder ejecutar el programa en memoria:

1. Leer de la memoria la instrucción que hay que ejecutar.
2. Leer de la memoria los datos necesarios para su ejecución.
3. Realizar el cálculo u operación indicada en la instrucción y, según la operación que se realice, grabar el resultado en la memoria.
4. Determinar cuál es la siguiente instrucción que hay que ejecutar.
5. Volver al primer paso.

La CPU hace referencia a las instrucciones y a los datos que pide a la memoria o a los resultados que quiere escribir mediante el número

ANOTACIONES

de posición que ocupan en la misma. Esta posición que ocupan los datos y las instrucciones se conoce como *dirección de memoria*.

En el nivel más bajo, cada dirección distinta de memoria es un único byte y los datos y las instrucciones se identifican por la dirección del primero de sus bytes. En este nivel, la CPU coincide con la CPU física de que dispone el computador.

En el nivel de la máquina abstracta que ejecuta C, se mantiene el hecho de que las referencias a los datos y a las instrucciones sea la dirección de la memoria física del ordenador, pero las instrucciones que puede ejecutar su CPU de alto nivel son más potentes que las que puede llevar a cabo la real.

Independientemente del nivel de abstracción en que se trabaje, la memoria es, de hecho, el entorno de la CPU. Cada instrucción realiza, en este modelo de ejecución, un cambio en el entorno: puede modificar algún dato en memoria y siempre implica determinar cuál es la dirección de la siguiente instrucción a ejecutar.

Dicho de otra manera: la ejecución de una instrucción supone un cambio en el estado del programa. Éste se compone de la dirección de la instrucción que se está ejecutando y del valor de los datos en memoria. Así pues, llevar a cabo una instrucción implica cambiar de estado el programa.

En los próximos apartados se describirán los tipos de datos básicos que puede emplear un programa en C y las instrucciones fundamentales para cambiar su estado: la asignación y la selección condicional de la instrucción siguiente. Finalmente, se verán las funciones estándar para tomar datos del exterior (del teclado) y para mostrarlos al exterior (a través de la pantalla).

1.4.1. Tipos de datos básicos

Los tipos de datos primitivos de un lenguaje son aquéllos cuyo tratamiento se puede realizar con las instrucciones del mismo lenguaje; es decir, que están soportados por el lenguaje de programación correspondiente.

ANOTACIONES

Compatibles con enteros

En C, los tipos de datos primitivos más comunes son los compatibles con enteros. La representación binaria de éstos no es codificada, sino que se corresponde con el valor numérico representado en base 2. Por tanto, se puede calcular su valor numérico en base 10 sumando los productos de los valores intrínsecos (0 o 1) de sus dígitos (bits) por sus valores posicionales ($2^{posición}$) correspondientes.

Se tratan bien como números naturales, o bien como representaciones de enteros en base 2, si pueden ser negativos. En este último caso, el bit más significativo (el de más a la izquierda) es siempre un 1 y el valor absoluto se obtiene restando el número natural representado del valor máximo representable con el mismo número de bits más 1.

En todo caso, es importante tener presente que el rango de valores de estos datos depende del número de bits que se emplee para su representación. Así pues, en la tabla siguiente se muestran los distintos tipos de datos compatibles con enteros en un computador de 32 bits con un sistema GNU.

Tabla 2.

Especificación	Número de bits	Rango de valores
`(signed) char`	8 (1 byte)	de –128 a +127
`unsigned char`	8 (1 byte)	de 0 a 255
`(signed) short (int)`	16 (2 bytes)	de –32.768 a +32.767
`unsigned short (int)`	16 (2 bytes)	de 0 a 65.535
`(signed) int`	32 (4 bytes)	de –2.147.483.648 a +2.147.483.647
`unsigned (int)`	32 (4 bytes)	de 0 a 4.294.967.295
`(signed) long (int)`	32 (4 bytes)	de –2.147.483.648 a +2.147.483.647
`unsigned long (int)`	32 (4 bytes)	de 0 a 4.294.967.295
`(signed) long long (int)`	64 (8 bytes)	de -2^{63} a $+(2^{63}-1) \approx \pm 9{,}2 \times 10^{18}$
`unsigned long long (int)`	64 (8 bytes)	de 0 a $2^{64}-1 \approx 1{,}8 \times 10^{19}$

Nota

Las palabras de la especificación entre paréntesis son opcionales en las declaraciones de las variables correspondientes. Por otra parte, hay que tener presente que las especificaciones pueden variar levemente con otros compiladores.

ANOTACIONES

Hay que recordar especialmente los distintos rangos de valores que pueden tomar las variables de cada tipo para su correcto uso en los programas. De esta manera, es posible ajustar su tamaño al que realmente sea útil.

Ejemplo

En este estándar, la letra A mayúscula se encuentra en la posición número 65.

El tipo carácter (`char`) es, de hecho, un entero que identifica una posición de la tabla de caracteres ASCII. Para evitar tener que traducir los caracteres a números, éstos se pueden introducir entre comillas simples (por ejemplo: `'A'`). También es posible representar códigos no visibles como el salto de línea (`'\n'`) o la tabulación (`'\t'`).

Números reales

Este tipo de datos es más complejo que el anterior, pues su representación binaria se encuentra codificada en distintos campos. Así pues, no se corresponde con el valor del número que se podría extraer de los bits que los forman.

Los números reales se representan mediante signo, mantisa y exponente. La mantisa expresa la parte fraccionaria del número y el exponente es el número al que se eleva la base correspondiente:

$$[+/-]\ \text{mantisa x base}^{\text{exponente}}$$

En función del número de bits que se utilicen para representarlos, los valores de la mantisa y del exponente serán mayores o menores. Los distintos tipos de reales y sus rangos aproximados se muestran en la siguiente tabla (válida en sistemas GNU de 32 bits):

Tabla 3.

Especificación	Número de bits	Rango de valores
`float`	32 (4 bytes)	$\pm 3{,}4 \times 10^{\pm 38}$
`double`	64 (8 bytes)	$\pm 1{,}7 \times 10^{\pm 308}$
`long double`	96 (12 bytes)	$\pm 1{,}1 \times 10^{\pm 4.932}$

Como se puede deducir de la tabla anterior, es importante ajustar el tipo de datos real al rango de valores que podrá adquirir una deter-

ANOTACIONES

minada variable para no ocupar memoria innecesariamente. También cabe prever lo contrario: el uso de un tipo de datos que no pueda alcanzar la representación de los valores extremos del rango empleado provocará que éstos no se representen adecuadamente y, como consecuencia, el programa correspondiente puede comportarse de forma errática.

Declaraciones

La declaración de una variable supone manifestar su existencia ante el compilador y que éste planifique una reserva de espacio en la memoria para contener sus datos. La declaración se realiza anteponiendo al nombre de la variable la especificación de su tipo (char, int, float, double), que puede, a su vez, ir precedida de uno o varios modificadores (signed, unsigned, short, long). El uso de algún modificador hace innecesaria la especificación int salvo para long. Por ejemplo:

```
unsigned short natural;     /* La variable 'natural' se         */
                            /* declara como un                  */
                            /* entero positivo.                 */
int            i, j, k;     /* Variables enteras con signo.     */
char           opcion;      /* Variable de tipo carácter.       */
float          percentil;   /* Variable de tipo real.           */
```

Por comodidad, es posible dotar a una variable de un contenido inicial determinado. Para ello, basta con añadir a la declaración un signo igual seguido del valor que tendrá al iniciarse la ejecución del programa. Por ejemplo:

```
int        importe = 0;
char       opcion = 'N';
float      angulo = 0.0;
unsigned   contador = MAXIMO;
```

El nombre de las variables puede contener cualquier combinación de caracteres alfabéticos (es decir, los del alfabeto inglés, sin 'ñ', ni 'ç', ni ningún tipo de carácter acentuado), numéricos y también el símbolo del subrayado (_); pero no puede empezar por ningún dígito.

ANOTACIONES

Es conveniente que los nombres con los que “bautizamos” las variables identifiquen su contenido o su utilización en el programa.

1.4.2. La asignación y la evaluación de expresiones

Tal como se ha comentado, en la programación imperativa, la ejecución de las instrucciones implica cambiar el estado del entorno del programa o, lo que es lo mismo, cambiar la referencia de la instrucción a ejecutar y, posiblemente, el contenido de alguna variable. Esto último ocurre cuando la instrucción que se ejecuta es la de asignación:

variable = expresión en términos de variables y valores constantes ;

La potencia (y la posible dificultad de lectura de los programas) de C se encuentra, precisamente en las expresiones.

Nota

Una expresión es cualquier combinación sintácticamente válida de operadores y operandos que pueden ser variables o constantes.

De hecho, cualquier expresión se convierte en una instrucción si se pone un punto y coma al final: todas las instrucciones de C acaban en punto y coma.

Evidentemente, evaluar una expresión no tiene sentido si no se asigna el resultado de su evaluación a alguna variable que pueda almacenarlo para operaciones posteriores. Así pues, el primer operador que se debe aprender es el de asignación:

```
entero = 23;
destino = origen;
```

Es importante no confundir el operador de asignación (=) con el de comparación de igualdad (==); pues en C, ambos son operadores que pueden emplearse entre datos del mismo tipo.

ANOTACIONES

Operadores

Aparte de la asignación, los operadores más habituales de C y que aparecen en otros lenguajes de programación se muestran en la tabla siguiente:

Tabla 4.

Clase	Operadores	Significado
Aritmético	+ -	suma y resta
	* /	producto y división
	%	módulo o resto de división entera (sólo para enteros)
Relacional	> >=	"mayor que" y "mayor e igual que"
	< <=	"menor que" y "menor e igual que"
	== !=	"igual a" y "diferente de"
Lógico	!	*NO* (proposición lógica)
	&& ||	Y (deben cumplirse todas las partes) y O lógicas

Los operadores aritméticos sirven tanto para los números reales, como para los enteros. Por este motivo, se realizan implícitamente todas las operaciones con el tipo de datos de mayor rango.

A este comportamiento implícito de los operadores se le llama *promoción de tipos de datos* y se realiza cada vez que se opera con datos de tipos distintos.

Por ejemplo, el resultado de una operación con un entero y un real (las constantes reales deben tener un punto decimal o bien contener la letra "e" que separa mantisa de exponente) será siempre un número real. En cambio, si la operación se efectúa sólo con enteros, el resultado será siempre el del tipo de datos entero de mayor rango. De esta manera:

```
real = 3 / 2 ;
```

tiene como resultado asignar a la variable `real` el valor `1.0`, que es el resultado de la división entera entre 3 y 2 transformado en un real

cuando se hace la operación de asignación. Por eso se escribe 1.0 (con el punto decimal) en lugar de 1.

Aun con esto, el operador de asignación siempre convierte el resultado de la expresión fuente al tipo de datos de la variable receptora. Por ejemplo, la asignación siguiente:

`entero` = 3.0 / 2 + 0.5;

asigna el valor 2 a `entero`. Es decir, se calcula la división real (el número 3.0 es real, ya que lleva un punto decimal) y se suma 0.5 al resultado. Esta suma actúa como factor de redondeo. El resultado es de tipo real y será truncado (su parte decimal será eliminada) al ser asignado a la variable `entero`.

Coerción de tipo

Para aumentar la legibilidad del código, evitar interpretaciones equivocadas e impedir el uso erróneo de la promoción automática de tipos, resulta conveniente indicar de forma explícita que se hace un cambio de tipo de datos. Para ello, se puede recurrir a la coerción de tipos; esto es: poner entre paréntesis el tipo de datos al que se desea convertir un determinado valor (esté en una variable o sea éste constante):

```
( especificación_de_tipo ) operando
```

Así pues, siguiendo el ejemplo anterior, es posible convertir un número real al entero más próximo mediante un redondeo del siguiente modo:

```
entero = (int) (real + 0.5);
```

En este caso, se indica que la suma se hace entre dos reales y el resultado, que será de tipo real, se convierte explícitamente a entero mediante la coerción de tipos.

Operadores relacionales

Hay que tener presente que en C no hay ningún tipo de datos lógico que se corresponda con `'`falso`'` y `'`cierto`'`. Así pues, cualquier dato

ANOTACIONES

de tipo compatible con entero será indicación de 'falso' si es 0, y 'cierto' si es distinto de 0.

Nota

Esto puede no suceder con los datos de tipo real, pues hay que tener presente que incluso los números infinitesimales cerca del 0 serán tomados como indicación de resultado lógico 'cierto'.

Como consecuencia de ello, los operadores relacionales devuelven 0 para indicar que la relación no se cumple y distinto de 0 en caso afirmativo. Los operadores && y || sólo evalúan las expresiones necesarias para determinar, respectivamente, si se cumplen todos los casos o sólo algunos. Así pues, && implica la evaluación de la expresión que constituye su segundo argumento sólo en caso de que el primero haya resultado positivo. Similarmente, || sólo ejecutará la evaluación de su segundo argumento si el primero ha resultado 'falso'.

Así pues:

```
(20 > 10) || ( 10.0 / 0.0 < 1.0 )
```

dará como resultado 'cierto ' a pesar de que el segundo argumento no se puede evaluar (¡es una división por cero!).

En el caso anterior, los paréntesis eran innecesarios, pues los operadores relacionales tienen mayor prioridad que los lógicos. Aun así, es conveniente emplear paréntesis en las expresiones para dotarlas de mayor claridad y despejar cualquier duda sobre el orden de evaluación de los distintos operadores que puedan contener.

Otros operadores

Como se ha comentado, C fue un lenguaje inicialmente concebido para la programación de sistemas operativos y, como consecuencia, con un alto grado de relación con la máquina, que se manifiesta en la existencia de un conjunto de operadores orientados a facilitar una traducción muy eficiente a instrucciones del lenguaje máquina.

ANOTACIONES

En particular, dispone de los operadores de autoincremento (++) y autodecremento (--), que se aplican directamente sobre variables cuyo contenido sea compatible con enteros. Por ejemplo:

```
contador++; /* Equivalente a: contador = contador + 1; */

--descuento; /* Equivalente a: descuento = descuento - 1; */
```

La diferencia entre la forma prefija (es decir, precediendo a la variable) y la forma postfija de los operadores estriba en el momento en que se hace el incremento o decremento: en forma prefija, se hace antes de emplear el contenido de la variable.

Ejemplo

Véase cómo se modifican los contenidos de las variables en el siguiente ejemplo:

```
a = 5;   /* ( a == 5 )                */
b = ++a; /* ( a == 6 ) && ( b == 6 ) */
c = b--; /* ( c == 6 ) && ( b == 5 ) */
```

Por otra parte, también es posible hacer operaciones entre bits. Estas operaciones se hacen entre cada uno de los bits que forma parte de un dato compatible con entero y otro. Así pues, es posible hacer una Y, una O, una O-EX (sólo uno de los dos puede ser cierto) y una negación lógica bit a bit entre los que forman parte de un dato y los de otro. (Un bit a cero representa 'falso' y uno a uno, 'cierto'.)

Los símbolos empleados para estos operadores a nivel de bit son:

– Para la Y lógica: & (*ampersand*)
– Para la O lógica: | (barra vertical)
– Para la O exclusiva lógica: ^ (acento circunflejo)
– Para la negación o complemento lógico: ~ (tilde)

A pesar de que son operaciones válidas entre datos compatibles con enteros, igual que los operadores lógicos, es muy importante tener presente que no dan el mismo resultado. Por ejemplo: `( 1 && 2 )`

ANOTACIONES

es cierto, pero (1 & 2) es falso, puesto que da 0. Para comprobarlo, basta con ver lo que pasa a nivel de bit:

```
    1 == 0000 0000 0000 0001
    2 == 0000 0000 0000 0010
1 & 2 == 0000 0000 0000 0000
```

La lista de operadores en C no termina aquí. Hay otros que se verán más tarde y algunos que quedarán en el tintero.

1.4.3. Instrucciones de selección

En el modelo de ejecución de los programas, las instrucciones se ejecutan en secuencia, una tras la otra, en el mismo orden en que aparecen en el código. Ciertamente, esta ejecución puramente secuencial no permitiría realizar programas muy complejos, pues siempre realizarían las mismas operaciones. Por ello, es necesario contar con instrucciones que permitan controlar el flujo de ejecución del programa. En otras palabras, disponer de instrucciones que permitan alterar el orden secuencial de su ejecución.

En este apartado se comentan las instrucciones de C que permiten seleccionar entre distintas secuencias de instrucciones. De forma breve, se resumen en la tabla siguiente:

Tabla 5.

Instrucción	Significado

if(condición **)**
  instrucción_si ;
else
  instrucción_no ;
	La condición tiene que ser una expresión cuya evaluación dé como resultado un dato de tipo compatible con entero. Si el resultado es distinto de cero, se considera que la condición se cumple y se ejecuta instrucción_si. En caso contrario, se ejecuta instrucción_no. El **else** es opcional.

switch(expresión **) {**
  case valor_1 :
  instrucciones
  case valor_2 :
  instrucciones
  default :
  instrucciones
} /* switch */
	La evaluación de la expresión debe resultar en un dato compatible con entero. Este resultado se compara con los valores indicados en cada case y, de ser igual a alguno de ellos, se ejecutan todas las instrucciones a partir de la primera indicada en ese caso y hasta el final del bloque del **switch**. Es posible "romper" esta secuencia introduciendo una instrucción **break;** que finaliza la ejecución de la secuencia de instrucciones. Opcionalmente, es posible indicar un caso por omisión (**default**) que permite especificar qué instrucciones se ejecutaran si el resultado de la expresión no ha producido ningún dato coincidente con los casos previstos.

En el caso del if es posible ejecutar más de una instrucción, tanto si la condición se cumple como si no, agrupando las instrucciones en

un bloque. Los bloques de instrucciones son instrucciones agrupadas entre llaves:

```
{ instrucción_1; instrucción_2; ... instrucción_N; }
```

En este sentido, es recomendable que todas las instrucciones condicionales agrupen las instrucciones que se deben ejecutar en cada caso:

```
if( condición )
{ instrucciones }
else
{ instrucciones }
/* if */
```

De esta manera se evitan casos confusos como el siguiente:

```
if( a > b )
 mayor = a ;
 menor = b ;
diferencia = mayor - menor;
```

En este caso se asigna `b` a `menor`, independientemente de la condición, pues la única instrucción del `if` es la de asignación a `mayor`.

Como se puede apreciar, las instrucciones que pertenecen a un mismo bloque empiezan siempre en la misma columna. Para facilitar la identificación de los bloques, éstos deben presentar un sangrado a la derecha respecto de la columna inicial de la instrucción que los gobierna (en este caso: `if`, `switch` y `case`).

Por convenio, cada bloque de instrucciones debe presentar un sangrado a la derecha respecto de la instrucción que determina su ejecución.

Dado que resulta frecuente tener que asignar un valor u otro a una variable es función de una condición, es posible, para estos casos,

ANOTACIONES

emplear un operador de asignación condicional en lugar de una instrucción `if`:

```
condición ? expresión_si_cierto : expresión_si_falso
```

Así pues, en lugar de:

```
if( condicion )    var = expresión_si_cierto;
else               var = expresión_si falso;
```

Se puede escribir:

```
var = condición ? expresión_si_cierto : expresión_si_falso;
```

Más aun, se puede emplear este operador dentro de cualquier expresión. Por ejemplo:

```
coste = ( km>km_contrato? km-km_contrato : 0 ) * COSTE_KM;
```

Es preferible limitar el uso del operador condicional a los casos en que se facilite la lectura.

1.4.4. Funciones estándar de entrada y de salida

El lenguaje C sólo cuenta con operadores e instrucciones de control de flujo. Cualquier otra operación que se desee realizar hay que programarla o bien emplear las funciones de que se disponga en nuestra biblioteca de programas.

Nota

Ya hemos comentado que una función no es más que una serie de instrucciones que se ejecuta como una unidad para llevar a cabo una tarea concreta. Como idea, puede tomarse la de las funciones matemáticas, que realizan alguna operación con los argumentos dados y devuelven el resultado calculado.

ANOTACIONES

El lenguaje C cuenta con un amplio conjunto de funciones estándar entre las que se encuentran las de entrada y de salida de datos, que veremos en esta sección.

El compilador de C tiene que saber (nombre y tipo de datos de los argumentos y del valor devuelto) qué funciones utilizará nuestro programa para poder generar el código ejecutable de forma correcta. Por tanto, es necesario incluir los ficheros de cabecera que contengan sus declaraciones en el código de nuestro programa. En este caso:

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

Funciones de salida estándar

La salida estándar es el lugar donde se muestran los datos producidos (mensajes, resultados, etcétera) por el programa que se encuentra en ejecución. Normalmente, esta salida es la pantalla del ordenador o una ventana dentro de la pantalla. En este último caso, se trata de la ventana asociada al programa.

```
printf( "formato" [, lista_de_campos ] )
```

Esta función imprime en la pantalla (la salida estándar) el texto contenido en “formato”. En este texto se sustituyen los caracteres especiales, que deben de ir precedidos por la barra invertida (\), por su significado en ASCII. Además, se sustituyen los especificadores de campo, que van precedidos por un %, por el valor resultante de la expresión (normalmente, el contenido de una variable) correspondiente indicada en la lista de campos. Este valor se imprime según el formato indicado en el mismo especificador.

La tabla siguiente muestra la correspondencia entre los símbolos de la cadena del formato de impresión y los caracteres ASCII.*n* se utiliza para indicar un dígito de un número:

Tabla 6.

Indicación de carácter	Carácter ASCII
`\n`	*new line* (salto de línea)
`\f`	*form feed* (salto de página)

ANOTACIONES

Indicación de carácter	Carácter ASCII
\b	*backspace* (retroceso)
\t	*tabulator* (tabulador)
\nnn	ASCII número *nnn*
\0nnn	ASCII número *nnn* (en octal)
\0Xnn	ASCII número *nn* (en hexadecimal)
\\	*backslash* (barra invertida)

Los especificadores de campo tienen el formato siguiente:

```
%[-][+][anchura[.precisión]]tipo_de_dato
```

Los corchetes indican que el elemento es opcional. El signo *menos* se emplea para indicar alineación derecha, cosa habitual en la impresión de números. Para éstos, además, si se especifica el signo *más* se conseguirá que se muestren precedidos por su signo, sea positivo o negativo. La anchura se utilizará para indicar el número mínimo de caracteres que se utilizarán para mostrar el campo correspondiente y, en el caso particular de los números reales, se puede especificar el número de dígitos que se desea mostrar en la parte decimal mediante la precisión. El tipo de dato que se desea mostrar se debe incluir obligatoriamente y puede ser uno de los siguientes:

Tabla 7.

Enteros		Reales		Otros	
%d	En decimal	%f	En punto flotante	%c	Carácter
%i	En decimal	%e	Con formato exponencial: [+/-]0.000e[+/-]000 con e minúscula o mayúscula (%E)	%s	Cadena de caracteres
%u	En decimal sin signo			%%	El signo de %
%o	En octal sin signo			(listado no completo)	
%x	En hexadecimal	%g	En formato e, f, o d.		

Ejemplo

```
printf( "El importe de la factura núm.: %5d", num_fact );
printf( "de Sr./-a. %s sube a %.2f_\n", cliente, importe );
```

Para los tipos numéricos es posible prefijar el indicador de tipo con una "ele" a la manera que se hace en la declaración de tipos con `long`. En este caso, el tipo `double` debe tratarse como un "`long float`" y, por tanto, como "`%lf`".

ANOTACIONES

Ejemplo

```
putchar( '\n' );
```

Ejemplo

```
puts( "Hola a todos!\n" );
```

Nota

El *buffer* del teclado es la memoria en la que se almacena lo que se escribe en él.

putchar(carácter **)**

Muestra el carácter indicado por la salida estándar.

puts("cadena de caracteres" **)**

Muestra una cadena de caracteres por la salida estándar.

Funciones de entrada estándar

Se ocupan de obtener datos de la entrada estándar que, habitualmente, se trata del teclado. Devuelven algún valor adicional que informa del resultado del proceso de lectura. El valor devuelto no tiene por qué emplearse si no se necesita, pues muchas veces se conoce que la entrada de datos se puede efectuar sin mayor problema.

scanf("formato" [, lista_de_&variables] **)**

Lee del *buffer* del teclado para trasladar su contenido a las variables que tiene como argumentos. La lectura se efectúa según el formato indicado, de forma similar a la especificación de campos empleada para `printf`.

Para poder depositar los datos leídos en las variables indicadas, esta función requiere que los argumentos de la lista de variables sean las direcciones de memoria en las que se encuentran. Por este motivo, es necesario emplear el operador "dirección de" (`&`). De esta manera, `scanf` deja directamente en las zonas de memoria correspondiente la información que haya leído y, naturalmente, la variable afectada verá modificado su contenido con el nuevo dato.

Es importante tener presente que si se especifican menos argumentos que especificadores de campo en el formato, los resultados pueden ser imprevisibles, pues la función cambiará el contenido de alguna zona de memoria totalmente aleatoria.

ANOTACIONES

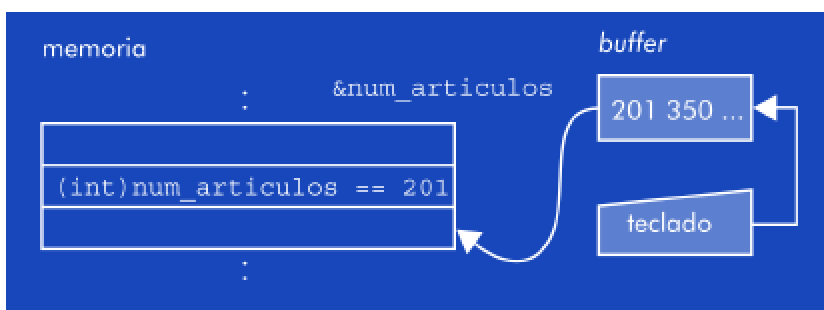
Ejemplo

```
scanf( "%d", &num_articulos );
```

En el ejemplo anterior, `scanf` lee los caracteres que se tecleen para convertirlos (presumiblemente) a un entero. El valor que se obtenga se colocará en la dirección indicada en su argumento, es decir, en el de la variable `num_articulos`. Esta lectura de caracteres del *buffer* de memoria del teclado se detiene cuando el carácter leído no se corresponde con un posible carácter del formato especificado. Este carácter se devuelve al *buffer* para una posible lectura posterior.

Para la entrada que se muestra en la figura siguiente, la función detiene la lectura después del espacio en blanco que separa los números tecleados. Éste y el resto de caracteres se quedan en el *buffer* para posteriores lecturas.

Figura 1.



La función `scanf` devuelve el número de datos correctamente leídos. Es decir, todos aquellos para los que se ha encontrado algún texto compatible con una representación de su tipo.

`getchar( )`

Devuelve un carácter leído por la entrada estándar (habitualmente, el *buffer* del teclado).

En caso de que no pueda leer ningún carácter, devuelve el carácter `EOF`. Esta constante está definida en `stdio.h` y, por tanto, puede emplearse para determinar si la lectura ha tenido éxito o, por lo contrario, se ha producido algún error o se ha llegado al final de los datos de entrada.

Ejemplo

```
opcion = getchar();
```

ANOTACIONES

Ejemplo

```
gets( nombre_usuario );
```

```
gets( cadena_caracteres )
```

Lee de la entrada estándar toda una serie de caracteres hasta encontrar un final de línea (carácter '\n'). Este último carácter es leído pero no se almacena en la cadena de caracteres que tiene por argumento.

De no leer nada devuelve `NULL`, que es una constante definida en `stdio.h` y cuyo valor es 0.

1.5. Resumen

En esta unidad se ha visto el procedimiento de ejecución de los programas en un computador. La unidad que se ocupa de procesar la información (unidad central de procesamiento o CPU) lee una instrucción de la memoria y procede a su ejecución. Esta operación implicará un cambio del estado del entorno del programa, es decir, del contenido de alguna de sus variables y de la dirección de la instrucción siguiente.

Este modelo de ejecución de las instrucciones, que siguen la mayoría de procesadores reales, se replica en el paradigma de la programación imperativa para los lenguajes de alto nivel. En particular, se ha visto cómo esto es cierto en el lenguaje de programación C.

Por este motivo, se han repasado las instrucciones de este lenguaje que permiten modificar el entorno mediante cambios en los datos y cambios en el orden secuencial en que se encuentran las instrucciones en la memoria.

Los cambios que afectan a los datos de los programas son, de hecho, asignaciones a las variables que los contienen. Se ha visto que las variables son espacios en la memoria del computador a los que podemos hacer referencia mediante el nombre con que se declaran y a los que, internamente, se hace referencia mediante la dirección de la primera palabra de la memoria que ocupan.

Se ha hecho hincapié en la forma de evaluación de las expresiones teniendo en cuenta la prioridad entre los operadores y cómo se pue-

ANOTACIONES

de organizar mejor mediante el uso de paréntesis. En este sentido, se ha comentado la conveniencia de emplear coerciones de tipo explícitas para evitar usos equívocos de la promoción automática de los tipos de datos. Esta promoción se debe realizar para que los operadores puedan trabajar siempre con operandos de un mismo tipo, que siempre coincide con el de mayor rango de representación.

En cuanto al control del flujo de ejecución que, normalmente, sigue el orden en que se encuentran las instrucciones en memoria, se han mencionado las instrucciones básicas de selección de secuencias de instrucciones: la instrucción `if` y la instrucción `switch`.

Aprovechando estas explicaciones, se ha introducido la forma especial de considerar los datos lógicos ('falso' y 'cierto') en C. Así pues, cualquier dato puede ser, en un momento dado, empleado como valor lógico. En relación a este tema, se ha comentado la forma especial de evaluación de los operadores Y y O lógicos, que no evalúan las expresiones derechas en caso de que puedan determinar el valor lógico resultante con el primer argumento (el que viene dado por la expresión que les precede).

En el último apartado, se han repasado las funciones estándar básicas de entrada y de salida de datos, de manera que sea posible construir programas para probar no sólo todos los conceptos y elementos de C explicados, sino también el entorno de desarrollo de GNU/C.

Así pues, se puede comprobar en la práctica en qué consiste la compilación y el enlace de los programas. La tarea del compilador de C es la de traducir el programa en C a un programa en lenguaje máquina. El enlazador se ocupa de añadir al código de esta versión el código de las funciones de la biblioteca que se utilicen en el programa. Con este proceso, se obtiene un código ejecutable.

1.6. Ejercicios de autoevaluación

1) Editad, compilad (y enlazad), ejecutad y comprobad el funcionamiento del siguiente programa:

```
#include <stdio.h>
main()
```

ANOTACIONES

```
{
   int a, b, suma;
   printf( "Teclea un número entero: " );
   scanf( "%d", &a );
   printf( "Teclea otro número entero: " );
   scanf( "%d", &b );
   suma = a + b;
   printf( "%d + %d = %d\n", a, b, suma );
} /* main */
```

2) Haced un programa que, dado un importe en euros y un determinado porcentaje de IVA, calcule el total.

3) Haced un programa que calcule cuánto costaría 1Kg o 1 litro de un producto sabiendo el precio de un envase y la cantidad de producto que contiene.

4) Modificad el programa anterior para que calcule el precio del producto para la cantidad deseada, que también deberá darse como entrada.

5) Haced un programa que calcule el cambio que hay que devolver conociendo el importe total a cobrar y la cantidad recibida como pago. El programa debe advertir si el importe pagado es insuficiente.

6) Haced un programa que, dado el número de litros aproximado que hay en el depósito de un coche, su consumo medio cada 100 km y una distancia en kilómetros, indique si es posible recorrerla. En caso negativo, debe indicar cuántos litros habría que añadir al depósito.

1.6.1. Solucionario

1) Basta con seguir los pasos indicados. Como ejemplo, si el fichero se llamase “suma.c”, esto es lo que debería hacerse después de haberlo creado:

```
$ gcc suma.c -o suma
$ suma
Teclea un número entero: 154
Teclea otro número entero: 703
154 + 703 = 857
$
```

ANOTACIONES

2)

```
#include <stdio.h>
main()
{
 float importe, IVA, total;

 printf( "Importe = " );
 scanf( "%f", &importe );
 printf( "%% IVA = " );
 scanf( "%f", &IVA );
 total = importe * ( 1.0 + IVA / 100.0 );
 printf( "Total = %.2f\n", total );
} /* main */
```

3)

```
#include <stdio.h>
main()
{
 float precio, precio_unit;
 int cantidad;

 printf( "Precio = " );
 scanf( "%f", &precio );
 printf( "Cantidad (gramos o mililitros) = " );
 scanf( "%d", &cantidad );
 precio_unit = precio * 1000.0 / (float) cantidad;
 printf( "Precio por Kg/Litro = %.2f\n", precio_unit );
} /* main */
```

4)

```
#include <stdio.h>
main()
{
 float precio, precio_unit, precio_compra;
 int cantidad, canti_compra;

 printf( "Precio = " );
 scanf( "%f", &precio );
 printf( "Cantidad (gramos o mililitros) = " );
 scanf( "%d", &cantidad );
 printf( "Cantidad a adquirir = " );
 scanf( "%d", &canti_compra );
 precio_unit = precio / (float) cantidad;
```

ANOTACIONES

```
  precio_compra = precio_unit * canti_compra;
  printf( "Precio de compra = %.2f\n", precio_compra );
} /* main */
```

5)

```
#include <stdio.h>
main()
{
  float importe, pago;

  printf( "Importe = " );
  scanf( "%f", &importe );
  printf( "Pago = " );
  scanf( "%f", &pago );
  if( pago < importe ) {
    printf( "Importe de pago insuficiente.\n" );
  } else {
    printf( "Cambio = %.2f euros.\n", pago - importe );
  } /* if */
} /* main */
```

6)

```
#include <stdio.h>
#define RESERVA 10
main()
{
  int litros, distancia, consumo;
  float consumo_medio;

  printf( "Litros en el depósito = " );
  scanf( "%d", &litros );
  printf( "Consumo medio cada 100Km = " );
  scanf( "%f", &consumo_medio );
  printf( "Distancia a recorrer = " );
  scanf( "%d", &distancia );
  consumo = consumo_medio * (float) distancia / 100.0;
  if( litros < consumo ) {
    printf( "Faltan %d Ltrs.\n", consumo-litros+RESERVA );
  } else {
    printf( "Se puede hacer el recorrido.\n" );
  } /* if */
} /* main */
```

ANOTACIONES

2. La programación estructurada

2.1. Introducción

La programación eficaz es aquella que obtiene un código legible y fácilmente actualizable en un tiempo de desarrollo razonable y cuya ejecución se realiza de forma eficiente.

Afortunadamente, los compiladores e intérpretes de código de programas escritos en lenguajes de alto nivel realizan ciertas optimizaciones para reducir el coste de su ejecución. Sirva como ejemplo el hecho de que muchos compiladores tienen la capacidad de utilizar el mismo espacio de memoria para diversas variables, siempre que éstas no se utilicen simultáneamente, claro está. Además de ésta y otras optimizaciones en el uso de la memoria, también pueden incluir mejoras en el código ejecutable tendentes a disminuir el tiempo final de ejecución. Pueden, por ejemplo, aprovechar los factores comunes de las expresiones para evitar la repetición de cálculos.

Con todo esto, los programadores pueden centrar su tarea en la preparación de programas legibles y de fácil mantenimiento. Por ejemplo, no hay ningún problema en dar nombres significativos a las variables, pues la longitud de los nombres no tiene consecuencias en un código compilado. En este mismo sentido, no resulta lógico emplear trucos de programación orientados a obtener un código ejecutable más eficiente si ello supone disminuir la legibilidad del código fuente. Generalmente, los trucos de programación no suponen una mejora significativa del coste de ejecución y, en cambio, implican dificultad de mantenimiento del programa y dependencia de un determinado entorno de desarrollo y de una determinada máquina.

Por este motivo, en esta unidad se explica cómo organizar el código fuente de un programa. La correcta organización de los programas supone un incremento notable de su legibilidad y, como consecuencia, una disminución de los errores de programación y facilidad de

ANOTACIONES

mantenimiento y actualización posterior. Más aún, resultará más fácil aprovechar partes de sus códigos en otros programas.

En el primer apartado se trata de la programación estructurada. Este paradigma de la programación está basado en la programación imperativa, a la que impone restricciones respecto de los saltos que pueden efectuarse durante la ejecución de un programa.

Nota

Un salto es un cambio en el orden de ejecución de las instrucciones por el que la siguiente instrucción no es la que encuentra a continuación de la que se está ejecutando.

Con estas restricciones se consigue aumentar la legibilidad del código fuente, permitiendo a sus lectores determinar con exactitud el flujo de ejecución de las instrucciones.

Es frecuente, en programación, encontrarse con bloques de instrucciones que deben ejecutarse repetitivamente. Por tanto, es necesario ver cómo disponer el código correspondiente de forma estructurada. En general, estos casos derivan de la necesidad de procesar una serie de datos. Así pues, en el primer apartado, no sólo se verá la programación estructurada, sino también los esquemas algorítmicos para realizar programas que traten con series de datos y, cómo no, las correspondientes instrucciones en C.

Por mucho que la programación estructurada esté encaminada a reducir errores en la programación, éstos no se pueden eliminar. Así pues, se dedica un apartado a la depuración de errores de programación y a las herramientas que nos pueden ayudar a ello: los depuradores.

El segundo apartado se dedica a la organización lógica de los datos de los programas. Hay que tener presente que la información que procesan los computadores y su resultado está constituido habitualmente por una colección variopinta de datos. Estos datos, a su vez, pueden estar constituidos por otros más simples.

En los lenguajes de programación se da soporte a unos tipos básicos de datos, es decir, se incluyen mecanismos (declaraciones, operaciones e instrucciones) para emplearlos en los códigos fuente que se escriben con ellos.

Por tanto, la representación de la información que se maneja en un programa debe hacerse en términos de variables que contengan da-

ANOTACIONES

tos de los tipos fundamentales a los que el lenguaje de programación correspondiente dé soporte. No obstante, resulta conveniente agrupar conjuntos de datos que estén muy relacionados entre sí en la información que representan. Por ejemplo: manejar como una única entidad el número de días de cada uno de los meses del año; el día, el mes y el año de una fecha; o la lista de los días festivos del año.

En el segundo apartado, pues, se tratará de aquellos aspectos de la programación que permiten organizar los datos en estructuras mayores. En particular, se verán las clases de estructuras que existen y cómo utilizar variables que contengan estos datos estructurados. También se verá cómo la definición de tipos de datos a partir de otros tipos, estructurados o no, beneficia a la programación. Dado que estos nuevos tipos no son reconocidos en el lenguaje de programación, son llamados *tipos de datos abstractos*.

Nota

Un tipo de datos abstracto es aquel que representa una información no contemplada en el lenguaje de programación empleado. Puede darse el caso de que haya tipos de datos soportados en algún lenguaje de programación que, sin embargo, en otros no lo estén y haya que tratarlos como abstractos.

El último apartado introduce los principios de la programación modular, que resulta fundamental para entender la programación en C. En este modelo de programación, el código fuente se divide en pequeños programas estructurados que se ocupan de realizar tareas muy específicas dentro del programa global. De hecho, con esto se consigue dividir el programa en subprogramas de más fácil lectura y comprensión. Estos subprogramas constituyen los llamados *módulos*, y de ahí se deriva el nombre de esta técnica de programación.

En C todos los módulos son funciones que suelen realizar acciones muy concretas sobre unas pocas variables del programa. Más aún, cada función suele especializarse en un tipo de datos concreto.

Dada la importancia de este tema, se insistirá en los aspectos de la declaración, definición y uso de las funciones en C. En especial, todo lo referente al mecanismo que se utiliza durante la ejecución de los programas para proporcionar y obtener datos de las funciones que incluyen.

Finalmente, se tratará sobre las macros del preprocesador de C. Estas macros tienen una apariencia similar a la de las llamadas de las funciones en C y pueden llevar a confusiones en la interpretación del código fuente del programa que las utilice.

El objetivo principal de esta unidad es que el lector aprenda a organizar correctamente el código fuente de un programa, pues es un indicador fundamental de la calidad de la programación. De forma más concreta, el estudio de esta unidad pretende que se alcancen los objetivos siguientes:

1. Conocer en qué consiste la programación estructurada.
2. Saber aplicar correctamente los esquemas algorítmicos de tratamiento de secuencias de datos.
3. Identificar los sistemas a emplear para la depuración de errores de un programa.
4. Saber cuáles son las estructuras de datos básicas y cómo emplearlas.
5. Saber en qué consiste la programación modular.
6. Conocer la mecánica de la ejecución de las funciones en C.

2.2. Principios de la programación estructurada

La programación estructurada es una técnica de programación que resultó del análisis de las estructuras de control de flujo subyacentes a todo programa de computador. El producto de este estudio reveló que es posible construir cualquier estructura de control de flujo mediante tres estructuras básicas: la secuencial, la condicional y la iterativa.

Lecturas complementarias

E.W. Dijkstra (1968). *The goto statement consireded harmful*

E.W. Dijkstra (1970). *Notes on structured programming*

La programación estructurada consiste en la organización del código de manera que el flujo de ejecución de sus instrucciones resulte evidente a sus lectores.

Un teorema formulado el año 1966 por Böhm y Jacopini dice que todo “programa propio” debería tener un único punto de entrada y un único punto de salida, de manera que toda instrucción entre estos dos puntos es ejecutable y no hay bucles infinitos.

ANOTACIONES

La conjunción de estas propuestas proporciona las bases para la construcción de programas estructurados en los que las estructuras de control de flujo se pueden realizar mediante un conjunto de instrucciones muy reducido.

De hecho, la estructura secuencial no necesita ninguna instrucción adicional, pues los programas se ejecutan normalmente llevando a cabo las instrucciones en el orden en que aparecen en el código fuente.

En la unidad anterior se comentó la instrucción `if`, que permite una ejecución condicional de bloques de instrucciones. Hay que tener presente que, para que sea un programa propio, debe existir la posibilidad de que se ejecuten todos los bloques de instrucciones.

Las estructuras de control de flujo iterativas se comentarán en el apartado siguiente. Vale la pena indicar que, en cuanto a la programación estructurada se refiere, sólo es necesaria una única estructura de control de flujo iterativa. A partir de ésta se pueden construir todas las demás.

2.3. Instrucciones iterativas

Las instrucciones iterativas son instrucciones de control de flujo que permiten repetir la ejecución de un bloque de instrucciones. En la tabla siguiente se muestran las que están presentes en C:

Tabla 8.

instrucción	Significado

`while( condición ) {`
`  instrucciones`
`} /* while */`
	Se ejecutan todas las `instrucciones` en el bloque del bucle mientras la expresión de la `condición` dé como resultado un dato de tipo compatible con entero distinto de cero; es decir, mientras la condición se cumpla. Las `instrucciones` pueden no ejecutarse nunca.

`do {`
`  instrucciones`
`} while ( condición );`
	De forma similar al bucle `while`, se ejecutan todas las `instrucciones` en el bloque del bucle mientras la expresión de la `condición` se cumpla. La diferencia estriba en que las `instrucciones` se ejecutarán, al menos, una vez. (La comprobación de la condición y, por tanto, de la posible repetición de las instrucciones, se realiza al final del bloque.)

`for(`
`  inicialización ;`
`  condición ;`
`  continuacón`
`) {`
`  instrucciones`
`} /* for */`
	El comportamiento es parecido a un bucle while; es decir, mientras se cumpla la condición se ejecutan las instrucciones de su bloque. En este caso, sin embargo, es posible indicar qué instrucción o instrucciones quieren ejecutarse de forma previa al inicio del bucle (`inicialización`) y qué instrucción o instrucciones hay que ejecutar cada vez que finaliza la ejecución de las instrucciones (`continuación`).

Como se puede apreciar, todos los bucles pueden reducirse a un bucle “mientras”. Aun así, hay casos en los que resulta más lógico emplear alguna de sus variaciones.

Hay que tener presente que la estructura del flujo de control de un programa en un lenguaje de alto nivel no refleja lo que realmente hace el procesador (saltos condicionales e incondicionales) en el aspecto del control del flujo de la ejecución de un programa. Aun así, el lenguaje C dispone de instrucciones que nos acercan a la realidad de la máquina, como la de salida forzada de bucle (`break;`) y la de la continuación forzada de bucle (`continue;`). Además, también cuenta con una instrucción de salto incondicional (`goto`) que no debería de emplearse en ningún programa de alto nivel.

Normalmente, la programación de un bucle implica determinar cuál es el bloque de instrucciones que hay que repetir y, sobre todo, bajo qué condiciones hay que realizar su ejecución. En este sentido, es muy importante tener presente que la condición que gobierna un bucle es la que determina la validez de la repetición y, especialmente, su finalización cuando no se cumple. Nótese que debe existir algún caso en el que la evaluación de la expresión de la condición dé como resultado un valor `'falso'`. En caso contrario, el bucle se repetiría indefinidamente (esto es lo que se llamaría un caso de “bucle infinito”).

Habiendo determinado el bloque iterativo y la condición que lo gobierna, también cabe programar la posible preparación del entorno antes del bucle y las instrucciones que sean necesarias a su conclusión: su inicialización y su finalización.

La instrucción iterativa debe escogerse en función de la condición que gobierna el bucle y de su posible inicialización.

En los casos en que sea posible que no se ejecuten las instrucciones del bucle, es conveniente emplear `while`. Por ejemplo, para calcular cuántos divisores tiene un número entero positivo dado:

ANOTACIONES

```
/* ... */
/* Inicialización: ___________________________*/
divisor = 1; /* Candidato a divisor  */
ndiv = 0;    /* Número de divisores  */
/* Bucle: ____________________________________*/
while( divisor < numero ) {
  if( numero % divisor == 0 ) ndiv = ndiv + 1;
  divisor = divisor + 1;
} /* while */
/* Finalización: _____________________________*/
if( numero > 0 ) ndiv = ndiv + 1;
/* ... */
```

A veces, la condición que gobierna el bucle depende de alguna variable que se puede tomar como un contador de repeticiones; es decir, su contenido refleja el número de iteraciones realizado. En estos casos, puede considerarse el uso de un bucle `for`. En concreto, se podría interpretar como "iterar el siguiente conjunto de instrucciones para todos los valores de un contador entre uno inicial y uno final dados". En el ejemplo siguiente, se muestra esta interpretación en código C:

```
/* ... */
unsigned int contador;
/* ... */
for( contador = INICIO ;
  contador ⇐ FINAL ;
  contador = contador + INCREMENTO
) {
  instrucciones
} /* for */
/* ... */
```

A pesar de que el ejemplo anterior es muy común, no es necesario que la variable que actúe de contador tenga que incrementarse, ni que tenga que hacerlo en un paso fijo, ni que la condición sólo deba consistir en comprobar que haya llegado a su valor final y, ni mucho menos, que sea una variable adicional que no se emplee en las instrucciones del cuerpo a iterar. De hecho sería muy útil que fuera alguna variable cuyo contenido se modificara a cada iteración y que, con ello, pudiese emplearse como contador.

ANOTACIONES

Es recomendable evitar el empleo del `for` para los casos en que no haya contadores. En su lugar, es mucho mejor emplear un `while`.

En algunos casos, gran parte de la inicialización coincidiría con el cuerpo del bucle, o bien se hace necesario evidenciar que el bucle se ejecutará al menos una vez. Si esto se da, es conveniente utilizar una estructura `do...while`. Como ejemplo, veamos el código de un programa que realiza la suma de distintos importes hasta que el importe leído sea cero:

```
/* ... */
float total, importe;
/* ... */
total = 0.00;
printf( "SUMA" );
do {
   printf( " + " );
   scanf( "%f", &importe );
   total = total + importe;
} while( importe != 0.00 );
printf( " = %.2f", total );
/* ... */
```

La constante numérica real `0.00` se usa para indicar que sólo son significativos dos dígitos fraccionarios, ya que, a todos los efectos, seria igual optar por escribir `0.0` o, incluso, `0` (en el último caso, el número entero seria convertido a un número real antes de realizar la asignación).

Sea cual sea el caso, la norma de aplicación es la de mantener siempre un código inteligible. Por otra parte, la elección del tipo de instrucción iterativa depende del gusto estético del programador y de su experiencia, sin mayores consecuencias en la eficiencia del programa en cuanto a coste de ejecución.

2.4. Procesamiento de secuencias de datos

Mucha de la información que se trata consta de secuencias de datos que pueden darse explícita o implícitamente.

Ejemplo

En el primer caso, se trataría del procesamiento de información de una serie de datos procedentes del dispositivo de entrada estándar.

Un ejemplo del segundo sería aquel en el que se deben procesar una serie de valores que adquiere una misma variable.

En los dos casos, el tratamiento de la secuencia se puede observar en el código del programa, pues debe realizarse mediante alguna instrucción iterativa. Habitualmente, este bucle se corresponde con un esquema algorítmico determinado. En los siguientes apartados veremos los dos esquemas fundamentales para el tratamiento de secuencias de datos.

2.4.1. Esquemas algorítmicos: recorrido y búsqueda

Los esquemas algorítmicos para el procesado de secuencias de datos son unos patrones que se repiten frecuentemente en muchos algoritmos. Consecuentemente, existen unos patrones equivalentes en los lenguajes de programación como el C. En este apartado veremos los patrones básicos de tratamiento de secuencias: el de recorrido y el de búsqueda.

Recorrido

Un recorrido de una secuencia implica realizar un tratamiento idéntico a todos los miembros de la misma.

En otras palabras, supone tratar cada uno de los elementos de la secuencia, desde el primero hasta el último.

Si el número de elementos de que constará la secuencia es conocido *a priori* y la inicialización del bucle es muy simple, entonces puede ser conveniente emplear un bucle `for`. En caso contrario, los bucles

más adecuados son el bucle `while` o el `do…while` si se sabe que habrá, al menos, un elemento en la secuencia de datos.

El esquema algorítmico del recorrido de una secuencia sería, en su versión para C, el que se presenta a continuación:

```
/* inicialización para el procesamiento de la secuencia*/
/* (puede incluir el tratamiento del primer elemento)   */
while( ! /* final de secuencia */ ) {
   /* tratar el elemento */
   /* avanzar en secuencia */
} /* while */
/* finalización del procesamiento de la secuencia       */
/* (puede incluir el tratamiento del último elemento)   */
```

El patrón anterior podría realizarse con alguna otra instrucción iterativa, si las circunstancias lo aconsejaran.

Para ilustrar varios ejemplos de recorrido, supongamos que se desea obtener la temperatura media en la zona de una estación meteorológica. Para ello, procederemos a hacer un programa al que se le suministran las temperaturas registradas a intervalos regulares por el termómetro de la estación y obtenga la media de los valores introducidos.

Así pues, el cuerpo del bucle consiste, simplemente, en acumular la temperatura (tratar el elemento) y en leer una nueva temperatura (avanzar en la secuencia):

```
/* ... */
acumulado = acumulado + temperatura;
cantidad = cantidad + 1;
scanf( "%f", &temperatura );
/* ... */
```

En este bloque iterativo se puede observar que `temperatura` debe tener un valor determinado antes de poderse acumular en la variable `acumulado`, la cual, a su vez, también tiene que estar inicializada. Similarmente, `cantidad` deberá inicializarse a cero.

ANOTACIONES

Por ello, la fase de inicialización y preparación de la secuencia ya está lista:

```
/* ... */
unsigned int cantidad;
float acumulado;
/* ... */
cantidad = 0;
acumulado = 0.00;
scanf( "%f", &temperatura );
/* bucle ... */
```

Aun queda por resolver el problema de establecer la condición de finalización de la secuencia de datos. En este sentido, puede ser que la secuencia de datos tenga una marca de final de secuencia o que su longitud sea conocida.

En el primero de los casos, la marca de final debe de ser un elemento especial de la secuencia que tenga un valor distinto del que pueda tener cualquier otro dato. En este sentido, como se sabe que una temperatura no puede ser nunca inferior a –273,16 °C (y, mucho menos, una temperatura ambiental), se puede emplear este valor como marca de final. Por claridad, esta marca será una constante definida en el preprocesador:

```
#define MIN_TEMP -273.16
```

Cuando se encuentre, no deberá ser procesada y, en cambio, sí que debe hacerse la finalización del recorrido, calculando la media:

```
/* ... */
float media;
/* ... fin del bucle */
if( cantidad > 0 ) {
   media = acumulado / (float) cantidad;
} else {
   media = MIN_TEMP;
} /* if */
/* ... */
```

ANOTACIONES

En el cálculo de la media, se comprueba primero que haya algún dato significativo para computarse. En caso contrario, se asigna a media la temperatura de marca de final. Con todo, el código del recorrido sería el mostrado a continuación:

```
/* ... */
cantidad = 0;
acumulado = 0.00;
scanf( "%f", &temperatura );
while( ! ( temperatura == MIN_TEMP ) ) {
   acumulado = acumulado + temperatura;
   cantidad = cantidad + 1;
   scanf( "%f", &temperatura );
} /* while */
if( cantidad > 0 ) {
 media = acumulado / (float) cantidad;
} else {
   media = MIN_TEMP;
} /* if */
/* ... */
```

Si la marca de final de secuencia se proporcionara aparte, la instrucción iterativa debería ser una `do..while`. En este caso, se supondrá que la secuencia de entrada la forman elementos con dos datos: la temperatura y un valor entero tomado como valor lógico que indica si es el último elemento:

```
/* ... */
cantidad = 0;
acumulado = 0.00;
do {
   scanf( "%f", &temperatura );
   acumulado = acumulado + temperatura;
   cantidad = cantidad + 1;
   scanf( "%u", &es_ultimo );
} while( ! es_ultimo );
if( cantidad > 0 ) {
   media = acumulado / (float) cantidad;
} else {
   media = MIN_TEMP;
} /* if */
/* ... */
```

ANOTACIONES

En caso de que se conociera el número de temperaturas (`NTEMP`) que se han registrado, bastaría con emplear un bucle de tipo `for`:

```
/* ... */
acumulado = 0.00;
for( cant = 1; cant ⇐ NTEMP; cant = cant + 1 ) {
   scanf( "%f", &temperatura );
   acumulado = acumulado + temperatura;
} /* for */
media = acumulado / (float) NTEMP;
/* ... */
```

Búsqueda

Las búsquedas consisten en recorridos, mayoritariamente parciales, de secuencias de datos de entrada. Se recorren los datos de una secuencia de entrada hasta encontrar el que satisfaga una determinada condición. Evidentemente, si no se encuentra ningún elemento que satisfaga la condición, se realizará el recorrido completo de la secuencia.

De forma general, la búsqueda consiste en recorrer una secuencia de datos de entrada hasta que se cumpla una determinada condición o se acaben los elementos de la secuencia. No es necesario que la condición afecte a un único elemento.

Siguiendo el ejemplo anterior, es posible hacer una búsqueda que detenga el recorrido cuando la media progresiva se mantenga en un margen de ±1 °C respecto de la temperatura detectada durante más de 10 registros.

El esquema algorítmico es muy parecido al del recorrido, salvo por el hecho de que se incorpora la condición de búsqueda y que, a la

salida del bucle, es necesario comprobar si la búsqueda se ha resuelto satisfactoriamente o no:

```
/* inicialización para el procesamiento de la secuencia  */
/* (puede incluir el tratamiento del primer elemento)    */
encontrado = FALSO;
while( ! /* final de secuencia */ && !encontrado ) {
   /* tratar el elemento */
   if( /* condición de encontrado */ ) {
    encontrado = CIERTO;
   } else {
    /* avanzar en secuencia */
   } /* if */
} /* while */
/* finalización del procesamiento de la secuencia        */
if( encontrado ) {
   /* instrucciones */
} else {
   /* instrucciones */
} /* if */
```

En este esquema se supone que se han definido las constantes FALSO y CIERTO del modo siguiente:

```
#define FALSO 0
#define CIERTO 1
```

Si se aplica el patrón anterior a la búsqueda de una media progresiva estable, el código fuente sería el siguiente:

```
/* ... */
cantidad = 0;
acumulado = 0.00;
scanf( "%f", &temperatura );
seguidos = 0;
encontrado = FALSO;
while( ! ( temperatura == MIN_TEMP ) && ! encontrado ) {
   acumulado = acumulado + temperatura;
   cantidad = cantidad + 1;
   media = acumulado / (float) cantidad;
```

ANOTACIONES

```
   if( media<=temperatura+1.0 || temperatura-1.0<=media ) {
     seguidos = seguidos + 1;
   } else {
     seguidos = 0;
   } /* if */
   if( seguidos == 10 ) {
     encontrado = CIERTO;
   } else {
     scanf( "%f", &temperatura );
   } /* if */
} /* while */
/* ... */
```

En los casos de búsqueda no suele ser conveniente emplear un `for`, ya que suele ser una instrucción iterativa que emplea un contador que toma una serie de valores desde uno inicial hasta uno final. Es decir, hace un recorrido por la secuencia implícita de todos los valores que toma la variable de conteo.

2.4.2. Filtros y tuberías

Los filtros son programas que generan una secuencia de datos a partir de un recorrido de una secuencia de datos de entrada. Habitualmente, la secuencia de datos de salida contiene los datos procesados de la de entrada.

El nombre de *filtros* se les aplica porque es muy común que la secuencia de salida sea, simplemente, una secuencia de datos como la de entrada en la que se han suprimido algunos de sus elementos.

Un filtro sería, por ejemplo, un programa que tuviera como salida las sumas parciales de los números de entrada:

```
#include <stdio.h>
main()
{
   float suma, sumando;
   suma = 0.00;
   while( scanf( "%f", &sumando ) == 1 ) {
```

```
    suma = suma + sumando;
    printf( "%.2f ", suma );
  } /* while */
} /* main */
```

Otro filtro, quizá más útil, podría tratarse de un programa que sustituya los tabuladores por el número de espacios en blanco necesarios hasta la siguiente columna de tabulación:

```
#include <stdio.h>
#define TAB 8
main()
{
  char caracter;
  unsigned short posicion, tabulador;
  posicion = 0;
  caracter = getchar();
  while( caracter != EOF ) {
    switch( caracter ) {
     case '\t':/* avanza hasta siguiente columna */
      for( tabulador = posicion;
          tabulador < TAB;
          tabulador = tabulador + 1 ) {
        putchar( ' ' );
      } /* for */
      posicion = 0;
      break;
     case '\n': /* nueva línea implica columna 0 */
      putchar( caracter );
      posicion = 0;
      break;
     default:
      putchar( caracter );
      posicion = (posicion + 1) % TAB;
    } /* switch */
    caracter = getchar();
  } /* while */
} /* main */
```

Estos pequeños programas pueden resultar útiles por sí mismos o bien combinados. Así pues, la secuencia de datos de salida de uno puede

ANOTACIONES

constituir la secuencia de entrada de otro, constituyéndose lo que denominamos una ***tubería*** (*pipe*, en inglés) la idea visual es que por un extremo de la tubería se introduce un flujo de datos y por el otro se obtiene otro flujo de datos ya procesado. En el camino, la tubería puede incluir uno o más filtros que retienen y/o transforman los datos.

Ejemplo

Un filtro podría convertir una secuencia de datos de entrada consistentes en tres números (código de artículo, precio y cantidad) a una secuencia de datos de salida de dos números (código e importe) y el siguiente podría consistir en un filtro de suma, para recoger el importe total.

Para que esto sea posible, es necesario contar con la ayuda del sistema operativo. Así pues, no es necesario que la entrada de datos se efectúe a través del teclado ni tampoco que la salida de datos sea obligatoriamente por la pantalla, como dispositivos de entrada y salida estándar que son. En Linux (y también en otros SO) se puede redirigir la entrada y la salida estándar de datos mediante los comandos de redirección. De esta manera, se puede conseguir que la entrada estándar de datos sea un fichero determinado y que los datos de salida se almacenen en otro fichero que se emplee como salida estándar.

En el ejemplo anterior, se puede suponer que existe un fichero (`ticket.dat`) con los datos de entrada y queremos obtener el total de la compra. Para ello, podemos emplear un filtro para calcular los importes parciales, cuya salida será la entrada de otro que obtenga el total.

Para aplicar el primer filtro, será necesario que ejecutemos el programa correspondiente (al que llamaremos `calcula_importes`) redirigiendo la entrada estándar al fichero `ticket.dat`, y la salida al fichero `importes.dat`:

```
$ calcula_importes <ticket.dat >importes.dat
```

Con ello, `importes.dat` recogerá la secuencia de pares de datos (código de artículo e importe) que el programa haya generado por

ANOTACIONES

la salida estándar. Las redirecciones se determinan mediante los símbolos “menor que” para la entrada estándar y “mayor que” para la salida estándar.

Si deseamos calcular los importes de otras compras para luego calcular la suma de todas ellas, será conveniente añadir a `importes.dat` todos los importes parciales de todos los boletos de compra. Esto es posible mediante el operador de redirección de salida doblado, cuyo significado podría ser “añadir al fichero la salida estándar del programa”:

```
$ calcula_importes <otro_ticket.dat >>importes.dat
```

Cuando hayamos recogido todos los importes parciales que queramos sumar, podremos proceder a llamar al programa que calcula la suma:

```
$ suma <importes.dat
```

Si sólo nos interesa la suma de un único boleto de compra, podemos montar una tubería en la que la salida del cálculo de los importes parciales sea la entrada de la suma:

```
$ calcula_importes <ticket.dat | suma
```

Como se puede observar en el comando anterior, la tubería se monta con el operador de tubería representado por el símbolo de la barra vertical. Los datos de la salida estándar de la ejecución de lo que le precede los transmite como entrada estándar al programa que tenga a continuación.

2.5. Depurado de programas

El depurado de programas consiste en eliminar los errores que éstos contengan. Los errores pueden ser debidos tanto a la programación como al algoritmo programado. Así pues, el depurado de un programa puede implicar un cambio en el algoritmo correspondiente. Cuando la causa del error se encuentra en el algoritmo o en su in-

ANOTACIONES

correcta programación se habla de **error de lógica**. En cambio, si el error tiene su razón en la violación de las normas del lenguaje de programación se habla de un **error de sintaxis** (aunque algunos errores tengan una naturaleza léxica o semántica).

> **Nota**
>
> *Debug* es el término inglés para referirse al depurado de errores de programas de computador. El verbo se puede traducir por "eliminar bichos" y tiene su origen en un reporte de 1945 sobre una prueba del computador Mark II realizada en la Universidad de Harvard. En el informe se registró que se encontró una polilla en un relé que provocaba su mal funcionamiento. Para probar que se había quitado el bicho (y resuelto el error), se incluyó la polilla en el mismo informe. Fue sujetada mediante cinta adhesiva y se añadió un pie que decía "primer caso de una polilla encontrada". Fue también la primera aparición del verbo *debug* (quitar bichos) que tomó la acepción actual.

Los errores de sintaxis son detectados por el compilador, ya que le impiden generar código ejecutable. Si el compilador puede generar código a pesar de la posible existencia de un error, el compilador suele emitir un aviso.

