UNIVERSIDAD DE JAÉN
Departamento de Informática

FUNDAMENTOS BÁSICOS DE PROGRAMACIÓN EN C++

Francisco Martínez del Río

En la asignatura “Fundamentos de Programación”, del Grado en Informática de la Universidad de Jaén, se estudian una serie de conceptos y mecanismos básicos de programación comunes a cualquier lenguaje de programación. El objetivo de estos apuntes es aprender a utilizar estos conceptos y mecanismos en un lenguaje de programación concreto: el lenguaje C++. Estos apuntes constituyen un punto de partida para el estudio de C++, limitándose al análisis de estos conceptos básicos. Concretamente, no se analizarán las características orientadas a objetos de C++, éstas se estudiarán en la asignatura “Programación orientada a objetos”.

C++ deriva del lenguaje C. El lenguaje C ha sido, y sigue siendo, uno de los lenguajes de programación más utilizados. Al existir una gran cantidad de código escrito en C se consideró interesante que C++ fuera compatible con C, en el sentido de que se pudiera utilizar cualquier código C previo en un programa escrito en C++. Esta compatibilidad se ha conseguido casi al cien por cien. Sin embargo, C++ es más moderno y mejor que C, permitiendo un estilo de programación más sencillo y seguro que el estilo de C. En aquellos aspectos en los que C++ ofrece alternativas más elegantes que C—como en la entrada y salida, en los flujos o en las cadenas de caracteres—se ha preferido estudiar únicamente la interfaz de programación de C++. Si el lector necesita trabajar y entender código escrito en C, entonces necesitará estudiar todas las características de programación de C. Mientras tanto le recomendamos que estudie sólamente las características de C++, pues son más sencillas y seguras.

A lo largo de estos apuntes se estudia cómo expresar en C++ una serie de características comunes a cualquier lenguaje de programación: los tipos de datos básicos y estructurados, las variables, la entrada y salida, las expresiones aritméticas y lógicas, las sentencias condicionales e iterativas, las funciones, la recursividad y los flujos de tipo texto. La sintaxis de C++ utilizada en estos apuntes sigue el estándar C++98.

El código C++ que aparece en estos apuntes está resaltado sintácticamente. El resaltado de sintaxis es una característica visual que se aplica al código para facilitar su lectura, por ejemplo, visualizando en negrita las palabras reservadas del lenguaje. Cuando usted escriba el código en un editor de texto es posible que estas palabras reservadas no se visualicen en negrita en su editor, no se preocupe, dependerá de la configuración del editor de texto.

Índice general

Tema 1
Introducción y entrada/salida

En este tema se describe la sintaxis necesaria para escribir un primer programa en C++ y las utilidades que proporciona C++ para la lectura de datos provenientes del teclado y para la escritura de información en el monitor.

1.1. Primer programa en C++

La Figura 1.1 muestra el primer programa que vamos a desarrollar en el lenguaje C++. El resultado de la ejecución del programa es que aparece el texto `Hola a todos` en la salida estándar. La salida estándar está conectada por defecto al monitor, por lo que el citado texto debe aparecer en el monitor. A continuación se describe brevemente el contenido del programa:

- La primera línea contiene la *directiva* include. Una directiva include sirve para indicar que se va a utilizar alguna función o código implementado en una biblioteca. Una *biblioteca* es un módulo—un fichero o conjunto de ficheros—que contiene código implementado que puede ser utilizado en programas. Concretamente, nuestro programa utiliza la biblioteca iostream, que define una serie de utilidades para la realización de operaciones de entrada y salida. El nombre de la biblioteca debe situarse entre los caracteres < y >.

- La segunda línea contiene la instrucción **using namespace** std;. Acostúmbrese a incluir esta instrucción en sus programas. Su fin es poder escribir de una forma abreviada el nombre de algunos objetos definidos en la biblioteca estándar de C++. La *biblioteca estándar de C++* es una biblioteca que contiene una serie de utilidades de programación que están disponibles en cualquier distribución de C++. Si no se hubiera utilizado la instrucción **using namespace** std;, entonces los nombres cout y endl, que aparecen con posterioridad en el código, se deberían haber escrito como std :: cout y std :: endl—compruébelo.

```
#include <iostream>
using namespace std;

int main ()
{
  cout << "Hola a todos" << endl;
  return 0;
}
```

Figura 1.1: Primer programa en C++.

- A continuación viene el programa principal o función main. Un programa en C++ puede constar de una o varias funciones de código. La función main siempre debe estar definida y contiene las instrucciones que empieza a ejecutar el programa. El conjunto de instrucciones que contiene una función, también llamado *cuerpo de la función*, debe encerrarse entre llaves.

- La primera instrucción del cuerpo de main es la que envía el texto `Hola a todos` a la salida estándar. Esta instrucción, al igual que toda instrucción en C++, debe terminar con el carácter punto y coma.

- La instrucción **return** 0; termina la ejecución de la función main, devolviendo el valor 0. Este valor de retorno no nos interesa de momento, pero puede ser consultado por el programa que ejecutó a nuestro programa. Esta última instrucción **return** no es estrictamente necesaria. Si no la incluye, es posible que reciba una advertencia por parte del compilador, porque main es una función que devuelve un entero y una función main que no incluya una sentencia **return** no devuelve nada.

A la hora de escribir el programa en el editor de texto tenga en cuenta que C++ es un lenguaje sensible a las mayúsculas. Esto quiere decir que se tiene en cuenta si una letra se escribe en mayúsculas o minúsculas. Si, por ejemplo, escribió Main en lugar de main, entonces su programa no será correcto sintácticamente.

1.2. Tipos de datos básicos de C++

La Tabla 1.1 enumera los principales tipos de datos básicos o primitivos del lenguaje C++. Un *tipo de dato básico o primitivo* es aquél que viene soportado directamente por el lenguaje. El rango de representación de cada tipo depende del modelo de ordenador en que se compila el programa. Por ejemplo, si su ordenador almacena un entero utilizando 32 bits entonces el rango válido para un entero es de $[2^{31}-1, -2^{31}]$, es decir, $[2{,}147{,}483{,}647, -2{,}147{,}483{,}648]$. Como puede observar, existen dos tipos de datos para representar un número de tipo real, el

Pseudocódigo	C++
entero	**int**
real	**float** y **double**
carácter	**char**
lógico	**bool**

Tabla 1.1: Principales tipos de datos básicos de C++

```
#include <iostream>
#include <limits>

using namespace std;

int main () {
  cout << "Máximo entero: " << numeric_limits<int>::max () << endl;
  cout << "Mínimo entero: " << numeric_limits<int>::min () << endl;
  cout << "Máximo float: " << numeric_limits<float>::max () << endl;
  cout << "Mínimo float: " << -numeric_limits<float>::max () << endl;
  cout << "Máximo double: " << numeric_limits<double>::max () << endl;
  cout << "Mínimo double: " << -numeric_limits<double>::max () << endl;
  cout << "Máx. entero largo: " << numeric_limits<long int>::max() << endl;
  cout << "Mín. entero largo: " << numeric_limits<long int>::min() << endl;
  return 0;
}
```

Figura 1.2: Programa que muestra algunos límites numéricos.

tipo **double** tiene un rango de representación superior al **float**—es decir, puede representar más números—y ocupa más espacio de almacenamiento.

Se puede modificar el rango de representación de algunos tipos básicos aplicándoles los modificadores **unsigned**, **short** y **long**, consulte un manual si está interesado en esta posibilidad.

El programa de la Figura 1.2 muestra los valores máximos y mínimos que pueden representar los tipos de datos **int**, **float**, **double** y **long int** en el ordenador en que se ejecuta el programa.

1.3. Definición de variables y operador de asignación

El operador de asignación de una expresión a una variable en C++ es el carácter =. Para definir una variable hay que utilizar una expresión del tipo:

```
tipo nombre_var;
```

```
#include <iostream>
using namespace std;

int main ()
{
  int x;
  double d = 2.5;
  cout << x << ' ' << d << endl;
  x = 8;
  cout << x << ' ' << d << endl;
  return 0;
}
```

Figura 1.3: Definición y asignación de valores a variables.

donde tipo es un tipo válido y nombre_var es el identificador de la variable. Existen ciertas reglas que definen los identificadores de variable válidos. Por ejemplo, un identificador de una variable no puede coincidir con el nombre de una palabra reservada del lenguaje, como **return** o **case**. Puede consultar en un manual estas reglas. En general, puede utilizar nombres de variables que incluyan caracteres del alfabeto, salvo la ñ, y números y algún carácter especial como _. No empiece el nombre de una variable con un número. Por ejemplo, nombre1 o tu_nombre son nombres válidos de variables, pero no lo son 1nombre o tu.nombre.

La Figura 1.3 contiene un programa que define dos variables. A la variable d se le asigna un valor en el momento de su definición, mientras que la variable x es definida en primer lugar y después se le asigna el valor 8. Las sentencias que comienzan por cout muestran el valor de las dos variables separadas por un espacio en blanco. Observe que x aún no está iniciada la primera vez que se muestra su valor, por lo que se mostrará un valor indefinido. Se pueden definir más de una variable del mismo tipo en una única expresión de definición de variables, separando sus identificadores mediante comas. Por ejemplo, la siguiente expresión define dos variables de tipo entero:

```
int x,y;
```

1.4. Salida de datos y comentarios

En C++ se puede enviar información a la salida estándar que, como se explicó previamente, está conectada por defecto al monitor o pantalla, mediante una instrucción del tipo:

```
cout << expresión1 << expresión2 << ... << expresiónn;
```

Esta instrucción se encarga de convertir el resultado de cada expresión en una cadena de caracteres y de enviar las cadenas de caracteres a la salida estándar. Si se utiliza endl como

```
#include <iostream>
using namespace std;
/* Programa que muestra un número
   y su cuadrado */
int main ()
{
  int x = 2;
  cout << "El número es: " << x << endl; // muestra x
  cout << "Su cuadrado vale: " << x*x << endl; // y su cuadrado
  return 0;
}
```

Figura 1.4: Ejemplo de instrucciones de salida y comentarios.

expresión, entonces se envía un salto de línea a la salida estándar. El programa de la Figura 1.4 ilustra el uso de esta instrucción, utilizándose expresiones de tipo cadena de caracteres y de tipo entero. Este programa también muestra cómo se pueden utilizar comentarios en C++. Un *comentario* es un texto que sirve para que el programador pueda documentar su código. Los comentarios no son analizados por el compilador. Existen dos tipos de comentarios:

- Los encerrados entre /* y */. Pueden ocupar más de una línea. No se pueden anidar, es decir, no se puede incluir un comentario dentro de otro.
- Los que empiezan con //. Una vez escritos estos dos caracteres lo que queda de línea se considera un comentario.

Volvamos al programa de la Figura 1.4. Observe que para enviar a la salida estándar una secuencia de caracteres estos deben aparecer delimitados por comillas dobles, como por ejemplo: "Su cuadrado vale: " en la segunda sentencia cout. De acuerdo con Bjarne Stroustup, el creador de C++, el nombre cout viene de *Character OUTput*, es decir, salida de caracteres o texto.

1.5. Lectura o entrada de datos

La lectura de datos se realiza mediante la siguiente instrucción:

```
cin >> variable1 >> variable2 >> ... >> variablen;
```

donde variable1, variable2 y variablen son variables. Esta instrucción provoca la lectura de datos desde la entrada estándar, que por defecto está asociada al teclado, y su almacenamiento en las variables especificadas en la instrucción. La instrucción lee los datos en

```
#include <iostream>
using namespace std;

int main ()
{
  char c;
  float f;
  cout << "Introduzca un carácter: ";
  cin >> c;
  cout << "Introduzca un número real: ";
  cin >> f;
  cout << "Carácter: " << c << " Número real: " << f << endl;
  return 0;
}
```

Figura 1.5: Ejemplo de instrucciones de entrada.

formato de texto y los convierte a la representación interna adecuada al tipo de las variables. Por ejemplo, a complemento a dos si el dato leído se almacena en una variable entera.

Esta instrucción supone que los datos de entrada se separan mediante uno o varios *caracteres blancos*. Se entiende por carácter blanco el espacio en blanco, el tabulador y el salto de línea. La Figura 1.5 muestra un programa que utiliza cin para leer dos datos. cin también permite la lectura de datos separados mediante otros caracteres, pero esto se estudiará más adelante—Tema 10, Sección 10.3. No obstante, la mayoría de las veces los datos de entrada vienen separados mediante caracteres blancos. El nombre cin viene de *Character INput*.

El uso de las instrucciones cin y cout para la lectura y escritura de información es una característica de C++ que no se encuentra en el lenguaje C. En C, la lectura y escritura de información se realiza principalmente mediante las funciones scanf y printf respectivamente. Si tiene interés en estas funciones consulte un manual de C.

1.6. Ejercicios

1. Escribe un programa que lea de la entrada estándar dos números y muestre en la salida estándar su suma, resta, multiplicación y división. Nota: suponiendo que las variables x e y almacenan dos números, las expresiones x+y, x−y, x∗y y x/d calculan la suma, resta, producto y división respectivamente de los valores almacenados en las variables. Por ejemplo, suponiendo que el usuario introduce los números 6 y 3 el programa mostrará lo siguiente:

   ```
   Suma: 9
   Resta: 3
   ```

```
Producto: 18
Division: 2
```

2. Escribe un programa que lea de la entrada estándar el precio de un producto y muestre en la salida estándar el precio del producto al aplicarle el IVA.

3. Realice un programa que lea de la entrada estándar los siguientes datos de una persona:

   - Edad: dato de tipo entero.
   - Sexo: dato de tipo carácter.
   - Altura en metros: dato de tipo real.

   Tras leer los datos, el programa debe mostrarlos en la salida estándar. Tenga en cuenta que un dato de tipo carácter—**char**—sólo puede almacenar un carácter, por lo que como usuario del programa sólo puede introducir un carácter para el sexo—por ejemplo, M o F.

4. Ejecute el programa del ejercicio anterior con entradas erróneas y observe los resultados. Por ejemplo, introduzca un dato de tipo carácter cuando se espera un dato de tipo entero.

Tema 2
Expresiones

En este tema se analiza la forma de escribir expresiones aritméticas en C++. Las expresiones aritméticas son importantes porque permiten expresar cálculos. Estudiaremos varios conceptos claves para la correcta escritura de una expresión, como los tipos de operadores aritméticos, su precedencia, el uso de paréntesis en expresiones o las conversiones implícitas y explícitas de tipos.

2.1. Expresiones aritméticas y precedencia

Una expresión aritmética es aquella formada por variables, constantes, literales de tipo numérico, operadores aritméticos y llamadas a función. La Tabla 2.1 enumera los principales operadores aritméticos de C++. Los operadores + y − se pueden aplicar tanto a uno como a dos operandos. Si se aplican a un operando indican signo y si se aplican a dos operandos la operación matemática de suma y resta respectivamente. El operador resto o módulo produce el resto de la división entera, sólo es aplicable a operandos de tipo entero.

La Tabla 2.2 muestra la precedencia de los operadores aritméticos ordenados de mayor a menor precedencia, los operadores que aparecen en la misma fila tienen la misma precedencia. Se puede utilizar paréntesis para alterar el orden en el que se evalúan los operadores. Una subexpresión encerrada ente paréntesis se evalúa antes que otra subexpresión no en-

Operador	**Tipo de operación**
+	suma binaria y más unitario
-	resta binaria y menos unitario
*	multiplicación
/	división
%	resto o módulo

Tabla 2.1: Principales operadores aritméticos de C++

Operador	Asociatividad
+ y - unitario	de derecha a izquierda
* / %	de izquierda a derecha
+ - binarios	de izquierda a derecha

Tabla 2.2: Precedencia y asociatividad de los operadores aritméticos de C++

```
#include <iostream>

int main ()
{
  std::cout << 4 + 2 * 3 << ' ' << (4 + 2) * 3 << std::endl;
  std::cout << 12 / 2 * 3 << ' ' << 12 / (2 * 3) << std::endl;
  return 0;
}
```

Figura 2.1: Precedencia de los operadores.

cerrada entre paréntesis. Los paréntesis se pueden anidar. Por ejemplo, en el programa de la Figura 2.1 la primera instrucción cout evalúa dos expresiones. En la expresión `4 + 2 * 3`, que vale 10, la multiplicación se realiza en primer lugar, pues el operador de multiplicación tiene mayor precedencia que el operador de suma. Sin embargo, en la expresión `(4 + 2) * 3`, que vale 18, los paréntesis provocan que la suma se realice en primer lugar.

La segunda instrucción cout del programa de la Figura 2.1 ilustra la asociatividad de los operadores. Si en una expresión aparecen varios operadores de igual precedencia, se utiliza el tipo de asociatividad de los operadores para establecer el orden de evaluación. Por ejemplo, en la expresión `12 / 2 * 3`, que vale 18, la división se realiza antes que la multiplicación, pues los operadores de división y multiplicación tienen igual precedencia, la asociatividad de estos operadores es de izquierda a derecha y el operador de división está situado más a la izquierda que el operador de multiplicación. Se puede utilizar paréntesis para establecer otro orden de ejecución, como en `12 / (2 * 3)`—se obtiene 2.

2.2. Conversiones de tipos

El *hardware* de un ordenador permite la realización de operaciones aritméticas entre operandos del mismo tipo básico, pero no entre operandos de distintos tipos básicos. Por ello, cuando en una operación aritmética los operandos son de distinto tipo, C++ realiza una conversión implícita del operando cuyo tipo tiene un rango de representación menor al tipo del operando cuyo rango de representación es mayor; de esta forma se puede realizar la operación. El resultado de la operación será del tipo del operando con mayor rango de

```
#include <iostream>

int main () {
  int x = 32;
  std::cout << x / 5 << ' ' << (float) x / 5 << std::endl;
  std::cout << float (x) / 5 << std::endl;
  return 0;
}
```

Figura 2.2: Conversión explícita de tipos.

representación. Por ejemplo, en una operación entre un número entero y un número real el entero se *promociona* o convierte implícitamente a real, realizándose la operación entre reales y siendo el tipo del resultado de la operación real. Del mismo modo, en una operación entre un **float** y un **double** el dato de tipo **float** se promociona a **double**.

Cuando se realiza una asignación, si el tipo de la variable a la que se le asigna el valor difiere del tipo del resultado de la expresión, entonces se realiza una conversión implícita del resultado de la expresión al tipo de la variable.

El programa de la Figura 2.2 muestra una *conversión explícita de tipos*. En la primera expresión: `x / 5`, ambos operandos son de tipo entero. Por lo tanto, se realiza una división entera cuyo resultado es 6. En la segunda expresión: `(float) x / 5`, se utiliza el operador de conversión explícita de tipos. Éste es un operador unitario cuya sintaxis es:

```
(tipo) operando
```

donde `tipo` es un tipo del lenguaje. El efecto del operador es convertir `operando` al tipo `tipo`. En el ejemplo, esto hace que el valor 32, que está representado como entero, se convierta a **float**. A continuación, se divide un **float** entre un **int**, con lo que el **int** se convierte implícitamente a **float** y se realiza la división real, de resultado 6.4.

El operador de conversión explícita de tipos es una herencia de C. En C++ también se puede utilizar la sintaxis, más legible, de la última sentencia de salida del programa de la Figura 2.2: `tipo (expresion)`, que convierte el resultado de evaluar la expresión al tipo `tipo`, siempre que la conversión esté definida.

2.3. Expresiones lógicas

Una expresión lógica o booleana es aquella que produce un resultado de verdadero o falso. Las expresiones lógicas se emplean principalmente para evaluar si se verifican ciertas condiciones y, en función de ello, ejecutar un código u otro. Por ello, estudiaremos este tipo de expresiones en el siguiente tema, en el que se explican las estructuras condicionales.

```
#include <iostream>

int main () {
  int x, y;
  x = 4;
  y = 7;
  std::cout << x << ' ' << y << std::endl;
  x = y + 3;
  std::cout << x << ' ' << y << std::endl;
  return 0;
}
```

Figura 2.3: Operador de asignación.

2.4. El operador de asignación y de incremento

El operador de asignación (el operador =) se utiliza para asignar un valor a una variable. Su sintaxis es la siguiente:

```
variable = expresion
```

Su efecto es evaluar la expresión y asignar el resultado de la expresión a la variable que aparece a la izquierda del operador =. La operación de asignación se dice que es destructiva, en el sentido de que la variable pierde el valor que tenía asociado al serle asignado un nuevo valor. Así, en el programa de la Figura 2.3 cuando se ejecuta la asignación `x = y + 3;` la variable `x` tiene asociado el valor 4, y tras la ejecución de la instrucción pasa a tener asociado el valor 10, el valor previo, 4, "se pierde".

En C++ una asignación es una expresión que produce como resultado el valor asignado a la variable. Por lo tanto, una asignación puede formar parte de una expresión más compleja. Por ejemplo, en el programa de la Figura 2.4, en la expresión: `(x = 2) * 10` se asigna el valor 2 a la variable `x`. El resultado de la subexpresión `x = 2` es 2, que se multiplica después por 10, produciendo 20 como resultado final de la expresión. Si utiliza el operador de asignación en expresiones complejas, tenga en cuenta que su precedencia es baja. Pruebe a eliminar los paréntesis del ejemplo anterior y observe que en ese caso se asigna el valor 20 a `x`. Se asigna ese valor porque en primer lugar se realizaría la multiplicación de 2 por 10 y, a continuación, la asignación de 20 a `x`.

El operador de asignación se asocia de derecha a izquierda. Por lo tanto, la penúltima instrucción del programa de la Figura 2.4 asigna a las tres variables—`a`, `b` y `c`—el valor 6.

Si se desea escribir una expresión como `x = x + 8`, en la que se incrementa el valor de una variable en una cantidad, entonces se puede utilizar el operador `+=` de la siguiente forma: `x += 8`. Esta última expresión incrementa la variable que aparece a la izquierda del operador `+=` en una cantidad de veces igual al resultado de evaluar la expresión situada a la derecha

```
#include <iostream>

int main () {
  int x, a, b, c;
  std::cout << (x = 2) * 10 << std::endl;
  std::cout << x << std::endl;
  a = b = c = 6; /* equivale a a = (b = (c = 6));
                    o lo que es lo mismo: a = 6; b = 6; c = 6; */
  return 0;
}
```

Figura 2.4: Otro ejemplo del operador de asignación.

del operador. El operador += resulta útil, pues reduce el tamaño de la expresión y facilita su lectura. Por ejemplo, nombreLargo += 6 resulta más corto que nombreLargo = nombreLargo + 6, y más legible, porque en la segunda expresión hay que comprobar que el nombre de las dos variables es el mismo. Los operadores −=, ∗=, /= y %= tienen un comportamiento análogo a += pero aplicados a la resta, multiplicación, división y módulo, respectivamente.

El incremento del valor de una variable en una unidad es una operación tan común que C++ proporciona el operador unitario de incremento, ++, para expresar esta operación de una manera concisa. Dada una variable var, las expresiones ++var o var++ incrementan el valor de la variable var en una unidad. En el programa de la Figura 2.5 la variable x se incrementa en dos unidades tras la ejecución de las instrucciones x++; y ++x;. Al incremento del tipo ++x se le denomina incremento prefijo y al incremento del tipo x++ posfijo. Independientemente del incremento de la variable, el operador de incremento produce un resultado, que será utilizado si el operando forma parte de una expresión compleja. Los operadores prefijo y posfijo producen distinto resultado; el operador prefijo produce como resultado el valor de la variable después del incremento, mientras que el operador posfijo produce el valor de la variable antes del incremento. Ejecute el programa de la Figura 2.5 y observe cómo la expresión y++ ∗ 3 produce como resultado 9, mientras que ++z ∗ 3 produce como resultado 12.

El operador −− decrementa en una unidad y es análogo al operador ++. Como nota curiosa, en el nombre del lenguaje C++ se utiliza el operador ++ para indicar que C++ es una versión mejorada del lenguaje C. Hay quien dice que un lenguaje ++C sería aún mejor.

2.5. Literales

Un literal es un valor concreto de un tipo que aparece en una expresión. Por ejemplo, en la expresión: x∗y + 2 el 2 es un literal de tipo entero. Si en una expresión aparece un número sin decimales, entonces el número es un literal entero. Si el número contiene un punto decimal,

```
#include <iostream>

int main () {
  int x = 3, y = 3, z = 3;
  x++;
  ++x;
  std::cout << x << std::endl;
  std::cout << y++ * 3 << std::endl;
  std::cout << y << std::endl;
  std::cout << ++z * 3 << std::endl;
  std::cout << z << std::endl;
  return 0;
}
```

Figura 2.5: Operador de incremento.

como 2.0, entonces es un literal de tipo **double**. Para utilizar un literal de tipo **float** hay que terminar el número con el sufijo f. Por ejemplo, 2.0f es un literal de tipo **float**.

Un literal de tipo cadena de caracteres se escribe rodeando una secuencia de caracteres mediante comillas dobles, por ejemplo: "hola". El tipo **bool** sólo admite dos literalas: **true** y **false**. Un literal de tipo **char** se escribe rodeando el carácter mediante comillas simples, por ejemplo: 'a'. Un literal de carácter que se utiliza con bastante frecuencia es '\n', que representa al carácter salto de línea.

2.6. Constantes

Una constante es un variable que almacena un valor que no varía a lo largo de la ejecución de un programa. En C++ se puede definir una constante anteponiendo al tipo de la variable la palabra reservada **const**. El programa de la Figura 2.6 ejemplifica el uso de una constante. Por convención, los nombres de variables constantes se escriben en mayúsculas. En C++ una variable constante debe iniciarse en el momento de su definición. Una vez definida no es posible asignarle otro valor.

2.7. Llamadas a funciones

Una función es un fragmento de código parametrizado. Para ejecutar el código asociado a una función hay que *llamar a la función* en una expresión. Llamar a una función consiste en escribir el nombre de la función y, entre paréntesis y separados por comas, los datos o parámetros con que queremos que se ejecute. Como resultado de la llamada o invocación, se ejecuta el código asociado a la función procesándose los parámetros para producir un

```
#include <iostream>
using namespace std;

int main () {
  const double IVA_LIBRO = 0.04;
  double precio;
  cout << "Introduzca el precio del libro: ";
  cin >> precio;
  cout << "Precio con IVA: " << precio + precio*IVA_LIBRO << '\n';
  return 0;
}
```

Figura 2.6: Constantes.

valor de salida. Este valor de salida es el resultado de la llamada a la función y sustituye a la llamada a la función en la expresión.

En el tema 6 aprenderemos a escribir nuestras propias funciones. Por el momento nos conformaremos con utilizar en nuestros programas las funciones de la biblioteca estándar de C++. En esta biblioteca existe una gran cantidad de funciones de utilidad.

En el programa de la Figura 2.7 se incluyen dos llamadas a las funciones pow y sqrt de la biblioteca estándar. Ambas están declaradas en la biblioteca cmath, que incluye utilidades para el cálculo matemático, por lo que hay que incluir la directiva: include <cmath>. Observe la invocación de las funciones: pow calcula la potencia de un número y tiene dos parámetros, la base y el exponente. sqrt—SQuare RooT—calcula la raíz cuadrada de un número que constituye su único parámetro. Si ejecutamos el programa e introducimos como entrada 4 se obtiene el siguiente resultado:

```
x^3+4 vale 68
Su raiz cuadrada vale 2
```

2.8. Ejercicios

En algunos de los ejercicios enumerados a continuación se ha incluido una posible entrada al programa con su correspondiente salida. El objeto es que el lector pueda comprobar si su programa funciona correctamente para dicha entrada. El lector debe, no obstante, acostumbrarse a pensar en casos de prueba para comprobar que su programa realiza los cálculos apropiados. Los ejercicios del 2 al 5 vienen resueltos al final, pero también puede escribir un programa para comprobar si sus respuestas son correctas.

En algunos ejercicios hay que calcular la raíz cuadrada de un número. Esto se puede realizar con la función sqrt que toma como parámetro un valor y devuelve su raíz cuadrada.

```
#include <iostream>
#include <cmath>
using namespace std;

int main () {
    double x, r;
    cout << "Introduce un numero: ";
    cin >> x;
    r = pow(x,3)+ 4;
    cout << "x^3+4 vale " << r << endl;
    cout << "Su raiz cuadrada vale " << sqrt(x) << endl;
    return 0;
}
```

Figura 2.7: Llamadas a funciones de la biblioteca estándar.

Por ejemplo, sqrt(4) devuelve 2. Esta función se encuentra definida en la biblioteca cmath; para poder utilizarla hay que realizar el siguiente include:

```
#include <cmath>
```

Otra función que será preciso utilizar en algún ejercicio es pow, que sirve para calcular potencias. Esta función también está definida en la biblioteca cmath y se invoca así: pow(base,exponente). Por ejemplo, pow(2,3) devuelve 8.

1. Escribe las siguientes expresiones matemáticas como expresiones en C++:

   a) $\frac{a}{b}+1$ **b)** $\frac{a+b}{c+d}$ **c)** $\frac{a+\frac{b}{c}}{d+\frac{e}{f}}$ **d)** $a+\frac{b}{c-d}$ **e)** $(a+b)\frac{c}{d}$ **f)** $\frac{-b\pm\sqrt{b^2-4ac}}{2a}$

2. Calcule el valor de la variable result, de tipo **double**, tras la ejecución de las siguientes asignaciones—la función pow de la biblioteca estándar de C++ eleva su primer argumento a la potencia indicada por su segundo argumento:

   a)
   ```
   result = 3.0 * 5.0;
   ```

   b)
   ```
   x = 2.0;
   y = 3.0;
   result = pow(x,y)-x;
   ```

   c)
   ```
   result = 4.0;
   x = 2.0;
   result = result * x;
   ```

3. ¿En cuáles de los siguientes pares de asignaciones es importante el orden en que aparecen las asignaciones? Es decir, si se modifica el orden cambian los resultados finales (suponga que $x \neq y \neq z$):

   a)
   ```
   x = y;
   y = z;
   ```

   b)
   ```
   x = z;
   x = y;
   ```

   c)
   ```
   x = y;
   z = x;
   ```

   d)
   ```
   z = y;
   x = y;
   ```

4. ¿Qué valor se obtiene en las variables a, b y c tras la ejecución de las siguientes instrucciones?

   ```
   a = 3;
   b = 20;
   c = a+b;
   b = a+b;
   a = b;
   ```

5. Escriba un fragmento de programa que intercambie los valores de dos variables. Sugerencia: apóyese en una variable auxiliar.

6. La calificación final de un estudiante es la media ponderada de tres notas: la nota de prácticas que cuenta un 30 % del total, la nota teórica que cuenta un 60 % y la nota de participación que cuenta el 10 % restante. Escriba un programa que lea de la entrada estándar las tres notas de un alumno y escriba en la salida estándar su nota final.

   Entrada caso de prueba: Nota de prácticas (6), nota de teoría (8) y nota de participación (10).

   Salida caso de prueba: Nota final (7.6).

7. Escriba un programa que lea la nota final de cuatro alumnos y calcule la nota final media de los cuatro alumnos.

   Entrada caso de prueba: Notas: 5.6, 6, 10 y 7.2.

   Salida caso de prueba: Nota media: 7.2.

8. Cuando se compra una cajetilla de tabaco en una máquina y no se introduce el importe exacto, la máquina utiliza un programa para devolver el mínimo número de monedas. Escriba un programa—considerando únicamente monedas de 5, 10, 20 y 50 céntimos de euro—que lea de la entrada estándar el importe de una cajetilla de tabaco y la cantidad de dinero introducida por el comprador en la máquina y escriba en la salida estándar la monedas devueltas por la máquina.

9. Escriba un programa que lea de la entrada estándar los dos catetos de un triángulo rectángulo y escriba en la salida estándar su hipotenusa.

   Entrada caso de prueba: Logitud de los catetos: 3 y 4.

   Salida caso de prueba: Longitud de la hipotenusa: 5.

10. Realice un programa que calcule el valor que toma la siguiente función para unos valores dados de x e y:

$$f(x, y) = \frac{\sqrt{x}}{y^2 - 1}$$

    Entrada caso de prueba: $x = 10$, $y = 3$.

    Salida caso de prueba: $f(x, y) = 0{,}395$.

11. Escriba un programa que calcule las soluciones de una ecuación de segundo grado de la forma $ax^2 + bx + c = 0$, teniendo en cuenta que:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

    Entrada caso de prueba: $2x^2 + 9x + 3 = 0$.

    Salida caso de prueba: Raíces: -0.36 y -4.14.

12. ¿Pueden existir entradas para las que los dos programas previos no funcionen?

13. Suponiendo que el recibo de la luz sube un 3 % cada año, realice un programa que solicite una factura de este año y una cantidad de años y muestre en la salida estándar cuánto valdría la factura dentro del número de años introducidos.

    Entrada caso de prueba: Factura: 100 euros. Número de años: 3.

    Salida caso de prueba: Factura: 109.273.

14. Escriba un programa que calcule la desviación estándar de cinco números:

$$\sigma = \sqrt{\frac{1}{4}\sum_{i=1}^{5}(x_i - \overline{x})^2}$$

Entrada caso de prueba: 1, 4.5, 7, 3 y 15.

Salida caso de prueba: Desviación estándar: 5.436.

Soluciones a los ejercicios que no son programas

1. a) a/b + 1
   b) (a+b)/(c+d)
   c) (a+b/c)/(d+e/f)
   d) a+ b/(c−d)
   e) (a+b)∗c/d
   f) (−b+sqrt(b∗b − 4∗a∗c))/(2∗a) y (−b−sqrt(b∗b − 4∗a∗c))/(2∗a)

2. a) 15.0 b) 6.0 c) 8.0

3. En todas salvo en la d)

4. Todas valen 23

5.
```
// Se intercambia los valores de x e y
int tmp = x;
x = y;
y = tmp;
```

Tema 3
Estructuras condicionales

En este tema se estudian las estructuras condicionales o selectivas de C++: las sentencias **if** y **switch**. Las estructuras condicionales permiten ejecutar código de manera selectiva. Es decir, si se dan ciertas condiciones se ejecuta un determinado código.