Por ejemplo, es posible que una expresión en un `if` contenga un operador de asignación, pero lo habitual es que se trate de una confusión entre operadores de asignación y de comparación:

```
/* ... */
if( a = 5 ) b = 6;
/* ... */
```

Más aún, en el código anterior se trata de un error de programación, puesto que la instrucción parece indicar que es posible que `b` pueda no ser 6. Si atendemos a la condición del `if`, se trata de una asignación del valor 5 a la variable `a`, con resultado igual al valor asignado. Así pues, como el valor 5 es distinto de cero, el resultado es siempre afirmativo y, consecuentemente, `b` siempre toma el valor 6.

ANOTACIONES

Por este motivo, es muy recomendable que el compilador nos dé todos los avisos que pueda. Para ello, debe ejecutarse con el argumento siguiente:

```
$ gcc -Wall -o programa programa.c
```

El argumento `–Wall` indica que se dé aviso de la mayoría de casos en los que pueda haber algún error lógico. A pesar de que el argumento parece indicar que se nos avisará sobre cualquier situación, aún hay algunos casos sobre los que no avisa.

Los errores más difíciles de detectar son los errores lógicos que escapan incluso a los avisos del compilador. Estos errores son debidos a una programación indebida del algoritmo correspondiente, o bien, a que el propio algoritmo es incorrecto. En todo caso, después de su detección hay que proceder a su localización en el código fuente.

Para la localización de los errores será necesario determinar en qué estado del entorno se producen; es decir, bajo qué condiciones ocurren. Por lo tanto, es necesario averiguar qué valores de las variables conducen el flujo de ejecución del programa a la instrucción en la que se manifiesta el error.

Desafortunadamente, los errores suelen manifestarse en un punto posterior al del estado en el que realmente se produjo el fallo del comportamiento del programa. Así pues, es necesario poder observar el estado del programa en cualquier momento para seguir su evolución hasta la manifestación del error con el propósito de detectar el fallo que lo causa.

Para aumentar la ***observabilidad*** de un programa, es habitual introducir testigos (también llamados *chivatos*) en su código de manera que nos muestren el contenido de determinadas variables. De todas maneras, este procedimiento supone la modificación del programa cada vez que se introducen nuevos testigos, se eliminan los que resultan innecesarios o se modifican.

ANOTACIONES

Por otra parte, para una mejor localización del error, es necesario poder tener control sobre el flujo de ejecución. La ***controlabilidad*** implica la capacidad de modificar el contenido de las variables y de elegir entre distintos flujos de ejecución. Para conseguir un cierto grado de control es necesario introducir cambios significativos en el programa que se examina.

En lugar de todo lo anterior, es mejor emplear una herramienta que nos permita observar y controlar la ejecución de los programas para su depuración. Estas herramientas son los llamados *depuradores* (*debuggers*, en inglés).

Para que un depurador pueda realizar su trabajo, es necesario compilar los programas de manera que el código resultante incluya información relativa al código fuente. Así pues, para depurar un programa, deberemos compilarlo con la opción `–g`:

```
$ gcc -Wall -o programa -g programa.c
```

En GNU/C existe un depurador llamado `gdb` que nos permitirá ejecutar un programa, hacer que se pare en determinadas condiciones, examinar el estado del programa cuando esté parado y, por último, cambiarlo para poder experimentar posibles soluciones.

El depurador se invoca de la siguiente manera:

```
$ gdb programa
```

En la tabla siguiente se muestran algunos de los comandos que podemos indicarle a GDB:

Tabla 8.

Comando	Acción
`run`	Empieza a ejecutar el programa por su primera instrucción. El programa sólo se detendrá en un punto de parada, cuando el depurador reciba un aviso de parada (es decir, con el tecleo de control y C simultáneamente), o bien cuando espere alguna entrada de datos.
`break núm_línea`	Establece un punto de parada antes de la primera instrucción que se encuentra en la línea indicada del código fuente. Si se omite el número de línea, entonces lo establece en la primera instrucción de la línea actual; es decir, en la que se ha detenido.
`clear núm_línea`	Elimina el punto de parada establecido en la línea indicada o, si se omite, en la línea actual.
`c`	Continúa la ejecución después de una detención.

Comando	Acción
`next`	Ejecuta la instrucción siguiente y se detiene.
`print expresión`	Imprime el resultado de evaluar la expresión indicada. En particular, la expresión puede ser una variable y el resultado de su evaluación, su contenido.
`help`	Muestra la lista de los comandos.
`quit`	Finaliza la ejecución de GDB

Los puntos de parada o *breakpoints* son marcas en el código ejecutable que permiten al depurador conocer si debe parar la ejecución del programa en curso o, por el contrario, debe continuar permitiendo su ejecución. Estas marcas se pueden fijar o eliminar a través del propio depurador. Con ello, es posible ejecutar porciones de código de forma unitaria.

En particular, puede ser conveniente introducir un punto de parada en `main` antes de proceder a su ejecución, de manera que se detenga en la primera instrucción y nos permita realizar un mejor seguimiento de la misma. Para tal efecto, basta con el comando `break main`, ya que es posible indicar nombres de funciones como puntos de parada.

De todas maneras, es mucho más práctico emplear algún entorno gráfico en el que pueda verse el código fuente al mismo tiempo que la salida del programa que se ejecuta. Para ello, se puede emplear, por ejemplo, el DDD (*Data Display Debugger*) o el XXGDB. Los dos entornos emplean el GDB como depurador y, por tanto, disponen de las mismas opciones. No obstante, su manejo es más fácil porque la mayoría de comandos están a la vista y, en todo caso, en los menús desplegables de que disponen.

2.6. Estructuras de datos

Los tipos de datos básicos (compatibles con enteros y reales) pueden agruparse en estructuras homogéneas o heterogéneas, de manera que se facilita (y aclara) el acceso a sus componentes dentro de un programa.

ANOTACIONES

Una estructura homogénea es aquella cuyos datos son todos del mismo tipo y una heterogénea puede estar formada por datos de tipo distinto.

En los apartados siguientes se revisarán las principales estructuras de datos en C, aunque existen en todos los lenguajes de programación estructurada. Cada apartado se organiza de manera que se vea cómo se pueden llevar a cabo las siguientes operaciones sobre las variables:

– Declararlas, para que el compilador les reserve el espacio correspondiente.

– Inicializarlas, de manera que el compilador les dé un contenido inicial (que puede cambiar) en el programa ejecutable resultante.

– Referenciarlas, de forma que se pueda acceder a su contenido, tanto para modificarlo como para leerlo.

Como es obligatorio anteponer el tipo de la variable en su declaración, resulta conveniente identificar a los tipos de datos estructurados con un nombre de tipo. Estos nuevos tipos de datos se conocen como *tipos de datos abstractos*. El último apartado estará dedicado a ellos.

2.7. Matrices

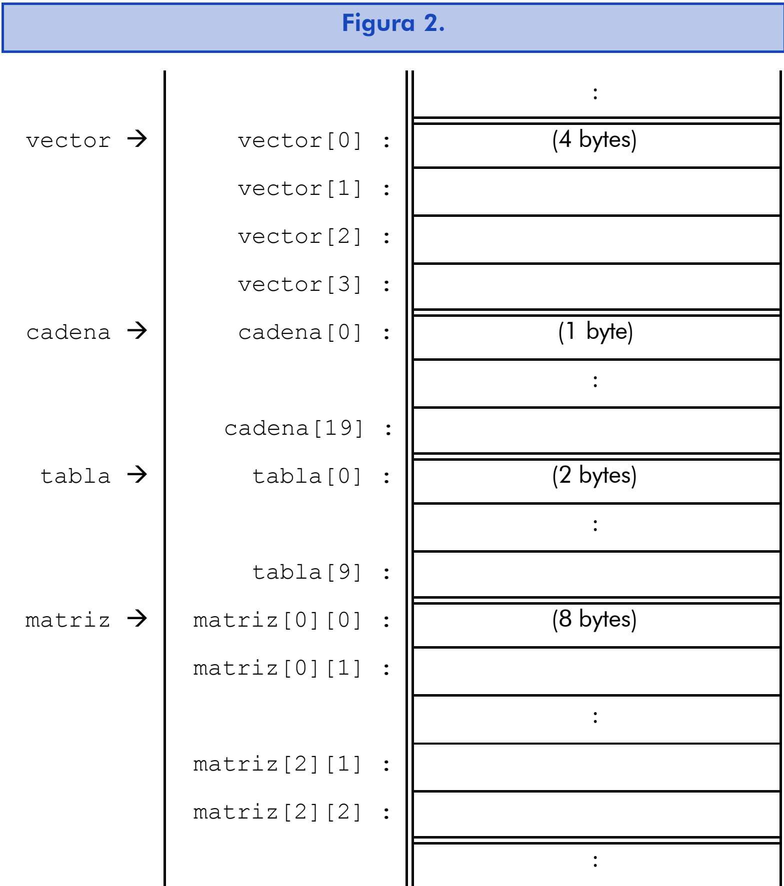
Las matrices son estructuras de datos homogéneas de tamaño fijo. Es decir, se representa siempre una información que emplea un número determinado de datos. También se llaman **arreglos** (una traducción bastante directa del término *array* en inglés), **tablas** (para las de una o dos dimensiones) o **vectores** (si son unidimensionales). En el caso particular de los vectores de caracteres, reciben el nombre de **cadenas de caracteres** o *strings*, en inglés.

ANOTACIONES

2.7.1. Declaración

A continuación se muestran cuatro declaraciones de matrices distintas y un esquema de su distribución en la memoria del computador. El número de bytes de cada división depende del tipo de dato de cada matriz correspondiente. En el esquema ya se avanza el nombre de cada dato dentro de la matriz, en el que se distingue el nombre común de la matriz y una identificación particular del dato, que se corresponde con la posición del elemento en ella. Es muy importante tener presente que las posiciones siempre se numeran desde 0, en C.

Figura 2.



A continuación, se muestran las declaraciones de las variables que conducirían hasta la distribución en memoria que se ha visto:

```
int             vector[4];
char            cadena[20];
unsigned short  tabla[10];
double          matriz[3][3];
```

ANOTACIONES

La primera declaración prepara un vector de 4 enteros con signo; la segunda, una cadena de 20 caracteres; la tercera, una tabla de 10 enteros positivos, y la última reserva espacio para una matriz de 3×3 números reales de doble precisión.

Nota

La matriz cuadrada se almacena en la memoria por filas; es decir, primero aparece la primera fila, luego la segunda y así hasta la última.

En caso de que hubiera que declarar una matriz con un número mayor de dimensiones, bastaría con añadir su tamaño entre corchetes en la posición deseada entre el nombre de la estructura y el punto y coma final.

Como se ha comentado, las cadenas de caracteres son, realmente, matrices unidimensionales en C. Es decir, que tienen una longitud máxima fijada por el espacio reservado al vector que les corresponde. Aun así, las cadenas de caracteres pueden ser de longitud variable y, por tanto, se utiliza un marcador de final. En este caso, se trata del carácter NUL del código ASCII, cuyo valor numérico es 0. En el ejemplo anterior, la cadena puede ser cualquier texto de hasta 19 caracteres, pues es necesario prever que el último carácter es el de fin de cadena (`'\0'`).

En todos los casos, especialmente cuando se trata de variables que hayan de contener valores constantes, se puede dar un valor inicial a cada uno de los elementos que contienen.

Hay que tener presente que las matrices se guardan en memoria por filas y que es posible no especificar la primera dimensión (la que aparece inmediatamente después del nombre de la variable) de una matriz. En este caso, tomará las dimensiones necesarias para contener los datos presentes en su inicialización. El resto de dimensiones deben de estar fijadas de manera que cada elemento de la primera dimensión tenga una ocupación de memoria conocido.

En los ejemplos siguientes, podemos observar distintas inicializaciones para las declaraciones de las variables anteriores.

ANOTACIONES

```
int             vector[4] = { 0, 1, 2, 3 };
char            cadena[20] = { 'H', 'o', 'l', 'a', '\0' };
unsigned short  tabla[10] = { 98, 76, 54, 32, 1, };
double          matriz[3][3] = {{0.0, 0.1, 0.2},
                                {1.0, 1.1, 1.2},
                                {2.0, 2.1, 2.2} };
```

En el caso de la cadena de caracteres, los elementos en posiciones posteriores a la ocupada por `'\0'` no tendrán ningún valor inicial. Es más, podrán tener cualquier valor. Se trata, pues, de una inicialización incompleta.

Para facilitar la inicialización de las cadenas de caracteres también es posible hacerlo de la manera siguiente:

```
char cadena[20] = "Hola";
```

Si, además, la cadena no ha de cambiar de valor, es posible aprovechar que no es necesario especificar la dimensión, si ésta se puede calcular a través de la inicialización que se hace de la variable correspondiente:

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

En el caso de la `tabla`, se realiza una inicialización completa al indicar, con la última coma, que todos los elementos posteriores tendrán el mismo valor que el último dado.

2.7.2. Referencia

Para hacer referencia, en alguna expresión, a un elemento de una matriz, basta con indicar su nombre y la posición que ocupa dentro de ella:

$$matriz[i_0][i_1]...[i_n]$$

donde i_k son expresiones el resultado de las cuales debe de ser un valor entero. Habitualmente, las expresiones suelen ser muy simples: una variable o una constante.

ANOTACIONES

Por ejemplo, para leer de la entrada estándar los datos para la matriz de reales dobles de 3×3 que se ha declarado anteriormente, se podría hacer el programa siguiente en el que, por supuesto, las variables `fila` y `columna` son enteros positivos:

```
/* ... */
for( fila = 0; fila < 3; fila = fila + 1 ) {
   for( columna = 0; columna < 3; columna = columna + 1 ) {
     printf( "matriz[%u][%u]=? ", fila, columna );
     scanf( "%lf ", &dato );
     matriz[fila][columna] = dato;
   } /* for */
} /* for */
/* ... */
```

Es muy importante tener presente que el compilador de C no añade código para comprobar la validez de los índices de las matrices. Por lo tanto, no se comprueban los límites de las matrices y se puede hacer referencia a cualquier elemento, tanto si pertenece a la matriz como si no. ¡Esto es siempre responsabilidad del programador!

Además, en C, los corchetes son operadores de acceso a estructuras homogéneas de datos (es decir, matrices) que calculan la posición de un elemento a partir de la dirección de memoria base en la que se encuentran y el argumento que se les da. Esto implica que es posible, por ejemplo, acceder a una columna de una matriz cuadrada (por ejemplo: `int A[3][3];`) indicando sólo su primer índice (por ejemplo: `pcol = A[0];`). Más aún, es posible que se cometa el error de referirse a un elemento en la forma `A[1,2]` (común en otros lenguajes de programación). En este caso, el compilador acepta la referencia al tratarse de una forma válida de acceder a la última columna de la matriz, puesto que la coma es un operador de concatenación de expresiones cuyo resultado es el de la última expresión evaluada; es decir, para el ejemplo dado, la referencia `A[1,2]` sería, en realidad, `A[2]`.

2.7.3. Ejemplos

En este primer ejemplo, el programa comprobará si una palabra o frase corta es un palíndromo; es decir, si se lee igual de izquierda a derecha que de derecha a izquierda.

Ejemplo

Uno de los palíndromos más conocidos en castellano es el siguiente: dábale arroz a la zorra el abad.

```
#include <stdio.h>
#define LONGITUD  81
#define NULO      '\0'
main( )
{
   char            texto[LONGITUD];
   unsigned int    longitud, izq, der;
   printf( "Comprobación de palíndromos.\n" );
   printf( "Introduzca texto: ");
   gets( texto );
   longitud = 0;
   while( texto[longitud] != NULO ) {
     longitud = longitud + 1;
   } /* while */
   izq = 0;
   der = longitud;
   while( (texto[izq]==texto[der]) && (izq<der) ) {
     izq = izq + 1;
     der = der - 1;
   } /* while */
   if( izq < der ) {
     printf( "No es palíndromo.\n" );
   } else {
     printf( "¡Es palíndromo!\n" );
   } /* if */
} /* main */
```

Como `gets` toma como argumento la referencia de toda la cadena de caracteres, es decir, la dirección de la posición inicial de memoria que ocupa, no es necesario emplear el operador de "dirección de".

El siguiente programa que se muestra almacena en un vector los coeficientes de un polinomio para luego evaluarlo en un determinado punto. El polinomio tiene la forma siguiente:

$$P(x) = a_{MAX_GRADO-1}x^{MAX_GRADO} + \ldots + a_2x^2 + a_1x + a_0$$

ANOTACIONES

Los polinomios serían almacenados en un vector según la correspondencia siguiente:

$$
\begin{aligned}
\texttt{a[MAX_GRADO-1]} &= a_{MAX_GRADO-1} \\
\vdots \quad &\quad \vdots \\
\texttt{a[2]} &= a_2 \\
\texttt{a[1]} &= a_1 \\
\texttt{a[0]} &= a_0
\end{aligned}
$$

El programa deberá evaluar el polinomio para una x determinada según el método de Horner, en el cual el polinomio se trata como si estuviera expresado en la forma:

$$P(x) = (\ldots\ (a_{MAX_GRADO-1}x + a_{MAX_GRADO-2})x + \ldots + a_1)x + a_0$$

De esta manera, el coeficiente de mayor grado se multiplica por x y le suma el coeficiente del grado precedente. El resultado se vuelve a multiplicar por x, siempre que en este proceso no se haya llegado al término independiente. Si así fuera, ya se habría obtenido el resultado final.

Nota

Con este método se reduce el número de operaciones que habrá que realizar, pues no es necesario calcular ninguna potencia de x.

```
#include <stdio.h>
#define MAX_GRADO 16
main( )
{
   double  a[MAX_GRADO];
   double  x, resultado;
   int     grado, i;

   printf( "Evaluación de polinomios.\n" );
   for( i = 0; i < MAX_GRADO; i = i + 1 ) {
     a[i] = 0.0;
   } /* for */
   printf( "grado máximo del polinomio = ? ");
   scanf( "%d", &grado );
   if( ( 0 <= grado ) && ( grado < MAX_GRADO ) ) {
     for( i = 0; i <= grado; i = i + 1 ) {
       printf( "a[%d]*x^%d = ? ", i, i );
       scanf( "%lf", &x);
```

```
      a[i] = x;
    } /* for */
    printf( "x = ? " );
    scanf( "%lf", &x );
    result = 0.0;
    for( i = grado; i > 0; i = i - 1 ) {
      resultado = x * resultado + a[i-1];
    } /* for */
    printf( "P(%g) = %g\n", x, resultado, x) );
  } else {
    printf( "El grado debe estar entre 0 y %d!\n",
      MAX_GRADO-1
    ); /* printf */
  } /* if */
} /* main */
```

Es conveniente, ahora, programar estos ejemplos con el fin de adquirir una cierta práctica en la programación con matrices.

2.8. Estructuras heterogéneas

Las estructuras de datos heterogéneas son aquellas capaces de contener datos de distinto tipo. Generalmente, son agrupaciones de datos (tuplas) que forman una unidad lógica respecto de la información que procesan los programas que las usan.

Ejemplo

Las fechas (día, mes y año), los datos personales (nombre, apellidos, dirección, población, etcétera), las entradas de las guías telefónicas (número, propietario, dirección), y tantas otras.

2.8.1. Tuplas

Las tuplas son conjuntos de datos de distinto tipo. Cada elemento dentro de una tupla se identifica con un nombre de campo específico. Estas tuplas, en C, se denominan *estructuras* (`struct`).

Del mismo modo que sucede con las matrices, son útiles para organizar los datos desde un punto de vista lógico. Esta organización lógica supone poder tratar conjuntos de datos fuertemente relacionados entre sí como una única entidad. Es decir, que los programas que las empleen reflejarán su relación y, por tanto, serán mucho más inteligibles y menos propensos a errores.

ANOTACIONES

Nota

Se consigue mucha más claridad si se emplea una tupla para una fecha que si se emplean tres enteros distintos (día, mes y año). Por otra parte, las referencias a los campos de la fecha incluyen una mención a que son parte de la misma; cosa que no sucede si estos datos están contenidos en variables independientes.

Declaración

La declaración de las estructuras heterogéneas o tuplas en C empieza por la palabra clave `struct`, que debe ir seguida de un bloque de declaraciones de las variables que pertenezcan a la estructura y, a continuación, el nombre de la variable o los de una lista de variables que contendrán datos del tipo que se declara.

Dado que el procedimiento que se acaba de describir se debe repetir para declarar otras tuplas idénticas, es conveniente dar un nombre (entre `struct` y el bloque de declaraciones de sus campos) a las estructuras declaradas. Con esto, sólo es necesario incluir la declaración de los campos de la estructura en la de la primera variable de este tipo. Para las demás, será suficiente con especificar el nombre de la estructura.

Los nombres de las estructuras heterogéneas suelen seguir algún convenio para que sea fácil identificarlas. En este caso, se toma uno de los más extendidos: añadir "`_s`" como posfijo del nombre.

El ejemplo siguiente describe cómo podría ser una estructura de datos relativa a un avión localizado por un radar de un centro de control de aviación y la variable correspondiente (`avion`). Como se puede observar, no se repite la declaración de los campos de la misma en la posterior declaración de un vector de estas estructuras para contener la información de hasta `MAXNAV` aviones (se supone que es la máxima concentración de aviones posible al alcance de ese punto de control y que ha sido previamente definido):

```
struct avion_s {
       double    radio, angulo;
```

ANOTACIONES

```
        double    altura;
        char      nombre[33];
        unsigned  codigo;
} avion;
struct avion_s aviones[MAXNAV];
```

También es posible dar valores iniciales a las estructuras empleando una asignación al final de la declaración. Los valores de los distintos campos tienen que separarse mediante comas e ir incluidos entre llaves:

```
struct persona_s {
   char    nombre[ MAXLONG ];
   unsigned short edad;
} persona = { "Carmen" , 31 };
struct persona_s ganadora = { "desconocida", 0 };
struct persona_s gente[] = { { "Eva", 43 },
                             { "Pedro", 51 },
                             { "Jesús", 32 },
                             { "Anna", 37 },
                             { "Joaquín", 42 }
                           }; /* struct persona_s gente */
```

Referencia

La referencia a un campo determinado de una tupla se hace con el nombre del campo después del nombre de la variable que lo contiene, separando los dos mediante un operador de acceso a campo de estructura (el punto).

Ejemplo

```
ganadora.edad = 25;
inicial = gente[i].nombre[0];
```

En el programa siguiente se emplean variables estructuradas que contienen dos números reales para indicar un punto en el plano de forma cartesiana (`struct cartesiano_s`) y polar (`struct polar_s`). El programa pide las coordenadas cartesianas de un punto y las transforma en coordenadas polares (ángulo y radio, o distancia respecto del origen). Obsérvese que se declaran dos variables con una inicialización directa: `prec` para indicar la precisión con que se trabajará y `pi` para almacenar el valor de la constante Π en la misma precisión.

ANOTACIONES

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

main( )
{
   struct cartesiano_s { double x, y; }   c;
   struct polar_s { double radio, angulo; } p;
   double prec = 1e-9;
   double pi = 3.141592654;
   printf( "De coordenadas cartesianas a polares.\n" );
   printf( "x = ? "); scanf( "%lf", &(c.x) );
   printf( "y = ? "); scanf( "%lf", &(c.y) );
   p.radio = sqrt( c.x * c.x + c.y * c.y );
   if( p.radio < prec ) { /* si el radio es cero ...   */
    p.angulo = 0.0; /* ... el ángulo es cero.         */
   } else {
    if( -prec<c.x && c.x<prec ) { /* si c.x es cero ... */
     if( c.y > 0.0 )p.angulo = 0.5*pi;
     elsep.angulo = -0.5*pi;
    } else {
     p.angulo = atan( c.y / c.x );
    } /* if */
   } /* if */
   printf( "radio = %g\n", p.radio );
   printf( "ángulo = %g (%g grados sexagesimales)\n",
    p.angulo,
    p.angulo*180.0/pi
   ); /* printf */
} /* main */
```

El programa anterior hace uso de las funciones matemáticas estándar `sqrt` y `atan` para calcular la raíz cuadrada y el arco tangente, respectivamente. Para ello, es necesario que se incluya el fichero de cabeceras (`#include <math.h>`) correspondiente en el código fuente.

2.8.2. Variables de tipo múltiple

Son variables cuyo contenido puede variar entre datos de distinto tipo. El tipo de datos debe de estar entre alguno de los que se indican

ANOTACIONES

en su declaración y el compilador reserva espacio para contener al que ocupe mayor tamaño de todos ellos. Su declaración es parecida a la de las tuplas.

Ejemplo

```
union numero_s {
  signed entero;
  unsigned natural;
  float real;
} numero;
```

El uso de esta clase de variables puede suponer un cierto ahorro de espacio. No obstante, hay que tener presente que, para gestionar estos campos de tipo variable, es necesario disponer de información (explícita o implícita) del tipo de dato que se almacena en ellos en un momento determinado.

Así pues, suelen ir combinados en tuplas que dispongan de algún campo que permita averiguar el tipo de datos del contenido de estas variables. Por ejemplo, véase la declaración de la siguiente variable (`seguro`), en la que el campo `tipo_bien` permite conocer cuál de las estructuras del tipo múltiple está presente en su contenido:

```
struct seguro_s {
   unsigned   poliza;
   char       tomador[31];
   char       NIF[9];
   char       tipo_bien;    /* 'C': vivienda, */
                            /* 'F': vida, */
                            /* ''M': vehículo. */
   union {
     struct {
       char ref_catastro[];
       float superficie;
     } vivienda;
     struct {
       struct fecha_s nacimiento;
       char beneficiario[31];
     } vida;
     struct {
```

ANOTACIONES

```
      char matricula[7];
      struct fecha_s fabricacion;
      unsigned short siniestros;
    } vehículo;
  } datos;
  unsigned valor;
  unsigned prima;
} seguro;
```

Con ello, también es posible tener información sobre una serie de seguros en una misma tabla, independientemente del tipo de póliza que tengan asociados:

```
struct seguro_s asegurados[ NUMSEGUROS ];
```

En todo caso, el uso de `union` resulta bastante infrecuente.

2.9. Tipos de datos abstractos

Los tipos de datos abstractos son tipos de datos a los que se atribuye un significado con relación al problema que se pretende resolver con el programa y, por tanto, a un nivel de abstracción superior al del modelo computacional. Se trata, pues, de tipos de datos transparentes al compilador y, consecuentemente, irrelevantes en el código ejecutable correspondiente.

En la práctica, cualquier estructura de datos que se defina es, de hecho, una agrupación de datos en función del problema y, por lo tanto, un tipo abstracto de datos. De todas maneras, también puede serlo un entero que se emplee con una finalidad distinta; por ejemplo, sólo para almacenar valores lógicos ('cierto' o 'falso').

En cualquier caso, el uso de los tipos abstractos de datos permite aumentar la legibilidad del programa (entre otras cosas que se verán más adelante). Además, hace posible emplear declaraciones de tipos anteriormente descritos sin tener que repetir parte de la declaración, pues basta con indicar el nombre que se le ha asignado.

ANOTACIONES

2.9.1. Definición de tipos de datos abstractos

Para definir un nuevo nombre de tipo de dato es suficiente con hacer una declaración de una variable antecedida por `typedef`. En esta declaración, lo que sería el nombre de la variable será, de hecho, el nombre del nuevo tipo de datos. A continuación, se muestran varios ejemplos.

```
typedef char boolean, logico;
#define MAXSTRLEN 81
typedef char cadena[MAXSTRLEN];
typedef struct persona_s {
   cadena     nombre, direccion, poblacion;
   char       codigo_postal[5];
   unsigned   telefono;
} persona_t;
```

En las definiciones anteriores se observa que la sintaxis no varía respecto de la de la declaración de las variables salvo por la inclusión de la palabra clave `typedef`, cuyo significado es, precisamente, un apócope de "define tipo". Con estas definiciones, ya es posible declarar variables de los tipos correspondientes:

```
boolean   correcto, ok;
cadena    nombre_profesor;
persona_t alumnos[MAX_GRUPO];
```

Es muy recomendable emplear siempre un nombre de tipo que identifique los contenidos de las variables de forma significativa dentro del problema que ha de resolver el programa que las utiliza.

Por ello, a partir de este punto, todos los ejemplos emplearán tipos abstractos de datos cuando sea necesario.

Por lo que atañe a este texto, se preferirá que los nombres de tipo terminen siempre en "_t".

ANOTACIONES

2.9.2. Tipos enumerados

Los tipos de datos enumerados son un tipo de datos compatible con enteros en el que se realiza una correspondencia entre un entero y un determinado símbolo (la constante de enumeración). En otras palabras, es un tipo de datos entero en el que se da nombre (enumeran) a un conjunto de valores. En cierta manera, su empleo sustituye al comando de definición de constantes simbólicas del preprocesador (#define) cuando éstas son de tipo entero.

El ejemplo siguiente ilustra cómo se declaran y cómo se pueden emplear:

```
/* ... */
enum { ROJO, VERDE, AZUL } rgb;
enum bool_e { FALSE = 0, TRUE = 1 } logico;
enum bool_e encontrado;
int color;
/* ... */
rgb = VERDE;
/* Se puede asignar un enumerado a un entero:*/
color = ROJO;
logico = TRUE;
/* Se puede asignar un entero a un enumerado,*/
/* aunque no tenga ningún símbolo asociado:   */
encontrado = -1;
/* ... */
```

La variable enumerada `rgb` podrá contener cualquier valor entero (de tipo `int`), pero tendrá tres valores enteros identificados con los nombres `ROJO`, `VERDE` y `AZUL`. Si el valor asociado a los símbolos importa, se tiene que asignar a cada símbolo su valor mediante el signo igual, tal como aparece en la declaración de `logico`.

El tipo enumerado puede tener un nombre específico (`bool_e` en el ejemplo) que evite la repetición del enumerado en una declaración posterior de una variable del mismo tipo (en el ejemplo: `encontrado`).

ANOTACIONES

También es posible y recomendable definir un tipo de dato asociado:

```
typedef enum bool_e { FALSE = 0, TRUE = 1 } bool;
```

En este caso particular, se emplea el nombre `bool` en lugar de `bool_t` o `logico` por coincidir con el nombre del tipo de datos primitivo de C++. Dado lo frecuente de su uso, en el resto del texto se considerará definido. (No obstante, habrá que tener presente que una variable de este tipo puede adquirir valores distintos de 1 y ser, conceptualmente, 'cierta' o `TRUE`.)

2.9.3. Ejemplo

En los programas vistos con anterioridad se emplean variables de tipos de datos estructurados y que son (o podrían ser) frecuentemente usados en otros programas de la misma índole. Así pues, es conveniente transformar las declaraciones de tipo de los datos estructurados en definición de tipos.

En particular, el programa de evaluación de polinomios por el método de Horner debería haber dispuesto de un tipo de datos estructurado que representara la información de un polinomio (grado máximo y coeficientes).

El programa que se muestra a continuación contiene una definición del tipo de datos `polinomio_t` para identificar a sus componentes como datos de un mismo polinomio. El grado máximo se emplea también para saber qué elementos del vector contienen los coeficientes para cada grado y cuáles no. Este programa realiza la derivación simbólica de un polinomio dado (la derivación simbólica implica obtener otro polinomio que representa la derivada de la función polinómica dada como entrada).

```
#include <stdio.h>

#define MAX_GRADO 16

typedef struct polinomio_s {
   int      grado;
   double   a[MAX_GRADO];
} polinomio_t;
```

ANOTACIONES

```
main( )
{
   polinomio_t p;
   double      x, coef;
   int         i, grado;
   p.grado = 0; /* inicialización de (polinomio_t) p       */
   p.a[0] = 0.0;
   printf( "Derivación simbólica de polinomios.\n" );
   printf( "Grado del polinomio = ? " );
   scanf( "%d", &(p.grado) );
   if( ( 0 ⇐ p.grado ) && ( p.grado < MAX_GRADO ) ) {
     for( i = 0; i ⇐ p.grado; i = i + 1 ) { /* lectura     */
       printf( "a[%d]*x^%d = ? ", i, i );
       scanf( "%lf", &coef );
       p.a[i] = coef;
     } /* for */
     for( i = 0; i < p.grado; i = i + 1 ) { /* derivación   */
       p.a[i] = p.a[i+1]*(i+1);
     } /* for */
     if( p.grado > 0 ) {
       p.grado = p.grado -1;
     } else {
       p.a[0] = 0.0;
     } /* if */
     printf( "Polinomio derivado:\n" );
     for( i = 0; i < p.grado; i = i + 1 ) { /* impresión    */
       printf( "%g*x^%d +", p.a[i], i );
     } /* for */
     printf( "%g\n", p.a[i] );
   } else {
     printf( "¡El grado del polinomio tiene que estar" );
     printf( " entre 0 y %d!\n", MAX_GRADO-1 );
   } /* if */
} /* main */
```

ANOTACIONES

2.10. Ficheros

Ejemplo

Unidades de disquete, de disco duro, de CD, de DVD, de tarjetas de memoria, etc.

Los ficheros son una estructura de datos homogénea que tiene la particularidad de tener los datos almacenados fuera de la memoria principal. De hecho son estructuras de datos que se encuentran en la llamada memoria externa o secundaria (no obstante, es posible que algunos ficheros temporales se encuentren sólo en la memoria principal).

Para acceder a los datos de un fichero, el ordenador debe contar con los dispositivos adecuados que sean capaces de leer y, opcionalmente, escribir en los soportes adecuados.

Por el hecho de residir en soportes de información permanentes, pueden mantener información entre distintas ejecuciones de un mismo programa o servir de fuente y depósito de información para cualquier programa.

Dada la capacidad de muchos de estos soportes, el tamaño de los ficheros puede ser mucho mayor incluso que el espacio de memoria principal disponible. Por este motivo, en la memoria principal sólo se dispone de una parte del contenido de los ficheros en uso y de la información necesaria para su manejo.

No menos importante es que los ficheros son estructuras de datos con un número indefinido de éstos.

En los próximos apartados se comentarán los aspectos relacionados con los ficheros en C, que se denominan ficheros de flujo de bytes (en inglés, *byte streams*). Estos ficheros son estructuras homogéneas de datos simples en los que cada dato es un único byte. Habitualmente, los hay de dos tipos: los ficheros de texto ASCII (cada byte es un carácter) y los ficheros binarios (cada byte coincide con algún byte que forma parte de algún dato de alguno de los tipos de datos que existen).

2.10.1. Ficheros de flujo de bytes

Los ficheros de tipo *byte stream* de C son secuencias de bytes que se pueden considerar bien como una copia del contenido de la memo-

ANOTACIONES

ria (binarios), bien como una cadena de caracteres (textuales). En este apartado nos ocuparemos especialmente de estos últimos por ser los más habituales.

Dado que están almacenados en un soporte externo, es necesario disponer de información sobre los mismos en la memoria principal. En este sentido, toda la información de control de un fichero de este tipo y una parte de los datos que contiene (o que habrá de contener, en caso de escritura) se recoge en una única variable de tipo `FILE`.

El tipo de datos `FILE` es una tupla compuesta, entre otros campos, por el nombre del fichero, la longitud del mismo, la posición del último byte leído o escrito y un *buffer* (memoria temporal) que contiene `BUFSIZ` byte del fichero. Este último es necesario para evitar accesos al dispositivo periférico afectado y, dado que las operaciones de lectura y escritura se hacen por bloques de bytes, para que éstas se hagan más rápidamente.

Afortunadamente, hay funciones estándar para hacer todas las operaciones que se acaban de insinuar. Al igual que la estructura `FILE` y la constante `BUFSIZ`, están declaradas en el fichero `stdio.h`. En el próximo apartado se comentarán las más comunes.

2.10.2. Funciones estándar para ficheros

Para acceder a la información de un fichero, primero se tiene que "abrir". Es decir, hay que localizarlo y crear una variable de tipo `FILE`. Para ello, la función de apertura da como resultado de su ejecución la posición de memoria en la que se encuentra la variable que crea o `NULL` si no ha conseguido abrir el fichero indicado.

Cuando se abre un fichero, es necesario especificar si se leerá su contenido (`modo_apertura = "r"`), si se le quieren añadir mas datos (`modo_apertura = "a"`), o si se quiere crear de nuevo (`modo_apertura = "w"`).

Nota

En el tercer caso es necesario tener cuidado: si el fichero ya existiera, ¡se perdería todo su contenido!

También es conveniente indicar si el fichero es un fichero de texto (los finales de línea pueden transformarse ligeramente) o si es binario. Esto se consigue añadiendo al modo de apertura una `"t"` o una

"b", respectivamente. Si se omite esta información, el fichero se abre en modo texto.

Una vez abierto, se puede leer su contenido o bien escribir nuevos datos en él. Después de haber operado con el fichero, hay que cerrarlo. Esto es, hay que indicar al sistema operativo que ya no se trabajará más con él y, sobre todo, hay que acabar de escribir los datos que pudieran haber quedado pendientes en el *buffer* correspondiente. De todo esto se ocupa la función estándar de cierre de fichero.

En el código siguiente se refleja el esquema algorítmico para el trabajo con ficheros y se detalla, además, una función de reapertura de ficheros que aprovecha la misma estructura de datos de control. Hay que tener en cuenta que esto supone cerrar el fichero anteriormente abierto:

```
/* ... */
/* Se declara una variable para que contenga                      */
/* la referencia de la estructura FILE:                           */
FILE* fichero;
/* ... */
fichero = fopen( nombre_fichero, modo_apertura );
/* El modo de apertura puede ser "r" para lectura,                */
/* "w" para escritura, "a" para añadidura y                       */
/* "r+", "w+" o "a+" para actualización (leer/escribir).          */
/* Se le puede añadir el sufijo                                   */
/* "t" para texto o "b" para binario.                             */
if( fichero != NULL ) {
   /* Tratamiento de los datos del fichero.                       */
   /* Posible reapertura del mismo fichero:                       */
   fichero = freopen(
          nombre_fichero,
          modo_apertura,
          fichero
        ); /* freopen */
   /* Tratamiento de los datos del fichero.                       */
   fclose( fichero );
} /* if */
```

En los próximos apartados se detallan las funciones estándar para trabajar con ficheros de flujo o *streams*. Las variables que se em-

ANOTACIONES

plean en los ejemplos son del tipo adecuado para lo que se usan y, en particular, `flujo` es de tipo `FILE*`; es decir, referencia a estructura de fichero.

Funciones estándar de entrada de datos (lectura) de ficheros

Estas funciones son muy similares a las ya vistas para la lectura de datos procedentes de la entrada estándar. En éstas, sin embargo, será muy importante saber si ya se ha llegado al final del fichero y, por lo tanto, ya no hay más datos para su lectura.

`fscanf(` `flujo, "formato" [,lista_de_&variables` **`)`**

De funcionamiento similar a `scanf()`, devuelve como resultado el número de argumentos realmente leídos. Por tanto, ofrece una manera indirecta de determinar si se ha llegado al final del fichero. En este caso, de todas maneras, activa la condición de fin de fichero. De hecho, un número menor de asignaciones puede deberse, simplemente, a una entrada inesperada, como por ejemplo, leer un carácter alfabético para una conversión `"%d"`.

Por otra parte, esta función devuelve `EOF` (del inglés *end of file*) si se ha llegado a final de fichero y no se ha podido realizar ninguna asignación. Así pues, resulta mucho más conveniente emplear la función que comprueba esta condición antes que comprobarlo de forma indirecta mediante el número de parámetros correctamente leídos (puede que el fichero contenga más datos) o por el retorno de `EOF` (no se produce si se ha leído, al menos, un dato).

Ejemplo

```
fscanf( flujo, "%u%c", &num_dni, &letra_nif );
fscanf( flujo, "%d%d%d", &codigo, &precio, &cantidad );
```

`feof(` `flujo` **`)`**

Devuelve 0 en caso de que no se haya llegado al final del fichero. En caso contrario devuelve un valor distinto de cero, es decir, que es cierta la condición de final de fichero.

ANOTACIONES

fgetc(flujo **)**

Lee un carácter del flujo. En caso de no poder efectuar la lectura por haber llegado a su fin, devuelve EOF. Esta constante ya está definida en el fichero de cabeceras stdio.h; por lo tanto, puede emplearse libremente dentro del código.

Nota

Es importante tener presente que puede haber ficheros que tengan un carácter EOF en medio de su flujo, pues el final del fichero viene determinado por su longitud.

fgets(cadena, longitud_maxima, flujo **)**

Lee una cadena de caracteres del fichero hasta encontrar un final de línea, hasta llegar a longitud_maxima (–1 para la marca de final de cadena) de caracteres, o hasta fin de fichero. Devuelve NULL si encuentra el final de fichero durante la lectura.

Ejemplo

```
if( fgets( cadena, 33, flujo ) != NULL ) puts( cadena );
```

Funciones estándar de salida de datos (escritura) de ficheros

Las funciones que aquí se incluyen también tienen un comportamiento similar al de las funciones para la salida de datos por el dispositivo estándar. Todas ellas escriben caracteres en el flujo de salida indicado:

fprintf(flujo, "formato" [, lista_de_variables] **)**

La función fprintf() escribe caracteres en el flujo de salida indicado con formato. Si ha ocurrido algún problema, esta función devuelve el último carácter escrito o la constante EOF.

fputc(caracter, flujo **)**

La función fputc() escribe caracteres en el flujo de salida indicado carácter a carácter. Si se ha producido un error de escritura o bien

ANOTACIONES

el soporte está lleno, la función `fputc()` activa un indicador de error del fichero. Este indicador se puede consultar con la función `ferror( flujo )`, que retorna un cero (valor lógico falso) cuando no hay error.

`fputs(` `cadena, flujo` **`)`**

La función `fputs()` escribe caracteres en el flujo de salida indicado permitiendo grabar cadenas completas. Si ha ocurrido algún problema, esta función actúa de forma similar a `fprintf()`.

Funciones estándar de posicionamiento en ficheros de flujo

En los ficheros de flujo es posible determinar la posición de lectura o escritura; es decir, la posición del último byte leído o que se ha escrito. Esto se hace mediante la función `ftell( flujo )`, que devuelve un entero de tipo `long` que indica la posición o –1 en caso de error.

También hay funciones para cambiar esta posición de lectura (y escritura, si se trata de ficheros que hay que actualizar):

`fseek(` `flujo, desplazamiento, direccion` **`)`**

Desplaza el "cabezal" lector/escritor respecto de la posición actual con el valor del entero largo indicado en `desplazamiento` si `direccion` es igual a `SEEK_CUR`. Si esta dirección es `SEEK_SET`, entonces `desplazamiento` se convierte en un desplazamiento respecto del principio y, por lo tanto, indica la posición final. En cambio, si es `SEEK_END`, indicará el desplazamiento respecto de la última posición del fichero. Si el reposicionamiento es correcto, devuelve 0.

`rewind(` `flujo` **`)`**

Sitúa el "cabezal" al principio del fichero. Esta función es equivalente a:

```
seek( flujo, 0L, SEEK_SET );
```

ANOTACIONES

donde la constante de tipo `long int` se indica con el sufijo "L". Esta función permitiría, pues, releer un fichero desde el principio.

Relación con las funciones de entrada/salida por dispositivos estándar

Las entradas y salidas por terminal estándar también se pueden llevar a cabo con las funciones estándar de entrada/salida, o también mediante las funciones de tratamiento de ficheros de flujo. Para esto último es necesario emplear las referencias a los ficheros de los dispositivos estándar, que se abren al iniciar la ejecución de un programa. En C hay, como mínimo, tres ficheros predefinidos: `stdin` para la entrada estándar, `stdout` para la salida estándar y `stderr` para la salida de avisos de error, que suele coincidir con `stdout`.

2.10.3. Ejemplo

Se muestra aquí un pequeño programa que cuenta el número de palabras y de líneas que hay en un fichero de texto. El programa entiende como palabra toda serie de caracteres entre dos espacios en blanco. Un espacio en blanco es cualquier carácter que haga que `isspace()` devuelva cierto. El final de línea se indica con el carácter de retorno de carro (ASCII número 13); es decir, con `'\n'`. Es importante observar el uso de las funciones relacionadas con los ficheros de flujo de bytes.

Como se verá, la estructura de los programas que trabajan con estos ficheros incluye la codificación de alguno de los esquemas algorítmicos para el tratamiento de secuencias de datos (de hecho, los ficheros de flujo son secuencias de bytes) En este caso, como se realiza un conteo de palabras y líneas, hay que recorrer toda la secuencia de entrada. Por lo tanto, se puede observar que el código del programa sigue perfectamente el esquema algorítmico para el recorrido de secuencias.

```
/* Fichero: npalabras.c                                              */
#include <stdio.h>
#include <ctype.h> /* Contiene: isspace()                             */

typedef enum bool_e { FALSE = 0, TRUE = 1 } bool;
```

ANOTACIONES

```
main()
{
   char nombre_fichero[FILENAME_MAX];
   FILE *flujo;
   bool en_palabra;
   char c;
   unsigned long int npalabras, nlineas;

   printf( "Contador de palabras y líneas.\n" );
   printf( "Nombre del fichero: ");
   gets( nombre_fichero );
   flujo = fopen( nombre_fichero, "rt" );
   if( flujo != NULL ) {
    npalabras = 0;
    nlineas = 0;
    en_palabra = FALSE;
    while( ! feof( flujo ) ) {
      c = fgetc( flujo );
      if( c == '\n' ) nlineas = nlineas + 1;
      if( isspace( c ) ) {
        if( en_palabra ) {
          en_palabra = FALSE;
          npalabras = npalabras + 1;
        } /* if */
      } else { /* si el carácter no es espacio en blanco      */
        en_palabra = TRUE;
      } /* if */
    } /* while */
    printf( "Numero de palabras = %lu\n", npalabras );
    printf( "Numero de líneas = %lu\n", nlineas );
   } else {
    printf( "¡No puedo abrir el fichero!\n" );
   } /* if */
} /* main */
```

Nota

La detección de las palabras se hace comprobando los finales de palabra, que tienen que estar formadas por un carácter distinto del espacio en blanco, seguido por uno que lo sea.

ANOTACIONES

2.11. Principios de la programación modular

La lectura del código fuente de un programa implica realizar el seguimiento del flujo de ejecución de sus instrucciones (el flujo de control). Evidentemente, una ejecución en el orden secuencial de las instrucciones no precisa de mucha atención. Pero ya se ha visto que los programas contienen también instrucciones condicionales o alternativas e iterativas. Con todo, el seguimiento del flujo de control puede resultar complejo si el código fuente ocupa más de lo que se puede observar (por ejemplo, más de una veintena de líneas).

Por ello, resulta conveniente agrupar aquellas partes del código que realizan una función muy concreta en un subprograma identificado de forma individual. Es más, esto resulta incluso provechoso cuando se trata de funciones que se realizan en diversos momentos de la ejecución de un programa.

En los apartados siguientes, se verá cómo se distribuye en distintos subprogramas un programa en C. La organización es parecida en otros lenguajes de programación.

2.12. Funciones

En C, a las agrupaciones de código en las que se divide un determinado programa se las llama, precisamente, *funciones*. Más aún, en C, todo el código debe estar distribuido en funciones y, de hecho, el propio programa principal es una función: la función principal (`main`).

Generalmente, una función incluirá en su código, como mucho, la programación de unos pocos esquemas algorítmicos de procesamiento de secuencias de datos y algunas ejecuciones condicionales o alternativas. Es decir, lo necesario para realizar una tarea muy concreta.

2.12.1. Declaración y definición

La declaración de cualquier entidad (variable o función) implica la manifestación de su existencia al compilador, mientras que definirla

ANOTACIONES

supone describir su contenido. Tal diferenciación ya se ha visto para las variables, pero sólo se ha insinuado para las funciones.

La declaración consiste exactamente en lo mismo que para las variables: manifestar su existencia. En este caso, de todas maneras hay que describir los argumentos que toma y el resultado que devuelve para que el compilador pueda generar el código, y poderlas emplear.

Los ficheros de cabecera contienen declaraciones de funciones.

En cambio, la definición de una función se corresponde con su programa, que es su contenido. De hecho, de forma similar a las variables, el contenido se puede identificar por la posición del primero de sus bytes en la memoria principal. Este primer byte es el primero de la primera instrucción que se ejecuta para llevar a cabo la tarea que tenga programada.

Declaraciones

La declaración de una función consiste en especificar el tipo de dato que devuelve, el nombre de la función, la lista de parámetros que recibe entre paréntesis y un punto y coma que finaliza la declaración:

```
tipo_de_dato nombre_función( lista_de_parámetros );
```

Hay que tener presente que no se puede hacer referencia a funciones que no estén declaradas previamente. Por este motivo, es necesario incluir los ficheros de cabeceras de las funciones estándar de la biblioteca de C como `stdio.h`, por ejemplo.

Si una función no ha sido previamente declarada, el compilador supondrá que devuelve un entero. De la misma manera, si se omite el tipo de dato que retorna, supondrá que es un entero.

ANOTACIONES

La lista de parámetros es opcional y consiste en una lista de declaraciones de variables que contendrán los datos tomados como argumentos de la función. Cada declaración se separa de la siguiente por medio de una coma. Por ejemplo:

```
float nota_media( float teo, float prb, float pract );
bool aprueba( float nota, float tolerancia );
```

Si la función no devuelve ningún valor o no necesita ningún argumento, se debe indicar mediante el tipo de datos vacío (`void`):

```
void avisa( char mensaje[] );
bool si_o_no( void );
int lee_codigo( void );
```

Definiciones

La definición de una función está encabezada siempre por su declaración, que ahora debe incluir forzosamente la lista de parámetros si los tiene. Esta cabecera no debe finalizar con punto y coma, sino que irá seguida del cuerpo de la función, delimitada entre llaves de apertura y cierre:

```
tipo_de_dato nombre_función( lista_de_parámetros )
{ /* cuerpo de la función: */
   /* 1) declaración de variables locales   */
   /* 2) instrucciones de la función        */
} /* nombre_función */
```

Tal como se ha comentado anteriormente, la definición de la función ya supone su declaración. Por lo tanto, las funciones que realizan tareas de otros programas y, en particular, del programa principal (la función `main`) se definen con anterioridad.

Llamadas

El mecanismo de uso de una función en el código es el mismo que se ha empleado para las funciones de la biblioteca estándar de C: basta

ANOTACIONES

con referirse a ellas por su nombre, proporcionarles los argumentos necesarios para que puedan llevar a cabo la tarea que les corresponda y, opcionalmente, emplear el dato devuelto dentro de una expresión, que será, habitualmente, de condición o de asignación.

El procedimiento por el cual el flujo de ejecución de instrucciones pasa a la primera instrucción de una función se denomina *procedimiento de llamada*. Así pues, se hablará de llamadas a funciones cada vez que se indique el uso de una función en un programa.

A continuación se presenta la secuencia de un procedimiento de llamada:

1. Preparar el entorno de ejecución de la función; es decir, reservar espacio para el valor de retorno, los parámetros formales (las variables que se identifican con cada uno de los argumentos que tiene), y las variables locales.

2. Realizar el paso de parámetros; es decir, copiar los valores resultantes de evaluar las expresiones en cada uno de los argumentos de la instrucción de llamada a los parámetros formales.

3. Ejecutar el programa correspondiente.

4. Liberar el espacio ocupado por el entorno local y devolver el posible valor de retorno antes de regresar al flujo de ejecución de instrucciones en donde se encontraba la llamada.

El último punto se realiza mediante la instrucción de retorno que, claro está, es la última instrucción que se ejecutará en una función:

```
return expresión;
```

Nota

Esta instrucción debe aparecer vacía o no aparecer si la función es de tipo `void`; es decir, si se la ha declarado explícitamente para no devolver ningún dato.

En el cuerpo de la función se puede realizar una llamada a la misma. Esta llamada se denomina *llamada recursiva*, ya que la defi-

ANOTACIONES

nición de la función se hace en términos de ella misma. Este tipo de llamadas no es incorrecto pero hay que vigilar que no se produzcan indefinidamente; es decir, que haya algún caso donde el flujo de ejecución de las instrucciones no implique realizar ninguna llamada recursiva y, por otra parte, que la transformación que se aplica a los parámetros de éstas conduzca, en algún momento, a las condiciones de ejecución anterior. En particular, **no** se puede hacer lo siguiente:

```
/* ... */
void menu( void )
{
   /* mostrar menú de opciones,       */
   /* ejecutar opción seleccionada    */
   menu();
/* ... */
```

La función anterior supone realizar un número indefinido de llamadas a `menu()` y, por tanto, la continua creación de entornos locales sin su posterior liberación. En esta situación, es posible que el programa no pueda ejecutarse correctamente tras un tiempo por falta de memoria para crear nuevos entornos.

2.12.2. Ámbito de las variables

El ámbito de las variables hace referencia a las partes del programa que las pueden emplear. Dicho de otra manera, el ámbito de las variables abarca todas aquellas instrucciones que pueden acceder a ellas.

En el código de una función se pueden emplear todas las variables globales (las que son “visibles” por cualquier instrucción del programa), todos los parámetros formales (las variables que equivalen a los argumentos de la función), y todas las variables locales (las que se declaran dentro del cuerpo de la función).

En algunos casos puede no ser conveniente utilizar variables globales, pues dificultarían la compresión del código fuente, cosa que di-

ANOTACIONES

ficultaría el posterior depurado y mantenimiento del programa. Para ilustrarlo, veamos el siguiente ejemplo:

```
#include <stdio.h>

unsigned int A, B;

void reduce( void )
{
   if( A < B ) B = B - A;
   else A = A - B;
} /* reduce */

void main( void )
{
   printf( "El MCD de: " );
   scanf( "%u%u", &A, &B );
 while( A!=0 && B!=0 ) reduce();
   printf( "... es %u\n", A + B );
} /* main */
```

Aunque el programa mostrado tiene un funcionamiento correcto, no es posible deducir directamente qué hace la función `reduce()`, ni tampoco determinar de qué variables depende ni a cuáles afecta. Por tanto, hay que adoptar como norma que ninguna función dependa o afecte a variables globales. De ahí el hecho de que, en C, todo el código se distribuye en funciones, se deduce fácilmente que no debe haber ninguna variable global.

Así pues, todas las variables son de ámbito local (parámetros formales y variables locales). En otras palabras, se declaran en el entorno local de una función y sólo pueden ser empleadas por las instrucciones dentro de la misma.

Las variables locales se crean en el momento en que se activa la función correspondiente, es decir, después de ejecutar la instrucción de llamada de la función. Por este motivo, tienen una clase de almacenamiento denominada automática, ya que son creadas y destruidas de forma automática en el procedimiento de llamada a función. Esta

ANOTACIONES

clase de almacenamiento se puede hacer explícita mediante la palabra clave auto:

```
int una_funcion_cualquiera( int a, int b )
{
   /* ... */
   auto int variable_local;
   /* resto de la función */
} /* una_funcion_cualquiera */
```

A veces resulta interesante que la variable local se almacene temporalmente en uno de los registros del procesador para evitar tener que actualizarla continuamente en la memoria principal y acelerar, con ello, la ejecución de las instrucciones involucradas (normalmente, las iterativas). En estos casos, se puede aconsejar al compilador que genere código máquina para que se haga así; es decir, para que el almacenamiento de una variable local se lleve a cabo en uno de los registros del procesador. De todas maneras, muchos compiladores son capaces de llevar a cabo tales optimizaciones de forma autónoma.

Esta clase de almacenamiento se indica con la palabra clave register:

```
int una_funcion_cualquiera( int a, int b )
{
   /* ... */
   register int contador;
   /* resto de la función */
} /* una_funcion_cualquiera */
```

Para conseguir el efecto contrario, se puede indicar que una variable local resida siempre en memoria mediante la indicación volatile como clase de almacenamiento. Esto sólo resulta conveniente cuando la variable puede ser modificada de forma ajena al programa.

```
int una_funcion_cualquiera( int a, int b )
{
```

ANOTACIONES

```
   /* ... */
   volatile float temperatura;
   /* resto de la función */
} /* una_funcion_cualquiera */
```

En los casos anteriores, se trataba de variables automáticas. Sin embargo, a veces resulta interesante que una función pueda mantener la información contenida en alguna variable local entre distintas llamadas. Esto es, permitir que el algoritmo correspondiente "recuerde" algo de su estado pasado. Para conseguirlo, hay que indicar que la variable tiene una clase de almacenamiento `static`; es decir, que se encuentra estática o inamovible en memoria:

```
int una_funcion_cualquiera( int a, int b )
{
   /* ... */
   static unsigned numero_llamadas = 0;
   numero_llamadas = numero_llamadas + 1;
   /* resto de la función */
} /* una_funcion_cualquiera */
```

En el caso anterior es muy importante inicializar las variables en la declaración; de lo contrario, no se podría saber el contenido inicial, previo a cualquier llamada a la función.

Como nota final, indicar que las clases de almacenamiento se utilizan muy raramente en la programación en C. De hecho, a excepción de `static`, las demás prácticamente no tienen efecto en un compilador actual.

2.12.3. Parámetros por valor y por referencia

El paso de parámetros se refiere a la acción de transformar los parámetros formales a parámetros reales; es decir, de asignar un contenido a las variables que representan a los argumentos:

```
tipo funcion_llamada(
   parámetro_formal_1,
   parámetro_formal_2,
   ...
);
```

ANOTACIONES

```
funcion_llamadora( ... )
{
   /* ... */
   funcion_llamada( parámetro_real_1, parámetro_real_2, ... )
   /* ... */
} /* funcion_llamadora */
```

En este sentido, hay dos posibilidades: que los argumentos reciban el resultado de la evaluación de la expresión correspondiente o que se sustituyan por la variable que se indicó en el parámetro real de la misma posición. El primer caso, se trata de un **paso de parámetros por valor**, mientras que el segundo, se trata de un **paso de variable** (cualquier cambio en el argumento es un cambio en la variable que consta como parámetro real).

El paso por valor consiste en asignar a la variable del parámetro formal correspondiente el valor resultante del parámetro real en su misma posición. El paso de variable consiste en sustituir la variable del parámetro real por la del parámetro formal correspondiente y, consecuentemente, poder emplearla dentro de la misma función con el nombre del parámetro formal.

En C, el paso de parámetros sólo se efectúa por valor; es decir, se evalúan todos los parámetros en la llamada y se asigna el resultado al parámetro formal correspondiente en la función.

Para modificar alguna variable que se desee pasar como argumento en la llamada de una función, es necesario pasar la dirección de memoria en la que se encuentra. Para esto se debe de emplear el operador de obtención de dirección (&) que da como resultado la dirección de memoria en la que se encuentra su argumento (variable, campo de tuplo o elemento de matriz, entre otros). Este es el mecanismo que se emplea para que la función `scanf` deposite en las variables que se le pasan como argumento los valores que lee.

ANOTACIONES

Por otra parte, en las funciones llamadas, los parámetros formales que reciben una referencia de una variable en lugar de un valor se deben declarar de manera especial, anteponiendo a su nombre un asterisco. El asterisco, en este contexto, se puede leer como el "contenido cuya posición inicial se encuentra en la variable correspondiente". Por tanto, en una función como la mostrada a continuación, se leería "el contenido cuya posición inicial se encuentra en el parámetro formal numerador" es de tipo entero. De igual forma se leería para el denominador:

```
void simplifica( int *numerador, int *denominador )
{
   int mcd;
   mcd=maximo_comun_divisor( *numerador, *denominador );
   *numerador = *numerador / mcd;
   *denominador = *denominador / mcd;
} /* simplifica */
/* ... */
   simplifica( &a, &b );
/* ... */
```

Aunque se insistirá en ello más adelante, hay que tener presente que el asterisco, en la parte del código, debería leerse como "el contenido de la variable que está almacenada en la posición de memoria del argumento correspondiente". Por lo tanto, deberemos emplear `*parámetro_formal` cada vez que se desee utilizar la variable pasada por referencia.

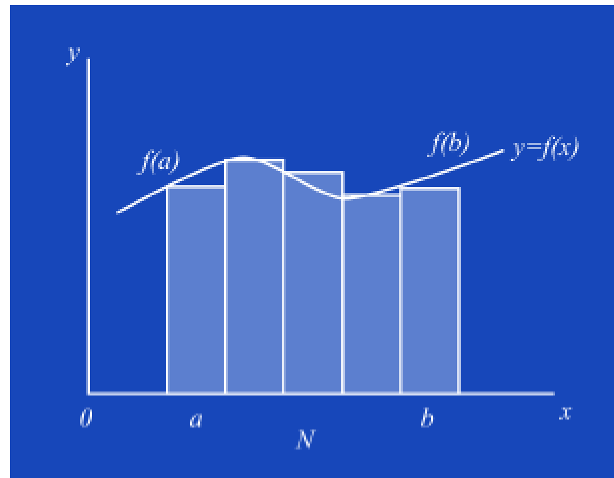
2.12.4. Ejemplo

El programa siguiente calcula numéricamente la integral de una función en un intervalo dado según la regla de Simpson. Básicamente, el método consiste en dividir el intervalo de integración en un número determinado de segmentos de la misma longitud que constituyen la base de unos rectángulos cuya altura viene determinada por el valor de la función a integrar en el punto inicial del segmento. La suma de las áreas de estos rectángulos dará la superficie aproximada defini-

ANOTACIONES

da por la función, el eje de las X y las rectas perpendiculares a él que pasan por los puntos inicial y final del segmento de integración:

Figura 3.




```
/* Fichero: simpson.c                                        */

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

double f( double x )
{
   return 1.0/(1.0 + x*x);
} /* f */

double integral_f( double a, double b, int n )
{
   double result;
   double x, dx;
   int    i;

   result = 0.0;
   if( (a < b) && (n > 0) ) {
     x = a;
     dx = (b-a)/n;
     for( i = 0; i < n; i = i + 1 ) {
       result = result + f(x);
```

ANOTACIONES

```
      x = x + dx;
    } /* for */
  } /* if */
  return result;
} /* integral_f */

void main( void )
{
  double  a, b;
  int     n;

  printf( "Integración numérica de f(x).\n" );
  printf( "Punto inicial del intervalo, a = ? " );
  scanf( "%lf", &a );
  printf( "Punto final del intervalo, b = ? " );
  scanf( "%lf", &b );
  printf( "Número de divisiones, n = ? " );
  scanf( "%d", &n );
  printf(
    "Resultado, integral(f)[%g,%g] = %g\n",
    a, b, integral_f( a, b, n )
  ); /* printf */
} /* main */
```

2.13. Macros del preprocesador de C

El preprocesador no sólo realiza sustituciones de símbolos simples como las que se han visto. También puede efectuar sustituciones con parámetros. A las definiciones de sustituciones de símbolos con parámetros se las llama “macros”:

```
#define símbolo expresión_constante
#define macro( argumentos ) expresión_const_con_argumentos
```

El uso de las macros puede ayudar a la clarificación de pequeñas partes del código mediante el uso de una sintaxis similar a la de las llamadas a las funciones.

ANOTACIONES

De esta manera, determinadas operaciones simples pueden beneficirse de un nombre significativo en lugar de emplear unas construcciones en C que pudieran dificultar la comprensión de su intencionalidad.

Ejemplo

```
#define absoluto( x ) ( x < 0 ? -x : x )
#define redondea( x ) ( (int) ( x + 0.5) )
#define trunca( x )   ( (int) x )
```

Hay que tener presente que el nombre de la macro y el paréntesis izquierdo no pueden ir separados y que la continuación de la línea, caso de que el comando sea demasiado largo) se hace colocando una barra invertida justo antes del carácter de salto de línea.

Por otra parte, hay que advertir que las macros hacen una sustitución de cada nombre de parámetro aparecido en la definición por la parte del código fuente que se indique como argumento. Así pues:

```
absoluto( 2*entero + 1 )
```

se sustituiría por:

```
( 2*entero + 1 < 0 ? -2*entero + 1 : 2*entero + 1 )
```

con lo que no sería correcto en el caso de que fuera negativo.

Nota

En este caso, sería posible evitar el error si en la definición se hubieran puesto paréntesis alrededor del argumento.

2.14. Resumen

La organización del código fuente es esencial para confeccionar programas legibles que resulten fáciles de mantener y de actualizar. Esto

ANOTACIONES

es especialmente cierto para los programas de código abierto, es decir, para el software libre.

En esta unidad se han repasado los aspectos fundamentales que intervienen en un código organizado. En esencia, la organización correcta del código fuente de un programa depende tanto de las instrucciones como de los datos. Por este motivo, no sólo se ha tratado de cómo organizar el programa sino que además se ha visto cómo emplear estructuras de datos.

La organización correcta del programa empieza por que éste tenga un flujo de ejecución de sus instrucciones claro. Dado que el flujo de instrucciones más simple es aquél en el que se ejecutan de forma secuencial según aparecen en el código, es fundamental que las instrucciones de control de flujo tengan un único punto de entrada y un único punto de salida. En este principio se basa el método de la programación estructurada. En este método, sólo hay dos tipos de instrucciones de control de flujo: las alternativas y las iterativas.

Las instrucciones iterativas suponen otro reto en la determinación del flujo de control, ya que es necesario determinar que la condición por la que se detiene la iteración se cumple alguna vez. Por este motivo, se han repasado los esquemas algorítmicos para el tratamiento de secuencias de datos y se han visto pequeños programas que, además de servir de ejemplo de programación estructurada, son útiles para realizar operaciones de filtro de datos en tuberías (procesos encadenados).

Para organizar correctamente el código y hacer posible el tratamiento de información compleja es necesario recurrir a la estructuración de los datos. En este aspecto, hay que tener presente que el programa debe reflejar aquellas operaciones que se realizan en la información y no tanto lo que supone llevar a cabo los datos elementales que la componen. Por este motivo, no sólo se ha explicado cómo declarar y emplear datos estructurados, sean éstos de tipo homogéneo o heterogéneo, sino que también se ha detallado cómo definir nuevos tipos de datos a partir de los tipos de datos básicos y estructurados. A estos nuevos tipos de datos se los llama *tipos abstractos de datos* pues son transparentes para el lenguaje de programación.

ANOTACIONES

Al hablar de los datos también se ha tratado de los ficheros de flujo de bytes. Estas estructuras de datos homogéneas se caracterizan por tener un número indefinido de elementos, por residir en memoria secundaria, es decir, en algún soporte de información externo y, finalmente, por requerir de funciones específicas para acceder a sus datos. Así pues, se han comentado las funciones estándar en C para operar con este tipo de ficheros. Fundamentalmente, los programas que los usan implementan esquemas algorítmicos de recorrido o de búsqueda en los que se incluye una inicialización específica para abrir los ficheros, una comprobación de final de fichero para la condición de iteración, operaciones de lectura y escritura para el tratamiento de la secuencia de datos y, para acabar, una finalización que consiste, entre otras cosas, en cerrar los ficheros empleados.

El último apartado se ha dedicado a la programación modular, que consiste en agrupar las secuencias de instrucciones en subprogramas que realicen una función concreta y susceptible de ser empleado más de una vez en el mismo programa o en otros. Así pues, se sustituye en el flujo de ejecución todo el subprograma por una instrucción que se ocupará de ejecutar el subprograma correspondiente. Estos subprogramas se denominan “funciones” en C y a la instrucción que se ocupa de ejecutarlas, “instrucción de llamada”. Se ha visto cómo se lleva a cabo una llamada a una función y que, en este aspecto, lo más importante es el paso de parámetros.

El paso de parámetros consiste en transmitir a una función el conjunto de datos con los que deberá realizar su tarea. Dado que la función puede necesitar devolver resultados que no se puedan almacenar en una variable simple, algunos de estos parámetros se emplean para pasar referencias a variables que también podrán contener valores de retorno. Así pues, se ha analizado también toda la problemática relacionada con el paso de parámetros por valor y por referencia.

2.15. Ejercicios de autoevaluación

1) Haced un programa para determinar el número de dígitos necesarios para representar a un número entero dado. El algoritmo

ANOTACIONES

consiste en hacer divisiones enteras por 10 del número hasta que el resultado sea un valor inferior a 10.

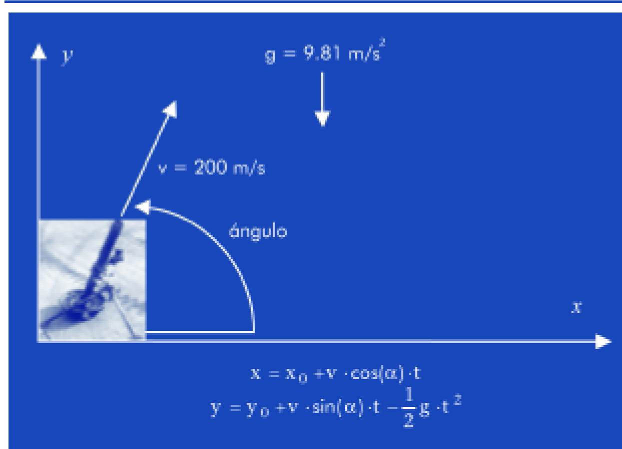
2) Haced un programa que determine a cada momento la posición de un proyectil lanzado desde un mortero. Se deberá mostrar la altura y la distancia a intervalos regulares de tiempo hasta que alcance el suelo. Para ello, se supondrá que el suelo es llano y que se le proporcionan, como datos de entrada, el ángulo del cañón y el intervalo de tiempo en el que se mostrarán los datos de salida. Se supone que la velocidad de salida de los proyectiles es de 200 m/s.

Nota

Para más detalles, se puede tener en cuenta que el tubo del mortero es de 1 m de longitud y que los ángulos de tiro varían entre 45 y 85 grados.

En el siguiente esquema se resumen las distintas fórmulas que son necesarias para resolver el problema:

Figura 4.



donde x_0 e y_0 es la posición inicial (puede considerarse 0 para los dos), y α es el ángulo en radianes (Π radianes = 180º).

3) Se desea calcular el capital final acumulado de un plan de pensiones sabiendo el capital inicial, la edad del asegurado (se supone que se jubilará a los 65) y las aportaciones y los porcentajes de interés rendidos de cada año. (Se supone que las aportaciones son de carácter anual.)

ANOTACIONES

4) Programad un filtro que calcule la media, el máximo y el mínimo de una serie de números reales de entrada.

5) Implementad los filtros del último ejemplo del apartado "2.4.2. Filtros y tuberías"; es decir: calculad los importes de una secuencia de datos { código de artículo, precio, cantidad } que genere otra secuencia de datos { código de artículo, importe } y realizad, posteriormente, la suma de los importes de esta segunda secuencia de datos.

6) Haced un programa que calcule la desviación típica que tienen los datos de ocupación de un aparcamiento público a lo largo de las 24 horas de un día. Habrá, por tanto, 24 datos de entrada.

Estos datos se refieren al porcentaje de ocupación (número de plazas ocupadas con relación al número total de plazas) calculado al final de cada hora. También se deberá indicar las horas del día que tengan un porcentaje de ocupación inferior a la media menos dos veces la desviación típica, y las que lo tengan superior a la media más dos veces esta desviación.

Nota

La desviación típica se calcula como la raíz cuadrada de la suma de los cuadrados de las diferencias entre los datos y la media, dividida por el número de muestras.

7) Averiguad si la letra de un NIF dado es o no correcta. El procedimiento de su cálculo consiste en realizar el módulo 23 del número correspondiente. El resultado da una posición en una secuencia de letras (TRWAGMYFPDXBNJZSQVHLCKE). La letra situada en dicha posición será la letra del NIF.

Nota

Para poder efectuar la comparación entre letras, es conveniente convertir la que proporcione el usuario a mayúscula. Para ello se debe emplear `toupper()`, cuya declaración está en `ctype.h` y que devuelve el carácter correspondiente a la letra mayúscula de la que ha recibido como argumento. Si no es un carácter alfabético o se trata de una letra ya mayúscula, devuelve el mismo carácter.

ANOTACIONES

8) Haced un programa que calcule el mínimo número de monedas necesario para devolver el cambio sabiendo el importe total a cobrar y la cantidad recibida como pago. La moneda de importe máximo es la de 2 euros y la más pequeña, de 1 céntimo.

Nota

Es conveniente tener un vector con los valores de las monedas ordenados por valor.

9) Resumid la actividad habida en un terminal de venta por artículos. El programa debe mostrar, para cada código de artículo, el número de unidades vendidas. Para ello, contará con un fichero generado por el terminal que consta de pares de números enteros: el primero indica el código del artículo y el segundo, la cantidad vendida. En caso de devolución, la cantidad aparecerá como un valor negativo. Se sabe, además, que no habrá nunca más de 100 códigos de artículos diferentes.

Nota

Es conveniente disponer de un vector de 100 tuplas para almacenar la información de su código y las unidades vendidas correspondientes. Como no se sabe cuántas tuplas serán necesarias, téngase en cuenta que se deberá disponer de una variable que indique los que se hayan almacenado en el vector (de 0 al número de códigos distintos -1).

10) Reprogramad el ejercicio anterior de manera que las operaciones que afecten al vector de datos se lleven a cabo en el cuerpo de funciones específicas.

Nota

Definid un tipo de dato nuevo que contenga la información de los productos. Se sugiere, por ejemplo, el que se muestra a continuación.

```
typedef struct productos_s {
  unsigned int n; /* Número de productos. */
  venta_t producto[MAX_PRODUCTOS];
} productos_t;
```

ANOTACIONES

Recuérdese que habrá de pasar la variable de este tipo por referencia.

11) Buscad una palabra en un fichero de texto. Para ello, realizad un programa que pida tanto el texto de la palabra como el nombre del fichero. El resultado deberá ser un listado de todas las líneas en que se encuentre dicha palabra.

Se supondrá que una palabra es una secuencia de caracteres alfanuméricos. Es conveniente emplear la macro `isalnum()`, que se encuentra declarada en `ctype.h`, para determinar si un carácter es o no alfanumérico.

Nota

En la solución propuesta, se emplean las funciones que se declaran a continuación.

```
#define LONG_PALABRA 81
typedef char palabra_t[LONG_PALABRA];
bool palabras_iguales( palabra_t p1, palabra_t p2 );
unsigned int lee_palabra( palabra_t p, FILE *entrada );
void primera_palabra( palabra_t palabra, char *frase );
```

2.15.1. Solucionario

1)

```
/* ---------------------------------------------------- */
/* Fichero: ndigitos.c                                   */
/* ---------------------------------------------------- */

#include <stdio.h>

main()
{
 unsigned int numero;
 unsigned int digitos;

   printf( "El número de dígitos para representar: " );
   scanf( "%u", &numero );
   digitos = 1;
```

ANOTACIONES

```
   while( numero > 10 ) {
     numero = numero / 10;
     digitos = digitos + 1;
   } /* while */
   printf( "... es %u.\n", digitos );
} /* main */
```

2)

```
/* ---------------------------------------------------    */
/* Fichero: mortero.c                                      */
/* ---------------------------------------------------    */

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

#define V_INICIAL 200.00       /* m/s                      */
#define L_TUBO    1.0          /* m                        */
#define G         9.81         /* m/(s*s)                  */
#define PI        3.14159265

main()
{
   double angulo, inc_tiempo, t;
   double v_x, v_y; /* Velocidades horizontal y vertical.  */
   double x0, x, y0, y;

   printf( "Tiro con mortero.\n " );
   printf( "Ángulo de tiro [grados sexagesimales] =? " );
   scanf( "%lf", &angulo );
   angulo = angulo * PI / 180.0;
   printf( "Ciclo de muestra [segundos] =? " );
   scanf( "%lf", &inc_tiempo );
   x0 = L_TUBO * cos( angulo );
   y0 = L_TUBO * sin( angulo );
   t = 0.0;
   v_x = V_INICIAL * cos( angulo );
   v_y = V_INICIAL * sin( angulo );
   do {
     x = x0 + v_x * t;
     y = y0 + v_y * t - 0.5 * G * t * t;
     printf( "%6.2lf s: ( %6.2lf, %6.2lf )\n", t, x, y );
```

ANOTACIONES

```
    t = t + inc_tiempo;
  } while ( y > 0.0 );
} /* main */
```

3)

```
/* ---------------------------------------------------- */
/* Fichero: pensiones.c                                  */
/* ---------------------------------------------------- */
#include <stdio.h>

#define EDAD_JUBILACION 65

main()
{
  unsigned int edad;
  float        capital, interes, aportacion;

  printf( "Plan de pensiones.\n " );
  printf( "Edad =? " );
  scanf( "%u", &edad );
  printf( "Aportación inicial =? " );
  scanf( "%f", &capital );
  while( edad < EDAD_JUBILACION ) {
   printf( "Interés rendido [en %%] =? " );
   scanf( "%f", &interes );
   capital = capital*( 1.0 + interes/100.0 );
   printf( "Nueva aportación =? " );
  scanf( "%f", &aportacion );
   capital = capital + aportacion;
   edad = edad + 1;
   printf( "Capital acumulado a %u años: %.2f\n",
   edad, capital
   ); /* printf */
  } /* while */
} /* main */
```

4)

```
/* ---------------------------------------------------- */
/* Fichero: estadistica.c                                */
/* ---------------------------------------------------- */

#include <stdio.h>
```

ANOTACIONES

```
main()
{
   double       suma, minimo, maximo;
   double       numero;
   unsigned int cantidad, leido_ok;

   printf( "Mínimo, media y máximo.\n " );
   leido_ok = scanf( "%lf", &numero );
   if( leido_ok == 1 ) {
     cantidad = 1;
     suma = numero;
     minimo = numero;
     maximo = numero;
     leido_ok = scanf( "%lf", &numero );
     while( leido_ok == 1 ) {
       suma = suma + numero;
       if( numero > maximo ) maximo = numero;
       if( numero < minimo ) minimo = numero;
       cantidad = cantidad + 1;
       leido_ok = scanf( "%lf", &numero );
     } /* while */
     printf( "Mínimo = %g\n", minimo );
     printf( "Media = %g\n", suma / (double) cantidad );
     printf( "Máximo = %g\n", maximo );
   } else {
     printf( "Entrada vacía.\n" );
   } /* if */
} /* main */
```

5)

```
/* ----------------------------------------------------- */
/* Fichero: calc_importes.c                                */
/* ----------------------------------------------------- */
#include <stdio.h>
main()
{
   unsigned int leidos_ok, codigo;
   int          cantidad;
   float        precio, importe;

   leidos_ok = scanf( "%u%f%i",
```

ANOTACIONES

```
    &codigo, &precio, &cantidad
   ); /* scanf */
   while( leidos_ok == 3 ) {
    importe = (float) cantidad * precio;
    printf( "%u %.2f\n", codigo, importe );
    leidos_ok = scanf( "%u%f%i",
     &codigo, &precio, &cantidad
    ); /* scanf */
   } /* while */
} /* main */

/* ----------------------------------------------------- */
/* Fichero: totaliza.c                                   */
/* ----------------------------------------------------- */
#include <stdio.h>
main()
{
   unsigned int leidos_ok, codigo;
   int          cantidad;
   float        importe;
   double       total = 0.0;

   leidos_ok = scanf( "%u%f", &codigo, &importe );
   while( leidos_ok == 2 ) {
    total = total + importe;
    leidos_ok = scanf( "%u%f", &codigo, &importe );
   } /* while */
   printf( "%.2lf\n", total );
} /* main */
```

6)

```
/* ----------------------------------------------------- */
/* Fichero: ocupa_pk.c                                   */
/* ----------------------------------------------------- */

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

main()
{
   unsigned int hora;
```

ANOTACIONES

```
float        porcentaje;
float        ratio_acum[24];
double       media, desv, dif;

printf( "Estadística de ocupación diaria:\n" );
/* Lectura de ratios de ocupación por horas:          */
for( hora = 0; hora < 24; hora = hora + 1 ) {
  printf(
    "Porcentaje de ocupación a las %02u horas =? ",
    hora
  ); /* printf */
  scanf( "%f", &porcentaje );
  ratio_acum[hora] = porcentaje;
} /* for */
/* Cálculo de la media:                               */
  media = 0.0;
for( hora = 0; hora < 24; hora = hora + 1 ) {
  media = media + ratio_acum[hora];
} /* for */
media = media / 24.0;
/* Cálculo de la desviación típica:                   */
desv = 0.0;
for( hora = 0; hora < 24; hora = hora + 1 ) {
  dif = ratio_acum[ hora ] - media;
  desv = desv + dif * dif;
} /* for */
desv = sqrt( desv ) / 24.0;
/* Impresión de los resultados:                       */
printf( "Media de ocupación en el día: %.2lf\n", media );
printf( "Desviación típica: %.2lf\n", desv );
printf( "Horas con porcentaje muy bajo: ", desv );
for( hora = 0; hora < 24; hora = hora + 1 ) {
  dif = media - 2*desv;
  if( ratio_acum[ hora ] < dif ) printf( "%u ", hora );
} /* for */
printf( "\nHoras con porcentaje muy alto: ", desv );
for( hora = 0; hora < 24; hora = hora + 1 ) {
  dif = media + 2*desv;
  if( ratio_acum[ hora ] > dif ) printf( "%u ", hora );
} /* for */
printf( "\nFin.\n" );
```

ANOTACIONES

```
} /* main */
```

7)

```
/* -------------------------------------------------------  */
/* Fichero: valida_nif.c                                    */
/* -------------------------------------------------------  */

#include <stdio.h>
#include <ctype.h>
main()
{
   char          NIF[ BUFSIZ ];
   unsigned int DNI;
   char          letra;
   char          codigo[] = "TRWAGMYFPDXBNJZSQVHLCKE" ;

   printf( "Dime el NIF (DNI-letra): " );
   gets( NIF );
   sscanf( NIF, "%u", &DNI );
   DNI = DNI % 23;
   letra = NIF[ strlen(NIF)-1 ];
   letra = toupper( letra );
   if( letra == codigo[ DNI ] ) {
    printf( "NIF válido.\n" );
   } else {
    printf( "¡Letra %c no válida!\n", letra );
   } /* if */
} /* main */
```

8)

```
/* -------------------------------------------------------  */
/* Fichero: cambio.c                                        */
/* -------------------------------------------------------  */

#include <stdio.h>

#define NUM_MONEDAS 8

main()
{
   double        importe, pagado;
   int           cambio_cents;
```

ANOTACIONES

```
   int          i, nmonedas;
   int          cents[NUM_MONEDAS] = { 1, 2, 5, 10,
                20, 50, 100, 200 };

   printf( "Importe : " );
   scanf( "%lf", &importe );
   printf( "Pagado : " );
   scanf( "%lf", &pagado );
   cambio_cents = (int) 100.00 * ( pagado - importe );
   if( cambio_cents < 0 ) {
    printf( "PAGO INSUFICIENTE\n");
   } else {
    printf( "A devolver : %.2f\n",
      (float) cambio_cents / 100.0
    ); /* printf */
    i = NUM_MONEDAS - 1;
    while( cambio_cents > 0 ) {
     nmonedas = cambio_cents / cents[i] ;
     if( nmonedas > 0 ) {
      cambio_cents = cambio_cents - cents[i]*nmonedas;
      printf( "%u monedas de %.2f euros.\n",
        nmonedas,
        (float) cents[i] / 100.0
      ); /* printf */
     } /* if */
     i = i - 1;
    } /* while */
   } /* if */
} /* main */
```

9)

```
/* -------------------------------------------------------- */
/* Fichero: resumen_tpv.c                                    */
/* -------------------------------------------------------- */

#include <stdio.h>

typedef struct venta_s {
   unsigned int codigo;
   int cantidad;
} venta_t;
```

ANOTACIONES

```
#define MAX_PRODUCTOS 100

main()
{
   FILE          *entrada;
   char          nombre[BUFSIZ];
   unsigned int leidos_ok;
   int           num_prod, i;
   venta_t       venta, producto[MAX_PRODUCTOS];

   printf( "Resumen del día en TPV.\n" );
   printf( "Fichero: ");
   gets( nombre );
   entrada = fopen( nombre, "rt" );
   if( entrada != NULL ) {
    num_prod = 0;
    leidos_ok = fscanf( entrada, "%u%i",
      &(venta.codigo), &(venta.cantidad)
    ); /* fscanf */
    while( leidos_ok == 2 ) {

      /* Búsqueda del código en la tabla:                    */
      i = 0;
      while( ( i < num_prod ) &&
           ( producto[i].codigo != venta.codigo )
      ) {
        i = i + 1;
      } /* while */
      if( i < num_prod ) { /* Código encontrado:             */
        producto[i].cantidad =producto[i].cantidad +
                              venta.cantidad;
      } else { /* Código no encontrado ⇒ nuevo producto:   */
        producto[num_prod].codigo = venta.codigo;
        producto[num_prod].cantidad = venta.cantidad;
        num_prod = num_prod + 1;
      } /* if */
      /* Lectura de siguiente venta:                         */
      leidos_ok = fscanf( entrada, "%u%i",
        &(venta.codigo), &(venta.cantidad)
      ); /* fscanf */
    } /* while */
```

ANOTACIONES

```
    printf( "Resumen:\n" );
    for( i=0; i<num_prod; i = i + 1 ) {
      printf( "%u : %i\n",
        producto[i].codigo, producto[i].cantidad
      ); /* printf */
    } /* for */
  } else {
    printf( "¡No puedo abrir %s!\n", nombre );
  } /* if */
} /* main */
```

10)

```
/* ... */

int busca( unsigned int codigo, productos_t *pref )
{
   int i;

   i = 0;
   while( (i < (*pref).n ) &&
      ( (*pref).producto[i].codigo != codigo )
   ) {
      i = i + 1;
   } /* while */
   if( i == (*pref).n ) i = -1;
   return i;
} /* busca */

void anyade( venta_t venta, productos_t *pref )
{
   unsigned int n;

   n = (*pref).n;
   if( n < MAX_PRODUCTOS ) {
     (*pref).producto[n].codigo = venta.codigo;
     (*pref).producto[n].cantidad = venta.cantidad;
     (*pref).n = n + 1;
   } /* if */
} /* anyade */

void actualiza( venta_t venta, unsigned int posicion,
productos_t *pref )
```

ANOTACIONES

```
{
   (*pref).producto[posicion].cantidad =
     (*pref).producto[posicion].cantidad + venta.cantidad;
} /* actualiza */

void muestra( productos_t *pref )
{
   int i;

   for( i=0; i<(*pref).n; i = i + 1 ) {
     printf( "%u : %i\n",
       (*pref).producto[i].codigo,
       (*pref).producto[i].cantidad
     ); /* printf */
   } /* for */
} /* muestra */

void main( void )
{
   FILE          *entrada;
   char          nombre[BUFSIZ];
   unsigned int leidos_ok;
   int           i;
   venta_t       venta;
   productos_t  productos;

   printf( "Resumen del día en TPV.\n" );
   printf( "Fichero: ");
   gets( nombre );
   entrada = fopen( nombre, "rt" );
   if( entrada != NULL ) {
     productos.n = 0;
     leidos_ok = fscanf( entrada, "%u%i",
       &(venta.codigo), &(venta.cantidad)
     ); /* scanf */
     while( leidos_ok == 2 ) {
       i = busca( venta.codigo, &productos );
       if( i < 0 )  anyade( venta, &productos );
       else         actualiza( venta, i, &productos );
       leidos_ok = fscanf( entrada, "%u%i",
         &(venta.codigo), &(venta.cantidad)
```

ANOTACIONES

```
      ); /* scanf */
     } /* while */
     printf( "Resumen:\n" );
     muestra( &productos );
   } else {
     printf( "¡No puedo abrir %s!\n", nombre );
   } /* if */
} /* main */
```

11)

```
/* ----------------------------------------------------- */
/* Fichero: busca.c                                       */
/* ----------------------------------------------------- */

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

typedef enum bool_e { FALSE = 0, TRUE = 1 } bool;

#define LONG_PALABRA 81

typedef char palabra_t[LONG_PALABRA];

bool palabras_iguales( palabra_t p1, palabra_t p2 )
{
   int i = 0;

   while( (p1[i]!='\0') && (p2[i]!='\0') && (p1[i]==p2[i])) {
     i = i + 1;
   } /* while */
   return p1[i]==p2[i];
} /* palabras_iguales */

unsigned int lee_palabra( palabra_t p, FILE *entrada )
{
   unsigned int i, nlin;
   bool         termino;
   char         caracter;
```

ANOTACIONES

```
   i = 0;
   nlin = 0;
   termino = FALSE;
   caracter = fgetc( entrada );
   while( !feof( entrada ) && !termino ) {
     if( caracter == '\n' ) nlin = nlin + 1;
     if( isalnum( caracter ) ) {
       p[i] = caracter;
       i = i + 1;
       caracter = fgetc( entrada );
     } else {
       if( i > 0 ) {
         termino = TRUE;
       } else {
         caracter = fgetc( entrada );
       } /* if */
     } /* if */
   } /* while */
   p[i] = '\0';
   return nlin;
} /* lee_palabra */

void primera_palabra( palabra_t palabra, char *frase )

{
   int i, j;

   i = 0;
   while( frase[i]!='\0' && isspace( frase[i] ) ) {
     i = i + 1;
   } /* while */
   j = 0;
   while( frase[i]!='\0' && !isspace( frase[i] ) ) {
     palabra[j] = frase[i];
     i = i + 1;
     j = j + 1;
   } /* while */
   palabra[j] = '\0';
} /* primera_palabra */

void main( void )
```

ANOTACIONES

```
{
   FILE          *entrada;
   char          nombre[BUFSIZ];
   palabra_t     palabra, palabra2;
   unsigned int  numlin;

   printf( "Busca palabras.\n" );
   printf( "Fichero: ");
   gets( nombre );
   entrada = fopen( nombre, "rt" );
   if( entrada != NULL ) {
     printf( "Palabra: ");
     gets( nombre );
     primera_palabra( palabra, nombre );
     printf( "Buscando %s en fichero...\n", palabra );
     numlin = 1;
     while( !feof( entrada ) ) {
       numlin = numlin + lee_palabra( palabra2, entrada );
       if( palabras_iguales( palabra, palabra2 ) ) {
         printf( "... línea %lu\n", numlin );
       } /* if */
     } /* while */
   } else {
     printf( "¡No puedo abrir %s!\n", nombre );
   } /* if */
} /* main */
```

3. Programación avanzada en C. Desarrollo eficiente de aplicaciones

3.1. Introducción

La programación de una aplicación informática suele dar como resultado un código fuente de tamaño considerable. Aun aplicando técnicas de programación modular y apoyándose en funciones estándar de biblioteca, resulta complejo organizar eficazmente el código. Mucho más si se tiene en cuenta que, habitualmente, se trata de un código desarrollado por un equipo de programadores.

Por otra parte, hay que tener presente que mucha de la información con que se trabaja no tiene un tamaño predefinido ni, la mayoría de las veces, se presenta en la forma más adecuada para ser procesada. Esto comporta tener que reestructurar los datos de manera que los algoritmos que los tratan puedan ser más eficientes.

Como último elemento a considerar, pero no por ello menos importante, se debe tener en cuenta que la mayoría de aplicaciones están constituidas por más de un programa. Por lo tanto, resulta conveniente organizarlos aprovechando las facilidades que para ello nos ofrece tanto el conjunto de herramientas de desarrollo de software como el sistema operativo en el que se ejecutará.

En esta unidad se tratará de diversos aspectos que alivian los problemas antes mencionados. Así pues, desde el punto de vista de un programa en el contexto de una aplicación que lo contenga (o del que sea el único), es importante la adaptación al tamaño real de los datos a procesar y su disposición en estructuras dinámicas, la organización del código para que refleje el algoritmo que implementa y, por último, contar con el soporte del sistema operativo para la coordinación con otros programas dentro y fuera de la misma aplicación y, también, para la interacción con el usuario.

ANOTACIONES

La representación de información en estructuras dinámicas de datos permite ajustar las necesidades de memoria del programa al mínimo requerido para la resolución del problema y, por otra parte, representar internamente la información de manera que su procesamiento sea más eficiente. Una estructura dinámica de datos no es más que una colección de datos cuya relación no está establecida *a priori* y que puede modificarse durante la ejecución del programa. Esto, por ejemplo, no es factible mediante un vector, pues los datos que contienen están relacionados por su posición dentro de él y, además, su tamaño debe de estar predefinido en el programa.

En el primer apartado se tratará de las variables dinámicas y de su empleo como contenedores de datos que responden a las exigencias de adaptabilidad a la información a representar y acomodamiento respecto del algoritmo que debe tratar con ellas.

El código fuente de una aplicación, sea ésta un conjunto de programas o uno único, debe organizarse de manera que se mantengan las características de un buen código (inteligible, de fácil mantenimiento y coste de ejecución óptimo). Para ello, no sólo hay que emplear una programación estructurada y modular, sino que hay que distribuirlo en diversos ficheros de manera que sea más manejable y, por otra parte, se mantenga su legibilidad.

En el apartado dedicado al diseño descendente de programas se tratará, precisamente, de los aspectos que afectan a la programación más allá de la programación modular. Se tratará de aspectos relacionados con la división del programa en términos algorítmicos y de la agrupación de conjuntos de funciones fuertemente relacionadas. Dado que el código fuente se reparte en diversos ficheros, se tratará también de aquellos aspectos relacionados con su compilación y, en especial, de la herramienta *make*.

Nota

Sobre este tema se vio un ejemplo al hablar de las tuberías en la unidad anterior.

Dada la complejidad del código fuente de cualquier aplicación, resulta conveniente emplear todas aquellas funciones estándar que aporta el sistema operativo. Con ello se consigue, además de reducir su complejidad al dejar determinadas tareas como simples instrucciones de llamada, que resulte más independiente de máquina. Así pues, en el último capítulo se trata de la relación entre los programas

ANOTACIONES

y los sistemas operativos, de manera que sea posible comunicar los unos con los otros y, además, los programas entre sí.

Para finalizar, se tratará, aunque brevemente, de cómo distribuir la ejecución de un programa en diversos flujos (o hilos) de ejecución. Esto es, de cómo realizar una programación concurrente de manera que varias tareas puedan resolverse en un mismo intervalo de tiempo.

Esta unidad pretende mostrar aquellos aspectos de la programación más involucrados con el nivel más alto de abstracción de los algoritmos. Así pues, al finalizar su estudio, el lector debería alcanzar los objetivos siguientes:

1) Emplear adecuadamente variables dinámicas en un programa.

2) Conocer las estructuras dinámicas de datos y, en especial, las listas y sus aplicaciones.

3) Entender el principio del diseño descendente de programas.

4) Ser capaz de desarrollar una biblioteca de funciones.

5) Tener conocimientos básicos para la relación del programa con el sistema operativo.

6) Asimilar el concepto de hilo de ejecución y los rudimentos de la programación concurrente.

3.2. Las variables dinámicas

La información procesada por un programa está formada por un conjunto de datos que, muy frecuentemente, no tiene ni un tamaño fijo, no se conoce su tamaño máximo, los datos no se relacionan entre ellos de la misma forma, etc.

Ejemplo

Un programa que realice análisis sintácticos (que, por otra parte, podría formar parte de una aplicación de tratamiento de textos) deberá procesar frases de tama-

ANOTACIONES

ños distintos en número y categoría de palabras. Además, las palabras pueden estar relacionadas de muy diferentes maneras. Por ejemplo, los adverbios lo están con los verbos y ambos se diferencian de las que forman el sintagma nominal, entre otras.

Las relaciones existentes entre los elementos que forman una información determinada se pueden representar mediante datos adicionales que las reflejen y nos permitan, una vez calculados, realizar un algoritmo más eficiente. Así pues, en el ejemplo anterior resultaría mucho mejor efectuar los análisis sintácticos necesarios a partir del árbol sintáctico que no de la simple sucesión de palabras de la frase a tratar.

El tamaño de la información, es decir, el número de datos que la forman, afecta significativamente al rendimiento de un programa. Más aún, el uso de variables estáticas para su almacenamiento implica o bien conocer *a priori* su tamaño máximo, o bien limitar la capacidad de tratamiento a sólo una porción de la información. Además, aunque sea posible determinar el tamaño máximo, se puede producir un despilfarro de memoria innecesario cuando la información a tratar ocupe mucho menos.

Las **variables estáticas** son aquellas que se declaran en el programa de manera que disponen de un espacio reservado de memoria durante toda la ejecución del programa. En C, sólo son realmente estáticas las variables globales.

Las **variables locales**, en cambio, son automáticas porque se les reserva espacio sólo durante la ejecución de una parte del programa y luego son destruidas de forma automática. Aun así, son variables de carácter estático respecto del ámbito en el que se emplean, pues tiene un espacio de memoria reservado y limitado.

Las **variables dinámicas**, sin embargo, pueden crearse y destruirse durante la ejecución de un programa y tienen un carácter global; es decir, son “visibles” desde cualquier punto del programa. Dado que es posible crear un número indeterminado de estas variables, permiten ajustarse al tamaño requerido para representar la información

ANOTACIONES

de un problema particular sin desperdiciar espacio de memoria alguno.

Para poder crear las variables dinámicas durante la ejecución del programa, es necesario contar con operaciones que lo permitan. Por otra parte, las variables dinámicas carecen de nombre y, por tanto, su única identificación se realiza mediante la dirección de la primera posición de memoria en la que residen.

Por ello, es necesario contar con datos que puedan contener referencias a variables dinámicas de manera que permitan utilizarlas. Como sus referencias son direcciones de memoria, el tipo de estos datos será, precisamente, el de dirección de memoria o *apuntador*, pues su valor es el indicador de dónde se encuentra la variable referenciada.

En los siguientes apartados se tratará, en definitiva, de todo aquello que atañe a las variables dinámicas y a las estructuras de datos que con ellas se pueden construir.

3.3. Los apuntadores

Los apuntadores son variables que contienen direcciones de memoria de otras variables; es decir, referencias a otras variables. Evidentemente, el tipo de datos es el de posiciones de memoria y, como tal, es un tipo compatible con enteros. Aun así, tiene sus particularidades, que veremos más adelante.

La declaración de un apuntador se hace declarando el tipo de datos de las variables de las cuales va a contener direcciones. Así pues, se emplea el operador de indirección (un asterisco) o, lo que es lo mismo, el que se puede leer como "contenido de la dirección". En los ejemplos siguientes se declaran distintos tipos de apuntadores:

```
int      *ref_entero;
char     *cadena;
otro_t   *apuntador;
nodo_t   *ap_nodo;
```

ANOTACIONES

- En `ref_entero` el contenido de la dirección de memoria almacenada es un dato de tipo entero.
- En `cadena` el contenido de la dirección de memoria almacenada es un carácter.
- En `apuntador` el contenido de la dirección de memoria almacenada es de tipo `otro_t`.
- En `ap_nodo`, el contenido de la dirección de memoria almacenada es de tipo `nodo_t`.

La referencia a las variables sin el operador de indirección es, simplemente, la dirección de memoria que contienen.

El tipo de apuntador viene determinado por el tipo de dato del que tiene la dirección. Por este motivo, por ejemplo, se dirá que `ref_entero` es un apuntador de tipo entero, o bien, que se trata de un apuntador "a" un tipo de datos entero. También es posible declarar apuntadores de apuntadores, etc.

En el ejemplo siguiente se puede observar que, para hacer referencia al valor apuntado por la dirección contenida en un apuntador, hay que emplear el operador de "contenido de la dirección de":

```
int  a, b;   /* dos variables de tipo entero.              */
int  *ptr;   /* un apuntador a entero.                     */
int  **pptr; /* un apuntador a un apuntador de entero.     */
/* ... */
a = b = 5;
ptr = &a;
*ptr = 20;  /* a == 20                                     */
ptr = &b;
pptr = &ptr;
**pptr = 40;/* b == 40                                     */
/* ... */
```

ANOTACIONES

En la figura siguiente se puede apreciar cómo el programa anterior va modificando las variables a medida que se va ejecutando. Cada columna vertical representa las modificaciones llevadas a cabo en el entorno por una instrucción. Inicialmente (la columna de más a la izquierda), se desconoce el contenido de las variables:

Figura 5.

:							
a	¿?	5		20			
b	¿?	5					40
ptr	¿?		&a		&b		
pptr	¿?					&ptr	
:							

En el caso particular de las tuplas, cabe recordar que el acceso también se hace mediante el operador de indirección aplicado a los apuntadores que contienen sus direcciones iniciales. El acceso a sus campos no cambia:

```
struct alumno_s {
   cadena_t nombre;
   unsigned short dni, nota;
} alumno;
struct alumno_s *ref_alumno;
/* ... */
alumno.nota = *ref_alumno.nota;
/* ... */
```

Para que quede claro el acceso a un campo de una tupla cuya dirección está en una variable, es preferible utilizar lo siguiente:

```
/* ... */
alumno.nota = (*ref_alumno).nota;
/* ... */
```

Si se quiere enfatizar la idea del apuntador, es posible emplear el operador de indirección para tuplos que se asemeja a una flecha que apunta a la tupla correspondiente:

ANOTACIONES

```
/* ... */
alumno.nota = ref_alumno→nota;
/* ... */
```

En los ejemplos anteriores, todas las variables eran de carácter estático o automático, pero su uso primordial es como referencia de las variables dinámicas.

3.3.1. Relación entre apuntadores y vectores

Los vectores, en C, son entelequias del programador, pues se emplea un operador (los corchetes) para calcular la dirección inicial de un elemento dentro de un vector. Para ello, hay que tener presente que los nombres con los que se declaran son, de hecho, apuntadores a las primeras posiciones de los primeros elementos de cada vector. Así pues, las declaraciones siguientes son prácticamente equivalentes:

```
/* ... */
int vector_real[DIMENSION];
int *vector_virtual;
/* ... */
```

En la primera, el vector tiene una dimensión determinada, mientras que en la segunda, el `vector_virtual` es un apuntador a un entero; es decir, una variable que contiene la dirección de datos de tipo entero. Aun así, es posible emplear el identificador de la primera como un apuntador a entero. De hecho, contiene la dirección del primer entero del vector:

```
vector_real == &(vector_real[0])
```

Es muy importante no modificar el contenido del identificador, ¡pues se podría perder la referencia a todo el vector!

Por otra parte, los apuntadores se manejan con una aritmética especial: se permite la suma y la resta de apuntadores de un mismo tipo

ANOTACIONES

y enteros. En el último caso, las sumas y restas con enteros son, en realidad, sumas y restas con los múltiplos de los enteros, que se multiplican por el tamaño en bytes de aquello a lo que apuntan.

Sumar o restar contenidos de apuntadores resulta algo poco habitual. Lo más frecuente es incrementarlos o decrementarlos para que apunten a algún elemento posterior o anterior, respectivamente. En el ejemplo siguiente se puede intuir el porqué de esta aritmética especial. Con ella, se libera al programador de tener que pensar cuántos bytes ocupa cada tipo de dato:

```
/* ... */
int vector[DIMENSION], *ref_entero;
/* ... */
ref_entero = vector;
ref_entero = ref_entero + 3;
*ref_entero = 15; /* Es equivalente a vector[3] = 15 */
/* ... */
```

En todo caso, existe el operador `sizeof` (“tamaño de”) que devuelve como resultado el tamaño en bytes del tipo de datos de su argumento. Así pues, el caso siguiente es, en realidad, un incremento de `ref_entero` de tal manera que se añade `3*sizeof(int)` a su contenido inicial.:

```
/* ... */
ref_entero = ref_entero + 3;
/* ... */
```

Por tanto, en el caso de los vectores, se cumple que:

```
vector[i] == *(vector+i)
```

Es decir, que el elemento que se encuentra en la posición i-ésima es el que se encuentra en la posición de memoria que resulta de sumar `i*sizeof(*vector)` a su dirección inicial, indicada en vector.

Nota

El operador `sizeof` se aplica al elemento apuntado por `vector`, puesto que, de aplicarse a `vector`, se obtendría el tamaño en bytes que ocupa un apuntador.

ANOTACIONES

En el siguiente ejemplo, se podrá observar con más detalle la relación entre apuntadores y vectores. El programa listado a continuación toma como entrada un nombre completo y lo separa en nombre y apellidos:

```
#include <stdio.h>
#include <ctype.h>
typedef char frase_t[256];
char *copia_palabra( char *frase, char *palabra )
/* Copia la primera palabra de la frase en palabra.        */
/* frase : apuntador a un vector de caracteres.            */
/* palabra : apuntador a un vector de caracteres.          */
/* Devuelve la dirección del último carácter leído         */
/* en la frase.                                            */
{
   while( *frase!='\0' && isspace( *frase ) ) {
     frase = frase + 1;
   } /* while */
   while( *frase!='\0' && !isspace( *frase ) ) {
     *palabra = *frase;
     palabra = palabra + 1;
     frase = frase + 1;
   } /* while */
   *palabra = '\0';
   return frase;
} /* palabra */

main( void ) {
   frase_t nombre_completo, nombre, apellido1, apellido2;
   char *posicion;

   printf( "Nombre y apellidos? " );
   gets( nombre_completo );
   posicion = copia_palabra( nombre_completo, nombre );
   posicion = copia_palabra( posicion, apellido1 );
   posicion = copia_palabra( posicion, apellido2 );
   printf(
     "Gracias por su amabilidad, Sr/a. %s.\n",
      apellido1
   ); /* printf */
} /* main */
```

ANOTACIONES

Nota

Más adelante, cuando se trate de las cadenas de caracteres, se insistirá de nuevo en la relación entre apuntadores y vectores.

3.3.2. Referencias de funciones

Las referencias de funciones son, en realidad, la dirección a la primera instrucción ejecutable de las mismas. Por lo tanto, se pueden almacenar en apuntadores a funciones.

La declaración de un apuntador a una función se realiza de manera similar a la declaración de los apuntadores a variables: basta con incluir en su nombre un asterisco. Así pues, un apuntador a una función que devuelve el número real, producto de realizar alguna operación con el argumento, se declararía de la manera siguiente:

```
float (*ref_funcion)( double x );
```

Nota

El paréntesis que encierra al nombre del apuntador y al asterisco que le precede es necesario para que no se confunda con la declaración de una función cuyo valor devuelto sea un apuntador a un número real.

Sirva como ejemplo un programa para la integración numérica de un determinado conjunto de funciones. Este programa es parecido al que ya se vio en la unidad anterior con la modificación de que la función de integración numérica tiene como nuevo argumento la referencia de la función de la que hay que calcular la integral:

```
/* Programa: integrales.c */
#include <stdio.h>
#include <math.h>
double f0( double x ) { return x/2.0;            }
double f1( double x ) { return 1+2*log(x);       }
double f2( double x ) { return 1.0/(1.0 + x*x);}
```

ANOTACIONES

```
double integral_f(  double a, double b, int n,
                    double (*fref)( double x )
) {
   double result;
   double x, dx;
   int i;
   result = 0.0;
   if( (a < b) && (n > 0) ) {
     x = a;
     dx = (b-a)/n;
     for( i = 0; i < n; i = i + 1 ) {
       result = result + (*fref)(x);
       x = x + dx;
     } /* for */
   } /* if */
   return result;
} /* integral_f */

void main( void )
{
   double   a, b;
   int      n, fnum;
   double   (*fref)( double x );
 printf( "Integración numérica de f(x).\n" );
 printf( "Punto inicial del intervalo, a = ? " );
 scanf( "%lf", &a );
 printf( "Punto final del intervalo, b = ? " );
   scanf( "%lf", &b );
   printf( "Número de divisiones, n = ? " );
   scanf( "%d", &n );
   printf( "Número de función, fnum = ?");
   scanf( "%d", &fnum );
   switch( fnum ) {
     case 1 : fref = f1; break;
     case 2 : fref = f2; break;
     default: fref = f0;
   } /* switch */
   printf(
     "Resultado, integral(f)[%g,%g] = %g\n",
     a, b, integral_f( a, b, n, fref )
   ); /* printf */
} /* main */
```

ANOTACIONES

Como se puede observar, el programa principal podría perfectamente sustituir las asignaciones de referencias de funciones por llamadas a las mismas. Con esto, el programa sería mucho más claro. Aun así, como se verá más adelante, esto permite que la función que realiza la integración numérica pueda residir en alguna biblioteca y ser empleada por cualquier programa.

3.4. Creación y destrucción de variables dinámicas

Tal como se dijo al principio de esta unidad, las variables dinámicas son aquellas que se crean y destruyen durante la ejecución del programa que las utiliza. Por el contrario, las demás son variables estáticas o automáticas, que no necesitan de acciones especiales por parte del programa para ser empleadas.

Antes de poder emplear una variable dinámica, hay que reservarle espacio mediante la función estándar (declarada en `stdlib.h`) que localiza y reserva en la memoria principal un espacio de tamaño `número_bytes` para que pueda contener los distintos datos de una variable:

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

Como la función desconoce el tipo de datos de la futura variable dinámica, devuelve un apuntador a tipo vacío, que hay que coercer al tipo correcto de datos:

```
/* ... */
char *apuntador;
/* ... */
apuntador = (char *)malloc( 31 );
/* ... */
```

Si no puede reservar espacio, devuelve `NULL`.

En general, resulta difícil conocer exactamente cuántos bytes ocupa cada tipo de datos y, por otra parte, su tamaño puede depender del compilador y de la máquina que se empleen. Por este motivo, es

Nota

El tipo de datos `size_t` es, simplemente, un tipo de entero sin signo al que se lo ha denominado así porque representa tamaños.

ANOTACIONES

conveniente emplear siempre el operador `sizeof`. Así, el ejemplo anterior tendría que haber sido escrito de la forma siguiente:

```
/* ... */
apuntador = (char *)malloc( 31 * sizeof(char) );
/* ... */
```

`sizeof` devuelve el número de bytes necesario para contener el tipo de datos de la variable o del tipo de datos que tiene como argumento, excepto en el caso de las matrices, en que devuelve el mismo valor que para un apuntador.

A veces, es necesario ajustar el tamaño reservado para una variable dinámica (sobre todo, en el caso de las de tipo vector), bien porque falta espacio para nuevos datos, bien porque se desaprovecha gran parte del área de memoria. Para tal fin, es posible emplear la función de "relocalización" de una variable:

```
void * realloc( void *apuntador, size_t nuevo_tamaño );
```

El comportamiento de la función anterior es similar a la de `malloc`: devuelve `NULL` si no ha podido encontrar un nuevo emplazamiento para la variable con el tamaño indicado.

Cuando una variable dinámica ya no es necesaria, se tiene que destruir; es decir, liberar el espacio que ocupa para que otras variables dinámicas lo puedan emplear. Para ello hay que emplear la función `free`:

```
/* ... */
free( apuntador );
/* ... */
```

Como sea que esta función sólo libera el espacio ocupado pero no modifica en absoluto el contenido del apuntador, resulta que éste aún tiene la referencia a la variable dinámica (su dirección) y, por

ANOTACIONES

tanto, existe la posibilidad de acceder a una variable inexistente. Para evitarlo, es muy conveniente asignar el apuntador a NULL:

```
/* ... */
free( apuntador );
apuntador = NULL;
/* ... */
```

De esta manera, cualquier referencia errónea a la variable dinámica destruida causará error; que podrá ser fácilmente corregido.

3.5. Tipos de datos dinámicos

Aquellos datos cuya estructura puede variar a lo largo de la ejecución de un programa se denominan tipos de datos dinámicos.

La variación de la estructura puede ser únicamente en el número de elementos, como en el caso de una cadena de caracteres, o también en la relación entre ellos, como podría ser el caso de un árbol sintáctico.

Los tipos de datos dinámicos pueden ser almacenados en estructuras de datos estáticas, pero al tratarse de un conjunto de datos, tienen que ser vectores, o bien de forma menos habitual, matrices multidimensionales.

Nota

Las estructuras de datos estáticas son, por definición, lo opuesto a las estructuras de datos dinámicas. Es decir, aquéllas en que tanto el número de los datos como su interrelación no varían en toda la ejecución del programa correspondiente. Por ejemplo, un vector siempre tendrá una longitud determinada y todos los elementos a excepción del primero y del último tienen un elemento precedente y otro siguiente.

En caso de almacenar las estructuras de datos dinámicas en estructuras estáticas, es recomendable comprobar si se conoce el nú-

mero máximo y la cantidad media de datos que puedan tener. Si ambos valores son parecidos, se puede emplear una variable de vector estática o automática. Si son muy diferentes, o bien se desconocen, es conveniente ajustar el tamaño del vector al número de elementos que haya en un momento determinado en la estructura de datos y, por lo tanto, almacenar el vector en una variable de carácter dinámico.

Las estructuras de datos dinámicas se almacenan, por lo común, empleando variables dinámicas. Así pues, puede verse una estructura de datos dinámica como una colección de variables dinámicas cuya relación queda establecida mediante apuntadores. De esta manera, es posible modificar fácilmente tanto el número de datos de la estructura (creando o destruyendo las variables que los contienen) como la propia estructura, cambiando las direcciones contenidas en los apuntadores de sus elementos. En este caso, es habitual que los elementos sean tuplos y que se denominen *nodos*.

En los apartados siguientes se verán los dos casos, es decir, estructuras de datos dinámicas almacenadas en estructuras de datos estáticas y como colecciones de variables dinámicas. En el primer caso, se tratará de las cadenas de caracteres, pues son, de largo, las estructuras de datos dinámicas más empleadas. En el segundo, de las listas y de sus aplicaciones.

3.5.1. Cadenas de caracteres

Las cadenas de caracteres son un caso particular de vectores en que los elementos son caracteres. Además, se emplea una marca de final (el carácter NUL o `'\0'`) que delimita la longitud real de la cadena representada en el vector.

La declaración siguiente sería una fuente de problemas derivada de la no inclusión de la marca de final, puesto que es una norma de C que hay que respetar para poder emplear todas las funciones estándar para el proceso de cadenas:

```
char cadena[20] = { 'H', 'o', 'l', 'a' } ;
```

ANOTACIONES

Por lo tanto, habría que inicializarla de la siguiente manera:

```
char cadena[20] = { 'H', 'o', 'l', 'a', '\0' } ;
```

Las declaraciones de cadenas de caracteres inicializadas mediante texto implican que la marca de final se añada siempre. Por ello, la declaración anterior es equivalente a:

```
char cadena[20] = "Hola" ;
```

Aunque el formato de representación de cadenas de caracteres sea estándar en C, no se dispone de instrucciones ni de operadores que trabajen con cadenas: no es posible hacer asignaciones ni comparaciones de cadenas, es necesario recurrir a las funciones estándar (declaradas en `string.h`) para el manejo de cadenas:

```
int    strlen   ( char *cadena );
char * strcpy   ( char *destino, char *fuente );
char * strncpy  ( char *destino, char *fuente, int núm_car );
char * strcat   ( char *destino, char *fuente );
char * strncat  ( char *destino, char *fuente, int núm_car );
char * strdup   ( char *origen );
char * strcmp   ( char *cadena1, char *cadena2 );
char * strncmp  ( char *kdna1, char *kdna2, int núm_car );
char * strchr   ( char *cadena, char carácter );
char * strrchr  ( char *cadena, char carácter );
```

La longitud real de una cadena de caracteres `kdna` en un momento determinado se puede obtener mediante la función siguiente:

```
strlen ( kdna )
```

El contenido de la cadena de caracteres apuntada por `kdna` a `kdna9`, se puede copiar con `strcpy( kdna9, kdna )`. Si la cadena fuente puede ser más larga que la capacidad del vector correspondiente a la de destino, con `strncpy( kdna9, kdna, LONG_KDNA9 – 1 )`. En este último caso, hay que prever que la cadena resultante no lleve un `'\0'` al final. Para solucionarlo, hay que reservar el último carácter de la copia con conteo para poner un

ANOTACIONES

'\0' de guarda. Si la cadena no tuviera espacio reservado, se tendría que hacer lo siguiente:

```
/* ... */
char *kdna9, kdna[LONG_MAX];
/* ... */
kdna9 = (char *) malloc( strlen( kdna ) + 1 );
if( kdna9 != NULL ) strcpy( kdna9, kdna );
/* ... */
```

Nota

De esta manera, kdna9 es una cadena con el espacio ajustado al número de caracteres de la cadena almacenada en kdna, que se deberá liberar meidante un free(kdna9) cuando ya no sea necesaria. El procedimiento anterior se puede sustituir por:

```
/* ... */
kdna9 = strdup( kdna );
/* ... */
```

La comparación entre cadenas se hace carácter a carácter, empezando por el primero de las dos cadenas a comparar y continuando por los siguientes mientras la diferencia entre los códigos ASCII sea 0. La función strcmp() retorna el valor de la última diferencia. Es decir, un valor negativo si la segunda cadena es alfabéticamente mayor que la primera, positivo en caso opuesto, y 0 si son iguales. Para entenderlo mejor, se adjunta un posible código para la función de comparación de cadenas:

```
int strcmp( char *cadena1, char *cadena2 )
{
   while( (*cadena1 != '\0') &&
          (*cadena2 != '\0') &&
          (*cadena1 == *cadena2)
   ) {
     cadena1 = cadena1 + 1;
     cadena2 = cadena2 + 1;
   } /* while */
   return *cadena1 - *cadena2;
} /* strcmp */
```

ANOTACIONES

La función `strncmp()` hace lo mismo que `strcmp()` con los primeros `núm_car` caracteres.

Finalmente, aunque hay más, se comentan las funciones de búsqueda de caracteres dentro de cadenas. Estas funciones retornan el apuntador al carácter buscado o `NULL` si no se encuentra en la cadena:

- `strchr()` realiza la búsqueda desde el primer carácter.
- `strrchr()` inspecciona la cadena empezando por la derecha; es decir, por el final.

Ejemplo

```
char *strchr( char *cadena, char caracter )
{
    while( (*cadena != '\0') && (*cadena2 != caracter
) ) {
      cadena = cadena + 1;
    } /* while */
    return cadena;
} /* strchr */
```

Todas las funciones anteriores están declaradas en `string.h` y, por tanto, para utilizarlas, hay que incluir este fichero en el código fuente del programa correspondiente.

En `stdio.h` también hay funciones estándar para operar con cadenas, como `gets()` y `puts()`, que sirven para la entrada y la salida de datos que sean cadenas de caracteres y que ya fueron descritas en su momento. También contiene las declaraciones de `sscanf()` y `sprintf()` para la lectura y la escritura de cadenas con formato. Estas dos últimas funciones se comportan exactamente igual que `scanf()` y `printf()` a excepción de que la lectura o escritura se realizan en una cadena de caracteres en lugar de hacerlo en el dispositivo estándar de entrada o salida:

```
sprintf(
   char *destino,  /* Cadena en la que "imprime".          */
   char *formato
   [, lista_de_variables]
); /* sprintf */
```

ANOTACIONES

```
int sscanf(          /* Devuelve el número de variables      */
                     /* cuyo contenido ha sido actualizado.  */
   char *origen,     /* Cadena de la que se "lee".           */
   char *formato
   [,lista_de_&variables]
); /* sscanf */
```

Cuando se utiliza `sprintf()` se debe comprobar que la cadena `destino` tenga espacio suficiente para contener aquello que resulte de la impresión con el formato dado.

Cuando se utiliza `sscanf()`,se debe comprobar siempre que se han leído todos los campos: la inspección de la cadena `origen` se detiene al encontrar la marca de final de cadena independientemente de los especificadores de campo indicados en el formato.

En el siguiente ejemplo, se puede observar el código de una función de conversión de una cadena que represente un valor hexadecimal (por ejemplo: "3D") a un entero positivo (siguiendo el ejemplo anterior: $3D_{(16} = 61$) mediante el uso de las funciones anteriormente mencionadas. La primera se emplea para prefijar la cadena con "0x", puesto que éste es el formato de los números hexadecimales en C, y la segunda, para efectuar la lectura de la cadena obtenida aprovechando que lee números en cualquier formato estándar de C.