3.1. La sentencia if

La sentencia **if** permite ejecutar código de manera selectiva en función del resultado de evaluar una condición lógica. Su sintaxis es la siguiente:

```
if (expresión lógica) {
   conjunto de instrucciones 1
} else {
   conjunto de instrucciones 2
}
```

El funcionamiento de la instrucción **if** se describe a continuación. En primer lugar se evalúa la expresión lógica. Si la expresión es verdadera, entonces se ejecuta el conjunto de instrucciones 1, en caso de ser falsa se ejecuta el conjunto de instrucciones 2. Sobre la sintaxis de la instrucción **if** tenga en cuenta lo siguiente:

- La expresión lógica debe aparecer encerrada entre paréntesis.
- Los conjuntos de instrucciones 1 y 2 se encierran entre llaves. Si un conjunto de instrucciones viene constituido por una única instrucción, entonces no es estrictamente necesario utilizar llaves para delimitar ese conjunto de instrucciones.
- La parte **else** es opcional.
- No existen restricciones sintácticas para el lugar en que se escriben las llaves y los conjuntos de instrucciones. Elija una forma que le guste y utilícela consistentemente. Se recomienda escribir los conjuntos de instrucciones un poco a la derecha—sangrados o indentados—, aumentará la legibilidad del programa.

```
#include <iostream>
using namespace std;

int main ()
{
  const char VOCAL = 'e';
  char vocal;
  cout << "Introduzca una vocal: ";
  cin >> vocal;
  if (vocal == VOCAL) {
     cout << "Felicidades, la ha acertado\n";
  } else {
     cout << "Mala suerte\n";
  }
  return 0;
}
```

Figura 3.1: Ejemplo de uso de la instrucción **if**.

La Figura 3.1 contiene un ejemplo de utilización de la instrucción **if**. En este ejemplo los conjuntos de instrucciones tienen una única instrucción, por lo que no es estrictamente necesario encerrarlos entre llaves. El programa lee una vocal de la entrada estándar y muestra un mensaje en la salida estándar en función de si la vocal leída coincide con una vocal prefijada. Si coincide muestra el mensaje `Felicidades, la ha acertado` y si no coincide muestra el mensaje `Mala suerte`.

3.2. Expresiones lógicas

Una expresión lógica es aquella que produce un resultado de verdadero o falso. Se puede construir una expresión lógica simple utilizando un literal de tipo lógico, es decir, **true** o **false**, o combinado dos expresiones aritméticas mediante un operador relacional. La Tabla 3.1 muestra los operadores relacionales de C++. El programa de la Figura 3.2 ilustra el uso de operadores relaciones. Este programa lee dos números enteros y determina cuál es el mayor de los dos.

En C++ también se puede utilizar una expresión aritmética como expresión lógica. En ese caso se evalúa la expresión aritmética en primer lugar. Si el resultado de evaluar la expresión aritmética es un número distinto de cero se considera que la expresión lógica es verdadera, en otro caso la expresión lógica es falsa. El programa de la Figura 3.3 ilustra el uso de una expresión aritmética como expresión lógica. El programa lee dos números enteros y determina si son distintos, para ello utiliza la expresión: $x - y$, que produce un valor distinto de cero si los números son diferentes. Este programa también contiene una segun-

Operador	Significado
<	menor que
>	mayor que
<=	menor o igual que
>=	mayor o igual que
==	igual que
!=	distinto de

Tabla 3.1: Operadores relacionales de C++

```
#include <iostream>
using namespace std;

int main () {
  int x, y;
  cout << "Introduzca dos valores enteros: ";
  cin >> x >> y;
  if (x > y) {
     cout << "El primer valor es el mayor" << endl;
  } else {
     if (y > x) {
        cout << "El segundo valor es el mayor" << endl;
     } else {
        cout << "Los dos valores son iguales" << endl;
     }
  }
  return 0;
}
```

Figura 3.2: Ejemplo de uso de operadores relaciones.

da instrucción **if** que utiliza un literal en la condición. Como el literal es **true**, la condición lógica siempre es cierta. En lugar de utilizar la expresión aritmética $x - y$ para comprobar si los números son diferentes el programa podría haber utilizado la expresión lógica `x != y`.

3.2.1. Expresiones lógicas compuestas y sentencias condicionales anidadas

Una expresión lógica compuesta es aquella formada por expresiones lógicas combinadas mediante operadores lógicos. La Tabla 3.2 muestra los operadores lógicos de C++. Una expresión lógica compuesta puede contener operadores aritméticos, relacionales y lógicos. Debe tener en cuenta la precedencia de estos operadores para poder construir correctamente la expresión lógica, utilizando paréntesis cuando sea necesario. La Tabla 3.3 muestra todos es-

```
#include <iostream>

int main () {
  int x, y;
  std::cout << "Introduzca dos valores enteros: ";
  std::cin >> x >> y;
  if (x - y)
     std::cout << "Los valores son distintos\n";
  if (true)
     std::cout << "Este mensaje siempre aparece\n";
  return 0;
}
```

Figura 3.3: Ejemplo de uso de expresión aritmética como expresión lógica.

Operador	Operador en C++
Y	&&
O	||
NO	!

Tabla 3.2: Operadores lógicos de C++

tos tipos de operadores ordenados de mayor a menor precedencia. La Figura 3.4 muestra un programa que contiene una expresión lógica compuesta. El programa lee una nota numérica y muestra en la salida estándar su valor correspondiente dentro del siguiente conjunto de calificaciones: suspenso, aprobado, notable y sobresaliente. En el programa la expresión lógica compuesta—nota < 0 || nota > 10—comprueba si la nota está fuera del rango de notas válido [0,10]. El programa también ilustra el uso de *instrucciones if anidadas*, es decir, instrucciones **if** cuyo código asociado contiene, a su vez, otras instrucciones **if**.

El programa de la Figura 3.5 contiene dos ejemplos más de expresiones lógicas compuestas. El programa solicita al usuario un carácter y determina si el carácter introducido es una vocal cerrada, abierta o no es una vocal. Para decidir el tipo de carácter se hace uso de dos expresiones lógicas compuestas. La primera comprueba—primer **if**—si el carácter introducido es una vocal cerrada, para ello se utiliza el operador *or*. Observe que como la variable c solo contiene un valor, a lo sumo uno de los operandos de la expresión puede ser verdadero. La segunda condición lógica comprueba—segundo **if**—si el carácter introducido es una vocal abierta. La expresión contiene dos operadores *or*. Recuerde que este operador es binario. El operador *or* se asocia de izquierda a derecha, por lo que la expresión equivale a:

```
(c == 'a' || c == 'e') || c == 'o'
```

Es decir, el operador de la izquierda se aplica primero. Estudie la expresión hasta que llegue a comprender que es cierta cuando la variable c contiene una vocal abierta.

```
#include <iostream>
using namespace std;

int main () {
  double nota;
  cout << "Introduzca la nota: ";
  cin >> nota;
  if (nota < 0 || nota > 10) {
     cout << "Nota no válida\n";
  } else {
     if (nota < 5.0) {
        cout << "Suspenso\n";
     } else {
        if (nota < 7.0) {
           cout << "Aprobado\n";
        } else {
           if (nota < 9.0) {
               cout << "Notable\n";
           } else {
               cout << "Sobresaliente\n";
           }
        }
     }
  }
  return 0;
}
```

Figura 3.4: Ejemplo de uso de expresión lógica compuesta.

Operador	Propósito
++	Incremento postfijo
--	Decremento postfijo
++	Incremento prefijo
--	Decremento prefijo
! + -	Negación lógica y signo
()	Conversión explícita
* / %	Multiplicativos
+ -	Aditivos
< > <= >=	Relacionales
== !=	Operadores de igualdad
&&	Y lógico
||	O lógico
?:	Operador condicional
= += -= *= /= %=	Operadores de asignación

Tabla 3.3: Precedencia de los operadores aritméticos, relacionales y lógicos

```
#include <iostream>
using namespace std;

int main() {
    char c;
    cout << "Introduzca una vocal: ";
    cin >> c;
    if (c == 'i' || c == 'u') {
        cout << "Introdujo una vocal cerrada" << endl;
    } else {
        if (c == 'a' || c == 'e' || c == 'o') {
            cout << "Introdujo una vocal abierta" << endl;
        } else {
            cout << "No introdujo una vocal" << endl;
        }
    }
    return 0;
}
```

Figura 3.5: Otro ejemplo de expresión lógica compuesta: determinar tipo de vocal.

3.2.2. Un error muy común

En esta sección se describe un error muy común que ocurre cuando el programador escribe una expresión lógica del tipo:

```
if (x = 5) ...
```

cuando realmente quería comprobar si la variable x vale 5. El error consiste en utilizar el operador de asignación (=) en lugar del operador de igualdad (==). El caso es que la expresión x = 5 es una expresión válida, pues es una expresión aritmética que se interpreta como una expresión lógica. El efecto de esta expresión es asignar el valor 5 a la variable x y producir como resultado de la expresión el valor asignado, o sea, 5. Como 5 es distinto de cero, el resultado de la expresión lógica es verdadero independientemente del valor que almacene la variable x.

Algunos compiladores avisan al programador cuando encuentran expresiones de este tipo.

3.2.3. Un error algo menos común

En esta sección se describe otro error que cometen algunos neófitos en la programación en C++. Se trata de escribir un código como el siguiente:

```
int x;
cin >> x;
if (1 <= x <= 5)
   cout << "x está en el rango [1,5]\n";
```

En este código la expresión lógica 1 <= x <= 5 trata de comprobar si el valor almacenado en la variable x está en el rango $[1, 5]$; sin embargo, esto debe realizarse mediante una expresión como la siguiente: 1 <= x && x <= 5. No obstante, la expresión lógica 1 <= x <= 5 es sintácticamente válida, por lo que un compilador no nos mostrará ningún error al analizarla. La expresión 1 <= x <= 5 se evalúa de la siguiente manera. En primer lugar se evalúa la subexpresión 1 <= x que produce un valor de verdadero o falso en función del valor almacenado en x. A continuación se compara este valor lógico para ver si es menor o igual que 5; esto provoca que se realice una conversión implícita del valor lógico a entero. La conversión implícita de un valor de tipo lógico a uno de tipo entero produce un 0 para un valor falso y un 1 para un valor verdadero. Puesto que tanto 0 como 1 son menores que 5, la expresión 1 <= x <= 5 es siempre verdadera, con independencia del valor almacenado en la variable x. Luego el código anterior siempre mostrará el mensaje `x esta en el rango [1,5]` independientemente del valor leído en x.

3.2.4. El operador condicional

El operador condicional es un operador ternario—es decir, toma tres argumentos—que tiene la siguiente sintaxis:

```
expresión lógica ? expresión1 : expresión2
```

Su modo de funcionamiento es el siguiente. En primer lugar se evalúa la expresión lógica. Si la expresión lógica es verdadera, entonces el resultado es lo que produzca la evaluación de expresion1. En el caso de que la expresión lógica sea falsa, el resultado es lo que produzca la evaluación de expresion2. expresion1 y expresion2 deben ser expresiones del mismo tipo. Como ejemplo, el siguiente fragmento de código asigna a la variable max el máximo de los valores almacenados en las variables x e y.

```
int x = 2, y = 3;
int max = (x > y) ? x : y;
```

En el ejemplo, la condición lógica, x > y, se ha encerrado entre paréntesis porque produce una expresión final más fácil de leer. Sin embargo, los paréntesis no son necesarios.

3.2.5. Cómo mostrar el valor de una expresión lógica

A veces resulta útil mostrar en la salida estándar el resultado de una expresión lógica. Por ejemplo, nos puede interesar mostrar en la salida estándar el resultado de la expresión: x % 2 == 1, que indica si el valor almacenado en la variable entera x es impar. El programa de la Figura 3.6 ilustra cuatro formas de mostrar el resultado de la expresión lógica. A continuación se comenta cada una de las cuatro posibilidades.

1. En la primera forma se utiliza la expresión lógica como condición de una sentencia **if**. En función del resultado de evaluar la expresión se escribe una cadena u otra en la salida estándar indicando la paridad de la variable x.

2. En la segunda forma se utiliza una única instrucción cout. La segunda expresión de la instrucción cout utiliza el operador condicional estudiado en la sección previa para mostrar una expresión de tipo cadena que valdrá "im" o la cadena vacía ("") en función de que la expresión x % 2 == 1 sea verdadera o falsa. Observe que la expresión se ha encerrado entre paréntesis, si no se hubieran utilizado los paréntesis habría problemas con el orden de aplicación de los operadores de la instrucción.

3. En la tercera forma se envía a la salida estándar el resultado de evaluar la expresión x % 2 == 1. Esto produce un 1 en la salida estándar si la expresión es verdadera y un 0 si la expresión es falsa.

4. La cuarta forma es una variante de la anterior, pero se utiliza el manipulador boolalpha antes de la expresión lógica. Esto provoca que en la salida estándar se muestren los

```
#include <iostream>
using namespace std;

int main () {
  int x = 7;
  // forma 1
  if (x %2 == 1)
    cout << "El entero es impar\n";
  else
    cout << "El entero es par\n";
  // forma 2
  cout << "El entero es " << (x %2 == 1 ? "im" : "") << "par\n";
  // forma 3
  cout << "Impar: " << (x %2 == 1) << endl;
  // forma 4
  cout << "Impar: " << boolalpha << (x %2 == 1) << endl;
  return 0;
}
```

Figura 3.6: Varias formas de mostrar en la salida estándar el resultado de una expresión lógica.

valores **true** o **false** en lugar de 1 ó 0 como resultado de evaluar la expresión lógica. A partir de haber utilizado el manipulador boolalpha siempre se mostrarán los valores **true** o **false**. Si se quiere volver a los valores por defecto—1 ó 0—hay que utilizar el manipulador noboolalpha—experimente con su uso en un programa.

3.3. La sentencia switch

La sentencia **switch** permite realizar una selección múltiple. La Figura 3.7 muestra su sintaxis; a continuación se describe su funcionamiento. En primer lugar, se evalúa la expresión, que debe aparecer entre paréntesis. La expresión debe producir un resultado de tipo entero, carácter o lógico. A continuación se compara el resultado de evaluar la expresión con los literales que aparecen en los casos. Si el resultado de evaluar la expresión coincide con el literal de algún caso, entonces se ejecuta el conjunto de instrucciones asociado al caso y el conjunto de instrucciones asociadas al resto de casos que aparecen con posterioridad hasta que se ejecuta la instrucción **break**—que provoca que se salga de la ejecución de la instrucción **switch**—o se han ejecutado todos los casos. Por lo tanto, si para un caso sólo queremos que se ejecute un conjunto de instrucciones debemos terminar este conjunto de instrucciones con la sentencia **break**—la sentencia **break** es opcional. La etiqueta **default** representa un

```
switch (expresión) {
   case literal1:
       conjunto de instrucciones 1
       break;
   case literal2:
       conjunto de instrucciones 2
       break;
   case literaln:
       conjunto de instrucciones n
       break;
   default:
       conjunto de instrucciones por defecto
       break;
}
```

Figura 3.7: Sintaxis de la sentencia **switch**.

caso que se ejecuta si el resultado de evaluar la expresión no coincide con ninguno de los literales de los casos. El especificar una etiqueta **default** es opcional.

La Figura 3.8 contiene un programa que ejemplifica el uso de la instrucción **switch**. Piense qué mensajes se mostrarán en la salida estándar en función del valor introducido por el usuario y experimente ejecutando el programa para comprobar si ha entendido bien el funcionamiento de la instrucción **switch**. Se sugiere que pruebe el programa introduciendo como entrada los valores del 1 al 6.

3.4. Selección múltiple con la sentencia if

Las instrucciones **if** de los programas de las Figuras 3.2 y 3.4 tienen la estructura lógica de una selección múltiple, en el sentido de que tienen asociados n conjuntos de instrucciones de los que sólo se ejecuta—selecciona—un conjunto.

Para implementar una estructura lógica de este tipo el cuerpo de la parte **else** de todas las instrucciones **if** viene constituido por una única instrucción **if**, lo que implica que no es necesario utilizar llaves para delimitar el cuerpo de la parte **else** . Esto posibilita el escribir la selección múltiple utilizando un formato alternativo como el mostrado en el programa de la Figura 3.9. Este formato resulta más legible para muchas personas porque todas las expresiones lógicas y conjuntos de instrucciones aparecen al mismo nivel de indentación, reforzando la idea de que es una selección múltiple en la que sólo se ejecuta un conjunto de instrucciones.

Como ejercicio reescriba los programas de las Figuras 3.2 y 3.5 dándole un formato a la selección múltiple como el utilizado en la Figura 3.9.

```
#include <iostream>

int main () {
  int x;
  std::cout << "Introduzca un valor entero: ";
  std::cin >> x;
  switch (x) {
     case 1:
         std::cout << "Mensaje 1\n";
         break;
     case 2:
         std::cout << "Mensaje 2\n";
     case 3:
         std::cout << "Mensaje 3\n";
         break;
     case 4:
     case 5:
         std::cout << "Mensaje 4\n";
         break;
     default:
         std::cout << "Mensaje por defecto\n";
         break;
  }
  return 0;
}
```

Figura 3.8: Ejemplo de uso de la sentencia **switch**.

```
#include <iostream>
using namespace std;

int main () {
  double nota;
  cout << "Introduzca la nota: ";
  cin >> nota;
  if (nota < 0.0 || nota > 10.0) {
     cout << "Nota no válida" << endl;
  } else if (nota < 5.0) {
     cout << "Suspenso" << endl;
  } else if (nota < 7.0) {
     cout << "Aprobado" << endl;
  } else if (nota < 9.0) {
     cout << "Notable" << endl;
  } else {
     cout << "Sobresaliente" << endl;
  }
  return 0;
}
```

Figura 3.9: Programa equivalente al de la Figura 3.4 pero escrito con otro formato.

3.5. Otro error típico

En esta sección describimos otro error muy común. Se trata de olvidar el uso de las comillas simples cuando se quiere especificar un literal de tipo carácter. Supongamos un programa muy simple en que el usuario introduce el sexo de una persona como un carácter (m o f). Tras la introducción queremos comprobar que el usuario ha introducido un valor de sexo válido, es decir, ha introducido el carácter *m* o el carácter *f*. Muchos programadores empiezan a escribir un programa como el siguiente:

```
#include <iostream>
int main () {
    char sexo;
    std::cout << "Introduce el sexo (m o f): ";
    std::cin >> sexo;
    if (sexo != m && sexo != f)
        std::cout << "El sexo introducido no es correcto\n";
}
```

En este programa se ha olvidado encerrar entre comillas simples los literales de carácter *m* y *f*. El programa falla al compilar, pues interpreta a *m* y *f* como variables y no están

declaradas. Algunos programadores "solucionan" el problema de compilación declarando las variables *m* y *f* de tipo **char**:

```
#include <iostream>
int main () {
    char sexo, m, f;
    std::cout << "Introduce el sexo (m o f): ";
    std::cin >> sexo;
    if (sexo != m && sexo != f)
        std::cout << "El sexo introducido no es correcto\n";
}
```

Esto hace que el programa sea sintácticamente correcto, pero lógicamente incorrecto. Ahora se compara el valor de la variable sexo, que ha sido introducido por el usuario, con los valores de las variables *m* y *f* que no se han iniciado a ningún valor. La solución consiste en utilizar un literal de carácter:

```
#include <iostream>
int main () {
    char sexo;
    std::cout << "Introduce el sexo (m o f): ";
    std::cin >> sexo;
    if (sexo != 'm' && sexo != 'f')
        std::cout << "El sexo introducido no es correcto\n";
}
```

3.6. Ejercicios

1. Evalúe las siguientes expresiones lógicas e indique el orden en que se evalúan las subexpresiones dentro de cada expresión. Los valores de las variables son: a = **true**, b = **true**, c = **false**, d = **false**.

   a) c || ! a && b

   b) ! (a || c) || b && ! c

   c) ! (! (! (a && c || d)))

   d) !(5<3) && a || !(d || c)

   e) a && !(d || !c && a) || !(! c || b)

2. Escriba una expresión lógica que sea verdadera si el valor de la variable entera x:

   a) Pertenece al rango $[5, 10]$.

   b) Vale 5, 7 u 11.

c) No pertenece al rango $[5, 10]$.

d) No vale 5, ni 7 ni 11.

3. Suponga que i y j son variables enteras cuyos valores son 6 y 12 respectivamente. ¿Cuáles de las siguientes expresiones lógicas son verdaderas?

   a) (2∗i) <= j

   b) (2∗i−1) < j

   c) (i > 0) && (i <= 10)

   d) (i > 25) || (i < 50 && j < 50)

   e) (i < 4) || (j > 5)

   f) ! (i > 6)

4. Suponga que a, b, c y d son variables de algún tipo numérico y que b1, b2, b3 y b4 son bloques de instrucciones.

   a) Dado el siguiente fragmento de programa utilice expresiones complejas para expresar la condición necesaria para que se ejecuten los bloques b1, b2, b3 y b4 respectivamente.

```
if (a > b) {
   if (b <= c) {
      if (c != d) {
         b1;
      } else {
         b2;
      }
   } else {
      b3;
   }
} else {
   b4;
}
```

   b) Exprese el siguiente fragmento de código mediante una estructura condicional anidada usando solamente expresiones simples (es decir, sin usar operadores lógicos):

```
if (a < b && c != d) && (b > d || b == d) b1;
if (a < b && c != d) && (b > d || b == e) b2;
```

5. Escriba un programa que lea dos números y determine cuál de ellos es el mayor.

6. Escriba un programa que lea tres números y determine cuál de ellos es el mayor.

7. Escriba un programa que lea cuatro números y determine cuál de ellos es el mayor.

8. Realice un programa que lea un valor entero y determine si se trata de un número par o impar. Sugerencia: un número es par si el resto de dividirlo entre dos es cero. Generalice el programa para que lea dos enteros, n y m, y determine si n divide a m.

9. Escriba un programa que lea de la entrada estándar un carácter e indique en la salida estándar si el carácter es una vocal minúscula o no.

10. Escriba un programa que lea de la entrada estándar un carácter e indique en la salida estándar si el carácter es una vocal minúscula, es una vocal mayúscula o no es una vocal.

11. Escriba un programa que solicite una edad (un entero) e indique en la salida estándar si la edad introducida está en el rango $[18, 25]$.

12. Escriba un programa que lea de la entrada estándar tres números. Después debe leer un cuarto número e indicar si el número coincide con alguno de los introducidos con anterioridad. Utiliza una única instrucción **if** con una expresión lógica compuesta.

13. Realice un programa que lea de la entrada estándar la longitud de los tres lados de un triángulo y muestre en la salida estándar qué tipo de triángulo es, de acuerdo con la siguiente casuística—a denota la longitud del lado más largo y b y c denotan la longitud de los otros dos lados:

    - Si $a \geq b + c$, no se trata de un triángulo
    - Si $a^2 = b^2 + c^2$, es un triángulo rectángulo
    - Si $a^2 < b^2 + c^2$, es un triángulo acutángulo
    - Si $a^2 > b^2 + c^2$, es un triángulo obtusángulo

14. Realice un programa que lea un número real (i), un carácter (c) y un segundo número real (j). En función del carácter leído el programa mostrará en la salida estándar:

    - $i + j$, si c vale '+'
    - $i - j$, si c vale '-'
    - $i * j$, si c vale '*' o 'x'
    - i/j, si c vale '/'
    - el mensaje "operación no contemplada" en otro caso

Soluciones a los ejercicios que no son programas

1. a) c || **false** && b ⇒ c || **false** ⇒ **false**

   b) ! **true** || b && ! c ⇒ **false** || b && ! c ⇒ **false** || b && **true** ⇒ **false** || **true** ⇒ **true**

   c) ! (! (! (**false** || d))) ⇒ ! (! (! **false**)) ⇒ ! (! **true**) ⇒ ! **false** ⇒ **true**

   d) ! **false** && a || !(d || c) ⇒ ! **false** && a || ! **false** ⇒ **true** && a || ! **false** ⇒ **true** && a || **true** ⇒ **true** || **true** ⇒ **true**

   e) a && !(d || **true** && a) || !(! c || b) ⇒ a && !(d || **true**) || !(! c || b) ⇒ a && !**true** || !(! c || b) ⇒ a && !**true** || !(**true** || b) ⇒ a && !**true** || !**true** ⇒ a && **false** || !**true** ⇒ a && **false** || **false** ⇒ **false** || **false** ⇒ **false**

2. a) x >= 5 && x <= 10
   b) x == 5 || x == 7 || x == 11
   c) x < 5 || x > 10 o !(x >= 5 && x <= 10)
   d) x != 5 && x != 7 && x != 11 o !(x == 5 || x == 7 || x == 11)

3. Todas son verdaderas.

4. a)

```
if (a > b && b <= c && c != d)
   b1;
if (a > b && b <= c && c == d)
   b2;
if (a > b && b > c)
   b3;
if (a <= b)
   b4;
```

   b)

```
if (a < b)
  if (c != d)
    if (b >= d)
      b1;

if (a < b) {
  if (c != d) {
    if (b > d) {
      b2;
    } else if (b == e) {
      b2;
    }
  }
}
```

Tema 4
Estructuras repetitivas

Las estructuras repetitivas o iterativas, también llamadas ciclos o bucles, permiten ejecutar un conjunto de instrucciones de manera reiterada mientras que se verifique una condición lógica. Estas estructuras son útiles para aplicar el mismo código a cada dato de un conjunto de datos. En este tema estudiaremos las tres estructuras repetitivas que proporciona C++: las sentencias **while**, **do while** y **for**.

4.1. La sentencia while

La sintaxis de la sentencia **while** es la siguiente:

```
while (expresión lógica) {
   conjunto de instrucciones
}
```

A continuación se describe su funcionamiento. En primer lugar, se evalúa la expresión lógica. Si el resultado de la evaluación es el valor falso, entonces termina la ejecución de la instrucción **while**. Si el resultado de la evaluación es el valor verdadero, entonces se ejecuta el conjunto de instrucciones asociado a la instrucción **while**. Una vez ejecutado el conjunto de instrucciones se vuelve a evaluar la expresión lógica y se procede como antes. Es decir, *el cuerpo de la instrucción **while** se ejecuta mientras que la expresión lógica sea cierta*.

El conjunto de instrucciones asociado a la sentencia **while**, o cuerpo de la sentencia, debe encerrarse entre llaves. No obstante, si este conjunto de instrucciones viene constituido por una única instrucción, entonces las llaves son opcionales.

La Figura 4.1 contiene un programa que ejemplifica el uso de la instrucción **while**. Este programa suma los valores introducidos por el usuario y muestra en la salida estándar el resultado de la suma. Se utiliza el valor 0 para terminar la introducción de datos y la salida del ciclo.

```
#include <iostream>
using namespace std;

int main ()
{
  float valor, suma = 0;
  cout << "Introduzca un valor (0 para terminar): ";
  cin >> valor;
  while (valor != 0) {
      suma += valor;                // equivale a suma = suma + valor;
      cout << "Introduzca un valor (0 para terminar): ";
      cin >> valor;
  }
  cout << "El resultado de la suma es: " << suma << endl;
  return 0;
}
```

Figura 4.1: Ejemplo de uso de la sentencia **while**.

4.2. La sentencia do while

La sentencia **do while** es muy parecida a la sentencia **while**, tiene la siguiente sintaxis:

```
do {
   conjunto de instrucciones
} while (expresión lógica);
```

La diferencia con la sentencia **while** es que en la sentencia **do while** el conjunto de instrucciones se ejecuta por lo menos una vez. Una vez ejecutado el conjunto de instrucciones se evalúa la expresión lógica. Si el resultado de la evaluación es verdadero se vuelve a ejecutar el conjunto de instrucciones, así hasta que la expresión lógica sea falsa, en cuyo momento termina la ejecución de la instrucción **do while**. La Figura 4.2 contiene un programa que ilustra el funcionamiento de la instrucción **do while**. El programa lee de la entrada estándar un dividendo y un divisor y muestra en la salida estándar el resultado de la división. La sentencia **do while** se utiliza para solicitar la reintroducción del divisor cuando éste vale cero. El control de los valores de entrada constituye uno de los principales usos de la sentencia **do while**.

El conjunto de instrucciones asociado a la sentencia **do while** debe encerrarse entre llaves. No obstante, si este conjunto viene constituido por una única instrucción, entonces las llaves son opcionales.

```
#include <iostream>
using namespace std;

int main ()
{
  double dividendo, divisor;
  cout << "Introduzca el dividendo: ";
  cin >> dividendo;
  do {
     cout << "Introduzca el divisor (distinto de cero): ";
     cin >> divisor;
  } while (divisor == 0);
  cout << "El resultado es: " << dividendo/divisor << endl;
  return 0;
}
```

Figura 4.2: Ejemplo de uso de la sentencia **do while**.

4.3. La sentencia for

La sentencia **for** está pensada principalmente para expresar ciclos que se ejecutan un número fijo de veces. En ese caso la sentencia **for** resulta más legible que las sentencias **while** y **do while**, pues permite expresar la iniciación, incremento y condición lógica asociada a la variable que controla el ciclo en una única línea al principio del ciclo. La sintaxis de la instrucción **for** es la siguiente:

```
for (expr1; expresión lógica; expr2) {
   conjunto de instrucciones
}
```

Entre paréntesis aparecen tres expresiones separadas por el carácter punto y coma. La segunda expresión debe ser de tipo lógico. El funcionamiento de la instrucción es el siguiente:

- En primer lugar se ejecuta la expresión $expr_1$. Esta expresión no vuelve a ejecutarse más, por lo que suele contener algún tipo de iniciación del ciclo.
- A continuación se evalúa la expresión lógica. Mientras que el resultado de evaluar la expresión lógica sea verdadero se ejecuta el conjunto de instrucciones y, a continuación, la expresión $expr_2$. Cuando la expresión lógica es falsa se termina de ejecutar la instrucción **for**.

Una sentencia **for** como la indicada unas líneas más arriba equivale a la siguiente sentencia **while**:

```
#include <iostream>
using namespace std;
int main ()
{
  int numero;
  for (numero = 1; numero <= 10; numero++) {
     cout << numero << " al cuadrado vale " << numero*numero << endl;
  }
}
```

Figura 4.3: Ejemplo de uso de la sentencia **for**.

```
#include <iostream>

int main ()
{
  int numero, i, fact = 1;
  std::cout << "Introduzca un entero positivo: ";
  std::cin >> numero;
  for (i = 2; i <= numero; i++) {
     fact *= i; // equivale a fact = fact * i;
  }
  std::cout << "El factorial de " << numero << " es " << fact << std::endl;
  return 0;
}
```

Figura 4.4: Cálculo del factorial con un ciclo **for**.

```
expr1;
while (expresión lógica) {
   conjunto de instrucciones
   expr2;
}
```

salvo para lo indicado para la sentencia **continue**, que se describirá en la siguiente sección. La Figura 4.3 ilustra el funcionamiento de la sentencia **for** mediante un programa que muestra en la salida estándar el cuadrado de los enteros del 1 al 10. La variable numero se utiliza para generar los enteros del 1 al 10 y controlar el número de ejecuciones del cuerpo del ciclo. El programa de la Figura 4.4 calcula el factorial de un número introducido desde la entrada estándar. Observe cómo se utiliza la variable i para controlar el número de veces que se ejecuta el ciclo y para obtener los factores.

Como en el resto de sentencias que llevan asociado un conjunto o bloque de instrucciones, en la sentencia **for** el conjunto de instrucciones debe encerrarse entre llaves, a no ser

que este conjunto venga determinado por una única instrucción, en cuyo caso las llaves son opcionales. La sintaxis de la sentencia **for** permite omitir cualquiera de las tres expresiones que lleva asociada. En el caso de que se omita la expresión lógica ésta se considera como verdadera. Por ejemplo, lo siguiente es un ciclo infinito:

```
for (;;)
  cout << "Mensaje infinito\n";
```

Se puede especificar más de una expresión en cada una de las tres expresiones asociadas a la sentencia **for** siempre que se separen mediante comas. Por ejemplo:

```
for (int i = 1, int j = 10; i <= 10; i++, j--)
  cout << i << ' ' << j << endl;
```

Aquí la primera expresión, llamada normalmente de iniciación, define e inicia dos variables: `i` y `j`. La última expresión, llamada normalmente de incremento, contiene dos expresiones, una de ellas incrementa el valor de la variable `i` y la otra decrementa el valor de `j`.

4.4. Las sentencias break y continue

La sentencia **break** ya se utilizó en el Tema 3, Sección 3.3, en la instrucción **switch**. Cuando aparece en un ciclo—ya sea **while**, **do while** o **for**—la ejecución de la sentencia **break** provoca el término de la ejecución del ciclo más interno en que se encuentra ubicada. El programa de la Figura 4.5 ejemplifica el uso de la sentencia **break**. Este programa lee números enteros de la entrada estándar y muestra en la salida estándar un mensaje indicando si son primos o no. El ciclo más externo, el **while**, se utiliza para la lectura de los números y el ciclo más interno, el **for**, para calcular si el número introducido es primo. Justo antes de comenzar el ciclo **for** se asigna a la variable lógica `primo` el valor verdadero y después el ciclo comprueba si algún entero menor que el número introducido es un divisor del número. En cuanto se encuentra un divisor se sabe que el número no es primo. Por lo tanto, se actualiza la variable `primo` al valor falso y se utiliza la sentencia **break** para terminar la ejecución del ciclo **for**.

El uso de una sentencia **break** nunca resulta estrictamente necesario, la Figura 4.6 muestra un programa alternativo al de la Figura 4.5 que no utiliza la sentencia **break**. En este programa se utiliza la variable lógica `primo` para salir del ciclo **for** cuando se descubre que el número no es primo.

Ejercicio 1 Escriba una versión del programa de la Figura 4.5 en la que no se utilice la variable lógica `primo`. Sugerencia: puede utilizar el valor de la variable `i` tras el ciclo **for** para comprobar si el número es primo.

Ejercicio 2 Los programas escritos son ineficientes. Para determinar que un número n es primo comprueban que no es divisible por ningún número en el rango $[2, n-1]$. Realmente no hace falta comprobar tantos números, basta con comprobar los números en

```
#include <iostream>
using namespace std;

int main ()
{
  int numero, i;
  bool primo;
  cout << "Introduzca un entero positivo (0 para terminar): ";
  cin >> numero;
  while (numero != 0) {
     primo = true;
     for (i = 2; i < numero; i++) {
        if (numero % i == 0) { //si el resto de la división entera es cero
            primo = false;
            break;
        }
     }
     if (primo)
        cout << "El número " << numero << " es primo" << endl;
     else
        cout << "El número " << numero << " no es primo" << endl;
     cout << "Introduzca un entero positivo (0 para terminar): ";
     cin >> numero;
  }
  return 0;
}
```

Figura 4.5: Ejemplo de uso de la sentencia **break**.

el rango $[2, i]$, donde i es el mayor entero tal que $i * i <= n$. Incorpore esta mejora para escribir un programa más eficiente.