```
unsigned hexaVal( char *hexadecimal )
{
   unsigned numero;
   char *hexaC;

   hexaC = (char *) malloc(
     ( strlen( hexadecimal ) + 3 ) * sizeof( char )
   ); /* malloc */
```

ANOTACIONES

```
   if( hexaC != NULL ) {
    sprintf( hexaC, "0x%s", hexadecimal );
    sscanf( hexaC, "%x", &numero );
    free( hexaC );
   } else {
    numero = 0; /* ¡La conversión no se ha realizado!*/
   } /* if */
   return numero;
} /* hexaVal */
```

Nota

Recordad la importancia de liberar el espacio de las cadenas de caracteres creadas dinámicamente que no se vayan a utilizar. En el caso anterior, de no hacer un `free( hexaC )`, la variable seguiría ocupando espacio, a pesar de que ya no se podría acceder a ella, puesto que la dirección está contenida en el apuntador `hexaC`, que es de tipo automático y, por tanto, se destruye al finalizar la ejecución de la función. Este tipo de errores puede llegar a causar un gran desperdicio de memoria.

3.5.2. Listas y colas

Las listas son uno de los tipos dinámicos de datos más empleados y consisten en secuencias homogéneas de elementos sin tamaño predeterminado. Al igual que las cadenas de caracteres, pueden almacenarse en vectores siempre que se sepa su longitud máxima y la longitud media durante la ejecución. Si esto no es así, se emplearán variables dinámicas "enlazadas" entre sí; es decir, variables que contendrán apuntadores a otras dentro de la misma estructura dinámica de datos.

La ventaja que aporta representar una lista en un vector es que elimina la necesidad de un campo apuntador al siguiente. De todas maneras, hay que insistir en que sólo es posible aprovecharla si no se produce un derroche de memoria excesivo y cuando el programa no deba hacer frecuentes inserciones y eliminaciones de elementos en cualquier posición de la lista.

En este apartado se verá una posible forma de programar las operaciones básicas con una estructura de datos dinámica de tipo lista

ANOTACIONES

mediante variables dinámicas. En este caso, cada elemento de la lista será un nodo del tipo siguiente:

```
typedef struct nodo_s {
   int            dato;
   struct nodo_s *siguiente;
} nodo_t, *lista_t;
```

El tipo `nodo_t` se corresponde con un nodo de la lista y que `lista_t` es un apuntador a un nodo cuya tupla tiene un campo `siguiente` que, a la vez, es un apuntador a otro nodo, y así repetidamente hasta tener enlazados toda una lista de nodos. A este tipo de listas se las denomina ***listas simplemente enlazadas***, pues sólo existe un enlace entre un nodo y el siguiente.

Las listas simplemente enlazadas son adecuadas para algoritmos que realicen frecuentes recorridos secuenciales. En cambio, para aquéllos en que se hacen recorridos parciales en ambos sentidos (hacia delante y hacia atrás en la secuencia de nodos) es recomendable emplear **listas doblemente enlazadas**; es decir, listas cuyos nodos contengan apuntadores a los elementos siguientes y anteriores.

En ambos casos, si el algoritmo realiza inserciones de nuevos elementos y destrucciones de elementos innecesarios de forma muy frecuente, puede ser conveniente tener el primer y el último elemento enlazados. Estas estructuras se denominan ***listas circulares*** y, habitualmente, los primeros elementos están marcados con algún dato o campo especial.

Operaciones elementales con listas

Una lista de elementos debe permitir llevar a cabo las operaciones siguientes:

- Acceder a un nodo determinado.
- Eliminar un nodo existente.
- Insertar un nuevo nodo.

ANOTACIONES

En los apartados siguientes se comentarán estas tres operaciones y se darán los programas de las funciones para llevarlas a cabo sobre una lista simplemente enlazada.

Para acceder a un nodo determinado es imprescindible obtener su dirección. Si se supone que se trata de obtener el enésimo elemento en una lista y devolver su dirección, la función correspondiente necesita como argumentos tanto la posición del elemento buscado, como la dirección del primer elemento de la lista, que puede ser `NULL` si ésta está vacía.

Evidentemente, la función retornará la dirección del nodo enésimo o `NULL` si no lo ha encontrado:

```
nodo_t *enesimo_nodo(lista_t lista, unsigned int n)
{
   while( ( lista != NULL ) && ( n != 0 ) ) {
     lista = lista →siguiente;
     n = n - 1;
   } /* while */
   return lista;
} /* enesimo_nodo */
```

En este caso, se considera que la primera posición es la posición número 0, de forma parecida a los vectores en C. Es imprescindible comprobar que `( lista != NULL )` se cumple, puesto que, de lo contrario, no podría ejecutarse `lista = lista→siguiente;`: no se puede acceder al campo siguiente de un nodo que no existe.

Para eliminar un elemento en una lista simplemente enlazada, es necesario disponer de la dirección del elemento anterior, ya que el campo siguiente de este último debe actualizarse convenientemente. Para ello, es necesario extender la función anterior de manera que devuelva tanto la dirección del elemento buscado como la del elemento anterior. Como sea que tiene que devolver dos datos, hay que hacerlo a través de un paso por referencia: hay que pasar las direcciones de los apuntadores a nodo que contendrán las direcciones de los nodos:

```
void enesimo_pq_nodo(
   lista_t    lista,   /* Apuntador al primer nodo.        */
   unsigned int n,     /* Posición del nodo buscado.       */
```

```
    nodo_t     **pref,/* Ref. apuntador a nodo previo.*/
    nodo_t     **qref)/* Ref. apuntador a nodo actual.*/
{
    nodo_t *p, *q;
    p = NULL; /* El anterior al primero no existe.           */
    q = lista;
    while( ( q != NULL ) && ( n != 0 ) ) {
     p = q;
     q = q→siguiente;
     n = n - 1;
    } /* while */
    *pref = p;
    *qref = q;
} /* enesimo_pq_nodo */
```

El programa de la función que destruya un nodo debe partir de que se conoce tanto su dirección (almacenada en `q`) como la del posible elemento anterior (guardada en el apuntador `p`). Dicho de otra manera, se pretende eliminar el elemento siguiente al apuntado por `p`.

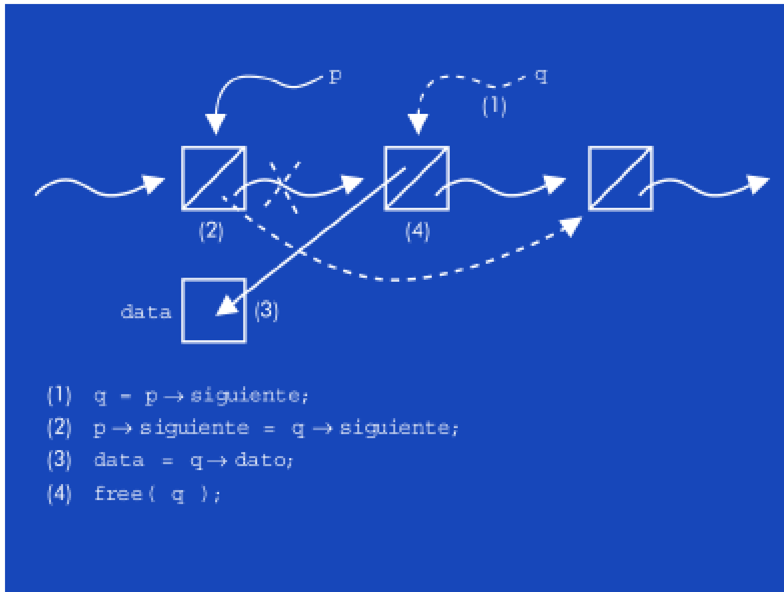
Cuando se realizan estos programas, es más que conveniente hacer un esquema de la estructura de datos dinámica sobre el que se indiquen los efectos de las distintas modificaciones en la misma.

Así pues, para programar esta función, primero estableceremos el caso general sobre un esquema de la estructura de la cual queremos eliminar un nodo y, luego, se programará atendiendo a las distintas excepciones que puede tener el caso general. Habitualmente, éstas vienen dadas por tratar el primer o el último elemento, pues son aquellos que no tienen precedente o siguiente y, como consecuencia, no siguen la norma de los demás en la lista.

ANOTACIONES

En la siguiente imagen se resume el procedimiento general de eliminación del nodo siguiente al nodo p:

Figura 6.



Nota

Evidentemente, no es posible eliminar un nodo cuando p == NULL. En este caso, tanto (1) como (2) no pueden ejecutarse por implicar referencias a variables inexistentes. Más aún, para p == NULL se cumple que se desea eliminar el primer elemento, pues es el único que no tiene ningún elemento que le preceda. Así pues, hemos de "proteger" la ejecución de (1) y (2) con una instrucción condicional que decida si se puede llevar a cabo el caso general o, por el contrario, se elimina el primero de los elementos. Algo parecido sucede con (3) y (4), que no pueden ejecutarse si q == NULL.

La función para destruir el nodo podría ser como la siguiente:

```
int destruye_nodo(
   lista_t *listaref,   /* Apuntador a referencia 1.er nodo.   */
   nodo_t *p,           /* Apuntador a nodo previo.            */
   nodo_t *q)           /* Apuntador a nodo a destruir.        */
{
   int data = 0          /* Valor por omisión de los datos.    */
```

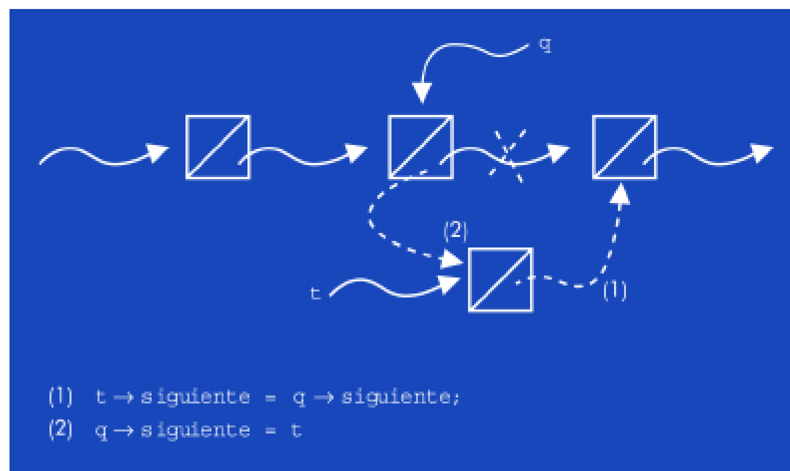
ANOTACIONES

```
  if( p != NULL ) {
    /* q = p->siguiente; (no es necesario) */
    p->siguiente = q->siguiente;
  } else {
    if( q != NULL ) *listaref = q->siguiente;
  } /* if */
  if( q!= NULL ) {
    data = q->dato;
    free( q );
  } /* if */
  return data;
} /* destruye_nodo */
```

Al eliminar el primer elemento, es necesario cambiar la dirección contenida en lista para que apunte al nuevo primer elemento, salvo que q también sea NULL).

Para insertar un nuevo nodo, cuya dirección esté en t, basta con realizar las operaciones indicadas en la siguiente figura:

Figura 7.



Nota

En este caso, el nodo t quedará insertado después del nodo q. Como se puede observar, es necesario que q != NULL para poder realizar las operaciones de inserción.

ANOTACIONES

El código de la función correspondiente sería como sigue:

```
void inserta_siguiente_nodo(
   lista_t   *listaref, /* Apuntador a referencia 1.er nodo.   */
   nodo_t    *q,        /* Apuntador a nodo en la posición.    */
   nodo_t    *t)        /* Apuntador a nodo a insertar.        */
{
   if( q != NULL ) {
     t->siguiente = q->siguiente;
     q->siguiente = t;
   } else { /* La lista está vacía.                            */
     *listaref = t;
   } /* if */
} /* inserta_siguiente_nodo */
```

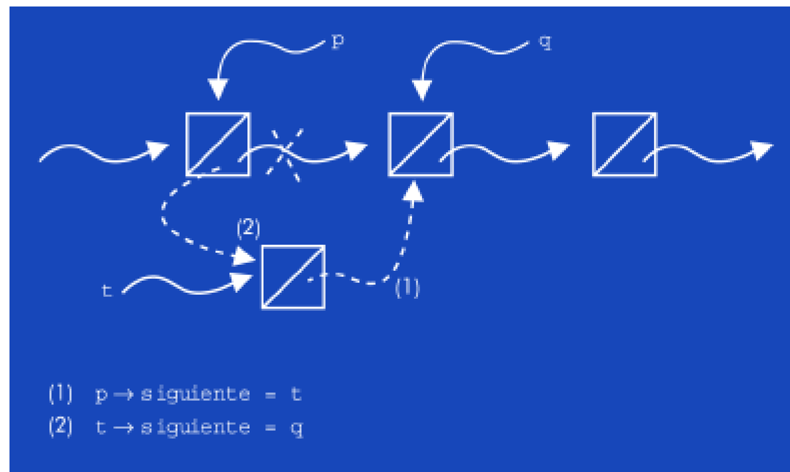
Para que la función anterior pueda ser útil, es necesario disponer de una función que permita crear nodos de la lista. En este caso:

```
nodo_t *crea_nodo( int data )
{
   nodo_t *nodoref;
   nodoref = (nodo_t *)malloc( sizeof( nodo_t ) );
   if( nodoref != NULL ) {
     nodoref->dato = data;
     nodoref->siguiente = NULL;
   } /* if */
   return nodoref;
} /* crea_nodo */
```

Si se trata de insertar en alguna posición determinada, lo más frecuente suele ser insertar el nuevo elemento como precedente del indicado; es decir, que ocupe la posición del nodo referenciado y desplace al resto de nodos una posición “a la derecha”. Las operaciones que hay que realizar para ello en el caso general se muestran en la siguiente figura:

ANOTACIONES

Figura 8.



Siguiendo las indicaciones de la figura anterior, el código de la función correspondiente sería el siguiente:

```
void inserta_nodo(
   lista_t *listaref,  /* Apuntador a referencia 1er nodo. */
   nodo_t *p,          /* Apuntador a nodo precedente.     */
   nodo_t *q,          /* Apuntador a nodo en la posición. */
   nodo_t *t)          /* Apuntador a nodo a insertar.     */
{
   if( p != NULL ) {
    p->siguiente = t;
   } else { /* Se inserta un nuevo primer elemento.        */
   *listaref = t;
   } /* if */
   t->siguiente = q;
} /* inserta_nodo */
```

Con todo, la inserción de un nodo en la posición enésima podría construirse de la siguiente manera:

```
bool inserta_enesimo_lista(
   lista_t    *listaref, /* Apuntador a referencia 1.er nodo.  */
   unsigned    int n,    /* Posición de la inserción.          */
   int         dato)     /* Dato a insertar.                   */
{                        /* Devuelve FALSE si no se puede.     */
   nodo_t *p, *q, *t;
   bool retval;
```

ANOTACIONES

```
   t = crea_nodo( dato );
   if( t != NULL ) {
     enesimo_pq_nodo( *listaref, n, &p, &q );
     inserta_nodo( listaref, p, q, t );
     retval = TRUE;
   } else {
     retval = FALSE;
   } /* if */
   return retval;
} /* inserta_enesimo_lista */
```

De igual manera, podría componerse el código para la función de destrucción del enésimo elemento de una lista.

Normalmente, además, las listas no serán de elementos tan simples como enteros y habrá que sustituir la definición del tipo de datos `nodo_t` por uno más adecuado.

Los criterios de búsqueda de nodos suelen ser más sofisticados que la búsqueda de una determinada posición. Por lo tanto, hay que tomar las funciones anteriores como un ejemplo de uso a partir del cual se pueden derivar casos reales de aplicación.

Colas

Las colas son, de hecho, listas en las que se inserta por un extremo y se elimina por el otro. Es decir, son listas en las que las operaciones de inserción y de eliminación se restringen a unos casos muy determinados. Esto permite hacer una gestión mucho más eficaz de las mismas. En este sentido, es conveniente disponer de un tuplo que facilite tanto el acceso directo al primer elemento como al último. De esta manera, las operaciones de eliminación e inserción se pueden resolver sin necesidad de búsquedas en la lista.

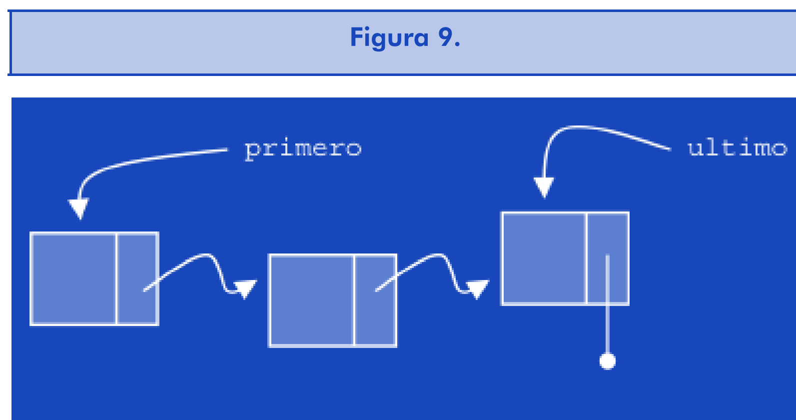
ANOTACIONES

Así pues, esta clase de colas debería tener una tupla de control como la siguiente:

```
typedef struct cola_s {
   nodo_t *primero;
   nodo_t *ultimo;
} cola_t;
```

Gráficamente:

Figura 9.



Para ejemplificar las dos operaciones, supondremos que los elementos de la cola serán simples enteros, es decir, que la cola será una lista de nodos del tipo de datos `nodo_t` que ya se ha visto.

Con ello, la inserción sería como sigue:

```
bool encola( cola_t *colaref, int dato )
/* Devuelve FALSE si no se puede añadir el dato.*/
{
   nodo_t *q, *t;
   bool   retval;

   t = crea_nodo( dato );
   if( t != NULL ) {
     t→siguiente = NULL;
     q = colaref→ultimo;
     if( q == NULL ) { /* Cola vacía: */
       colaref→primero = t;
```

```
    } else {
     q→siguiente = t;
    } /* if */
    colaref→ultimo = t;
    retval = TRUE;
  } else {
    retval = FALSE;
  } /* if */
  return retval;
} /* encola */
```

Y la eliminación:

```
bool desencola( cola_t *colaref, int *datoref )
/* Devuelve FALSE si no se puede eliminar el dato.    */
{
  nodo_t *q;
  bool retval;
  q = colaref→primero;
  if( q != NULL ) {
    colaref→primero = q→siguiente;
    *datoref = destruye_nodo( &q );
    if( colaref→primero == NULL ) { /* Cola vacía:    */
     colaref→ultimo = NULL;
    } /* if */
    retval = TRUE;
  } else {
    retval = FALSE;
  } /* if */
  return retval;
} /* desencola */
```

La función `destruye_nodo` de la eliminación anterior es como sigue:

```
int destruye_nodo( nodo_t **pref )
{
```

ANOTACIONES

```
   int dato = 0;
   if( *pref != NULL ) {
     dato = (*pref) →dato;
     free( *pref );
     *pref = NULL;
   } /* if */
   return dato;
} /* destruye_nodo */
```

Las colas se utilizan frecuentemente cuando hay recursos compartidos entre muchos usuarios.

Ejemplo

– Para gestionar una impresora, el recurso es la propia impresora y los usuarios, los ordenadores conectados a la misma.

– Para controlar una máquina del tipo “su turno”: el recurso es el vendedor y los usuarios son los clientes.

En general, cuando se hacen inserciones en cola se tiene en cuenta qué elemento se está insertando; es decir, no se realizan siempre por el final, sino que el elemento se coloca según sus privilegios sobre los demás. En este caso, se habla de **colas con prioridad**. En este tipo de colas, la eliminación siempre se realiza por el principio, pero la inserción implica situar al nuevo elemento en la última posición de los elementos con la misma prioridad.

Ciertamente, hay otros tipos de gestiones especializadas con listas y, más allá de las listas, otros tipos de estructuras dinámicas de datos como los árboles (por ejemplo, un árbol sintáctico) y los grafos (por ejemplo, una red de carreteras). Desafortunadamente, no se dispone de suficiente tiempo para tratarlos, pero hay que tenerlos en cuenta cuando los datos del problema puedan requerirlo.

3.6. Diseño descendente de programas

Recordemos que la programación modular estriba en dividir el código en subprogramas que realicen una función concreta. En esta uni-

ANOTACIONES

dad se tratará especialmente de la manera como pueden agruparse estos subprogramas de acuerdo a las tareas que les son encomendadas y, en definitiva, de cómo organizarlos para una mejor programación del algoritmo correspondiente.

Los algoritmos complejos suelen reflejarse en programas con muchas líneas de código. Por lo tanto se impone una programación muy cuidadosa para que el resultado sea un código legible y de fácil mantenimiento.

3.6.1. Descripción

El fruto de una programación modular es un código constituido por diversos subprogramas de pocas líneas relacionados entre ellos mediante llamadas. Así pues, cada uno de ellos puede ser de fácil comprensión y, consecuentemente, de fácil mantenimiento.

La técnica de diseño descendente es, de hecho, una técnica de diseño de algoritmos en la que se resuelve el algoritmo principal abstrayendo los detalles, que luego se solucionan mediante otros algoritmos de la misma manera. Es decir, se parte del nivel de abstracción más alto y, para todas aquellas acciones que no puedan trasladarse de forma directa a alguna instrucción del lenguaje de programación elegido, se diseñan los algoritmos correspondientes de forma independiente del principal siguiendo el mismo principio.

El diseño descendente consiste en escribir programas de algoritmos de unas pocas instrucciones e implementar las instrucciones no primitivas con funciones cuyos programas sigan las mismas normas anteriores.

En la práctica, esto comporta diseñar algoritmos de manera que permite que la programación de los mismos se haga de forma totalmente modular.

Nota

Una instrucción primitiva sería aquella que pueda programarse directamente en un lenguaje de programación.

ANOTACIONES

3.6.2. Ejemplo

El diseño descendente de programas consiste, pues, en empezar por el programa del algoritmo principal e ir refinando aquellas instrucciones "gruesas" convirtiéndolas en subprogramas con instrucciones más "finas". De ahí la idea de refinar. Evidentemente, el proceso termina cuando ya no hay más instrucciones "gruesas" para refinar.

En este apartado se verá un ejemplo simple de diseño descendente para resolver un problema bastante habitual en la programación: el de la ordenación de datos para facilitar, por ejemplo, su consulta.

Este problema es uno de los más estudiados en la ciencia informática y existen diversos métodos para solucionarlo. Uno de los más simples consiste en seleccionar el elemento que debería encabezar la clasificación (por ejemplo, el más pequeño o el mayor, si se trata de números), ponerlo en la lista ordenada y repetir el proceso con el resto de elementos por ordenar. El programa principal de este algoritmo puede ser el siguiente:

```
/* ... */
lista_t     pendientes, ordenados;
elemento_t elemento;
/* ... */
inicializa_lista( &ordenados );
while( ! esta_vacia_lista( pendientes ) ) {
   elemento = extrae_minimo_de_lista( &pendientes );
   pon_al_final_de_lista( &ordenados, elemento );
} /* while */
/* ... */
```

En el programa anterior hay pocas instrucciones primitivas de C y, por tanto, habrá que refinarlo. Como contrapartida, su funcionamiento resulta fácil de comprender. Es importante tener en cuenta que los operadores de "dirección de" (el signo `&`) en los parámetros de las llamadas de las funciones indican que éstas pueden modificar el contenido de los mismos.

La mayor dificultad que se encuentra en el proceso de refinado es, habitualmente, identificar las partes que deben describirse con ins-

ANOTACIONES

trucciones primitivas; es decir, determinar los distintos niveles de abstracción que deberá tener el algoritmo y, consecuentemente, el programa correspondiente. Generalmente, se trata de conseguir que el programa refleje al máximo el algoritmo del que proviene.

Es un error común pensar que aquellas operaciones que sólo exigen una o dos instrucciones primitivas no pueden ser nunca contempladas como una instrucción no primitiva.

Una norma de fácil adopción es que todas las operaciones que se realicen con un tipo de datos abstracto sean igualmente abstractas; es decir, se materialicen en instrucciones no primitivas (funciones).

3.7. Tipos de datos abstractos y funciones asociadas

La mejor manera de implementar la programación descendente es programar todas las operaciones que puedan hacerse con cada uno de los tipos de datos abstractos que se tengan que emplear. De hecho, se trata de crear una máquina virtual para ejecutar aquellas instrucciones que se ajustan al algoritmo, a la manera que un lenguaje dispone de todas las operaciones necesarias para la máquina que es capaz de procesarlo y, obviamente, luego se traslada a las operaciones del lenguaje de la máquina real que realizará el proceso.

En el ejemplo del algoritmo de ordenación anterior, aparecen dos tipos de datos abstractos (lista_t y elemento_t) para los que, como mínimo, son necesarias las operaciones siguientes:

```
void inicializa_lista( lista_t *ref_lista );
bool esta_vacia_lista( lista_t lista );
elemento_t extrae_minimo_de_lista( lista_t *ref_lista );
void pon_al_final_de_lista( lista_t *rlst, elemento_t e );
```

Como se puede observar, no hay operaciones que afecten a datos del tipo elemento_t. En todo caso, es seguro que el programa hará uso de ellas (lectura de datos, inserción de los mismos en la lista, comparación entre elementos, escritura de resultados, etc.) en al-

ANOTACIONES

guna otra sección. Por lo tanto, también se habrán de programar las operaciones correspondientes. En particular, si se observa el siguiente código se comprobará que aparecen funciones para tratar con datos del tipo `elemento_t`:

```
elemento_t extrae_minimo_de_lista( lista_t *ref_lista )
{
   ref_nodo_t  actual, minimo;
   bool        es_menor;
   elemento_t  peque;
   principio_de_lista( ref_lista );
   if( esta_vacia_lista( *ref_lista ) ) {
     inicializa_elemento( &peque );
   } else {
     minimo = ref_nodo_de_lista( *ref_lista );
     peque = elemento_en_ref_nodo( *ref_lista, minimo );
     avanza_posicion_en_lista( ref_lista );
     while( !es_final_de_lista( *ref_lista ) ) {
       actual = ref_nodo_de_lista( *ref_lista );
       es_menor = compara_elementos(
         elemento_en_ref_nodo( *ref_lista, actual ), peque
       ); /* compara_elementos */
       if( es_menor ) {
         minimo = actual;
         peque = elemento_en_ref_nodo( *ref_lista, minimo );
       } /* if */
       avanza_posicion_en_lista( ref_lista );
     } /* while */
     muestra_elemento( peque );
     elimina_de_lista( ref_lista, minimo );
   } /* if */
   return peque;
} /* extrae_minimo_de_lista */
```

Como se puede deducir del código anterior, al menos son necesarias dos operaciones para datos del tipo `elemento_t`:

```
inicializa_elemento();
compara_elementos();
```

ANOTACIONES

Además, son necesarias cuatro operaciones más para listas:

```
principio_de_lista();
es_final_de_lista();
avanza_posicion_en_lista();
elimina_de_lista();
```

Se ha añadido también el tipo de datos `ref_nodo_t` para tener las referencias de los nodos en las listas y dos operaciones: `ref_nodo_de_lista` para obtener la referencia de un determinado nodo en la lista y `elemento_en_ref_nodo` para obtener el elemento que se guarda en el nodo indicado.

Con esto se demuestra que el refinamiento progresivo sirve para determinar qué operaciones son necesarias para cada tipo de datos y, por otra parte, se refleja que las listas constituyen un nivel de abstracción distinto y mayor que el de los elementos.

Para completar la programación del algoritmo de ordenación, es necesario desarrollar todas las funciones asociadas a las listas, y luego a los elementos. De todas maneras, lo primero que hay que hacer es determinar los tipos de datos abstractos que se emplearán.

Las listas pueden materializarse con vectores o con variables dinámicas, según el tipo de algoritmos que se empleen. En el caso del ejemplo de la ordenación, dependerá en parte de los mismos criterios generales que se aplican para tomar tal decisión y, en parte, de las características del mismo algoritmo. Generalmente, se elegirá una implementación con vectores si el desaprovechamiento medio de los mismos es pequeño, y particularmente se tiene en cuenta que el algoritmo que nos ocupa sólo se puede aplicar para la clasificación de cantidades modestas de datos (por ejemplo, unos centenares como mucho).

Así pues, de escoger la primera opción, el tipo de datos `lista_t` sería:

```
#define LONG_MAXIMA 100
typedef struct lista_e {
```

ANOTACIONES

```
   elemento_t    nodo[ LONG_MAXIMA ];
   unsigned short posicion; /* Posición actual de acceso.  */
   unsigned short cantidad; /* Longitud de la lista.       */
} lista_t;
```

También haría falta definir el tipo de datos para la referencia de los nodos:

```
typedef unsigned short ref_nodo_t;
```

De esta manera, el resto de operaciones que son necesarias se corresponderían con las funciones siguientes:

```
void inicializa_lista( lista_t *ref_lista )
{
   (*ref_lista).cantidad = 0;
   (*ref_lista).posicion = 0;
} /* inicializa_lista */

bool esta_vacia_lista( lista_t lista )
{
   return ( lista.cantidad == 0 );
} /* esta_vacia_lista */

bool es_final_de_lista( lista_t lista )
{
   return ( lista_posicion == lista.cantidad );
} /* es_final_de_lista */

void principio_de_lista( lista_t *lista_ref )
{
   lista_ref->posicion = 0;
} /* principio_de_lista */

ref_nodo_t ref_nodo_de_lista( lista_t lista )
{
   return lista.posicion;
} /* ref_nodo_de_lista */

elemento_t elemento_en_ref_nodo(
   lista_t lista,
```

ANOTACIONES

```
   ref_nodo_t refnodo)
{
   return lista.nodo[ refnodo ];
} /* elemento_en_ref_nodo */

void avanza_posicion_en_lista( lista_t *lista_ref )
{
   if( !es_final_de_lista( *lista_ref ) ) {
     (*lista_ref).posicion= (*lista_ref).posicion+1;
   } /* if */
} /* avanza_posicion_en_lista */

elemento_t elimina_de_lista(
   lista_t *ref_lista,
   ref_nodo_t refnodo)
{
   elemento_t eliminado;
   ref_nodo_t pos, ultimo;

   if( esta_vacia_lista( *ref_lista ) ) {
   inicializa_elemento( &eliminado );
   } else {
     eliminado = (*ref_lista).nodo[ refnodo ];
     ultimo = (*ref_lista).cantidad - 1;
     for(pos= refnodo; pos < ultimo; pos= pos+ 1 ) {
       (*ref_lista).nodo[pos] = (*ref_lista).nodo[pos+1];
     } /* for */
     (*ref_lista).cantidad= (*ref_lista).cantidad- 1;
   } /* if */
   return eliminado;
} /* elimina_de_lista */

elemento_t extrae_minimo_de_lista( lista_t *ref_lista );

void pon_al_final_de_lista(
   lista_t *ref_lista,
   elemento_t elemento )
{
   if( (*ref_lista).cantidad < LONG_MAXIMA ) {
     (*ref_lista).nodo[(*ref_lista).cantidad] = elemento;
     (*ref_lista).cantidad = (*ref_lista).cantidad + 1;
```

ANOTACIONES

```
   } /* if */
} /* pon_al_final_de_lista */
```

Si se examinan las funciones anteriores, todas asociadas al tipo de datos `lista_t`, se verá que sólo requieren de una operación con el tipo de datos `elemento_t`: `compara_elementos`. Por lo tanto, para completar el programa de ordenación, ya sólo falta definir el tipo de datos y la operación de comparación.

Si bien las operaciones con las listas sobre vectores eran genéricas, todo aquello que afecta a los elementos dependerá de la información cuyos datos se quieran ordenar.

Por ejemplo, suponiendo que se deseen ordenar por número de DNI las notas de un examen, el tipo de datos de los elementos podría ser como sigue:

```
typedef struct elemento_s {
       unsigned int    DNI;
       float           nota;
} dato_t, *elemento_t;
```

La función de comparación sería como sigue:

```
bool compara_elementos(
   elemento_t menor,
   elemento_t mayor )
{
   return ( menor->DNI < mayor->DNI );
} /* compara_elementos */
```

La función de incialización sería como sigue:

```
void inicializa_elemento( elemento_t *ref_elem )
{
   *ref_elem = NULL;
} /* inicializa_elemento */
```

Nótese que los elementos son, en realidad, apuntadores de variables dinámicas. Esto, a pesar de requerir que el programa las construya

ANOTACIONES

y destruya, simplifica, y mucho, la codificación de las operaciones en niveles de abstracción superiores. No obstante, es importante recalcar que siempre habrá que preparar funciones para la creación, destrucción, copia y duplicado de cada uno de los tipos de datos para los que existen variables dinámicas.

Las funciones de copia y de duplicado son necesarias puesto que la simple asignación constituye una copia de la dirección de una variable a otro apuntador, es decir, se tendrán dos referencias a la misma variable en lugar de dos variables distintas con el mismo contenido.

Ejemplo

Comprobad la diferencia que existe entre estas dos funciones:

```
/* ... */
elemento_t original, otro, copia;
/* ... */
otro = original; /* Copia de apuntadores. */
/* la dirección guardada en 'otro'
 es la misma que la contenida en 'original'
*/

/* ... */
copia_elemento( original, copia );
otro = duplica_elemento( original );
/* las direcciones almacenadas en 'copia' y en 'otro'
 son distintas de la contenida en 'original'
*/
```

La copia de contenidos debe, pues, realizarse mediante una función específica y, claro está, si la variable que debe contener la copia no está creada, deberá procederse primero a crearla, es decir, se hará un duplicado.

En resumen, cuando hayamos de programar un algoritmo en diseño descendente, habremos de programar las funciones de creación, destrucción y copia para cada uno de los tipos de datos abstractos que contenga. Además, se programará como función toda operación que se realice con cada tipo de datos para que queden reflejados los distintos niveles de abstracción presentes en el algoritmo. Con todo, aun a costa de alguna línea de código más, se consigue un programa inteligible y de fácil manutención.

ANOTACIONES

3.8. Ficheros de cabecera

Es común que varios equipos de programadores colaboren en la realización de un mismo programa, si bien es cierto que, muchas veces, esta afirmación es más bien la manifestación de un deseo que el reflejo de una realidad. En empresas pequeñas y medianas se suele traducir en que los equipos sean de una persona e, incluso, que los diferentes equipos se reduzcan a uno solo. De todas maneras, la afirmación, en el fondo, es cierta: la elaboración de un programa debe aprovechar, en una buena práctica de la programación, partes de otros programas. El reaprovechamiento permite no sólo reducir el tiempo de desarrollo, sino también tener la garantía de emplear componentes de probada funcionalidad.

Todo esto es, además, especialmente cierto para el software libre, en el que los programas son, por norma, producto de un conjunto de programadores diverso y no forzosamente coordinado: un programador puede haber aprovechado código elaborado previamente por otros para una aplicación distinta de la que motivó su desarrollo original.

Para permitir el aprovechamiento de un determinado código, es conveniente eliminar los detalles de implementación e indicar sólo el tipo de datos para el cual está destinado y qué operaciones se pueden realizar con las variables correspondientes. Así pues, basta con disponer de un fichero donde se definan el tipo de datos abstracto y se declaren las funciones que se proveen para las variables del mismo. A estos ficheros se los llama ficheros de cabecera por constituirse como la parte inicial del código fuente de las funciones cuyas declaraciones o cabeceras se han incluido en los mismos.

En los ficheros de cabecera se da a conocer todo aquello relativo a un tipo de datos abstracto y a las funciones para su manejo.

3.8.1. Estructura

Los ficheros de cabecera llevan la extensión ".h" y su contenido debe organizarse de tal manera que sea de fácil lectura. Para ello, primero

ANOTACIONES

se deben colocar unos comentarios indicando la naturaleza de su contenido y, sobre todo, de la funcionalidad del código en el fichero ".c" correspondiente. Después, es suficiente con seguir la estructura de un programa típico de C: inclusión de ficheros de cabecera, definición de constantes simbólicas y, por último, declaración de las funciones.

La definición debe estar siempre en el fichero ".c".

Sirva como ejemplo el siguiente fichero de cabecera para operar con números complejos:

```
/* Fichero:    complejos.h                              */
/* Contenido:  Funciones para operar con números        */
/*             complejos del tipo (X + iY) en los que   */
/*             X es la parte real e Y, la imaginaria.   */
/* Revisión: 0.0 (original)                             */

#ifndef _NUMEROS_COMPLEJOS_H_
#define _NUMEROS_COMPLEJOS_H_

#include <stdio.h>
#define PRECISION 1E-10
typedef struct complex_s {
      double real, imaginario;
} *complex_t;
complex_t nuevo_complejo( double real, double imaginario );
void borra_complejo( complex_t complejo );
void imprime_complejo( FILE *fichero, complex_t complejo );
double modulo_complejo( complex_t complejo );
complex_t opuesto_complejo( complex_t complejo );
complex_t suma_complejos( complex_t c1, complex_t c2 );
/* etcétera */

#endif /* _NUMEROS_COMPLEJOS_H_ */
```

Las definiciones de tipos y constantes y las declaraciones de funciones se han colocado como el cuerpo del comando del preprocesador

`#ifndef` ... `#endif`. Este comando pregunta si una determinada constante está definida y, de no estarlo, traslada al compilador lo que contenga el fichero hasta la marca de final. El primer comando del cuerpo de este comando condicional consiste, precisamente, en definir la constante `_NUMEROS_COMPLEJOS_H_` para evitar que una nueva inclusión del mismo fichero genere el mismo código fuente para el compilador (es innecesario si ya lo ha procesado una vez).

Los llamados ***comandos de compilación condicional*** del preprocesador permiten decidir si un determinado trozo de código fuente se suministra al compilador o no. Los resumimos en la tabla siguiente:

Tabla 8.

Comando	Significado
`#if expresión`	Las líneas siguientes son compiladas si `expresión ? 0.`
`#ifdef SÍMBOLO`	Se compilan las líneas siguientes si `SÍMBOLO` está definido.
`#ifndef SÍMBOLO`	Se compilan las líneas siguientes si `SÍMBOLO` no está definido.
`#else`	Finaliza el bloque compilado si se cumple la condición e inicia el bloque a compilar en caso contrario.
`#elif expresión`	Encadena un `else` con un `if`.
`#endif`	Indica el final del bloque de compilación condicional.

Nota

Un símbolo definido (por ejemplo: `SÍMBOLO`) puede anularse mediante:

```
#undef SÍMBOLO
```

y, a partir de ese momento, se considera como no definido.

Las formas:

```
#ifdef SÍMBOLO
#ifndef SÍMBOLO
```

son abreviaciones de:

```
#if defined( SÍMBOLO )
#if !defined( SÍMBOLO )
```

respectivamente. La función `defined` puede emplearse en expresiones lógicas más complejas.

Para acabar esta sección, cabe indicar que el fichero de código fuente asociado debe incluir, evidentemente, su fichero de cabecera:

```
/* Fichero:   complejos.c                        */
/* ... */
#include "complejos.h"
/* ... */
```

Nota

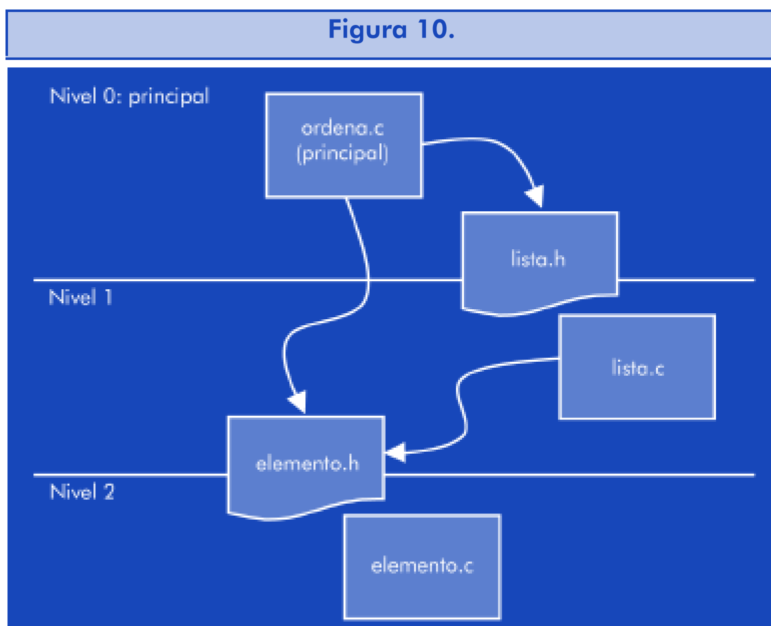
La inclusión se hace indicando el fichero entre comillas dobles en lugar de utilizar "paréntesis angulares" (símbolos de mayor y menor que) porque se supone que el fichero que se incluye se encuentra en el mismo directorio que el fichero que contiene la directiva de inclusión. Los paréntesis angulares se deben usar si se requiere que el preprocesador examine el conjunto de caminos de acceso a directorios estándar de ficheros de inclusión, como es el caso de `stdio.h`, por ejemplo.

3.8.2. Ejemplo

En el ejemplo de la ordenación por selección, tendríamos un fichero de cabecera para cada tipo de datos abstracto y, evidentemente, los correspondientes ficheros con el código en C. Además, dispondríamos de un tercer fichero que contendría el código del programa principal. La figura siguiente refleja el esquema de relaciones entre estos ficheros:

ANOTACIONES

Figura 10.



Cada fichero de código fuente puede compilarse independientemente. Es decir, en este ejemplo habría tres unidades de compilación. Cada una de ellas puede, pues, ser desarrollada independientemente de las demás. Lo único que deben respetar aquellas unidades que tienen un fichero de cabecera es no modificar ni los tipos de datos ni las declaraciones de las funciones. De esta manera, las demás unidades que hagan uso de ellas no deberán modificar sus llamadas y, por tanto, no requerirán ningún cambio ni compilación.

Es importante tener en cuenta que sólo puede existir una unidad con una función `main` (que es la que se corresponde con `ordena.c`, en el ejemplo dado). Las demás deberán ser compendios de funciones asociadas a un determinado tipo de datos abstracto.

El uso de ficheros de cabecera permite, además, cambiar el código de alguna función, sin tener que cambiar por ello nada de los programas que la emplean. Claro está, siempre que el cambio no afecte al "contrato" establecido en el fichero de cabecera correspondiente.

En el fichero de cabecera se describe toda la funcionalidad que se provee para un determinado tipo de datos abstracto y, con esta descripción, se adquiere el compromiso de que se mantendrá independientemente de cómo se materialice en el código asociado.

ANOTACIONES

Para ilustrar esta idea se puede pensar en que, en el ejemplo, es posible cambiar el código de las funciones que trabajan con las listas para que éstas sean de tipo dinámico sin cambiar el contrato adquirido en el fichero de cabecera.

A continuación se lista el fichero de cabecera para las listas en el caso de la ordenación:

```
/* Fichero: lista.h                                            */
#ifndef _LISTA_VEC_H_
#define _LISTA_VEC_H_

#include <stdio.h>
#include "bool.h"
#include "elemento.h"

#define LONG_MAXIMA 100
typedef struct lista_e {
  elemento_t     nodo[ LONG_MAXIMA ];
  unsigned short posicion; /* Posición actual de acceso.  */
  unsigned short cantidad; /* Longitud de la lista.       */
} lista_t;

void inicializa_lista( lista_t *ref_lista );
bool esta_vacia_lista( lista_t lista );
bool es_final_de_lista( lista_t lista );
void principio_de_lista( lista_t *lista_ref );
ref_nodo_t ref_nodo_de_lista( lista_t lista );
elemento_t elemento_en_ref_nodo(
  lista_t    lista,
  ref_nodo_t refnodo
);
void avanza_posicion_en_lista( lista_t *lista_ref );
elemento_t extrae_minimo_de_lista( lista_t *ref_lista );
void pon_al_final_de_lista(
  lista_t *ref_lista,
  elemento_t elemento
);

#endif /* _LISTA_VEC_H_ */
```

ANOTACIONES

Como se puede observar, se podría cambiar la forma de extracción del elemento mínimo sin necesidad de modificar el fichero de cabecera, y menos aún la llamada en el programa principal. Sin embargo, un cambio en el tipo de datos para, por ejemplo, implementar las listas mediante variables dinámicas, implicaría volver a compilar todas las unidades que hagan uso de las mismas, aunque no se modifiquen las cabeceras de las funciones. De hecho, en estos casos, resulta muy conveniente mantener el contrato al respecto de las mismas, pues evitará modificaciones en los códigos fuente de las unidades que hagan uso de ellas.

3.9. Bibliotecas

Las bibliotecas de funciones son, de hecho, unidades de compilación. Como tales, cada una dispone de un fichero de código fuente y uno de cabecera. Para evitar la compilación repetitiva de las bibliotecas, el código fuente ya ha sido compilado (ficheros de extensión ".o") y sólo quedan pendientes de su enlace con el programa principal.

Las bibliotecas de funciones, de todas maneras, se distinguen de las unidades de compilación por cuanto sólo se incorporan dentro del programa final aquellas funciones que son necesarias: las que no se utilizan, no se incluyen. Los ficheros compilados que permiten esta opción tienen la extensión ".l".

3.9.1. Creación

Para obtener una unidad de compilación de manera que sólo se incluyan en el programa ejecutable las funciones utilizadas, hay que compilarla para obtener un fichero de tipo objeto; es decir, con el código ejecutable sin enlazar:

```
$ gcc –c –o biblioteca.o biblioteca.c
```

Se supone, que en el fichero de `biblioteca.c` tal como se ha indicado, hay una inclusión del fichero de cabecera apropiado.

ANOTACIONES

Una vez generado el fichero objeto, es necesario incluirlo en un archivo (con extensión ".a") de ficheros del mismo tipo (puede ser el único si la biblioteca consta de una sola unidad de compilación). En este contexto, los "archivos" son colecciones de ficheros objeto reunidos en un único fichero con un índice para localizarlos y, sobre todo, para determinar qué partes del código objeto corresponden a qué funciones. Para crear un archivo hay que ejecutar el siguiente comando:

```
$ ar biblioteca.a biblioteca.o
```

Para construir el índice (la tabla de símbolos y localizaciones) debe de ejecutarse el comando:

```
$ ar -s biblioteca.a
```

o bien:

```
$ ranlib biblioteca.a
```

El comando de gestión de archivos `ar` permite, además, listar los ficheros objeto que contiene, añadir o reemplazar otros nuevos o modificados, actualizarlos (se lleva a cabo el reemplazo si la fecha de modificación es posterior a la fecha de inclusión en el archivo) y eliminarlos si ya no son necesarios. Esto se hace, respectivamente, con los comandos siguientes:

```
$ ar -t biblioteca.a
$ ar -r nuevo.o biblioteca.a
$ ar -u actualizable.o biblioteca.a
$ ar -d obsoleto.o biblioteca.a
```

Con la información de la tabla de símbolos, el enlazador monta un programa ejecutable empleando sólo aquellas funciones a las que se refiere. Para lo demás, los archivos son similares a los ficheros de código objeto simples.

3.9.2. Uso

El empleo de las funciones de una biblioteca es exactamente igual que el de las funciones de cualquier otra unidad de compilación.

ANOTACIONES

Basta con incluir el fichero de cabecera apropiado e incorporar las llamadas a las funciones que se requieran dentro del código fuente.

3.9.3. Ejemplo

En el ejemplo de la ordenación se ha preparado una unidad de compilación para las listas. Como las listas son un tipo de datos dinámico que se utiliza mucho, resulta conveniente disponer de una biblioteca de funciones para operar con ellas. De esta manera, no hay que programarlas de nuevo en ocasiones posteriores.

Para transformar la unidad de listas en una biblioteca de funciones de listas hay que hacer que el tipo de datos no dependa en absoluto de la aplicación. En caso contrario, habría que compilar la unidad de listas para cada nuevo programa.

En el ejemplo, las listas contenían elementos del tipo `elemento_t`, que era un apuntador a `dato_t`. En general, las listas podrán tener elementos que sean apuntadores a cualquier tipo de datos. Sea como sea, todo son direcciones de memoria. Por este motivo, en lo que atañe a las listas, los elementos serán de un tipo vacío, que el usuario de la biblioteca de funciones habrá de definir. Así pues, en la unidad de compilación de las listas se incluye:

```
typedef void *elemento_t;
```

Por otra parte, para realizar la función de extracción del elemento más pequeño, ha de saber a qué función llamar para realizar la comparación. Por tanto, se le añade un parámetro más que consiste en la dirección de la función de comparación:

```
elemento_t extrae_minimo_de_lista(
  lista_t *ref_lista,
  bool (*compara_elementos)( elemento_t, elemento_t )
); /* extrae_minimo_de_lista */
```

El segundo argumento es un apuntador a una función que toma como parámetros dos elementos (no es necesario dar un nombre a

ANOTACIONES

los parámetros formales) y devuelve un valor lógico de tipo `bool`. En el código fuente de la definición de la función, la llamada se efectuaría de la siguiente manera:

```
/* ... */
es_menor = (*compara_elementos)(
          elemento_en_ref_nodo( lista, actual ),
          peque
); /* compara_elementos */
/* ... */
```

Por lo demás, no habría de hacerse cambio alguno.

Así pues, la decisión de transformar alguna unidad de compilación de un programa en biblioteca dependerá fundamentalmente de dos factores:

- El tipo de datos y las operaciones han de poder ser empleados en otros programas.
- Raramente se deben emplear todas las funciones de la unidad.

3.10. Herramienta *make*

La compilación de un programa supone, normalmente, compilar algunas de sus unidades y luego enlazarlas todas junto con las funciones de biblioteca que se utilicen para montar el programa ejecutable final. Por lo tanto, para obtenerlo, no sólo hay que llevar a cabo una serie de comandos, sino que también hay que tener en cuenta qué ficheros han sido modificados.

Las herramientas de tipo *make* permiten establecer las relaciones entre ficheros de manera que sea posible determinar cuáles dependen de otros. Así, cuando detecta que alguno de los ficheros tiene una fecha y hora de modificación anterior a alguno de los que depende, se ejecuta el comando indicado para generarlos de nuevo.

ANOTACIONES

De esta manera, no es necesario preocuparse de qué ficheros hay que generar y cuáles no hace falta actualizar. Por otra parte, evita tener que ejecutar individualmente una serie de comandos que, para programas grandes, puede ser considerable.

El propósito de las herramientas de tipo *make* es determinar automáticamente qué piezas de un programa deben ser recompiladas y ejecutar los comandos pertinentes.

La herramienta `gmake` (o, simplemente, `make`) es una utilidad *make* de GNU (www.gnu.org/software/make) que se ocupa de lo anteriormente mencionado. Para poder hacerlo, necesita un fichero que, por omisión, se denomina `makefile`. Es posible indicarle un fichero con otro nombre si se la invoca con la opción `–f`:

```
$ make –f fichero_objetivos
```

3.10.1. Fichero *makefile*

En el fichero *makefile* hay que especificar los objetivos (habitualmente, ficheros a construir) y los ficheros de los que dependen (los prerrequisitos para cumplir los objetivos). Para cada objetivo es necesario construir una regla, cuya estructura es la siguiente:

```
# Sintaxis de una regla:
objetivo : fichero1 fichero2 ... ficheroN
   comando1
   comando2
   ...
   comandoK
```

Nota

La primera de estas líneas debe empezar forzosamente por un carácter de tabulación.

El símbolo # se emplea para introducir una línea de comentarios. Toda regla requiere que se indique cuál es el objetivo y, después de los dos puntos, se indiquen los ficheros de los cuáles depende. En las líneas siguientes se indicarán los comandos que deben de ejecutarse. Cualquier línea que quiera continuarse deberá finali-

zar con un salto de línea precedido del carácter de “escape” o barra inversa (\).

Tomando como ejemplo el programa de ordenación de notas de un examen por DNI, podría construirse un *makefile* como el mostrado a continuación para simplificar la actualización del ejecutable final:

```
# Compilación del programa de ordenación:
clasifica : ordena.o nota.o lista.a
   gcc -g -o clasifica ordena.o nota.o lista.a
ordena.o : ordena.c
   gcc -g -c -o ordena.o ordena.c
nota.o : nota.c nota.h
   gcc -g -c -o nota.o nota.c
lista.a : lista.o
   ar -r lista.a lista.o ;
   ranlib lista.a
lista.o : lista.c lista.h
   gcc -g -c -o lista.o lista.c
```

La herramienta *make* procesaría el fichero anterior revisando los prerrequisitos del primer objetivo, y si éstos son a su vez objetivos de otras reglas, se procederá de la misma manera para estos últimos. Cualquier prerrequisito terminal (que no sea un objetivo de alguna otra regla) modificado con posterioridad al objetivo al que afecta provoca la ejecución en serie de los comandos especificados en las siguientes líneas de la regla.

Es posible especificar más de un objetivo, pero `make` sólo examinará las reglas del primer objetivo final que encuentre. Si se desea que procese otros objetivos, será necesario indicarlo así en su invocación.

Es posible tener objetivos sin prerrequisitos, en cuyo caso, siempre se ejecutarán los comandos asociados a la regla en la que aparecen.

Es posible tener un objetivo para borrar todos los ficheros objeto que ya no son necesarios.

ANOTACIONES

Ejemplo

Retomando el ejemplo anterior, sería posible limpiar el directorio de ficheros innecesarios añadiendo el siguiente texto al final del *makefile*:

```
# Limpieza del directorio de trabajo:
limpia :
   rm -f ordena.o nota.o lista.o
```

Para conseguir este objetivo, bastaría con introducir el comando:

```
$ make limpieza
```

Es posible indicar varios objetivos con unos mismos prerrequisitos:

```
# Compilación del programa de ordenación:
todo depura óptimo : clasifica
depura : CFLAGS := -g
optimo : CFLAGS := -O
clasifica : ordena.o nota.o lista.a
   gcc $(CFLAGS) -o clasifica ordena.o nota.o lista.a
# resto del fichero ...
```

Nota

A la luz del fragmento anterior, podemos ver lo siguiente:

– Si se invoca `make` sin argumento, se actualizará el objetivo `todo` (el primero que encuentra), que depende de la actualización del objetivo secundario `clasifica`.

– Si se especifica el objetivo `depura` o el `optimo`, habrá de comprobar la consistencia de dos reglas: en la primera se indica que para conseguirlos hay que actualizar `clasifica` y, en la segunda, que hay que asignar a la variable `CCFLAGS` un valor.

ANOTACIONES

Dado que *make* primero analiza las dependencias y luego ejecuta los comandos oportunos, el resultado es que el posible montaje de *compila* se hará con el contenido asignado a la variable CCFLAGS en la regla precedente: el acceso al contenido de una variable se realiza mediante el operador correspondiente, que se representa por el símbolo del dólar.

En el ejemplo anterior ya puede apreciarse que la utilidad de *make* va mucho más allá de lo que aquí se ha contado y, de igual forma, los *makefile* tienen muchas maneras de expresar reglas de forma más potente. Aun así, se han repasado sus principales características desde el punto de vista de la programación y visto algunos ejemplos significativos.

3.11. Relación con el sistema operativo. Paso de parámetros a programas

Los programas se traducen a instrucciones del lenguaje máquina para ser ejecutados. Sin embargo, muchas operaciones relacionadas con los dispositivos de entrada, de salida y de almacenamiento (discos y memoria, entre otros) son traducidas a llamadas a funciones del sistema operativo.

Una de las funciones del sistema operativo es, precisamente, permitir la ejecución de otros programas en la máquina y, en definitiva, del software. Para ello, provee al usuario de ciertos mecanismos para que elija qué aplicaciones desea ejecutar. Actualmente, la mayoría son, de hecho, interfaces de usuario gráficos. Aun así, siguen existiendo los entornos textuales de los intérpretes de comandos, denominados comúnmente *shells*.

En estos *shells*, para ejecutar programas (algunos de ellos, utilidades del propio sistema y otros, de aplicaciones) basta con suministrarles el nombre del fichero de código ejecutable correspondiente. Ciertamente, también existe la posibilidad de que un programa sea invocado por otro.

De hecho, la función main de los programas en C puede tener distintos argumentos. En particular, existe la convención de que el pri-

ANOTACIONES

mer parámetro sea el número de argumentos que se le pasan a través del segundo, que consiste en un vector de cadenas de caracteres. Para aclarar cómo funciona este procedimiento de paso de parámetros, sirva como ejemplo el siguiente programa, que nos descubrirá qué sucede cuando lo invocamos desde un *shell* con un cierto número de argumentos:

```
/* Fichero: args.c */
#include <stdio.h>
int main( int argc, char *argv[] )
{
   int i;
   printf( "Núm. argumentos, argc = %i\n", argc );
   for( i = 0; i < argc; i = i + 1 ) {
    printf( "Argumento argv[%i] = \"%s\"\n", i, argv[i] );
   } /* for */
   return argc;
} /* main */
```

Si se hace la invocación con el comando siguiente, el resultado será el que se indica a continuación:

```
$ args -prueba número 1

Núm. argumentos, argc = 4
Argumento argv[0] = "args"
Argumento argv[1] = "-prueba"
Argumento argv[2] = "número"
Argumento argv[3] = "1"
$
```

Es importante tener presente que la propia invocación de programa se toma como argumento 0 del comando. Así pues, en el ejemplo anterior, la invocación en la que se dan al programa tres parámetros se convierte, finalmente, en una llamada a la función `main` en la que se adjuntará también el texto con el que ha sido invocado el propio programa como primera cadena de caracteres del vector de argumentos.

Puede consultarse el valor retornado por `main` para determinar si ha habido algún error durante la ejecución o no. Generalmente, se

ANOTACIONES

toma por convenio que el valor devuelto debe ser el código del error correspondiente o 0 en ausencia de errores.

Nota

En el código fuente de la función es posible emplear las constantes `EXIT_FAILURE` y `EXIT_SUCCESS`, que están definidas en `stdlib.h`, para indicar el retorno con o sin error, respectivamente.

En el siguiente ejemplo, se muestra un programa que efectúa la suma de todos los parámetros presentes en la invocación. Para ello, emplea la función `atof`, declarada en `stdlib.h`, que convierte cadenas de texto a números reales. En caso de que la cadena no representara un número, devuelve un cero:

```
/* Fichero: suma.c */
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char *argv[] )
{
   float   suma = 0.0;
   int     pos = 1;

   while( pos < argc ) {
     suma = suma + atof( argv[ pos ] );
     pos = pos + 1;
   } /* while */
   printf( " = %g\n", suma );
   return 0;
} /* main */
```

Nota

En este caso, el programa devuelve siempre el valor 0, puesto que no hay errores de ejecución que indicar.

3.12. Ejecución de funciones del sistema operativo

Un sistema operativo es un software que permite a las aplicaciones que se ejecutan en un ordenador abstraerse de los detalles de la máquina. Es decir, las aplicaciones pueden trabajar con una máquina virtual que es capaz de realizar operaciones que la máquina real no puede entender.

ANOTACIONES

Además, el hecho de que los programas se definan en términos de las rutinas (o funciones) proveídas por el SO aumenta su portabilidad, su capacidad de ser ejecutados en máquinas distintas. En definitiva, los hace independientes de la máquina, pero no del SO, obviamente.

En C, muchas de las funciones de la biblioteca estándar emplean las rutinas del SO para llevar a cabo sus tareas. Entre estas funciones se encuentran las de entrada y salida de datos, de ficheros y de gestión de memoria (variables dinámicas, sobre todo).

Generalmente, todas las funciones de la biblioteca estándar de C tienen la misma cabecera y el mismo comportamiento, incluso con independencia del sistema operativo; no obstante, hay algunas que dependen del sistema operativo: puede haber algunas diferencias entre Linux y Microsoft. Afortunadamente, son fácilmente detectables, pues las funciones relacionadas a un determinado sistema operativo están declaradas en ficheros de cabecera específicos.

En todo caso, a veces resulta conveniente ejecutar los comandos del *shell* del sistema operativo en lugar de ejecutar directamente las funciones para llevarlos a término. Esto permite, entre otras cosas, que el programa correspondiente pueda describirse a un nivel de abstracción más alto y, con ello, aprovechar que sea el mismo intérprete de comandos el que complete los detalles necesarios para llevar a término la tarea encomendada. Generalmente, se trata de ejecutar órdenes internas del propio *shell* o aprovechar sus recursos (caminos de búsqueda y variables de entorno, entre otros) para ejecutar otros programas.

Para poder ejecutar un comando del *shell*, basta con suministrar a la función `system` la cadena de caracteres que lo describa. El valor que devuelve es el código de retorno del comando ejecutado o –1 en caso de error.

En Linux, `system( comando )` ejecuta `/bin/sh -c comando;` es decir, emplea `sh` como intérprete de comandos. Por lo tanto, éstos tienen que ajustarse a la sintaxis del mismo.

ANOTACIONES

En el programa siguiente muestra el código devuelto por la ejecución de un comando, que hay que introducir entre comillas como argumento del programa:

```
/* Fichero: ejecuta.c */
#include <stdio.h>
#include <stdlib.h>

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

   if( argc == 2 ) {
     codigo = system( argv[1] );
     printf( "%i = %s\n", codigo, argv[1] );
   } else {
     printf( "Uso: ejecuta \"comando\"\n" );
   } /* if */
   return 0;
} /* main */
```

Aunque simple, el programa anterior nos da una cierta idea de cómo emplear la función `system`..

El conjunto de rutinas y servicios que ofrece el sistema operativo va más allá de dar soporte a las funciones de entrada y salida de datos, de manejo de ficheros y de gestión de memoria, también permite lanzar la ejecución de programas dentro de otros. En el apartado siguiente, se tratará con más profundidad el tema de la ejecución de programas.

3.13. Gestión de procesos

Los sistemas operativos actuales son capaces, además de todo lo visto, de ejecutar una diversidad de programas en la misma máquina en un mismo período de tiempo. Evidentemente, esto sólo es posible si se ejecutan en procesadores distintos o si su ejecución se realiza de forma secuencial o intercalada en un mismo procesador.

ANOTACIONES

Normalmente, a pesar de que el sistema operativo sea capaz de gestionar una máquina con varios procesadores, habrá más programas a ejecutar que recursos para ello. Por este motivo, siempre tendrá que poder planificar la ejecución de un determinado conjunto de programas en un único procesador. La planificación puede ser:

- Secuencial. Una vez finalizada la ejecución de un programa, se inicia la del programa siguiente (también se conoce como *ejecución por lotes*).
- Intercalada. Cada programa dispone de un cierto tiempo en el que se lleva a cabo una parte de su flujo de ejecución de instrucciones y, al término del periodo de tiempo asignado, se ejecuta una parte de otro programa.

De esta manera, se pueden ejecutar diversos programas en un mismo intervalo de tiempo dando la sensación de que su ejecución progresa paralelamente.

Apoyándose en los servicios (funciones) que ofrece el SO al respecto de la ejecución del software, es posible, entre otras cosas, ejecutarlo disociado de la entrada/salida estándar y/o partir su flujo de ejecución de instrucciones en varios paralelos.

3.13.1. Definición de proceso

Respecto del sistema operativo, cada flujo de instrucciones que debe gestionar es un proceso. Por lo tanto, repartirá su ejecución entre los diversos procesadores de la máquina y en el tiempo para que lleven a cabo sus tareas progresivamente. Habrá, eso sí, algunos procesos que se dividirán en dos paralelos, es decir, el programa consistirá, a partir de ese momento, en dos procesos distintos.

Cada proceso tiene asociado, al iniciarse, una entrada y salida estándar de datos de la que puede disociarse para continuar la ejecución en segundo plano, algo habitual en los procesos permanentes, que tratamos en el apartado siguiente.

ANOTACIONES

Los procesos que comparten un mismo entorno o estado, con excepción evidente de la referencia a la instrucción siguiente, son denominados *hilos* o *hebras* (en inglés, *threads*), mientras que los que tienen entornos distintos son denominados simplemente *procesos*.

Con todo, un programa puede organizar su código de manera que lleve a término su tarea mediante varios flujos de instrucciones paralelos, sean éstos simples hilos o procesos completos.

3.13.2. Procesos permanentes

Un proceso permanente es aquel que se ejecuta indefinidamente en una máquina. Suelen ser procesos que se ocupan de la gestión automatizada de entradas y salidas de datos y, por lo tanto, con escasa interacción con los usuarios.

Así pues, muchas de las aplicaciones que funcionan con el modelo cliente-servidor se construyen con procesos permanentes para el servidor y con procesos interactivos para los clientes. Un ejemplo claro de tales aplicaciones son las relacionadas con Internet: los clientes son programas, como el gestor de correo electrónico o el navegador, y los servidores son programas que atienden a las peticiones de los clientes correspondientes.

En Linux, a los procesos permanentes se les llama, gráficamente, demonios (*daemons*, en inglés) porque aunque los usuarios no pueden observarlos puesto que no interactúan con ellos (en especial, no lo hacen a través del terminal estándar), existen: los demonios son "espíritus" de la máquina que el usuario no ve pero cuyos efectos percibe.

Nota

Literalmente, un demonio es un espíritu maligno, aunque se supone que los procesos denominados como tales no deberían serlo.

Para crear un demonio, basta con llamar a la función `daemon`, declarada en `unistd.h`, con los parámetros adecuados. El primer argumento indica si no cambia de directorio de trabajo y el segundo, si no se disocia del terminal estándar de entrada/salida; es decir, una llamada común habría de ser como sigue:

```
/* ... */
if( daemon( FALSE, FALSE ) ) == 0 ) {
   /* cuerpo */
} /* if */
/* resto del programa, tanto si se ha creado como si no.  */
```

ANOTACIONES

> **Nota**
>
> Esta llamada consigue que el cuerpo del programa sea un demonio que trabaja en el directorio raíz (como si hubiera hecho un `cd /`) y que está disociado de las entradas y salidas estándar (en realidad, redirigidas al dispositivo vacío: `/dev/null`). La función devuelve un código de error, que es cero si todo ha ido bien.

Ejemplo

Para ilustrar el funcionamiento de los demonios, se muestra un programa que avisa al usuario de que ha transcurrido un cierto tiempo. Para ello, habrá que invocar al programa con dos parámetros: uno para indicar las horas y minutos que deben transcurrir antes de advertir al usuario y otro que contenga el texto del aviso. En este caso, el programa se convertirá en un demonio no disociado del terminal de entrada/salida estándar, puesto que el aviso aparecerá en el mismo.

```
/* Fichero: alarma.c                                          */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "bool.h"

int main( int argc, char *argv[] )
{
   unsigned int horas;
   unsigned int minutos;
   unsigned int segundos;
   cha          *aviso, *separador;

   if( argc == 3 ) {
     separador = strchr( argv[1], ':' );
     if( separador != NULL ) {
      horas = atoi( argv[1] );
      minutos = atoi( separador+1 );
     } else {
      horas = 0;
minutos = atoi( argv[1] );
```

ANOTACIONES

```
    } /* if */
    segundos = (horas*60 + minutos) * 60;
    aviso = argv[2];
    if( daemon( FALSE, TRUE ) ) {
     printf( "No puede instalarse el avisador :-(\n" );
    } else {
     printf( "Alarma dentro de %i horas y %i minutos.\n",
      horas, minutos
     ); /* printf */
     printf( "Haz $ kill %li para apagarla.\n",
      getpid()
     ); /* printf */
    } /* if */
    sleep( segundos );
    printf( "%s\007\n", aviso );
    printf( "Alarma apagada.\n" );
   } else {
    printf( "Uso: %s horas:minutos \"aviso\"\n", argv[0] );
   } /* if */
   return 0;
} /* main */
```

La lectura de los parámetros de entrada ocupa buena parte del código. En particular, lo que necesita más atención es la extracción de las horas y de los minutos; para ello, se buscan los dos puntos (con `strchr`, declarada en `string.h`) y luego se toma la cadena entera para determinar el valor de las horas y la cadena a partir de los dos puntos para los minutos.

La espera se realiza mediante una llamada a la función `sleep`, que toma como argumento el número de segundos en los que el programa debe "dormir", es decir, suspender su ejecución.

Finalmente, para dar al usuario la posibilidad de parar la alarma, se le informa del comando que debe introducir en el *shell* para "matar" el proceso (es decir, para finalizar su ejecución). A tal efecto, se le muestra el número de proceso que se corresponde con el demonio instalado. Este identificador se consigue llamando a la función `getpid()`, en el que PID significa, precisamente, identificador de proceso

ANOTACIONES

Uno de los usos fundamentales de los demonios es el de la implementación de procesos proveedores de servicios.

3.13.3. Procesos concurrentes

Los procesos concurrentes son aquellos que se ejecutan simultáneamente en un mismo sistema. Al decir *simultáneamente*, en este contexto, entendemos que se llevan a cabo en un mismo período de tiempo bien en procesadores distintos, bien repartidos temporalmente en un mismo procesador, o bien en los dos casos anteriores.

El hecho de repartir la ejecución de un programa en diversos flujos de instrucciones concurrentes puede perseguir alguno de los objetivos siguientes:

- Aprovechar los recursos en un sistema multiprocesador. Al ejecutarse cada flujo de instrucciones en un procesador distinto, se consigue una mayor rapidez de ejecución. De hecho, sólo en este caso se trata de procesos de ejecución verdaderamente simultánea.

Nota

Cuando dos o más procesos comparten un mismo procesador, no hay más remedio que ejecutarlos por tramos en un determinado período de tiempo dentro del que, efectivamente, se puede observar una evolución progresiva de los mismos.

- Aumentar el rendimiento respecto de la entrada/salida de datos. Para aumentar el rendimiento respecto de la E/S del programa puede resultar conveniente que uno de los procesos se ocupe de la relación con la entrada de datos, otro del cálculo que haya que hacer con ellos y, finalmente, otro de la salida de resultados. De esta manera, es posible realizar el cálculo sin detenerse a dar salida a los resultados o esperar datos de entrada. Ciertamente, no siempre cabe hacer tal partición y el número de procesos puede variar mucho en función de las necesidades del

ANOTACIONES

programa. De hecho, en este caso se trata, fundamentalmente, de separar procesos de naturaleza lenta (por ejemplo, los que deben comunicarse con otros bien para recibir bien para transmitir datos) de otros procesos más rápidos, es decir, con mayor atención al cálculo.

En los siguientes apartados se comentan varios casos de programación concurrente tanto con “procesos ligeros” (los hilos) como con procesos completos o “pesados” (por contraposición a los ligeros).

3.14. Hilos

Un hilo, hebra o *thread* es un proceso que comparte el entorno con otros del mismo programa, lo que comporta que el espacio de memoria sea el mismo. Por tanto, la creación de un nuevo hilo sólo implica disponer de información sobre el estado del procesador y la instrucción siguiente para el mismo. Precisamente por este motivo son denominados “procesos ligeros”.

Los hilos son flujos de ejecución de instrucciones independientes que tienen mucha relación entre sí.

Para emplearlos en C sobre Linux, es necesario hacer llamadas a funciones de hilos del estándar de POSIX. Este estándar define una interfaz portable de sistemas operativos (originalmente, Unix) para entornos de computación, de cuya expresión en inglés toma el acrónimo.

Las funciones de POSIX para hilos están declaradas en el fichero `pthread.h` y se debe de enlazar el archivo de la biblioteca correspondiente con el programa. Para ello, hay que compilar con el comando siguiente:

```
$ gcc -o ejecutable codigo.c -lpthread
```

ANOTACIONES

Nota

La opción –`lpthread` indica al enlazador que debe incluir también la biblioteca de funciones POSIX para hilos.

3.14.1. Ejemplo

Mostraremos un programa para determinar si un número es primo o no como ejemplo de un programa desenhebrado en dos hilos. El hilo principal se ocupará de buscar posibles divisores mientras que el secundario actuará de "observador" para el usuario: leerá los datos que maneja el hilo principal para mostrarlos por el terminal de salida estándar. Evidentemente, esto es posible porque comparten el mismo espacio de memoria.

La creación de los hilos requiere que su código esté dentro de una función que sólo admite un parámetro del tipo `(void *)`. De hecho las funciones creadas para ser hilos POSIX deben obedecer a la cabecera siguiente:

```
(void *)hilo( void *referencia_parámetros );
```

Así pues, será necesario colocar en una tupla toda la información que se quiera hacer visible al usuario y pasar su dirección como parámetro de la misma. Pasamos a definir la tupla de elementos que se mostrará:

```
/* ... */
typedef struct s_visible {
   unsigned long numero;
   unsigned long divisor;
   bool fin;
} t_visible;
/* ... */
```

Nota

El campo `fin` servirá para indicar al hilo hijo que el hilo principal (el padre) ha acabado su tarea. En este caso, ha determinado si el número es o no, primo.

ANOTACIONES

La función del hilo hijo será la siguiente:

```
/* ... */
void *observador( void *parametro )
{
   t_visible *ref_vista;

   ref_vista = (t_visible *)parametro;
   printf( " ... probando %012lu", 0 );
   do {
     printf( "\b\b\b\b\b\b\b\b\b\b\b\b" );
     printf( "%12lu", ref_vista->divisor );
   } while( !(ref_vista->fin) );
   printf( "\n" );
   return NULL;
} /* observador */
/* ... */
```

Nota

El carácter `'\b'` se corresponde con un retroceso y que, dado que los números se imprimen con 12 dígitos (los ceros a la izquierda se muestran como espacios), la impresión de 12 retrocesos implica borrar el número que se haya escrito anteriormente.

Para crear el hilo del observador, basta con llamar a `pthread_create()` con los argumentos adecuados. A partir de ese momento, un nuevo hilo se ejecuta concurrentemente con el código del programa:

```
/* ... */
int main( int argc, char *argv[] )
{
   int         codigo_error;  /* Código de error a devolver. */
   pthread_t   id_hilo;       /* Identificador del hilo.     */
   t_visible   vista;         /* Datos observables.          */
   bool        resultado;     /* Indicador de si es primo.   */
```

ANOTACIONES

```
{
   int          codigo_error;   /* Código de error a devolver.   */
   pthread_t    id_hilo;        /* Identificador del hilo.       */
   t_visible    vista;          /* Datos observables.            */
   bool         resultado;      /* Indicador de si es primo.     */

   if( argc == 2 ) {
     vista.numero = atol( argv[1] );
     vista.fin = FALSE;
     codigo_error = pthread_create(
     &id_hilo,      /* Referencia en la que poner el ID.       */
     NULL,          /* Referencia a posibles atributos.        */
     observador,    /* Función que ejecutará el hilo.          */
    (void *)&vista /* Argumento de la función.                */
     ); /* pthread_create */
     if( codigo_error == 0 ) {
       resultado = es_primo( &vista );
       vista.fin = TRUE;
       pthread_join( id_hilo, NULL );
       if( resultado )printf( "Es primo.\n" );
       else            printf( "No es primo.\n" );
       codigo_error = 0;
       } else {
       printf( "No he podido crear un hilo observador!\n" );
       codigo_error = 1;
     } /* if */
   } else {
     printf( "Uso: %s número\n", argv[0] );
     codigo_error = -1;
   } /* if */
   return codigo_error;
} /* main */
```

Nota

Después de crear el hilo del observador, se comprueba que el número sea primo y, al regresar de la función `es_primo()`, se pone el campo fin a `TRUE` para que el observador finalice su ejecución. Para esperar a que efectivamente haya acabado, se llama a `pthread_join()`. Esta función espera a que el hilo cuyo identificador se haya dado como primer argumento llegue al final de su ejecución y, por tanto, se produzca una unión de los hilos (de ahí el apelativo en inglés *join*). El segundo argumento se emplea para recoger posibles datos devueltos por el hilo.

ANOTACIONES

Para terminar el ejemplo, sería necesario codificar la función `es_primo()`, que tendría como cabecera la siguiente:

```
/* ... */
bool es_primo( t_visible *ref_datos );
/* ... */
```

La programación se deja como ejercicio. Para resolverlo adecuadamente, cabe tener presente que hay que emplear `ref_datos->divisor` como tal, puesto que la función `observador()` la lee para mostrarla al usuario.

En este caso, no existe ningún problema en que los hilos de un programa tengan el mismo espacio de memoria; es decir, el mismo entorno o contexto. De todas maneras, suele ser habitual que el acceso a datos compartidos por más de un hilo sea sincronizado. En otras palabras, que se habilite algún mecanismo para impedir que dos hilos accedan simultáneamente al mismo dato, especialmente para modificarlo, aunque también para que los hilos lectores lean los datos debidamente actualizados. Estos mecanismos de exclusión mutua son, de hecho, convenios de llamadas a funciones previas y posteriores al acceso a los datos.

Nota

Se puede imaginar como funciones de control de un semáforo de acceso a una plaza de aparcamiento: si está libre, el semáforo estará en verde y, si está ocupada, estará en rojo hasta que se libere.

3.15. Procesos

Un proceso es un flujo de ejecución de instrucciones con un entorno propio y, por tanto, con todas las atribuciones de un programa. Puede dividir, pues, el flujo de ejecución de instrucciones en otros procesos (ligeros o no) si así se considera conveniente por razones de eficiencia.

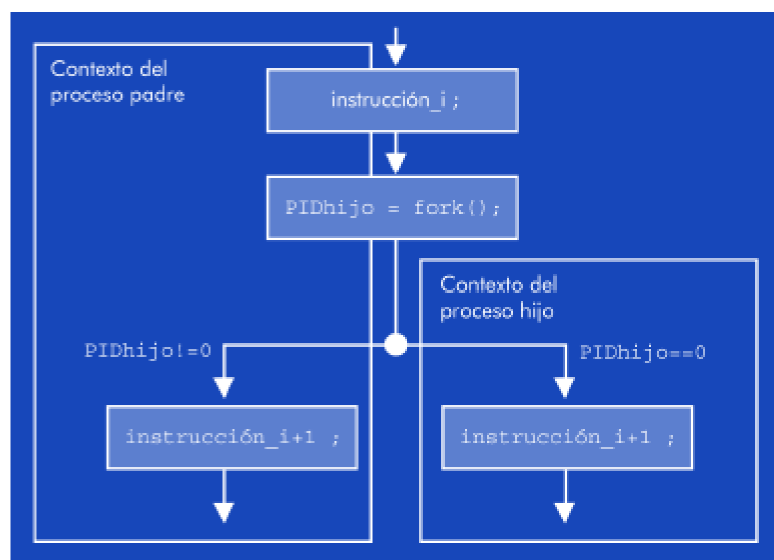
ANOTACIONES

En este caso, la generación de un nuevo proceso a partir del proceso principal implica realizar la copia de todo el entorno de este último. Con ello, el proceso hijo tiene una copia exacta del entorno del padre en el momento de la división. A partir de ahí, tanto los contenidos de las variables como la indicación de la instrucción siguiente puede divergir. De hecho, actúan como dos procesos distintos con entornos evidentemente diferentes del mismo código ejecutable.

Dada la separación estricta de los entornos de los procesos, generalmente éstos se dividen cuando hay que realizar una misma tarea sobre datos distintos, que lleva a cabo cada uno de los procesos hijos de forma autónoma. Por otra parte, existen también mecanismos para comunicar procesos entre sí: las tuberías, las colas de mensajes, las variables compartidas (en este caso, se cuenta con funciones para implementar la exclusión mutua) y cualquier otro tipo de comunicación que pueda establecerse entre procesos distintos.

Para crear un nuevo proceso, basta con llamar a `fork()`, cuya declaración se encuentra en `unistd.h`, y que devuelve el identificador del proceso hijo en el padre y cero en el nuevo proceso:

Figura 11.



Nota

El hecho de que `fork()` devuelva valores distintos en el proceso padre y en el hijo permite a los flujos de instrucciones siguientes determinar si pertenecen a uno u otro.

ANOTACIONES

El programa siguiente es un ejemplo simple de división de un proceso en dos: el original o padre y la copia o hijo. Para simplificar, se supone que tanto el hijo como el padre realizan una misma tarea. En este caso, el padre espera a que el hijo finalice la ejecución con `wait()`, que requiere de la inclusión de los ficheros `sys/types.h` y `sys/wait.h`:

```
/* Fichero: ej_fork.c                                          */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

/* ... */

int main( void )

{
  pid_t  proceso;
  int    estado;
  printf( "Proceso padre (%li) iniciado.\n", getpid() );
  proceso = fork();
  if( proceso == 0 ) {
    printf( "Proceso hijo (%li) iniciado.\n", getpid() );
    tarea( "hijo" );
    printf( "Fin del proceso hijo.\n" );
  } else {
    tarea( "padre" );
    wait( &estado ); /* Espera la finalización del hijo.       */
  printf( "Fin del proceso padre.\n" );
  } /* if */
  return 0;
} /* main */
```

Para poner de manifiesto que se trata de dos procesos que se ejecutan paralelamente, resulta conveniente que las tareas del padre y del hijo sean distintas o que se hagan con datos de entrada diferentes.

ANOTACIONES

Para ilustrar tal caso, se muestra una posible programación de la función, tarea en la que se hace una repetición de esperas. Para poder observar que la ejecución de los dos procesos puede no estar intercalada siempre de la misma manera, tanto el número de repeticiones como el tiempo de las esperas se pone en función de números aleatorios provistos por la función random(), que arranca con un valor "semilla" calculado mediante srandom() con un argumento que varíe con las distintas ejecuciones y con el tipo de proceso (padre o hijo):

```
/* ... */
void tarea( char *nombre )
{
   unsigned int contador;

   srandom( getpid() % ( nombre[0] * nombre[2]) );
   contador = random() % 11 + 1;
   while( contador > 0 ) {
     printf( "... paso %i del %s\n", contador, nombre );
     sleep( random() % 7 + 1 );
     contador = contador - 1;
   } /* while */
} /* tarea */
/* ... */
```

En el ejemplo anterior, el padre espera a un único hijo y realiza una misma tarea. Esto, evidentemente, no es lo habitual. Es mucho más común que el proceso padre se ocupe de generar un proceso hijo para cada conjunto de datos a procesar. En este caso, el programa principal se complica ligeramente y hay que seleccionar en función del valor devuelto por fork() si las instrucciones pertenecen al padre o a uno de los hijos.

Para ilustrar la codificación de tales programas, se muestra un programa que toma como argumentos una cantidad indeterminada de números naturales para los que averigua si se trata o no, de números primos. En este caso, el programa principal creará un proceso hijo para cada número natural a tratar:

ANOTACIONES

```
/* ... */
int main( int argc, char *argv[] )
{
   int                 contador;
   unsigned long int   numero, divisor;
   pid_t               proceso;
   int                 estado;

   if( argc > 1 ) {
     proceso = getpid();
     printf( "Proceso %li iniciado.\n", proceso );
     contador = 1;
     while( proceso != 0 && contador < argc ) {
       /* Creación de procesos hijo:             */
       numero = atol( argv[ contador ] );
       contador = contador + 1;
       proceso = fork();
       if( proceso == 0 ) {
         printf( "Proceso %li para %lu\n",
           getpid(),
           numero
         ); /* printf */
         divisor = es_primo( numero );
         if( divisor > 1 ) {
           printf( "%lu no es primo.\n", numero );
           printf(“Su primer divisor es %lu\n”, divisor );
         } else {
           printf( "%lu es primo.\n", numero );
         } /* if */
       } /* if */
     } /* while */
     while( proceso != 0 && contador > 0 ) {
       /* Espera de finalización de procesos hijo:*/
       wait( &estado );
       contador = contador - 1;
     } /* while */
     if( proceso !=0 ) printf( "Fin.\n");
   } else {
     printf( "Uso: %s natural_1 ... natural_N\n", argv[0] );
   } /* if */
   return 0;
} /* main */
```

Nota

El bucle de creación se interrumpe si `proceso==0` para evitar que los procesos hijo puedan crear "nietos" con los mismos datos que algunos de sus "hermanos".

Hay que tener presente que el código del programa es el mismo tanto para el proceso padre, como para el hijo.

Por otra parte, el bucle final de espera sólo debe aplicarse al padre, es decir, al proceso en el que se cumpla que la variable `proceso` sea distinta de cero. En este caso, basta con descontar del contador de procesos generados una unidad para cada espera cumplida.

Para comprobar su funcionamiento, falta diseñar la función `es_primo()`, que queda como ejercicio. Para ver el funcionamiento de tal programa de manera ejemplar, es conveniente introducir algún número primo grande junto con otros menores o no primos.

3.15.1. Comunicación entre procesos

Tal como se ha comentado, los procesos (tanto si son de un mismo programa como de programas distintos) se pueden comunicar entre sí mediante mecanismos de tuberías, colas de mensajes y variables compartidas, entre otros. Por lo tanto, estos mecanismos también se pueden aplicar en la comunicación entre programas distintos de una misma aplicación o, incluso, de aplicaciones distintas. En todo caso, siempre se trata de una comunicación poco intensa y que requiere de exclusión mutua en el acceso a los datos para evitar conflictos (aun así, no siempre se evitan todos).

Como ejemplo, se mostrará un programa que descompone en suma de potencias de divisores primos cualquier número natural dado. Para ello, dispone de un proceso de cálculo de divisores primos y otro, el padre, que los muestra a medida que se van calculando. Cada factor de la suma es un dato del tipo:

```
typedef struct factor_s {
   unsigned long int divisor;
   unsigned long int potencia;
} factor_t;
```

ANOTACIONES

La comunicación entre ambos procesos se realiza mediante una tubería.

Nota

Recordad que en la unidad anterior ya se definió *tubería*. Una tubería consiste, de hecho, en dos ficheros de flujo de bytes, uno de entrada y otro de salida, por el que se comunican dos procesos distintos.

Como se puede apreciar en el código siguiente, la función para abrir una tubería se llama `pipe()` y toma como argumento la dirección de un vector de dos enteros en donde depositará los descriptores de los ficheros de tipo *stream* que haya abierto: en la posición 0 el de salida, y en la 1, el de entrada. Después del `fork()`, ambos procesos tienen una copia de los descriptores y, por tanto, pueden acceder a los mismos ficheros tanto para entrada como para salida de datos. En este caso, el proceso hijo cerrará el fichero de entrada y el padre, el de salida; puesto que la tubería sólo comunicará los procesos en un único sentido: de hijo a padre. (Si la comunicación se realizara en ambos sentidos, sería necesario establecer un protocolo de acceso a los datos para evitar conflictos.):

```
/* ... */
int main( int argc, char *argv[] )
{
   unsigned long int  numero;
   pid_t              proceso;
   int                estado;
   int                desc_tuberia[2];

   if( argc == 2 ) {
     printf( "Divisores primos.\n" );
     numero = atol( argv[ 1 ] );
     if( pipe( desc_tuberia ) != -1 ) {
       proceso = fork();
       if( proceso == 0 ) { /* Proceso hijo:                   */
         close( desc_tuberia[0] );
         divisores_de( numero, desc_tuberia[1] );
         close( desc_tuberia[1] );
       } else { /* Proceso principal o padre:                  */
```

ANOTACIONES

```
      close( desc_tuberia[1] );
      muestra_divisores( desc_tuberia[0] );
      wait( &estado );
      close( desc_tuberia[0] );
      printf( "Fin.\n" );
     } /* if */
    } else {
     printf( "No puedo crear la tuberia!\n" );
    } /* if */
   } else {
    printf( "Uso: %s numero_natural\n", argv[0] );
   } /* if */
   return 0;
} /* main */
```

Con todo, el código de la función `muestra_divisores()` en el proceso padre podría ser como el que se muestra a continuación. En él, se emplea la función de lectura `read()`, que intenta leer un determinado número de bytes del fichero cuyo descriptor se le pasa como primer argumento. Devuelve el número de bytes efectivamente leídos y su contenido lo deposita a partir de la dirección de memoria indicada:

```
/* ... */
void muestra_divisores( int desc_entrada )
{
   size_t    nbytes;
   factor_t  factor;

   do {
    nbytes = read( desc_entrada,
      (void *)&factor,
      sizeof( factor_t )
    ); /* read */
    if( nbytes > 0 ) {
     printf( "%lu ^ %lu\n",
       factor.divisor,
       factor.potencia
     ); /* printf */
   } while( nbytes > 0 );
} /* muestra_divisores */
/* ... */
```

Para completar el ejemplo, se muestra una posible programación de la función `divisores_de()` en el proceso hijo. Esta función emplea `write()` para depositar los factores recién calculados en el fichero de salida de la tubería:

```
/* ... */
void divisores_de(
   unsigned long  int numero,
   int            desc_salida )
{
   factor_t f;

   f.divisor = 2;
   while( numero > 1 ) {
     f.potencia = 0;
     while( numero % f.divisor == 0 ) {
       f.potencia = f.potencia + 1;
       numero = numero / f.divisor;
     } /* while */
     if( f.potencia > 0 ) {
       write( desc_salida, (void *)&f, sizeof( factor_t ) );
     } /* if */
     f.divisor = f.divisor + 1;
   } /* while */
} /* divisores_de */
/* ... */
```

Con este ejemplo se ha mostrado una de las posibles formas de comunicación entre procesos. En general, cada mecanismo de comunicación tiene unos usos preferentes

Nota

Las tuberías son adecuadas para el paso de una cantidad relativamente alta de datos entre procesos, mientras que las colas de mensajes se adaptan mejor a procesos que se comunican poco frecuentemente o de forma irregular.

ANOTACIONES

En todo caso, hay que tener presente que repartir las tareas de un programa en varios procesos supondrá un cierto incremento de la complejidad por la necesaria introducción de mecanismos de comunicación entre ellos. Así pues, es importante valorar los beneficios que tal división pueda aportar al desarrollo del programa correspondiente.

3.16. Resumen

Los algoritmos que se emplean para procesar la información pueden ser más o menos complejos según la representación que se escoja para la misma. Como consecuencia, la eficiencia de la programación está directamente relacionada con las estructuras de datos que se empleen en ésta.

Por este motivo se han introducido las estructuras dinámicas de datos, que permiten, entre otras cosas, aprovechar mejor la memoria y cambiar la relación entre ellos como parte del procesado de la información.

Las estructuras de datos dinámicas son, pues, aquéllas en las que el número de datos puede variar durante la ejecución del programa y cuyas relaciones, evidentemente, pueden cambiar. Para ello, se apoyan en la creación y destrucción de variables dinámicas y en los mecanismos para acceder a ellas. Fundamentalmente, el acceso a tales variables se debe hacer mediante apuntadores, puesto que las variables dinámicas no disponen de nombres con los que identificarlas.

Se ha visto también un ejemplo común de estructuras de datos dinámicas como las cadenas de caracteres y las listas de nodos. En particular, para este último caso se ha revisado no sólo la posible programación de las funciones de gestión de los nodos en una lista, sino también una forma especial de tratamiento de las mismas en la que se emplean como representaciones de colas.

Dado lo habitual del empleo de muchas de estas funciones para estructuras de datos dinámicas comunes, resulta conveniente agruparlas en archivos de ficheros objeto: las bibliotecas de funciones. De

ANOTACIONES

esta manera, es posible emplear las mismas funciones en programas diversos sin preocuparse de su programación. Aun así, es necesario incluir los ficheros de cabecera para indicar al compilador la forma de invocar a tales funciones. Con todo, se repasa el mecanismo de creación de bibliotecas de funciones y, además, se introduce el uso de la utilidad *make* para la generación de ejecutables que resultan de la compilación de diversas unidades del mismo programa y de los archivos de biblioteca requeridos.

Por otra parte, también se ha visto cómo la relación entre los distintos tipos de datos abstractos de un programa facilitan la programación modular. De hecho, tales tipos se clasifican según niveles de abstracción o, según se mire, de dependencia de otros tipos de datos. Así pues, el nivel más bajo de abstracción lo ostentan los tipos de datos abstractos que se definen en términos de tipos de datos primitivos.

De esta manera, el programa principal será aquel que opere con los tipos de datos de mayor nivel de abstracción. El resto de módulos del programa serán los que provean al programa principal de las funciones necesarias para realizar tales operaciones.

Por lo tanto, el diseño descendente de algoritmos, basado en la jerarquía que se establece entre los distintos tipos de datos que emplean, es una técnica con la que se obtiene una programación modular eficiente.

En la práctica, cada tipo de datos abstracto deberá acompañarse de las funciones para operaciones elementales como creación, acceso a datos, copia, duplicado y destrucción de las variables dinámicas correspondientes. Más aun, deberá estar contenida en una unidad de compilación independiente, junto con el fichero de cabecera adecuado.

Finalmente, en el último capítulo, se ha insistido en la organización del código, no tanto con relación a la información que debe procesar, sino más en relación con la forma de hacerlo. En este sentido, resulta conveniente aprovechar al máximo las facilidades que nos ofrece el lenguaje de programación C para utilizar las rutinas de servicio del sistema operativo.

ANOTACIONES

Cuando la información a tratar deba ser procesada por otro programa, es posible ejecutarlos desde el flujo de ejecución de instrucciones del que se está ejecutando. En este caso, sin embargo, la comunicación entre el programa llamado y el llamador es mínima. Como consecuencia, debe ser el mismo programa llamado el que obtenga la mayor parte de la información a tratar y el que genere el resultado.

Se ha tratado también de la posibilidad de dividir el flujo de ejecución de instrucciones en varios flujos diferentes que se ejecutan concurrentemente. De esta manera, es posible especializar cada flujo en un determinado aspecto del tratamiento de la información o, en otros casos, realizar el mismo tratamiento sobre partes distintas de la información.

Los flujos de ejecución de instrucciones se pueden dividir en hilos (*threads*) o procesos. A los primeros también se les denomina procesos ligeros, pues son procesos que comparten el mismo contexto (entorno) de ejecución. El tipo de tratamiento de la información será el que determine qué forma de división es la mejor. Como norma se puede tomar la del grado de compartimiento de la información: si es alto, entonces es mejor un hilo, y si es bajo, un proceso (entre ellos, no obstante, hay diversos mecanismos de comunicación según el grado particular de relación que tengan).

En todo caso, parte del contenido de esta unidad se verá de nuevo en las próximas pues, tanto C++ como Java, facilitan la programación con tipos de datos abstractos, el diseño modular y la distribución de la ejecución en diversos flujos de instrucciones.

3.17. Ejercicios de autoevaluación

1) Haced un buscador de palabras en ficheros, de forma similar al último ejercicio de la unidad anterior. El programa deberá pedir el nombre del fichero y la palabra a buscar. En este caso, la función principal deberá ser la siguiente:

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

typedef enum bool_e { FALSE = 0, TRUE = 1 } bool;
typedef char *palabra_t;
```

ANOTACIONES

```
palabra_t siguiente_palabra(
   char *frase,
   unsigned int inicio
) { /* ... */ }
int main( void )
{
   FILE          *entrada;
   char          nombre[BUFSIZ];
   palabra_t     palabra, palabra2;
   unsigned int numlin, pos;

   printf( "Busca palabras.\n" );
   printf( "Fichero: ");
   gets( nombre );
   entrada = fopen( nombre, "rt" );
   if( entrada != NULL ) {
     printf( "Palabra: ");
     gets( nombre );
     palabra = siguiente_palabra( nombre, 0 );
     printf( "Buscando %s en fichero...\n", palabra );
     numlin = 1;
     while( fgets( nombre, BUFSIZ-1, entrada ) != NULL ) {
       numlin = numlin + 1;
       pos = 0;
       palabra2 = siguiente_palabra( nombre, pos );
       while( palabra2 != NULL ) {
         if( !strcmp( palabra, palabra2 ) ) {
           printf( "... línea %lu\n", numlin );
         } /* if */
         pos = pos + strlen( palabra2 );
         free( palabra2 );
         palabra2 = siguiente_palabra( nombre, pos );
       } /* while */
     } /* while */
     free( palabra );
     fclose( entrada );
     printf( "Fin.\n" );
   } else {
     printf( "¡No puedo abrir %s!\n", nombre );
   } /* if */
   return 0;
} /* main */
```

ANOTACIONES

Se debe de programar, pues, la función `siguiente_palabra()`.

2) Componed, a partir de las funciones provistas en el apartado 3.5.2, la función para eliminar el elemento enésimo de una lista de enteros. El programa principal deberá ser el siguiente:

```
int main( void )
{
   lista_t       lista;
   char          opcion;
   int           dato;
   unsigned int  n;

   printf( "Gestor de listas de enteros.\n" );
   lista = NULL;
   do {
     printf(
       "[I]nsertar, [E]liminar, [M]ostrar o [S]alir? "
     ); /* printf */
     do opcion = getchar(); while( isspace(opcion) );
     opcion = toupper( opcion );
     switch( opcion ) {
       case 'I':
        printf( "Dato =? ");
        scanf( "%i", &dato );
        printf( "Posicion =? " );
        scanf( "%u", &n );
        if( !inserta_enesimo_lista( &lista, n, dato ) ) {
          printf( "No se insertó.\n" );
        } /* if */
        break;
       case 'E':
        printf( "Posicion =? " );
        scanf( "%u", &n );
        if( elimina_enesimo_lista( &lista, n, &dato ) ) {
          printf( "Dato = %i\n", dato );
        } else {
        printf( "No se eliminó.\n" );
        } /* if */
        break;
       case 'M':
        muestra_lista( lista );
        break;
     } /* switch */
   } while( opcion != 'S' );
   while( lista != NULL ) {
     elimina_enesimo_lista( &lista, 0, &dato );
   } /* while */
   printf( "Fin.\n" );
   return 0;
} /* main */
```

También hay que programar la función `muestra_lista()` para poder ver su contenido.

3) Haced un programa que permita insertar y eliminar elementos de una cola de enteros. Las funciones que deben emplearse se encuentran en el apartado referente a colas del apartado 3.5.2. Por lo tanto, sólo cabe desarrollar la función principal de dicho programa, que puede inspirarse en la mostrada en el ejercicio anterior.

4) Programad el algoritmo de ordenación por selección visto en el apartado 3.6 para clasificar un fichero de texto en el que cada línea tenga el formato siguiente:

```
DNI nota '\n'
```

Por tanto, los elementos serán del mismo tipo de datos que el visto en el ejemplo. El programa principal será:

```
int main( void )
{
  FILE        *entrada;
  char        nombre[ BUFSIZ ];
  lista_t     pendientes, ordenados;
  ref_nodo_t  refnodo;
  elemento_t  elemento;

  printf( "Ordena listado de nombres.\n" );
  printf( "Fichero =? " ); gets( nombre );
  entrada = fopen( nombre, "rt" );
  if( entrada != NULL ) {
   inicializa_lista( &pendientes );
   while( fgets( nombre, BUFSIZ-1, entrada ) != NULL ) {
    elemento = lee_elemento( nombre );
    pon_al_final_de_lista( &pendientes, elemento );
   } /* if */
   inicializa_lista( &ordenados );
   while( ! esta_vacia_lista( pendientes ) ) {
    elemento = extrae_minimo_de_lista( &pendientes );
    pon_al_final_de_lista( &ordenados, elemento );
   } /* while */
   printf( "Lista ordenada por DNI:\n" );
   principio_de_lista( &ordenados );
```

ANOTACIONES

```
    while( !es_final_de_lista( ordenados ) ) {
      refnodo = ref_nodo_de_lista( ordenados );
      elemento = elemento_en_ref_nodo(ordenados, refnodo);
      muestra_elemento( elemento );
      avanza_posicion_en_lista( &ordenados );
    } /* while */
    printf( "Fin.\n" );
  } else {
    printf( "¡No puedo abrir %s!\n", nombre );
  } /* if */
  return 0;
} /* main */
```

Por lo tanto, también será necesario programar las funciones siguientes:

- `elemento_t crea_elemento( unsigned int DNI, float nota );`
- `elemento_t lee_elemento( char *frase );`
- `void muestra_elemento( elemento_t elemento );`

> ***Nota***
>
> En este caso, los elementos de la lista no son destruidos antes de finalizar la ejecución del programa porque resulta más simple y, además, se sabe que el espacio de memoria que ocupa se liberará en su totalidad. Aun así, no deja de ser una mala práctica de la programación y, por lo tanto, se propone como ejercicio libre, la incorporación de una función para eliminar las variables dinámicas correspondientes a cada elemento antes de acabar la ejecución del programa.

5) Implementad el programa anterior en tres unidades de compilación distintas: una para el programa principal, que también puede dividirse en funciones más manejables, una para los elementos y otra para las listas, que puede transformarse en biblioteca.

6) Haced un programa que acepte como argumento un NIF y valide la letra. Para ello, tómese como referencia el ejercicio de autoevaluación número 7 de la unidad anterior.

ANOTACIONES

7) Transformad la utilidad de búsqueda de palabras en ficheros de texto del primer ejercicio para que tome como argumentos en la línea de comandos tanto la palabra a buscar como el nombre del fichero de texto en el que tenga que realizar la búsqueda.

8) Cread un comando que muestre el contenido del directorio como si de un `ls -als | more` se tratara. Para ello, hay que hacer un programa que ejecute tal mandato y devuelva el código de error correspondiente.

9) Programad un "despertador" para que muestre un aviso cada cierto tiempo o en una hora determinada. Para ello, tomar como referencia el programa ejemplo visto en la sección de procesos permanentes.

El programa tendrá como argumento la hora y minutos en que se debe mostrar el aviso indicado, que será el segundo argumento. Si la hora y minutos se precede con el signo `'+'`, entonces se tratará como en el ejemplo, es decir, como el lapso de tiempo que debe pasar antes de mostrar el aviso.

Hay que tener presente que la lectura del primer valor del primer argumento puede hacerse de igual forma que en el programa "avisador" del tema, puesto que el signo `'+'` se interpreta como indicador de signo del mismo número. Eso sí, hay que leer específicamente `argv[1][0]` para saber si el usuario ha introducido el signo o no.

Para saber la hora actual, es necesario emplear las funciones de biblioteca estándar de tiempo, que se encuentran declaradas en `time.h`, y cuyo uso se muestra en el programa siguiente:

```
/* Fichero: horamin.c                                    */
#include <stdio.h>
#include <time.h>
int main( void )
{
   time_ttiempo;
   struct tm *tiempo_desc;
   time( &tiempo );
   tiempo_desc = localtime( &tiempo );
   printf( "Son las %2d y %2d minutos.\n",
```

ANOTACIONES

```
      tiempo_desc->tm_hour,
      tiempo_desc->tm_min
    ); /* printf */
    return 0;
} /* main */
```

10) Probad los programas de detección de números primos mediante hilos y procesos. Para ello, es necesario definir la función `es_primo()` de manera adecuada. El siguiente programa es una muestra de tal función, que aprovecha el hecho de que ningún divisor entero será mayor que la raíz cuadrada del número (se aproxima por la potencia de 2 más parecida):

```
/* Fichero: es_primo.c                                          */
#include <stdio.h>
#include "bool.h"

int main( int argc, char *argv[] )
{
  unsigned long int numero, maximo, divisor;
  bool              primo;

  if( argc == 2 ) {
   numero = atol( argv[1] );
   primo = numero < 4; /* 0 ... 3, considerados primos.       */
   if( !primo ) {
    divisor = 2;
    primo = numero % divisor != 0;
    if( primo ) {
     maximo = numero / 2;
     while( maximo*maximo > numero ) maximo = maximo/2;
     maximo = maximo * 2;
     divisor = 1;
     while( primo && (divisor < maximo) ) {
      divisor = divisor + 2;
      primo = numero % divisor != 0;
     } /* while */
    } /* if */
   } /* if */
   printf( "... %s primo.\n", primo? "es" : "no es" );
  } else {
   printf( "Uso: %s número_natural\n", argv[0] );
  } /* if */
  return 0;
} /* main */
```

ANOTACIONES

3.17.1. Solucionario

1) Como ya tenemos el programa principal, es suficiente con mostrar la función `siguiente_palabra`:

```
palabra_t siguiente_palabra(
   char          *frase,
   unsigned int inicio)
{
   unsigned int fin, longitud;
   palabra_t    palabra;

   while( frase[inicio]!='\0' && !isalnum(frase[inicio]) ) {
     inicio = inicio + 1;
   } /* while */
   fin = inicio;
   while( frase[fin]!='\0' && isalnum( frase[fin] ) ) {
     fin = fin + 1;
   } /* while */
   longitud = fin - inicio;
   if( longitud > 0 ) {
     palabra = (palabra_t)malloc((longitud+1)*sizeof(char));
     if( palabra != NULL ) {
       strncpy( palabra, &(frase[inicio]), longitud );
       palabra[longitud] = '\0';
     } /* if */
   } else {
     palabra = NULL;
   } /* if */
   return palabra;
} /* siguiente_palabra */
```

2)

```
bool elimina_enesimo_lista(
   lista_t *listaref, /* Apuntador a referencia 1er nodo.*/
   unsigned int n,     /* Posición de la eliminación.        */
   int      *datoref)  /* Referencia del dato eliminado.     */
{                      /* Devuelve FALSE si no se puede.     */
   nodo_t *p, *q, *t;
   bool retval;
```

ANOTACIONES

```
   enesimo_pq_nodo( *listaref, n, &p, &q );
   if( q != NULL ) {
     *datoref = destruye_nodo( listaref, p, q );
     retval = TRUE;
   } else {
     retval = FALSE;
   } /* if */
   return retval;
} /* elimina_enesimo_lista */

void muestra_lista( lista_t lista )
{
   nodo_t *q;
   if( lista != NULL ) {
     q = lista;
     printf( "Lista = " );
     while( q != NULL ) {
       printf( "%i ", q->dato );
       q = q->siguiente;
     } /* while */
     printf( "\n" );
   } else {
     printf( "Lista vacía.\n" );
   } /* if */
} /* muestra_lista */
```

3)

```
int main( void )
{
   cola_t   cola;
   char     opcion;
   int      dato;

   printf( "Gestor de colas de enteros.\n" );
   cola.primero = NULL; cola.ultimo = NULL;
   do {
     printf(
       "[E]ncolar, [D]esencolar, [M]ostrar o [S]alir? "
     ); /* printf */
     do opcion = getchar(); while( isspace(opcion) );
     opcion = toupper( opcion );
```

ANOTACIONES

```
    switch( opcion ) {
      case 'E':
        printf( "Dato =? ");
        scanf( "%i", &dato );
        if( !encola( &cola, dato ) ) {
          printf( "No se insertó.\n" );
        } /* if */
        break;
      case 'D':
        if( desencola( &cola, &dato ) ) {
          printf( "Dato = %i\n", dato );
        } else {
          printf( "No se eliminó.\n" );
        } /* if */
        break;
      case 'M':
        muestra_lista( cola.primero );
        break;
    } /* switch */
  } while( opcion != 'S' );
  while( desencola( &cola, &dato ) ) { ; }
  printf( "Fin.\n" );
  return 0;
} /* main */
```

4)

```
elemento_t crea_elemento( unsigned int DNI, float nota )
{
  elemento_t elemento;
  elemento = (elemento_t)malloc( sizeof( dato_t ) );
  if( elemento != NULL ) {
    elemento->DNI = DNI;
    elemento->nota = nota;
  } /* if */
  return elemento;
} /* crea_elemento */

elemento_t lee_elemento( char *frase )
{
  unsigned int DNI;
  double       nota;
  int          leido_ok;
```

ANOTACIONES

```
   elemento_t   elemento;
   leido_ok = sscanf( frase, "%u%lf", &DNI, &nota );
   if( leido_ok == 2 ) {
    elemento = crea_elemento( DNI, nota );
   } else {
    elemento = NULL;
   } /* if */
   return elemento;
} /* lee_elemento */
void muestra_elemento( elemento_t elemento )
{
   printf( "%10u %.2f\n", elemento->DNI, elemento->nota );
} /* muestra_elemento */
```

5) Véase el apartado 3.6.2.

6)

```
char letra_de( unsigned int DNI )
{
   char codigo[] = "TRWAGMYFPDXBNJZSQVHLCKE" ;
   return codigo[ DNI % 23 ];
} /* letra_de */

int main( int argc, char *argv[] )
{
   unsigned int DNI;
   char         letra;
   int          codigo_error;

   if( argc == 2 ) {
    sscanf( argv[1], "%u", &DNI );
    letra = argv[1][ strlen(argv[1])-1 ];
    letra = toupper( letra );
    if( letra == letra_de( DNI ) ) {
     printf( "NIF válido.\n" );
     codigo_error = 0;
    } else {
     printf( "¡Letra %c no válida!\n", letra );
     codigo_error = -1;
    } /* if */
   } else {
    printf( "Uso: %s DNI-letra\n", argv[0] );
    codigo_error = 1;
   } /* if */
   return codigo_error;
} /* main */
```

ANOTACIONES

7)

```
int main( int argc, char *argv[] )
{
   FILE          *entrada;
   char          nombre[BUFSIZ];
   palabra_t     palabra, palabra2;
   unsigned int numlin, pos;
   int           codigo_error;
   if( argc == 3 ) {
     palabra = siguiente_palabra( argv[1], 0 );
     if( palabra != NULL ) {
       entrada = fopen( argv[2], "rt" );
       if( entrada != NULL ) {
         /* (Véase: Enunciado del ejercicio 1.)                    */
       } else {
         printf( "¡No puedo abrir %s!\n", argv[2] );
         codigo_error = -1;
       } /* if */
     } else {
       printf( "¡Palabra %s inválida!\n", argv[1] );
       codigo_error = -1;
     } /* if */
   } else {
     printf( "Uso: %s palabra fichero\n", argv[0] );
     codigo_error = 0;
   } /* if */
   return codigo_error;
} /* main */
```

8)

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

   codigo_error = system( "ls -als | more" );
   return codigo_error;
} /* main */
```

ANOTACIONES

9)

```
int main( int argc, char *argv[] )
{
   unsigned int horas;
   unsigned int minutos;
   unsigned int segundos;
   char         *aviso, *separador;
   time_t       tiempo;
   struct tm    *tiempo_desc;

   if( argc == 3 ) {
    separador = strchr( argv[1], ':' );
    if( separador != NULL ) {
     horas = atoi( argv[1] ) % 24;
     minutos = atoi( separador+1 ) % 60;
    } else {
     horas = 0;
     minutos = atoi( argv[1] ) % 60;
    } /* if */
    if( argv[1][0]!='+' ) {
     time( &tiempo );
     tiempo_desc = localtime( &tiempo );
     if( minutos < tiempo_desc->tm_min ) {
      minutos = minutos + 60;
      horas = horas - 1;
     } /* if */
     if( horas < tiempo_desc->tm_hour ) {
      horas = horas + 24;
     } /* if */
     minutos = minutos - tiempo_desc->tm_min;
     horas = horas - tiempo_desc->tm_hour;
    } /* if */
    segundos = (horas*60 + minutos) * 60;
    aviso = argv[2];
    if( daemon( FALSE, TRUE ) ) {

     printf( "No puede instalarse el avisador :-(\n" );

    } else {
     printf( "Alarma dentro de %i horas y %i minutos.\n",
```

ANOTACIONES

```
      horas, minutos
     ); /* printf */
     printf( "Haz $ kill %li para apagarla.\n",
      getpid()
     ); /* printf */
    } /* if */
    sleep( segundos );
    printf( "%s\007\n", aviso );
    printf( "Alarma apagada.\n" );
  } else {
    printf( "Uso: %s [+]HH:MM \"aviso\"\n", argv[0] );
    printf( "(con + es respecto de la hora actual)\n" );
    printf( "(sin + es la hora del dia)\n" );
  } /* if */
  return 0;
} /* main */
```

10) Se trata de repetir el código dado dentro de una función que tenga la cabecera adecuada para cada caso.

ANOTACIONES

4. Programación orientada a objetos en C++

4.1. Introducción

Hasta el momento se ha estudiado cómo abordar un problema utilizando los paradigmas de programación modular y el diseño descendente de algoritmos. Con ellos se consigue afrontar un problema complejo mediante la descomposición en problemas más simples, reduciendo progresivamente su nivel de abstracción, hasta obtener un nivel de detalle manejable. Al final, el problema se reduce a estructuras de datos y funciones o procedimientos.

Para trabajar de forma eficiente, las buenas prácticas de programación nos aconsejan agrupar los conjuntos de rutinas y estructuras relacionados entre sí en unidades de compilación, que luego serían enlazados con el archivo principal. Con ello, se consigue lo siguiente:

- Localizar con rapidez el código fuente que realiza una tarea determinada y limitar el impacto de las modificaciones a unos archivos determinados.

- Mejorar la legibilidad y comprensión del código fuente en conjunto al no mezclarse entre sí cada una de las partes.

No obstante, esta recomendable organización de los documentos del proyecto sólo proporciona una separación de los diferentes archivos y, por otro lado, no refleja la estrecha relación que suele haber entre los datos y las funciones.

En la realidad muchas veces se desea implementar entidades de forma que se cumplan unas propiedades generales: conocer las entradas que precisan, una idea general de su funcionamiento y las salidas que generan. Generalmente, los detalles concretos de la implemen-

ANOTACIONES

tación no son importantes: seguramente habrá decenas de formas posibles de hacerlo.

Se puede poner como ejemplo un televisor. Sus propiedades pueden ser la marca, modelo, medidas, número de canales; y las acciones a implementar serían encender o apagar el televisor, cambiar de canal, sintonizar un nuevo canal, etc. Cuando utilizamos un aparato de televisión, lo vemos como una caja cerrada, con sus propiedades y sus conexiones. No nos interesa en absoluto sus mecanismos internos, sólo deseamos que actúe cuando apretamos el botón adecuado. Además, éste se puede utilizar en varias localizaciones y siempre tendrá la misma función. Por otro lado, si se estropea puede ser sustituido por otro y sus características básicas (tener una marca, encender, apagar, cambiar de canal, etc.) continúan siendo las mismas independientemente de que el nuevo aparato sea más moderno. El televisor es tratado como un objeto por sí mismo y no como un conjunto de componentes.

Este mismo principio aplicado a la programación se denomina **encapsulamiento**. El encapsulamiento consiste en implementar un elemento (cuyos detalles se verán más adelante) que actuará como una "caja negra", donde se especificarán unas entradas, una idea general de su funcionamiento y unas salidas. De esta forma se facilita lo siguiente:

- La reutilización de código. Si ya se dispone de una "caja negra" que tenga unas características coincidentes con las necesidades definidas, se podrá incorporar sin interferir con el resto del proyecto.

- El mantenimiento del código. Se pueden realizar modificaciones sin que afecten al proyecto en conjunto, mientras se continúen cumpliendo las especificaciones de dicha "caja negra".

A cada uno de estos elementos los llamaremos *objetos* (en referencia a los objetos de la vida real a los que representa). Al trabajar con objetos, lo que supone un nivel de abstracción mayor, se afronta el diseño de una aplicación no pensando en la secuencia de instrucciones a realizar, sino en la definición de los objetos que intervienen y las relaciones que se establecen entre ellos.

ANOTACIONES

En esta unidad estudiaremos un nuevo lenguaje que nos permite implementar esta nueva visión que supone el paradigma de la programación orientada a objetos: C++.

Este nuevo lenguaje se basa en el lenguaje C al que se le dota de nuevas características. Por este motivo, en primer lugar, se establece una comparación entre ambos lenguajes en los ámbitos comunes que nos permite un aprendizaje rápido de sus bases. A continuación, se nos presenta el nuevo paradigma y las herramientas que el nuevo lenguaje proporciona para la implementación de los objetos y sus relaciones. Finalmente, se muestra cómo este cambio de filosofía afecta al diseño de aplicaciones.

En esta unidad se pretende que los lectores, partiendo de sus conocimientos del lenguaje C, puedan conocer los principios básicos de la programación orientada a objetos utilizando el lenguaje C++ y del diseño de aplicaciones siguiendo este paradigma. En concreto, al finalizar el estudio de esta unidad, el lector habrá alcanzado los objetivos siguientes:

1) Conocer las diferencias principales entre C y C++, inicialmente sin explorar aún la tecnología de objetos.

2) Comprender el paradigma de la programación orientada a objetos.

3) Saber implementar clases y objetos en C++.

4) Conocer las propiedades principales de los objetos: la herencia, la homonimia y el polimorfismo.

5) Poder diseñar una aplicación simple en C++ aplicando los principios del diseño orientado a objetos.

4.2. De C a C++

4.2.1. El primer programa en C++

Elegir el entorno de programación C++ para la implementación del nuevo paradigma de la programación orientada a objetos supone una gran ventaja por las numerosas similitudes existentes con el len-

ANOTACIONES

guaje C. No obstante, puede convertirse en una limitación si el programador no explora las características adicionales que nos proporciona el nuevo lenguaje y que aportan una serie de mejoras bastante interesantes.

Tradicionalmente, en el mundo de la programación, la primera toma de contacto con un lenguaje de programación se hace a partir del clásico mensaje de "hola mundo" y, en este caso, no haremos una excepción.

Por tanto, en primer lugar, escribid en vuestro editor el siguiente texto y guardadlo con el nombre *ejemplo01.cpp*:

Nota

La extensión ".cpp" indica al compilador que el tipo de código fuente es C++.

```
#include <iostream>
int main()
{
   cout << "hola mundo \n" ;
   return 0;
}
```

Comparando este programa con el primer programa en C, observamos que la estructura es similar. De hecho, como se ha comentado, el C++ se puede ver como una evolución del C para implementar la programación orientada a objetos y, como tal, mantiene la compatibilidad en un porcentaje muy alto del lenguaje.

La única diferencia observable la encontramos en la forma de gestionar la salida que se hace a través de un objeto llamado `cout`. La naturaleza de los objetos y de las clases se estudiará en profundidad más adelante, pero, de momento, podremos hacernos una idea considerando la clase como un tipo de datos nuevo que incluye atributos y funciones asociadas, y el objeto como una variable de dicho tipo de datos.

La definición del objeto `cout` se halla dentro de la librería <iostream>, que se incluye en la primera línea del código fuente También llama la atención la forma de uso, mediante el direccionamiento (con el símbolo <<) del texto de "hola mundo" sobre el objeto `cout`, que genera la salida de este mensaje en la pantalla.

ANOTACIONES

Como el tema del tratamiento de las funciones de entrada/salida es una de las principales novedades del C++, comenzaremos por él para desarrollar las diferencias entre un lenguaje y el otro.

4.2.2. Entrada y salida de datos

Aunque ni el lenguaje C ni el C++ definen las operaciones de entrada/salida dentro del lenguaje en sí, es evidente que es indispensable su tratamiento para el funcionamiento de los programas. Las operaciones que permiten la comunicación entre los usuarios y los programas se encuentran en bibliotecas que provee el compilador. De este modo, podremos trasladar un código fuente escrito para un entorno Sun a nuestro PC en casa y así obtendremos una independencia de la plataforma. Al menos, en teoría.

Tal como se ha comentado en unidades anteriores, el funcionamiento de la entrada/salida en C se produce a través de librerías de funciones, la más importante de las cuales es la <stdio.h> o <cstdio> (entrada/salida estándar). Dichas funciones (`printf`, `scanf`, `fprint`, `fscanf`, etc.) continúan siendo operativas en C++, aunque no se recomienda su uso al no aprovechar los beneficios que proporciona el nuevo entorno de programación.

Nota

Ambas formas de expresar el nombre de la librería <xxxx.h> o <cxxxx> son correctas, aunque la segunda se considera actualmente la forma estándar de incorporar librerías C dentro del lenguaje C++ y la única recomendada para su uso en nuevas aplicaciones.

C++, al igual que C, entiende la comunicación de datos entre el programa y la pantalla como un flujo de datos: el programa va enviando datos y la pantalla los va recibiendo y mostrando. De la misma manera se entiende la comunicación entre el teclado (u otros dispositivos de entrada) y el programa.

ANOTACIONES

Para gestionar estos flujos de datos, C++ incluye la clase iostream, que crea e inicializa cuatro objetos:

- `cin`. Maneja flujos de entrada de datos.
- `cout`. Maneja flujos de salida de datos.
- `cerr`. Maneja la salida hacia el dispositivo de error estándar, la pantalla.
- `clog`. Maneja los mensajes de error.

A continuación, se presentan algunos ejemplos simples de su uso.

```
#include <iostream>
int main()
{
   int numero;
   cout << "Escribe un número: ";
   cin >> número;
}
```

En este bloque de código se observa lo siguiente:

- La declaración de una variable entera con la que se desea trabajar.
- El texto "Escribe un número" (que podemos considerar como un flujo de datos literal) que deseamos enviar a nuestro dispositivo de salida.

Para conseguir nuestro objetivo, se direcciona el texto hacia el objeto `cout` mediante el operador >>. El resultado será que el mensaje saldrá por pantalla.

- Una variable donde se desea guardar la entrada de teclado. Otra vez el funcionamiento deseado consistirá en direccionar el flujo de entrada recibido en el teclado (representado/gestionado por el objeto `cin`) sobre dicha variable.

La primera sorpresa para los programadores en C, acostumbrados al `printf` y al `scanf`, es que no se le indica en la instrucción el

ANOTACIONES

formato de los datos que se desea imprimir o recibir. De hecho, ésta es una de las principales ventajas de C++: el compilador reconoce el tipo de datos de las variables y trata el flujo de datos de forma consecuente. Por tanto, simplificando un poco, se podría considerar que los objetos `cin` y `cout` se adaptan al tipo de datos. Esta característica nos permitirá adaptar los objetos `cin` y `cout` para el tratamiento de nuevos tipos de datos (por ejemplo, `structs`), cosa impensable con el sistema anterior.

Si se desea mostrar o recoger diversas variables, simplemente, se encadenan flujos de datos:

```
#include <iostream>
int main()
{
   int i, j, k;
   cout << "Introducir tres números";
   cin >> i >> j >> k;
   cout << "Los números son: "
   cout << i << ", " << j << " y " << k;
}
```

En la última línea se ve cómo en primer lugar se envía al `cout` el flujo de datos correspondiente al texto "Los números son: "; después, el flujo de datos correspondiente a la variable i; posteriormente, el texto literal " , ", y así hasta el final.

En el caso de la entrada de datos por teclado, `cin` leerá caracteres hasta la introducción de un carácter de salto de línea (return o "\n"). Después, irá extrayendo del flujo de datos introducido caracteres hasta encontrar el primer espacio y dejará el resultado en la variable i. El resultado de esta operación será también un flujo de datos (sin el primer número que ya ha sido extraído) que recibirá el mismo tratamiento: ir extrayendo caracteres del flujo de datos hasta el siguiente separador para enviarlo a la siguiente variable. El proceso se repetirá hasta leer las tres variables.

Por tanto, la línea de lectura se podría haber escrito de la siguiente manera y hubiera sido equivalente, pero menos clara:

```
( ( ( cin >> i ) >> j ) >> k )
```

ANOTACIONES

Si se desea mostrar la variable en un formato determinado, se debe enviar un manipulador del objeto que le indique el formato deseado. En el siguiente ejemplo, se podrá observar su mecánica de funcionamiento:

```
#include <iostream>
#include <iomanip>
// Se debe incluir para la definición de los
// manipuladores de objeto cout con parámetros

int main()
{
   int i = 5;
   float j = 4.1234;

   cout << setw(4) << i << endl;
   //muestra i con anchura de 4 car.
   cout << setprecision(3) << j << endl;
   // muestra j con 3 decimales
}
```

Hay muchas otras posibilidades de formato, pero no es el objetivo de este curso. Esta información adicional está disponible en la ayuda del compilador.

4.2.3. Utilizando C++ como C

Como se ha comentado, el lenguaje C++ nació como una evolución del C, por lo que, para los programadores en C, es bastante simple adaptarse al nuevo entorno. No obstante, además de introducir todo el tratamiento para la programación orientada a objetos, C++ aporta algunas mejoras respecto a la programación clásica que es interesante conocer y que posteriormente, en la programación orientada a objetos, adquieren toda su dimensión.

A continuación, procederemos a analizar diferentes aspectos del lenguaje.

4.2.4. Las instrucciones básicas

En este aspecto, C++ se mantiene fiel al lenguaje C: las instrucciones mantienen su aspecto general (finalizadas en punto y coma, los bloques de código entre llaves, etc.) y las instrucciones básicas de control de flujo, tanto las de selección como las iterativas, conservan su sintaxis (if, `switch`, `for`, `while`, `do` ... `while`). Estas características garantizan una aproximación rápida al nuevo lenguaje.

Dentro de las instrucciones básicas, podríamos incluir las de entrada/salida. En este caso, sí que se presentan diferencias significativas entre C y C++, diferencias que ya hemos comentado en el apartado anterior.

Además, es importante resaltar que se ha incluido una nueva forma de añadir comentarios dentro del código fuente para contribuir a mejorar la lectura y el mantenimiento del mismo. Se conserva la forma clásica de los comentarios en C como el texto incluido entre las secuencias de caracteres /* (inicio de comentario) y */ (fin de comentario), pero se añade una nueva forma: la secuencia // que nos permite un comentario hasta final de línea.

Ejemplo

```
/*
Este texto está comentado utilizando la forma clásica de C.
Puede contener la cantidad de líneas que se desee.
*/

//
// Este texto utiliza el nuevo formato de comentarios
// hasta final de línea que incorpora C++
//
```

4.2.5. Los tipos de datos

Los tipos de datos fundamentales de C (`char`, `int`, `long int`, `float` y `double`) se conservan en C++, y se incorpora el nuevo tipo `bool` (tipo booleano o lógico), que puede adquirir dos valores posibles: falso (false) o verdadero (true), ahora definidos dentro del lenguaje.

ANOTACIONES

```
// ...
{
   int i = 0, num;
   bool continuar;

   continuar = true;
   do
   {
     i++;
     cout<<"Para finalizar este bucle que ha pasado";
   cout << i << "veces, teclea un 0 ";
     cin >> num;
     if (num == 0) continuar = false;
   } while (continuar);
}
```

Aunque el uso de variables de tipo lógico o booleano ya era común en C, utilizando como soporte los números enteros (el 0 como valor falso y cualquier otro valor como verdadero), la nueva implementación simplifica su uso y ayuda a reducir errores. Además, el nuevo tipo de datos sólo ocupa un byte de memoria, en lugar de los dos bytes que utilizaba cuando se simulaba con el tipo `int`.

Por otro lado, hay que destacar las novedades respecto de los tipos estructurados (`struct`, `enum` o `union`). En C++ pasan a ser considerados descriptores de tipos de datos completos, evitando la necesidad del uso de la instrucción `typedef` para definir nuevos tipos de datos.

En el ejemplo que sigue, se puede comprobar que la parte de la definición del nuevo tipo no varía:

```
struct fecha {
   int dia;
   int mes;
   int anyo;
};
enum diasSemana {LUNES, MARTES, MIERCOLES, JUEVES,
                 VIERNES, SABADO, DOMINGO};
```

ANOTACIONES

Lo que se simplifica es la declaración de una variable de dicho tipo, puesto que no se tienen que volver a utilizar los términos struct, enum o union, o la definición de nuevos tipos mediante la instrucción typedef:

```
fecha aniversario;
diasSemana festivo;
```

Por otro lado, la referencia a los datos tampoco varía.

```
// ...
aniversario.dia = 2;
aniversario.mes = 6;
aniversario.anyo = 2001;
festivo = LUNES;
```

En el caso de la declaración de variables tipo enum se cumplen dos funciones:

– Declarar diasSemana como un tipo nuevo.

– Hacer que LUNES corresponda a la constante 0, MARTES a la constante 1 y así sucesivamente.

Por tanto, cada constante enumerada corresponde a un valor entero. Si no se especifica nada, el primer valor tomará el valor de 0 y las siguientes constantes irán incrementando su valor en una unidad. No obstante, C++ permite cambiar este criterio y asignar un valor determinado a cada constante:

```
enum comportamiento {HORRIBLE = 0, MALO, REGULAR = 100,
   BUENO = 200, MUY_BUENO, EXCELENTE};
```

De esta forma, HORRIBLE tomaría el valor de 0, MALO el valor de 1, REGULAR el valor de 100, BUENO el de 200, MUY_BUENO, el de 201 y EXCELENTE, el de 202.

ANOTACIONES

Otro aspecto a tener en cuenta es la recomendación en esta nueva versión que se refiere a realizar las coerciones de tipos de forma explícita. La versión C++ se recomienda por su legibilidad:

```
int i = 0;
long v = (long) i; // coerción de tipos en C
long v = long (i); // coerción de tipos en C++
```

4.2.6. La declaración de variables y constantes

La declaración de variables en C++ continua teniendo el mismo formato que en C, pero se introduce un nuevo elemento que aportará más seguridad en nuestra forma de trabajar. Se trata del especificador `const` para la definición de constantes.

En programación se utiliza una constante cuando se conoce con certeza que dicho valor no debe variar durante el proceso de ejecución de la aplicación:

```
const float PI = 3.14159;
```

Una vez definida dicha constante, no se le puede asignar ningún valor y, por tanto, siempre estará en la parte derecha de las expresiones. Por otro lado, una constante siempre tiene que estar inicializada:

```
const float radio;// ERROR!!!!!!!!
```

El uso de constantes no es nuevo en C. La forma clásica para definirlas es mediante la instrucción de prepocesador `#define`.

```
#define PI 3.14159
```

En este caso, el comportamiento real es sustituir cada aparición del texto `PI` por su valor en la fase de preprocesamiento del texto. Por tanto, cuando analiza el texto, el compilador sólo ve el número 3.14159 en lugar de `PI`.

ANOTACIONES

No obstante, mientras el segundo caso corresponde a un tratamiento especial durante el proceso previo a la compilación, el uso de `const` permite un uso normalizado y similar al de una variable pero con capacidades limitadas. A cambio, recibe el mismo tratamiento que las variables respecto al ámbito de actuación (sólo en el fichero de trabajo, a menos que se le indique lo contrario mediante la palabra reservada `extern`) y tiene un tipo asignado, con lo cual se podrán realizar todas las comprobaciones de tipos en fase de compilación haciendo el código fuente resultante más robusto.

4.2.7. La gestión de variables dinámicas

La gestión directa de la memoria es una de las armas más poderosas de las que dispone el C, y una de las más peligrosas: cualquier acceso inadecuado a zonas de memoria no correspondiente a los datos deseados puede provocar resultados imprevisibles en el mejor de los casos, cuando no desastrosos.

En los capítulos anteriores, se ha visto que las operaciones con direcciones de memoria en C se basan en los apuntadores (`*apunt`), usados para acceder a una variable a partir de su dirección de memoria. Utilizan el operador de indirección o desreferencia (`*`), y sus características son:

- `apuntador` contiene una dirección de memoria.
- `*apuntador` indica al contenido existente en dicha dirección de memoria.
- Para acceder a la dirección de memoria de una variable se utiliza el operador dirección (&) precediendo el nombre de la variable.

```
// Ejemplo de uso de apuntadores
int i = 10;
int *apunt_i = &i;
      // apunt_i toma la dirección
      // de la variable i de tipo entero
      // Si no lo asignáramos aquí, sería
      // recomendable inicializarlo a NULL

*apunt_i = 3;
      // Se asigna el valor 3 a la posición
```

ANOTACIONES

```
     // de memoria apunt_i
     //Por tanto, se modifica el valor de i.

cout << "Valor Original         : " << i << endl ;
cout << "A través del apuntador : "
           << *apunt_i << endl ;
           // La salida mostrará:
           // Valor original: 3
           // A través del apuntador: 3
```

Operadores `new` y `delete`

El principal uso de los apuntadores está relacionado con las variables dinámicas. Las dos principales funciones definidas en C para realizar estas operaciones son `malloc()` y `free()` para reservar memoria y liberarla, respectivamente. Ambas funciones continúan siendo válidas en C++. No obstante, C++ proporciona dos nuevos operadores que permiten un control más robusto para dicha gestión. Son `new` y `delete`. Al estar incluidos en el lenguaje, no se precisa la inclusión de ninguna librería específica.

El operador `new` realiza la reserva de memoria. Su formato es `new` y un tipo de datos. A diferencia del `malloc`, en este caso no es preciso indicarle el tamaño de la memoria a reservar, pues el compilador lo calcula a partir del tipo de datos utilizado.

Para liberar la memoria se dispone del operador `delete` que también ofrece más robustez que la función `free()`, pues protege internamente el hecho de intentar liberar memoria apuntada por un apuntador nulo.