La otra sentencia que se va a estudiar en esta sección es **continue**. Esta sentencia debe escribirse dentro del cuerpo de un ciclo, siendo su uso menos frecuente que el de la sentencia **break**. El efecto de la ejecución de la sentencia **continue** en el cuerpo de un ciclo **while** o **do while** es que se deja de ejecutar el resto de instrucciones del cuerpo del ciclo y se pasa a evaluar la condición lógica que controla el ciclo. El efecto en una sentencia **for** es que se deja de ejecutar el resto de instrucciones del cuerpo del ciclo y se pasa a evaluar la expresión de incremento del ciclo para, a continuación, evaluar la condición lógica asociada al ciclo. El programa de la Figura 4.7 ilustra el funcionamiento de la sentencia **continue**. Este programa muestra en la salida estándar los números del 0 al 10, excepto el 5. El uso de la sentencia **continue** nunca resulta estrictamente necesario. El ciclo del programa de la Figura 4.7 podría escribirse sin utilizar la sentencia **continue** de la siguiente forma:

```
#include <iostream>
using namespace std;

int main ()
{
  int numero, i;
  bool primo;
  cout << "Introduzca un entero positivo (0 para terminar): ";
  cin >> numero;
  while (numero != 0) {
     primo = true;
     for (i = 2; i < numero && primo; i++) {
        if (numero % i == 0) //si el resto de la división entera es cero
            primo = false;
     }
     if (primo)
        cout << "El número " << numero << " es primo" << endl;
     else
        cout << "El número " << numero << " no es primo" << endl;
     cout << "Introduzca un entero positivo (0 para terminar): ";
     cin >> numero;
  }
  return 0;
}
```

Figura 4.6: Programa alternativo al de la Figura 4.5 que no usa **break**.

```
#include <iostream>
using namespace std;

int main ()
{
  for (int i = 0; i < 11; i++) {
     if (i == 5)
        continue;
     cout << i << ' ';
  }
  cout << endl;
  return 0;
}
```

Figura 4.7: Ejemplo de uso de la sentencia **continue**.

```
for (int i = 0; i < 11; i++) {
   if (i != 5)
      cout << i << ' ';
}
```

4.5. Lugar de definición y ámbito de las variables

C++ proporciona unas reglas muy flexibles para elegir el lugar en el que se define una variable. En función del lugar de su definición, una variable tiene un *ámbito*, que es la zona del programa en la que puede ser usada. A continuación se enumeran las principales reglas sobre el lugar de definición y ámbito de las variables:

- Una variable puede definirse fuera del cuerpo de las funciones del programa. Se estudiará esta circunstancia en el Tema 6, Sección 6.4, en el que se estudian las funciones. En lo que resta de sección se supone que una variable está definida dentro del cuerpo de una función, es decir, entre las llaves que delimitan el cuerpo de la función.
- Una variable puede definirse en cualquier lugar en el que se puede escribir una instrucción. El ámbito de la variable abarca desde su lugar de definición hasta el final del bloque más interno en que se ha definido la variable—un *bloque* es un conjunto de instrucciones encerradas entre llaves.
- Se puede definir una variable en la parte de iniciación de una sentencia **for**. En ese caso el ámbito de la variable es el cuerpo y cabecera de la instrucción **for**.
- Se puede definir más de una vez una misma variable—o sea, con el mismo identificador—dentro del cuerpo de una función siempre que las definiciones ocurran en bloques no

solapados o en bloques solapados de distinto nivel de profundidad. Si los ámbitos de las distintas definiciones no se solapan, entonces no existe ambigüedad en su uso. Si los ámbitos se solapan entonces la que está definida en el bloque más interno es la que tiene validez y se dice que *oculta* al resto.

La Figura 4.8 contiene un programa que ilustra la definición y ámbito de variables. A continuación se analiza las distintas definiciones de variables en el programa:

- La primera definición de variable se encuentra en el cuerpo de la primera instrucción **if**. El ámbito de esta variable abarca desde su lugar de definición hasta el final del bloque en que se haya definida, es decir, hasta el final del cuerpo de la instrucción **if**.
- A continuación viene la instrucción **for**, que define una nueva variable x en su expresión de iniciación. Observe que el ámbito de esta nueva variable no se solapa con el ámbito de la variable x definida con anterioridad. El ámbito de la variable definida en la instrucción **for** abarca el cuerpo y cabecera del ciclo **for**—el cuerpo está formado por una única instrucción.
- Después del ciclo **for** vienen dos instrucciones de salida que provocan un error, pues utilizan las variables x y z y no hay variables de ese nombre en ámbito en esa zona. Deberá eliminar o comentar estas dos instrucciones si quiere que el programa compile.
- A continuación se define la variable z, cuyo ámbito abarca hasta el final del cuerpo de la función main. Sin embargo, el ámbito de z se ve solapado en el cuerpo de la instrucción **if** con una nueva definición de z. La segunda definición de z oculta a la primera en el cuerpo del **if**.
- La última definición de la variable z produce un error, pues está definida en el mismo bloque y al mismo nivel de profundidad que la definición: **int** z = 3;.

Aunque C++ permite definir varias veces una variable con el mismo nombre dentro de un bloque, nuestro consejo es que lo evite, pues con esta práctica se disminuye la legibilidad del programa. Evite especialmente el definir variables que se solapan, pues dificulta en gran medida la lectura y compresión de un programa.

En cuanto al lugar donde se define una variable hay quien prefiere definir todas las variables al principio del cuerpo de una función, mientras que otros prefieren definir cada variable justo antes de empezar a utilizarla. Hágalo como se sienta más cómodo.

4.6. Ejercicios

En esta sección se proponen varios ejercicios para practicar con las instrucciones iterativas de C++. Al final vienen soluciones a los tres primeros ejercicios, aunque el lector puede escribir

```
#include <iostream>
using namespace std;

int main ()
{                          // bloque de profundidad 0 (cuerpo de main)
   if (true) {             // bloque de profundidad 1 (cuerpo del if)
      int x = 0;
      cout << x << endl;
   }                       // se vuelve al bloque de profundidad 0
   for (int x = 1; x < 11; x++)
      cout << x << endl;
   cout << z << endl;      // error, z no está definida
   cout << x << endl;      // error, x no está definida
   int z = 3;
   if (true) {             // bloque de profundidad 1 (cuerpo del if)
      int z = 4;           // oculta a la z previa
      cout << z << endl; // muestra 4
   }                       // se vuelve al bloque de profundidad 0
   cout << z << endl;      // muestra 3
   int z = 5;         // error, mismo bloque y profundidad que int z = 3
}
```

Figura 4.8: Ejemplo de lugar de definición y ámbito de variables.

los programas en su entorno de programación en C++ para comprobar si sus respuestas eran correctas.

1. Indique para cada uno de los siguientes ciclos si el ciclo termina o se ejecuta de manera infinita. En este último caso señale las razones por las que no termina.

a)
```
int contador = 0;
int total = 0;
while (contador >= 0) {
   total = total + 2;
}
```

b)
```
int contador = 0;
int total = 0;
while (contador <= 0) {
   total = total + 2;
   contador = contador + 2;
}
```

c)
```
int contador = 0;
int total = 0;
while (contador <= 10) {
   total = total + 2;
   contador = contador + 1;
}
```

2. Indique el valor que se muestra en salida estándar al ejecutarse los siguientes fragmentos de código:

a)
```
int var = 0;
for (int num = 1; num <= 10; num++)
   var = var + 1;
cout << var << endl;
```

b)
```
int var = 0;
for (int num = 4; num <= 16; num++)
   var++;
cout << var << endl;
```

c)
```
int var = 0;
for (int num = 1; num <= 5; num++)
   for (int mult = 1; mult <= 8; mult++)
      ++var;
cout << var << endl;
```

3. Indique el valor o valores que envían a la salida estándar cada uno de los siguientes fragmentos de código:

a)
```
for (int i = 1; i <= 10; i++)
   cout << i << endl;
```

b)
```
for (int i = 1; i <= 10; i++)
   cout << i+1 << endl;
```

c)
```
for (int i = 1; i <= 10; i++)
   cout << 2*i << endl;
```

d)
```
cin >> n;
for (int i = 1; i <= 10; i++)
   cout << n*i << endl;
```

e)
```
int s = 0;
for (int i = 1; i <= 10; i++)
   s = i;
cout << s << endl;
```

f)
```
int s = 0;
for (int i = 1; i <= 10; i++)
   s = s + 1;
cout << s << endl;
```

4. Realice un programa que solicite de la entrada estándar un entero del 1 al 10 y muestre en la salida estándar su tabla de multiplicar.

5. Realice un programa que lea de la entrada estándar números hasta que se introduzca un cero. En ese momento el programa debe terminar y mostrar en la salida estándar el número de valores mayores que cero leídos.

6. Realice un programa que calcule y muestre en la salida estándar la suma de los cuadrados de los números enteros del 1 al 10—la solución es 385.

7. Escriba un programa que tome cada 4 horas la temperatura exterior, leyéndola durante un período de 24 horas. Es decir, debe leer 6 temperaturas. Calcule la temperatura media del día, la temperatura más alta y la más baja.

8. Escriba un programa que lea valores enteros hasta que se introduzca un valor en el rango $[20, 30]$ o se introduzca el valor 0.

9. Escriba un programa que calcule x^y, donde tanto x como y son enteros positivos, sin utilizar la función pow.

10. Para cada uno de los siguientes apartados escriba un programa que calcule el valor de la suma o el producto expresado:

    a) $\sum_{i=1}^{n} i = 1 + 2 + 3 + ... + n$
    b) $\sum_{i=1}^{n} 2i - 1 = 1 + 3 + 5 + ... + 2n - 1$
    c) $\prod_{i=1}^{n} i = 1 * 2 * 3 * ... * n$
    d) $\sum_{i=1}^{n} i! = 1! + 2! + 3! + ... + n!$
    e) $\sum_{i=1}^{n} 2^i = 2^1 + 2^2 + 2^3 + ... + 2^n$

11. Existen muchos métodos numéricos capaces de proporcionar aproximaciones al número Π. Uno de ellos es el siguiente:

$$\Pi = \sqrt{\sum_{i=1}^{\infty} \frac{6}{i^2}}$$

    Cree un programa que lea el grado de aproximación—número de términos de la sumatoria—y devuelva un valor aproximado de Π.

12. Los reglamentos de pesca imponen un límite a la cantidad total de kilos permitida en un día de pesca. Diseñe un programa que, en primer lugar, lea el límite diario—en kilos. A continuación el programa debe leer los pesos de las presas según el orden en que se pescaron—un valor de cero indica el final de las presas. El programa debe mostrar tras cada presa introducida el peso total acumulado. Si en un momento dado se excede la cantidad total de kilos permitida, entonces el programa debe mostrar un mensaje indicativo y terminar.

13. En una clase de 5 alumnos se han realizado tres exámenes y se requiere determinar el número de:

    a) Alumnos que aprobaron todos los exámenes.

b) Alumnos que aprobaron al menos un examen.

c) Alumnos que aprobaron únicamente el último examen.

Realice un programa que permita la lectura de los datos y el cálculo de las estadísticas.

14. Una compañía de seguros está preparando un estudio concerniente a todos los accidentes ocurridos en la provincia de Jaén en el último año. Para cada conductor involucrado en un accidente se tienen los siguientes datos: edad del conductor, sexo—'M' o 'F'—y código de registro—1 para los registrados en la ciudad de Jaén, 0 para los registrados fuera de la ciudad de Jaén. Realice un programa que lea los datos de conductores hasta que se introduzca una edad de 0 y muestre las siguientes estadísticas:

    a) Porcentaje de conductores menores de 25 años

    b) Porcentaje de conductores de sexo femenino.

    c) De los conductores de sexo masculino porcentaje de conductores con edades comprendidas entre 18 y 25 años.

15. Modifique el programa del ejercicio anterior de manera que sólo se acepten edades en el rango [18,100] o la edad 0, sólo se acepte como sexo las letras M y F y sólo se acepte como código 0 ó 1.

16. Realice un programa que solicite al usuario que piense un número entero entre el 1 y el 1000. El programa tiene la oportunidad de preguntarle al usuario si el número pensado es uno dado, por ejemplo, x. El usuario debe responder si lo es, y en caso de no serlo si x es menor o mayor al número que pensó. El programa debe realizar un máximo de 10 preguntas para descubrir el número.

17. Realice un programa que calcule la descomposición en factores primos de un número entero.

18. Realice un programa que solicite al usuario un entero positivo e indique en la salida estándar si el entero leído es una potencia de dos.

19. Realice un programa que lea de la entrada estándar 10 números e indique en la salida estándar si el número cero estaba entre los números leídos.

Soluciones a los ejercicios que no son programas

1. El primer ciclo se ejecuta de manera infinita, pues la condición del ciclo, contador $>= 0$, se verifica al comenzar el ciclo y el cuerpo del ciclo no modifica el valor de la variable contador.

   El cuerpo del segundo ciclo se ejecuta una vez y el cuerpo del tercer ciclo once veces.

2. a) 10, b) 13, c) 40

3. a) los valores del 1 al 10, b) los valores del 2 al 11, c) la tabla de multiplicar del 2, d) la tabla de multiplicar del número leído de la entrada estándar, e) 10, f) 10.

Tema 5
Tipos de datos estructurados

En el Tema 1 se estudiaron los tipos de datos primitivos de C++, es decir, los tipos de datos numéricos, lógico y carácter. En este tema se analizan los tipos de datos estructurados. Estos se utilizan para almacenar colecciones de datos e incluyen a los vectores, las estructuras y las cadenas de caracteres. Los vectores permiten almacenar una serie de datos del mismo tipo, mientras que las estructuras permiten almacenar una colección de datos de distinto tipo. Dada la importancia de la información de tipo texto, C++ posee un tipo de dato específico, las cadenas de caracteres, que permiten almacenar y manipular de una forma sencilla una secuencia de caracteres.

5.1. Los vectores

Un vector o *array* unidimensional es un tipo de dato que almacena una secuencia de datos del mismo tipo. Los elementos de un vector se almacenan en zonas contiguas de memoria y se puede acceder a ellos de manera directa mediante un índice o posición. El programa de la Figura 5.1 ilustra la definición y utilización de un vector. En la primera línea de la función main se define un vector de 6 enteros de nombre v. Para definir un vector se utiliza la siguiente sintaxis:

```
tipo nombre[tamaño];
```

donde tipo es el tipo de los elementos del vector, nombre es el nombre de la variable que sirve para referenciar al vector y tamaño indica el número de elementos del vector. El tamaño debe expresarse mediante una expresión constante de tipo entero. En C++ los índices de los vectores empiezan en cero. Por lo tanto, los índices de un vector de tamaño tam pertenecen al rango $[0, tam - 1]$. Por ejemplo, el rango de los índices del vector definido en el programa de la Figura 5.1 es $[0, 5]$.

Para acceder a los elementos de un vector se utiliza la sintaxis: v[i], donde v es el nombre de la variable de tipo vector e i es una expresión entera que indica el índice del elemento del vector al que se quiere acceder. La expresión v[i] se puede utilizar en cualquier lugar

```
#include <iostream>
using namespace std;

int main () {
  int v[6];
  cout << "Introduzca cinco enteros separados por espacios: ";
  for (int i = 0; i < 5; i++)
     cin >> v[i];
  v[5] = 10;
  cout << "Los enteros del vector son : " << endl;
  for (int i = 0; i < 6; i++)
     cout << v[i] << endl;
  return 0;
}
```

Figura 5.1: Ejemplo de definición y uso de un vector.

en el que se pueda utilizar una variable del tipo de los elementos del vector. Por ejemplo, en el programa de la Figura 5.1 los 5 primeros elementos del vector se inician con valores introducidos desde la entrada estándar, se asigna el valor 10 al elemento del vector de índice cinco y, por último, se accede a los elementos del vector para mostrar su contenido.

La utilidad de los vectores es que permiten crear múltiples datos de un mismo tipo con una única sentencia de definición y una única variable. Además, el acceso a los datos almacenados en el vector es muy eficiente y sencillo. Para acceder a un dato de un vector se requiere el nombre de la variable de tipo vector y el índice del dato al que se quiere acceder. Por ejemplo, suponga que necesita almacenar la nota final de los 60 alumnos de una clase. El uso de 60 variables reales para almacenar las notas daría como resultado un programa sumamente largo y engorroso. Alternativamente, se puede utilizar un vector de tamaño 60 para almacenar las notas. Esta alternativa es mucho más elegante, pues sólo precisa el declarar una variable. Además, el código de acceso a las notas en conjunto—por ejemplo, para calcular su valor medio—será mucho más conciso, al estar apoyado en ciclos, como se describe a continuación.

Como sucede en el programa de la Figura 5.1, el código de trabajo con vectores suele ir asociado al uso de ciclos **for**. Mediante la variable de control del ciclo se generan los índices del vector y en el cuerpo del ciclo se indexa el vector usando como índice la variable de control del ciclo. De esta forma, en cada iteración se accede a un elemento distinto del vector y el ciclo, en su conjunto, procesa todos los elementos del vector.

El programa de la Figura 5.2 representa una alternativa al programa de la Figura 5.1 sin usar vectores. Observe que en este caso hay que definir tantas variables como datos se precisa leer y no es posible usar ciclos para la lectura e impresión de los datos. Suponga que en lugar de seis datos se necesitan mil, los cambios a realizar en el programa de la Figura

```
#include <iostream>
using namespace std;

int main () {
  int v1, v2, v3, v4, v5, v6;
  cout << "Introduzca cinco enteros separados por espacios: ";
  cin >> v1 >> v2 >> v3 >> v4 >> v5;
  v6 = 10;
  cout << "Los enteros introducidos son : " << endl;
  cout << v1 << endl << v2 << endl << v3 << endl
       << v4 << endl << v5 << endl << v6 << endl;
  return 0;
}
```

Figura 5.2: Programa alternativo al de la Figura 5.1 que no usa vectores.

5.1 serían mínimos. Sin embargo, en el programa de la Figura 5.2 ¡habría que definir mil variables e incluir la mil variables en las instrucciones cin y cout!

5.1.1. Iniciación de un vector en su definición

Se puede iniciar un vector en la misma definición de la variable de tipo vector, como ilustra el programa de la Figura 5.3. Para realizar esto hay que encerrar entre llaves la lista de valores con que se inicia el vector, separando los valores entre sí mediante comas. Observe que tras la llave que cierra la iniciación hay que poner un punto y coma. Se puede iniciar menos elementos de los que tiene el vector, pero es un error escribir una lista de iniciación con más valores que el tamaño del vector. Si a la vez que se define un vector se inicia, no es necesario especificar el tamaño del vector. Por ejemplo, en lugar de la definición utilizada en el programa de la Figura 5.3 se podría haber utilizado la siguiente definición:

```
int v[] = { 2, 3, 5, 4, 1 };
```

En este caso el compilador cuenta el tamaño de la lista de iniciación y asigna ese tamaño al vector. Por lo tanto, en el ejemplo anterior el tamaño del vector será 5.

5.1.2. Implementación del tipo de dato vector

Resulta práctico e interesante conocer cómo se implementa el tipo de dato vector. Cuando se define un vector de un tamaño determinado, se reserva una zona de memoria contigua del tamaño suficiente para albergar los elementos del vector. Por ejemplo, la Figura 5.4 ilustra la definición e implementación de un vector de enteros de tamaño 5. Se supone que cada entero ocupa 4 bytes, por lo que se reservan 20 bytes contiguos de memoria. Si el vector

```
#include <iostream>

int main () {
  int v[5] = { 2, 3, 5, 4, 1 };
  for (int i = 0; i < 5; i++)
     std::cout << v[i] << std::endl;
  return 0;
}
```

Figura 5.3: Ejemplo de iniciación de un vector en su definición.

almacenara 5 datos de tipo **double**, y cada **double** ocupara 8 bytes, entonces el espacio reservado sería de 40 bytes. La variable de tipo vector—v en la Figura 5.4—almacena la dirección de inicio en memoria del vector. Para hacer referencia a un elemento del vector hay que especificar el vector y su índice, como en v[i]. Con esta información, y teniendo en cuenta que la memoria se direcciona a nivel de byte, el compilador realiza un sencillo cálculo aritmético para obtener la dirección de inicio en memoria de un elemento del vector y, por lo tanto, tener acceso a dicho elemento. Concretamente, la dirección de inicio en memoria del elemento de índice i del vector v—o sea, la dirección de v[i]—se obtiene realizando el siguiente cálculo: $dir\_ini + i * tam\_bytes$, donde dir_ini es la dirección de memoria donde empieza el almacenamiento del vector v y tam_bytes es el tamaño en bytes de los elementos de v. Este cálculo es correcto porque los elementos del vector se almacenan en posiciones contiguas de memoria y porque todos los elementos del vector son del mismo tipo. Como ejemplo, para el vector de la Figura 5.4 la dirección de memoria del elemento v[2] se calcula como: $1000 + 2 * 4 = 1008$.

Como se indicó anteriormente, cuando se define un vector de un determinado tamaño, *tam*, se reserva almacenamiento para *tam* elementos y se accede a los elementos del vector a través del nombre de un vector y un índice, que debe pertenecer al rango $[0, tam - 1]$. Un error muy común consiste en intentar acceder a un elemento de un vector especificando un índice que no está en el rango válido. Un ejemplo sería intentar acceder al elemento v[5] en el vector de la Figura 5.4. C++ no comprueba si el índice especificado al acceder a un elemento de un vector está en el rango correcto, simplemente calcula la dirección asociada a ese índice e intenta acceder. Un acceso a un vector fuera del rango válido es un error insidioso. Por ejemplo, para el vector definido en la Figura 5.4 una instrucción del tipo v[5] = 3; provocaría que se intentara escribir el valor 3 en complemento a dos ocupando 4 bytes a partir de la dirección de memoria 1020. Dependiendo de la disposición de la memoria esto podría ocurrir o no. Si a partir de la dirección 1020 se guardan variables que se pueden modificar, la asignación tendría éxito, pero, sin darnos cuenta, estaríamos sobrescribiendo el valor de otra variable. Si la configuración de memoria es tal que no se puede escribir en la dirección

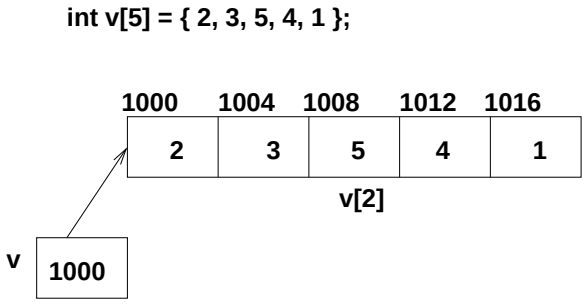

Figura 5.4: Implementación del tipo de dato vector.

```
#include <iostream>
using namespace std;

int main () {
  char v[5] = { 'a', 'e', 'i', 'o', 'u' };
  int i;
  cout << "Introduzca un índice (-1 para terminar): ";
  cin >> i;
  while (i != -1) {
    cout << "Elemento v[" << i << "] = " << v[i] << endl;
    cout << "Introduzca un índice (-1 para terminar): ";
    cin >> i;
  }
  return 0;
}
```

Figura 5.5: Ejemplo de programa que puede provocar un acceso incorrecto en un vector.

1020, entonces nuestro programa finalizaría abruptamente y aparecería algún mensaje en la pantalla del ordenador.

El programa de la Figura 5.5 permite consultar los elementos de un vector, pero no comprueba si el índice proporcionado por el usuario está en el rango correcto. Ejecútelo e introduzca índices correctos e incorrectos hasta que obtenga un error en tiempo de ejecución.

5.1.3. El tamaño de un vector

El tamaño de un vector, tal como se indicó en la Sección 5.1, debe especificarse mediante una expresión constante de tipo entero. Por lo tanto, en el momento de escribir un programa hay que especificar el tamaño de sus vectores, aunque el tamaño exacto no se sepa en algunas ocasiones hasta el momento en que se ejecuta el programa—usando terminología de pro-

gramación se dice que el tamaño de un vector hay que especificarlo en *tiempo de compilación* y que, a veces, su tamaño exacto no se conoce hasta *tiempo de ejecución*. La estrategia que se emplea en estos casos es especificar un tamaño de vector lo suficientemente grande como para albergar el número de datos precisado por cualquier ejecución del programa. Por ejemplo, el programa de la Figura 5.6 lee una serie de valores de la entrada estándar y envía a la salida estándar los valores que son mayores que la media de los valores introducidos. En este programa hay que consultar dos veces el conjunto de valores introducidos: una primera vez para calcular su media y una segunda vez para determinar qué valores son mayores que la media. En lugar de solicitar que se introduzcan dos veces los valores—lo cual sería tedioso para el usuario y propenso a errores—, éstos se almacenan en un vector, lo que permite su consulta reiterada. Pasemos ahora a analizar el tamaño del vector en el que se almacenan los datos. Para que el programa sea más útil se deja a voluntad del usuario elegir el número de datos que desea introducir. Sin embargo, hay que especificar un tamaño para el vector en tiempo de compilación. Se decide especificar un tamaño—1000—lo suficientemente grande como para que quepan los valores introducidos en cualquier ejecución del programa. En cada ejecución se solicita al usuario con cuántos datos desea trabajar—en el programa esta información se almacena en la variable tam—y se trabaja únicamente con las primeras tam posiciones del vector. Observe que el programa, en el ciclo **do while**, controla que el usuario no introduzca más valores que los que puede albergar el vector. Dada una ejecución de este programa el vector sólo trabaja con las primeras tam posiciones y, por lo tanto, no se utilizan las $1000 - tam$ posiciones restantes. La variable TAM almacena el *tamaño físico* del vector, mientras que la variable tam almacena su *tamaño lógico*.

Un error muy común que comenten los programadores principiantes en C++ es escribir un código como el siguiente:

```
int tam;
cout << "Número de datos [1," << TAM << "]: ";
cin >> tam;
float v[tam];
```

Con este código se especifica el tamaño del vector una vez que se conoce su número de elementos en tiempo de ejecución. El problema es que este código no es correcto, puesto que el tamaño de un vector debe especificarse mediante una constante entera, y la expresión tam es una expresión entera, pero no constante. Desgraciadamente, muchos compiladores no detectan este error. Sin embargo, el código es erróneo y no se tiene ninguna certeza sobre el tamaño reservado para ubicar al vector.

C++ posee mecanismos para reservar memoria en tiempo de ejecución, dichos mecanismos implican el uso de punteros—véase el Tema 7—y memoria dinámica—véase el Tema 8, Sección 8.3. También se pude utilizar la clase *vector* de la biblioteca estándar de C++, que utiliza punteros y memoria dinámica en su implementación, para trabajar con vectores cuyo tamaño se puede especificar—incluso pueden crecer y decrecer—en tiempo de ejecución. En estos apuntes no se estudiará la citada clase *vector* de la biblioteca estándar de C++.

```
#include <iostream>
using namespace std;

int main () {
   const int TAM = 1000;
   float v[TAM];
   int tam;
   float media = 0.0f;
   do {
      cout << "Número de datos [1," << TAM << "]: ";
      cin >> tam;
   } while (tam < 1 || tam > TAM);
   for (int i = 0; i < tam; i++) {
      cout << "Introduzca dato " << i << ": ";
      cin >> v[i];
      media += v[i]; // equivale a media = media + v[i];
   }
   media /= tam;      // equivale a media = media / tam;
   cout << "Valores mayores que la media (" << media << "):\n";
   for (int i = 0; i < tam; i++)
      if (v[i] > media)
         cout << v[i] << '\n';
   return 0;
}
```

Figura 5.6: El tamaño de un vector.

```
#include <iostream>
using namespace std;

int main () {
   const int TAM = 3;
   float v1[TAM] = {1, 2, 3};
   float v2[TAM];
   for (int i = 0; i < TAM; i++) // copia los elementos
          v2[i] = v1[i];
   for (int i = 0; i < TAM; i++) // muestra los elementos de v2
          cout << v2[i] << '\n';
   bool iguales = true;
   for (int i = 0; i < TAM; i++) // compara los elementos de v1 y v2
          if (v1[i] != v2[i])
              iguales = false;
   if (iguales)
      cout << "Vectores iguales\n";
   else
      cout << "Vectores distintos\n";
   return 0;
}
```

Figura 5.7: Operaciones con vectores.

5.1.4. Operaciones con vectores

Los vectores permiten definir de una forma sencilla varios datos del mismo tipo. Se puede utilizar la indexación para acceder a los elementos del vector y realizar con estos elementos las mismas operaciones aplicables a variables que tengan el mismo tipo que los elementos del vector. Sin embargo, a nivel de tipo vector no es posible realizar ninguna operación, salvo la indexación. Por lo tanto, no se puede asignar un vector a otro, ni enviarlo a la salida estándar con cout, ni comparar, sumar o restar vectores utilizando operadores de C++. Para realizar estas operaciones habrá que escribir el código pertinente. Por ejemplo, el programa de la Figura 5.7 copia el contenido de un vector en otro, muestra en la pantalla el contenido del vector copiado y compara el vector original y su copia para ver si contienen los mismos elementos. Todo este procesamiento se realiza trabajando con los vectores a nivel de elemento, pues los operadores de asignación y comparación no asignan ni comparan el contenido de vectores.

5.2. Los *arrays* multidimensionales

C++ permite trabajar con *arrays* multidimensionales, es decir, de varias dimensiones. Los *arrays* de dos dimensiones o bidimensionales permiten representar matrices. Para definir un *array* multidimensional de dimensión *n* se utiliza la siguiente sintaxis:

```
tipo nombre[tam1][tam2]...[tamn];
```

donde tipo es el tipo de los elementos del *array*, nombre es el nombre de la variable asociada al *array* y tam1, tam2, ..., tamn son expresiones constantes de tipo entero que indican los tamaños de las distintas dimensiones del *array*. Para acceder a un elemento del *array* se utiliza la sintaxis: v[ind1][ind2]...[indn], donde ind1, ind2, ..., indn son los índices, debiendo ser expresiones de tipo entero en el rango $[0, tam1 - 1]$, $[0, tam2 - 1]$, ..., $[0, tamn - 1]$ respectivamente.

El programa de la Figura 5.8 ilustra la definición y uso de un *array* bidimensional o matriz formado por dos filas y cinco columnas. La matriz se inicia al ser definida. Observe la sintaxis para iniciar una matriz en su definición. Cada fila de la matriz se especifica encerrando sus valores entre llaves y separando los valores individuales por comas, por ejemplo, la primera fila es {1, 2, 3, 4, 5}. A su vez, cada fila se separa de la siguiente fila mediante una coma y el conjunto de filas se encierra entre llaves. Realmente no es necesario encerrar las distintas filas entre llaves, por lo que se podría haber iniciado la matriz así:

```
int m[2][5] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

Para mejorar la legibilidad se puede situar cada fila en una línea distinta:

```
int m[2][5] = { 1, 2, 3, 4, 5,
                6, 7, 8, 9, 10 };
```

Observe que para acceder a todos los elementos de un *array* bidimensional es preciso usar dos ciclos **for**, uno para generar los índices de la primera dimensión—las filas—y otro anidado para generar, por cada fila, los índices de la segunda dimensión—las columnas.

5.3. Las cadenas de caracteres

Una cadena de caracteres es un tipo de dato que permite almacenar una secuencia de caracteres. La biblioteca estándar de C++ dispone del tipo de dato string para almacenar y manipular cadenas de caracteres. Para poder utilizar el tipo de dato string hay que incluir—realizar un include—el fichero de cabecera string.

El programa de la Figura 5.9 ilustra el uso del tipo de dato string. La segunda línea del programa utiliza la directiva include para incluir el fichero de cabecera string. La función main contiene dos definiciones de variables de tipo string, la segunda de ellas inicia la variable en la propia definición utilizando el literal de cadena “Francisco”. El cuerpo del ciclo

```
#include <iostream>
using namespace std;

int main () {
  int m[2][5] = { {1, 2, 3, 4, 5}, {6, 7, 8, 9, 10} };
  m[0][0] = -1;
  for (int f = 0; f < 2; f++) {
     cout << "Fila " << f << ": ";
     for (int c = 0; c < 5; c++)
        cout << m[f][c] << ' ';
     cout << endl;
  }
  return 0;
}
```

Figura 5.8: Ejemplo de array bidimensional.

do while contiene la instrucción: cin >> nombre; que permite la lectura de una cadena de la entrada estándar y su almacenamiento en la variable nombre. El programa también utiliza los operadores de comparación != y == para comparar cadenas de caracteres. Tenga en cuenta que las comparaciones entre cadenas son sensibles a las mayúsculas, es decir, un carácter en minúsculas se considera distinto del mismo carácter en mayúsculas. Por último, en el cuerpo de la parte **else** de la instrucción **if** se utiliza cout para mostrar en la salida estándar el contenido de una variable de tipo string. El efecto de una instrucción como: cout << ADIVINA; es enviar a la salida estándar la secuencia de caracteres almacenada en la variable de tipo string ADIVINA.

5.3.1. Acceso a caracteres y comparaciones entre cadenas

Dada una variable s de tipo string, se puede acceder a un carácter individual de la cadena almacenada en s mediante la expresión s[i], donde i es una expresión entera que está en el rango $[0, tam-1]$ y tam es el número de caracteres de la cadena s. Si el índice i está fuera del rango válido, el resultado de la expresión s[i] es indefinido y probablemente desastroso. La expresión s[i] es de tipo carácter, es decir, del tipo **char** de C++.

El programa de la Figura 5.10 cuenta el número de ocurrencias de un carácter leído de la entrada estándar en una cadena de caracteres. El ciclo **for** realiza la cuenta, utilizando la expresión FRASE[i] para acceder al carácter que ocupa la posición i en la cadena almacenada en FRASE. Como en los vectores, los índices comienzan en cero. La expresión s.length() produce el número de caracteres que contiene la cadena s. En el programa se utiliza una expresión de este tipo en la condición del ciclo **for**.