```
fecha * aniversario = new fecha;
...
delete aniversario;
```

Si se desea crear varios elementos, basta especificarlo en forma de vector o matriz. El resultado es declarar la variable como apuntador al primer elemento del vector.

ANOTACIONES

De forma similar, se puede liberar toda la memoria reservada por el vector (cada uno de los objetos creados y el vector mismo) utilizando la forma `delete []`.

```
fecha * lunasLlenas = new fecha[12];
...
delete [] lunasLlenas;
```

Si se omiten los corchetes, el resultado será eliminar únicamente el primer objeto del vector, pero no el resto creando una fuga de memoria; es decir, memoria reservada a la cual no se puede acceder en el futuro.

Apuntadores const

En la declaración de apuntadores en C++ permite el uso de la palabra reservada `const`. Y además existen diferentes posibilidades.

```
const int * ap_i;         // *ap_i permanece constante
int * const ap_j;
         // Dirección ap_j es constante, pero no su valor
const int * const ap_k;
         // Tanto la dirección ap_k como su valor *ap_k serán constantes
```

Es decir, en el caso de apuntadores se puede hacer constante su valor (`*ap_i`) o su dirección de memoria (`ap_i`), o las dos cosas. Para no confundirse, basta con fijarse en el texto posterior a la palabra reservada `const`.

Con la declaración de apuntadores constantes el programador le indica al compilador cuándo se desea que el valor o la dirección que contiene un apuntador no sufran modificaciones. Por tanto, cualquier intento de asignación no prevista se detecta en tiempo de compilación. De esta forma se reduce el riesgo de errores de programación.

Referencias

Para la gestión de variables dinámicas C++ añade un nuevo elemento para facilitar su uso: las referencias. Una referencia es un

ANOTACIONES

alias o un sinónimo. Cuando se crea una referencia, se inicializa con el nombre de otra variable y actúa como un nombre alternativo de ésta.

Para crearla se escribe el tipo de la variable destino, seguido del operador de referencia (&) y del nombre de la referencia. Por ejemplo,

```
int i;
int & ref_i = i;
```

En la expresión anterior se lee: la variable `ref_i` es una referencia a la variable `i`. Las referencias siempre se tienen que inicializar en el momento de la declaración (como si fuera una `const`).

Hay que destacar que aunque el operador de referencia y el de dirección se representan de la misma forma (&), corresponden a operaciones diferentes, aunque están relacionados entre sí. De hecho, la característica principal de las referencias es que si se pide su dirección, devuelven la de la variable destino.

```
#include <iostream>

int main()
{
   int i = 10;
   int & ref_i = i;

ref_i = 3; //Se asigna el valor 3 a la posición

   cout << "valor de i          " << i << endl;
   cout << "dirección de i      " << &i << endl;
   cout << "dirección de ref_i " << &ref_i <<endl;
}
```

Con este ejemplo se puede comprobar que las dos direcciones son idénticas, y como la asignación sobre `ref_i` tiene el mismo efecto que si hubiera sido sobre `i`.

Otras características de las referencias son las siguientes:

- No se pueden reasignar. El intento de reasignación se convierte en una asignación en la variable sinónima.
- No se le pueden asignar un valor nulo.

El uso principal de las referencias es el de la llamada a funciones, que se verá a continuación.

4.2.8. Las funciones y sus parámetros

El uso de las funciones, elemento básico de la programación modular, continúa teniendo el mismo formato: un tipo del valor de retorno, el nombre de la función y un número de parámetros precedidos por su tipo. A esta lista de parámetros de una función también se la conoce como la ***firma de una función***.

Uso de parámetros por valor o por referencia

Como se ha comentado en unidades anteriores, en C existen dos formas de pasar parámetros a una función: por valor o por variable. En el primer caso, la función recibe una copia del valor original del parámetro mientras que en el segundo se recibe la dirección de dicha variable. De esta forma, se puede acceder directamente a la variable original, que puede ser modificada. Para hacerlo, la forma tradicional en C es pasar a la función como parámetro el apuntador a una variable.

Nota

Aquí no utilizaremos el término por referencia para no inducir a confusión con las referencias de C++.

A continuación, veremos una función que permite intercambiar el contenido de dos variables:

```
#include <iostream>

void intercambiar(int *i, int *j);

int main()
{
```

ANOTACIONES

```
   int x = 2, y = 3;
   cout << " Antes. x = " << x << " y = " << y << endl;
   intercambiar(&x , &y);
   cout << " Después. x = " << x << " y = " << y << endl;
}
void intercambiar(int *i, int *j)
{
   int k;
   k = *i;
   *i = *j;
   *j = k;
}
```

Se puede comprobar que el uso de las desreferencias (*) dificulta su comprensión. Pero en C++ se dispone de un nuevo concepto comentado anteriormente: las referencias. La nueva propuesta consiste en recibir el parámetro como referencia en lugar de apuntador:

```
#include <iostream>

void intercambiar(int &i, int &j);

int main()
{
   int x = 2, y = 3;
   cout << " Antes. x = " << x << " y = " << y << endl;
   intercambiar(x , y); //NO intercambiar(&x , &y);
   cout << " Despues. x = " << x << " y = " << y << endl;
}
void intercambiar(int & i, int & j)
{
   int k;
   k = i;
   i = j;
   j = k;
}
```

El funcionamiento de esta nueva propuesta es idéntico al de la anterior, pero la lectura del código fuente es mucho más simple al utilizar el operador de referencia (&) para recoger las direcciones de memoria de los parámetros.

ANOTACIONES

No obstante, hay que recordar que las referencias tienen una limitaciones (no pueden tomar nunca un valor nulo y no pueden reasignarse). Por tanto, no se podrán utilizar las referencias en el paso de parámetros cuando se desee pasar un apuntador como parámetro y que éste pueda ser modificado (por ejemplo, obtener el último elemento en una estructura de datos de cola). Tampoco se podrán utilizar referencias para aquellos parámetros que deseamos considerar como opcionales, al existir la posibilidad de que no pudieran ser asignados a ningún parámetro de la función que los llama, por lo que tendrían que tomar el valor `null`, (lo que no es posible).

En estos casos, el paso de parámetros por variable se debe continuar realizando mediante el uso de apuntadores.

Uso de parámetros `const`

En la práctica, al programar en C, a veces se utiliza el paso por variable como una forma de eficiencia al evitar tener que realizar una copia de los datos dentro de la función. Con estructuras de datos grandes (estructuras, etc.) esta operación interna de salvaguarda de los valores originales puede ocupar un tiempo considerable y, además, se corre el riesgo de una modificación de los datos por error.

Para reducir estos riesgos, C++ permite colocar el especificador `const` justo antes del parámetro (tal como se ha comentado en el apartado de apuntadores `const`).

Si en el ejemplo anterior de la función intercambiar se hubiera definido los parámetros `i` y `j` como `const` (lo cual no tiene ningún sentido práctico y sólo se considera a efectos explicativos), nos daría errores de compilación:

```
void intercambiar(const int & i, const int & j);
{
   int k;
   k = i;
   i = j;   // Error de compilación. Valor i constante.
   j = k;   // Error de compilación. Valor j constante.
}
```

ANOTACIONES

De este modo se pueden conseguir las ventajas de eficiencia al evitar los procesos de copia no deseados sin tener el inconveniente de estar desprotegido ante modificaciones no deseadas.

Sobrecarga de funciones

C admite un poco de flexibilidad en las llamadas a funciones al permitir el uso de un número de parámetros variables en la llamada a una función, siempre y cuando sean los parámetros finales y en la definición de dicha función se les haya asignado un valor para el caso que no se llegue a utilizar dicho parámetro.

C++ ha incorporado una opción mucho más flexible, y que es una de las novedades respecto al C más destacadas: permite el uso de diferentes funciones con el mismo nombre (*homonímia de funciones*). A esta propiedad también se la denomina *sobrecarga de funciones*.

Las funciones pueden tener el mismo nombre pero deben tener diferencias en su lista de parámetros, sea en el número de parámetros o sea en variaciones en el tipo de éstos.

Hay que destacar que el tipo del valor de retorno de la función no se considera un elemento diferencial de la función, por tanto, el compilador muestra error si se intenta definir dos funciones con el mismo nombre e idéntico número y tipo de parámetros pero que retornen valores de distintos tipos. El motivo es que el compilador no puede distinguir a cuál de las funciones definidas se desea llamar.

A continuación se propone un programa que eleva números de diferente tipo al cuadrado:

```
#include <iostream>

int elevarAlCuadrado (int);
float elevarAlCuadrado (float);

int main()
```

ANOTACIONES

```
{
   int numEntero = 123;
   float numReal = 12.3;
   int numEnteroAlCuadrado;
   float numRealAlCuadrado;

   cout << "Ejemplo para elevar números al cuadrado\n";
   cout << "Números originales \n";
   cout << "Número entero: " << numEntero << "\n";
   cout << "Número real: " << numReal << "\n";

   numEnteroAlCuadrado = elevarAlCuadrado (numEntero);
   numRealAlCuadrado = elevarAlCuadrado (numReal);

   cout << "Números elevados al cuadrado \n";
   cout << "Número entero:"<< numEnteroAlCuadrado << "\n";
   cout << "Número real: " << numRealAlCuadrado << "\n";

   return 0;
}

int elevarAlCuadrado (int num)
{
   cout << "Elevando un número entero al cuadrado \n";
   return ( num * num);
}
float elevarAlCuadrado (float num)
{
   cout << "Elevando un número real al cuadrado \n";
   return ( num * num);
}
```

El hecho de sobregargar la función `ElevarAlCuadrado` ha permitido que con el mismo nombre de función se pueda realizar lo que es intrínsicamente la misma operación. Con ello, hemos evitado tener que definir dos nombres de función diferentes:

– `ElevarAlCuadradoNumerosEnteros`
– `ElevarAlCuadradoNumerosReales`

De esta forma, el mismo compilador identifica la función que se desea ejecutar por el tipo de sus parámetros y realiza la llamada correcta.

ANOTACIONES

4.3. El paradigma de la programación orientada a objetos

En las unidades anteriores, se han analizado una serie de paradigmas de programación (modular y descendente) que se basan en la progresiva organización de los datos y la resolución de los problemas a partir de su división en un conjunto de instrucciones secuenciales. La ejecución de dichas instrucciones sólo se apoya en los datos definidos previamente.

Este enfoque, que nos permite afrontar múltiples problemas, también muestra sus limitaciones:

– El uso compartido de los datos provoca que sea difícil modificar y ampliar los programas por sus interrelaciones.

– El mantenimiento de los grandes programas se vuelve realmente complicado al no poder asegurar el control de todas las implicaciones que suponen cambios en el código.

– La reutilización de código también puede provocar sorpresas, otra vez por no poder conocer todas las implicaciones que suponen.

Nota

¿Cómo es posible que se tengan tantas dificultades si las personas son capaces de realizar acciones complejas en su vida cotidiana? La razón es muy sencilla: en nuestra vida cotidiana no se procede con los mismos criterios. La descripción de nuestro entorno se hace a partir de objetos: puertas, ordenadores, automóviles, ascensores, personas, edificios, etc., y estos objetos cumplen unas relaciones más o menos simples: si una puerta está abierta se puede pasar y si está cerrada no se puede. Si un automóvil tiene una rueda pinchada, se sustituye y se puede volver a circular. ¡Y no necesitamos conocer toda la mecánica del automóvil para poder realizar esta operación! Ahora bien, ¿nos podríamos imaginar nuestro mundo si al cambiar el neumático nos dejara de funcionar el limpiapa-

ANOTACIONES

rabrisas? Sería un caos. Eso es casi lo que sucede, o al menos no podemos estar completamente seguros de que no suceda, con los paradigmas anteriores.

El paradigma de la orientación a objetos nos propone una forma diferente de enfocar la programación sobre la base de la definición de objetos y de las relaciones entre ellos.

Cada objeto se representa mediante una **abstracción** que contiene su información esencial, sin preocuparse de sus demás características.

Esta información está compuesta de datos (variables) y acciones (funciones) y, a menos que se indique específicamente lo contrario, su ámbito de actuación está limitado a dicho objeto (**ocultamiento de la información**). De esta forma, se limita el alcance de su código de programación, y por tanto su repercusión, sobre el entorno que le rodea. A esta característica se le llama **encapsulamiento**.

Las relaciones entre los diferentes objetos pueden ser diversas, y normalmente suponen acciones de un objeto sobre otro que se implementan mediante mensajes entre los objetos.

Ejemplo

Ejemplo de acciones sobre objetos en la vida cotidiana: llamar por teléfono, descolgar el teléfono, hablar, contestar, etc.

Una de las relaciones más importante entre los objetos es la pertenencia a un tipo más general. En este caso, el objeto más específico comparte una serie de rasgos (información y acciones) con el más genérico que le vienen dados por esta relación de inclusión. El nuevo paradigma proporciona una herramienta para poder reutilizar todos estos rasgos de forma simple: **la herencia**.

Ejemplo

Un perro es un animal, un camión es un vehículo, etc.

Finalmente, una característica adicional es el hecho de poder comportarse de forma diferente según el contexto que lo envuelve. Es conocida como **polimorfismo** (un objeto, muchas formas). Además, esta propiedad adquiere toda su potencia al ser capaz de adaptar este comportamiento en el momento de la ejecución y no en tiempo de compilación.

El paradigma de programación orientada a objetos se basa en estos cuatro pilares: abstracción, encapsulación, herencia y polimorfismo.

ANOTACIONES

4.3.1. Clases y objetos

A nivel de implementación, una clase corresponde a un nuevo tipo de datos que contiene una colección de datos y de funciones que nos permiten su manipulación.

Ejemplo

Deseamos describir un aparato de vídeo.

La descripción se puede hacer a partir de sus características como marca, modelo, número de cabezales, etc. o a través de sus funciones como reproducción de cintas de vídeo, grabación, rebobinado, etc. Es decir, tenemos dos visiones diferentes para tratar el mismo aparato.

El primer enfoque correspondería a una colección de variables, mientras que el segundo correspondería a una colección de funciones.

El uso de las clases nos permite integrar datos y funciones dentro de la misma entidad.

El hecho de reunir el conjunto de características y funciones en el mismo contenedor facilita su interrelación, así como su aislamiento del resto del código fuente. En el ejemplo del aparato de vídeo, la reproducción de una cinta implica, una vez puesta la cinta, la acción de unos motores que hacen que la cinta se vaya desplazando por delante de unos cabezales que leen la información.

En realidad, a los usuarios el detalle del funcionamiento nos es intrascendente, sencillamente, vemos el aparato de vídeo como una caja que tiene una ranura y unos botones, y sabemos que en su interior contiene unos mecanismos que nos imaginamos bastante complejos. Pero también sabemos que a nosotros nos basta con pulsar el botón "Play".

A este concepto se le denomina **encapsulación de datos y funciones en una clase**. Las variables que están dentro de la clase reciben el

ANOTACIONES

nombre de *variables miembro* o *datos miembro* , y a las funciones se las denomina *funciones miembro* o *métodos de la clase*.

Pero las clases corresponden a elementos abstractos; es decir, a ideas genéricas y en nuestra casa no disponemos de un aparato de vídeo en forma de idea, sino de un elemento real con unas características y funciones determinadas. De la misma forma, en C++ necesitaremos trabajar con los elementos concretos. A estos elementos los llamaremos *objetos*.

Declaración de una clase

La sintaxis para la declaración de una clase es utilizar la palabra reservada `class` seguida del nombre de la clase y, entre llaves, la lista de las variables miembro y de las funciones miembro.

```
class Perro
{
   // lista de variables miembro
   int edad;
   int altura;

   // lista de funciones miembro
   void ladrar();
};
```

La declaración de la clase no implica reserva de memoria alguna, aunque informa de la cantidad de memoria que precisará cada uno de los objetos de dicha clase.

Ejemplo

En la clase Perro presentada, cada uno de los objetos ocupará 8 bytes de memoria: 4 bytes para la variable miembro `edad` de tipo entero y 4 bytes para la variable miembro `altura`. Las definiciones de las funciones miembro, en nuestro caso `ladrar`, no implican reserva de espacio.

ANOTACIONES

Implementación de las funciones miembro de una clase

Hasta el momento, en la clase Perro hemos declarado como miembros dos variables (`edad` y `altura`) y una función (`ladrar`). Pero no se ha especificado la implementación de la función.

La definición de una función miembro se hace mediante el nombre de la clase seguido por el operador de ámbito (`::`) , el nombre de la función miembro y sus parámetros.

```
class Perro
{
   // lista de variables miembro
   int edad;
   int altura;

   // lista de funciones miembro
   void ladrar();
};

Perro:: ladrar()
{
   cout << "Guau";
}
```

Nota

Aunque ésta es la forma habitual, también es posible implementar las funciones miembro de forma *inline*. Para ello, después de la declaración del método y antes del punto y coma (;) se introduce el código fuente de la función:

```
class Perro
{
   // lista de variables miembro
   int edad;
   int altura;

   // lista de funciones miembro
```

ANOTACIONES

Nota

```
   void ladrar()
     { cout << "Guau"; };
};
```

Este tipo de llamadas sólo es útil cuando el cuerpo de la función es muy reducido (una o dos instrucciones).

Funciones miembros const

En el capítulo anterior se comentó la utilidad de considerar las variables que no deberían sufrir modificaciones en el transcurso de la ejecución del programa, y su declaración mediante el especificador const. También se comentó la seguridad que aportaban los apuntadores const. Pues de forma similar se podrá definir a una función miembro como const.

```
void ladrar() const;
```

Para indicar que una función miembro es const, sólo hay que poner la palabra reservada const entre el símbolo de cerrar paréntesis después del paso de parámetros y el punto y coma (;) final.

Al hacerlo, se le indica al compilador que esta función miembro no puede modificar al objeto. Cualquier intento en su interior de asignar una variable miembro o llamar a alguna función no constante generará un error por parte del compilador. Por tanto, es una medida más a disposición del programador para asegurar la coherencia de las líneas de código fuente.

Declaración de un objeto

Así como una clase se puede asimilar como un nuevo tipo de datos, un objeto sólo corresponde a la definición de un elemento de dicho tipo. Por tanto, la declaración de un objeto sigue el mismo modelo:

```
unsigned int numeroPulgas; // variable tipo unsigned int
Perro sultan;              // objeto de la clase Perro.
```

ANOTACIONES

Un objeto es una instancia individual de una clase.

4.3.2. Acceso a objetos

Para acceder a las variables y funciones miembro de un objeto se utiliza el operador punto (.), es decir, poner el nombre del objeto seguido de punto y del nombre de la variable o función que se desea.

En el apartado anterior donde se ha definido un objeto `sultan` de la clase Perro, si se desea inicializar la edad de `sultán` a 4 o llamar a su función `ladrar`(), sólo hay que referirse a lo siguiente:

```
Sultan.edad = 4;
Sultan.ladrar();
```

No obstante, una de las principales ventajas proporcionadas por las clases es que sólo son visibles aquellos miembros (tanto datos como funciones) que nos interesa mostrar. Por este motivo, si no se indica lo contrario, los miembros de una clase sólo son visibles desde las funciones de dicha clase. En este caso, diremos que dichos miembros son **privados**.

Para poder controlar el ámbito de los miembros de una clase, se dispone de las siguientes palabras reservadas: `public`, `private` y `protected`.

Nota

`protected` corresponde a un caso más específico y se estudiará en el apartado "Herencia".

Cuando se declara un miembro (tanto variables como funciones) como `private` se le está indicando al compilador que el uso de esta variable es privado y está restringido al interior de dicha clase. En cambio, si se declara como `public`, es accesible desde cualquier lugar donde se utilicen objetos de dicha clase.

En el código fuente, estas palabras reservadas se aplican con forma de etiqueta delante de los miembros del mismo ámbito:

```
class Perro
{
   public:
     void ladrar();
```

ANOTACIONES

```
  private:
    int edad;
    int altura;
};
```

Nota

Se ha declarado la función `ladrar()` como pública permitiendo el acceso desde fuera de la clase, pero se han mantenido ocultos los valores de `edad` y `altura` al declararlos como privados.

Veámoslo en la implementación de un programa en forma completa:

```
#include <iostream>
class Perro
{
   public:
     void ladrar() const
     { cout << "Guau"; };

   private:
     int edad;
     int altura;
};

int main()
{
   Perro sultan;

   //Error compilación. Uso variable privada
   sultan.edad = 4;
   cout << sultan.edad;
}
```

En el bloque `main` se ha declarado un objeto `sultan` de la clase Perro. Posteriormente, se intenta asignar a la variable miembro edad el valor de 4. Como dicha variable no es pública, el compilador da error indicando que no se tiene acceso a ella. Igualmente, nos mostraría un error de compilación similar para la siguiente fila. Una solución en este caso sería declarar la variable miembro `edad` como `public`.

ANOTACIONES

Privacidad de los datos miembro

El hecho de declarar una variable miembro como pública limita la flexibilidad de las clases, pues una modificación del tipo de la variable afectará a los diferentes lugares del código donde se utilicen dichos valores.

Una regla general de diseño recomienda mantener los datos miembro como privados, y manipularlos a través de funciones públicas de acceso donde se obtiene o se le asigna un valor.

En nuestro ejemplo, se podría utilizar las funciones `obtenerEdad`() y `asignarEdad`() como métodos de acceso a los datos. Declarar la variable edad como privada nos permitiría cambiar el tipo en entero a byte o incluso sustituir el dato por la fecha de nacimiento. La modificación se limitaría a cambiar el código fuente en los métodos de acceso, pero continuaría de forma transparente fuera de la clase, pues se puede calcular la edad a partir de la fecha actual y la fecha de nacimiento o asignar una fecha de nacimiento aproximada a partir de un valor para la edad.

```
class Perro
{
  public:

    Perro(int, int);   // método constructor
    Perro(int);        //
    Perro();           //
    ~Perro();          // método destructor

    void asignarEdad(int);   // métodos de acceso
    int obtenerEdad();       //
    void asignarAltura(int); //
    int obtenerAltura();     //

    void ladrar();           // métodos de la clase
```

ANOTACIONES

```
   private:
     int edad;
     int altura;
};

Perro:: ladrar()
{ cout << "Guau"; }

void Perro:: asignarAltura (int nAltura)
{ altura = nAltura; }

int Perro:: obtenerAltura (int nAltura)
{ return (altura); }

void Perro:: asignarEdad (int nEdad)
{ edad = nEdad; }

int Perro:: obtenerEdad ()
{ return (edad); }
```

El apuntador `this`

Otro aspecto a destacar de las funciones miembro es que siempre tienen acceso al propio objeto a través del apuntador `this`.

De hecho, la función miembro `obtenerEdad` también se podría expresar de la siguiente forma:

```
int Perro:: obtenerEdad ()
{ return (this→edad); }
```

Visto de este modo, parece que su importancia sea escasa. No obstante, poder referirse al objeto sea como apuntador `this` o en su forma desreferenciada (`*this`) le da mucha potencia. Veremos ejemplos más avanzados al respecto cuando se estudie la sobrecarga de operadores.

ANOTACIONES

4.3.3. Constructores y destructores de objetos

Generalmente, cada vez que se define una variable, después se inicializa. Ésta es una práctica correcta en la que se intenta prevenir resultados imprevisibles al utilizar una variable sin haberle asignado ningún valor previo.

Las clases también se pueden inicializar. Cada vez que se crea un nuevo objeto, el compilador llama a un método específico de la clase para inicializar sus valores que recibe el nombre de **constructor**.

El constructor siempre recibe el nombre de la clase, sin valor de retorno (ni siquiera `void`) y puede tener parámetros de inicialización.

```
Perro::Perro()
{
   edad = 0;
}
```

o

```
Perro::Perro(int nEdad)// Nueva edad del perro
{
   edad = nEdad;
}
```

En caso de que no se defina específicamente el constructor en la clase, el compilador utiliza el **constructor predeterminado**, que consiste en el nombre de la clase, sin ningún parámetro y tiene un bloque de instrucciones vacío. Por tanto, no hace nada.

```
Perro::Perro()
{ }
```

Esta característica suena desconcertante, pero permite mantener el mismo criterio para la creación de todos los objetos.

En nuestro caso, si deseamos inicializar un objeto de la clase Perro con una edad inicial de 1 año, utilizamos la siguiente definición del constructor:

```
Perro(int nuevaEdad);
```

De esta manera, la llamada al constructor sería de la siguiente forma:

```
Perro sultan(1);
```

Si no hubiera parámetros, la declaración del constructor a utilizar dentro de la clase sería la siguiente:

```
Perro();
```

La declaración del constructor en `main` o cualquier otra función del cuerpo del programa quedaría del modo siguiente:

```
Perro sultan();
```

Pero en este caso especial se puede aplicar una excepción a la regla que indica que todas las llamadas a funciones van seguidas de paréntesis aunque no tengan parámetros. El resultado final sería el siguiente:

```
Perro sultan;
```

El fragmento anterior es una llamada al constructor `Perro()` y coincide con la forma de declaración presentada inicialmente y ya conocida.

De la misma forma, siempre que se declara un método constructor, se debería declarar un método **destructor** que se encarga de limpiar cuando ya no se va a usar más el objeto y liberar la memoria utilizada.

El destructor siempre va antecedido por una tilde (~), tiene el nombre de la clase, y no tiene parámetros ni valor de retorno.

ANOTACIONES

```
~Perro();
```

En caso de que no definamos ningún destructor, el compilador define un **destructor predeterminado**. La definición es exactamente la misma, pero siempre tendrá un cuerpo de instrucciones vacío:

```
Perro::~Perro()
{ }
```

Incorporando las nuevas definiciones al programa, el resultado final es el siguiente:

```
#include <iostream>

class Perro
{
   public:
     Perro(int edad); // constructor con un parámetro
     Perro();         // constructor predeterminado
     ~Perro();        // destructor
     void ladrar();

   private:
     int edad;
     int altura;
};

Perro:: ladrar()
{ cout << "Guau"; }

int main()
{
   Perro sultan(4);    // Inicializando el objeto
                       // con una edad de 4.
   sultan.ladrar();
}
```

ANOTACIONES

El constructor de copia

El compilador, además de proporcionar de forma predeterminada a las clases un método constructor y un método destructor, también proporciona un método constructor de copia.

Cada vez que se crea una copia de un objeto se llama al constructor de copia. Esto incluye los casos en que se pasa un objeto como parámetro por valor a una función o se devuelve dicho objeto como retorno de la función. El propósito del constructor de copia es realizar una copia de los datos miembros del objeto a uno nuevo. A este proceso también se le llama *copia superficial* .

Este proceso, que generalmente es correcto, puede provocar fuertes conflictos si entre las variables miembros a copiar hay apuntadores. El resultado de la copia superficial haría que dos apuntadores (el del objeto original y el del objeto copia) apunten a la misma dirección de memoria: si alguno de ellos liberara dicha memoria, provocaría que el otro apuntador, al no poder percatarse de la operación, se quedara apuntando a una posición de memoria perdida generando una situación de resultados impredecibles.

La solución en estos casos pasa por sustituir la copia superficial por una **copia profunda** en la que se reservan nuevas posiciones de memoria para los elementos tipo apuntador y se les asigna el contenido de las variables apuntadas por los apuntadores originales.

La forma de declarar dicho constructor es la siguiente:

```
Perro :: Perro (const Perro & unperro);
```

En esta declaración se observa la conveniencia de pasar el parámetro como referencia constante pues el constructor no debe alterar el objeto.

La utilidad del constructor de copia se observa mejor cuando alguno de los atributos es un apuntador. Por este motivo y para esta prueba cambiaremos el tipo de edad a apuntador a entero. El resultado final sería el siguiente:

ANOTACIONES

```
class Perro
{
   public:
     Perro();                          // constructor predeterminado
     ~Perro();                         // destructor
     Perro(const Perro & rhs);         // constructor de copia
     int obtenerEdad();                // método de acceso

   private:
     int *edad;
};

Perro :: obtenerEdad()
{ return (*edad) }

Perro :: Perro ()                      // Constructor
{
   edad = new int;
   * edad = 3;
}

Perro :: ~Perro ()                     // Destructor
{
   delete edad;
   edad = NULL;
}
Perro :: Perro (const Perro & rhs) // Constructor de copia
{
   edad = new int;                     // Se reserva nueva memoria
   *edad = rhs.obtenerEdad();          // Copia el valor edad
                                       // en la nueva posición


}
int main()
{
   Perro sultan(4);                   // Inicializando con edad 4.
}
```

ANOTACIONES

Inicializar valores en los métodos constructores

Hay otra forma de inicializar los valores en un método constructor de una forma más limpia y eficiente que consiste en interponer dicha inicialización entre la definición de los parámetros del método y la llave que indica el inicio del bloque de código.

```
Perro:: Perro () :
   edad (0),
   altura (0)
{ }

Perro:: Perro (int nEdad, int nAltura):
   edad (nEdad),
   altura(nAltura)
{ }
```

Tal como se ve en el fragmento anterior, la inicialización consiste en un símbolo de dos puntos (:) seguido de la variable a inicializar y, entre paréntesis, el valor que se le desea asignar. Dicho valor puede corresponder tanto a una constante como a un parámetro de dicho constructor. Si hay más de una variable a inicializar, se separan por comas (,).

Variables miembro y funciones miembro estáticas

Hasta el momento, cuando nos hemos referido a las clases y a los objetos los hemos situado en planos diferentes: las clases describían entes abstractos, y los objetos describían elementos creados y con valores concretos.

No obstante, hay momentos en que los objetos necesitan referirse a un atributo o a un método común con los demás objetos de la misma clase.

Ejemplo

Si estamos creando una clase Animales, nos puede interesar conservar en alguna variable el número total de

ANOTACIONES

perros que se han creado hasta el momento, o hacer una función que nos permita contar los perros aunque todavía no se haya creado ninguno.

La solución es preceder la declaración de las variables miembro o de las funciones miembro con la palabra reservada `static`. Con ello le estamos indicando al compilador que dicha variable, o dicha función, se refiere a la clase en general y no a ningún objeto en concreto. También se puede considerar que se está compartiendo este dato o función con todas las instancias de dicho objeto.

En el siguiente ejemplo se define una variable miembro y una función miembro estáticas:

```
class Perro {
// ...
static int numeroDePerros; //normalmente será privada
static int cuantosPerros() { return numeroDePerros; }
};
```

Para acceder a ellos se puede hacer de dos formas:

- Desde un objeto de la clase Perro.

```
Perro sultan = new Perro();
sultan.numeroDePerros; //normalmente será privada
sultan.cuantosPerros();
```

- Utilizando el identificador de la clase sin definir ningún objeto.

```
Perro::numeroDePerros; //normalmente será privada
Perro::cuantosPerros();
```

Pero hay que tener presente un aspecto importante: las variables y las funciones miembro `static` se refieren siempre a la clase y no a ningún objeto determinado, con lo cual el objeto `this` no existe.

ANOTACIONES

Como consecuencia, en las funciones miembro estáticas no se podrá hacer referencia ni directa ni indirectamente al objeto `this` y:

- Sólo podrán hacer llamadas a funciones estáticas, pues las funciones no estáticas siempre esperan implícitamente a dicho objeto como parámetro.
- Sólo podrán tener acceso a variables estáticas, porque a las variables no estáticas siempre se accede a través del mencionado objeto.
- No se podrán declarar dichas funciones como `const` pues ya no tiene sentido.

4.3.4. Organización y uso de bibliotecas en C++

Hasta el momento se ha incluido en el mismo archivo la definición de la clase y el código que la utiliza, pero ésta no es la organización aconsejada si se desea reutilizar la información.

Se recomienda dividir el código fuente de la clase en dos ficheros separando la definición de la clase y su implementación:

- El fichero de cabecera incorpora la definición de la clase. La extensión de dichos ficheros puede variar entre diversas posibilidades, y la decisión final es arbitraria.
- El fichero de implementación de los métodos de la clase y que contiene un include del fichero de cabecera. La extensión utilizada para dichos ficheros también puede variar.

Posteriormente, cuando se desee reutilizar esta clase, bastará con incluir en el código fuente una llamada al fichero de cabecera de la clase.

En nuestro ejemplo, la implementación se encuentra en el fichero perro.cpp, que hace un `include` del fichero de cabecera perro.hpp.

Nota

Las extensiones utilizadas de forma más estándar son .hpp (más utilizadas en entornos Windows), .H y .hxx (más utilizadas en entornos Unix) o incluso .h (al igual que en C).

Nota

Las extensiones utilizadas de forma más estándar son .cpp (más frecuentes en entornos Windows) .C y .cxx (más utilizadas en entornos Unix).

ANOTACIONES

Fichero perro.hpp (fichero de cabecera de la clase)

```
class Perro
{
   public:

   Perro(int edad);     //métodos constructores
   Perro();
   ~Perro();            //método destructor
   int obtenerEdad();   // métodos de acceso
   int asignarEdad(int);
   int asignarAltura(int);
   int obtenerAltura();
   void ladrar();       // métodos propios

   private:
   int edad;
   int altura;
};
```

Fichero perro.cpp (fichero de implementación de la clase)

```
#include <iostream>    //necesaria para el cout
#include <perro.hpp>

Perro:: ladrar()
{ cout << "Guau"; }

void Perro:: asignarAltura (int nAltura)
{ altura = nAltura; }

int Perro:: obtenerAltura (int nAltura)
{ return (altura); }

void Perro:: asignarEdad (int nEdad)
{ edad = nEdad; }

int Perro:: obtenerEdad ()
{ return (edad); }
```

ANOTACIONES

Fichero ejemplo.cpp

```
#include <perro.hpp>
int main()
{
   //Inicializando el objeto con edad 4.
   Perro sultan (4);
    sultan.ladrar();
}
```

Las bibliotecas estándar

Los compiladores suelen incluir una serie de funciones adicionales para que el programador las utilice. En el caso de GNU, proporciona una biblioteca estándar de funciones y objetos para los programadores de C++ denominada libstdc++.

Esta librería proporciona las operaciones de entrada/salida con *streams*, *strings*, vectores, listas, algoritmos de comparación, operaciones matemáticas y algoritmos de ordenación entre muchas otras operaciones.

Uso de la biblioteca STL

C++ ha incorporado un nivel más de abstracción al introducir las **plantillas**, que también se conocen como ***tipos parametrizados***. No es objeto de este curso el desarrollar este tema, pero la potencia que proporciona al C++ la inclusión de STL (Biblioteca Estándar de Plantillas) nos obliga a hacer una breve reseña de cómo utilizarla.

La idea básica de las plantillas es simple: cuando se implementa una operación general con un objeto (por ejemplo, una lista de perros) definiremos las diferentes operaciones de manipulación de una lista sobre la base de la clase Perro. Si posteriormente se desea realizar una operación similar con otros objetos (una lista de gatos), el código resultante para el mantenimiento de la lista es similar pero definiendo los elementos en base a la clase Gato. Por tanto, seguramente, nuestra forma de actuar sería hacer un copiar y pegar y modificar el bloque copiado para trabajar con la nueva clase deseada. No obstante, este proceso se repite

ANOTACIONES

cada vez que se desea implementar una nueva lista de otro tipo de objetos (por ejemplo, una lista de caballos).

Además, cada vez que se deseara modificar una operación de las listas, se debería cambiar cada una de sus personalizaciones. Por tanto, rápidamente se intuye que ésta implementación no sería eficiente.

La forma eficiente de hacerlo es generar un código genérico que realice las operaciones de las listas para un tipo que se le puede indicar con posterioridad. Este código genérico es el de las plantillas o tipos parametrizados.

Después de este breve comentario, se intuye la eficiencia y la potencia de esta nueva característica y también su complejidad, que, como ya hemos comentado, sobrepasa el objetivo de este curso. No obstante, para un dominio avanzado del C++ este tema es imprescindible y se recomienda la consulta de otras fuentes bibliográficas para completar estos conocimientos.

No obstante, mientras que la definición e implementación de una librería de plantillas puede llegar a adquirir una gran complejidad, el uso de la Librería Estándar de Plantillas (STL) es accesible.

En el siguiente ejemplo, se trabaja con la clase `set` que define un conjunto de elementos. Para ello, se ha incluido la librería `set` incluida en la STL.

```
#include <iostream>
#include <set>

int main()
{
// define un conjunto de enteros <int>
set<int> setNumeros;

//añade tres números al conjunto de números
setNumeros.insert(123);
setNumeros.insert(789);
setNumeros.insert(456);

// muestra cuántos números tiene
// el conjunto de números
```

ANOTACIONES

```
cout << "Conjunto números: "
      << setNumeros.size() << endl;

// se repite el proceso con un conjunto de letras
//define conjunto de carácteres <char>
set<char> setLetras;

setLetras.insert('a');
setLetras.insert('z');
cout << "Conjunto letras: "
      << setLetras.size() << endl;
return 0;
}
```

En este ejemplo, se han definido un conjunto de números y un conjunto de letras. Para el conjunto de números, se ha definido la variable `setNumeros` utilizando la plantilla `set` indicándole que se utilizarán elementos de tipo `<int>`. Este conjunto define varios métodos, entre los cuales el de insertar un elemento al conjunto (`.insert`) o contar el número de elementos (`.size`). Para el segundo se ha definido la variable `setLetras` también utilizando la misma plantilla `set` pero ahora con elementos tipo `<char>`.

La salida del programa nos muestra el número de elementos introducidos en el conjunto de números y, posteriormente, el número de elementos introducidos en el conjunto de letras.

4.4. Diseño de programas orientados a objetos

La potencia del paradigma de la programación orientada a objetos no radica sólo en la definición de clases y objetos, sino en todas las consecuencias que implican y que se pueden implementar en el lenguaje de programación.

En esta unidad se estudiarán las principales propiedades de este paradigma:

- La homonimia
- La herencia
- El polimorfismo

ANOTACIONES

Una vez asimilado el alcance del cambio de paradigma, se proporcionarán nuevas reglas para el diseño de aplicaciones.

4.4.1. La homonimia

Tal como su definición indica, homonimia se refiere al uso de dos o más sentidos (en nuestro caso, léase operaciones) con el mismo nombre.

En programación orientada a objetos, hablaremos de homonimia al utilizar el mismo nombre para definir la misma operación diversas veces en diferentes situaciones aunque, normalmente, con la misma idea de fondo. El ejemplo más claro es definir con el mismo nombre las operaciones que básicamente cumplen el mismo objetivo pero para diferentes objetos.

En nuestro caso, diferenciaremos entre sus dos formas principales: la **homonimia** (o **sobrecarga**) **de funciones** y la **homonimia de operadores**.

Sobrecarga de funciones y métodos

La sobrecarga de funciones se estudió anteriormente como una de las mejoras que proporciona más flexibilidad a C++ respecto de C, y es una de las características más utilizadas dentro de la definición de las clases.

Los constructores son un caso práctico de sobrecarga de métodos. Para cada clase, se dispone de un constructor por defecto que no tiene parámetros y que inicializa los objetos de dicha clase.

En nuestro ejemplo,

```
Perro::Perro()
{ }
```

ANOTACIONES

O, tal como se vio, se puede dar el caso de que siempre se desee inicializar dicha clase a partir de una edad determinada, o de una edad y una altura determinadas:

```
Perro::Perro(int nEdad) // Nueva edad del perro
{ edad = nEdad; }

Perro::Perro(int nEdad, int n:altura) // Nueva defin.
{
   edad = nEdad;
   altura = nAltura;
}
```

En los tres casos, se crea una instancia del objeto `Perro`. Por tanto, básicamente están realizando la misma operación aunque el resultado final sea ligeramente diferente.

Del mismo modo, cualquier otro método o función de una clase puede ser sobrecargado.

Sobrecarga de operadores

En el fondo, un operador no es más que una forma simple de expresar una operación entre uno o más operandos, mientras que una función nos permite realizar operaciones más complejas.

Por tanto, la sobrecarga de operadores es una forma de simplificar las expresiones en las operaciones entre objetos.

En nuestro ejemplo, se podría definir una función para incrementar la edad de un objeto `Perro`.

```
Perro Perro::incrementarEdad()
{
   ++edad;
   return (*this);
}
// la llamada resultante sería Sultan.IncrementarEdad()
```

ANOTACIONES

Aunque la función es muy simple, podría resultar un poco incómoda de utilizar. En este caso, podríamos considerar sobrecargar el operador ++ para que, al aplicarlo sobre un objeto `Perro`, se incrementara automáticamente su edad.

La sobrecarga del operador se declara de la misma forma que una función. Se utiliza la palabra reservada `operator`, seguida del operador que se va a sobrecargar. En el caso de las funciones de operadores unitarios, no llevan parámetros (a excepción del incremento o decremento postfijo que utilizan un entero como diferenciador).

```
#include <iostream>
class Perro
{
   public:
     Perro();
     Perro(nEdad);
     ~Perro();
     int obtenerEdad();
     const Perro & operator++();    // Operador ++i
     const Perro & operator++(int); // Operador i++

   private:
     int edad;
};

Perro::Perro():
   edad(0)
{ }

Perro::Perro(int nEdad):
   edad(nEdad)
{ }

int Perro::obtenerEdad()
{ return (edad); }

const Perro & Perro::operator++()
{
```

ANOTACIONES

```
   ++edad;
   return (*this);
}

const Perro & Perro::operator++(int x)
  {
   Perro temp = *this;
   ++edad;
   return (temp);
}

int main()
{
   Perro sultan(3);
   cout << "Edad de Sultan al comenzar el programa \n " ;
   cout << sultan.obtenerEdad() << endl;

   ++sultan;
   cout << "Edad de Sultan después de un aniversario \n ";
   cout << sultan.obtenerEdad() << endl;

   sultan++;
   cout << "Edad de Sultan después de otro aniversario\n";
   cout << sultan.obtenerEdad();
}
```

En la declaración de sobrecarga de los operadores, se observa cómo se devuelve una referencia `const` a un objeto tipo `Perro`. De esta forma, se protege la dirección del objeto original de cualquier cambio no deseado.

También es posible observar que las declaraciones para el operador postfijo y prefijo son prácticamente iguales, y sólo cambia el tipo de argumento. Para diferenciar ambos casos, se estableció la convención de que el operador postfijo tuviera en la declaración un parámetro tipo `int` (aunque este parámetro no se usa).

```
   const Perro & operator++();      // Operador ++i
   const Perro & operator++(int);   // Operador i++
```

ANOTACIONES

En la implementación de ambas funciones, también hay diferencias significativas:

- En el caso del operador prefijo, se procede a incrementar el valor de la edad del objeto y se retorna el objeto modificado a través del apuntador `this`.

```
const Perro & Perro::operator++()
{
    ++edad;
    return (*this);
}
```

- En el caso del operador postfijo, se requiere devolver el valor del objeto anterior a su modificación. Por este motivo, se establece una variable temporal que recoge el objeto original, se procede a su modificación y se retorna la variable temporal.

```
const Perro & Perro::operator++(int x)
{
    Perro temp = *this;
    ++edad;
    return (temp);
}
```

La definición de la sobrecarga del operador suma, que es un operador binario, sería como sigue:

```
// En este caso, la suma de dos objetos tipo Perro
// no tiene NINGÚN SENTIDO LÓGICO.
// SÓLO a efectos de mostrar como sería
// la declaración del operador, se ha considerado
// como resultado “posible”
// retornar el objeto Perro de la izquierda
// del operador suma con la edad correspondiente
// a la suma de las edades de los dos perros.

const Perro & Perro::operator+(const Perro & rhs)
{
   Perro temp = *this;
   temp.edad = temp.edad + rhs.edad;
   return (temp);
```

ANOTACIONES

Nota

Dado lo desconcertante de la lógica empleada en el ejemplo anterior, queda claro también que no se debe abusar de la sobrecarga de operadores. Sólo debe utilizarse en aquellos casos en que su uso sea intuitivo y ayude a una mayor legibilidad del programa.

4.4.2. La herencia simple

Los objetos no son elementos aislados. Cuando se estudian los objetos, se establecen relaciones entre ellos que nos ayudan a comprenderlos mejor.

Ejemplo

Un perro y un gato son objetos diferentes, pero tienen una cosa en común: los dos son mamíferos. También lo son los delfines y las ballenas, aunque se muevan en un entorno muy diferente, aunque no los tiburones, que entrarían dentro de la categoría de peces. ¿Qué tienen todos estos objetos en común? Todos son animales.

Se puede establecer una jerarquía de objetos, en la cual un perro es un mamífero, un mamífero es un animal, un animal es un ser vivo, etc. Entre ellos se establece la relación *es un*. Este tipo de relación es muy habitual: un guisante es una verdura, que es un tipo de vegetal; un disco duro es una unidad de almacenamiento, que a su vez es un tipo de componente de un ordenador.

Al decir que un elemento es un tipo de otro, establecemos una especialización: decimos que el elemento tiene unas características generales compartidas y otras propias.

La herencia es una manera de representar las características que se reciben del nivel más general.

La idea de perro hereda todas las características de mamífero; es decir, mama, respira con pulmones, se mueve, etc., pero también

ANOTACIONES

presenta unas características concretas propias como ladrar o mover la cola. A su vez, los perros también se pueden dividir según su raza: pastor alemán, caniche, doberman, etc. Cada uno con sus particularidades, pero heredando todas las características de los perros.

Para representar estas relaciones, C++ permite que una clase derive de otra. En nuestro caso, la clase Perro deriva de la clase Mamífero. Por tanto, en la clase Perro no será necesario indicarle que mama, ni que respira con pulmones, ni que se mueve. Al ser un mamífero, hereda dichas propiedades además de aportar datos o funciones nuevas.

De la misma forma, un Mamífero se puede implementar como una clase derivada de la clase Animal, y a su vez hereda información de dicha clase como la de pertenecer a los seres vivos y moverse.

Dada la relación entre la clase Perro y la clase Mamifero, y entre la clase Mamifero y la clase Animal, la clase Perro también heredará la información de la clase Animal: ¡Un perro es un ser vivo que se mueve!

Implementación de la herencia

Para expresar esta relación en C++, en la declaración de una clase después de su nombre se pone dos puntos (:), el tipo de derivación (pública o privada) y el nombre de la clase de la cual deriva:

```
class Perro : public Mamifero
```

El tipo de derivación, que de momento consideraremos pública, se estudiará con posterioridad. Ahora, enfocaremos nuestra atención sobre cómo queda la nueva implementación:

```
#include <iostream>

enum RAZAS { PASTOR_ALEMAN, CANICHE,
             DOBERMAN, YORKSHIRE };

class Mamifero
```

ANOTACIONES

```
{
   public:
    Mamifero();                // método constructor

    ~Mamifero();               // método destructor
   void asignarEdad(int nEdad)
    { edad = nEdad } ;         // métodos de acceso
   int obtenerEdad() const
    { return (edad) };
   protected:
    int edad;
};

class Perro : public Mamifero
{
   public:
    Perro();                   // método constructor

    ~Perro();                  // método destructor
    void asignarRaza(RAZAS);  // métodos de acceso
    int obtenerRaza() const;
    void ladrar() const
     { cout << "Guau "; };    // métodos propios
   private:
    RAZAS raza;
};

class Gato : public Mamifero
{
   public:
    Gato();   // método constructor
    ~Gato();  // método destructor
    void maullar() const
     { cout << "Miauuu"; } // métodos propios
};
```

En la implementación de la clase Mamífero, en primer lugar se han definido su constructor y destructor por defecto. Dado que el dato miembro `edad` que disponíamos en la clase Perro no es una característica exclusiva de esta clase, sino que todos los mamíferos tienen una edad, se ha trasladado el dato miembro `edad` y sus métodos de acceso (`obtenerEdad` y `asignarEdad`) a la nueva clase.

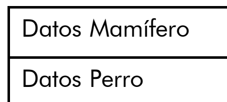
Nota

Cabe destacar que la declaración de tipo `protected` para el dato miembro `edad` permite que sea accesible desde las clases derivadas. Si se hubiera mantenido la declaración de `private`, no hubieran podido verlo ni utilizarlo otras clases, ni siquiera las derivadas. Si se hubiese declarado `public`, habría sido visible desde cualquier objeto, pero se recomienda evitar esta situación.

Dentro de la clase Perro, hemos añadido como dato nuevo su raza, y hemos definido sus métodos de acceso (`obtenerRaza` y `asignarRaza`), así como su constructor y destructor predefinido. Se continúa manteniendo el método `ladrar` como una función de la clase Perro: los demás mamíferos no ladran.

Constructores y destructores de clases derivadas

Al haber hecho la clase Perro derivada de la clase Mamífero, en esencia, los objetos `Perro` son objetos `Mamífero`. Por tanto, en primer lugar llama a su constructor base, con lo que se crea un Mamifero, y posteriormente se completa la información llamando al constructor de Perro.



A la hora de destruir un objeto de la clase Perro, el proceso es el inverso: en primer lugar se llama al destructor de Perro, liberando así su información específica, y, posteriormente, se llama al destructor de Mamifero.

Hasta el momento, sabemos inicializar los datos de un objeto de la clase que estamos definiendo, pero también es habitual que en el constructor de una clase se desee inicializar datos pertenecientes a su clase base.

El constructor de la clase Mamifero ya realiza esta tarea, pero es posible que sólo nos interese hacer esta operación para los perros y no

ANOTACIONES

Ejemplo

Siguiendo con el ejemplo, con la clase Perro, además de inicializar su raza, también podemos inicializar su edad (como es un mamífero, tiene una edad).

para todos los animales. En este caso, podemos realizar la siguiente inicialización en el constructor de la clase Perro:

```
Perro :: Perro()
{
   asignarRaza(CANICHE);   // Acceso a raza
   asignarEdad(3);         // Acceso a edad
};
```

Al ser `asignarEdad` un método público perteneciente a su clase base, ya lo reconoce automáticamente.

En el ejemplo anterior, hemos definido dos métodos `ladrar` y `maullar` para las clases Perro y Gato respectivamente. La gran mayoría de los animales tienen la capacidad de emitir sonidos para comunicarse; por tanto, se podría crear un método común que podríamos llamar `emitirSonido`, al cual podríamos dar un valor general para todos los animales, excepto para los perros y los gatos, caso en que se podría personalizar:

```
#include <iostream>

enum RAZAS {PASTOR_ALEMAN, CANICHE,
            DOBERMAN, YORKSHIRE };

class Mamifero
{
   public:
     Mamifero();                 // método constructor
                                 // método destructor
     ~Mamifero();
     void asignarEdad(int nEdad)
       { edad = nEdad; } ;       // métodos de acceso
     int obtenerEdad() const
       { return (edad); };
     void emitirSonido() const
       { cout << "Sonido"´; };
   protected:
     int edad;
};

class Perro : public Mamifero
```

ANOTACIONES

```
{
  public:
   Perro();                    // método constructor
   ~Perro();                   // método destructor
   void asignarRaza(RAZAS);// métodos de acceso
   int obtenerRaza() const;
   void emitirSonido() const
{ cout << "Guau"; };          // métodos propios
  private:
   RAZAS raza;
};

class Gato : public Mamifero
{
  public:
   Gato(); // método constructor
   ~Gato();// método destructor
   void emitirSonido () const
   { cout << "Miauuu"; }      // métodos propios
};

int main()
{
  Perro unperro;
  Gato ungato;
  Mamifero unmamifero;

  unmamifero.emitirSonido; // Resultado: "Sonido"
  unperro.emitirSonido;    // Resultado: Guau
  ungato.emitirSonido;     // Resultado: Miau
}
```

El método `emitirSonido` tendrá un resultado final según lo llame un Mamífero, un Perro o un Gato. En el caso de las clases derivadas (Perro y Gato) se dice que se ha **redefinido la función** miembro de la clase base. Para ello, la clase derivada debe definir la función base con la misma signatura (tipo de retorno, parámetros y sus tipos, y el especificador `const`).

ANOTACIONES

Hay que diferenciar la redefinición de funciones de la sobrecarga de funciones. En el primer caso, se trata de funciones con el mismo nombre y la misma signatura en clases diferentes (la clase base y la clase derivada). En el segundo caso, son funciones con el mismo nombre y diferente signatura, que están dentro de la misma clase.

El resultado de la redefinición de funciones es la **ocultación** de la función base para las clases derivadas. En este aspecto, hay que tener en cuenta que si se redefine una función en una clase derivada, quedarán ocultas también todas las sobrecargas de dicha función de la clase base. Un intento de usar alguna función ocultada generará un error de compilación. La solución consistirá en realizar en la clase derivada las mismas sobrecargas de la función existentes en la clase base.

No obstante, si se desea, todavía se puede acceder al método ocultado anteponiendo al nombre de la función el nombre de la clase base seguido del operador de ámbito (::).

```
unperro.Mamifero::emitirSonido();
```

4.4.3. El polimorfismo

En el ejemplo que se ha utilizado hasta el momento, sólo se ha considerado que la clase Perro (clase derivada) hereda los datos y métodos de la clase Mamifero (clase base). De hecho, la relación existente es más fuerte.

C++ permite el siguiente tipo de expresiones:

```
Mamifero *ap_unmamifero = new Perro;
```

En estas expresiones a un apuntador a una clase Mamifero no le asignamos directamente ningún objeto de la clase Mamifero, sino

ANOTACIONES

que le asignamos un objeto de una clase diferente, la clase Perro, aunque se cumple que Perro es una clase derivada de Mamifero.

De hecho, ésta es la naturaleza del polimorfismo: un mismo objeto puede adquirir diferentes formas. A un apuntador a un objeto mamifero se le puede asignar un objeto mamifero o un objeto de cualquiera de sus clases derivadas.

Además, como se podrá comprobar más adelante, esta asignación se podrá hacer en tiempo de ejecución.

Funciones virtuales

Con el apuntador que se presenta a continuación, se podrá llamar a cualquier método de la clase Mamifero. Pero lo interesante sería que, en este caso concreto, llamara a los métodos correspondientes de la clase Perro. Esto nos lo permiten las **funciones o métodos virtuales**:

```
#include <iostream>

enum RAZAS { PASTOR_ALEMAN, CANICHE,
             DOBERMAN, YORKSHIRE };

class Mamifero
{
   public:
    Mamifero();                       // método constructor
    virtual ~Mamifero();              // método destructor
    virtual void emitirSonido() const
      { cout << "emitir un sonido" << endl; };
   protected:
    int edad;
};

class Perro : public Mamifero
{
   public:
    Perro();                         // método constructor
    virtual ~Perro();                // método destructor
    int obtenerRaza() const;
```

ANOTACIONES

```
    virtual void emitirSonido() const
      { cout << "Guau" << endl; }; // métodos propios
  private:
    RAZAS raza;
};

class Gato : public Mamifero
{
  public:
    Gato();                        // método constructor
    virtual ~Gato();               // método destructor
    virtual void emitirSonido() const
      { cout << "Miau" << endl; }; // métodos propios
};

class Vaca : public Mamifero
{
  public:
    Vaca();                        // método constructor
    virtual ~Vaca();               // método destructor
    virtual void emitirSonido() const
      { cout << "Muuu" << endl; }; //métodos propios
};

int main()
{
  Mamifero * ap_mamiferos[3];
  int i;

  ap_mamiferos [0] = new Gato;
  ap_mamiferos [1] = new Vaca;
  ap_mamiferos [2] = new Perro;

  for (i=0; i<3 ; i++)
    ap_mamiferos[i] → emitirSonido();
return 0;
}
```

ANOTACIONES

Al ejecutar el programa, en primer lugar se declara un vector de 3 elementos tipo apuntador a Mamifero, y se inicializa a diferentes tipos de Mamifero (Gato, Vaca y Perro).

Posteriormente, se recorre este vector y se procede a llamar al método `emitirSonido` para cada uno de los elementos. La salida obtenida será:

- `Miau`
- `Muuu`
- `Guau`

El programa ha detectado en cada momento el tipo de objeto que se ha creado a través del `new` y ha llamado a la función `emitirSonido` correspondiente.

Esto hubiera funcionado igualmente aunque se le hubiera pedido al usuario que le indicara al programa el orden de los animales. El funcionamiento interno se basa en detectar en tiempo de ejecución el tipo del objeto al que se apunta y éste, en cierta forma, sustituye las funciones virtuales del objeto de la clase base por las que correspondan al objeto derivado.

Por todo ello, se ha definido la función miembro `emitirSonido` de la clase Mamifero como función virtual.

Declaración de las funciones virtuales

Al declarar una función de la clase base como virtual, implícitamente se están declarando como virtuales las funciones de las clases derivadas, por lo que no es necesaria su declaración explícita como tales. No obstante, para una mayor claridad en el código, se recomienda hacerlo.

Si la función no estuviera declarada como virtual, el programa entendería que debe llamar a la función de la clase base, independientemente del tipo de apuntador que fuera.

También es importante destacar que, en todo momento, se trata de apuntadores a la clase base (aunque se haya inicializado con un

ANOTACIONES

objeto de la clase derivada), con lo cual sólo pueden acceder a funciones de la clase base. Si uno de estos apuntadores intentara acceder a una función específica de la clase derivada, como por ejemplo `obtenerRaza()`, provocaría un error de función desconocida. A este tipo de funciones sólo se puede acceder directamente desde apuntadores a objetos de la clase derivada.

Constructores y destructores virtuales

Por definición, los constructores no pueden ser virtuales. En nuestro caso, al inicializar `new Perro` se llama al constructor de la clase Perro y al de la clase Mamifero, por lo que ya crea un apuntador a la clase derivada.

Al trabajar con estos apuntadores, una operación posible es su eliminación. Por tanto, nos interesará que, para su destrucción, primero se llame al destructor de la clase derivada y después al de la clase base. Para ello, nos basta con declarar el destructor de la clase base como virtual.

La regla práctica a seguir es declarar un destructor como virtual en el caso de que haya alguna función virtual dentro de la clase.

Tipos abstractos de datos y funciones virtuales puras

Ya hemos comentado que las clases corresponden al nivel de las ideas mientras que los objetos corresponden a elementos concretos.

De hecho, nos podemos encontrar clases en las que no tenga sentido instanciar ningún objeto, aunque sí que lo tuviera instanciarlos de clases derivadas. Es decir, clases que deseáramos mantener exclusivamente en el mundo de las ideas mientras que sus clases derivadas generaran nuestros objetos.

Un ejemplo podría ser una clase ObraDeArte, de la cual derivarán las subclases Pintura, Escultura, Literatura, Arquitectura, etc. Se podría considerar a la clase ObraDeArte como un concepto abstracto y cuando se tratara de obras concretas, referirnos a través de una de

ANOTACIONES

las variedades de arte (sus clases derivadas). El criterio para declarar una clase como tipo de datos abstracto siempre es subjetivo y dependerá del uso que se desea de las clases en la aplicación.