```
#include <iostream>
#include <string>
using namespace std;

int main () {
  string nombre;
  const string ADIVINA = "Francisco";
  do {
     cout << "Avidina el nombre (FIN para terminar): ";
     cin >> nombre;
  } while (nombre != ADIVINA && nombre != "FIN");
  if (nombre == ADIVINA)
     cout << "Enhorabuena, lo adivinaste\n";
  else
     cout << "El nombre es: " << ADIVINA << '\n';
  return 0;
}
```

Figura 5.9: Ejemplo de uso del tipo de dato string.

Se pueden utilizar los operadores relacionales <, >, <= y >= para comparar lexicográficamente dos cadenas de caracteres. Por ejemplo, si s vale "Juan" y t vale "Julio", entonces la expresión s < t es cierta, puesto que en la primera posición de las cadenas en que difieren en un carácter se tiene que el carácter de s ('a') es menor o lo que es lo mismo, precede lexicográficamene, al carácter de t ('l').

La biblioteca estándar de C++ contiene un gran número de utilidades para la manipulación de datos de tipo cadena que permiten, por ejemplo, concatenar cadenas, extraer subcadenas o buscar la primera ocurrencia de una subcadena dentro de otra. Sin embargo, estas utilidades no se estudiarán en estos apuntes.

5.3.2. Lectura de cadenas con espacios en blanco

La lectura de una cadena con una instrucción del tipo: cin >> nombre;, como la realizada en el programa de la Figura 5.9, produce la lectura de una cadena de caracteres de la entrada estándar de la siguiente manera:

- En primer lugar se ignoran todos los caracteres blancos que se encuentren inicialmente en la entrada.
- A continuación se lee de la entrada, y se almacena en la variable nombre, la secuencia de caracteres no blancos que se encuentren en la entrada.

```
#include <iostream>
#include <string>
using namespace std;

int main () {
  const string FRASE = "Burro grande, ande o no ande";
  int cont = 0;
  char c;
  cout << "Introduzca un carácter: ";
  cin >> c;
  for (int i = 0; i < FRASE.length(); i++)
     if (c == FRASE[i])
         cont++;
  cout << cont << "apariciones de " << c << " en " << FRASE << endl;
  return 0;
}
```

Figura 5.10: Ejemplo de acceso individual a los caracteres de una cadena.

- Cuando se detecta un nuevo carácter blanco la lectura termina y el carácter blanco permanece en el flujo de entrada.

Se recuerda que los caracteres blancos son el espacio en blanco, el tabulador y el salto de línea. Esta forma de lectura es la que se desea cuando se quiere leer un dato como el nombre propio de una persona. Sin embargo, si se desea leer el nombre completo—con apellidos—de una persona o si se quiere leer un nombre compuesto, como `Luis Alfonso`, esta forma de lectura no nos sirve. En el caso del nombre compuesto `Luis Alfonso` sólo nos serviría para leer `Luis`. Una instrucción del tipo: cin >> nombre; no permite leer cadenas que contengan espacios en blanco.

El programa de la Figura 5.11 ilustra la lectura de una cadena con espacios en blanco. Esto se realiza mediante la siguiente instrucción:

```
getline (cin, var);
```

donde var es el nombre de una variable de tipo string. El efecto de esta instrucción es la lectura de todos los caracteres encontrados en la entrada estándar y su almacenamiento en la cadena var, incluidos espacios en blanco y tabuladores, hasta que se encuentre un carácter de nueva línea, que provoca la detención de la lectura. El carácter nueva línea es eliminado del flujo de entrada, pero no se inserta en la cadena var.

```
#include <iostream>
#include <string>
using namespace std;

int main () {
  string nombre;
  cout << "Introduzca su nombre completo: ";
  getline (cin, nombre);
  cout << "Nombre completo: " << nombre << endl;
  cout << "Introduzca su nombre completo: ";
  cin >> nombre;            // Solo lee una palabra
  cout << "Nombre completo: " << nombre << endl;
  return 0;
}
```

Figura 5.11: Ejemplo de lectura de cadenas con espacios en blanco.

5.3.3. Un problema al leer cadenas con getline

El programa de la Figura 5.12 ilustra un problema que ocurre cuando se mezcla la lectura de datos con instrucciones del tipo: cin >> var con la lectura de cadenas mediante getline. Lo que ocurre en este programa es lo siguiente. El usuario teclea un número—por ejemplo 18—y a continuación pulsa la tecla *enter*, que produce el carácter nueva línea. Es preciso que el usuario pulse la tecla *enter* puesto que hasta que no lo haga los datos tecleados no llegan al programa. La instrucción cin >> n provoca la lectura del número y su almacenamiento en la varible n. La lectura se detiene cuando se llega a un carácter no numérico en la entrada estándar, en este caso el carácter nueva línea, que se simboliza como \n en la Figura 5.13. El carácter no numérico leído, el carácter nueva línea, se deja en la entrada estándar. Por lo tanto, cuando se ejecuta la instrucción getline la entrada estándar aún contiene el carácter nueva línea—la posición de la flecha en la Figura 5.13 indica dónde comienza la siguiente lectura de la entrada estándar. Como se indicó en la sección previa, getline lee caracteres en una cadena hasta que encuentra el carácter nueva línea, el carácter nueva línea no se introduce en la cadena y se elimina de la entrada estándar. Al ser un carácter nueva línea el primer carácter a procesar de la entrada estándar, el efecto de la instrucción getline del programa es leer una cadena vacía y eliminar el carácter nueva línea de la entrada estándar—veáse la Figura 5.13. Si ejecuta el programa verá que no puede teclear ninguna cadena de caracteres y en la pantalla aparecerá lo siguiente—suponiendo que ha introducido el 18 como número:

```
18
```

```
#include <iostream>
using namespace std;

int main () {
    int n;
    cout << "Número: ";
    cin >> n;
    string s;
    cout << "Cadena: ";
    getline (cin, s);
    cout << n << ' ' << s << endl;
    return 0;
}
```

Figura 5.12: Problema al leer cadenas con getline.

Existen varias formas de solucionar este problema. A continuación se proponen dos. La primera consiste en preceder la instrucción getline con la instrucción cin >> ws;. O sea, escribir:

```
cin >> ws;
getline (cin, s);
```

El efecto de la instrucción cin >> ws; es eliminar de la entrada estándar todos los caracteres blancos consecutivos que se encuentren. Por lo tanto, se eliminaría el carácter nueva línea que ha quedado pendiente en la entrada estándar. Se puede obtener el mismo efecto producido con las dos instrucciones anteriores utilizando la siguiente instrucción:

```
getline (cin >> ws, s);
```

Sustituya en el programa de la Figura 5.12 la sentencia getline(cin, s) por getline(cin >> ws, s) y observe cómo ahora sí puede introducir la cadena de caracteres.

La Figura 5.14 ilustra el estado de la entrada estándar para una ejecución en la que se utiliza la instrucción cin >> ws . Puesto que esta instrucción elimina de la entrada estándar una secuencia consecutiva de espacios en blanco, en el ejemplo de la Figura 5.14 también elimina el espacio blanco inicial de la cadena `` la casa''. Si se desea que una cadena pueda empezar con espacios en blanco entonces se propone sustituir la instrucción getline inicial por el siguiente ciclo:

```
do {
    getline (cin, s);
} while (s.find_first_not_of (" \t") == string::npos);
```

Este ciclo lee una cadena con getline mientras que la cadena leída no contenga al menos un carácter distinto del carácter blanco o el tabulador.

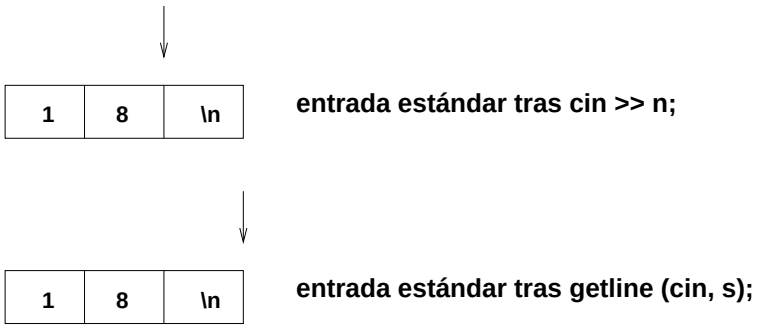

Figura 5.13: Estado de la entrada estándar en una ejecución del programa de la Figura 5.12.

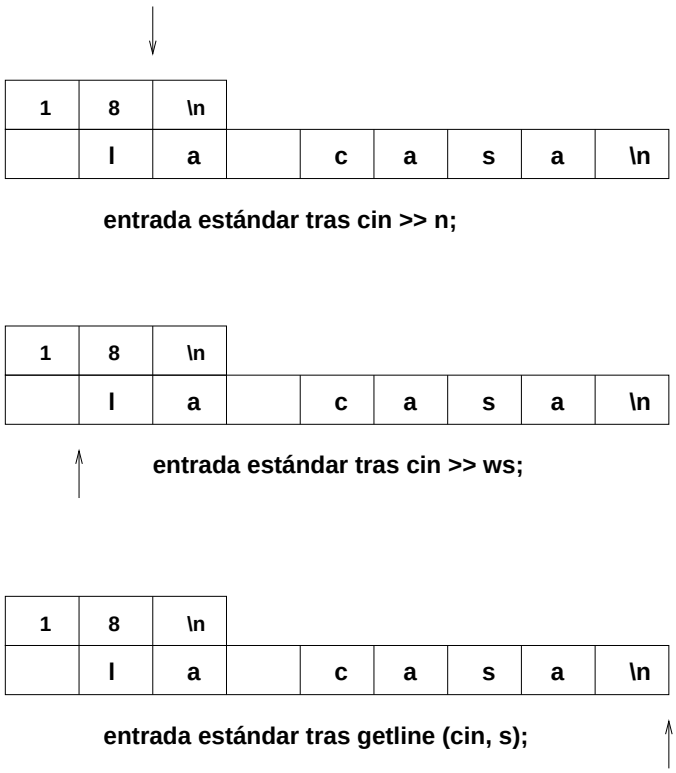

Figura 5.14: Estado de la entrada estándar si se utiliza cin >> ws.

```
#include <iostream>
using namespace std;

int main () {
   string s;    // si no se asigna valor, la cadena queda vacía
   cout << s << " - " << s.length() << " caracteres\n";
   s = "Meursault";
   cout << s << " - " << s.length() << " caracteres\n";
   s = "";      // la cadena queda vacía
   cout << s << " - " << s.length() << " caracteres\n";
   return 0;
}
```

Figura 5.15: Programa que usa una cadena vacía.

5.3.4. La cadena nula o vacía

Una cadena nula o vacía es aquella que no contiene ningún carácter. Su longitud es, por tanto, cero caracteres. El programa de la Figura 5.15 trabaja con cadenas vacías. Cuando no se asigna un valor en la definición de una cadena, como en el caso de la variable `s`, la cadena queda vacía. Si se observa la salida del programa, se comprobará que `s.length()` devuelve el valor cero. Para asignarle una cadena vacía a una cadena hay que utilizar el literal `""`—dobles comillas seguidas de dobles comillas, sin ningún espacio en medio—, como en la última asignación del programa de la Figura 5.15.

5.3.5. El tamaño dinámico de los *strings*

A diferencia de los vectores estudiados en este tema, cuyo tamaño es estático y se define en tiempo de compilación—véase la Sección 5.1.3—, los *strings* tienen un tamaño dinámico, es decir, puede cambiar durante la ejecución de un programa. Esto, sin duda, facilita su uso—los *strings* son dinámicos porque en su implementación se utilizan técnicas de memoria dinámica explicadas en el Tema 8.

Existen operaciones que cambian el tamaño de un *string*, mientras que otras no lo hacen. Vamos a estudiar algunas de las operaciones sobre *strings* con el programa de la Figura 5.16 para ver si cambian su tamaño:

- El programa comienza definiendo la variable `s` de tipo `string`. Cuando no se asigna un valor en la definición de una cadena, como en el caso de `s`, la cadena queda vacía, es decir, con cero caracteres.

```
#include <iostream>
#include <string>
using namespace std;

int main () {
   string s;
   cout << s << '\n';
   s = "hola";
   cout << s << '\n';
   cin >> s;
   cout << s << '\n';
   s = "hola";
   s[20] = "X";
   return 0;
}
```

Figura 5.16: El tamaño dinámico de los *strings*.

- A continuación se utiliza el operador de asignación para asignarle el valor `hola` a `s`. El operador de asignación puede cambiar el tamaño de los *strings*. En este ejemplo `s` pasa de tener tamaño 0 a tener tamaño 4.
- El operador de lectura—>>—también puede cambiar el tamaño de un *string*. Tras la lectura, `s` cambiará su tamaño al de la longitud de la cadena de caracteres leída de la entrada estándar.
- Finalmente, como ya se indicó en la Sección 5.3.1, el operador de indexación—[]—no cambia el tamaño de un *string*. Por lo tanto, la última instrucción no cambia el tamaño de `s` y, puesto que intenta acceder a la posición 20 de `s` y el tamaño de `s` es 4, su uso es incorrecto.

5.4. Las estructuras

C++ permite que el programador defina nuevos tipos de datos. Una forma de hacerlo es mediante una *estructura*. Una estructura define un nuevo tipo de dato que está compuesto por una colección de datos de tipos existentes—ya sean tipos básicos o estructurados como *arrays*, cadenas de caracteres o estructuras ya definidas. Las estructuras son muy útiles pues permiten definir tipos de datos que se ajustan con exactitud a las necesidades de representación de cualquier entidad. Con una estructura se puede representar un alumno, un seguro, un vehículo, etc.

La sintaxis de definición de una estructura es:

```
struct nombre {
    tipo1 campo1;
    tipo2 campo2;
    ...
    tipon campon;
};
```

La definición de una estructura comienza con la palabra reservada **struct**, seguida del nombre del nuevo tipo que se está definiendo. A continuación, y encerrados entre llaves, aparecen los datos que componen el nuevo tipo. A cada uno de los datos se le llama *campo* o, en C++, *miembro*. La definición de cada campo tiene la estructura de la definición de una variable, es decir, en primer lugar se especifica el tipo del campo y a continuación su nombre. Observe que tras la llave que cierra la especificación de los campos de la estructura debe aparecer un punto y coma. Este punto y coma tiende a olvidarse, pues es uno de los pocos lugares en que es necesario utilizar un punto y coma tras una llave de cierre de bloque.

El programa de la Figura 5.17 ejemplifica la definición y uso de una estructura. El nombre de la estructura es Persona y contiene información sobre un individuo, en este caso sólo dos campos: su nombre y edad. Una vez definida la estructura Persona se ha creado un nuevo tipo que puede ser utilizado casi de la misma manera que cualquier tipo básico. Por ejemplo, la primera línea de main define una variable, llamada p, de tipo Persona.

Para acceder al campo de nombre c de una variable e de tipo estructura hay que utilizar la sintaxis e.c. La expresión e.c puede utilizarse en cualquier lugar en que pueda utilizarse una variable del mismo tipo que el tipo del campo c. Por ejemplo, en el programa de la Figura 5.17 el campo nombre de la estructura p es leído desde la entrada estándar: cin >> p.nombre;. Al campo edad se le asigna el valor 18: p.edad = 18;, y el valor de ambos campos es enviado a la salida estándar en las dos instrucciones que preceden a la sentencia **return**.

El programa de la Figura 5.18 ilustra dos características de las estructuras. La primera característica es que se puede iniciar una variable de tipo estructura en el momento de su definición. Para ello—vea la definición de p1 en el programa—hay que aportar valores adecuados para los distintos campos de la variable. Los valores se separan mediante comas y el conjunto de valores debe aparecer encerrado entre llaves, tras la llave de cierre es obligatorio el punto y coma. La segunda característica es que se puede utilizar el operador de asignación para asignar a una variable de tipo estructura otra estructura del mismo tipo. A partir de la asignación ambas variables contienen el mismo valor.

No es posible utilizar directamente los operadores relacionales (==, !=, <, ...) sobre variables de tipo estructura. Si se quiere comprobar si dos estructuras son iguales habrá que comprobar si todos sus campos son iguales. Tampoco es posible utilizar directamente la sintaxis: cout << e;, donde e es una variable de tipo estructura. Si se quiere enviar el contenido de una variable de tipo estructura a la salida estándar habrá que enviar el contenido de sus campos uno a uno. No obstante, C++ posee mecanismos para que el programador pueda

```
#include <iostream>
#include <string>
using namespace std;

struct Persona {
   string nombre;
   int edad;
};

int main () {
  Persona p;
  cout << "Introduzca un nombre: ";
  cin >> p.nombre;
  p.edad = 18;
  cout << "Datos:\n";
  cout << "Nombre: " << p.nombre << endl;
  cout << "Edad: " << p.edad << endl;
  return 0;
}
```

Figura 5.17: Ejemplo de definición y uso de estructura.

definir los operadores de comparación para un tipo de estructura, así como el operador <<, pero esto requiere unos conocimientos que quedan fuera de los objetivos de estos apuntes.

5.4.1. Estructuras anidadas

Una *estructura anidada* es aquella que incluye un campo de tipo estructura. Por ejemplo, el programa de la Figura 5.19 define la estructura anidada Alumno que consta de un campo, nac, que es a su vez una estructura de tipo Fecha. Es posible iniciar una estructura anidada en su definición, como ocurre con la variable al en el programa, observe el uso de llaves para especificar los campos de la estructura nac.

En el mismo programa se muestra el contenido de la estructura en la salida estándar. Observe la sintaxis para mostrar la fecha. Como el campo nac es una estructura se usa la sintaxis: al.nac.dia para acceder al campo dia de la estructura nac que, a su vez, es un campo de la estructura al. Se podría utilizar la sintaxis (al.nac).dia, pero no es necesaria, pues el operador punto se asocia de izquierda a derecha.

Dada la definición: Alumno al;, al es una variable de tipo Alumno, al.nac es una variable de tipo Fecha y al.nac.dia, al.nac.mes y al.nac.anio son variables de tipo integer.

```
#include <iostream>
#include <string>
using namespace std;

struct Persona {
   string nombre;
   int edad;
};

int main () {
  Persona p1 = { "Luis", 18 };
  Persona p2 = p1;
  p2.edad++;
  cout << Datos:\n";
  cout << "Nombre: " << p2.nombre << endl;
  cout << "Edad: " << p2.edad << endl;
  return 0;
}
```

Figura 5.18: Ejemplo de iniciación y asignación de estructuras.

```
#include <iostream>
#include <string>
using namespace std;

struct Fecha {
   int dia, mes, anio;
};

struct Alumno {
   string nombre;
   Fecha nac;
};

int main () {
  Alumno al = { "Pilar", {28, 8, 1990} };
  cout << "Nombre: " << al.nombre << endl;
  cout << "Fecha de nacimiento: " << al.nac.dia << "/"
       << al.nac.mes << "/" << al.nac.anio << endl;
  return 0;
}
```

Figura 5.19: Ejemplo de estructura anidada.

```
#include <iostream>
#include <string>
using namespace std;

struct Fecha { int dia, mes, anio; };

struct Alumno {
   Fecha nac;
   string nombre;
   double notas[3];
};

int main () {
  Alumno v[2] = { {{28, 8, 1990}, "Pilar", {5, 7, 9} },
                  {{16, 12, 1991}, "María José", {6.5, 7, 8} }
                };
  for (int i = 0; i < 2; i++) {
    cout << v[i].nombre;
    cout << " (" << v[i].nac.dia << "/" << v[i].nac.mes << "/"
         << v[i].nac.anio << ")";
    cout << " Notas: " << v[i].notas[0] << " " << v[i].notas[1] << " "
         << v[i].notas[2] << endl;
  }
  return 0;
}
```

Figura 5.20: Ejemplo de vector de estructuras.

5.4.2. Vector de estructuras

Una estructura de datos que se utiliza con mucha frecuencia es un vector de estructuras. Mediante un vector de estructuras se puede representar un conjunto de entidades de cualquier tipo, por ejemplo, los clientes de un dentista o los zapatos que se venden en una zapatería.

El programa de la Figura 5.20 define un vector con dos estructuras de tipo Alumno. El vector se inicia en su declaración, usando llaves para definir los campos de tipo estructura y vector. Después se utiliza un ciclo para mostrar en la salida estándar el contenido del vector. Observe la sintaxis: v[i].nombre. Con v[i] se accede al elemento de índice i del vector. Como este elemento es una estructura se utiliza el operador punto para acceder a sus campos—en el ejemplo, al campo nombre.

5.5. Ejercicios

1. Escribe un programa que lea de la entrada estándar un vector de números y muestre en la salida estándar los números del vector con sus índices asociados.

2. Escribe un programa que defina un vector de números y muestre en la salida estándar el vector en orden inverso—del último al primer elemento.

3. Escribe un programa que defina un vector de números y calcule la suma de sus elementos.

4. Escribe un programa que defina un vector de números y calcule la suma acumulada de sus elementos.

5. Desarrolle un programa que lea de la entrada estándar un vector de enteros y determine el mayor elemento del vector.

6. Escriba un programa que defina un vector de números enteros, solicite al usuario un entero y muestre un mensaje en la salida estándar indicando si el entero introducido por el usuario se encuentra en el vector.

7. Escribe un programa que defina un vector de números y calcule si existe algún número en el vector cuyo valor equivale a la suma del resto de números del vector.

8. Desarrolle un programa que determine el primer y segundo elementos mayores de un vector de enteros.

9. Escribe un programa que calcule la desviación típica de n números. El valor de n se lee de la entrada estándar. La desviación típica se calcula mediante la siguiente fórmula:

$$\sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2}$$

10. Escriba un programa que lea el nombre y la nota de 10 alumnos y calcule la calificación promedio. El programa debe mostrar en la salida estándar el nombre de aquellos alumnos cuya nota supera la media. Haga dos versiones del programa. En la primera versión utilice dos vectores: uno para almacenar las notas de los alumnos y otro para almacenar sus nombres. En la segunda versión del programa utilice un único vector de estructuras de tipo alumno con dos campos: nota y nombre.

11. Un número primo es un entero mayor que 1 cuyos únicos divisores enteros positivos son el 1 y él mismo. Un método para encontrar todos los números primos en un rango de 1 a N es el conocido como Criba de Eratóstenes. Considere la lista de números entre el 2 y N. Dos es el primer número primo, pero sus múltiplos (4, 6, 8, ...) no lo son, por

lo que se tachan de la lista. El siguiente número después del 2 que no está tachado es el 3, el siguiente primo. Entonces tachamos de la lista todos los múltiplos de 3 (6, 9, 12, ...). El siguiente número que no está tachado es el 5, el siguiente primo, y entonces tachamos todos los múltiplos de 5 (10, 15, 20, ...). Repetimos este procedimiento hasta que lleguemos al primer elemento de la lista cuyo cuadrado sea mayor que N. Todos los números que no se han tachado en la lista son los primos entre 2 y N.

12. Realice un programa que dado un vector de números reales obtenga la menor diferencia absoluta entre dos elementos consecutivos del vector.

13. Realiza un programa que defina dos vectores de caracteres y después almacene el contenido de ambos vectores en un nuevo vector situando en primer lugar los elementos del primer vector seguido por los elementos del segundo vector. Muestre el contenido del nuevo vector en la salida estándar.

14. Dados dos vectores A y B de n y m enteros respectivamente, ordenados de menor a mayor y sin repetidos, escribe un programa que construya un nuevo vector C de tamaño k, con $k \leq n + m$, que sea el resultado de mezclar los vectores A y B, de manera que el vector C esté ordenado y no contenga elementos repetidos.

15. Realice un programa que defina un vector de enteros y sitúe al principio del vector los enteros pares y después los impares. Sugerencia: utilice dos vectores auxiliares.

16. Realice un programa como el del ejercicio anterior pero sin utilizar vectores auxiliares.

17. Realiza un programa que defina una matriz de 3x3 y escriba un ciclo para que muestre la diagonal principal de la matriz.

18. Realizar un programa que lea los datos de un *array* de 3x3 desde la entrada estándar y calcule su determinante.

19. Realice un programa que lea una matriz de 3x3 y cree su matriz traspuesta. La matriz traspuesta es aquella en la que la columna *i* era la fila *i* de la matriz original.

20. Realice un programa que calcule la suma de dos matrices cuadradas de 3x3.

21. Una matriz nxm se dice que es simétrica si $n = m$ y $a_{ij} = a_{ji} \forall i, j : 1 \leq i \leq n, 1 \leq j \leq n$. Desarrollar un programa que determine si una matriz es simétrica o no.

22. Realice un programa que calcule el producto de dos matrices cuadradas de 3x3.

23. Dada una matriz bidimensional notas cuyo elemento notas[i][j] contiene la calificación del j-ésimo problema de un examen del i-ésimo estudiante, y un factor de ponderación pesos, cuyo elemento pesos[j] denota la ponderación correspondiente al j-ésimo problema de un examen, formule un programa que construya un vector alumno, de tal forma que alumno[i] contenga la calificación final del i-ésimo estudiante.

24. En un congreso cuya duración es de 5 días, tienen lugar conferencias en 5 salas. Se desea saber:

    a) El total de congresistas que asisten a cada una de las salas.

    b) El total de congresistas asistentes cada día al congreso.

    c) La media de asistencia a cada sala.

    d) La media de asistencia diaria.

    Como datos de entrada tenemos el número de asistentes para las diferentes salas, para cada uno de los días del congreso. Realice un programa que calcule estos datos.

25. Realice un programa que lea una cadena de caracteres de la entrada estándar y muestre en la salida estándar cuántas ocurrencias de cada vocal existen en la cadena.

26. Realice un programa que lea líneas de la entrada estándar hasta que se lea una línea cuyo contenido sea la cadena `FIN`. El programa debe mostrar en la salida estándar la línea más larga de las leídas y también la menor desde un punto de vista lexicográfico.

27. Escribir un programa que solicite de la entrada estándar un nombre y compruebe si el nombre se encuentra o no en un vector iniciado con una serie de nombres.

28. Realice un programa que lea una cadena de caracteres e indique si es un palíndromo. Un palíndromo es una frase o palabra que se lee igual de delante hacia atrás que de atrás hacia delante, por ejemplo: reconocer o anilina. Para simplificar suponga que la cadena leída no contiene ni mayúsculas, ni signos de puntuación, ni espacios en blanco ni tildes.

29. Realice un programa como el del ejercicio anterior pero en el que se permitan espacios en blanco. Este programa debe reconocer como palíndromos las frases: “dabale arroz a la zorra el abad” y “la ruta nos aporto otro paso natural”.

30. Realice un programa que lea una cadena de caracteres e indique cuántas palabras tiene. Una palabra puede venir separada de otra por uno o más caracteres de espacio en blanco.

31. Implementar el juego del ahorcado. El programa dispondrá de un menú con tres opciones: introducir palabra, adivinar palabra y salir. La primera opción permite introducir la palabra que otro jugador—o nosotros mismos, para probar el programa—ha de adivinar. La segunda opción sólo podrá llevarse a cabo si ha sido introducida previamente una palabra. De ser así aparecerá una cadena formada por guiones—tantos como letras contiene la palabra. El programa irá pidiendo una letra tras otra. Si la letra es válida aparecerá en la cadena en la posición correspondiente, si no es así contaremos un fallo. El programa termina cuando se han acertado todas las letras o se ha fallado seis veces.

32. Defina una estructura que indique el tiempo empleado por un ciclista en una etapa. La estructura debe tener tres campos: horas, minutos y segundos. Escriba un programa que dado un vector con los tiempos que un ciclista ha empleado en cada etapa calcule el tiempo total empleado en correr todas las etapas.

33. Defina una estructura que sirva para representar a una persona. La estructura debe contener dos campos: el nombre de la persona y un valor de tipo lógico que indica si la persona tiene algún tipo de minusvalía. Realice un programa que dado un vector de personas rellene dos nuevos vectores: uno que contenga las personas que no tienen ninguna minusvalía y otro que contenga las personas con minusvalía.

34. Escriba un programa que almacene y trate la información sobre las elecciones a delegado de una clase. El programa debe leer el número de candidatos que se presentan a delegado (al menos deben presentarse 3 alumnos). Para cada candidato el programa debe leer su nombre, DNI y fecha de nacimiento. A continuación, el programa debe ir leyendo y almacenando en una estructura adecuada los votos emitidos por cada uno de los alumnos presentes en la clase. El voto será un número entero que indica lo siguiente:

    - Si el número es 0 el voto se considera "voto en blanco".
    - Si el número está en el intervalo $[1, N]$, siendo N el número de candidatos, el voto se contabilizará como voto para el candidato correspondiente.
    - Si el número es -1, significa que la emisión de votos ha finalizado.
    - Si el número es distinto a los anteriores, el voto se considera "voto nulo".

    El programa debe mostrar el número de votos emitidos, el número de votos nulos, el número de votos en blanco y el número de votos conseguidos por cada candidato, mostrando además su nombre. Por último, el programa debe determinar los candidatos que han resultado elegidos como delegado y subdelegado mostrando su nombre, DNI y fecha de nacimiento. Si más de un candidato empataron con el mayor número de votos elija como delegado y subdelegado a cualquiera de los candidatos.

35. Se decide cambiar el modo de votación del ejercicio anterior. Ahora el voto se realiza indicando el DNI del candidato al que se quiere votar. Teniendo en cuenta la siguiente casuística:

    - Un DNI "-" significa voto en blanco.
    - Un DNI "FIN" significa que ya no hay más votos.
    - Un DNI distinto de cualquiera de los candidatos y de "-" y "FIN" es un voto nulo.

    Calcule y muestre los mismos datos que en el ejercicio anterior.

36. Desarrolle un programa en C++ que trabaje con un vector de personas. Cada persona se almacena en una estructura con los siguiente campos: nombre, peso en kilos y altura en metros. El programa debe comenzar leyendo de la entrada estándar los datos de varias personas. A continuación debe hacer lo siguiente:

    - Mostrar en la salida estándar un listado con los datos de las personas introducidas. El listado debe incluir el índice de masa corporal de cada persona, éste se calcula como:
    $$IMC = \frac{peso}{altura^2}$$
    - Mostrar en la salida estándar el nombre de la persona con mayor IMC.
    - Un listado con los nombres de las personas cuya altura supera una introducida por el usuario.
    - Indicar si la persona con menor IMC es también la más pequeña.

37. Modifica el programa sobre personas con posible minusvalía para que los datos se lean de la entrada estándar.

Tema 6
Funciones

Una función es un fragmento de código que permite ejecutar una serie de instrucciones parametrizadas. Las funciones son de gran utilidad porque permiten estructurar el código de un programa. Una función realiza una tarea concreta y puede ser diseñada, implementada y depurada de manera independiente al resto de código. En este tema se analizan los principales aspectos relacionados con la definición, invocación y uso de funciones en C++.

6.1. Definición de funciones

La sintaxis básica para la definición de una función en C++ es la siguiente:

```
tipo nombre (tipo1 var1, tipo2 var2, ..., tipon varn)
{
   conjunto de instrucciones
}
```

A la primera línea se le llama *signatura* de la función e incluye el tipo del valor devuelto por la función—o *valor de retorno*—, el nombre de la función y la lista de *parámetros formales* de la función. La lista de parámetros formales consiste en un listado de variables separadas por comas y encerradas entre paréntesis. Cada variable de la lista de parámetros formales viene precedida por su tipo. A continuación, y delimitado por llaves, viene el conjunto de instrucciones asociado a la función, también llamado *cuerpo de la función*.

La función maximo del programa de la Figura 6.1 calcula el máximo de los dos valores que recibe como parámetro. La función almacena el máximo calculado en la variable m y termina ejecutando la sentencia: **return** m;. La sentencia **return** tiene la siguiente sintaxis:

```
return expresión;
```

donde expresion es una expresión del mismo tipo que el tipo que devuelve la función. También se permite una expresión de un tipo que admita una conversión implícita al tipo de retorno de la función. El efecto de la sentencia **return** es evaluar su expresión asociada y terminar la ejecución de la función devolviendo el resultado de evaluar la expresión. Una

```
#include <iostream>
using namespace std;

double maximo (double x, double y) {
   double m;
   if (x > y)
      m = x;
   else
      m = y;
   return m;
}

int main () {
  double a, b;
  cout << "Introduzca dos números: ";
  cin >> a >> b;
  cout << "El máximo es: " << maximo (a, b) << endl;
  cout << "El máximo de " << a << " y 10 es: " << maximo(a, 10) << endl;
  a = 8;
  double z = 3 + maximo (a, a-1);
  cout << "z vale: " << z << endl;
  return 0;
}
```

Figura 6.1: Una función que calcula el máximo de dos valores.

función puede contener más de una instrucción **return**, en ese caso la ejecución de la función terminará cuando se ejecute por primera vez una instrucción **return**. Por ejemplo, la siguiente función maximo es una alternativa a la función maximo que aparece en el programa de la Figura 6.1:

```
double maximo (double x, double y) {
   if (x > y)
      return x;
   return y;
}
```

Se analiza ahora el flujo de control asociado a la ejecución de una función. Exceptuando la función main, por cuyo código empieza a ejecutarse un programa, para que se ejecute una función, ésta debe ser *invocada* o *llamada* desde el cuerpo de otra función. Las funciones se pueden invocar o llamar en expresiones, escribiendo el nombre de la función y, encerrados entre paréntesis y separados por comas, una lista de *parámetros reales*. Los parámetros reales son expresiones con las que se inician los parámetros formales de la función a la que se invoca. Es decir, la sintaxis de invocación de una función es la siguiente:

```
nombre (expresión1, expresión2, ..., expresiónn)
```

donde nombre es el nombre de la función y la lista de expresiones es la lista de parámetros reales con que se invoca a la función. La lista de parámetros reales debe coincidir en número y en tipo con la lista de parámetros formales de la función que se invoca. La correspondencia en tipo significa que la expresión que ocupa la posición *i* en la lista de parámetros reales debe ser del mismo tipo que el tipo del parámetro formal que ocupa la posición *i* en la lista de parámetros formales de la función—o en su defecto, de un tipo que admita una conversión implícita al tipo del parámetro formal.

Una vez invocada una función, sus parámetros formales se inician con las expresiones especificadas como parámetros reales en la invocación y se ejecuta el código—el cuerpo—de la función. Cuando termina la ejecución de la función se utiliza su valor de retorno como resultado de evaluar la subexpresión constituida por la llamada a la función.

El programa de la Figura 6.1 contiene tres invocaciones, o llamadas, a la función maximo. La última de estas llamadas se realiza en la instrucción: **double** z = 3 + maximo (a, a−1);. En esta llamada los parámetros reales son las expresiones a y a−1, cuya evaluación produce los valores 8 y 7 respectivamente. Por lo tanto, en esta invocación los parámetros formales x e y de la función maximo se inician con los valores 8 y 7 respectivamente. La función maximo devuelve el valor 8 como resultado de su ejecución; luego, en la expresión 3 + maximo (a, a−1) se utiliza el valor 8 como resultado de evaluar la subexpresión maximo (a, a−1) y, por tanto, se asigna 11 a la variable z.

Una invocación a una función debe realizarse en una expresión. Como C++ admite como instrucción válida a una expresión, se puede escribir una instrucción que sólo contenga una invocación a función como:

```
maximo (a, 10);
```

en este caso el valor de retorno de la función no se utiliza como parte de otra expresión y, por lo tanto, "se pierde".

Por último, indicar que los parámetros formales de una función se pueden utilizar en el cuerpo de la función del mismo modo que cualquier variable definida en la función. La única diferencia es que estos parámetros se inician cada vez que se invoca a la función con los valores de los parámetros reales.

Ejercicio Modifique la función maximo del programa de la Figura 6.1 de manera que contenga una única sentencia **return** y no declare ninguna variable. Sugerencia: utilice uno de los parámetros formales para almacenar el máximo.

```
double maximo (double x, double y) {
   if (x > y)
      y = x;
   return y;
}
```

```
#include <iostream>

void saludo () {
   std::cout << "Hola a todos\n";
}

int main () {
   saludo ();
   return 0;
}
```

Figura 6.2: Ejemplo de función *void*.

6.1.1. Tipos de datos que puede devolver una función

En C++ una función puede devolver valores de aquellos tipos de datos para los que la asignación está definida—la asignación a un tipo de dato está definida si es posible asignar valores a las variables de dicho tipo con el operador de asignación (=).

De los tipos de datos que hemos visto el único para el que la asignación no está definida es el tipo vector. Para el resto de tipos: tipos básicos, estructuras y cadenas de caracteres la asignación está definida y, por tanto, una función puede devolver un dato de tipo básico, de tipo string o de tipo estructura, pero no puede devolver un vector.

6.2. Funciones void y funciones sin parámetros

Se suele utilizar el término *función void* para designar a una función que no tiene valor de retorno. Puesto que la sintaxis de C++ obliga a indicar un tipo de retorno, las funciones *void* utilizan el tipo especial **void** como tipo de retorno. El programa de la Figura 6.2 contiene una función *void* que se limita a enviar un mensaje de saludo a la salida estándar. Además, la función ejemplifica la definición de una función sin parámetros. En la función main puede observar que las funciones sin parámetros se invocan especificando una lista de parámetros reales vacía—es necesario escribir los paréntesis.

Aunque una función no tenga valor de retorno—es decir, aunque sea una función *void*—es posible utilizar la sentencia **return** en su cuerpo para terminar la ejecución de la función. En una función *void* la sentencia **return** debe utilizarse sin expresión asociada, o sea, así:

```
return;
```

Si una función *void* no utiliza ninguna sentencia **return**, la ejecución de la función termina cuando se ejecuta la última instrucción de la función.

Las funciones *void* deben invocarse mediante una instrucción que incluya únicamente la llamada a la función. Por ejemplo, en el programa de la Figura 6.2 la invocación a la función *void* saludo se realiza así: saludo (); . Es decir, la invocación a una función *void* no puede constituir una subexpresión que forma parte de una expresión más compleja.

A diferencia de algunos lenguajes, C++ no permite definir procedimientos, sólo permite definir funciones. En C++ una función *void* es lo más parecido a la definición de un procedimiento.

6.3. La pila: almacenamiento de variables locales

A las variables definidas en una función se les llama *variables locales*, en contraste con las variables definidas fuera de las funciones que son llamadas *variables globales*—véase la Sección 6.4. En C y C++ a las variables locales también se les llama *variables automáticas*.

El ámbito de una variable local—véase el Tema 4, Sección 4.5—abarca desde su lugar de definición en la función hasta el final del bloque más interno en que ha sido definida. Los parámetros formales de una función son variables locales de la función y, a efecto de ámbito, se encuentran definidas en el bloque más externo de la función, es decir, entre las llaves que delimitan el cuerpo de la función.

Las variables locales de una función se almacenan en la *pila*. La pila es una zona de memoria contigua donde se guardan las variables locales de las funciones y alguna otra información, como la dirección de memoria de retorno de una función. El tamaño de la pila varía dinámicamente durante la ejecución de un programa; cuando se invoca a una función la pila crece hacia direcciones superiores de memoria para albergar a sus variables locales—se dice que las variables se apilan—; cuando una función termina la pila decrece y sus variables locales dejan de ser accesibles—se dice que las variables se desapilan. Una variable almacenada en la pila tiene un tiempo de existencia limitado al tiempo en que se ejecuta el código asociado al ámbito de la variable.

La Figura 6.3 ilustra el estado de la pila durante algunos instantes de una ejecución del programa de la Figura 6.1 en la que se han leído los valores 6 y 8 de la entrada estándar. Las variables locales de la función main ocupan las primeras posiciones de la pila. Cuando se invoca a la función maximo la pila crece para albergar—apilar—a las variables locales de la función maximo. Cuando termina la ejecución de la función maximo sus variables locales se desapilan y el tamaño de la pila decrece. De esta forma, la memoria ocupada por las variables desapiladas puede ser utilizada para albergar a las variables locales de las funciones que se invoquen con posterioridad.

Observe que las variables definidas en el bloque más externo de la función main no se desapilan hasta que no termina el programa.

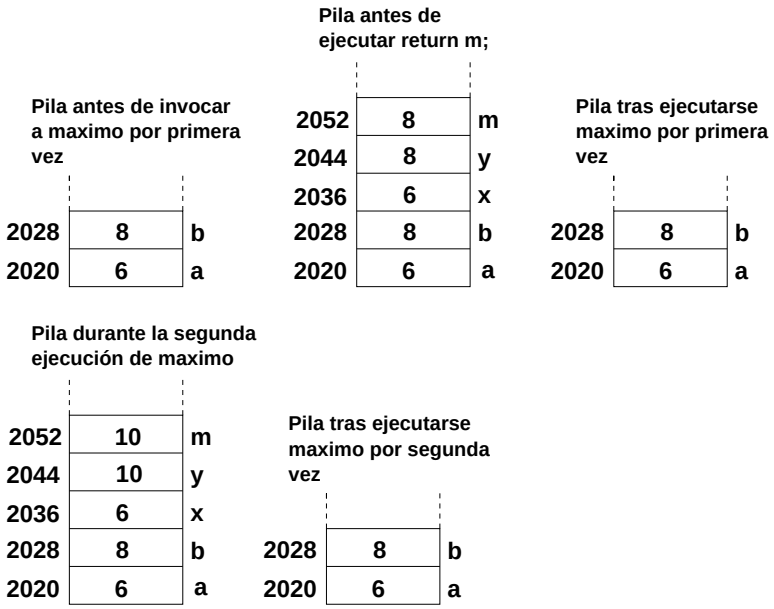

Figura 6.3: Estado de la pila en algunos momentos de una ejecución del programa de la Figura 6.1.

6.4. Variables globales

En el Tema 4, Sección 4.5, se estudió el lugar de definición y ámbito de utilización de las variables locales de una función. En esta sección se continúa con dicho estudio pero se amplía la zona de definición a la totalidad del fichero de texto donde está escrito el programa.

Es posible definir una variable fuera del cuerpo y signatura de cualquier función. A una variable definida de esta forma se le suele llamar *variable global*—en contraste con una variable definida en una función, que recibe el nombre de variable local. El ámbito de una variable global abarca desde el lugar de su definición hasta el final del fichero en que se ha definido, pudiendo ser utilizada en cualquier función situada en dicho ámbito. Una variable local con el mismo nombre que una variable global oculta a la variable global. El programa de la Figura 6.4 ilustra el uso de variables globales. Las variables TAM, v y maximo están definidas fuera de cualquier función y son, por tanto, variables globales. Puesto que están definidas antes de las funciones max y main, éstas pueden utilizarlas. La variable local v definida en main oculta a la variable global v en el cuerpo de main.

El uso de variables globales para implementar la comunicación entre funciones presenta muchos inconveniente, entre otros:

```
#include <iostream>

const int TAM = 4;                    // constante global
float v[TAM] = { 5.2f, 5.4f, 3, 2}; // vector global
float maximo;                         // otra variable global

void max () {
   maximo = v[0];
   for (int i = 1; i < TAM; i++)
      if (v[i] > maximo)
         maximo = v[i];
}

int main () {
   int v = 4;  // esta variable oculta al vector global v
   max ();     // calcula el máximo del vector y guárdalo en maximo
   std::cout << "Máximo: " << maximo << " v: " << v << std::endl;
   return 0;
}
```

Figura 6.4: Ejemplo de definición y uso de variables globales.

1. Una función que se comunica con el resto mediante variables globales no es reutilizable, salvo en un entorno en el que se utilicen las mismas variables globales. Se dice que la función está *fuertemente acoplada* al resto del código.

2. El uso de variables globales puede producir los llamados *efectos colaterales*. Esto ocurre cuando una variable global es ocultada por una variable local inadvertidamente.

No obstante, la comunicación mediante variables globales es muy eficiente, pues se evita la iniciación de parámetros formales con parámetros reales y el apilar y desapilar a los parámetros formales. Su utilización resulta justificable en programas muy específicos, pequeños y que requieran un alto rendimiento. Un programa cuyas funciones se comunican mediante variables globales resulta difícil de entender y mantener, aunque si el programa es pequeño la dificultad puede ser controlable.

Las variables globales se almacenan en una zona de memoria denominada *memoria global*. Una variable global permanece almacenada en la memoria global durante todo el tiempo que dure la ejecución de un programa. Esto resulta necesario porque una variable global puede ser utilizada por más de una función.

En el tema 8 se estudian las características de las tres zonas de memoria de las que dispone un programa para almacenar sus datos: la pila, la memoria global y la memoria dinámica.

6.5. Paso por variable o referencia

Una función en C++ sólo tiene un valor de retorno. Por ejemplo, el programa de la Figura 6.1 devuelve el máximo de los dos valores que recibe como parámetro. Sin embargo, a veces se necesita que una función devuelva más de un valor; por ejemplo, podríamos necesitar una función que devuelva tanto el máximo como el mínimo de los valores recibidos como parámetros. Se puede conseguir devolver más de un valor utilizando como tipo de retorno una estructura que contenga tantos campos como valores a devolver: la función maxmin del programa de la Figura 6.5 devuelve el máximo y el mínimo de los dos valores que recibe como parámetro. Para devolver estos dos valores se define una estructura con dos campos en los que se almacena el máximo y el mínimo calculado. Esta solución no es muy elegante pues implica la definición de una estructura con el único objeto de almacenar los valores de retorno.

En C++ se puede utilizar los parámetros formales de una función para que la función "devuelva" datos. Una posibilidad consiste en utilizar parámetros de tipo puntero, esto se estudia en el Tema 7, Sección 7.2. En lo que resta de sección se estudia la otra posibilidad, que consiste en utilizar variables de tipo referencia.

6.5.1. Referencias

Una *referencia* es un nombre alternativo para un dato u objeto. La sintaxis T& significa referencia a un objeto del tipo T. El programa de la Figura 6.6 ilustra la utilización de referencias. En este programa, las variables r1 y r2 son referencias a datos de tipo entero. Una variable de tipo referencia debe iniciarse en su definición con un objeto del mismo tipo que el tipo asociado a la referencia. Al objeto con que se inicia la referencia lo llamaremos el *objeto referenciado* por la referencia. Una referencia siempre tiene asociado el mismo objeto referencia, a saber, el objeto con que se ha iniciado la referencia. Una vez iniciada una referencia, cuando un programa utiliza la referencia realmente se trabaja con su objeto referenciado. La Figura 6.7 muestra el estado de la pila durante algunos momentos de la ejecución del programa de la Figura 6.6. Para representar el contenido de las variables de tipo referencia se ha utilizado el texto "ref objeto-referenciado" y se ha empleado una flecha para apuntar al objeto o dato referenciado. Observe que cuando el programa ejecuta la instrucción r1 = 4 está almacenando el valor 4 en la zona de memoria asociada a la variable x—puesto que r1 se inició con el valor x. Del mismo modo, cuando el programa ejecuta la instrucción r2++ se incrementa en una unidad el primer elemento del vector v.

6.5.2. Paso por referencia

Las referencias se utilizan principalmente para implementar el *paso por variable* o *por referencia*. El paso por variable se puede utilizar para que una función "devuelva" información

```
#include <iostream>
struct Maxmin {
  double max, min;
};

Maxmin maxmin (double x, double y) {
   Maxmin m;
   if (x > y) {
      m.max = x;
      m.min = y;
   } else {
      m.max = y;
      m.min = x;
   }
   return m;
}

int main () {
   double a, b;
   std::cout << "Introduzca dos números: ";
   std::cin >> a >> b;
   Maxmin sol = maxmin (a, b);
   std::cout << "Máximo: " << sol.max << " Mínimo: " << sol.min << '\n';
   return 0;
}
```

Figura 6.5: Función que devuelve dos datos mediante una estructura.

```
#include <iostream>

int main () {
   int x = 6;
   int v[] = {1, 2};
   int& r1 = x;
   int& r2 = v[0];
   std::cout << x << ' ' << v[0] << ' ' << v[1] << '\n';
   r1 = 4;
   r2++;
   std::cout << x << ' ' << v[0] << ' ' << v[1] << '\n';
   std::cout << r1 << ' ' << r2 << '\n';
   return 0;
}
```

Figura 6.6: Ejemplo de uso de referencias.

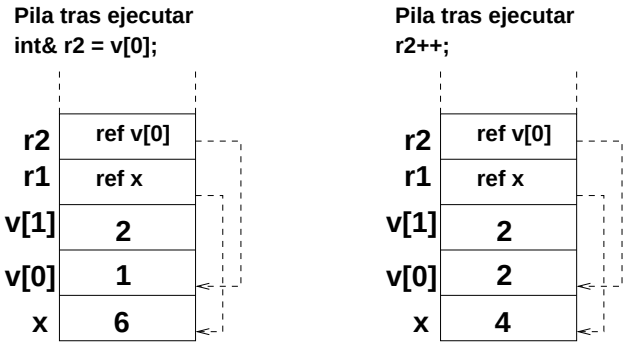

Figura 6.7: Estado de la pila en algunos momentos de la ejecución del programa de la Figura 6.6.

mediante un parámetro. El programa de la Figura 6.8 incluye una función, llamada maxmin, cuyos dos últimos parámetros formales, max y min, tienen semántica de paso por referencia. Cuando se invoca a la función maxmin, max y min se inician con los dos objetos pasados como parámetros reales. Cuando la función maxmin realiza alguna modificación—o consulta—sobre las variables max o min realmente está modificando—o consultando—los objetos referenciados por estas dos variables. Esto permite que tras la ejecución de la función maxmin la función main pueda consultar los valores de máximo y mínimo calculados por la función maxmin.

La Figura 6.9 muestra el contenido de la pila durante algunos momentos de una ejecución del programa de la Figura 6.8 en la que el usuario ha introducido los valores 3 y 7. Cuando la función maxmin asigna valores a las referencias max y min los valores asignados se almacenan en las variables maximo y minimo respectivamente. Esto permite que tras ejecutar la función maxmin las variables maximo y minimo contengan los dos valores calculados por maxmin.

Cuando un parámetro formal es de tipo referencia el parámetro real utilizado en la invocación debe ser una variable—si se usa el calificativo **const** como calificador del parámetro formal no es necesario utilizar una variable—, de ahí el nombre de paso por variable. Aquí el término variable se aplica en el sentido de una zona de memoria accesible mediante un nombre, como una variable, un campo de una estructura o un elemento de un vector.

La función maxmin se invoca desde la función main así:

```
maxmin (a, b, maximo, minimo);
```

Como los dos primeros parámetros de la función maxmin no son referencias no es necesario invocar a maxmin utilizando variables en los primeros dos parámetros reales. Por ejemplo, la siguiente invocación es válida:

```
maxmin (10, 15.4, maximo, minimo);
```

```
#include <iostream>
using namespace std;

void maxmin (double x, double y, double& max, double& min) {
   if (x > y) {
      max = x;
      min = y;
   } else {
      max = y;
      min = x;
   }
}

int main () {
  double a, b, maximo, minimo;
  cout << "Introduzca dos números: ";
  cin >> a >> b;
  maxmin (a, b, maximo, minimo);
  cout << "Máximo: " << maximo << " Mínimo: " << minimo << endl;
  return 0;
}
```

Figura 6.8: Ejemplo de paso por variable o referencia.

Sin embargo, la siguiente invocación es incorrecta, porque el tercer parámetro real no es una variable y su parámetro formal correspondiente es una referencia:

```
maxmin (10, 15.4, 7.7, minimo);
```

6.5.3. Paso por referencia y por copia

Un parámetro formal de una función se puede clasificar en uno de dos tipos posibles teniendo en cuenta si la función trabaja con los datos originales asociados a los parámetros reales o con copias. Los tipos son los siguientes:

- *Parámetro por copia o valor*. La función trabaja con una copia del dato utilizado como parámetro real. Por lo tanto, los cambios realizados por la función en el parámetro formal no permanecen cuando se termina la ejecución de la función.

- *Parámetro por variable o referencia*. La función trabaja con la zona de memoria asociada al parámetro real. Por lo tanto, los cambios realizados por la función empleando el parámetro formal permanecen cuando se termina la ejecución de la función.

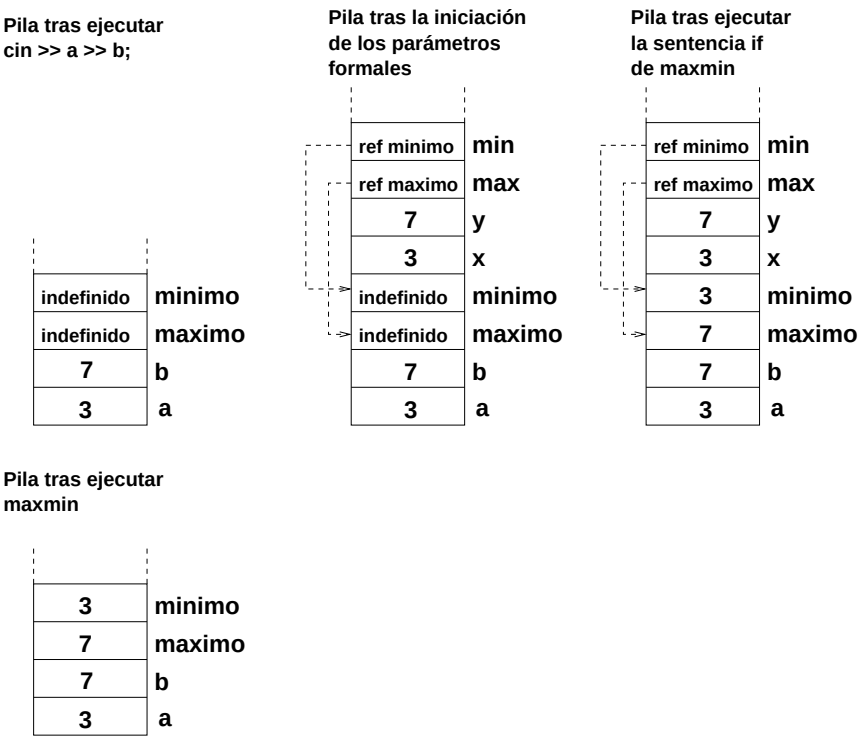

Figura 6.9: Estado de la pila en algunos momentos de una ejecución del programa de la Figura 6.8.

```
#include <iostream>

void f (int x, int& y) {
   x = 1;
   y = 1;
}

int main () {
   int a = 0, b = 0;
   std::cout << a  << ' ' << b << '\n';
   f (a, b);
   std::cout << a  << ' ' << b << '\n';
   return 0;
}
```

Figura 6.10: Función con un parámetro por valor y otro por referencia.

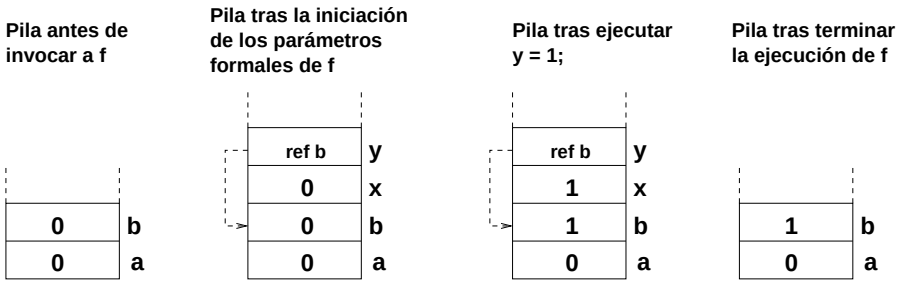

Figura 6.11: Estado de la pila en algunos momentos de la ejecución del programa de la Figura 6.10.

La función f del programa de la Figura 6.10 ilustra los dos tipos de paso de parámetros. El primer parámetro formal, x, se pasa por valor y el segundo parámetro formal, y, se pasa por referencia. La figura 6.11 ilustra el contenido de la pila en algunos momentos de la ejecución de este programa. Como puede observarse las modificaciones realizadas en la variable x se hacen sobre la copia local de la función y, por tanto, se "pierden" al terminar la ejecución de la función f, en contraste con las modificaciones realizadas sobre la variable y.

Dado un tipo T, un parámetro formal x del tipo T& x—es decir, una referencia—tiene semántica de paso por referencia, mientras que un parámetro formal del tipo T x tiene semántica de paso por valor. Una excepción a esto la constituyen los vectores—véase la Sección 6.10.

6.6. Parámetros de entrada y de salida de una función

En la sección previa se ha estudiado que, aunque una función sólo tiene un valor de retorno, se pueden utilizar referencias como parámetros formales con el fin de que una función "devuelva" más de un valor. Los parámetros de una función se pueden clasificar en tres categorías en relación a si la función los utiliza para obtener o para devolver información. Las categorías son las siguientes:

- *Parámetros de entrada*. Son aquellos utilizados por la función para recibir información.
- *Parámetros de salida*. Son aquellos utilizados por la función para devolver información.
- *Parámetros de entrada y salida*. Son aquellos utilizados por la función tanto para recibir como para devolver información.

En la función maxmin del programa de la Figura 6.8 los dos primeros parámetros formales, x e y, son parámetros de entrada; en ellos la función recibe los dos números de los que hay que calcular el máximo y el mínimo. Los dos últimos parámetros de la función maxmin, max y min, son parámetros de salida en los que la función devuelve el máximo y el mínimo calculados. El único parámetro de la función cuadrado del programa de la Figura 6.12 es un parámetro de entrada y salida. En este parámetro la función cuadrado recibe un número y se utiliza el mismo parámetro para devolver en él su cuadrado.

Para implementar los parámetros de salida y de entrada y salida se utilizan parámetros formales de tipo referencia o parámetros formales de tipo puntero—véase el Tema 7, Sección 7.2. Los vectores—véase la Sección 6.10—constituyen una excepción.

Los parámetros de entrada de tipos básicos se suelen pasar por copia, es decir, si el tipo del parámetro formal es T el parámetro se especifica como: T x. Sin embargo, los parámetros de entrada de un tipo estructura se suele pasar por referencia por motivos de eficiencia—véase la Sección 6.9.

6.7. Orden de definición de las funciones

Cuando un compilador de C++ encuentra una llamada a función, como la llamada maximo (a, b) del programa de la Figura 6.1, debe comprobar que esa llamada es correcta sintácticamente. La comprobación consiste en verificar lo siguiente:

- existe una función con ese nombre—en el ejemplo maximo;
- los parámetros reales de la llamada coinciden en número y tipo con los parámetros formales de la función;
- el valor de retorno de la función se utiliza en una expresión adecuada a su tipo.

```
#include <iostream>

void cuadrado (int& x) {
   x = x*x;
}

int main () {
   int valor;
   std::cout << "Introduzca un entero: ";
   std::cin >> valor;
   cuadrado (valor);
   std::cout << "Su cuadrado vale " << valor << '\n';
   return 0;
}
```

Figura 6.12: Ejemplo de función con parámetro de entrada y salida.

Para poder realizar estas comprobaciones el compilador necesita conocer la signatura de la función. Esto se consigue definiendo la función en el fichero antes de ser invocada por primera vez. Modifique el programa de la Figura 6.1 situando la definición de la función maximo tras la definición de la función main y comprobará que al compilar el programa se genera un error de compilación en la primera invocación a la función maximo en main.

Existe, por lo tanto, una restricción en el orden en que deben definirse las funciones en un fichero: una función debe definirse antes de su primera invocación. Como esta limitación puede resultar molesta, C++ proporciona un mecanismo para que se puedan definir las funciones en cualquier orden. Una función puede ser invocada siempre que haya sido definida o *declarada* con anterioridad en el fichero. La declaración de una función consiste en incluir en el fichero el *prototipo* de la función, que equivale a su signatura seguida de un punto y coma. Por ejemplo, en el programa de la Figura 6.13 la función maximo se define tras la función main. Sin embargo, main puede invocar a maximo porque se ha colocado una declaración de maximo antes de la definición de main. De esta forma, cuando el compilador analiza la llamada maximo (a, b) conoce la signatura de la función maximo y puede comprobar si la llamada es correcta sintácticamente.

Al declarar una función no es preciso especificar el nombre de los parámetros formales, pues el compilador sólo necesita conocer el número de parámetros y su tipo. La declaración de la función maximo en el programa de la Figura 6.13 podría haberse escrito así:

```
double maximo (double, double);
```

El uso de declaraciones permite definir las funciones en cualquier orden. Sin embargo, implica cierta redundancia, pues la signatura de la función aparece repetida en el fichero: en la definición y en la declaración. Si modifica la definición de una función y esa modifica-

```
#include <iostream>
using namespace std;

double maximo (double x, double y); // declaración de maximo

int main () {
   double a = 6.5, b = 4.0;
   cout << "El máximo es: " << maximo (a, b) << endl;
   return 0;
}

double maximo (double x, double y) { // definición de maximo
   return (x > y) ? x : y;
}
```

Figura 6.13: Ejemplo de declaración de una función.

ción afecta al tipo o número de sus parámetros, entonces tendrá que actualizar también la declaración de la función. Si no lo hace, el compilador detectará la inconsistencia.

Cuando un fichero con código C++ utiliza una directiva include para usar una biblioteca eso provoca que en una fase de preprocesamiento se sustituya la línea donde aparece la directiva include por el contenido de un fichero de cabecera. Este fichero contiene información que necesita el compilador sobre los objetos definidos en la biblioteca. En esa información se incluye los prototipos de las funciones que define la biblioteca.

6.8. Funciones recursivas

Una función recursiva es aquella que se invoca a sí misma. La recursividad facilita la escritura de código que trabaja con estructuras de datos o problemas que tienen una naturaleza recursiva—como la estructura de datos árbol. Además, produce un código más legible para este tipo de problemas.

C++ permite la definición de funciones recursivas. El programa de la Figura 6.14 incluye una función recursiva que calcula el factorial de un número. En la última instrucción de la función se efectúa la llamada recursiva. Por supuesto, la función factorial admite una implementación no recursiva, como:

```
int factorial (int n) {
   int fact = 1;
   for (int i = 2; i <= n; i++)
      fact *= i;
   return fact;
}
```

```
#include <iostream>

int factorial (int n) {
   if (n < 2)
      return 1;
   return n * factorial (n-1);
}

int main () {
   std::cout << "Factorial(5) =  " << factorial (5) << std::endl;
   return 0;
}
```

Figura 6.14: Ejemplo de función recursiva.

En este caso la versión no recursiva resulta más sencilla que la recursiva. Además, es más eficiente, pues evita llamadas a funciones con el ahorro consiguiente en iniciación de parámetros formales.

En el Tema 9 se estudia en profundidad las funciones recursivas.

6.9. Paso de parámetros de tipo estructura y string

El paso de parámetros de tipo estructura y string se realiza de la misma manera que el de un tipo básico. El programa de la Figura 6.15 ejemplifica el uso de estructuras como parámetros. El programa define una estructura llamada Complejo que sirve para representar números complejos. La función suma toma como parámetros por valor dos datos de tipo Complejo y devuelve su suma. Para almacenar la suma utiliza el parámetro formal x. La función muestra envía a la salida estándar el complejo que recibe como parámetro utilizando un formato de salida adecuado. Por último, la función main define e inicia las variables de tipo Complejo x e y. La variable z se inicia como la suma de las variables x e y. Tras ello, la función main invoca a la función muestra para enviar a la salida estándar el contenido de las tres variables.

El paso de una estructura como parámetro por valor de una función puede ser algo ineficiente pues implica la iniciación de la estructura—de todos sus campos. Cuantos más campos tenga la estructura más costoso resultará su paso por valor. Por este motivo muchas veces los parámetros de entrada de tipo estructura se pasan por variable. El costo en procesamiento del paso por variable es pequeño y siempre es el mismo para cualquier tipo de dato—en concreto, es independiente del tamaño de la estructura. Por ejemplo, la función muestra del programa de la Figura 6.15 podría escribirse así:

```
void muestra (const Complejo& c) {
   cout << "(" << c.real << "," << c.imaginario << ")" << endl;
```

```
#include <iostream>
using namespace std;

struct Complejo {
   double real, imaginaria;
};

Complejo suma (Complejo x, Complejo y) {
   x.real += y.real;
   x.imaginaria += y.imaginaria;
   return x;
}

void muestra (Complejo c) {
   cout << "(" << c.real << "," << c.imaginaria << ")" << endl;
}

int main () {
   Complejo x = { 1, 2 }, y = { 3, 3 };
   Complejo z = suma (x, y);
   muestra (z);
   muestra (x);
   muestra (y);
   return 0;
}
```

Figura 6.15: Ejemplo de parámetros de tipo estructura.

}

de esta forma el paso del parámetro c se realiza por variable, aunque la función muestra no modifique a c. El uso del calificativo **const** en la variable c impide a la función muestra la modificación de c. Puesto que c no puede ser modificado, cualquier persona que lea la signatura de la función muestra deduce que c es un parámetro de entrada de la función—y que se pasa por referencia por motivos de eficiencia.

Todo lo dicho en esta sección sobre estructuras es aplicable al tipo de dato string.

6.10. Paso de parámetros de tipo vector

El paso de parámetros de tipo vector se ilustra en el programa de la Figura 6.16. Un parámetro de tipo vector se especifica de la misma forma que se define una variable de tipo vector, salvo que no es necesario especificar su tamaño. Como a través de una variable de tipo vector no es posible saber cuántos elementos almacena el vector, se utiliza otro parámetro para recibir el número de elementos que almacena el vector. En la función cuadrado del programa de la Figura 6.16 el parámetro v recibe el vector y el parámetro tam el número de elementos de v.

Como se describió en el Tema 5, Sección 5.1.2, una variable de tipo vector almacena la dirección de memoria a partir de la cual se almacenan los elementos del vector. Por lo tanto, lo que recibe un parámetro formal de tipo vector es la dirección de memoria donde se almacena un vector—véase la Figura 6.17. Al recibir la dirección de memoria donde se almacena el vector, la función tiene total acceso a los elementos originales del vector y, por tanto, las modificaciones que realice en los elementos del vector serán visibles al terminar la ejecución de la función. Dicho de otro modo, el paso de parámetro de un vector siempre se realiza por referencia. Esta forma de pasar los vectores es muy eficiente, pues sólo se recibe una copia de la dirección de inicio del vector—los elementos del vector no se copian, se trabaja con los "elementos originales". Para invocar a una función que tiene un parámetro de tipo vector se utiliza como parámetro real una variable de tipo vector—realmente, hay que utilizar una expresión que proporcione una dirección de memoria que contenga elementos del mismo tipo que los del vector. Por ejemplo, en la función main del programa de la Figura 6.16 la función cuadrado se invoca mediante la sentencia: cuadrado (vec, TAM);

Hay funciones que reciben como parámetro de entrada un vector—los elementos del vector se leen o consultan, pero no se modifican. Un ejemplo de este tipo de función es la función muestra del programa de la Figura 6.16. En este caso resulta útil anteponer al parámetro formal de tipo vector el calificativo **const**. Con este calificativo se impide que la función modifique los elementos del vector. Además, el uso del calificativo **const** en la signatura de una función proporciona la información de que la función no modifica el contenido del vector—dicho de otro modo, que el vector es un parámetro de entrada de la función.