```
class ObraDeArte
{
public:
   ObraDeArte();
   virtual ~ObraDeArte ();
   virtual voidmostrarObraDeArte() = 0; //virtual pura
   void asignarAutor(String autor);
   String obtenerAutor();
   String autor;
};

class Pintura : public ObraDeArte
{
   public:
   Pintura();
   Pintura ( const Pintura & );
   virtual ~Pintura ();
   virtual void mostrarObraDeArte();
   void asignarTitulo(String titulo);
   String obtenerTitulo();
private:
   String titulo;
};

Pintura :: mostrarObraDeArte()
{ cout << "Fotografía Pintura \n" }
class Escultura : public ObraDeArte
{
public:
   Escultura();
   Escultura ( const Escultura & );
   virtual ~Escultura ();
   virtual void mostrarObraDeArte();
   void asignarTitulo(String titulo);
   String obtenerTitulo();
private:
   String titulo;
};

Escultura :: mostrarObraDeArte()
{ cout << "Fotografía Escultura \n" }
```

ANOTACIONES

Dentro de esta clase abstracta, se ha definido una función virtual que nos muestra una reproducción de la obra de arte. Esta reproducción varia según el tipo de obra. Podría ser en forma de fotografía, de video, de una lectura de un texto literario o teatral, etc.

Para declarar la clase ObraDeArte como un tipo de datos abstracto, y por tanto, no instanciable por ningún objeto sólo es necesario declarar una **función virtual pura**.

Para asignar una función virtual pura, basta con tomar una función virtual y asignarla a 0:

```
virtual void mostrarObraDeArte() = 0;
```

Ahora, al intentar instanciar un objeto ObraDeArte (mediante `new ObraDeArte`) daría error de compilación.

Al declarar funciones virtuales puras se debe tener en cuenta que esta función miembro también se hereda. Por tanto, en las clases derivadas se debe redefinir esta función. Si no se redefine, la clase derivada se convierte automáticamente en otra clase abstracta.

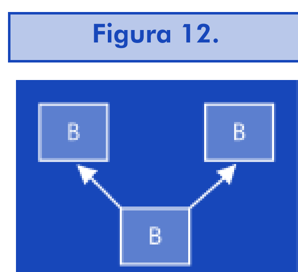
4.4.4. Operaciones avanzadas con herencia

A pesar de la potencia que se vislumbra en la herencia simple, hay situaciones en las que no es suficiente. A continuación, se presenta una breve introducción a los conceptos más avanzados relativos a la herencia.

Herencia múltiple

La herencia múltiple permite que una clase se derive de más de una clase base.

Figura 12.



ANOTACIONES

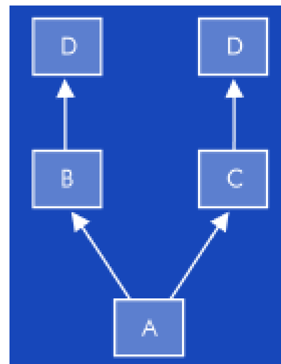
```
class A : public B, public C
```

En este ejemplo, la clase A se deriva de la clase B y de la clase C. Ante esta situación, surgen algunas preguntas:

- ¿Qué sucede cuando las dos clases derivadas tienen una función con el mismo nombre? Se podría producir un conflicto de ambigüedad para el compilador que se puede resolver añadiendo a la clase A una función virtual que redefina esta función, con lo que se resuelve explícitamente la ambigüedad.

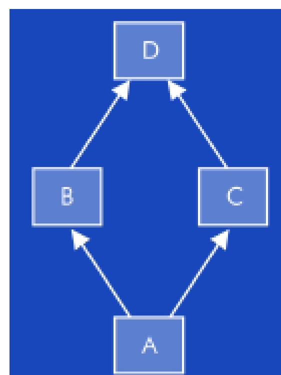
- ¿Qué sucede si las clases derivan de una clase base común? Como la clase A deriva de la clase D por parte de B y por parte de C, se producen dos copias de la clase D (ved la ilustración), lo cual puede provocar ambigüedades. La solución en este caso la proporciona la **herencia virtual**.

Figura 13.



Mediante la herencia virtual se indica al compilador que sólo se desea una clase base D compartida; para ello, las clases B y C se definen como virtuales.

Figura 14.



ANOTACIONES

```
class B: virtual D
class C: virtual D
class A : public B, public C
```

Generalmente, una clase inicializa sólo sus variables y su clase base. Al declarar una clase como virtual, el constructor que inicializa las variables corresponde al de la clase más derivada.

Herencia privada

A veces no es necesario, o incluso no se desea, que las clases derivadas tengan acceso a los datos o funciones de la clase base. En este caso se utiliza la herencia privada.

Con la herencia privada las variables y funciones miembro de la clase base se consideran como privadas, independientemente de la accesibilidad declarada en la clase base. Por tanto, para cualquier función que no sea miembro de la clase derivada son inaccesibles las funciones heredadas de la clase base.

4.4.5. Orientaciones para el análisis y diseño de programas

La complejidad de los proyectos de software actuales hace que se utilice la estrategia "divide y vencerás" para afrontar el análisis de un problema, descomponiendo el problema original en tareas más reducidas y más fácilmente abordables.

La forma tradicional para realizar este proceso se basa en descomponer el problema en funciones o procesos más simples (diseño descendente), de manera que se obtiene una estructura jerárquica de procesos y subprocesos.

Con la programación orientada a objetos, la descomposición se realiza de forma alternativa enfocando el problema hacia los objetos que lo componen y sus relaciones, y no hacia las funciones.

ANOTACIONES

El proceso a seguir es el siguiente:

- **Conceptualizar**. Los proyectos normalmente surgen de una idea que guía el desarrollo completo de éste. Es muy útil identificar el objetivo general que se persigue y velar porque se mantenga en las diferentes fases del proyecto.

- **Analizar**. Es decir, determinar las necesidades (requerimientos) que debe cubrir el proyecto. En esta fase, el esfuerzo se centra en comprender el dominio (el entorno) del problema en el mundo real (qué elementos intervienen y cómo se relacionan) y capturar los requerimientos.

  El primer paso para conseguir el análisis de requerimientos es identificar los **casos de uso** que son descripciones en lenguaje natural de los diferentes procesos del dominio. Cada caso de uso describe la interacción entre un actor (sea persona o elemento) y el sistema. El actor envía un mensaje al sistema y éste actúa consecuentemente (respondiendo, cancelando, actuando sobre otro elemento, etc.).

  Ejemplo
  Si se diseña una aplicación para cajeros automáticos, un caso de uso podría ser retirar dinero de la cuenta.

  A partir de un conjunto completo de casos de uso, se puede comenzar a desarrollar el **modelo del dominio**, el documento donde se refleja todo lo que se conoce sobre el dominio. Como parte de esta modelización, se describen todos los objetos que intervienen (que al final podrán llegar a corresponder a las clases del diseño).

  Ejemplo
  En el caso del proyecto del cajero automático, serían objetos el cliente, el cajero automático, el banco, el recibo, el dinero, la tarjeta de crédito, etc.

  El modelo se suele expresar en **UML** (lenguaje de modelado unificado), cuya explicación no es objetivo de esta unidad.

  Nota
  UML sólo es una convención comúnmente establecida para representar la información de un modelo.

  A partir de los casos de uso, podemos describir diferentes **escenarios**, circunstancias concretas en las cuales se desarrolla el caso de uso. De esta forma, se puede ir completando el conjunto de interacciones posibles que debe cumplir nuestro modelo. Cada escenario se caracteriza también en un entorno, con unas condiciones previas y elementos que lo activan.

  Ejemplo
  En el caso del proyecto del cajero automático, un posible escenario sería que el cliente deseara retirar el dinero de la cuenta y no hubiera fondos.

  Todos estos elementos se pueden representar gráficamente mediante diagramas que muestren estas interacciones.

Además, se deberán tener en cuenta las restricciones que suponen el entorno en que funcionará u otros requerimientos proporcionados por el cliente.

- **Diseñar**. A partir de la información del análisis, se enfoca el problema en crear la solución. Podemos considerar el diseño como la conversión de los requerimientos obtenidos en un modelo implementable en software. El resultado es un documento que contiene el plan del diseño.

En primer lugar, se debe identificar las clases que intervienen. Una primera (y simple) aproximación a la solución del problema consiste en escribir los diferentes escenarios, y crear una clase para cada sustantivo. Posteriormente, se puede reducir este número mediante la agrupación de los sinónimos.

Una vez definidas las clases del modelo, podemos añadir las clases que nos serán útiles para la implementación del proyecto (las vistas, los informes, clases para conversiones o manipulaciones de datos, uso de dispositivos, etc.).

Establecido el conjunto inicial de clases, que posteriormente se puede ir modificando, se puede proceder a modelar las relaciones e interacciones entre ellas. Uno de los puntos más importantes en la definición de una clase es determinar sus **responsabilidades**: un principio básico es que cada clase debe ser responsable de algo. Si se identifica claramente esta responsabilidad única, el código resultante será más fácil de mantener. Las responsabilidades que no corresponden a una clase, las delega a las clases relacionadas.

En esta fase también se establecen las **relaciones** entre los objetos del diseño que pueden coincidir, o no, con los objetos del análisis. Pueden ser de diferentes tipos. El tipo de relación que más se ha comentado en esta unidad son las relaciones de generalización que posteriormente se han implementado a partir de la herencia pública, pero hay otras cada una con sus formas de implementación.

La información del diseño se completa con la inclusión de la **dinámica** entre las clases: la modelación de la interacción de las clases entre ellas a través de diversos tipos de diagramas gráficos.

ANOTACIONES

El documento que recoge toda la información sobre el diseño de un prorgama se denomina *plan de diseño*.

- **Implementar**. Para que el proyecto se pueda aplicar, se debe convertir el plan de diseño a un código fuente, en nuestro caso, en C++. El lenguaje elegido proporciona las herramientas y mecánicas de trabajo para poder trasladar todas las definiciones de las clases, sus requerimientos, sus atributos y sus relaciones desde el mundo del diseño a nuestro entorno real. En esta fase nos centraremos en codificar de forma eficiente cada uno de los elementos del diseño.

- **Probar**. En esta fase se comprueba que el sistema realiza lo que se espera de él. Si no es así, se deben revisar las diferentes especificaciones a nivel de análisis, diseño o implementación. El diseño de un buen conjunto de pruebas basado en los casos de uso nos puede evitar muchos disgustos en el producto final. Siempre es preferible disponer de un buen conjunto de pruebas que provoque muchos fallos en las fases de análisis, diseño o implementación (y por tanto se pueden corregir), a encontrarse dichos errores en la fase de distribución.

- **Distribuir**. Se entrega al cliente una implementación del proyecto para su evaluación (del prototipo) o para su instalación definitiva.

Formas de desarrollo de un proyecto

Normalmente, el proceso descrito se lleva a cabo mediante un **proceso de cascada**: se va completando sucesivamente cada una de las etapas y cuando se ha finalizado y revisado, se pasa a la siguiente sin posibilidad de retroceder a la etapa anterior.

Este método, que en teoría parece perfecto, en la práctica es fatal. Por eso, en el análisis y diseño orientado a objetos se suele utilizar un **proceso iterativo**. En este proceso, a medida que se va desarrollando el software, se va repitiendo las etapas procediendo

ANOTACIONES

cada vez a un refinamiento superior, lo que permite adaptarlo a los cambios producidos por el mayor conocimiento del proyecto por parte de los diseñadores, de los desarrolladores y del mismo cliente.

Este método también proporciona otra ventaja en la vida real: se facilita la entrega en la fecha prevista de versiones completas aunque en un estado más o menos refinado. De alguna manera, permite introducir la idea de versiones "lo suficientemente buenas", y que posteriormente se pueden ir refinando según las necesidades del cliente.

4.5. Resumen

En esta unidad, hemos evolucionado desde un entorno de programación C que sigue el modelo imperativo, donde se toma como base de actuación las instrucciones y su secuencia, a un modelo orientado a objetos donde la unidad base son los objetos y su interrelación.

Para ello, hemos tenido que comprender las ventajas que supone utilizar un modelo de trabajo más abstracto, pero más cercano a la descripción de las entidades que se manejan en el mundo real y sus relaciones, y que nos permite enfocar nuestra atención en los conceptos que se desea implementar más que en el detalle final que suponen las líneas de código.

Posteriormente, hemos estudiado las herramientas que proporciona C++ para su implementación: las clases y los objetos. Además de su definición, también se han revisado las propiedades principales relativas al modelo de orientación a objetos que proporciona C++. En concreto, se han estudiado la herencia entre clases, la homonimia y el polimorfismo.

Finalmente, hemos visto que, debido al cambio de filosofía en el nuevo paradigma de programación, no se pueden aplicar los mismos principios para el diseño de los programas y por ello se han introducido nuevas reglas de diseño coherentes con él.

ANOTACIONES

4.6. Ejercicios de autoevaluación

1) Diseñad una aplicación que simule el funcionamiento de un ascensor. Inicialmente, la aplicación debe definir tres operaciones:

ASCENSOR: [1] Entrar [2] Salir [0] Finalizar

Después de cada operación, debe mostrar la ocupación del ascensor.

[1] Entrar corresponde a que una persona entra en el ascensor.
[2] Salir corresponde a que una persona sale del ascensor.

2) Ampliad el ejercicio anterior para incorporar los requerimientos siguientes:

- Una operación que sea dar el estado del ascensor

ASCENSOR: [1] Entrar [2] Salir [3] Estado [0] Finalizar

- Limitación de capacidad del ascensor a 6 plazas.

- Limitación de carga del ascensor a 500 kgs.

- Petición de código y peso de los usuarios para permitirles acceder al ascensor según los límites establecidos.

Si no se permite el acceso del usuario al ascensor, se le presenta un mensaje con los motivos:

– < El ascensor está completo >
– < El ascensor superaría la carga máxima autorizada >

Si se permite el acceso del usuario al ascensor, se le presenta el mensaje siguiente: < #Codigo# entra en el ascensor>.

Al entrar, como muestra de educación, saluda en general a las personas del ascensor (<#codigo# dice> Hola) si las hubiera, y éstas le corresponden el saludo de forma individual (<#codigo# responde> Hola).

Al salir un usuario del ascensor, se debe solicitar su código y actualizar la carga del ascensor al tiempo que se presenta el siguiente mensaje: <#codigo# sale del ascensor>.

Al salir, el usuario se despide de las personas del ascensor(<#codigo# dice> Adiós) si las hubiera, y éstas le corresponden el saludo de forma individual (<#codigo# responde> Adiós).

Para simplificar, consideraremos que no puede haber nunca dos pasajeros con el mismo código.

Después de cada operación, se debe poder mostrar el estado del ascensor (ocupación y carga).

3) Ampliad el ejercicio anterior incorporando tres posibles idiomas en los que los usuarios puedan saludar.

Al entrar, se debe solicitar también cuál es el idioma de la persona:

IDIOMA: [1] Catalán [2] Castellano [3] Inglés

- En catalán, el saludo es “Bon dia” y la despedida, “Adéu”.
- En castellano, el saludo es “Buenos días” y la despedida, “Adiós”.
- En inglés, el saludo es “Hello” y la despedida, “Bye”.

4.6.1. Solucionario

1)

ascensor01.hpp

```
class Ascensor {
  private:
    int ocupacion;
    int ocupacionmaxima;
  public:
    // Constructores y destructores
    Ascensor();
    ~Ascensor();

    // Funciones de acceso
```

ANOTACIONES

```
    void mostrarOcupacion();
    int obtenerOcupacion();
    void modificarOcupacion(int difOcupacion);

    // Funciones del método
    bool persona_puedeEntrar();
    bool persona_puedeSalir();

    void persona_entrar();
    void persona_salir();
};
```

ascensor01.cpp

```
#include <iostream>
#include "ascensor01.hpp"

// Constructores y destructores
Ascensor::Ascensor():
   ocupacion(0), ocupacionmaxima(6)
{ }

Ascensor::~Ascensor()
{ }

// Funciones de acceso
int Ascensor::obtenerOcupacion()
{ return (ocupacion); }

void Ascensor::modificarOcupacion(int difOcupacion)
{ ocupacion += difOcupacion; }

void Ascensor::mostrarOcupacion()
{ cout << "Ocupacion actual: " << ocupacion << endl;}

bool Ascensor::persona_puedeEntrar()
{ return (true); }

bool Ascensor::persona_puedeSalir()
{
   bool hayOcupacion;
   if (obtenerOcupacion() > 0) hayOcupacion = true;
```

ANOTACIONES

```
   else hayOcupacion = false;

   return (hayOcupacion);
}

void Ascensor::persona_entrar()
{ modificarOcupacion(1); }

void Ascensor::persona_salir()
{
   int ocupacionActual = obtenerOcupacion();
   if (ocupacionActual>0)
     modificarOcupacion(-1);
}
```

ejerc01.cpp

```
#include <iostream>
#include "ascensor01.hpp"

int main(int argc, char *argv[])
{
   char opc;
   bool salir = false;
   Ascensor unAscensor;
   do
   {
     cout << endl
     cout << "ASCENSOR: [1]Entrar  [2]Salir   [0]Finalizar ";
     cin >> opc;
     switch (opc)
     {
       case '1':
         cout << "opc Entrar" << endl;
         unAscensor.persona_entrar();
         break;
       case '2':
         cout << "opc Salir" << endl;
         if (unAscensor.persona_puedeSalir())
           unAscensor.persona_salir();
         else cout << "Ascensor vacio " << endl;
         break;
```

ANOTACIONES

```
      case '0':
        salir = true;
        break;
      }
      unAscensor.mostrarOcupacion();
    } while (! salir);
    return 0;
}
```

2)

ascensor02.hpp

```
#ifndef _ASCENSOR02
#define _ASCENSOR02
#include "persona02.hpp"

class Ascensor {
   private:
     int ocupacion;
     int carga;
     int ocupacionMaxima;
     int cargaMaxima;
     Persona *pasajeros[6];
   public:
     // Constructores y destructores
     Ascensor();
     ~Ascensor();

     // Funciones de acceso
     void mostrarOcupacion();
     int obtenerOcupacion();
     void modificarOcupacion(int difOcupacion);
     void mostrarCarga();
     int obtenerCarga();
     void modificarCarga(int difCarga);

     void mostrarListaPasajeros();

     // Funciones del método
     bool persona_puedeEntrar(Persona *);
```

ANOTACIONES

```
    bool persona_seleccionar(Persona *localizarPersona,
                             Persona **unaPersona);

    void persona_entrar(Persona *);
    void persona_salir(Persona *);

    void persona_saludarRestoAscensor(Persona *);
    void persona_despedirseRestoAscensor(Persona *);
};
#endif
```

ascensor 02.cpp

```
#include <iostream>
#include "ascensor02.hpp"

//
// Constructores y destructores
//

// En el constructor, inicializamos los valores máximos
// de ocupación y carga máxima de ascensor
// y el vector de pasajeros a apuntadores NULL

Ascensor::Ascensor():
   ocupacion(0), carga(0),
   ocupacionMaxima(6), cargaMaxima(500)
{ for (int i=0;i<=5;++i) {pasajeros[i]=NULL;} }

Ascensor::~Ascensor()
{ // Liberar codigos de los pasajeros
   for (int i=0;i<=5;++i)
   { if (!(pasajeros[i]==NULL)) {delete(pasajeros[i]);} }
}

// Funciones de acceso
int Ascensor::obtenerOcupacion()
{ return (ocupacion); }

void Ascensor::modificarOcupacion(int difOcupacion)
{ ocupacion += difOcupacion; }
```

ANOTACIONES

```
void Ascensor::mostrarOcupacion()
{ cout << "Ocupacion actual: " << ocupacion ; }

int Ascensor::obtenerCarga()
{ return (carga); }

void Ascensor::modificarCarga(int difCarga)
{ carga += difCarga; }

void Ascensor::mostrarCarga()
{ cout << "Carga actual: " << carga ; }

bool Ascensor::persona_puedeEntrar(Persona *unaPersona)
{
   bool tmpPuedeEntrar;

   // si la ocupación no sobrepasa el límite de ocupación y
   // si la carga no sobrepasa el limite de carga
   // → puede entrar

   if (ocupacion + 1 > ocupacionMaxima)
   {
    cout << " Aviso: Ascensor completo. No puede entrar. "
    cout << endl;
    return (false);
   }

   if (unaPersona→obtenerPeso() + carga > cargaMaxima)
   {
    cout << "Aviso: El ascensor supera su carga máxima.";
    cout << " No puede entrar. " << endl;
    return (false);
   }
   return (true);
}

bool Ascensor::persona_seleccionar(Persona *localizarPersona,
                                   Persona **unaPersona)
{
   int contador;
```

ANOTACIONES

```
   // Se debe seleccionar un pasajero del ascensor.
   bool personaEncontrada = false;
   if (obtenerOcupacion() > 0)
   {
     contador=0;
     do
     {
       if (pasajeros[contador]!=NULL)
       {
         if ((pasajeros[contador]→obtenerCodigo()==
             localizarPersona→obtenerCodigo() ))
         {
           *unaPersona=pasajeros[contador];
           personaEncontrada=true;
           break;
         }
       }
       contador++;
     } while (contador<ocupacionMaxima);
     if (contador>=ocupacionMaxima) {*unaPersona=NULL;}
   }
   return (personaEncontrada);
}

void Ascensor::persona_entrar(Persona *unaPersona)
{
   int contador;
   modificarOcupacion(1);
   modificarCarga(unaPersona->obtenerPeso());
   cout << unaPersona→obtenerCodigo();
   cout << " entra en el ascensor " << endl;
   contador=0;
   // hemos verificado anteriormente que hay plazas libres
   do
   {
     if (pasajeros[contador]==NULL )
     {
       pasajeros[contador]=unaPersona;
       break;
     }
```

ANOTACIONES

```
    contador++;
  } while (contador<ocupacionMaxima);
}

void Ascensor::persona_salir(Persona *unaPersona)
{
int contador;
  contador=0;
  do
  {
    if ((pasajeros[contador]==unaPersona ))
    {
      cout << unaPersona→obtenerCodigo();
      cout << " sale del ascensor " << endl;
      pasajeros[contador]=NULL;
      // Modificamos la ocupación y la carga
      modificarOcupacion(-1);
      modificarCarga(-1 * (unaPersona->obtenerPeso()));
      break;
    }
    contador++;
  } while (contador<ocupacionMaxima);
  if (contador == ocupacionMaxima)
  {cout << "Ninguna persona con este código. ";
    cout << "Nadie sale del ascensor" << endl;}
}

void Ascensor::mostrarListaPasajeros()
{
  int contador;
  Persona *unaPersona;

  if (obtenerOcupacion() > 0)
  {
    cout << "Lista de pasajeros del ascensor: " << endl;
    contador=0;
    do
    {
      if (!(pasajeros[contador]==NULL ))
      {
        unaPersona=pasajeros[contador];
```

ANOTACIONES

```
        cout << unaPersona->obtenerCodigo() << "; ";
      }
      contador++;
    } while (contador<ocupacionMaxima);
    cout << endl;
  }
  else
  { cout << "El ascensor esta vacío" << endl; }
}

void Ascensor::persona_saludarRestoAscensor( Persona *unaPersona)
{
  int contador;
  Persona *otraPersona;
  if (obtenerOcupacion() > 0)
  {
    contador=0;
    do
    {
      if (!(pasajeros[contador]==NULL ))
      {
        otraPersona=pasajeros[contador];
        if (!(unaPersona->obtenerCodigo()==
             otraPersona->obtenerCodigo()))
        {
          cout << otraPersona->obtenerCodigo();
          cout << " responde: " ;
          otraPersona->saludar();
          cout << endl;
        }
      }
    contador++;
    } while (contador<ocupacionMaxima);
  }
}

void Ascensor::persona_despedirseRestoAscensor( Persona *unaPersona)
{
  int contador;
  Persona *otraPersona;

  if (obtenerOcupacion() > 0)
```

ANOTACIONES

```
    {
      contador=0;
      do
      {
        if (!(pasajeros[contador]==NULL ))
        {
          otraPersona=pasajeros[contador];
          if (!(unaPersona→obtenerCodigo()==
                otraPersona→obtenerCodigo()))
          {
            cout << otraPersona->obtenerCodigo();
            cout << " responde: " ;
            otraPersona->despedirse();
            cout << endl;
          }
        }
        contador++;
      } while (contador<ocupacionMaxima);
    }
}
```

persona02.hpp

```
#ifndef _PERSONA02
#define _PERSONA02

class Persona
{
    private:
      int codigo;
      int peso;
    public:
      // Constructores
      Persona();
      Persona(int codigo, int peso);
      Persona(const Persona &);
      ~Persona();

      // Funciones de acceso
      int obtenerCodigo();
      void asignarCodigo(int);
      int obtenerPeso() const;
```

ANOTACIONES

```
    void asignarPeso(int nPeso);
    void asignarPersona(int);
    void asignarPersona(int,int);
    void solicitarDatos();
    void solicitarCodigo();

    void saludar();
    void despedirse();
};
#endif
```

persona02.cpp

```
#include <iostream>
#include "persona02.hpp"

Persona::Persona()
{ }
Persona::Persona(int nCodigo, int nPeso)
{
   codigo = nCodigo;
   peso = nPeso;
}

Persona::~Persona()
{ }

int Persona::obtenerPeso() const
{ return (peso); }

void Persona::asignarPeso(int nPeso)
{ peso = nPeso; }

int Persona::obtenerCodigo()
{ return (codigo); }

void Persona::asignarCodigo(int nCodigo)
{ codigo= nCodigo;}

void Persona::asignarPersona(int nCodigo)
{ this→codigo = nCodigo;}
```

ANOTACIONES

```
void Persona::asignarPersona(int nCodigo, int nPeso)
{
   asignarCodigo(nCodigo);
   asignarPeso(nPeso);
}

void Persona:: saludar()
{ cout << "Hola \n" ; };

void Persona:: despedirse ()
{ cout << "Adios \n" ; };

void Persona::solicitarCodigo()
{
   int nCodigo;
   cout << "Codigo: ";
   cin >> nCodigo;
   cout << endl;
   codigo = nCodigo;
}
```

ejerc02.cpp

```
#include <iostream>
#include "ascensor02.hpp"
#include "persona02.hpp"

void solicitarDatos(int *nCodigo, int *nPeso)
{
   cout << endl;
   cout << "Codigo: ";
   cin >> *nCodigo;
   cout << endl;
   cout << "Peso: ";
   cin >> *nPeso;
   cout << endl;
}

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

ANOTACIONES

```
bool salir = false;
Ascensor unAscensor;
Persona * unaPersona;
Persona * localizarPersona;

do
{
  cout << endl << "ASCENSOR: ";
  cout<<"[1]Entrar [2]Salir [3]Estado [0]Finalizar ";
  cin >> opc;
  switch (opc)
  {
   case '1': // opción Entrar
   {
    int nPeso;
    int nCodigo;

    solicitarDatos(&nCodigo, &nPeso);
    unaPersona = new Persona(nCodigo, nPeso);
    if (unAscensor.persona_puedeEntrar(unaPersona))
    {
     unAscensor.persona_entrar(unaPersona);
     if (unAscensor.obtenerOcupacion()>1)
     {
      cout << unaPersona->obtenerCodigo();
      cout << " dice: " ;
      unaPersona->saludar();
      cout << endl; // Ahora responden las demás
      unAscensor.persona_saludarRestoAscensor(unaPersona);
     }
    }
    break;
   }
   case '2': // opción Salir
   {
    unaPersona = NULL;
    localizarPersona = new Persona;
    localizarPersona->solicitarCodigo();
    if (unAscensor.persona_seleccionar ( localizarPersona,
                                          &unaPersona))
    {
```

ANOTACIONES

```
        unAscensor.persona_salir(unaPersona);
        if (unAscensor.obtenerOcupacion()>0)
        {
          cout << unaPersona->obtenerCodigo()
          cout << " dice: " ;
          unaPersona->despedirse();
          cout << endl; // Ahora responden las demás
          unAscensor.persona_despedirseRestoAscensor(unaPersona);
          delete (unaPersona);
        }
      }
      else
      {
        cout<<"No hay ninguna persona con este código";
        cout << endl;
      }
      delete localizarPersona;
      break;
    }
    case '3': //Estado
    {
      unAscensor.mostrarOcupacion();
      cout << " - "; // Para separar ocupación de carga
      unAscensor.mostrarCarga();
      cout << endl;
      unAscensor.mostrarListaPasajeros();
      break;
    }
    case '0':
    {
      salir = true;
      break;
    }
   }
  } while (! salir);
  return 0;
}
```

3) `ascensor03.hpp` y `ascensor03.cpp` coinciden con `ascensor02.hpp` y `ascensor02.cpp` del ejercicio 2.

ANOTACIONES

persona03.hpp

```
#ifndef _PERSONA03
#define _PERSONA03

class Persona
{
   private:
     int codigo;
     int peso;
   public:
     // Constructores
     Persona();
     Persona(int codigo, int peso);
     Persona(const Persona &);
     ~Persona();

     // Funciones de acceso
     int obtenerCodigo();
     void asignarCodigo(int);
     int obtenerPeso() const;
     void asignarPeso(int nPeso);
     void asignarPersona(int,int);
     void solicitarCodigo();

     virtual void saludar();
     virtual void despedirse();
};

class Catalan: public Persona
{
   public:
     Catalan()
     {
       asignarCodigo (0);
       asignarPeso (0);
     };

     Catalan(int nCodigo, int nPeso)
     {
       asignarCodigo (nCodigo);
       asignarPeso (nPeso);
     };
```

ANOTACIONES

```
    virtual void saludar()
    { cout << "Bon dia"; };

    virtual void despedirse()
    { cout << "Adeu"; };
};

class Castellano: public Persona
{
  public:
    Castellano()
    {
     asignarCodigo (0);
     asignarPeso (0);
    };

    Castellano(int nCodigo, int nPeso)
    {
     asignarCodigo (nCodigo);
     asignarPeso (nPeso);
    };


    virtual void saludar()
    { cout << "Buenos días"; };

    virtual void despedirse()
    { cout << "Adiós"; };
};

class Ingles : public Persona
{
  public:
    Ingles()
    {
     asignarCodigo (0);
     asignarPeso (0);
    };


    Ingles(int nCodigo, int nPeso)
    {
     asignarCodigo (nCodigo);
     asignarPeso (nPeso);
    };
```

ANOTACIONES

```
    virtual void saludar()
    { cout << "Hello"; };

    virtual void despedirse()
    { cout << "Bye"; };
};
#endif
```

persona03.cpp

```
#include <iostream>
#include "persona03.hpp"

Persona::Persona()
{ }

Persona::Persona(int nCodigo, int nPeso)
{
   codigo = nCodigo;
   peso = nPeso;
}

Persona::~Persona()
{ }

int Persona::obtenerPeso() const
{ return (peso); }

void Persona::asignarPeso(int nPeso)
{ peso = nPeso; }

int Persona::obtenerCodigo()
{ return (codigo); }

void Persona::asignarCodigo(int nCodigo)
{ this->codigo = nCodigo; }

void Persona::asignarPersona(int nCodigo, int nPeso)
{
   asignarCodigo(nCodigo);
   asignarPeso(nPeso);
}
```

ANOTACIONES

```
void Persona:: saludar()
{ cout << "Hola \n" ; };

void Persona:: despedirse ()
{ cout << "Adiós \n" ; };

void Persona::solicitarCodigo{
   int nCodigo;

   cout << "Codigo: ";
   cin >> nCodigo;
   cout << endl;

   asignarCodigo (nCodigo);
}
```

ejerc03.cpp

```
#include <iostream>
#include "ascensor03.hpp"
#include "persona03.hpp"

void solicitarDatos(int *nCodigo, int *nPeso, int *nIdioma)
{
   cout << endl;
   cout << "Codigo: ";
   cin >> *nCodigo;
   cout << endl;
   cout << "Peso: ";
   cin >> *nPeso;
   cout << endl;
   cout << "Idioma: [1] Catalán [2] Castellano [3] Inglés ";
   cin >> *nIdioma;
   cout << endl;
}

int main(int argc, char *argv[])
{
   char opc;
   bool salir = false;
   Ascensor unAscensor;
   Persona * unaPersona;
   Persona * localizarPersona;
```

ANOTACIONES

```
do
{
 cout << endl << "ASCENSOR: ";
 cout << "[1]Entrar [2]Salir [3]Estado [0]Finalizar";
 cin >> opc;
 switch (opc)
 {
  case '1': // Opción Entrar
  {
   int nPeso;
   int nCodigo;
   int nIdioma;
   solicitarDatos(&nCodigo, &nPeso, &nIdioma);
   switch (nIdioma)
   {
    case 1:
    {
     unaPersona = new Catalan(nCodigo, nPeso);
     break;
    }
    case 2:
    {
     unaPersona=new Castellano(nCodigo, nPeso);
     break;
    }
    case 3:
    {
     unaPersona = new Ingles(nCodigo, nPeso);
     break;
    }
   }
   if (unAscensor.persona_puedeEntrar(unaPersona))
   {
    unAscensor.persona_entrar(unaPersona);
    if (unAscensor.obtenerOcupacion()>1)
    {
     cout << unaPersona->obtenerCodigo();
     cout << " dice: " ;
     unaPersona->saludar();
     cout << endl; // Ahora responden las demás
     unAscensor.persona_saludarRestoAscensor (unaPersona);
    }
   }
```

ANOTACIONES

```
        break;
      }
      case '2': //Opción Salir
      {
        localizarPersona = new Persona;
        unaPersona = NULL;

        localizarPersona→solicitarCodigo();
        if (unAscensor.persona_seleccionar(localizarPersona,
            & unaPersona))
        {
          unAscensor.persona_salir(unaPersona);
          if (unAscensor.obtenerOcupacion()>0)
          {
            cout << unaPersona->obtenerCodigo();
            cout << " dice: " ;
            unaPersona->despedirse();
            cout << endl; // Ahora responden las demás
            unAscensor.persona_despedirseRestoAscensor (unaPersona);
            delete (unaPersona) ;
          }
        }
        else
        {
          cout<<"No hay ninguna persona con este código";
          cout << endl;
        }
        delete localizarPersona;
        break;
      }
      case '3': //Estado
      {
        unAscensor.mostrarOcupacion();
        cout << " - "; // Para separar Ocupacion de Carga
        unAscensor.mostrarCarga();
        cout << endl;
        unAscensor.mostrarListaPasajeros();
        break;
      }
      case '0':
```

ANOTACIONES

```
      {
        salir = true;
        break;
      }
    }
  } while (! salir);
  return 0;
}
```

Unidad 5. Programación en Java

5.1. Introducción

En las unidades anteriores, se ha mostrado la evolución que han experimentado los lenguajes de programación en la historia y que han ido desembocando en los diferentes paradigmas de programación.

Inicialmente, el coste de un sistema informático estaba marcado principalmente por el hardware: los componentes internos de los ordenadores eran voluminosos, lentos y caros. En comparación, el coste que generaban las personas que intervenían en su mantenimiento y en el tratamiento de la información era casi despreciable. Además, por limitaciones físicas, el tipo de aplicaciones que se podían manejar eran más bien simples. El énfasis en la investigación en informática se centraba básicamente en conseguir sistemas más pequeños, más rápidos y más baratos.

Con el tiempo, esta situación ha cambiado radicalmente. La revolución producida en el mundo del hardware ha permitido la fabricación de ordenadores en los que no se podía ni soñar hace 25 años, pero esta revolución no ha tenido su correspondencia en el mundo del software. En este aspecto, los costes materiales se han reducido considerablemente mientras que los relativos a personal han aumentado progresivamente. También se ha incrementado la complejidad en el uso del software, entre otras cosas debido al aumento de interactividad con el usuario.

En la actualidad muchas de las líneas de investigación buscan mejorar el rendimiento en la fase de desarrollo de software donde, de momento, la intervención humana es fundamental. Mucho de este esfuerzo se centra en la generación de código correcto y en la reutilización del trabajo realizado.

En este camino, el paradigma de la programación orientada a objetos ha supuesto una gran aproximación entre el proceso de desarrollo de

ANOTACIONES

aplicaciones y la realidad que intentan representar. Por otro lado, la incorporación de la informática en muchos componentes que nos rodean también ha aumentado en gran medida el número de plataformas diversas sobre las cuales es posible desarrollar programas.

Java es un lenguaje moderno que ha nacido para dar solución a este nuevo entorno. Básicamente, es un lenguaje orientado a objetos pensado para trabajar en múltiples plataformas. Su planteamiento consiste en crear una plataforma común intermedia para la cual se desarrollan las aplicaciones y, después, trasladar el resultado generado para dicha plataforma común a cada máquina final.

Este paso intermedio permite:

– Escribir la aplicación **sólo una vez**. Una vez compilada hacia esta plataforma común, la aplicación podrá ser ejecutada por todos los sistemas que dispongan de dicha plataforma intermedia.

– Escribir la plataforma común **sólo una vez**. Al conseguir que una máquina real sea capaz de ejecutar las instrucciones de dicha plataforma común, es decir, que sea capaz de trasladarlas al sistema subyacente, se podrán ejecutar en ella todas las aplicaciones desarrolladas para dicha plataforma.

Por tanto, se consigue el máximo nivel de reutilización. El precio es el sacrificio de parte de la velocidad.

En el orden de la generación de código correcto, Java dispone de varias características que se irán viendo a lo largo de esta unidad. En todo caso, de momento se desea destacar que Java se basa en C++, con lo cual se consigue mayor facilidad de aprendizaje para gran número de desarrolladores (reutilización del conocimiento), pero se le libera de muchas de las cadenas que C++ arrastraba por su compatibilidad con el C.

Esta “limpieza” tiene consecuencias positivas:

– El lenguaje es más simple, pues se eliminan conceptos complejos raras veces utilizados.

ANOTACIONES

– El lenguaje es más directo. Se ha estimado que Java permite reducir el número de líneas de código a la cuarta parte.

– El lenguaje es más puro, pues sólo permite trabajar en el paradigma de la orientación a objetos.

Además, la juventud del lenguaje le ha permitido incorporar dentro de su núcleo algunas características que sencillamente no existían cuando se crearon otros lenguajes, como las siguientes:

– La programación de hilos de ejecución (*threads*), que permite aprovechar las arquitecturas con multiprocesadores.

– La programación de comunicaciones (TCP/IP, etc.) que facilita el trabajo en red, sea local o Internet.

– La programación de *applets*, miniaplicaciones pensadas para ser ejecutadas por un navegador web.

– El soporte para crear interfaces gráficas de usuario y un sistema de gestión de eventos, que facilitan la creación de aplicaciones siguiendo el paradigma de la programación dirigida por eventos.

En esta unidad se desea introducir al lector en este nuevo entorno de programación y presentar sus principales características, y se pretende que, partiendo de sus conocimientos del lenguaje C++, alcance los objetivos siguientes:

1) Conocer el entorno de desarrollo de Java.

2) Ser capaz de programar en Java.

3) Entender los conceptos del uso de los hilos de ejecución y su aplicación en el entorno Java.

4) Comprender las bases de la programación dirigida por eventos y ser capaz de desarrollar ejemplos simples.

5) Poder crear *applets* simples.

ANOTACIONES

5.2. Origen de Java

En 1991, ingenieros de Sun Microsystems intentaban introducirse en el desarrollo de programas para electrodomésticos y pequeños equipos electrónicos donde la potencia de cálculo y memoria era reducida. Ello requería un lenguaje de programación que, principalmente, aportara fiabilidad del código y facilidad de desarrollo, y pudiera adaptarse a múltiples dispositivos electrónicos.

Nota

Por la variedad de dispositivos y procesadores existentes en el mercado y sus continuos cambios buscaban un entorno de trabajo que no dependiera de la máquina en la que se ejecutara.

Para ello diseñaron un esquema basado en una plataforma intermedia sobre la cual funcionaría un nuevo código máquina ejecutable, y esta plataforma se encargaría de la traslación al sistema subyacente. Este código máquina genérico estaría muy orientado al modo de funcionar de la mayoría de dichos dispositivos y procesadores, por lo cual la traslación final había de ser rápida.

El proceso completo consistiría, pues, en escribir el programa en un lenguaje de alto nivel y compilarlo para generar código genérico (los *bytecodes*) preparado para ser ejecutado por dicha plataforma (la "máquina virtual"). De este modo se conseguiría el objetivo de poder escribir el código una sola vez y poder ejecutarlo en todas partes donde estuviera disponible dicha plataforma (*Write Once, Run EveryWhere*).

Teniendo estas referencias, su primer intento fue utilizar C++, pero por su complejidad surgieron numerosas dificultades, por lo que decidieron diseñar un nuevo lenguaje basándose en C++ para facilitar su aprendizaje. Este nuevo lenguaje debía recoger, además, las propiedades de los lenguajes modernos y reducir su complejidad eliminando aquellas funciones no absolutamente imprescindibles.

El proyecto de creación de este nuevo lenguaje recibió el nombre inicial de *Oak*, pero como el nombre estaba registrado, se rebautizó

ANOTACIONES

con el nombre final de Java. Consecuentemente, la máquina virtual capaz de ejecutar dicho código en cualquier plataforma recibió el nombre de máquina virtual de Java (JVM - *Java virtual machine*).

Los primeros intentos de aplicación comercial no fructificaron, pero el desarrollo de Internet fomentó tecnologías multiplataforma, por lo que Java se reveló como una posibilidad interesante para la compañía. Tras una serie de modificaciones de diseño para adaptarlo, Java se presentó por primera vez como lenguaje para ordenadores en el año 1995, y en enero de 1996, Sun formó la empresa Java Soft para desarrollar nuevos productos en este nuevo entorno y facilitar la colaboración con terceras partes. El mismo mes se dio a conocer una primera versión, bastante rudimentaria, del kit de desarrollo de Java, el JDK 1.0.

A principios de 1997 apareció la primera revisión Java, la versión 1.1, mejorando considerablemente las prestaciones del lenguaje, y a finales de 1998 apareció la revisión Java 1.2, que introdujo cambios significativos. Por este motivo, a esta versión y posteriores se las conoce como plataformas Java 2. En diciembre del 2003, la última versión de la plataforma Java2 disponible para su descarga en la página de Sun es Java 1.4.2.

Nota

Podéis encontrar esta versión en la dirección siguiente: http://java.sun.com

La verdadera revolución que impulsó definitivamente la expansión del lenguaje la causó la incorporación en 1997 de un intérprete de Java en el navegador Netscape.

5.3. Características generales de Java

Sun Microsystems describe Java como un lenguaje simple, orientado a objetos, distribuido, robusto, seguro, de arquitectura neutra, portable, interpretado, de alto rendimiento, multitarea y dinámico.

Analicemos esta descripción:

- **Simple** . Para facilitar el aprendizaje, se consideró que los lenguajes más utilizados por los programadores eran el C y el C++.

ANOTACIONES

Descartado el C++, se diseñó un nuevo lenguaje que fuera muy cercano a él para facilitar su comprensión.

Con este objetivo, Java elimina una serie de características poco utilizadas y de difícil comprensión del C++, como, por ejemplo, la herencia múltiple, las coerciones automáticas y la sobrecarga de operadores.

- **Orientado a objetos**. En pocas palabras, el diseño orientado a objetos enfoca el diseño hacia los datos (objetos), sus funciones e interrelaciones (métodos). En este punto, se siguen esencialmente los mismos criterios que C++.

- **Distribuido**. Java incluye una amplia librería de rutinas que permiten trabajar fácilmente con los protocolos de TCP/IP como HTTP o FTP. Se pueden crear conexiones a través de la red a partir de direcciones URL con la misma facilidad que trabajando en forma local.

- **Robusto**. Uno de los propósitos de Java es buscar la fiabilidad de los programas. Para ello, se puso énfasis en tres frentes:

– Estricto control en tiempo de compilación con el objetivo de detectar los problemas lo antes posible. Para ello, utiliza una estrategia de fuerte control de tipos, como en C++, aunque evitando algunos de sus agujeros normalmente debidos a su compatibilidad con C. También permite el control de tipos en tiempo de enlace.

– Chequeo en tiempo de ejecución de los posibles errores dinámicos que se pueden producir.

– Eliminación de situaciones propensas a generar errores. El caso más significativo es el control de los apuntadores. Para ello, los trata como vectores verdaderos, controlando los valores posibles de índices. Al evitar la aritmética de apuntadores (sumar desplazamiento a una posición de memoria sin controlar sus límites) se evita la posibilidad de sobreescritura de memoria y corrupción de datos.

ANOTACIONES

- **Seguro**. Java está orientado a entornos distribuidos en red y, por este motivo, se ha puesto mucho énfasis en la seguridad contra virus e intrusiones, y en la autenticación.

- **Arquitectura neutra**. Para poder funcionar sobre variedad de procesadores y arquitecturas de sistemas operativos, el compilador de Java proporciona un código común ejecutable desde cualquier sistema que tenga la presencia de un sistema en tiempo de ejecución de Java.

  Esto evita que los autores de aplicaciones deban producir versiones para sistemas diferentes (como PC, Apple Macintosh, etc.). Con Java, el mismo código compilado funciona para todos ellos.

  Para ello, Java genera instrucciones *bytecodes* diseñadas para ser fácilmente interpretadas por una plataforma intermedia (la máquina virtual de Java) y traducidas a cualquier código máquina nativo al vuelo.

- **Portable**. La arquitectura neutra ya proporciona un gran avance respecto a la portabilidad, pero no es el único aspecto que se ha cuidado al respecto.

  Ejemplo

  En Java no hay detalles que dependan de la implementación, como podría ser el tamaño de los tipos primitivos. En Java, a diferencia de C o C++, el tipo `int` siempre se refiere a un número entero de 32 bits con complemento a 2 y el tipo `float` un número de 32 bits siguiendo la norma IEEE 754.

  La portabilidad también viene dada por las librerías. Por ejemplo, hay una clase Windows abstracta y sus implementaciones para Windows, Unix o Macintosh.

- **Interpretado**. Los *bytecodes* en Java se traducen en tiempo de ejecución a instrucciones de la máquina nativa (son interpretadas) y no se almacenan en ningún lugar.

- **Alto rendimiento.** A veces se requiere mejorar el rendimiento producido por la interpretación de los *bytecodes*, que ya es bas-

ANOTACIONES

tante bueno de por sí. En estos casos, es posible traducirlos en tiempo de ejecución al código nativo de la máquina donde la aplicación se está ejecutando. Esto es, compilar el lenguaje de la JVM al lenguaje de la máquina en la que se haya de ejecutar el programa.

Por otro lado, los *bytecodes* se han diseñado pensando en el código máquina por lo que el proceso final de la generación de código máquina es muy simple. Además, la generación de los *bytecodes* es eficiente y se le aplican diversos procesos de optimización.

- **Multitarea**. Java proporciona dentro del mismo lenguaje herramientas para construir aplicaciones con múltiples hilos de ejecución, lo que simplifica su uso y lo hace más robusto.

- **Dinámico**. Java se diseñó para adaptarse a un entorno cambiante. Por ejemplo, un efecto lateral del C++ se produce debido a la forma en la que el código se ha implementado. Si un programa utiliza una librería de clases y ésta cambia, hay que recompilar todo el proyecto y volverlo a redistribuir. Java evita estos problemas al hacer las interconexiones entre los módulos más tarde, permitiendo añadir nuevos métodos e instancias sin tener ningún efecto sobre sus clientes.

Mediante las interfaces se especifican un conjunto de métodos que un objeto puede realizar, pero deja abierta la manera como los objetos pueden implementar estos métodos. Una clase Java puede implementar múltiples interfaces, aunque sólo puede heredar de una única clase. Las interfaces proporcionan flexibilidad y reusabilidad conectando objetos según lo que queremos que hagan y no por lo que hacen.

Las clases en Java se representan en tiempo de ejecución por una clase llamada Class, que contiene las definiciones de las clases en tiempo de ejecución. De este modo, se pueden hacer comprobaciones de tipo en tiempo de ejecución y se puede confiar en los tipos en Java, mientras que en C++ el compilador solo confía en que el programador hace lo correcto.

ANOTACIONES

5.4. El entorno de desarrollo de Java

Para desarrollar un programa en Java, existen diversas opciones comerciales en el mercado. No obstante, la compañía Sun distribuye de forma gratuita el Java Development Kit (JDK) que es un conjunto de programas y librerías que permiten el desarrollo, compilación y ejecución de aplicaciones en Java además de proporcionar un *debugger* para el control de errores.

También existen herramientas que permiten la integración de todos los componentes anteriores (IDE – *integrated development environment*), de utilización más agradable, aunque pueden presentar fallos de compatibilidad entre plataformas o ficheros resultantes no tan optimizados. Por este motivo, y para familiarizarse mejor con todos los procesos de la creación de software, se ha optado en este material por desarrollar las aplicaciones directamente con las herramientas proporcionadas por Sun.

Nota

Entre los IDEs disponibles actualmente se puede destacar el proyecto Eclipse, que, siguiendo la filosofía de código abierto, ha conseguido un paquete de desarrollo muy completo (SDK .- *standard development kit*) para diversos sistemas operativos (Linux, Windows, Sun, Apple, etc.).

Este paquete está disponible para su descarga en http://www.eclipse.org.

Otro IDE interesante es JCreator, que además de desarrollar una versión comercial, dispone de una versión limitada, de fácil manejo.

Este paquete está disponible para su descarga en http://www.jcreator.com.

Otra característica particular de Java es que se pueden generar varios tipos de aplicaciones:

- **Aplicaciones independientes**. Un fichero que se ejecuta directamente sobre la máquina virtual de la plataforma.

ANOTACIONES

- *Applets*. Miniaplicaciones que no se pueden ejecutar directamente sobre la máquina virtual, sino que están pensadas para ser cargadas y ejecutadas desde un navegador web. Por este motivo, incorpora unas limitaciones de seguridad extremas.

- *Servlets*. Aplicaciones sin interfaz de usuario para ejecutarse desde un servidor y cuya función es dar respuesta a las acciones de navegadores remotos (petición de páginas HTML, envío de datos de un formulario, etc.). Su salida generalmente es a través de ficheros, como por ejemplo, ficheros HTML.

Para generar cualquiera de los tipos de aplicaciones anteriores, sólo se precisa lo siguiente:

- Un editor de textos donde escribir el código fuente en lenguaje Java.

- La plataforma Java, que permite la compilación, depurado, ejecución y documentación de dichos programas.

5.4.1. La plataforma Java

Entendemos como plataforma el entorno hardware o software que necesita un programa para ejecutarse. Aunque la mayoría de plataformas se pueden describir como una combinación de sistema operativo y hardware, la plataforma Java se diferencia de otras en que se compone de una plataforma software que funciona sobre otras plataformas basadas en el hardware (GNU/Linux, Solaris, Windows, Macintosh, etc.).

La plataforma Java tiene dos componentes:

- Máquina virtual (MV). Como ya hemos comentado, una de las principales características que proporciona Java es la independencia de la plataforma hardware: una vez compilados, los programas se deben poder ejecutar en cualquier plataforma.

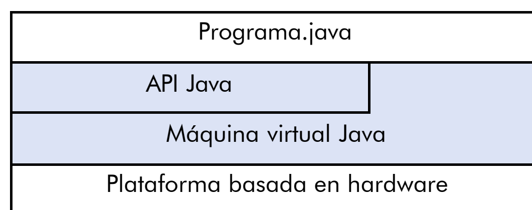
La estrategia utilizada para conseguirlo es generar un código ejecutable "neutro" (*bytecode*) como resultado de la compilación. Este código neutro, que está muy orientado al código máquina, se ejecuta

ANOTACIONES

desde una “máquina hipotética” o “máquina virtual”. Para ejecutar un programa en una plataforma determinada basta con disponer de una “máquina virtual” para dicha plataforma.

- *Application programming interface* (API). El API de Java es una gran colección de software ya desarrollado que proporciona múltiples capacidades como entornos gráficos, comunicaciones, multiproceso, etc. Está organizado en librerías de clases relacionadas e interfaces. Las librerías reciben el nombre de *packages*.

En el siguiente esquema, se puede observar la estructura de la plataforma Java y como la máquina virtual aísla el código fuente (.java) del hardware de la máquina:



5.4.2. Mi primer programa en Java

Otra vez nuestro primer contacto con el lenguaje será mostrando un saludo al mundo. El desarrollo del programa se dividirá en tres fases:

1) **Crear un fichero fuente**. Mediante el editor de textos escogido, escribiremos el texto y lo salvaremos con el nombre *HolaMundo.java*.

HolaMundo.java

```
/**
 * La clase HolaMundo muestra el mensaje
 * "Hola Mundo" en la salida estándar.
 */
public class HolaMundo {
    public static void main(String[] args) {
      // Muestra "Hola Mundo!"
      System.out.println("¡Hola Mundo!");
    }
}
```

ANOTACIONES

2) **Compilar el programa** generando un fichero *bytecode*. Para ello, utilizaremos el compilador **javac**, que nos proporciona el entorno de desarrollo, y que traduce el código fuente a instrucciones que la JVM pueda interpretar.

   Si después de teclear "`javac HolaMundo.java`" en el intérprete de comandos, no se produce ningún error, obtenemos nuestro primer programa en Java: un fichero *HolaMundo.class*.

3) **Ejecutar el programa** en la máquina virtual de Java. Una vez generado el fichero de *bytecodes*, para ejecutarlo en la JVM sólo deberemos escribir la siguiente instrucción, para que nuestro ordenador lo pueda interpretar, y nos aparecerá en pantalla el mensaje de bienvenida ¡Hola mundo!:

   ```
   java HolaMundo
   ```

5.4.3. Las instrucciones básicas y los comentarios

En este punto, Java continua manteniéndose fiel a C++ y C y conserva su sintaxis.

La única consideración a tener en cuenta es que, en Java, las expresiones condicionales (por ejemplo, la condición `if`) deben retornar un valor de tipo *boolean*, mientras que C++, por compatibilidad con C, permitía el retorno de valores numéricos y asimilaba 0 a *false* y los valores distintos de 0 a *true*.

Respecto a los comentarios, Java admite las formas provenientes de C++ (/* ... */ y // ...) y añade una nueva: incluir el texto entre las secuencias /** (inicio de comentario) y */ (fin de comentario).

De hecho, la utilidad de esta nueva forma no es tanto la de comentar, sino la de documentar. Java proporciona herramientas (por ejemplo, *javadoc*) para generar documentación a partir de los códigos fuentes que extraen el contenido de los comentarios realizados siguiendo este modelo.

Ejemplo

```
/**
*  Texto comentado con la nueva forma de Java para su
*  inclusión en documentación generada automáticamente
*/
```

ANOTACIONES

5.5. Diferencias entre C++ y Java

Como se ha comentado, el lenguaje Java se basó en C++ para proporcionar un entorno de programación orientado a objetos que resultará muy familiar a un gran número de programadores. Sin embargo, Java intenta mejorar C++ en muchos aspectos y, sobre todo, elimina aquellos que permitían a C++ trabajar de forma "no orientada a objetos" y que fueron incorporados por compatibilidad con el lenguaje C.

5.5.1. Entrada/salida

Como Java está pensado principalmente para trabajar de forma gráfica, las clases que gestionan la entrada / salida en modo texto se han desarrollado de manera muy básica. Están reguladas por la clase System que se encuentra en la librería java.lang, y de esta clase se destacan tres objetos estáticos que son los siguientes:

- **`System.in`**. Recibe los datos desde la entrada estándar (normalmente el teclado) en un objeto de la clase InputStream (flujo de entrada).

- **`System.out`**. Imprime los datos en la salida estándar (normalmente la pantalla) un objeto de la clase OutputStream (flujo de salida).

- **`System.err`**. Imprime los mensajes de error en pantalla.

Los métodos básicos de que disponen estos objetos son los siguientes:

- **`System.in.read()`**. Lee un carácter y lo devuelve en forma de entero.

- **`System.out.print(var)`**. Imprime una variable de cualquier tipo primitivo.

- **`System.out.println(var)`**. Igual que el anterior pero añadiendo un salto de línea final.

Por tanto, para escribir un mensaje nos basta utilizar básicamente las instrucciones `System.out.print()` y `System.out.println()`:

```
int unEntero = 35;
double unDouble = 3.1415;

System.out.println("Mostrando un texto");
System.out.print("Mostrando un entero ");
System.out.println (unEntero);
System.out.print("Mostrando un double ");
System.out.println (unDouble);
```

Mientras que la salida de datos es bastante natural, la entrada de datos es mucho menos accesible pues el elemento básico de lectura es el carácter. A continuación se presenta un ejemplo en el que se puede observar el proceso necesario para la lectura de una cadena de caracteres:

```
String miVar;
InputStreamReader isr = new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(isr);
// La entrada finaliza al pulsar la tecla Entrar
miVar = br.readLine();
```

Si se desea leer líneas completas, se puede hacer a través del objeto `BufferedReader`, cuyo método `readLine()` llama a un lector de caracteres (un objeto `Reader`) hasta encontrar un símbolo de final de línea ("\n" o "\r"). Pero en este caso, el flujo de entrada es un objeto `InputStream`, y no tipo `Reader`. Entonces, necesitamos una clase que actúe como lectora para un flujo de datos `InputStream`. Será la clase `InputStreamReader`.

No obstante, el ejemplo anterior es válido para *Strings*. Cuando se desea leer un número entero u otros tipos de datos, una vez realizada la lectura, se debe hacer la conversión. Sin embargo, esta conversión puede llegar a generar un error fatal en el sistema si el texto introducido no coincide con el tipo esperado. En este caso, Java nos obliga a considerar siempre dicho control de errores. La gestión de errores (que provocan las llamadas excepciones) se hace, igual que en C++, a través de la sentencia `try {... } catch {...} finally {...}`.

ANOTACIONES

A continuación, veremos cómo se puede diseñar una clase para que devuelva un número entero leído desde teclado:

```
Leer.java
import java.io.*;
public class Leer
{
   public static String getString()
   {
     String str = "";
     try
     {
      InputStreamReader isr = new
        InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      str = br.readLine();
     }
     catch(IOException e)
     {
      System.err.println("Error: " + e.getMessage());
     }
     return str; // devolver el dato tecleado
   }

   public static int getInt()
     {
      try
      {
        return Integer.parseInt(getString());
      }
      catch(NumberFormatException e)
      {
        return Integer.MIN_VALUE; // valor más pequeño
      }
   }
  // getInt
   // se puede definir una función para cada tipo...
   public static double getDouble() {} // getDouble

}
 // Leer
```

En el bloque `try { ... }` se incluye el trozo de código susceptible de sufrir un error. En caso de producirse, se lanza una excepción que es recogida por el bloque `catch { ... }`.

En el caso de la conversión de tipos `string` a números, la excepción que se puede producir es del tipo `NumberFormatException`. Podría haber más bloques `catch` para tratar diferentes tipos de excepción. En el ejemplo, si se produce error el valor numérico devuelto corresponde al mínimo valor posible que puede tomar un número entero.

El bloque `finally { ... }` corresponde a un trozo de código a ejecutar tanto si ha habido error, como si no (por ejemplo, cerrar ficheros), aunque su uso es opcional.

De forma similar, se pueden desarrollar funciones para cada uno de los tipos primitivos de Java. Finalmente, la lectura de un número entero sería como sigue:

```
int i;
...
i = Leer.getInt( );
```

5.5.2. El preprocesador

Java no dispone de preprocesador, por lo que diferentes órdenes (generalmente, originarias de C) se eliminan. Entre éstas, las más conocidas son las siguientes:

- `defines`. Estas órdenes para la definición de constantes, ya en C++ habían perdido gran parte de su sentido al poder declarar variables `const`, y ahora se implementan a partir de las variables `final`.

- `include`. Esta orden, que se utilizaba para incluir el contenido de un fichero, era muy útil en C++, principalmente para la reutilización de los ficheros de cabeceras. En Java, no hay ficheros de cabecera y las librerías (o paquetes) se incluyen mediante la instrucción `import`.

ANOTACIONES

5.5.3. La declaración de variables y constantes

La declaración de variables se mantiene igual, pero la definición de constantes cambia de forma: en Java, se antecede la variable con la palabra reservada `final`; no es necesario asignarle un valor en el momento de la declaración. No obstante, en el momento en que se le asigne un valor por primera vez, ya no puede ser modificado.

```
final int i;
int j = 2;
...
i=j + 2; // asignado el valor, no se podrá modificar
```

5.5.4. Los tipos de datos

Java clasifica los tipos de datos en dos categorías: primitivos y referencias. Mientras el primero contiene el valor, el segundo sólo contiene la dirección de memoria donde está almacenada la información.

Los tipos primitivos de datos de Java (`byte`, `short`, `int`, `long`, `float`, `double`, `char` y `boolean`) básicamente coinciden con los de C++, aunque con algunas modificaciones, que presentamos a continuación:

- Los tipos numéricos tienen el mismo tamaño independientemente de la plataforma en que se ejecute.
- Para los tipos numéricos no existe el especificador `unsigned`.
- El tipo `char` utiliza el conjunto de caracteres Unicode, que tiene 16 bits. Los caracteres del 0 al 127 coinciden con los códigos ASCII.
- Si no se inicializa las variables explícitamente, Java inicializa los datos a cero (o a su equivalente) automáticamente eliminando así los valores basura que pudieran contener.

Los tipos referencia en Java son los vectores, clases e interfaces. Las variables de estos tipos guardan su dirección de memoria, lo que po-

ANOTACIONES

dría asimilarse a los apuntadores en otros lenguajes. No obstante, al no permitir las operaciones explícitas con las direcciones de memoria, para acceder a ellas bastará con utilizar el nombre de la variable.

Por otro lado, Java elimina los tipos `struct` y `union` que se pueden implementar con `class` y que se mantenían en C++ por compatibilidad con C. También elimina el tipo `enum`, aunque se puede emular utilizando constantes numéricas con la palabra clave `final`.

También se eliminan definitivamente los `typedefs` para la definición de tipos, que en C++ ya habían perdido gran parte de su sentido al hacer que las clases, Structs, Union y Enum fueran tipos propios.

Finalmente, sólo admite las coerciones de tipos automáticas (*type casting*) en el caso de conversiones seguras; es decir, donde no haya riesgo de perder ninguna información. Por ejemplo, admite las conversiones automáticas de tipo `int` a `float`, pero no en sentido inverso donde se perderían los decimales. En caso de posible pérdida de información, hay que indicarle explícitamente que se desea realizar la conversión de tipos.

Otra característica muy destacable de Java es la implementación que realiza de los vectores. Los trata como a objetos reales y genera una excepción (error) cuando se superan sus límites. También dispone de un miembro llamado `length` para indicar su longitud, lo que proporciona un incremento de seguridad del lenguaje al evitar accesos indeseados a la memoria.

Para trabajar con cadenas de caracteres, Java dispone de los tipos `String` y `StringBuffer`. Las cadenas definidas entre comillas dobles se convierten automáticamente a objetos `String`, y no pueden modificarse. El tipo `StringBuffer` es similar, pero permite la modificación de su valor y proporciona métodos para su manipulación.

5.5.5. La gestión de variables dinámicas

Tal como se comentó al explicar C++, la gestión directa de la memoria es un arma muy potente pero también muy peligrosa: cual-

quier error en su gestión puede acarrear problemas muy graves en la aplicación y, quizás, en el sistema.

De hecho, la presencia de los apuntadores en C y C++ se debía al uso de cadenas y de vectores. Java proporciona objetos tanto para las cadenas, como los vectores, por lo que, para estos casos, ya no son necesarios los apuntadores. La otra gran necesidad, los pasos de parámetros por variable, queda cubierta por el uso de referencias.

Como en Java el tema de la seguridad es primordial, se optó por no permitir el uso de apuntadores, al menos en el sentido en que se entendían en C y C++.

En C++, se preveían dos formas de trabajar con apuntadores:

- Con su dirección, permitiendo incluso operaciones aritméticas sobre ella (apuntador).
- Con su contenido (* apuntador).

En Java se eliminan todas las operaciones sobre las direcciones de memoria. Cuando se habla de *referencias* se hace con un sentido diferente de C++. Una variable dinámica corresponde a la referencia al objeto (apuntador):

- Para ver el contenido de la variable dinámica, basta utilizar la forma (apuntador).
- Para crear un nuevo elemento, se mantiene el operador `new`.
- Si se asigna una variable tipo referencia (por ejemplo, un objeto) a otra variable del mismo tipo (otro objeto de la misma clase) el contenido no se duplica, sino que la primera variable apunta a la misma posición de la segunda variable. El resultado final es que el contenido de ambas es el mismo.

Java no permite operar directamente con las direcciones de memoria, lo que simplifica el acceso a su contenido: se hace a través del nombre de la variable (en lugar de utilizar la forma desreferenciada *nombre_variable).

ANOTACIONES

Otro de los principales riesgos que entraña la gestión directa de la memoria es la de liberar correctamente el espacio ocupado por las variables dinámicas cuando se dejan de utilizar. Java resuelve esta problemática proporcionando una herramienta que libera automáticamente dicho espacio cuando detecta que ya no se va a volver a utilizar más. Esta herramienta conocida como *recolector de basura* (*garbage collector*) forma parte del Java durante la ejecución de sus programas. Por tanto, no es necesaria ninguna instrucción `delete`, basta con asignar el apuntador a `null`, y el recolector de memoria detecta que la zona de memoria ya no se utiliza y la libera.

Si lo deseamos, en lugar de esperar a que se produzca la recolección de basura automáticamente, podemos invocar el proceso a través de la funcion `gc`(). No obstante, para la JVM dicha llamada sólo se considera como una sugerencia.

5.5.6. Las funciones y el paso de parámetros

Como ya sabemos, Java sólo se permite programación orientada a objetos. Por tanto, no se admiten las funciones independientes (siempre deben incluirse en clases) ni las funciones globales. Además, la implementación de los métodos se debe realizar dentro de la definición de la clase. De este modo, también se elimina la necesidad de los ficheros de cabeceras. El mismo compilador detecta si una clase ya ha sido cargada para evitar su duplicación. A pesar de su similitud con las funciones `inline`, ésta sólo es formal porque internamente tienen comportamientos diferentes: en Java no se implementan las funciones `inline`.

Por otro lado, Java continua soportando la sobrecarga de funciones, aunque no permite al programador la sobrecarga de operadores, a pesar de que el compilador utiliza esta característica internamente.

En Java todos los parámetros se pasan por valor.

ANOTACIONES

En el caso de los tipos de datos primitivos, los métodos siempre reciben una copia del valor original, que no se puede modificar.

En el caso de tipo de datos de referencia, también se copia el valor de dicha referencia. No obstante, por la naturaleza de las referencias, los cambios realizados en la variable recibida por parámetro también afectan a la variable original.

Para modificar las variables pasadas por parámetro a la función, debemos incluirlas como variables miembro de la clase y pasar como argumento la referencia a un objeto de dicha clase.