```
#include <iostream>
using namespace std;

void cuadrado (int v[], int tam) {
   for (int i = 0; i < tam; i++)
      v[i] *= v[i]; // equivale a v[i] = v[i]*v[i];
}

void muestra (const int v[], int tam) {
   for (int i = 0; i < tam; i++)
      cout << v[i] << ' ';
   cout << endl;
}

int main () {
   const int TAM = 3;
   int vec[TAM] = { 1, 2, 3 };
   cuadrado (vec, TAM);
   muestra (vec, TAM);
   return 0;
}
```

Figura 6.16: Ejemplo de parámetros de tipo vector.

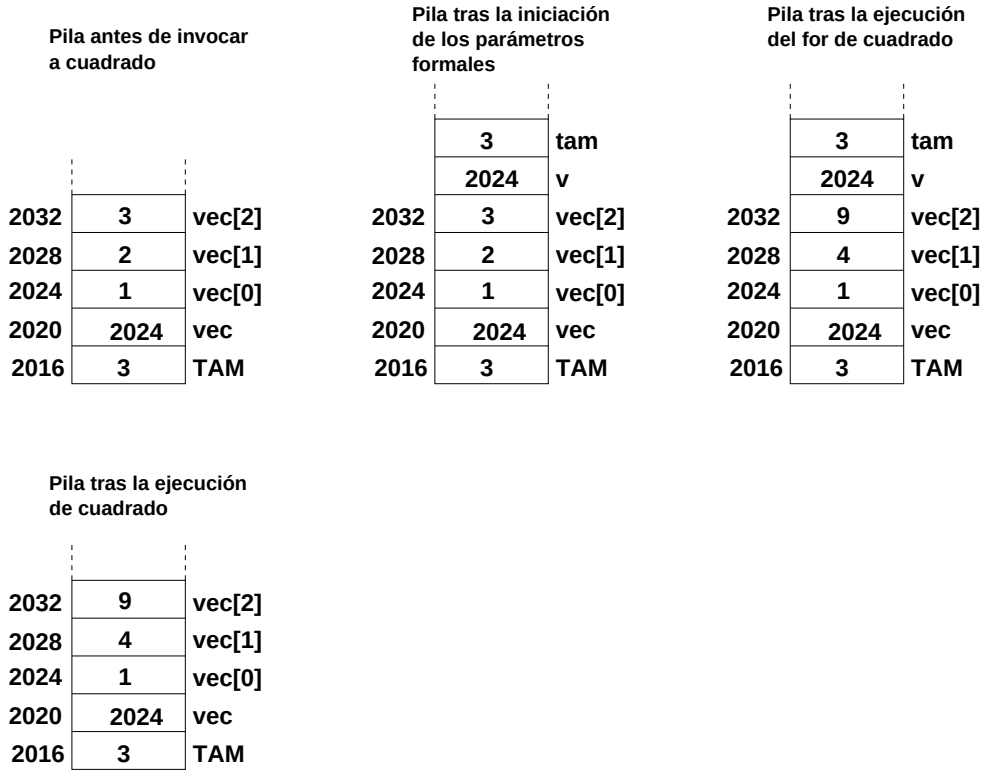

Figura 6.17: Estado de la pila en algunos momentos de la ejecución del programa de la Figura 6.16.

Una función no puede devolver un valor de tipo vector. Si se quiere que una función inicie o modifique un vector habrá que utilizar un parámetro de tipo vector—sin usar el calificativo **const**.

6.10.1. Paso de parámetros de tipo *array* multidimensional

Todo lo comentado en esta sección sobre vectores es también aplicable a *arrays* multidimensionales. El programa de la Figura 6.18 contiene dos funciones que reciben como parámetro un *array* bidimensional—una matriz. La primera función actualiza los elementos de la matriz a su cuadrado y la segunda función envía a la salida estándar el contenido de la matriz, situando cada fila en una línea distinta. Además de la propia matriz, ambas funciones reciben como parámetro el número de filas y de columnas de la matriz. Observe que en el parámetro formal de tipo *array*: **int** m[][NCOL], se ha especificado el número de columnas, pero no el número de filas. Es válido también especificar el número de filas, es decir, hacerlo así: **int** m[NFILAS][NCOL]. Si un parámetro de tipo *array* es de dimensión d, C++ obliga a especificar el tamaño de las últimas $d-1$ dimensiones al escribir el parámetro formal. Esto es debido a que, por la forma en que se implementan los *arrays*, se necesita dicho tamaño para poder calcular las direcciones de memoria donde se ubican los elementos del *array*. En el siguiente párrafo se describe cómo se implementan los *arrays*.

Los elementos de un *array* se almacenan en posiciones contiguas de memoria. Por ejemplo, dado un *array* bidimensional primero se almacenan la primera fila del *array*, después la segunda fila y así sucesivamente. La Figura 6.19 ilustra una ocupación de memoria para el *array* definido en la función main del programa de la Figura 6.18. Para calcular la dirección de inicio en memoria del elemento que ocupa la fila i, columna j, de un *array* bidimensional hay que realizar el siguiente cálculo: $dir\_inicio + (i * NCOL + j) * tam$, donde dir_inicio es la dirección de inicio en memoria del *array*, $NCOL$ es el número de columnas del *array* y tam es el tamaño en *bytes* de los elementos del *array*. Para realizar este cálculo no hace falta conocer el número de filas del array. Por ejemplo, en el *array* representado en la Figura 6.19, se puede calcular la dirección de inicio en memoria del elemento mat[1][1] como: $1000 + (1 * 2 + 1) * 4 = 1012$. Observe que en la función cuadrado del programa de la Figura 6.18 toda esta información está disponible: la dirección de inicio en memoria se almacena en el parámetro formal m, $NCOL$ se ha especificado en el parámetro formal y tam se deduce del tipo de los elementos del vector—el tipo también se especifica con el parámetro.

Un inconveniente de las funciones que toman parámetros de tipo *array* multidimensional es que son poco generales, en el sentido de que si un parámetro formal de tipo *array* tiene dimensión d entonces el parámetro formal fija las $d-1$ últimas dimensiones del *array*. Por ejemplo, las funciones cuadrado y muestra del programa de la Figura 6.18 sólo admiten como parámeto matrices de dos columnas—NCOL vale 2. Es decir, la función muestra sirve para enviar a la salida estándar matrices de dos columnas, pero no, por ejemplo, de tres o de cuatro columnas. Se puede solucionar este problema utilizando *arrays* unidimensiona-

```
#include <iostream>
using namespace std;

const int NFILAS = 3;
const int NCOL = 2;

void cuadrado (int m[][NCOL], int nfilas, int ncol) {
   for (int i = 0; i < nfilas; i++)
      for (int j = 0; j < ncol; j++)
         m[i][j] *= m[i][j]; // equivale a m[i][j] = m[i][j]*m[i][j];
}

void muestra (const int m[][NCOL], int nfilas, int ncol) {
   for (int i = 0; i < nfilas; i++) {
      for (int j = 0; j < ncol; j++)
         cout << m[i][j] << ' ';
      cout << '\n';
   }
}


int main () {
   int mat[NFILAS][NCOL] = { {1, 1}, {2, 2}, {3, 3} };
   cuadrado (mat, NFILAS, NCOL);
   muestra (mat, NFILAS, NCOL);
   return 0;
}
```

Figura 6.18: Ejemplo de parámetros de tipo array.

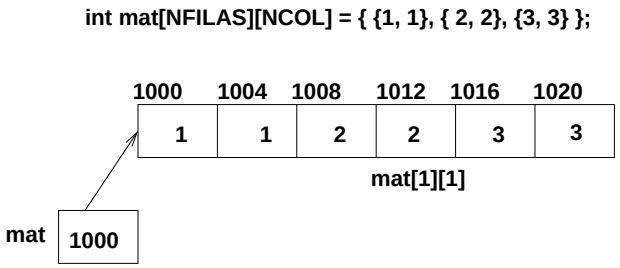

Figura 6.19: Disposición en memoria de un *array* bidimensional.

```
#include <iostream>

void cuadrado (int m[], int nfilas, int ncol) {
   for (int i = 0; i < nfilas; i++)
      for (int j = 0; j < ncol; j++)
         m[i*ncol + j] *= m[i*ncol+ j];
}

void muestra (const int m[], int nfilas, int ncol) {
   for (int i = 0; i < nfilas; i++) {
      for (int j = 0; j < ncol; j++)
         std::cout << m[i*ncol + j] << ' ';
      std::cout << '\n';
   }
}


int main () {
   const int NFILAS = 3;
   const int NCOL = 2;
   int mat[NFILAS*NCOL] = { 1, 1, 2, 2, 3, 3 };
   cuadrado (mat, NFILAS, NCOL);
   muestra (mat, NFILAS, NCOL);
   int mat2[2*4] = { 1, 1, 1, 1, 2, 2, 2, 2};
   muestra (mat2, 2, 4);
   return 0;
}
```

Figura 6.20: Alternativa al programa de la Figura 6.18 utilizando un vector.

les para albergar información multidimensional. El programa de la Figura 6.20 constituye una alternativa al programa de la Figura 6.18 en el que se utiliza un vector—*array unidimensional*—para almacenar una matriz bidimensional. El vector almacena los elementos de la matriz, al igual que se hace en los *arrays* bidimensionales, una fila detrás de otra. Las funciones utilizan la sintaxis: `m[i*ncol + j]` para acceder al elemento que, de manera lógica, ocupa la fila `i`, columna `j`, en la matriz almacenada en el vector. Las funciones `cuadrado` y `muestra` del programa de la Figura 6.20 trabajan con *arrays* unidimensionales que albergan matrices bidimensionales de cualquier dimensión. Por ejemplo, en el programa se invoca a la función `muestra` para enviar a la salida estándar tanto una matriz de dimensión 3x2 como una matriz de dimensión 2x4.

```
#include <iostream>
using namespace std;

double maximo (double x, double y) {
   return (x > y) ? x : y;
}

double maximo (const double v[], int tam) {
   double max = v[0];
   for (int i = 1; i < tam; i++)
      if (v[i] > max)
         max = v[i];
   return max;
}

int main () {
   const int TAM = 4;
   double v[TAM] = { 2.3, 7, 8.2, 4.0 };
   cout << "Máximo del vector: " << maximo (v, TAM) << endl;
   cout << "Máximo de dos valores: " << maximo (v[0], v[1]) << endl;
   return 0;
}
```

Figura 6.21: Ejemplo de función sobrecargada.

6.11. Funciones sobrecargadas

La sobrecarga de funciones es una característica de C++ que no está presente en algunos lenguajes de programación. Consiste en poder definir más de una función con el mismo nombre, siempre que los parámetros de las funciones difieran en número o tipo. El programa de la Figura 6.21 muestra un ejemplo de función sobrecargada. Existen dos funciones de nombre maximo, la primera calcula el máximo de dos valores y la segunda el máximo de los elementos de un vector. El compilador analiza los parámetros utilizados al invocar a la función maximo para determinar a cuál de las dos funciones se quiere llamar.

6.12. Parámetros por defecto

El uso de parámetros por defecto es otra característica propia de C++ que no está presente en algunos lenguajes de programación. Un parámetro por defecto es un parámetro que no tiene por qué ser especificado al invocar a una función. Si en la invocación de la función no se especifica un parámetro por defecto entonces se le asigna un valor por defecto al parámetro. Por ejemplo, dentro del programa de la Figura 6.15, que trabaja con números complejos,

se podría haber incluido una función como la siguiente para establecer los valores de una variable complejo:

```
void establece (Complejo& c, double real = 0, double imaginaria = 0) {
   c.real = real;
   c.imaginaria = imaginaria;
}
```

Un parámetro por defecto se especifica utilizando una asignación en el parámetro formal. El valor asignado es el valor que toma por defecto el parámetro en caso de no ser especificado en la invocación. Por ejemplo, dada una variable c de tipo Complejo la llamada establece (c, 1, 2); hace que c valga (1,2). La llamada establece (c, 1); hace que c valga (1,0) y la llamada establece (c); hace que c valga (0, 0).

Los parámetros por defecto deben aparecer al final de la lista de parámetros formales de la función.

Como ejercicio, integre la función establece dentro del programa de la Figura 6.15.

Se puede sobrecargar una función para conseguir el mismo efecto que se logra mediante un parámetro por defecto, pero utilizando una función sobrecargada es necesario escribir más código. Por ejemplo, para conseguir el mismo efecto que la función establece utilizando una función sobrecargada habría que escribir lo siguiente:

```
void establece (Complejo& c, double real, double imaginaria) {
   c.real = real;
   c.imaginaria = imaginaria;
}
void establece (Complejo& c, double real)  {
   c.real = real;
   c.imaginaria = 0;
}
void establece (Complejo& c)  { c.real = c.imaginaria = 0; }
```

Si en un fichero aparece tanto la definición como una declaración de una función con parámetros por defecto, entonces los valores por defecto de los parámetros por defecto sólo se especifican en la declaración de la función. Por ejemplo:

```
void establece (Complejo& c, double real = 0, double imaginaria = 0);
...
void establece (Complejo& c, double real, double imaginaria) {
   c.real = real;
   c.imaginaria = imaginaria;
}
```

6.13. Especificación de la interfaz de una función

La especificación de la interfaz de una función consiste en una descripción de lo que hace la función a partir de su prototipo. Una buena especificación de una función permite que un programador pueda comprender su funcionalidad simplemente leyendo el prototipo y la especificación de la función, sin necesidad de leer ni comprender su código. Esto facilita enormemente el uso de funciones que han sido desarrolladas por otros programadores o incluso por nosotros mismos una vez que ha transcurrido un tiempo desde que escribimos la función.

Es bueno acostumbrarse a especificar de una forma rigurosa la interfaz de las funciones desarrolladas. En esta sección se propone una especificación basada en la descripción de las siguientes características de una función:

- **Descripción**: Indica lo que hace la función.
- **Parámetros de entrada**: Describe los parámetros de entrada de la función.
- **Parámetros de salida**: Describe los parámetros de salida de la función.
- **Parámetros de entrada y salida**: Describe los parámetros de entrada y salida de la función.
- **Precondiciones**: Indica ciertas condiciones que deben verificar los parámetros. Si no se verifican estas condiciones el comportamiento de la función es indefinido.
- **Valor de retorno**: Indica el significado del valor de retorno de la función.

Por ejemplo, una especificación de la interfaz de la segunda función maximo del programa de la Figura 6.21 es la siguiente:

```
/* Descripción: Calcula el máximo de los elementos de un vector de datos
                de tipo double
 * Parámetros de entrada:
   - v: el vector
   - tam: número de elementos del vector
 * Precondiciones: el vector debe tener al menos un elemento
 * Valor de retorno: el máximo de los elementos del vector
 */
```

Observe que en las precondiciones se especifica que el vector debe tener al menos un elemento, pues el código accede siempre al elemento de índice cero del vector sin comprobar que el tamaño del vector es mayor que cero. En una implementación alternativa la función podría comprobar si el vector no tiene elementos para, en ese caso, devolver un valor especial. Esto se refleja en la siguiente especificación:

```
/* Descripción: Calcula el máximo de los elementos de un vector de datos
                de tipo double mayores o iguales que cero
 * Parámetros de entrada:
   - v: el vector
   - tam: número de elementos del vector
 * Precondiciones: los elementos del vector son mayores o iguales que cero
 * Valor de retorno: el máximo de los elementos del vector o -1 si el vector
                     no contiene elementos
 */
```

Observe que en esta nueva especificación no existe la precondición de que el vector tenga al menos un elemento. La implementación de la función debe, por tanto, comprobar si el vector de entrada está vacío. Como se establece la precondición de que los elementos del vector son mayores o iguales que cero se devuelve un valor negativo, -1, para indicar que no hay máximo. Como ejercicio puede implementar una función que satisfaga esta especificación y una función main que pruebe su funcionamiento.

6.14. Ejercicios

1. Realiza una función que tome como parámetro un número real y devuelva su valor absoluto.

2. Realiza una función que tome como parámetros dos cadenas de caracteres y devuelva la cadena más larga.

3. Realice una función que tome como parámetros los extremos de un intervalo y un valor y devuelva un valor lógico que indique si el valor pertenece al intervalo cerrado formado por los extremos. Por ejemplo, si los extremos son a y b, con $a <= b$, y el valor es x, entonces debe devolver si $x \in [a, b]$.

4. Realice una función que, dado un carácter, devuelva si es un dígito.

5. Realice una función que, dada una cadena de caracteres, devuelva cuántos dígitos contiene. Sugerencia: use la función del ejercicio anterior.

6. Realice una función que determine si un entero positivo es primo.

7. Implemente una función que calcule la suma de los dígitos de un número entero positivo. Por ejemplo, dado 3452 la suma de sus dígitos es $3 + 4 + 5 + 2 = 14$.

8. Realiza una función que tome como parámetros dos cadenas de caracteres y devuelva la cadena más larga y la más corta.

9. Escribe una función que intercambie el valor de sus dos parámetros.

Tabla 6.1: Asociación entre resto de dividir el número del DNI entre 23 y su letra asociada.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
T	R	W	A	G	M	Y	F	P	D	X	B	N	J	Z	S	Q	V	H	L	C	K	E

10. Realice una función que tome como parámetros un vector de números y devuelva la suma de sus elementos.

11. Realice una función que tome como parámetros un vector y su tamaño y diga si el vector está ordenado crecientemente. Sugerencia: compruebe que para todas las posiciones del vector, salvo para la 0, el elemento del vector es mayor o igual que el elemento que le precede en el vector.

12. Realiza una función que tome como parámetros un vector de números y su tamaño y cambie el signo de los elementos del vector.

13. El DNI incluye una letra con el objeto de detectar errores al ser escrito. Dado un número de DNI la letra correspondiente se obtiene calculando el resto de dividir dicho número entre 23. Una vez obtenido el resto la Tabla 6.1 indica la letra correspondiente. Realice una función que dado un número de DNI devuelva su letra asociada y un programa que solicite un DNI hasta que su letra asociada sea correcta.

14. Realiza una función que tome como parámetros un vector de enteros y su tamaño y devuelva un vector con los elementos impares del vector recibido.

15. Realice una función con las siguientes características:

    - Entradas: un entero positivo n.
    - Salidas: un vector de enteros con los primeros n números perfectos. Un número perfecto es un número natural que es igual a la suma de sus divisores positivos, sin incluirse él mismo. Así, 6 es un número perfecto, porque sus divisores positivos son 1, 2 y 3; y $6 = 1 + 2 + 3$. El siguiente número perfecto es el 28.

16. Realice un módulo que implemente un menú de opciones. El módulo debe mostrar en la salida estándar una serie de opciones precedidas por un número. Por ejemplo:

    ```
    0 - Introducir palabra
    1 - Adivinar palabra
    2 - Salir
    Introduzca la opcion elegida:
    ```

    El módulo debe leer de la entrada estándar la opción elegida por el usuario, hasta que se lea un valor de opción elegida correcto, que debe ser devuelto como resultado de ejecución del módulo.

17. Generalice el módulo del ejercicio anterior de forma que las opciones del menú se reciban en un vector de cadenas. Cada cadena del vector representa una de las opciones a elegir.

18. Una matriz nxm se dice que es simétrica si $n = m$ y $a_{ij} = a_{ji} \forall i, j : 1 \leq i \leq n, 1 \leq j \leq n$. Desarrollar una función que determine si una matriz es simétrica o no.

19. Realice un módulo que calcule el producto de dos matrices cuadradas de 3x3.

20. Realice una función que dada una matriz y un número de fila de la matriz devuelva el mínimo de los elementos almacenados en dicha fila.

21. Realice una función con las siguientes características:

    - Entradas: un vector de enteros v y su tamaño.
    - Salidas: un vector de enteros *sal* y su tamaño. El vector *sal* debe contener los elementos de v pero sin contener repetidos. Por ejemplo, si v contiene los elementos $\{2,\ 4,\ 2,\ 7,\ 7,\ 2\}$ entonces *sal* debe contener los elementos $\{2,\ 4,\ 7\}$.

    Realice también una función main que permita probar el funcionamiento de la función diseñada. Sugerencia: antes de insertar un elemento en el vector de salida verifique que no se encuentra ya en dicho vector.

22. Dado un vector de *n* números enteros, construya un programa que calcule el número de veces que aparece cada entero en el vector.

23. Realice una función que dadas dos cadenas de caracteres, s1 y s2, devuelva si la primera cadena es menor lexicográficamente que la segunda. Esto se puede realizar trivialmente con la expresión s1 $<$ s2. Realice una implementación alternativa en la que vaya comparando los caracteres de las dos cadenas. Tenga en cuenta que una cadena s1 es menor lexicográficamente que otra cadena s2 si se verifica que:

    - En el primer carácter en que difieren las dos cadenas se verifica que el carácter de s1 es menor lexicográficamente que el carácter de s2. Por ejemplo, `abc` es menor que `abd` porque en el primer carácter en que difieren se verifica que `'c'` $<$ `'v'`.
    - La longitud de la cadena s1 es inferior a la longitud de la cadena s2 y todos los caracteres existentes de s1 coinciden con los caracteres que ocupan la misma posición en s2. Por ejemplo, `cama` es menor que `camastro`.
    - En otro caso, s1 no es menor que s2. Por ejemplo, `abc` no es menor que `abb`, ni que `ab`, ni que `abc`.

24. El operador == permite comparar si dos strings son iguales, pero diferenciando entre mayúsculas y minúsculas. Por ejemplo ''Agosto'' == ''Agosto'', pero ''Agosto'' != ''agosto''.

Desarrolle una función que compare dos strings y devuelva si son iguales independientemente de que los caracteres se encuentren en mayúsculas o en minúsculas. Sugerencia: en la implementación puede utilizar la función toupper de la biblioteca estándar—tendrá que utilizar el include **#include** <cctype>—que devuelve el carácter que toma como parámetro de entrada a su valor en mayúsculas. Si el parámetro recibido como entrada no es una letra minúscula del alfabeto inglés entonces devuelve el mismo carácter que recibió como parámetro. Por ejemplo, toupper ('a') devuelve 'A', toupper ('B') devuelve 'B' y toupper ('*') devuelve '*'.

25. En este ejercicio se va a implementar el algoritmo de selección, que sirve para ordenar de forma creciente una secuencia de elementos, en nuestro caso, almacenados en un vector. Dado un vector de n elementos, por ejemplo: $\{5, 9, 2, 1, 4\}$, el algoritmo selecciona el mínimo de los elementos del vector y lo intercambia por el elemento que ocupa la primera posición del vector, en el ejemplo esto produce el vector $\{1, 9, 2, 5, 4\}$. A continuación selecciona el mínimo de los elementos entre las posiciones 1 y $n-1$ y lo intercambia por el elemento en la posición 1. En el ejemplo, esto produce el vector $\{1, 2, 9, 5, 4\}$. En general, en la iteración i se selecciona el mínimo entre las posiciones $[i, n-1]$ y se intercambia con el valor en la posición i, tras ello se verifica que las posiciones $[0, i]$ del vector contienen los $i+1$ elementos menores del vector ordenados. La función de ordenación por selección es la siguiente:

```
void seleccion (int v[], int tamv) {
   for (int i = 0; i < tamv - 1; ++i) {
      int posmin = minimo (v, i, tamv -1);
      if (posmin != i)
         intercambia (v[i], v[posmin]);
   }
}
```

donde la función intercambia es como la especificada en un ejercicio anterior y la función minimo tiene la siguiente especificación:

```
/* Descripción: Calcula la posición del menor elemento del
 *                  vector entre las posiciones [i,f]
 * Parametros de entrada:
 *  - v: el vector
 *  - i: posición inicial
 *  - f: posición final
 * Precondiciones: i <= f
 * Valor de retorno: la posición del menor elemento del vector
 *                   entre las posiciones [i,f]
 */
```

Implemente las funciones intercambia, minimo y seleccion, y una función main que permita probar la función de ordenación.

26. Una aplicación almacena la siguiente información sobre cada persona: nombre, número del DNI y edad. Las personas se almacenan en un vector ordenado según el DNI. Realice las siguientes funciones:

    - Función que, dado un número de DNI *dni* y un vector de personas, devuelva la primera posición del vector que contiene a una persona cuyo número de DNI es igual o superior a *dni* o el número de elementos del vector si no hay tal persona en el vector.
    - Función que, dado un vector de personas y una posición p del vector, desplace una posición a la derecha a todos los elementos del vector cuya posición sea igual o mayor que p.
    - Función que, dado un vector de personas y una posición p del vector, desplace una posición a la izquierda a todos los elementos del vector cuya posición sea mayor que p.
    - Función que, dado un vector de personas, muestre en la salida estándar su contenido.
    - Función que lea de la entrada estándar, y devuelva, los datos de una persona. El rango de edades admitido es: [0,150].

    Realice un programa principal que permita mediante un menú gestionar un vector ordenado de personas. El programa debe permitir:

    - Introducir una nueva persona. Sugerencia: utilice las dos primeras funciones. Dada una nueva persona utilice la primera función para comprobar si ya está en el vector. En caso de que ya esté rechace su introducción. En el caso de que no esté cree un hueco en el vector para ella—con la segunda función—e insértela en la posición correspondiente.
    - Eliminar una persona. Sugerencia: utilice la primera función para localizar su posición en el vector—si es que está—y la tercera función para suprimirla.
    - Mostrar el contenido del vector.

27. Un año es bisiesto si es divisible por 4, excepto el último de cada siglo (aquel divisible por 100), salvo que este último sea divisible por 400. De este modo 2004, 2008 o 2012 son bisiestos, pero no lo son 2100, 2200 o 2300, y sí lo es 2400. Realice una función que dado un año devuelva si es bisiesto.

28. Defina una estructura fecha que almacene el día, mes y año como enteros. Realice funciones que permitan:

    - La lectura de una fecha correcta de la entrada estándar.
    - El envío a la salida estándar de una fecha con el formato día/mes/año.

- La comparación de dos fechas, indicando si la primera es anterior a la segunda.
- El incremento en un año de una fecha. Tenga en cuenta los años bisiestos: el 29/2/2008 más un año es el 28/2/2009.

Realice también un programa principal que permita probar el funcionamiento de las funciones.

29. Una librería mantiene información de sus libros, clientes y ventas. Por cada libro almacena un código, su título y su precio. Por cada cliente almacena un código y su nombre. Por cada venta almacena el código del libro y el código del cliente implicados en la venta—se supone que en una venta un cliente compra un único libro. Se utiliza un vector para almacenar el conjunto de libros, otro vector para almacenar a los clientes y otro vector más para almacenar las ventas. Dadas estas estructuras de datos realice funciones que permitan:

    a) Dado un código de libro obtener cuántos libros con ese código se han vendido.

    b) Dado un código de libro, obtener su posición en el vector de libros.

    c) Obtener el título del libro más vendido.

    d) Obtener el título del libro que más dinero ha recaudado.

    e) Dado un código de cliente obtener un vector con los títulos de los libros que ha comprado.

    f) Obtener el nombre del cliente que ha comprado más libros.

    g) Obtener el nombre del cliente que ha gastado más dinero.

    Realice un programa principal que permita probar las últimas cinco funciones.

30. Realice un programa que compruebe si es válido el relleno de un Sudoku.

    Análisis del problema: El objetivo del Sudoku es rellenar una matriz de 9x9 celdas (81 celdas) dividida en submatrices de 3x3 (también llamadas "cajas") con las cifras del 1 al 9 partiendo de algunos números ya dispuestos en algunas de las celdas. No se debe repetir ninguna cifra en una misma fila, columna o caja.

    Un ejemplo de relleno válido de un Sudoku es el siguiente:

1	2	3	4	5	6	7	8	9
4	5	6	7	8	9	1	2	3
7	8	9	1	2	3	4	5	6
2	1	4	3	6	5	8	9	7
3	6	5	8	9	7	2	1	4
8	9	7	2	1	4	3	6	5
5	3	1	6	4	2	9	7	8
6	4	2	9	7	8	5	3	1
9	7	8	5	3	1	6	4	2

Para resolver este problema realice las siguientes funciones:

a) Función que dado un Sudoku y un número de fila del Sudoku devuelva un valor de tipo lógico indicando si no se repiten dos dígitos en la fila.

b) Función que dado un Sudoku y un número de columna del Sudoku devuelva un valor de tipo lógico indicando si no se repiten dos dígitos en la columna.

c) Función que dado un Sudoku y la primera fila y columna de una caja del Sudoku devuelva un valor de tipo lógico indicando si no se repiten dos dígitos en la caja.

d) Función main que lea un relleno de un Sudoku en una matriz y que utilice las funciones previas para determinar si el relleno es válido. Se debe mostrar en la salida estándar la información sobre si el relleno es válido o no.

31. Realice una función con la siguiente especificación:

- Entradas: una matriz de caracteres representando una sopa de letras—y su tamaño—y una palabra.
- Salidas: La posición de inicio en la matriz de la primera ocurrencia horizontal de la palabra. En caso de que la palabra no aparezca horizontalmente en la sopa de letras se debe devolver un valor adecuado.

Por ejemplo, dada la matriz—sopa de letras—que aparece más abajo y la palabra MARIO, la función debe devolver la fila 4, columna 2.

A	C	F	L	U	I	M	A	R
B	M	T	U	M	D	I	F	G
J	Z	U	I	O	L	S	C	I
E	Z	S	S	B	A	O	L	S
S	T	**M**	**A**	**R**	**I**	**O**	G	A
I	E	R	R	A	F	I	J	T
J	Z	U	I	O	L	S	C	I
S	A	L	B	E	R	T	O	S
A	M	I	N	O	T	E	N	I

32. Se va a trabajar en un programa de codificación sencilla de cadenas de caracteres. Una *codificación* es una aplicación biyectiva del conjunto de los caracteres del alfabeto en el propio conjunto de los caracteres del alfabeto. Un ejemplo de codificación es la siguiente aplicación biyectiva:

a	b	c	d	e	f	g	h	i	j	k	l	m	n	ñ	o	p	q	r	s	t	u	v	w	x	y	z
b	c	d	e	f	g	h	i	j	k	l	m	n	ñ	o	p	q	r	s	t	u	v	w	x	y	z	a

Dada esta codificación, una cadena como "hola" se codifica como "ipmb".

Piense y describa una forma de representar en un programa C++ una codificación. Realice las siguientes funciones.

a) Realice una función que lea de la entrada estándar una codificación y devuelva la codificación leída.

b) Función que tome como entrada una codificación y calcule si la codificación es correcta. Una codificación es correcta si representa una aplicación biyectiva de los caracteres del alfabeto. Nota: una forma sencilla de comprobar si una codificación es correcta consiste en comprobar que tanto en el dominio como en el codominio aparece cada carácter del alfabeto una única vez. Es decir, hay que comprobar que el carácter *a* aparece una única vez en el dominio, que el carácter *b* aparece una única vez en el dominio y así sucesivamente hasta el carácter *z*. Después hay que hacer lo mismo con el codominio.

c) Realice una función que reciba como parámetros de entrada una codificación y como parámetro de entrada y salida una cadena de caracteres. La función debe utilizar la codificación para codificar la cadena de caracteres. Por ejemplo, dada la codificación inicial y la cadena "si", debe cambiar la cadena a "tj".

d) Realice una función que tome como parámetro una codificación. La función debe leer líneas de la entrada estándar y escribirlas en la salida estándar codificadas. Sugerencia: utilice la función realizada en el ejercicio anterior. Por ejemplo, si se leen las siguientes líneas:

```
hola
querido
luis
```

la función debe escribir en la salida estándar las siguientes líneas:

```
ipmb
rvfsjep
mvjt
```

e) Realice una función main que permita probar las funciones previas. La función debe leer una codificación de la entrada estándar o utilizar una por defecto. En caso de que la codificación a utilizar sea correcta, debe leer líneas de la entrada estándar y escribirlas codificadas en la salida estándar.

Tema 7
Punteros

En este tema se estudia el tipo de dato puntero. Los punteros sirven para expresar direcciones de memoria y permiten modificar el contenido de la memoria a la que apuntan. El tipo de dato puntero se utiliza con profusión en C++, por ejemplo, para implementar el paso de parámetros por referencia, para trabajar con la memoria dinámica o para almacenar un vector de datos que se tratan polimórficamente.

7.1. El tipo puntero

Un puntero a un dato de tipo `T` es una dirección de memoria donde se almacena un dato de tipo `T`. La sintaxis de C++ para definir una variable `p` de tipo puntero a un dato de tipo `T` es la siguiente:

```
T* p;
```

Existen dos operadores especialmente útiles cuando se trabaja con punteros: los operadores `&` y `*`, que se describen a continuación:

- El *operador* `&` u *operador dirección de*: es un operador unitario que se aplica a una variable—o a objetos como un elemento de un vector o un campo de una estructura que pueden utilizarse en los mismos lugares en los que se utiliza una variable. El resultado de la expresión `&x`, donde `x` es una variable, es la dirección de memoria donde se almacena `x`.
- El *operador* `*` u *operador de indirección o de desreferencia*: es un operador unitario que se aplica a un puntero. Dada una expresión de tipo puntero `p`, la expresión `*p` permite trabajar con la variable u objeto al que apunta `p`.

El programa de la Figura 7.1 ejemplifica el uso de punteros. En este programa se define la variable `p` de tipo puntero a un dato de tipo **float**. Después se utiliza el operador `&` para asignar a `p` sucesivamente la dirección del campo `x` de la estructura `punto`, la dirección del

```
#include <iostream>

struct Punto {
   float x, y;
};

int main () {
   float* p;
   float v[3] = {1.2f, 19f, 6f};
   float f = 4;
   Punto punto = {2.6f, 6f};
   p = &punto.x;
   p = &v[1];
   p = &f;
   std::cout << "La variable f vale: " << f << ' ' << *p << std::endl;
   *p = 5.5f;
   std::cout << "La variable f vale: " << f << std::endl;
   return 0;
}
```

Figura 7.1: Ejemplo de uso de punteros.

segundo elemento del vector v y la dirección de la variable f. Por último, se utiliza el operador * para modificar indirectamente—a través del puntero p—el valor almacenado en la variable f.

La Figura 7.2 refleja el estado de la pila tras la ejecución de las distintas asignaciones del programa de la Figura 7.1. Observe que la variable p almacena un valor indefinido hasta que se le asigna un valor. Un error muy común cuando se trabaja con punteros consiste en desreferenciar un puntero no iniciado, por ejemplo:

```
float* p;
*p = 5.5f;
```

Este código intenta almacenar el valor 5.5 en la zona de memoria donde apunta p. El problema es que no se ha establecido dónde apunta p. Dependiendo del lugar donde apunte p se producirán unos resultados u otros, en cualquier caso indeseables. Si p apunta a una zona de memoria modificable del programa ésta se modificará, perdiéndose de manera inadvertida el valor anterior almacenado en dicha zona; si p apunta a una zona de memoria no modificable del programa o fuera de una zona de acceso permitida al programa entonces el programa terminará de manera abrupta.

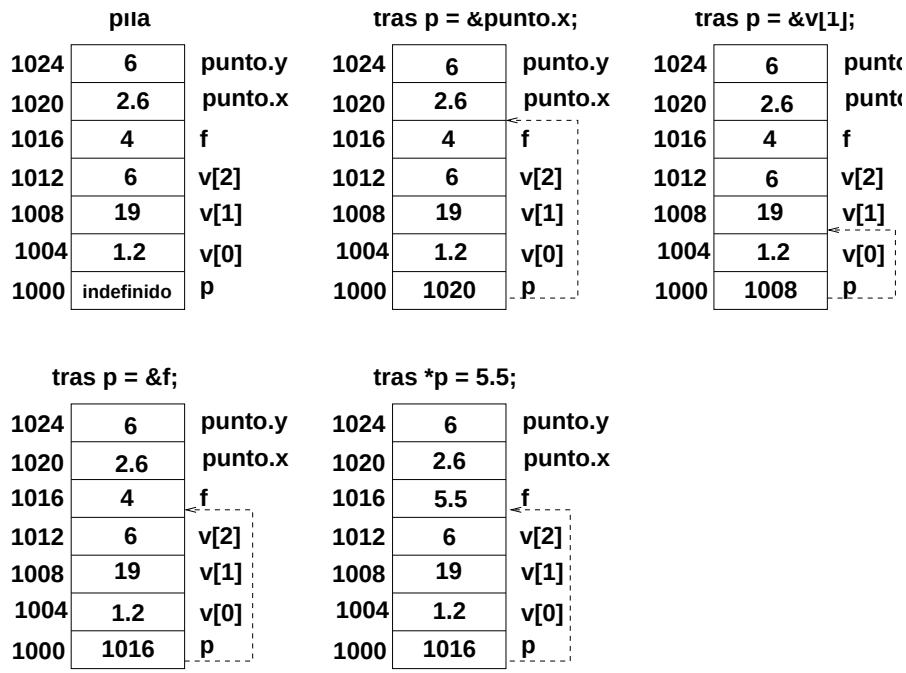

Figura 7.2: Estado de la memoria durante algunos momentos de la ejecución del programa de la Figura 7.1.

7.1.1. Punteros a estructuras

El programa de la Figura 7.3 ilustra la sintaxis necesaria para el acceso indirecto—desreferencia—a un campo de una estructura mediante un puntero a dicha estructura. El programa utiliza las dos posibilidades sintácticas existentes:

- (∗puntero).campo: con (∗puntero) se desreferencia y se tiene acceso a la estructura, y con .campo se selecciona el campo campo de la estructura. Los paréntesis son necesarios porque el operador . tiene mayor precedencia que el operador ∗. Si no se utilizaran paréntesis la expresión equivaldría a ∗(puntero.campo), es decir, se intentaría desreferenciar puntero.campo—para ello puntero debería ser una estructura y campo un campo de la estructura de tipo puntero.
- puntero−>campo: constituye una alternativa a la sintaxis previa, resultando más legible. Utiliza el operador −> que tiene como operandos un puntero a una estructura y el nombre de un campo de la estructura apuntada.

7.1.2. Puntero nulo

Es posible asignarle el valor cero a un puntero, por ejemplo:

```
#include <iostream>

struct Punto {
   float x, y;
};

int main () {
   Punto punto = {2.6f, 6f};
   Punto* p = &punto;
   std::cout << (*p).x << ' ' << p->y << '\n';
   return 0;
}
```

Figura 7.3: Ejemplo de uso de punteros a estructuras.

```
int* p = 0;
```

En la iniciación previa el literal 0 es un entero que se convierte implícitamente a una dirección de memoria. Lo interesante es que la conversión implícita produce una dirección de memoria en la que no puede estar almacenado ningún objeto. Por lo tanto, en C++ se utiliza el valor 0 para indicar una dirección que no apunta a nada—es decir, a ningún objeto. Por ejemplo, en el programa de la Figura 7.4 la función primerImpar recibe como parámetros de entrada un vector de enteros y su longitud y devuelve como resultado la dirección del primer elemento impar del vector. ¿Qué se puede devolver si el vector recibido como parámetro no contiene valores impares? La solución adoptada es devolver la dirección cero, a veces llamada puntero nulo. La función main comprueba si el valor devuelto por la función primerImpar es la dirección cero para determinar si el vector contiene o no algún valor impar.

Como curiosidad se indica que un programador experimentado en C++ suele comprobar si un puntero es distinto de cero utilizando el código: **if** (p), en lugar de utilizar el código: **if** (p != 0) empleado en el programa.

7.2. Paso de parámetros por referencia mediante punteros

En el Tema 6, Sección 6.5, se estudió una manera de realizar el paso por variable o referencia en C++. En dicha sección se utilizaban las referencias.

En esta sección se analiza cómo se puede realizar el paso por referencia utilizando punteros. El lenguaje C carece del tipo referencia, por lo que en C los punteros constituyen el único modo de realizar el paso por referencia.

El programa de la Figura 7.5 contiene la función maxmin que devuelve el máximo y el mínimo de los valores recibidos en los parámetros formales x e y utilizando paso por refe-

```
#include <iostream>
using namespace std;

int* primerImpar (int v[], int tam) {
   for (int i = 0; i < tam; i++)
      if (v[i] %2 == 1)
         return &v[i];
   return 0;
}

int main () {
   const int TAM = 5;
   int vec[TAM] = {6, 2, 3, 4, 5 };
   int* p = primerImpar (vec, TAM);
   if (p != 0)
      cout << "El primer elemento impar del vector es " << *p << endl;
   else
      cout << "El vector no tiene valores impares\n";
   return 0;
}
```

Figura 7.4: Ejemplo de uso de un puntero nulo.

rencia mediante punteros. Para ello utiliza otros dos parámetros formales: max y min, donde se reciben las direcciones de memoria—punteros—del lugar en el que hay que guardar los resultados. En el cuerpo de la función maxmin se utiliza la sintaxis de desreferencia para guardar los resultados en las zonas de memoria cuyas direcciones han sido recibidas como parámetros. Observe también la llamada a la función en main: maxmin (a, b, &maximo, &minimo), se utiliza el operador & para pasar como parámetro real las direcciones de memoria de las variables maximo y minimo, para que se guarde en la memoria asociada a estas variables el resultado de la ejecución de la función maxmin.

La Figura 7.6 ilustra el estado de la pila durante una ejecución del programa de la Figura 7.5. Observe cómo la función maxmin utiliza los punteros recibidos como parámetros para almacenar los resultados en las zonas de memoria asociadas a las variables implicadas en el paso por variable. De esta forma, cuando termina la ejecución de la función maxmin los datos calculados están disponibles en las variables maximo y minimo.

7.3. Aritmética de punteros y vectores

Dada una expresión p de tipo puntero a un dato de tipo T y una expresión e de tipo entero, la expresión p+e produce un valor de tipo puntero a un dato de tipo T y de valor p + e∗**sizeof**(T),

```
#include <iostream>
using namespace std;

void maxmin (double x, double y, double* max, double* min) {
   *max = (x > y) ? x : y;
   *min = (x > y) ? y : x;
}

int main () {
  double a, b, maximo = 0, minimo = 0;
  cout << "Introduzca dos números: ";
  cin >> a >> b;
  maxmin (a, b, &maximo, &minimo);
  cout << "Máximo: " << maximo << " Mínimo: " << minimo << endl;
  return 0;
}
```

Figura 7.5: Ejemplo de paso por referencia empleando punteros.

donde **sizeof**(T) es una expresión que devuelve el tamaño en bytes ocupado por un dato de tipo T. Es decir, si p vale 1000 y apunta a un entero, entonces p+2 vale 1008—suponiendo que un entero ocupa cuatro bytes, o sea, que **sizeof**(**int**) vale 4. Como los elementos de un vector ocupan posiciones contiguas de un vector, se puede utilizar un puntero para recorrer sus elementos, como ilustra el programa de la Figura 7.7.

Si p es un puntero que apunta al primer elemento de un vector v—es decir, p = &v[0]—, entonces la expresión p+i, donde i es una expresión entera, equivale a &v[i]. Por lo tanto, *(p+i) equivale a v[i]. En la expresión *(p+i) los paréntesis son necesarios, puesto que el operador desreferencia, *, tiene mayor precedencia que el operador suma, +, lo que implica que la expresión *p+i equivale a la expresión (*p)+i, es decir, significa sumarle i al elemento al que apunta p—como en el ejemplo p vale &v[0], *p+i equivale a v[0]+i.

Los punteros y los vectores están tan íntimamente relacionados en C++ que es posible aplicar el operador [] a un puntero. Dado un puntero p la expresión p[i], donde i es una expresión entera, equivale a la expresión *(p+i). El programa de la Figura 7.8 equivale al programa de la Figura 7.7, pero utiliza el operador []. Observe también que se ha cambiado la iniciación del puntero, en lugar de aparecer **int*** p = &v[0]; aparece **int*** p = v;. La expresión **int*** p = v; es válida sintácticamente, realiza una conversión implícita de la variable v de tipo vector de enteros al tipo puntero a entero, y almacena la dirección de inicio del vector v en p.

Los punteros también pueden utilizarse para especificar los parámetros de tipo vector de una función de una manera alternativa a la estudiada en el Tema 6, Sección 6.10. En el programa de la Figura 7.9 la función dobla utiliza la sintaxis que se ha utilizado hasta

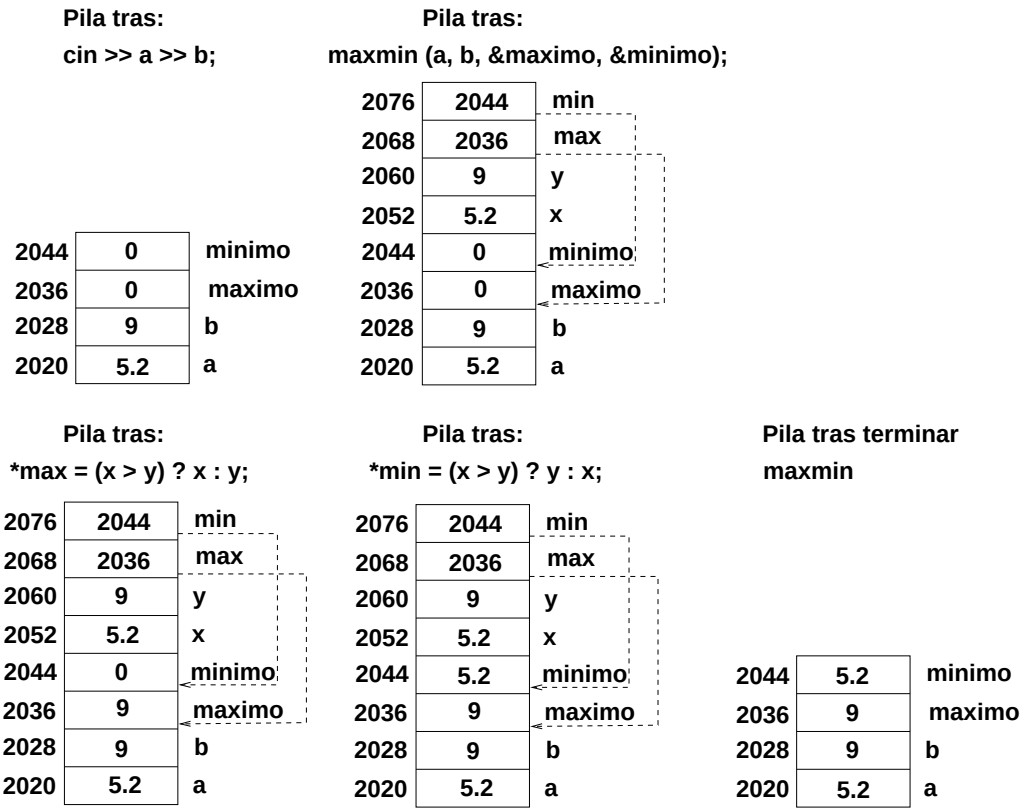

Figura 7.6: Estado de la pila durante algunos momentos de una ejecución del programa de la Figura 7.5.

```
#include <iostream>

int main () {
   int v[3] = {3, 2, 1};
   int* p = &v[0];
   for (int i = 0; i < 3; i++)
      std::cout << *(p + i) << '\n';
  return 0;
}
```

Figura 7.7: Ejemplo de recorrido de un vector utilizando punteros. El programa muestra el contenido del vector.

```
#include <iostream>

int main () {
   int v[3] = {3, 2, 1};
   int* p = v;
   for (int i = 0; i < 3; i++)
      std::cout << p[i] << '\n';
  return 0;
}
```

Figura 7.8: Programa alternativo al de la Figura 7.7.

ahora. La función doblaPunteros utiliza la sintaxis alternativa que utiliza punteros. Las dos funciones realizan el mismo cálculo—de hecho, tienen el mismo cuerpo—obteniendo una copia del vector recibido como primer parámetro pero con sus elementos multiplicados por dos. Observe el tipo del primer parámetro de la función doblaPunteros: **const int*** v, el uso del calificativo **const** impide a la función la modificación de los elementos cuya dirección comienza en v. La utilización de este calificativo resulta útil porque viendo la signatura de la función se deduce que el vector v es un parámetro de entrada de la función. Intente escribir dentro de cualquiera de las dos funciones una instrucción del tipo v[0] = 2; y obtendrá un error al compilar el programa. Por último, observe la última invocación en main a la función doblaPuntero: ¿cuál cree que será el resultado de dicha invocación?

7.3.1. Resta de punteros

También es posible restar dos expresiones de tipo puntero, pero la resta sólo tiene sentido si las dos expresiones apuntan a elementos del mismo vector. En dicho caso, la resta devuelve el número de posiciones que existen entre los dos elementos del vector a los que apuntan los punteros. Por ejemplo, en el programa de la Figura 7.4, se puede modificar la función main para que muestre la posición que ocupa en el vector el primer elemento impar de la siguiente manera:

```
int main () {
   const int TAM = 5;
   int vec[TAM] = {6, 2, 3, 4, 5 };
   int* p = primerImpar (vec, TAM);
   if (p != 0) {
      cout << "El primer elemento impar del vector es " << *p << endl;
      cout << "Ocupa la posición: " << p - &vec[0] << endl;
   } else
      cout << "El vector no tiene valores impares\n";
   return 0;
}
```

```
void dobla (const int v[], int result[], int tam) {
   for (int i = 0; i < tam; i++)
      result[i] = v[i] * 2;
}

void doblaPunteros (const int* v, int* result, int tam) {
   for (int i = 0; i < tam; i++)
      result[i] = v[i] * 2;
}

int main () {
   const int MAX = 4;
   int v[MAX] = {6, 4, 3, 1};
   int r[MAX];
   dobla (v, r, MAX);
   doblaPunteros (v, r, MAX);
   doblaPunteros (&v[2], &r[2], 2);
   return 0;
}
```

Figura 7.9: Uso de punteros en el paso de parámetros de tipo vector.

7.4. Ejercicios

1. Indique si existe algún tipo de error sintáctico o semántico en las invocaciones a la función doblar realizadas desde la función main del siguiente programa:

```
#include <iostream>

void doblar (int* x) {
   *x = *x * 2;
}

int main () {
  int a = 3, b = 10;
  doblar (&a);
  doblar (b);
  int* c;
  doblar (c);
}
```

2. Escriba una función que tome como parámetros dos punteros a datos del mismo tipo e intercambie el contenido de las zonas de memoria a las que apuntan los punteros.

Es decir, la función debe intercambiar a sus dos parámetros (recibidos por referencia mediante sintaxis de paso de punteros).

3. Escriba una función que tome como parámetro un vector y su tamaño y devuelva el máximo y el mínimo de sus elementos. Utilice parámetros de tipo puntero para especificar el vector y los parámetros de salida de la función.

4. Defina una estructura fecha que almacene el día, mes y año como enteros. Realice las siguientes funciones:

   - Función que reciba un puntero a una fecha y lea valores para la fecha desde la entrada estándar.
   - Función que reciba un puntero a una fecha y envíe la fecha a la salida estándar con el formato día/mes/año.
   - Función que dados dos punteros a fechas indique si la primera es anterior a la segunda.
   - El incremento en un año de una fecha. Tenga en cuenta los años bisiestos: el 29/2/2008 más un año es el 28/2/2009.

   Realice también un programa principal que permita probar el funcionamiento de las funciones.

5. Utilizando una estructura fecha como la del ejercicio anterior realice las siguientes funciones.

   - Función que reciba como entrada un vector de fechas y una fecha determinada y devuelva la dirección de memoria de la primera fecha del vector que coincide con la fecha. Si la fecha no se encuentra en el vector debe devolver la dirección de memoria 0.
   - Función que utilice la función previa para contar cuántas ocurrencias de una fecha existen en un vector de fechas.
   - Función main que defina un vector de fechas e invoque a la primera función para comprobar si una fecha está en el vector y para comprobar si una fecha está en la segunda mitad de un vector.

6. Defina una estructura persona que conste de los campos nombre y edad. Defina también un vector de personas. Cree un vector de punteros que mantenga el vector de personas ordenado por el nombre y otro vector de punteros que mantenga el vector de personas ordenado por la edad—véase la Figura 7.10. Muestre en la salida estándar el vector ordenado por nombre y luego el vector ordenado por edad apoyándose en los vectores de punteros.

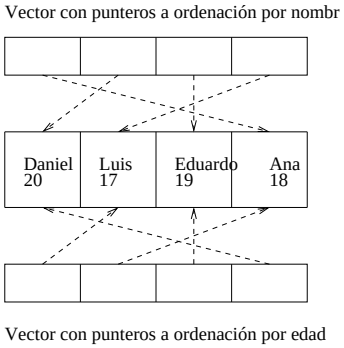

Figura 7.10: Ejemplo de vector y vectores de punteros a los elementos del vector.

Soluciones a los ejercicios que no son programas

1. La segunda invocación es sintácticamente incorrecta, pues el parámetro real b es de tipo entero y el parámetro formal es de tipo puntero. La tercera invocación es sintácticamente correcta pero semánticamente incorrecta pues la variable c apunta a una zona indeterminada de memoria.

Tema 8
Memoria dinámica

La memoria dinámica es una zona de la memoria de un programa en la que se puede reservar y liberar memoria explícitamente, y en tiempo de ejecución, para el almacenamiento de datos. La posibilidad de poder reservar la memoria dinámica explícitamente en tiempo de ejecución conlleva que este tipo de memoria sea especialmente útil para albergar datos cuyo tamaño exacto es conocido al ejecutar un programa o para albergar conjuntos de datos cuyo tamaño varía dinámicamente al ejecutar un programa.

8.1. Zonas de memoria

Un programa en ejecución dispone de tres zonas de memoria donde almacenar sus datos: la memoria global, la pila y la memoria dinámica. A continuación se analizan las características de cada una de estas zonas de memoria.

8.1.1. La memoria global

La zona de memoria global se caracteriza porque su tamaño no varía durante la ejecución de un programa, es decir, el tamaño de la zona de memoria global es estático y se conoce en tiempo de compilación. Los datos almacenados en esta zona de memoria permanecen durante toda la ejecución del programa. La memoria global se utiliza para almacenar las variables o datos globales—véase el Tema 6, Sección 6.4—, pues algunos de estos datos globales se necesitan durante toda la ejecución del programa. Un ejemplo de variables globales son `cin` y `cout`, que están definidas en la biblioteca estándar.

8.1.2. La pila

La pila es la zona de memoria contigua donde se almacenan las variables locales de las funciones y alguna otra información, como la dirección de memoria de retorno de una función. La pila es una zona de memoria cuyo tamaño varía dinámicamente durante la ejecución

de un programa; cuando se invoca a una función la pila crece para albergar a sus variables locales—las variables se apilan—; cuando una función termina la pila decrece y sus variables locales dejan de ser accesibles—las variables se desapilan. Una variable almacenada en la pila tienen un tiempo de existencia limitado al tiempo en que se ejecuta el código asociado a su ámbito. Por lo tanto, es un error devolver como resultado de la ejecución de una función la dirección de memoria de una de sus variables locales. Por ejemplo, la primera función maximo que aparece en el programa de la Figura 8.1 está mal diseñada. Esta función calcula, en la variable local max, el máximo de los elementos del vector que recibe como parámetro de entrada. El error de diseño consiste en devolver la dirección de memoria de la variable local max, donde se almacena el máximo. Esto es un error porque cuando se vaya a utilizar la dirección de memoria donde se almacenaba max la función maximo ya ha terminado y la zona de memoria donde se ubicaban sus variables locales ya se ha desapilado y probablemente se habrá utilizado para almacenar las variables locales de otras funciones. La Figura 8.2 ilustra algunos estados de la pila durante una ejecución del programa de la Figura 8.1. Observe que tras la ejecución de la función maximo la zona de memoria ocupada por sus variables locales se desapila, pero no se pierde su contenido hasta que no se llama a otra función—maximo2—que utiliza la zona de memoria en la que se ubicaban las variables. Por lo tanto, la primera instrucción de main que utiliza cout posiblemente imprima el valor correcto del máximo. En cualquier caso, utilizar una dirección de memoria ya desapila constituye un pobre estilo de programación. Es muy poco probable, sin embargo, que la tercera ejecución de cout en main imprima el valor correcto del máximo.

Observe que las variables locales definidas en el bloque más externo de la función main, aunque se almacenan en la pila, nunca son desapiladas durante la ejecución de un programa, pues hasta que no termina la ejecución de la función main no termina la ejecución del programa.

8.1.3. La memoria dinámica

La memoria dinámica, también conocida como montículo o *heap*, es una zona de la memoria de un programa que presenta las siguientes características:

- El programador es el responsable, mediante instrucciones explícitas, de reservar la memoria en el *heap*. Esto contrasta con la memoria global y la pila, en las que el programador se limita a definir variables, lo que provoca que el compilador, implícitamente, reserve la memoria necesaria para almacenar dichas variables.

- El programador es el responsable, mediante instrucciones explícitas, de liberar la memoria previamente reservada del *heap*. Esto contrasta con la memoria global y la pila. La memoria global se libera cuando termina la ejecución del programa, mientras que las variables de la pila se desapilan—y por tanto, se libera la zona que ocupaban—

```
#include <iostream>
using namespace std;

int* maximo (const int vec[], int tam) {
   int max = vec[0];
   for (int i = 1; i < tam; i++)
      if (vec[i] > max)
         max = vec[i];
   return &max;
}

int maximo2 (int x, int y, int z) {
   return (x > y) ? (x > z ? x : z) : (y > z ? y : z);
}

int main () {
   int v[3] = {5, 7, 3};
   int* p = maximo (v, 3);
   cout << "Máximo: " << *p << '\n';
   cout << "Máximo: " << maximo2 (v[0], v[1], v[2]) << '\n';
   cout << "Máximo: " << *p << '\n';
   return 0;
}
```

Figura 8.1: Función que devuelve la dirección de una variable local.

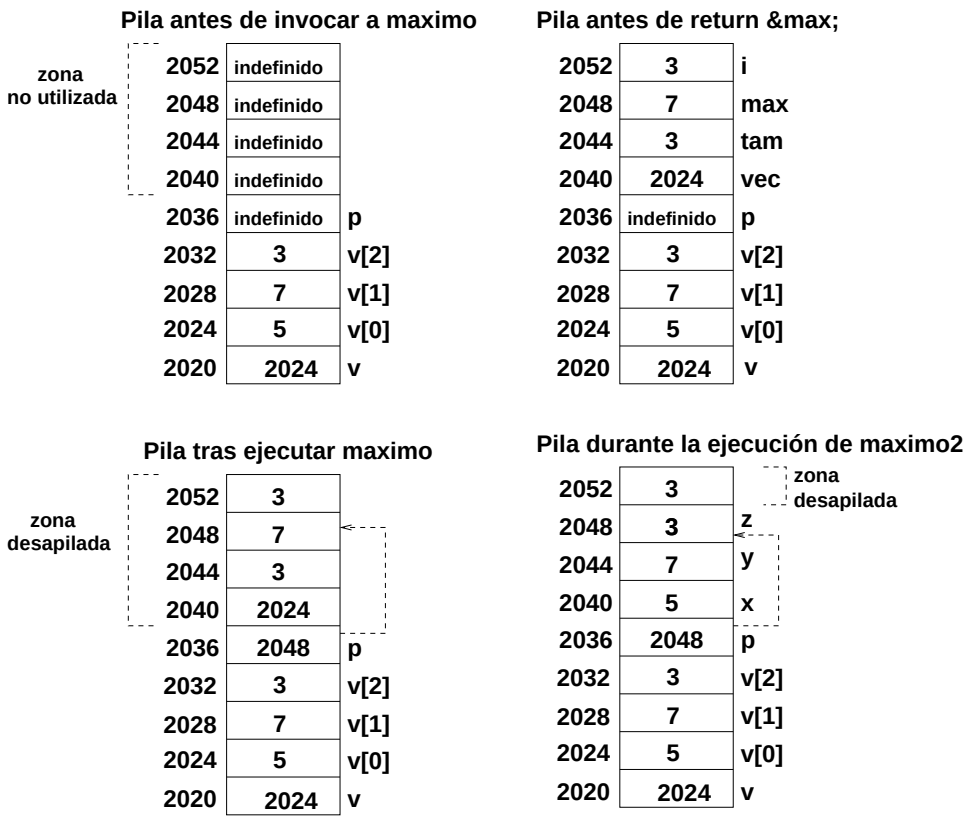

Figura 8.2: Estado de la pila en algunos momentos de una ejecución del programa de la Figura 8.1.

cuando dejan de estar en ámbito, es decir, cuando termina la ejecución del bloque más interno en el que se encuentran definidas.

- Al igual que la pila, y a diferencia de la memoria global, la memoria dinámica utilizada en un programa tiene un tamaño que varía dinámicamente al ejecutar el programa. Cuando se reserva memoria dinámica el tamaño de la memoria dinámica disponible mengua; cuando se libera memoria dinámica el tamaño de la memoria dinámica disponible crece.

8.2. Reserva y liberación de un objeto en la memoria dinámica

La expresión para solicitar la reserva—también llamada asignación—de una zona de memoria donde albergar un objeto en el *heap* es la siguiente:

```
new tipo
```

donde tipo es un tipo válido, es decir, un tipo básico o un tipo definido por el usuario—una estructura o una clase. En caso de que haya suficiente memoria dinámica como para almacenar un dato del tipo solicitado se reserva la memoria para dicho dato y la expresión devuelve la dirección de inicio de la memoria reservada. El tamaño de la memoria solicitada queda determinado implícitamente por el tamaño de un dato del tipo tipo. Por ejemplo, si se reserva memoria para un entero—ejecutando **new int**—y en el ordenador en el que se ejecuta el programa un entero ocupa cuatro *bytes* entonces se están solicitando implícitamente cuatro *bytes*. En caso de que no haya suficiente memoria para albergar a un dato del tipo solicitado se elevará una excepción que terminará la ejecución del programa—en estos apuntes no se estudia cómo capturar y gestionar las excepciones para que no terminen abruptamente la ejecución de un programa.

Si el tipo de dato implicado en la solicitud es una estructura o una clase que necesita obligatoriamente, u opcionalmente, un constructor, entonces la solicitud se debe, o puede, realizarse así:

```
new tipo (expresión1, expresión2, ..., expresiónn)
```

donde la lista de expresiones es, en número y tipo, compatible con los parámetros formales del constructor de la estructura o de la clase tipo.

El programa de la Figura 8.3 ilustra el uso de la memoria dinámica. En la primera instrucción se solicita memoria dinámica para almacenar un dato de tipo entero. Puesto que el operador **new** devuelve la dirección de inicio de la memoria reservada se utiliza una variable de tipo puntero—edad—para almacenar la dirección y poder trabajar con la memoria reservada. La Figura 8.4 refleja el estado de la memoria tras la ejecución de la instrucción ∗edad = 18;. Observe cómo la variable edad se almacena en la pila, mientras que el entero al que apunta edad se almacena en el *heap*. El programa también almacena un dato de tipo string en la memoria dinámica, e inicia la cadena cuando utiliza el operador **new**. Los

```
#include <iostream>
#include <string>
using namespace std;

int main () {
   int* edad = new int;
   *edad = 18;
   string* nombre = new string ("Fernando");
   cout << "Nombre: " << *nombre << ", edad: " << *edad << '\n';
   delete edad;
   delete nombre;
   return 0;
}
```

Figura 8.3: Programa que solicita y libera memoria dinámica.

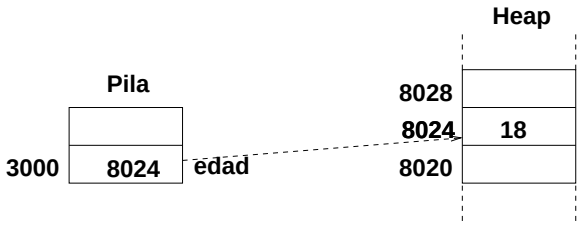

Figura 8.4: Estado de la memoria tras la ejecución de la instrucción *edad = 18; en el programa de la Figura 8.3.

datos de tipos básicos también admiten la sintaxis de iniciación mediante constructor. Las dos primeras instrucciones del programa se podrían haber sustituido por la instrucción: **int*** edad = **new int** (18);. Un dato de tipo básico se puede iniciar en su definición utilizando cualquiera de las dos opciones sintácticas siguientes:

```
tipo var = valor;          // por ejemplo, int x = 18;
tipo var (valor);          // por ejemplo, int x (18);
```

Después de enviar los datos almacenados en el heap a la salida estándar, el programa utiliza el operador **delete** para liberar la memoria dinámica reservada. Como puede observarse, la sintaxis para liberar una zona de memoria es la siguiente:

```
delete direccion;
```

donde direccion es la dirección devuelta en una ejecución previa del operador **new**; direccion también puede valer 0, en cuyo caso el operador **delete** no tiene efecto. Es un error utilizar el operador **delete** con una dirección que no ha sido devuelta por **new** o utilizar más de una vez el operador **delete** sobre una dirección devuelta por **new**. El efecto de un error de este tipo

```
int main () {
   int* p1 = 0;
   delete p1; // correcto, p1 vale 0
   int* p2;
   delete p2; // error: p2 no almacena una dirección devuelta por new
   p1 = new int (5);
   cout << *p1 << '\n'; // correcto
   delete p1; // correcto, p1 contiene una dirección devuelta por new
   cout << *p1 << '\n'; // error: p1 apunta a una zona de memoria liberada
   delete p1; // error: la dirección almacenada en p1 ya fue liberada
   int x = 3;
   int* p3 = &x;
   delete p3; // error: p3 almacena una dirección de la pila
              // y, por tanto, una dirección no devuelta por new
   int* p4 = new int (5);
   int* p5 = p4;
   delete p5; // correcto, p5 contiene una dirección devuelta por new
   delete p4; // error: la dirección almacenada en p4 ya fue liberada
}
```

Figura 8.5: Usos correctos e incorrectos de la memoria dinámica.

es potencialmente desastroso; además, la detección de este tipo de errores entraña especial dificultad. Otro tipo de error potencialmente desastroso es utilizar una zona de memoria ya liberada. Es posible que la memoria se haya concedido para un nuevo uso o que, incluso peor, se esté utilizando para la gestión de las zonas libres de la memoria dinámica; si ocurre esto último la memoria dinámica se corrompe. Por estos motivos hay que tener bastante cuidado cuando se trabaja con memoria dinámica, pues resulta relativamente fácil cometer errores difíciles de detectar. El programa de la Figura 8.5 ilustra algunos usos incorrectos de la memoria dinámica.

Es conveniente liberar la memoria dinámica que no se va a utilizar más, pues es posible que se necesite más adelante en el programa. El siguiente fragmento de código produce un desperdio de memoria dinámica:

```
int* edad = new int;
edad = new int;
```

Observe que al realizar la segunda asignación sobre la variable edad se está perdiendo el valor anterior de la variable, que contenía la dirección de una zona de memoria previamente reservada. Por lo tanto, al perder su dirección se pierde la posibilidad de utilizar la zona de memoria y también la posibilidad de liberarla. A este tipo de pérdida de una zona de memoria se le llama *goteo de memoria*, desde el punto de vista de la ejecución del programa no constituye un error—el programa funciona bien—, sólo es un desperdicio de memoria.

Existen lenguajes, como Java, que disponen de *recolector de basura*. Un recolector de basura se encarga de llevar un registro de las zonas de memoria dinámica que, como en el fragmento de código anterior, han dejado de ser accesibles al programa y liberarlas. En los lenguajes con recolector de basura el programador queda eximido de la reponsabilidad de liberar la memoria dinámica. Existen bibliotecas en C++ que implementan recolectores de basura. El uso de un recolector de basura resulta muy cómodo y seguro para el programador, sin embargo, puede resultar ineficiente en cuanto al tiempo de ejecución del programa—el recolector de basura ejecuta código y no se puede controlar cuándo se ejecuta dicho código.

Los operadores **new** y **delete** son propios de C++ y no existen en C. En C la memoria dinámica se solicita utilizando las funciones malloc, calloc y realloc. A diferencia de **new** estas funciones requieren especificar explícitamente el tamaño de la zona de memoria solicitada. En C la memoria dinámica se libera utilizando las funciones free y realloc. Consulte un manual si está interesado en estas funciones.

La memoria dinámica se utiliza en la construcción de estructuras de datos dinámicas como las listas o los árboles. Estas estructuras van creciendo o decreciendo en tiempo de ejecución en función de las necesidades de memoria de la estructura. Otro uso es en el almacenamiento de un conjunto de datos que se van a utilizar polimórficamente. Estos dos tipos de usos quedan fuera de los objetivos didácticos de estos apuntes. En la siguiente sección también se estudian ejemplos en los que resulta adecuada la memoria dinámica. El programa de la Figura 8.3 constituye un ejemplo de uso de la sintaxis de trabajo con memoria dinámica, pero, ciertamente, no representa un problema para el que sea adecuado el uso de la memoria dinámica. Este programa podría haber utilizado únicamente la pila para almacenar sus datos, lo que hubiera resultado más sencillo y más eficiente tanto en términos de gasto de memoria como de tiempo de ejecución. Una alternativa al programa de la Figura 8.3 que utiliza únicamente la pila para almacenar sus datos es:

```
int main () {
    int edad = 18; // también vale: int edad (18);
    string nombre ("Fernando");
    cout << "Nombre: " << nombre << ", edad: " << edad << '\n';
    return 0;
}
```

8.3. Reserva y liberación de más de un objeto en la memoria dinámica

El operador **new**[] permite la reserva de una zona de memoria dinámica contigua en la que se puede almacenar más de un objeto. Su sintaxis es la siguiente:

```
new tipo[tam]
```

donde tipo es un tipo válido, es decir, un tipo básico o una estructura o clase definida por el programador y tam es una expresión aritmética de tipo entero que al evaluarse produce un valor mayor o igual que cero. El operador **new**[] solicita tam zonas contiguas de memoria dinámica, siendo cada zona del mismo tamaño—el tamaño de los datos del tipo tipo. Si existe suficiente memoria dinámica como para albergar la zona de memoria solicitada, **new**[] devuelve la dirección de inicio de la zona reservada; si no hay espacio suficiente **new**[] eleva una excepción.

El programa de la Figura 8.6 ilustra un caso típico de aplicación de esta nueva sintaxis de **new**[]. El programa necesita almacenar un conjunto de datos de tipo entero, el cardinal de dicho conjunto es solicitado al usuario y, por tanto, se desconoce al escribir el programa. En el Tema 5 se estudió cómo utilizar un vector para almacenar este conjunto de datos en la pila. El problema—véase la Sección 5.1.3—era que al definir un vector hay que especificar su tamaño mediante una expresión entera *constante*, por lo tanto, hay que definir un tamaño lo suficientemente grande como para que el conjunto de datos utilizado en cualquier ejecución del programa quepa en el vector, con el consiguiente desperdicio de memoria en algunas ejecuciones. Afortunadamente, el tamaño de la zona de memoria dinámica solicitado con **new**[] no es una expresión constante, por lo que el programa puede solicitar exactamente el número de memoria que necesita en la instrucción: **int**∗ datos = **new int**[tam];. Observe que la dirección devuelta por **new**[] se almacena en la variable datos para tener acceso a la zona de memoria dinámica reservada. A partir del puntero datos se puede acceder a los elementos de la zona de memoria reservada utilizando sintaxis de acceso a los elementos de un vector, es decir, el operador []—véase el Tema 7, Sección 7.3. En concreto, el programa inicia cada elemento del vector reservado al cuadrado de la posición que ocupa el elemento en el vector. La Figura 8.7 ilustra el estado de la memoria en algunos momentos de una ejecución del programa en la que el usuario decide utilizar tres datos. Un vez iniciado el vector, el programa utiliza otro puntero—pi—para acceder al primer elemento del vector. Por último, el programa libera la memoria reservada. Cuando se utiliza el operador **new**[] para reservar memoria dinámica hay que utilizar el operador **delete**[], en lugar de **delete**, para liberar la memoria reservada. La sintaxis es:

```
delete[] direccion;
```

donde direccion es la dirección devuelta en una ejecución previa del operador **new**[]; direccion también puede valer 0, en cuyo caso el operador **delete**[] no tiene efecto. Es un error utilizar el operador **delete**[] con una dirección que no ha sido devuelta por el operador **new**[] o utilizar más de una vez el operador **delete**[] sobre una dirección devuelta por **new**[]. Por ejemplo, si en el programa de la Figura 8.6 después de la instrucción: **delete**[] datos; apareciera la instrucción **delete**[] pi; el programa cometería un error.

El programa de la Figura 8.8 es similar al programa de la Figura 8.6, pero utiliza las funciones dobla y muestra para iniciar los datos del vector y enviarlos a la salida estándar respectivamente. La Figura 8.9 ilustra el estado de la memoria en algunos instantes de una

```
#include <iostream>
using namespace std;

int main () {
   int tam, i;
   cout << "Introduzca el número de datos: ";
   cin >> tam;
   int* datos = new int[tam];
   for (i = 0; i < tam; i++)
      datos[i] = i*i;
   int* pi = datos;
   pi[0] = 6;
   for (i = 0; i < tam; i++)
      cout << datos[i] << '\n';
   delete[] datos;
   return 0;
}
```

Figura 8.6: Uso del operador **new**[] para solicitar más de un dato.

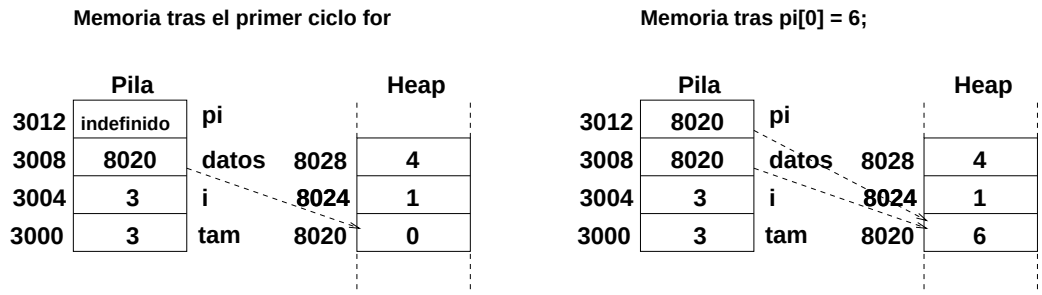

Figura 8.7: Estado de la memoria en algunos momentos de una ejecución del programa de la Figura 8.6 en la que tam vale 3.