5.6. Las clases en Java

Como ya hemos comentado, uno de los objetivos que motivaron la creación de Java fue disponer de un lenguaje orientado a objetos "puro", en el sentido que siempre se debería cumplir dicho paradigma de programación. Esto, por su compatibilidad con C, no ocurría en C++. Por tanto, las clases son el componente fundamental de Java: todo está incluido en ellas. La manera de definir las clases en Java es similar a la utilizada en C++, aunque se presentan algunas diferencias:

Punto2D.java

```
class Punto2D
{
   int x, y;

// inicializando al origen de coordenadas
   Punto2D()
   {
     x = 0;
     y = 0;
   }

// inicializando a una coordenada x,y determinada
   Punto2D(int coordx, int coordy)
   {
```

ANOTACIONES

```
    x = coordx;
    y = coordy;
  }

  // calcula la distancia a otro punto
  float distancia(Punto2D npunto)
  {
    int dx = x - npunto.x;
    int dy = y - npunto.y;
    return ( Math.sqrt(dx * dx + dy * dy));
  }
}
```

- La primera diferencia es la inclusión de la definición de los métodos en el interior de la clase y no separada como en C++. Al seguir este criterio, ya no es necesario el operador de ámbito (::).
- La segunda diferencia es que en Java no es preciso el punto y coma (;) final.
- Las clases se guardan en un fichero con el mismo nombre y con la extensión .java (Punto2.java).

Una característica común a C y C++ es que Java también es sensible a las mayúsculas, por lo cual la clase `Punto2D` es diferente a punto2d o `pUnTo2d`.

Java permite guardar más de una clase en un fichero pero sólo permite que una de ellas sea pública. Esta clase será la que dará el nombre al archivo. Por tanto, salvo raras excepciones, se suele utilizar un archivo independiente para cada clase.

En la definición de la clase, de forma similar a C++, se declaran los atributos (o variables miembro) y los métodos (o funciones miembro) tal como se puede observar en el ejemplo anterior.

5.6.1. Declaración de objetos

Una vez definida una clase, para declarar un objeto de dicha clase basta con anteponer el nombre de la clase (como un tipo más) al del objeto.

ANOTACIONES

```
Punto2D    puntoUno;
```

El resultado es que puntoUno es una referencia a un objeto de la clase Punto2D. Inicialmente, esta referencia tiene valor null y no ha hecho ninguna reserva de memoria. Para poder utilizar esta variable para guardar información, es necesario crear una instancia mediante el operador new. Al utilizarlo, se llama al **constructor** del objeto Punto2D definido.

```
puntoUno = new Punto2D(2,2); // inicializando a (2,2)
```

Una diferencia importante en Java respecto a C++, es el uso de referencias para manipular los objetos. Como se ha comentado anteriormente, la asignación de dos variables declaradas como objetos sólo implica la asignación de su referencia:

```
Punto2D    puntoDos;
puntoDos = puntoUno;
```

Si se añade la instrucción anterior, no se ha hecho ninguna reserva específica de memoria para la referencia a objeto puntoDos. Al realizar la asignación, puntoDos hará referencia al mismo objeto apuntado por puntoUno, y no a una copia. Por tanto, cualquier cambio sobre los atributos de puntoUno se verán reflejados en puntoDos.

5.6.2. Acceso a los objetos

Una vez creado un objeto, se accede a cualquiera de sus atributos y métodos a través del operador punto (.) tal como hacíamos en C++.

```
int i;
float dist;

i = puntoUno.x;
dist = puntoUno.distancia(5,1);
```

En C++ se podía acceder al objeto a través de la desreferencia de un apuntador a dicho objeto (*apuntador), en cuyo caso, el acceso a sus atributos o métodos podía hacerse a través del operador

ANOTACIONES

punto (`*apuntador.atributo`) o a través de su forma de acceso abreviada mediante el operador → (`apuntador→atributo`). En Java, al no existir la forma desreferenciada *apuntador, tampoco existe el operador →.

Finalmente Java, igual que C++, permite el acceso al objeto dentro de los métodos de la clase a través del objeto `this`.

5.6.3. Destrucción de objetos

Cada vez que se crea un objeto, cuando se deja de utilizar debe ser destruido. La forma de operar de la gestión de memoria en Java permite evitar muchos de los conflictos que aparecen en otros lenguajes y es posible delegar esta responsabilidad a un proceso automático: el recolector de basura (*garbage collector*), que detecta cuando una zona de memoria no está referenciada y, cuando el sistema dispone de un momento de menor intensidad de procesador, la libera.

Algunas veces, al trabajar con una clase se utilizan otros recursos adicionales, como los ficheros. Frecuentemente, al finalizar la actividad de la clase, también se debe poder cerrar la actividad de dichos recursos adicionales. En estos casos, es preciso realizar un proceso manual semejante a los destructores en C++. Para ello, Java permite la implementación de un método llamado `finalize()` que, en caso de existir, es llamado por el mismo recolector. En el interior de este método, se escribe el código que libera explícitamente los recursos adicionales utilizados. El método `finalize` siempre es del tipo `static void`.

```
class MiClase
{
   MiClase()   //constructor
   {
     ...       //instrucciones de inicialización
   }

   static void finalize()   //destructor
   {
     ... //instrucciones de liberación de recursos
   }
}
```


ANOTACIONES

5.6.4. Constructores de copia

C++ dispone de los constructores de copia para asegurar que se realiza una copia completa de los datos en el momento de hacer una asignación, o de asignar un parámetro o un valor de retorno de una función.

Tal como se ha comprobado, Java tiene una filosofía diferente. Las asignaciones entre objetos no implican una copia de su contenido, sino que la segunda referencia pasa a referenciar al primer objeto. Por tanto, siempre se accede al mismo contenido y no es necesaria ninguna operación de reserva de memoria adicional. Como consecuencia de este cambio de filosofía, Java no precisa de constructores de copia.

5.6.5. Herencia simple y herencia múltiple

En Java, para indicar que una clase deriva de otra (es decir, hereda total o parcialmente sus atributos y métodos) se hace a través del término `extends`. Retomaremos el ejemplo de los perros y los mamíferos.

```
class Mamifero
{
   int edad;

   Mamifero()
   { edad = 0; }

   void asignarEdad(int nEdad)
   { edad = nEdad; }

   int obtenerEdad()
   { return (edad); }

   void emitirSonido()
   { System.out.println("Sonido "); }
}

class Perro extends Mamifero
{
   void emitirSonido()
   { System.out.println("Guau "); }
}
```

ANOTACIONES

En el ejemplo anterior, se dice que la clase Perro es una clase derivada de la clase Mamifero. También es posible leer la relación en el sentido contrario indicando que la clase Mamifero es una superclase de la clase Perro.

En C++ era posible la herencia múltiple, es decir, recibir los atributos y métodos de varias clases. Java no admite esta posibilidad, aunque en cierta manera permite una funcionalidad parecida a través de las interfaces.

5.7. Herencia y polimorfismo

La herencia y el polimorfismo son propiedades esenciales dentro del paradigma del diseño orientado a objetos. Estos conceptos ya han sido comentados en la unidad dedicada a C++ y continúan siendo vigentes en Java. No obstante, hay variaciones en su implementación que comentamos a continuación.

5.7.1. Las referencias this y super

En algunas ocasiones, es necesario acceder a los atributos o métodos del objeto que sirve de base al objeto en el cual se está. Tal como se ha visto, tanto Java como C++ proporciona este acceso a través de la referencia `this`.

La novedad que proporciona Java es poder acceder también a los atributos o métodos del objeto de la superclase a través de la referencia `super`.

5.7.2. La clase Object

Otra de las diferencias de Java respecto a C++ es que todos los objetos pertenecen al mismo árbol de jerarquías, cuya raíz es la clase Object de la cual heredan todas las demás: si una clase, en su definición, no tiene el término `Extends`, se considera que hereda directamente de Object.

ANOTACIONES

Podemos decir que la clase Object es la superclase de la cual derivan directa o indirectamente todas las demás clases en Java.

La clase Object proporciona una serie de métodos comunes, entre los cuales los siguientes:

- `public boolean equals(Object obj)`. Se utiliza para comparar el contenido de dos objetos y devuelve *true* si el objeto recibido coincide con el objeto que lo llama. Si sólo se desean comparar dos referencias a objeto, se pueden utilizar los operadores de comparación == y !=.
- `protected Object Clone()`. Retorna una copia del objeto.

5.7.3. Polimorfismo

C++ implementaba la capacidad de una variable de poder tomar varias formas a través de apuntadores a objetos. Como se ha comentado, Java no dispone de apuntadores y cubre esta función a través de referencias, pero el funcionamiento es similar.

```
Mamifero mamiferoUno = new Perro;
Mamifero mamiferoDos = new Mamifero;
```

Recordemos que, en Java, la declaración de un objeto siempre corresponde a una referencia a éste.

5.7.4. Clases y métodos abstractos

En C++ se comentó que, en algunos casos, las clases corresponden a elementos teóricos de los cuales no tiene ningún sentido instanciar objetos, sino que siempre se tenía que crear objetos de sus clases derivadas.

ANOTACIONES

La implementación en C++ se hacía a través de las funciones virtuales puras, cuya forma de representarla es, como menos, un poco peculiar: se declaraban asignando la función virtual a 0.

La implementación de Java para estos casos es mucho más sencilla: anteponer la palabra reservada `abstract` al nombre de la función. Al declarar una función como `abstract`, ya se indica que la clase también lo es. No obstante, es recomendable explicitarlo en la declaración anteponiendo la palabra `abstract` a la palabra reservada `class`.

El hecho de definir una función como abstract obliga a que las clases derivadas que puedan recibir este método la redefinan. Si no lo hacen, heredan la función como abstracta y, como consecuencia, ellas también lo serán, lo que impedirá instanciar objetos de dichas clases.

```
abstract class ObraDeArte
{
   String autor;

   ObraDeArte(){} //constructor

   abstract void mostrarObraDeArte(); //abstract
   void asignarAutor(String nAutor)
     { autor = nAutor; }
   String obtenerAutor();
   { return (autor); }
};
```

En el ejemplo anterior, se ha declarado como abstracta la función `mostrarObraDeArte`(), lo que obliga a redefinirla en las clases derivadas. Por tanto, no incluye ninguna definición. Por otro lado, destacamos que, al ser una clase abstracta, no será posible hacer un `new ObraDeArte`.

5.7.5. Clases y métodos finales

En la definición de variables, ya se ha tratado el concepto de variables finales. Hemos dicho que las variables finales, una vez iniciali-

ANOTACIONES

zadas, no pueden ser modificadas. El mismo concepto se puede aplicar a clases y métodos:

- Las clases finales no tienen ni pueden tener clases derivadas.
- Los métodos finales no pueden ser redefinidos en las clases derivadas.

El uso de la palabra reservada `final` se convierte en una medida de seguridad para evitar usos incorrectos o maliciosos de las propiedades de la herencia que pudiesen suplantar funciones establecidas.

5.7.6. Interfaces

Una interfaz es una colección de definiciones de métodos (sin sus implementaciones), cuya función es definir un protocolo de comportamiento que puede ser implementado por cualquier clase independientemente de su lugar en la jerarquía de clases.

Al indicar que una clase implementa una interfaz, se le obliga a redefinir todos los métodos definidos. En este aspecto, las interfaces se asemejan a las clases abstractas. No obstante, mientras una clase sólo puede heredar de una superclase (sólo permite herencia simple), puede implementar varias interfaces. Ello sólo indica que cumple con cada uno de los protocolos definidos en cada interfaz.

A continuación presentamos un ejemplo de declaración de interfaz:

```
public interface NombreInterfaz
Extends SuperInterfaz1, SuperInterfaz2
   { cuerpo interfaz }
```

Si una interfaz no se especifica como pública, sólo será accesible para las clases definidas en su mismo paquete.

ANOTACIONES

El cuerpo de la interfaz contiene las declaraciones de todos los métodos incluidos en ella. Cada declaración se finaliza en punto y coma (;) pues no tienen implementaciones e implícitamente se consideran `public` y `abstract`.

El cuerpo también puede incluir constantes en cuyo caso se consideran `public, static` y `final`.

Para indicar que una clase implementa una `interface`, basta con añadir la palabra clave `implements` en su declaración. Java permite la herencia múltiple de interfaces:

```
class MiClase extends SuperClase
implements Interfaz1, interfaz2
{ ... }
```

Cuando una clase declara una interfaz, es como si firmara un contrato por el cual se compromete a implementar los métodos de la interfaz y de sus superinterfaces. La única forma de no hacerlo es declarar la clase como `abstract`, con lo cual no se podrá instanciar objetos y se transmitirá esa obligación a sus clases derivadas.

De hecho, a primera vista parece que hay muchas similitudes entre las clases abstractas y las interfaces pero las diferencias son significativas:

- Una interfaz no puede implementar métodos, mientras que las clases abstractas si que lo hacen.
- Una clase puede tener varias interfaces, pero sólo una superclase.
- Las interfaces no forman parte de la jerarquía de clases y, por tanto, clases no relacionadas pueden implementar la misma interfaz.

Otra característica relevante de las interfaces es que al definirlas se está declarando un nuevo tipo de datos referencia. Una variable de dicho tipo de datos se podrá instanciar por cualquier clase que implemente esa interfaz. Esto proporciona otra forma de aplicar el polimorfismo.

ANOTACIONES

5.7.7. Paquetes

Para organizar las clases, Java proporciona los paquetes. Un paquete (*package*) es una colección de clases e interfaces relacionadas que proporcionan protección de acceso y gestión del espacio de nombres. Las clases e interfaces siempre pertenecen a un paquete.

Nota

De hecho, las clases e interfaces que forman parte de la plataforma de Java pertenecen a varios paquetes organizados por su función: `java.lang` incluye las clases fundamentales, `java.io` las clases para entrada/salida, etc.

El hecho de organizar las clases en paquetes evita en gran medida que pueda haber una colisión en la elección del nombre.

Para definir una clase o una interfaz en un paquete, basta con incluir en la primera línea del archivo la expresión siguiente:

```
package miPaquete;
```

Si no se define ningún paquete, se incluye dentro del paquete por defecto (`default package`), lo que es una buena solución para pequeñas aplicaciones o cuando se comienza a trabajar en Java.

Para acceder al nombre de la clase, se puede hacer a través del nombre largo:

```
miPaquete.MiClase
```

Otra posibilidad es la importación de las clases públicas del paquete mediante la palabra clave `import`. Después, es posible utilizar el nombre de la clase o de la interfaz en el programa sin el prefijo de éste:

```
import miPaquete.MiClase; //importa sólo la clase
import miPaquete.*        // importa todo el paquete
```

ANOTACIONES

Ejemplo

La importación de `java.awt` no incluye las clases del subpaquete `java.awt.event`.

Hay que tener en cuenta que importar un paquete no implica importar los diferentes subpaquetes que pueda contener.

Por convención, Java siempre importa por defecto del paquete `java.lang`.

Para organizar todas las clases y paquetes posibles, se crea un subdirectorio para cada paquete donde se incluyen las diferentes clases de dicho paquete. A su vez, cada paquete puede tener sus subpaquetes, que se encontrarán en un subdirectorio. Con esta organización de directorios y archivos, tanto el compilador como el intérprete tienen un mecanismo automático para localizar las clases que necesitan otras aplicaciones.

Ejemplo

La clase `graficos.figuras.rectangulo` se encontraría dentro del paquete `graficos.figuras` y el archivo estaría localizado en graficos\figuras\rectangulo.java.

5.7.8. El API (*applications programming interface*) de Java

La multitud de bibliotecas de funciones que proporciona el mismo lenguaje es una de las bazas primordiales de Java; bibliotecas, que están bien documentadas, son estándar y funcionan para las diferentes plataformas.

Este conjunto de bibliotecas está organizado en paquetes e incluido en la API de Java. Las principales clases son las siguientes:

Tabla 9.

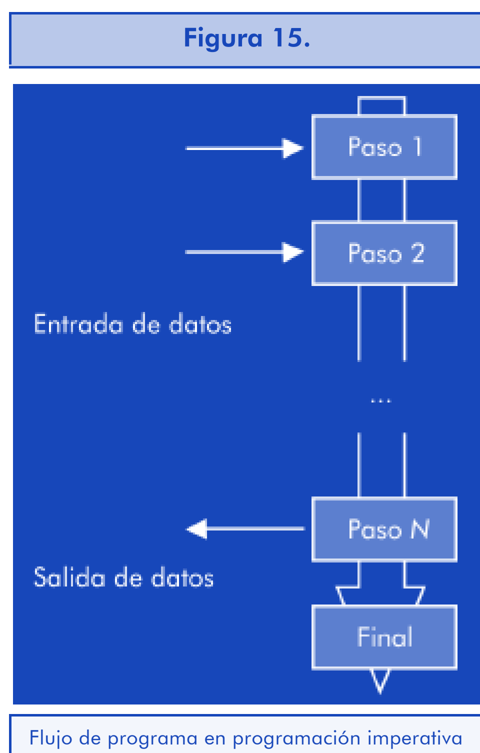
Paquete	Clases incorporadas
java.lang.	Clases fundamentales para el lenguaje como la clase String y otras.
java.io	Clases para la entrada y salida a través de flujos de datos, y ficheros del sistema.
java.util	Clases de utilidad como colecciones de datos y clases, el modelo de eventos, facilidades horarias, generación aleatoria de números, y otras.

Paquete	Clases incorporadas
java.math	Clase que agrupa todas las funciones matemáticas.
java.applet	Clase con utilidades para crear *applets* y clases que las *applets* utilizan para comunicarse con su contexto.
java.awt	Clases que permiten la creación de interfaces gráficas con el usuario, y dibujar imágenes y gráficos.
javax.swing	Clases con componentes gráficos que funcionan igual en todas las plataformas Java.
java.security	Clases responsables de la seguridad en Java (encriptación, etc.).
java.net	Clases con funciones para aplicaciones en red.
java.sql	Clase que incorpora el JDBC para la conexión de Java con bases de datos.

5.8. El paradigma de la programación orientada a eventos

Los diversos paradigmas de programación que se han revisado hasta el momento se caracterizan por tener un flujo de instrucciones secuencial y considerar los datos como el complemento necesario para el desarrollo de la aplicación. Su funcionamiento implica normalmente un inicio, una secuencia de acciones y un final de programa:

Figura 15.



Flujo de programa en programación imperativa

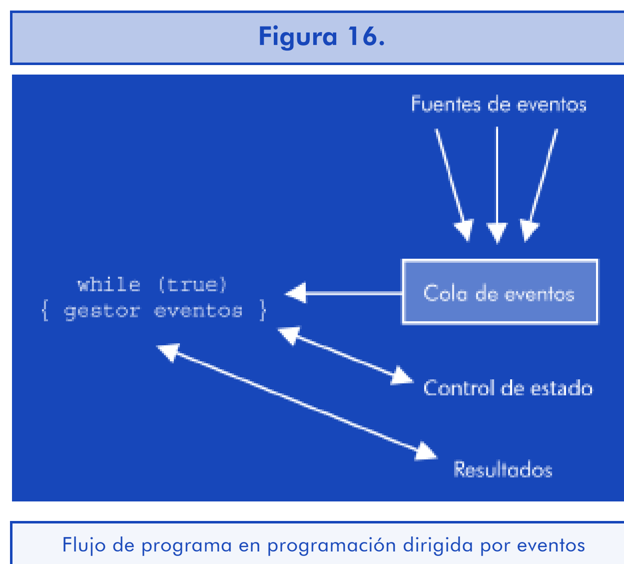
ANOTACIONES

Dentro de este funcionamiento secuencial, el proceso recibe sucesos externos que pueden ser esperados (entradas de datos del usuario por teclado, ratón u otras formas, lecturas de información del sistema, etc.) o inesperados (errores de sistema, etc.). A cada uno de estos sucesos externos lo denominaremos ***evento***.

En los paradigmas anteriores, los eventos no alteran el orden del flujo de instrucciones previsto: se les atiende para resolverlos o, si no es posible, se produce una finalización del programa.

En el paradigma de programación dirigida por eventos no se fija una secuencia única de acciones, sino que prepara reacciones a los eventos que puedan ir sucediendo una vez iniciada la ejecución del programa. Por tanto, en este modelo son los datos introducidos los que regulan la secuencia de control de la aplicación. También se puede observar que las aplicaciones difieren en su diseño respecto de los paradigmas anteriores: están preparadas para permanecer en funcionamiento un tiempo indefinido, recibiendo y gestionando eventos.

Figura 16.



Flujo de programa en programación dirigida por eventos

5.8.1. Los eventos en Java

Para la gestión de los eventos, Java propone utilizar **el modelo de delegación de eventos**. En este modelo, un componente recibe un evento y se lo transmite al gestor de eventos que tiene asignado para que lo gestione (*event listener*). Por tanto, tendremos una separación

ANOTACIONES

del código entre la generación del evento y su manipulación que nos facilitará su programación.

Diferenciaremos los cuatro tipos de elementos que intervienen:

- El evento (qué se recibe). En la gran mayoría de los casos, es el sistema operativo quien proporciona el evento y gestiona finalmente todas las operaciones de comunicaciones con el usuario y el entorno. Se almacena en un objeto derivado de la clase Event y que depende del tipo de evento sucedido. Los principales tienen relación con el entorno gráfico y son: `ActionEvent`, `KeyEvent`, `MouseEvent`, `AdjustmentEvent`, `WindowEvent`, `TextEvent`, `ItemEvent`, `FocusEvent`, `ComponentEvent`, `ContainerEvent`.

Cada una de estas clases tiene sus atributos y sus métodos de acceso.

- La fuente del evento (dónde se recibe). Corresponde al elemento donde se ha generado el evento y, por tanto, recoge la información para tratarla o, en nuestro caso, para traspasarla a su gestor de eventos. En entornos gráficos, suele corresponder al elemento con el cual el usuario ha interactuado (un botón, un cuadro de texto, etc.).

- El gestor de eventos (quién lo gestiona). Es la clase especializada que indica, para cada evento, cuál es la respuesta deseada. Cada gestor puede actuar ante diferentes tipos de eventos con sólo asignarle los perfiles adecuados.

- El perfil del gestor (qué operaciones debe implementar el gestor). Para facilitar esta tarea existen interfaces que indican los métodos a implementar para cada tipo de evento. Normalmente, el nombre de esta interfaz es de la forma <nombreEvento>Listener (literalmente, “el que escucha el evento”).

Ejemplo

`KeyListener` es la interfaz para los eventos de teclado y consideralos tres métodos siguientes: `keyPressed`, `keyReleased` y `keyTyped`. En algunos casos, la obligación de implementar todos los métodos supone una carga inútil. Para estas situaciones, Java proporciona adaptadores <nombreEvento>Adapter que implementan los diferentes métodos vacíos permitiendo así redefinir sólo aquellos métodos que nos interesan.

ANOTACIONES

Los principales perfiles (o interfaces) definidos por Java son los siguientes: `ActionListener`, `KeyListener`, `MouseListener`, `WindowListener`, `TextListener`, `ItemListener`, `FocusListener`, `AdjustmentListener`, `ComponentListener` y `ContainerListener`. Todos ellos derivados de la interfaz `EventListener`.

Ejemplo

Si a un objeto botón de la clase Button deseamos añadirle un *Listener* de los eventos de ratón haremos: `boton.addMouseListener(gestorEventos)`.

Finalmente, basta con establecer la relación entre la fuente del evento y su gestor. Para ello, en la clase fuente añadiremos un método del tipo `add<nombreEvento>Listener`.

De hecho, se podría considerar que los eventos no son realmente enviados al gestor de eventos, sino que es el propio gestor de eventos el que es asignado al evento.

> ***Nota***
>
> Comprenderemos más fácilmente el funcionamiento de los eventos a través de un ejemplo práctico, como el que muestra la creación de un *applet* mediante la librería gráfica Swing que se verá más adelante en esta unidad.

5.9. Hilos de ejecución (*threads*)

Los sistemas operativos actuales permiten la multitarea, al menos en apariencia, pues si el ordenador dispone de un único procesador, solo podrá realizar una actividad a la vez. No obstante, se puede organizar el funcionamiento de dicho procesador para que reparta su tiempo entre varias actividades o para que aproveche el tiempo que le deja libre una actividad para continuar la ejecución de otra.

A cada una de estas actividades se le llama *proceso*. Un proceso es un programa que se ejecuta de forma independiente y con un espacio propio de memoria. Por tanto, los sistemas operativos multitarea permiten la ejecución de varios procesos a la vez.

Cada uno de estos procesos puede tener uno o varios hilos de ejecución, cada uno de los cuales corresponde a un flujo secuencial de

ANOTACIONES

instrucciones. En este caso, todos los hilos de ejecución comparten el mismo espacio de memoria y se utiliza el mismo contexto y los mismos recursos asignados al proceso.

Java incorpora la posibilidad de que un proceso tenga múltiples hilos de ejecución simultáneos. El conocimiento completo de su implementación en Java supera los objetivos del curso y, a continuación, nos limitaremos a conocer las bases para la creación de los hilos y su ciclo de vida.

5.9.1. Creación de hilos de ejecución

En Java, hay dos formas de crear hilos de ejecución:

- Crear una nueva clase que herede de `java.lang.Thread` y sobrecargar el método `run()` de dicha clase.

- Crear una nueva clase con la interfaz `java.lang.Runnable` donde se implementará el método `run()`, y después crear un objeto de tipo *Thread* al que se le pasa como argumento un objeto de la nueva clase.

Siempre que sea posible se utilizará la primera forma, por su simplicidad. No obstante, si la clase ya hereda de alguna otra superclase, no será posible derivar también de la clase *Thread* (Java no permite la herencia múltiple), con lo cual se deberá escoger la segunda forma.

Veamos un ejemplo de cada una de las formas de crear hilos de ejecución:

Creación de hilos de ejecución derivando de la clase *Thread*
`ProbarThread.java`

```
class ProbarThread
{
   public static void main(String args[] )
   {
   AThread a = new AThread();
   BThread b = new BThread();
```

ANOTACIONES

```
   a.start();
   b.start();
   }
}

class AThread extends Thread
{
   public void run()
   {
   int i;
   for (i=1;i<=10; i++)
     System.out.print(" A"+i);
   }
}

class BThread extends Thread
{
   public void run()
   {
     int i;
     for (i=1;i<=10; i++)
       System.out.print(" B"+i);
   }
}
```

En el ejemplo anterior, se crean dos nuevas clases que derivan de la clase *Thread*: las clases `AThread` y `BThread`. Cada una de ellas muestra en pantalla un contador precedido por la inicial del proceso.

En la clase `ProbarThreads`, donde tenemos el método `main()`, se procede a la instanciación de un objeto para cada una de las clases *Thread* y se inicia su ejecución. El resultado final será del tipo (aunque no por fuerza en este orden):

A1 B1 A2 B2 A3 B3 A4 B4 A5 B5 A6 B6 A7 B7 A8 B8 A9 B9 A10 B10

Finalmente, solo hacer notar que en la ejecución `ProbarThreads` se ejecutan 3 hilos: el principal y los dos creados.

ANOTACIONES

Creación de hilos de ejecución implementando la interfaz `Runnable`

`Probar2Thread.java`

```
class Probar2Thread
{
   public static void main(String args[])
   {
     AThread a = new AThread();
     BThread b = new BThread();
     a.start();
     b.start();
   }
}

class AThread implements Runnable
{
   Thread t;
   public void start()
   {
     t = new Thread(this);
     t.start();
   }

   public void run()
   {
     int i;
     for (i=1;i<=50; i++)
       System.out.print(" A"+i);
   }
}

class BThread implements Runnable
{
   Thread t;

   public void start()
   {
     t = new Thread(this);
     t.start();
   }

   public void run()
   {
     int i;
     for (i=1;i<=50; i++)
       System.out.print(" B"+i);
   }
}
```

ANOTACIONES

En este ejemplo, se puede observar que la clase principal main() no ha cambiado, pero sí lo ha hecho la implementación de cada una de las *clases AThread* y *BThread*. En cada una de ellas, además de implementar la interfaz `Runnable`, se tiene que definir un objeto de la clase `Thread` y redefinir el método `start`() para que llame al `start`() del objeto de la clase `Thread` pasándole el objeto actual `this`.

Para finalizar, dos cosas: es posible pasarle un nombre a cada hilo de ejecución para identificarlo, puesto que la clase *Thread* tiene el constructor sobrecargado para admitir esta opción:

```
public Thread (String nombre);
public Thread (Runnable destino, String nombre);
```

Siempre es posible recuperar el nombre a través del método:

```
public final String getName();
```

5.9.2. Ciclo de vida de los hilos de ejecución

El ciclo de vida de los hilos de ejecución se puede representar a partir de los estados por los que pueden pasar:

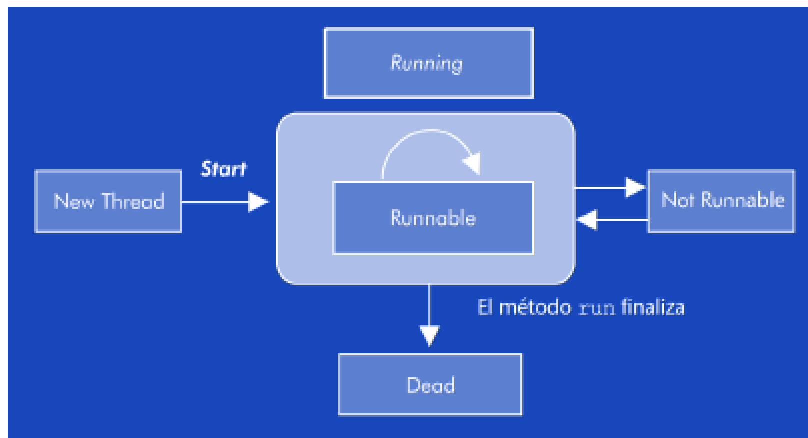
- Nuevo (*new*): el *thread* se acaba de crear pero todavía no está inicializado, es decir, todavía no se ha ejecutado el método `start()`.
- Ejecutable (*runnable*): el *thread* se está ejecutando o está en disposición para ello.
- Bloqueado (*blocked* o *not runnable*): el *thread* está bloqueado por algún mensaje interno `sleep()`, `suspend()` o `wait()` o por alguna actividad interna, por ejemplo, en espera de una entrada de datos. Si está en este estado, no entra dentro de la lista de tareas a ejecutar por el procesador.

Para volver al estado de Ejecutable, debe recibir un mensaje interno `resume()` o `notify()` o finalizar la situación que provocaba el bloqueo.

ANOTACIONES

- Muerto (*dead*): el método habitual de finalizar un *thread* es que haya acabado de ejecutar las instrucciones del método `run()`. También podría utilizarse el método `stop()`, pero es una opción considerada "peligrosa" y no recomendada.

Figura 17.



5.10. Los *applets*

Un *applet* es una miniaplicación Java preparada para ser ejecutada en un navegador de Internet. Para incluir un *applet* en una página HTML, basta con incluir su información por medio de las etiquetas <APPLET> ... </APPLET>.

La mayoría de navegadores de Internet funcionan en un entorno gráfico. Por tanto, los *applet* deben adaptarse a él a través de bibliotecas gráficas. En este apartado, se utilizará la biblioteca `java.awt` que es la biblioteca proporcionada originalmente desde sus primeras versiones. Una discusión más profunda entre las diferentes bibliotecas disponibles se verá más adelante en esta unidad.

Las características principales de los *applets* son las siguientes:

- Los ficheros .class se descargan a través de la red desde un servidor HTTP hasta el navegador, donde la JVM los ejecuta.

- Dado que se usan a través de Internet, se ha establecido que tengan unas restricciones de seguridad muy fuertes, como por ejem-

ANOTACIONES

plo, que sólo puedan leer y escribir ficheros desde su servidor (y no desde el ordenador local), que sólo puedan acceder a información limitada en el ordenador donde se ejecutan, etc.

- Los *applets* no tienen ventana propia, que se ejecutan en una ventana del navegador.
- Desde el punto de vista del programador, destacan los siguientes aspectos:
- No necesitan método `main`. Su ejecución se inicia por otros mecanismos.
- Derivan siempre de la clase java.applet.Applet y, por tanto, deben redefinir algunos de sus métodos como `init()`, `start()`, `stop()` y `destroy()`.
- También suelen redefinir otros métodos como `paint()`, `update()` y `repaint()` heredados de clases superiores para tareas gráficas.
- Disponen de una serie de métodos para obtener información sobre el *applet* o sobre otros *applets* en ejecución en la misma página como `getAppletInfo()`, `getAppletContext()`, `getParameter()`, etc.

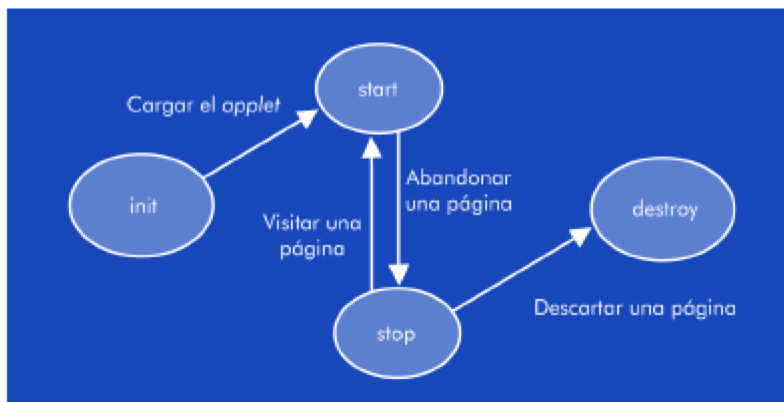
5.10.1. Ciclo de vida de los *applets*

Por su naturaleza, el ciclo de vida de un *applet* es algo más complejo que el de una aplicación normal. Cada una de las fases del ciclo de vida está marcada con una llamada a un método del *applet*:

- `void init()`. Se llama cuando se carga el *applet*, y contiene las inicializaciones que necesita.
- `void start()`. Se llama cuando la página se ha cargado, parado (por minimización de la ventana, cambio de página web, etc.) y se ha vuelto a activar.
- `void stop()`. Se llama de forma automática al ocultar el *applet*. En este método, se suelen parar los hilos que se están ejecutando para no consumir recursos innecesarios.
- `void destroy()`. Se llama a este método para liberar los recursos (menos la memoria) del *applet*.

ANOTACIONES

Figura 18.



Al ser los *applets* aplicaciones gráficas que aparecen en una ventana del navegador, también es útil redefinir el siguiente método:

- void paint(Graphics g). En esta función se debe incluir todas las operaciones con gráficos, porque este método es llamado cuando el *applet* se dibuja por primera vez y cuando se redibuja.

5.10.2. Manera de incluir *applets* en una página HTML

Como ya hemos comentado, para llamar a un *applet* desde una página html utilizamos las etiquetas `<APPLET> ... <\APPLET>`, entre las que, como mínimo, incluimos la información siguiente:

- `CODE` = nombre del *applet* (por ejemplo, miApplet.class)
- `WIDTH` = anchura de la ventana
- `HEIGHT` = altura de la ventana

Y opcionalmente, los atributos siguientes:

- `NAME` = "unnombre" lo cual le permite comunicarse con otros *applets*
- `ARCHIVE` = "unarchivo" donde se guardan las clases en un .zip o un .jar
- `PARAM NAME` = "param1" VALUE = "valor1" para poder pasar parámetros al *applet*.

ANOTACIONES

5.10.3. Mi primer *applet* en Java

La mejor manera de comprender el funcionamiento de los *applets* es a través de un ejemplo práctico. Para crear nuestro primer *applet* seguiremos estos pasos:

1) Crear un fichero fuente. Mediante el editor escogido, escribiremos el texto y lo salvaremos con el nombre `HolaMundoApplet.java`.

HolaMundoApplet.java

```
import java.applet.*;
import java.awt.*;
/**
 * La clase HolaMundoApplet muestra el mensaje
 * "Hola Mundo" en la salida estándar.
 */
public class HolaMundoApplet extends Applet{
   public void paint(Graphics g)
   {
     // Muestra "Hola Mundo!"
     g.drawString("¡Hola Mundo!", 75, 30 );
   }
}
```

2) Crear un fichero HTML. Mediante el editor escogido, escribiremos el texto.

HolaMundoApplet.html

```
<HTML>
<HEAD>
<TITLE>Mi primer applet</TITLE>
</HEAD>
<BODY>
Os quiero dar un mensaje:
<APPLET CODE="HolaMundoApplet.class" WIDTH=150 HEIGHT=25>
</APPLET>
</BODY>
</HTML>
```

3) Compilar el programa generando un fichero *bytecode*.

```
javac HolaMundoApplet.java
```

4) Visualizar la página HolaMundoApplet.html desde un navegador.

ANOTACIONES

5.11. Programación de interfaces gráficas en Java

La aparición de las interfaces gráficas supuso una gran evolución en el desarrollo de sistemas y aplicaciones. Hasta su aparición, los programas se basaban en el modo texto (o consola) y, generalmente, el flujo de información de estos programas era secuencial y se dirigía a través de las diferentes opciones que se iban introduciendo a medida que la aplicación lo solicitaba.

Las interfaces gráficas permiten una comunicación mucho más ágil con el usuario facilitando su interacción con el sistema en múltiples puntos de la pantalla. Se puede elegir en un momento determinado entre múltiples operaciones disponibles de naturaleza muy variada (por ejemplo, introducción de datos, selección de opciones de menú, cambios de formularios activos, cambios de aplicación, etc.) y, por tanto, múltiples flujos de instrucciones, siendo cada uno de ellos respuesta a eventos diferenciados.

Los programas que utilizan dichas interfaces son un claro ejemplo del paradigma de programación dirigido por eventos.

Con el tiempo, las interfaces gráficas han ido evolucionando y han ido surgiendo nuevos componentes (botones, listas desplegables, botones de opciones, etc.) que se adaptan mejor a la comunicación entre los usuarios y los ordenadores. La interacción con cada uno de estos componentes genera una serie de cambios de estado y cada cambio de estado es un suceso susceptible de necesitar o provocar una acción determinada. Es decir, un posible evento.

La programación de las aplicaciones con interfaces gráficas se elabora a partir de una serie de componentes gráficos (desde formularios hasta controles, como los botones o las etiquetas), que se definen como objetos propios, con sus variables y sus métodos.

ANOTACIONES

Mientras que las variables corresponden a las diferentes propiedades necesarias para la descripción del objeto (longitudes, colores, bloqueos, etc.), los métodos permiten la codificación de una respuesta a cada uno de los diferentes eventos que pueden sucederle a dicho componente.

5.11.1. Las interfaces de usuario en Java

Java, desde su origen en la versión 1.0, implementó un paquete de rutinas gráficas denominadas AWT (*abstract windows toolkit*) incluidas en el paquete `java.awt` en la que se incluyen todos los componentes para construir una interfaz gráfica de usuario (GUI-*graphic user interface*) y para la gestión de eventos. Este hecho hace que las interfaces generadas con esta biblioteca funcionen en todos los entornos Java, incluidos los diferentes navegadores.

Este paquete sufrió una revisión que mejoró muchos aspectos en la versión 1.1, pero continuaba presentando un inconveniente: AWT incluye componentes que dependen de la plataforma, lo que ataca frontalmente uno de los pilares fundamentales en la filosofía de Java.

En la versión 1.2 (o Java 2) se ha implementado una nueva versión de interfaz gráfica que soluciona dichos problemas: el paquete Swing. Este paquete presenta, además, una serie de ventajas adicionales respecto a la AWT como aspecto modificable (diversos *look and feel,* como Metal que es la presentación propia de Java, Motif propia de Unix, Windows) y una amplia variedad de componentes, que se pueden identificar rápidamente porque su nombre comienza por J.

Swing conserva la gestión de eventos de AWT, aunque la enriquece con el paquete javax.swing.event.

Su principal inconveniente es que algunos navegadores actuales no la incluyen inicialmente, con lo cual su uso en los *applets* queda limitado.

Aunque el objetivo de este material no incluye el desarrollo de aplicaciones con interfaces gráficas, un pequeño ejemplo del uso de la biblioteca Swing nos permitirá presentar sus ideas básicas así como el uso de los eventos.

ANOTACIONES

5.11.2. Ejemplo de *applet* de Swing

En el siguiente ejemplo, se define un *applet* que sigue la interfaz Swing. La primera diferencia respecto al *applet* explicado anteriormente corresponde a la inclusión del paquete `javax.swing.*`.

Se define la clase `HelloSwing` que hereda de la clase `Japplet` (que corresponde a los *applets* en Swing). En esta clase, se define el método `init` donde se define un nuevo botón (`new Jbutton`) y se añade al panel de la pantalla (`.add`).

Los botones reciben eventos de la clase `ActionEvent` y, para su tratamiento, la clase que gestiona sus eventos debe implementar la interfaz `ActionListener`.

Para esta función se ha declarado la clase `GestorEventos` que, en su interior, redefine el método `actionPerformed` (el único método definido en la interfaz `ActionListener`) de forma que abra una nueva ventana a través del método showMessageDialog.

Finalmente, sólo falta indicarle a la clase `HelloSwing` que la clase `GestorEventos` es la que gestiona los mensajes del botón. Para ello, usamos el método `.addActionListener(GestorEventos)`

HelloSwing.java

```
import javax.swing.*;
import java.awt.event.*;
public class HelloSwing extends Japplet
{
 public void init()
 { //constructor
  JButton boton = new JButton("Pulsa aquí!");
  GestorEventos miGestor = new GestorEventos();
   boton.addActionListener(miGestor); //Gestor del botón
   getContentPane().add(boton);
  } // init
} // HelloSwing

class GestorEventos implements ActionListener
{
  public void actionPerformed(ActionEvent evt)
  {
   String titulo = "Felicidades";
   String mensaje = "Hola mundo, desde Swing";
```

ANOTACIONES

```
    JOptionPane.showMessageDialog(null, mensaje,
        titulo, JOptionPane.INFORMATION_MESSAGE);

  } // actionPerformed
} // clase GestorEventos
```

5.12. Introducción a la información visual

Aunque por motivos de simplicidad se han utilizado entornos de desarrollo en modo texto, en la realidad se utilizan entornos de desarrollo integrados (IDE) que incorporan las diferentes herramientas que facilitan al programador el proceso de generación de código fuente y ejecutable (editor, compilador, *debugger*, etc.).

En ambos casos, no hay ninguna diferencia en el lenguaje de programación: se considera que el lenguaje es "textual", pues las instrucciones primitivas se expresan mediante texto.

Actualmente, estos entornos de desarrollo proporcionan la posibilidad de trabajar de una forma más visual permitiendo confeccionar las pantallas de trabajo (formularios, informes, etc.), mediante el arrastre de los diferentes controles a su posición final, y después procediendo a la introducción de valores para sus atributos (colores, medidas, etc.) y el código para cada uno de los eventos que puede provocar. No obstante, la naturaleza del lenguaje no varía y se continúa considerando "textual".

Otro paradigma diferente correspondería a la programación mediante lenguajes "visuales". Hablamos de *lenguaje visual* cuando el lenguaje manipula información visual, soporta interacciones visuales o permite la programación mediante expresiones visuales. Por tanto, sus primitivas son gráficos, animaciones, dibujos o iconos.

En otras palabras, los programas se constituyen como una relación entre distintas instrucciones que se representan gráficamente. Si la relación fuese secuencial y las instrucciones se expresaran mediante palabras, tal programación sería fácilmente reconocible. En todo caso, ya se ha comentado que la programación concurrente y la que se dirige por eventos no presentan una relación secuencial entre sus instrucciones que, además, suelen ser de alto nivel de abstracción.

ANOTACIONES

Así pues, la programación visual resultaría ser una técnica para describir programas cuyos flujos de ejecución se adapten a los paradigmas anteriormente citados.

Por tanto, a pesar de la posible confusión aportada por los nombres de varios entornos de programación como la familia Visual de Microsoft (Visual C++, Visual Basic, etc.), a estos lenguajes se les debe continuar clasificando como lenguajes “textuales”, aunque su entorno gráfico de desarrollo sí que puede suponer una aproximación hacia la programación visual.

5.13. Resumen

En esta unidad se ha presentado un nuevo lenguaje de programación orientado a objetos que nos proporciona independencia de la plataforma sobre la que se ejecuta. Para ello, proporciona una máquina virtual sobre cada plataforma. De este modo, el desarrollador de aplicaciones sólo debe escribir su código fuente una única vez y compilarlo para generar un código “ejecutable” común, consiguiendo, de esta manera, que la aplicación pueda funcionar en entornos dispares como sistemas Unix, sistemas Pc o Apple McIntosh. Esta filosofía es la que se conoce como “*write once, run everywhere*”.

Java nació como evolución del C++ y adaptándose a las condiciones anteriormente descritas. Se aprovecha el conocimiento previo de los programadores en los lenguajes C y C++ para facilitar una aproximación rápida al lenguaje.

Al necesitar Java un entorno de poco tamaño, permite incorporar su uso en navegadores web. Como el uso de estos navegadores implica, normalmente, la existencia de un entorno gráfico, se ha aprovechado esta situación para introducir brevemente el uso de bibliotecas gráficas y el modelo de programación dirigido por eventos.

Asimismo, Java incluye de forma estándar dentro de su lenguaje operaciones avanzadas que en otros lenguajes realiza el sistema operativo o bibliotecas adicionales. Una de estas características es la programación de varios hilos de ejecución (*threads*) dentro del mismo proceso. En esta unidad, hemos podido introducirnos en el tema.

ANOTACIONES

5.14. Ejercicios de autoevaluación

1. Ampliad la clase Leer.java para implementar la lectura de variables tipo `double`.

2. Introducid la fecha (solicitando una cadena para la población y tres números para la fecha) y devolvedlo en forma de texto.

Ejemplo

Entrada: Barcelona 15 02 2003

Salida: Barcelona, 15 de febrero de 2003

3. Implementad una aplicación que pueda diferenciar si una figura de cuatro vértices es un cuadrado, un rectángulo, un rombo u otro tipo de polígono.

Se definen los casos de la siguiente forma:

- Cuadrado: lados 1,2,3 y 4 iguales; 2 diagonales iguales
- Rectángulo: lados 1 y 3, 2 y 4 iguales; 2 diagonales iguales
- Rombo: lados 1,2,3 y 4 iguales, diagonales diferentes
- Polígono: los demás casos

Para ello, se definen la clase Punto2D definiendo las coordenadas x, y, y el método “distancia a otro punto”.

Ejemplo

(0,0) (1,0) (1,1) (0,1) Cuadrado

(0,1) (1,0) (2,1) (1,2) Cuadrado

(0,0) (2,0) (2,1) (0,1) Rectángulo

(0,2) (1,0) (2,2) (1,4) Rombo

4. Convertid el código del ejercicio del ascensor (ejercicio 3 de la unidad 4) a Java.

ANOTACIONES

5.14.1. Solucionario

1.

Leer.java

```
import java.io.*;

public class Leer
{
   public static String getString()
   {
     String str = "";
     try
     {
       InputStreamReader isr = new
       InputStreamReader(System.in);
       BufferedReader br = new BufferedReader(isr);
       str = br.readLine();
     }
     catch(IOException e)
  { System.err.println("Error: " + e.getMessage());
}
     return str; // devolver el dato tecleado
   }

   public static int getInt()
   {
     try
     { return Integer.parseInt(getString()); }
     catch(NumberFormatException e)
     { return 0; // Integer.MIN_VALUE }
   } // getInt

   public static double getDouble()
   {
     try
     {
       return Double.parseDouble(getString());
     }
     catch(NumberFormatException e)
     {
       return 0; // Double.MIN_VALUE
     }
   } // getDouble
} // Leer
```

ANOTACIONES

2.

construirFecha.java

```
import java.io.*;
public class construirFecha
{
   static String nombreMes(int nmes)
   {
    String strmes;
    switch (nmes)
    {
     case 1: { strmes = "enero"; break; }
     case 2: { strmes = "febrero"; break; }
     case 3: { strmes = "marzo"; break; }
     case 4: { strmes = "abril"; break; }
     case 5: { strmes = "mayo"; break; }
     case 6: { strmes = "junio"; break; }
     case 7: { strmes = "julio"; break; }
     case 8: { strmes = "agosto"; break; }
     case 9: { strmes = "septiembre"; break; }
     case 10: { strmes = "octubre"; break; }
     case 11: { strmes = "noviembre"; break; }
     case 12: { strmes = "diciembre"; break; }
     default: { strmes = " -- "; break; }
    } // switch nmes
    return (strmes);
   } // nombreMes

   public static void main(String args[])
   {
    String poblacion;
    int dia, mes, año;
    String mifecha, strmes;
    System.out.print(" Población: ");
    poblacion = Leer.getString();
    System.out.print(" Dia: ");
    dia = Leer.getInt();
    System.out.print(" Mes: ");
    mes = Leer.getInt();
    System.out.print(" Año: ");
    año = Leer.getInt();
    mifecha = poblacion + ", " + dia;
   mifecha = mifecha +" de "+ nombreMes(mes) +"
                         de "+ año;
   System.out.print(" La fecha introducida es: ");
   System.out.println(mifecha);
   } // main
} // class
```

ANOTACIONES

3.

Punto2D.java

```
class Punto2D
{
   public int x, y;
   // inicializando al origen de coordenadas
   Punto2D()
   { x = 0; y = 0; }

  // inicializando a una coordenada x,y determinada
   Punto2D(int coordx, int coordy)
   { x = coordx; y = coordy; }


   // calcula la distancia a otro punto
   double distancia(Punto2D miPunto)
   {
    int dx = x - miPunto.x;
    int dy = y - miPunto.y;
    return ( Math.sqrt(dx * dx + dy * dy));
   }
}
```

AppReconocerFigura.java

```
class AppReconocerFigura
{
static public void main(String args[])
   {
   int i;
   int coordx, coordy;

   // Introducir 4 puntos e
   // indicar cuál es el más cercano al origen.

   Punto2D listaPuntos[];
   listaPuntos = new Punto2D[4];

   // entrar datos
   for (i=0; i<4; i++)
   {
    System.out.println("Entrar el punto (" + i + ")" );
    System.out.print("Coordenada x " );
    coordx = Leer.getInt();
    System.out.print("Coordenada y " );
    coordy = Leer.getInt();
```

ANOTACIONES

```
  listaPuntos[i] = new Punto2D(coordx, coordy);
 } //for

 // indicar si los 4 puntos forman un
 // cuadrado: dist1 = dist2 = dist3 = dist4
 // diag1 = diag2
 // rombo: dist1 = dist2 = dist3 = dist4
 // diag1 <> diag2
 // rectangulo: dist1 = dist3, dist2 = dist4
 // diag1 = diag2
 // poligono: otros casos

 double dist[] = new double[4];
 double diag[] = new double[3];

 // calculo de distancias
 for (i=0; i<3; i++)
 {
  dist[i] = listaPuntos[i].distancia(listaPuntos[i+1]);
  System.out.print("Distancia "+i + " " + dist[i] );
 } //for
 dist[3] = listaPuntos[3].distancia(listaPuntos[0]);
 System.out.println("Distancia "+i + " " + dist[3] );

 // calculo de diagonales
 for (i=0; i<2; i++)
 {
  diag[i] = listaPuntos[i].distancia(listaPuntos[i+2]);
 } //for
if ( (dist[0] == dist[2]) && (dist[1] == dist[3]) )
 {

 // es cuadrado, rectángulo o rombo
  if (dist[1] == dist[2]) {
  // es cuadrado o rombo
  if (diag[0] == diag[1])
   { System.out.println("Es un cuadrado"); }
   else
   { System.out.println("Es un rombo"); } // if
   }
   else
   {
  // es rectangulo
  if (diag[0] == diag[1])
   { System.out.println("Es un rectángulo"); }
   } else
   { System.out.println("Es un polígono"); } // if
  }
 } else
  { System.out.println("Es un poligono"); } // if
  } // main
} // class
```

4.

AppAscensor.java

```
import java.io.*;

public class AppAscensor{

   static int n_codigo, n_peso, n_idioma;

   public static void solicitarDatos()
   {
     System.out.print ("Codigo: ");
     n_codigo = Leer.getInt();

     System.out.print("Peso: ");
     n_peso = Leer.getInt();

     System.out.print(
         "Idioma: [1] Catalán [2] Castellano [3] Inglés ");
     n_idioma = Leer.getInt();
   } // solicitarDatos


   public static void mostrarEstadoAscensor(Ascensor nA)
   {
     nA.mostrarOcupacion();
     System.out.print(" - ");
     nA.mostrarCarga();
     System.out.println(" ");
     nA.mostrarListaPasajeros();
   } // mostrarEstadoAscensor

   public static void main( String[] args)
   {
     int opc;
     boolean salir = false;
     Ascensor unAscensor;
     Persona unaPersona;
     Persona localizarPersona;

     opc=0;

   unAscensor = new Ascensor();// inicializamos ascensor
   unaPersona = null;// inicializamos unaPersona

   do {

      System.out.print(
"ASCENSOR: [1]Entrar [2]Salir [3]Estado [0]Finalizar ");
     opc = Leer.getInt();

     switch (opc)
     {
     case 1: { // Opcion Entrar
      solicitarDatos();
      switch (n_idioma)
```

ANOTACIONES

```
      {
        case 1: { //"Catalan"
          unaPersona = new Catalan (n_codigo, n_peso);
          break;
        }
        case 2: { //"Castellano"
          unaPersona = new Castellano (n_codigo, n_peso);
          break;
        }
        case 3: { //"Ingles"
          unaPersona = new Ingles(n_codigo, n_peso);
          break;
        }
          default: { //"Ingles"
          unaPersona = new Ingles(n_codigo, n_peso);
          break;
        }
      } //switch n_idioma

      if (unAscensor.persona_PuedeEntrar(unaPersona))
      {
        unAscensor.persona_Entrar(unaPersona);
        if (unAscensor.obtenerOcupacion()>1)
        {
          System.out.print(unaPersona.obtenerCodigo());
          System.out.print(" dice: ");
          unaPersona.saludar();
          System.out.println(" "); // Responden las demas
          unAscensor.restoAscensor_Saludar(unaPersona);
        }
      } //puede entrar
      break;
    }
    case 2: { //Opcion Salir

      localizarPersona = new Persona(); //Por ejemplo
      unaPersona = null;

      localizarPersona.solicitarCodigo();
      if (unAscensor.persona_Seleccionar(localizarPersona))
      {
        unaPersona = unAscensor.obtenerRefPersona();
        unAscensor.persona_Salir( unaPersona );
        if (unAscensor.obtenerOcupacion()>0)
        {
          System.out.print(unaPersona.obtenerCodigo());
          System.out.print(" dice: ");
          unaPersona.despedirse();
          System.out.println(" "); // Responden las demas
          unAscensor.restoAscensor_Despedirse(unaPersona);
          unaPersona=null;
        }
      } else {
          System.out.println(
            "No hay ninguna persona con este código");
      } // seleccionar
```

ANOTACIONES

```
         localizarPersona=null;
         break;
        }
        case 3: { //Estado
         mostrarEstado(unAscensor);
         break;
        }

        case 0: { //Finalizar
         System.out.println("Finalizar");
         salir = true;
         break;
        }

        } //switch opc
      } while (! salir);
      } // main
} //AppAscensor
```

Ascensor.java

```
import java.io.*;

class Ascensor {
   private int ocupacion;
   private int carga;
   private int ocupacionMaxima;
   private int cargaMaxima;
   private Persona pasajeros[];
   private Persona refPersonaSeleccionada;
//
// Constructores y destructores
//
Ascensor()
{
   ocupacion = 0;
   carga=0;
   ocupacionMaxima=6;
   cargaMaxima=500;
   pasajeros = new Persona[6];
   refPersonaSeleccionada = null;
} //Ascensor()
```

ANOTACIONES

```
// Funciones de acceso
int ObtenerOcupacion()
{ return (ocupacion); }

void ModificarOcupacion(int dif_ocupacion)
{ ocupacion += dif_ocupacion; }

void MostrarOcupacion()
{
   System.out.print("Ocupacion actual: ");
   System.out.print(ocupacion );
}

int obtenerCarga()
{ return (carga); }

void modificarCarga(int dif_carga)
{ carga += dif_carga; }

void mostrarCarga()
{ System.out.print("Carga actual: ");
   System.out.print(carga) ;
}

Persona obtenerRefPersona()
{return (refPersonaSeleccionada);}

boolean persona_PuedeEntrar(Persona unaPersona)
{
   // si la ocupación no sobrepasa el límite de ocupación y
   // si la carga no sobrepasa el límite de carga
   // ->puede entrar

   boolean tmpPuedeEntrar;
   if (ocupacion + 1 > ocupacionMaxima)
   {
    System.out.println(
   "Aviso: El ascensor está completo. No puede entrar");
    return (false);
   }

   if (unaPersona.obtenerPeso() + carga > cargaMaxima )
   {
    System.out.println(
"Aviso: El ascensor supera su carga máxima. No puede entrar");
```

ANOTACIONES

```
    return (false);
   }
   return (true);
}
boolean persona_Seleccionar(Persona localizarPersona)
{
   int contador;

   // Se selecciona persona entre pasajeros del ascensor.
   boolean personaEncontrada = false;
   if (obtenerOcupacion() > 0)
   {
    contador=0;
    do {
     if (pasajeros[contador] != null)
     {
      if(pasajeros[contador].igualCodigo(localizarPersona)
      {
       refPersonaSeleccionada=pasajeros[contador];
       personaEncontrada=true;
       break;
      }
     }
     contador++;
    } while (contador<ocupacionMaxima);
    if (contador>=ocupacionMaxima)
     {refPersonaSeleccionada=null;}
   }
   return (personaEncontrada);
}

void persona_Entrar(Persona unaPersona)
{
   int contador;
   modificarOcupacion(1);
   modificarCarga(unaPersona.obtenerPeso());
   System.out.print(unaPersona.obtenerCodigo());
   System.out.println(" entra en el ascensor ");

   contador=0;
   // hemos verificado anteriormente que hay plazas libres
   do {
    if (pasajeros[contador]==null )
```

ANOTACIONES

```
    {
     pasajeros[contador]=unaPersona;
     break;
    }
    contador++;
   } while (contador<ocupacionMaxima);
}

void persona_Salir(Persona unaPersona)
{
   int contador;

   contador=0;
   do {
    if ((pasajeros[contador]==unaPersona ))
    {
     System.out.print(unaPersona.obtenerCodigo());
     System.out.println(" sale del ascensor ");
     pasajeros[contador]=null;

     // Modificamos la ocupacion y la carga
     modificarOcupacion(-1);
     modificarCarga(-1 * (unaPersona.obtenerPeso()));
     break;
    }
    contador++;
   } while (contador<ocupacionMaxima);
   if (contador==ocupacionMaxima)
    {System.out.println(
    " No hay persona con este código. No sale nadie ");}
}

void mostrarListaPasajeros()
{
   int contador;
   Persona unaPersona;

   if (obtenerOcupacion() > 0)
   {
    System.out.println("Lista de pasajeros del ascensor:");
    contador=0;
    do {
     if (!(pasajeros[contador]==null ))
     {
```

ANOTACIONES

```
        unaPersona=pasajeros[contador];
        System.out.print(unaPersona.obtenerCodigo());
        System.out.print("; ");
       }
       contador++;
     } while (contador<ocupacionMaxima);
     System.out.println("");
   } else {
     System.out.println("El ascensor esta vacío");
   }
}

void restoAscensor_Saludar(Persona unaPersona)
{
   int contador;
   Persona otraPersona;


   if (obtenerOcupacion() > 0)
   {
     contador=0;
     do {
       if (!(pasajeros[contador]==null ))
       {
        otraPersona=pasajeros[contador];
        if (!unaPersona.igualCodigo(otraPersona) )
        {
          System.out.print(otraPersona.obtenerCodigo());
          System.out.print(" responde: ");
          otraPersona.saludar();
          System.out.println("");
        }
       }
       contador++;
     } while (contador<ocupacionMaxima);
   }
}

void restoAscensor_Despedirse(Persona unaPersona)
{
   int contador;
   Persona otraPersona;

   if (obtenerOcupacion() > 0)
```

ANOTACIONES

```
    {
      contador=0;
      do {
        if (!(pasajeros[contador]==null ))
        {
          otraPersona=pasajeros[contador];
          if (!(unaPersona.igualCodigo(otraPersona))
          {
            System.out.print(otraPersona.obtenerCodigo());
            System.out.print(" responde: ");
            otraPersona.despedirse();
            System.out.print(" ");
          }
        }
        contador++;
      } while (contador<ocupacionMaxima);
    }
}

} // class Ascensor
```

Persona.java

```
import java.io.*;

class Persona {
   private int codigo;
   private int peso;

   Persona()
   { }

   Persona(int n_codigo, int n_peso)
   {
     codigo = n_codigo;
     peso = n_peso;
   }

   public int obtenerPeso()
    { return (peso); }

   public void asignarPeso(int n_peso)
    { peso = n_peso; }

   public int obtenerCodigo()
```

ANOTACIONES

```
   { return (codigo); }
   public void asignarCodigo(int n_codigo)
{ this.codigo = n_codigo; }

   public void asignarPersona(int n_codigo, int n_peso)
   {
     asignarCodigo( n_codigo );
     asignarPeso( n_peso );
   }
   void saludar() {};
   void despedirse() {};

   public void solicitarCodigo()
   {
     int n_codigo=0;
     System.out.print ("Codigo: ");
     n_codigo = Leer.getInt();
     asignarCodigo (n_codigo);
   }

   public boolean igualCodigo(Persona otraPersona)
   {
     return
       (this.obtenerCodigo()==otraPersona.obtenerCodigo());
   }
} //class Persona
```

Catalan.java

```
class Catalan extends Persona
{

   Catalan()
   { Persona(0, 0); };

   Catalan(int n_codigo, int n_peso)
   { Persona (n_codigo, n_peso); };

   void saludar()
   { System.out.println("Bon dia"); };

   void despedirse()
   { System.out.println("Adéu"); };
}
```

ANOTACIONES

Castellano.java

```
class Castellano extends Persona
{
   Castellano()
   { Persona(0, 0); };

   Castellano(int n_codigo, int n_peso)
   { Persona (n_codigo, n_peso); };

   void saludar()
   { System.out.println("Buenos días"); };

   void despedirse()
   { System.out.println("Adiós"); };
}
```

Ingles.java

```
class Ingles extends Persona
{

   Ingles ()
   { Persona(0, 0); };

   Ingles (int n_codigo, int n_peso)
   { Persona (n_codigo, n_peso); };

   void saludar()
   { System.out.println("Hello"); };
   void despedirse()
   { System.out.println("Bye"); };
}
```

ANOTACIONES

GNU Free Documentation License

GNU Free Documentation License
Version 1.2, November 2002

Copyright (C) 2000,2001,2002 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

ANOTACIONES

ANOTACIONES

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent.

An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

ANOTACIONES

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition.

Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

ANOTACIONES

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material.

If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

ANOTACIONES

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

ANOTACIONES

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

ANOTACIONES

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number.

Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

ANOTACIONES

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit.

When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form.

Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the

ANOTACIONES

translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

ANOTACIONES

Copyright (c) YEAR YOUR NAME.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.

A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

ANOTACIONES