```
#include <iostream>
using namespace std;

void dobla (int* v, int tam) {
   for (int i = 0; i < tam; i++)
      v[i] = i*i;
}

void muestra (const int* v, int tam) {
   for (int i = 0; i < tam; i++)
      cout << v[i] << '\n';
}

int main () {
   int tam;
   cout << "Introduzca el número de datos: ";
   cin >> tam;
   int* datos = new int[tam];
   dobla (datos, tam);
   muestra (datos, tam);
   delete[] datos;
   return 0;
}
```

Figura 8.8: Funciones que reciben la dirección de inicio de datos ubicados en la memoria dinámica.

ejecución del programa de la Figura 8.8 en la que el usuario decide utilizar tres datos. Observe que las funciones reciben como parámetro por copia la dirección de inicio en memoria dinámica de los elementos. Esto implica que cualquier modificación que realice la función dobla en los datos no se "pierde" al terminar la ejecución de la función, pues la función dobla modifica los datos almacenados en la memoria dinámica y estos datos persisten tras la ejecución de la función dobla. La función muestra utiliza el calificativo **const** en el parámetro v para indicar que no va a modificar el vector que empieza en la dirección recibida en v, es decir, para indicar que el vector es un parámetro de entrada de la función.

La clase string de la biblioteca estándar utiliza vectores de caracteres ubicados en la memoria dinámica para almacenar los caracteres asociados a las cadenas. Esto hace posible que un string pueda cambiar de tamaño dinámicamente, por ejemplo, al asignarle una nueva cadena de caracteres mediante el operador de asignación. La casi totalidad de las clases de la biblioteca estándar de C++ que almacenan colecciones de datos guardan dichas colecciones de datos en la memoria dinámica.

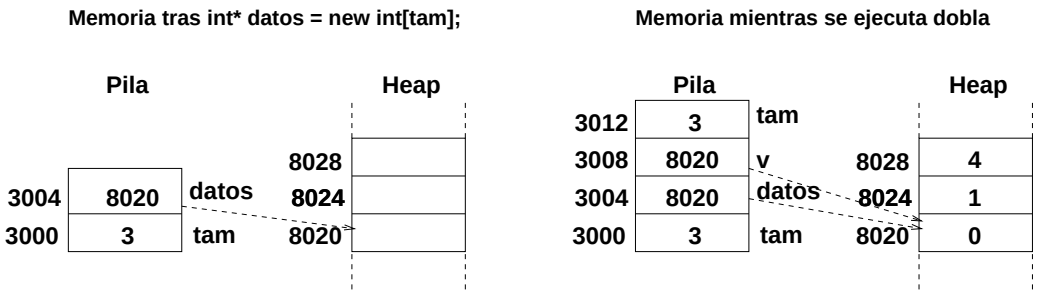

Figura 8.9: Estado de la memoria en una ejecución del programa de la Figura 8.8 en la que tam vale 3.

8.4. Ejemplos de funciones que solicitan memoria dinámica

En esta sección se muestran varios ejemplos de funciones que solicitan memoria dinámica y devuelven la dirección de inicio de la memoria reservada para que esta memoria pueda ser utilizada una vez terminada la ejecución de la función. Las funciones solicitan al usuario con cuántos datos quiere trabajar e inician los datos. Se han elaborado tres funciones que ilustran distintas formas de devolver la dirección de inicio de la memoria dinámica reservada; las funciones también devuelven el número de datos reservados en la memoria dinámica—es decir, el tamaño del vector reservado. El programa de la Figura 8.10 ilustra la función más sencilla para realizar este cometido, esta función devuelve un dato de tipo **int**$*$ con la dirección de inicio de la zona de memoria dinámica reservada; además utiliza un parámetro formal con semántica de paso por referencia para devolver el número de elementos reservados. La segunda función, que aparece en el programa de la Figura 8.11, devuelve el número de elementos solicitados y utiliza un parámetro formal con semántica de paso por referencia para devolver la dirección de inicio de la memoria dinámica reservada. Observe que el parámetro formal es del tipo: **int**$*$&, lo que significa que es una referencia a una variable de tipo **int**$*$. Por último, la función de la Figura 8.12 devuelve la dirección de inicio de la memoria reservada utilizando un paso de parámetro basado en la utilización de punteros—véase el Tema 7, Sección 7.2. Esta función es la más complicada de las tres, para comprenderla hay que estar familiarizado con el funcionamiento de los punteros. Los paréntesis utilizados en la expresión `(*v)[i] = i*i;` son necesarios porque el operador de indexación—[]— tiene mayor precedencia que el operador de desreferencia—*.

Como ejercicio puede representar el estado de la memoria durante la ejecución de los tres programas.

```
#include <iostream>

int* inicia (int& tam) {
   std::cout << "Introduzca el número de datos: ";
   std::cin >> tam;
   int* v = new int[tam];
   for (int i = 0; i < tam; i++)
      v[i] = i*i;
   return v;
}

int main () {
   int t;
   int* datos = inicia (t);
   for (int i = 0; i < t; i++)
      std::cout << datos[i] << '\n';
   delete[] datos;
   return 0;
}
```

Figura 8.10: Función que devuelve la dirección de inicio de la memoria dinámica reservada.

8.5. Asignación de estructuras con campos de tipo vector y de tipo puntero

Cuando se asigna una estructura *E1* que tiene un campo de tipo vector a otra estructura *E2*, la estructura *E2* almacena una copia de todos los campos de la estructura *E1*, incluyendo todos los elementos del vector. Por lo tanto, se puede modificar de manera independiente cualquiera de las dos estructuras sin afectar al contenido de la otra estructura. Por ejemplo, en el programa de la Figura 8.13 se le asigna a la estructura a2 una copia de la estructura a1. Los cambios realizados en a1 tras dicha asignación no afectan a la estructura a2. Por lo tanto, la primera nota del vector en a1 almacena el valor 8.2, mientras que en a2 almacena el valor 7.7.

Sin embargo, cuando se asigna una estructura que tiene un campo de tipo puntero a otra estructura, el campo puntero de ambas estructuras apunta a la misma dirección y, por tanto, las dos estructuras comparten dicha información. Por ejemplo, en el programa de la Figura 8.14 las dos estructuras tienen una copia distinta del nombre del alumno. Sin embargo, el campo notas de ambas estructuras apunta al mismo vector, por lo que el cambio realizado con la asignación: a1.notas[0] += 0.5 afecta a la nota con que trabaja la estructura a2.

```
#include <iostream>

int inicia (int*& v) {
   int tam;
   std::cout << "Introduzca el número de datos: ";
   std::cin >> tam;
   v = new int[tam];
   for (int i = 0; i < tam; i++)
      v[i] = i*i;
   return tam;
}

int main () {
   int* datos;
   int t = inicia (datos);
   for (int i = 0; i < t; i++)
      std::cout << datos[i] << '\n';
   delete[] datos;
   return 0;
}
```

Figura 8.11: Función que devuelve la dirección de inicio de la memoria dinámica reservada mediante un parámetro formal de tipo referencia.

```
#include <iostream>

int inicia (int** v) {
   int tam;
   std::cout << "Introduzca el número de datos: ";
   std::cin >> tam;
   *v = new int[tam];
   for (int i = 0; i < tam; i++)
      (*v)[i] = i*i;
   return tam;
}

int main () {
   int* datos;
   int t = inicia (&datos);
   for (int i = 0; i < t; i++)
      std::cout << datos[i] << '\n';
   delete[] datos;
   return 0;
}
```

Figura 8.12: Función que devuelve la dirección de inicio de la memoria dinámica reservada mediante un parámetro formal de tipo puntero.

```
#include <iostream>
#include <string>

struct Alumno {
   std::string nombre;
   float notas[3];
};

int main () {
   Alumno a1 = { "Antonio", { 7.7f, 9.1f, 6.5f} };
   Alumno a2 = a1;
   a1.notas[0] += 0.5;
   std::cout << a1.notas[0] << ' ' << a2.notas[0] << '\n';
   return 0;
}
```

Figura 8.13: Asignación de estructuras con un campo de tipo vector.

```
#include <iostream>
#include <string>

struct Alumno {
   std::string nombre;
   float* notas;
};

int main () {
   float notas[3] = { 7.7f, 9.1f, 6.5f};
   Alumno a1 = { "Antonio", notas };
   Alumno a2 = a1;
   a1.notas[0] += 0.5;
   std::cout << a1.notas[0] << ' ' << a2.notas[0] << '\n';
   return 0;
}
```

Figura 8.14: Asignación de estructuras con un campo de tipo puntero.

8.6. Campos de tipo puntero que apuntan a datos no compartidos

El programa de la Figura 8.15 es similar al programa de la Figura 8.14, sólo que la información compartida a través del campo notas de la estructura Alumno reside en memoria dinámica en lugar de residir en la pila. Observe que la memoria dinámica reservada se libera al final del programa. En caso de que no queramos compartir la información a la que apunta un campo de tipo puntero deberemos obtener una copia de los datos almacenándolos en memoria dinámica. Este enfoque se ilustra en el programa de la Figura 8.16. Observe que este programa reserva memoria dinámica para obtener una copia de las notas. Puesto que el programa ha almacenado dos vectores en memoria dinámica se deben realizar dos invocaciones a **delete**[] para liberar la memoria dinámica reservada. Como ejercicio modifique el programa de la Figura 8.16 desarrollando una función que realice la copia de una estructura Alumno en otra estructura, la copia debe solicitar memoria dinámica para que las dos estructuras no compartan la información sobre las notas.

La Figura 8.17 ilustra el estado de la memoria para la ejecución de los programas de las Figuras 8.13, 8.14, 8.15 y 8.16 justo antes de terminar el programa.

```
#include <iostream>
#include <string>

struct Alumno {
   std::string nombre;
   float* notas;
};

int main () {
   Alumno a1;
   a1.nombre = "Antonio";
   a1.notas = new float[3];
   a1.notas[0] = 7.7f; a1.notas[1] = 9.1f; a1.notas[2] = 6.5f;
   Alumno a2 = a1;
   a1.notas[0] += 0.5;
   std::cout << a1.notas[0] << ' ' << a2.notas[0] << '\n';
   delete[] a1.notas;
   return 0;
}
```

Figura 8.15: Asignación de estructuras con un campo de tipo puntero. Los datos se albergan en el *heap*

8.7. Conclusiones

En este tema se han analizado las tres zonas de memoria de las que dispone un programa: memoria global, pila y memoria dinámica. También se ha estudiado cómo se puede reservar y liberar memoria dinámica en C++ mediante los operadores **new**, **new**[], **delete** y **delete**[].

La memoria dinámica es muy flexible, pues permite solicitar y liberar memoria en tiempo de ejecución. Esto hace que la gran parte de las clases contenedoras de C++, como las clases string, vector, list o map, utilicen memoria dinámica para poder albergar un conjunto de datos cuyo tamaño varía durante la ejecución del programa.

Sin embargo, la programación con memoria dinámica es compleja. La responsabilidad de la reserva y liberación de memoria recae en el programador, esto puede provocar errores de difícil depuración como el acceso a una zona no reservada de memoria, el acceso a una zona ya liberada de memoria, el liberar más de una vez una zona de memoria o el goteo de memoria.

Al final del tema también se ha analizado la compartición por varias estructuras de datos de una misma zona de datos a la que las distintas estructuras mantienen acceso mediante punteros.

```
#include <iostream>
#include <string>

struct Alumno {
   std::string nombre;
   float* notas;
};

int main () {
   Alumno a1;
   a1.nombre = "Antonio";
   a1.notas = new float[3];
   a1.notas[0] = 7.7f; a1.notas[1] = 9.1f; a1.notas[2] = 6.5f;
   Alumno a2 = a1;
   a2.notas = new float[3];
   for (int i = 0; i < 3; i++)
      a2.notas[i] = a1.notas[i];
   a1.notas[0] += 0.5;
   std::cout << a1.notas[0] << ' ' << a2.notas[0] << '\n';
   delete[] a1.notas;
   delete[] a2.notas;
   return 0;
}
```

Figura 8.16: Copia de la información apuntada por el campo de tipo puntero

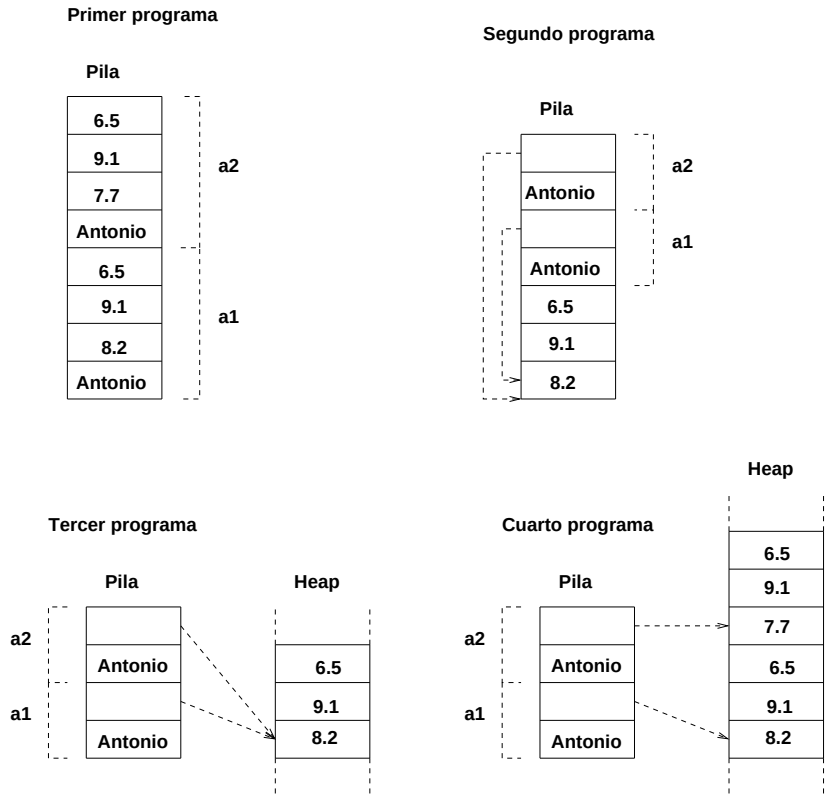

Figura 8.17: Estado de la memoria para los programas con una estructura de campo puntero.

En el diseño de estructuras de datos que utilizan memoria dinámica se suelen utilizar una serie de técnicas que evitan el cometer los citados errores. El estudio de estas técnicas cae fuera del ámbito de estos apuntes. En C++ la encapsulación de las estructuras de datos en clases y la definición correcta de destructores y constructores y asignadores de copia para dichas clases facilita en gran medida el desarrollo de estructuras de datos que almacenan sus datos en memoria dinámica.

8.8. Ejercicios

1. Realice un programa que solicite al usuario un tamaño de vector y reserve memoria dinámica para un vector de enteros con el tamaño indicado. A continuación rellene el vector con valores solicitados al usuario. Realice también una función que muestre el contenido de un vector y utilícela para comprobar que los valores solicitados al usuario se han leído correctamente.

2. La serie de Fibonacci está formada por la secuencia de números: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... Por definición, los dos primeros números de la serie son 0 y 1 y los números siguientes se calculan como la suma de los dos números previos de la secuencia. Realice un programa que solicite al usuario un número n y calcule y muestre en la pantalla los primeros n números de la serie de Fibonacci. Almacena los números de la serie en un vector dinámico.

3. Se almacena en una estructura la siguiente información sobre una persona: nombre, edad y localidad de nacimiento. Realice una función que tome como parámetros de entrada un vector de personas, su longitud y una localidad y devuelva como salida un vector con los datos de las personas cuya localidad de nacimiento coincide con la localidad recibida como parámetro. El vector devuelto debe albergarse en memoria dinámica. Realice también una función main que permita probar la función.

4. Realice las siguientes funciones para el trabajo con vectores de strings:

   - Función que recibe como entrada un tamaño y devuelve un vector ubicado en memoria dinámica de ese tamaño.
   - Función que recibe como entrada un vector de strings y su tamaño y muestre en la salida estándar los strings. Cada string debe aparecer en una línea distinta.
   - Función que reciba un vector de strings ubicado en memoria dinámica, su tamaño y un nuevo tamaño. La función debe trasladar el vector original a otro vector en memoria dinámica que tenga el nuevo tamaño. Nuevo tamaño puede ser menor o mayor que el tamaño previo. Se debe liberar la memoria ocupada por el vector original y se debe devolver la dirección de inicio en memoria dinámica del nuevo vector.

- Función que reciba un vector de strings y devuelva una copia del vector.
- Función main que permita probar las funciones anteriores.

Tema 9
Recursividad

Los lenguajes de programación modernos permiten las funciones recursivas. Una función recursiva es aquella que se invoca a sí misma. Existen problemas que por su naturaleza recurrente o recursiva se solucionan de una forma elegante mediante una función recursiva. Un ejemplo de algoritmo recursivo es el famoso algoritmo de ordenación *Quicksort* de Tony Hoare.

9.1. Funciones recursivas

Como se ha indicado con anterioridad una función recursiva es aquella que se llama a sí misma. Un ejemplo clásico es el cálculo del factorial. El factorial de un entero no negativo n se define como el producto de todos los enteros positivos menores o iguales que n. Por ejemplo, $4! = 4 * 3 * 2 * 1 = 24$. El 0! vale 1. El programa de la Figura 9.1 contiene una función que calcula el factorial utilizando la definición previa. La función utiliza un ciclo **for** para calcular el factorial del número que recibe como parámetro, en cada iteración la variable de control del ciclo se utiliza como un factor. Tenga presente que si los enteros se almacenan utilizando 32 bits, el máximo entero que se puede representar es el 2147483647, lo que implica que la función `factorial` sólo puede calcular el factorial de los enteros del 0 al 12. El factorial de un número también se puede definir de una manera recurrente:

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

La función recursiva `fact` de la Figura 9.2 calcula el factorial de un número aplicando de una forma natural la definición recurrente del factorial de un número.

Una función recursiva consta de uno o varios casos base y general. Un *caso base* es un valor de los parámetros de la función para los que la función se soluciona sin realizar ninguna llamada recursiva. Por ejemplo, en el programa de la Figura 9.2 existe un único caso base: cuando `n` vale cero; en ese caso el factorial se calcula como 1 sin necesidad de llamadas re-

cursivas. Un *caso general o recursivo* es un valor de los parámetros de la función para los que la función utiliza una o varias llamadas recursivas para calcular su solución. La llamada recursiva de un caso general utiliza un parámetro "más pequeño", en el sentido de que genera un caso más próximo a un caso base. Por ejemplo, en el programa de la Figura 9.2 cuando n almacena un valor mayor que cero se genera un caso general, la función fact calcula el factorial utilizando la sentencia res = n * fact (n-1);, que implica una llamada recursiva con un caso más pequeño: $n - 1$. Toda función recursiva debe de constar de al menos un caso base y un caso general. Si no existiera caso base la recursividad nunca terminaría, porque todos los casos serían generales e implicarían una llamada recursiva para su solución. Por otro lado, una función sin caso general no sería una función recursiva porque no utilizaría llamadas recursivas.

La Figura 9.3 muestra el estado de la pila durante la ejecución del programa de la Figura 9.2. En la parte superior se muestra cómo crece la pila conforme las llamadas recursivas se van sucediendo. Por ejemplo, cuando se invoca a fact(4) la función fact calcula el factorial como res = 4 * fact (3);, lo que implica una llamada recursiva a fact(3), fact(3) a su vez realiza una llamada recursiva a fact(2). Y así sucesivamente hasta llegar a la invocación a fact(0). Éste es un caso base, lo que produce la detención de la recursión y el retorno de las llamadas recursivas—parte inferior de la Figura 9.3. Así, fact(0) calcula el resultado como: res = 1; y termina su ejecución devolviendo el valor 1. El control llega ahora a fact(1) que evalúa la expresión res = 1 * fact (0); como res = 1 * 1; y devuelve el valor 1 y, de este modo, van calculándose los factoriales y las funciones recursivas van terminando su ejecución y desapilándose.

Como curiosidad indicar que un programador experimentado en C++ probablemente escribiría una versión mucho más compacta de la función factorial recursiva, como la siguiente:

```
int fact (int n) {
   return (n == 0) ? 1 : n * fact (n - 1);
}
```

9.2. Ejemplos de funciones recursivas

En las siguientes subsecciones se describen varios ejemplos de problemas que se pueden resolver mediante funciones recursivas.

9.2.1. Suma de los dígitos de un entero

Suponga que debe calcular la suma de los dígitos de un entero no negativo n. Este problema se puede plantear de una manera recurrente:

```
#include <iostream>

int factorial (int n) {
   int fact = 1;
   for (i = 2; i <= n; i++)
      fact = fact * i;
   return fact;
}

int main () {
   std::cout << "Factorial(4) =  " << factorial (4) << std::endl;
   return 0;
}
```

Figura 9.1: Cálculo iterativo del factorial de un número.

```
#include <iostream>

int fact (int n) {
   int res;
   if (n == 0)
      res = 1;
   else
      res = n * fact (n-1);
   return res;
}

int main () {
   std::cout << "Factorial(4) =  " << fact (4) << std::endl;
   return 0;
}
```

Figura 9.2: Cálculo recursivo del factorial de un número.

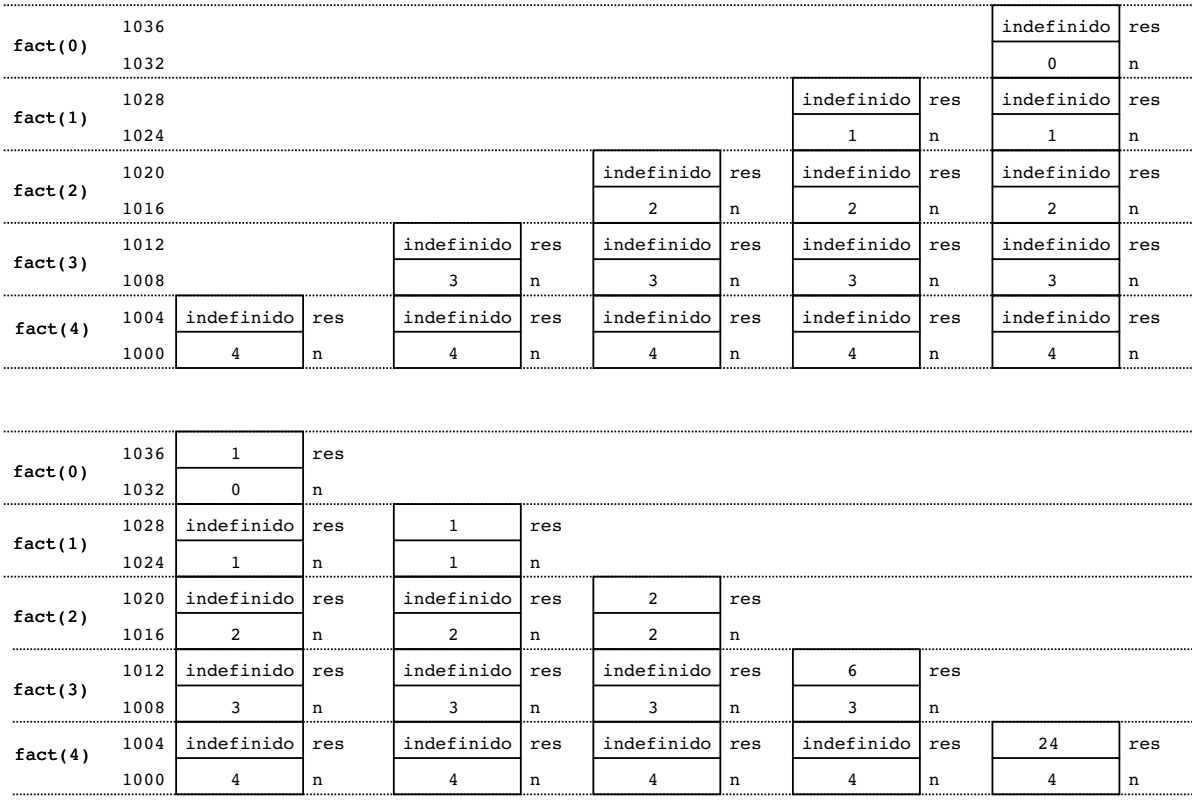

Figura 9.3: Estado de la pila durante la ejecución del programa de la Figura 9.2.

```
#include <iostream>

int sumadig (int n) {
   if (n < 10)
      return n;                          // caso base
   return n %10 + sumadig (n / 10); // caso general
}

int main () {
   int x;
   std::cout << "Introduzca un entero no negativo: ";
   std::cin >> x;
   std::cout << "La suma de sus dígitos es " << sumadig (x) << std::endl;
   return 0;
}
```

Figura 9.4: Suma de los dígitos de un entero no negativo.

$$sumadig(n) = \begin{cases} n & \text{si } n < 10 \\ n\,\%10 + sumadig(n/10) & \text{si } n \geq 10 \end{cases}$$

Es decir, si n es menor que 10 la suma es el propio número y en otro caso la suma es el resto del número entre 10 más la suma de los dígitos del cociente de dividir n entre 10. Dada esta definición recurrente el programa de la Figura 9.4 contiene una función recursiva que implementa esta estrategia para sumar los dígitos de un entero. Este problema también admite una solución iterativa simple, como la siguiente:

```
int sumadig (int n) {
   int suma = 0;
   while (n >= 10) {
      suma += n %10;
      n = n / 10; // equivale a n /= 10;
   }
   return suma + n;
}
```

9.2.2. La serie de Fibonacci

La serie de Fibonacci está formada por la secuencia de números: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...Por definición, los dos primeros números de la serie son 0 y 1 y los números siguientes se calculan como la suma de los dos números previos de la secuencia. La secuencia puede expresarse de una forma elegante mediante la siguiente recurrencia:

```
#include <iostream>

int fibo (int n) {
   if (n < 2)
      return n;
   return fibo (n-1) + fibo (n-2);
}

int main () {
   int x;
   std::cout << "Introduzca un valor mayor o igual que cero: ";
   std::cin >> x;
   std::cout << "Fibonacci (" << x << ") = " << fibo (x) << std::endl;
   return 0;
}
```

Figura 9.5: Calcula un número de la serie de Fibonacci.

$$F_n = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ F_{n-1} + F_{n-2} & \text{si } n > 1 \end{cases}$$

El programa de la Figura 9.5 calcula un número de la serie de Fibonacci. Observe que cuando se da el caso general se realizan dos llamadas recursivas para resolverlo; hasta que no se resuelva la llamada fibo(n−1) no se realizará la llamada fibo(n −2). Como ejercicio puede realizar un programa que calcule de una manera iterativa los n primeros términos de la sucesión.

9.2.3. La búsqueda binaria

Otro ejemplo clásico de algoritmo recursivo es el de la búsqueda binaria en un vector ordenado. Se trata de buscar el índice que ocupa un dato en un vector ordenado. Si el dato se encuentra más de una vez se devuelve cualquiera de sus posiciones; si no se encuentra en el vector se devuelve una posición no válida como -1. En caso de que el vector no estuviera ordenado habría que utilizar un código como el siguiente:

```
int busca (const int v[], int tamv, int dato) {
   for (int i = 0; i < tamv; ++i)
      if (v[i] == dato)
         return i;
   return -1;
}
```

Esta función precisa consultar, en el peor de los casos, todos los elementos del vector para encontrar la posición del dato buscado. Si el vector está ordenado se puede disminuir el número de consultas promedio con el siguiente código:

```
int busca (const int v[], int tamv, int dato) {
   for (int i = 0; i < tamv && v[i] <= dato; ++i)
      if (v[i] == dato)
         return i;
   return -1;
}
```

Como el vector está ordenado en orden creciente y lo recorremos en ese orden, en cuanto se encuentra un elemento mayor que el dato buscado se puede detener la búsqueda porque sabemos que el resto de elementos del vector son mayores que el dato buscado.

Sin embargo, si el vector está ordenado se puede realizar una búsqueda binaria para encontrar el dato de un modo más eficiente. La estrategia utilizada en la búsqueda binaria es la siguiente. En primer lugar se compara el dato con el elemento situado en la posición central del vector. En caso de que coincidan la búsqueda termina y se devuelve la posición central del vector. Si no coinciden y el elemento central es mayor que el dato, entonces se busca en la mitad inferior del vector, en otro caso se busca en la mitad superior. Las posteriores búsquedas utilizan la misma estrategia. Se sigue buscando hasta que se encuentra el dato o se descubre que el dato no está en el vector. En cada paso del algoritmo, en caso de que el dato no coincida con el elemento central se descartan la mitad de los elementos sobre los que se busca, esto implica que sólo se hagan $\log_2 n$ consultas en el vector, donde n es el número de elementos del vector. El programa de la Figura 9.6 muestra una implementación de la búsqueda binaria mediante un algoritmo recursivo. La búsqueda binaria también se puede implementar mediante un algoritmo iterativo como el de la Figura 9.7.

9.3. Recursividad *versus* iteratividad

Todo algoritmo recursivo se puede resolver mediante un algoritmo iterativo, a veces apoyándose en una estructura de datos de tipo pila. En la mayoría de los lenguajes de programación la solución iterativa es más eficiente que la recursiva, pues evita la sobrecarga asociada a una llamada a función, a la copia de los parámetros reales en los formales y al apilar los parámetros formales en la pila. Para muchos problemas a resolver, la solución recursiva es más elegante y fácil de comprender, para este tipo de problemas la solución recursiva suele ser la elegida pese a ser menos eficiente.

La recursividad es una técnica muy aplicada en programación. Se emplea, por ejemplo, en muchos algoritmos que utilizan la técnica divide y vencerás o la técnica de la vuelta atrás—*backtracking*. La recursividad también se emplea ampliamente para trabajar con estructuras de datos de naturaleza recursiva como los árboles.

```
#include <iostream>

int busqueda_rec (const int v[], int dato, int ini, int fin) {
   if (ini > fin)
      return -1;
   int medio = (ini + fin) / 2;
   if (dato == v[medio]) {
      return medio;
   } else if (dato < v[medio]) {
      return busqueda_rec (v, dato, ini, medio-1);
   } else {
      return busqueda_rec (v, dato, medio+1, fin);
   }
}

int main () {
   int v[5] = {1, 3, 5, 7, 9};
   int dato;
   std::cout << "Introduzca dato: ";
   std::cin >> dato;
   int pos = busqueda_rec (v, dato, 0, 4);
   if (pos == -1)
      std::cout << "El dato no está en el vector\n";
   else
      std::cout << "El dato ocupa la posición " << pos << '\n';
  return 0;
}
```

Figura 9.6: Búsqueda binaria recursiva.

```
#include <iostream>

int busqueda_ite (const int v[], int tamv, int dato) {
   int ini = 0;
   int fin = tamv-1;
   while (ini <= fin) {
      int medio = (ini + fin) / 2;
      if (dato == v[medio]) {
         return medio;
      } else if (dato < v[medio]) {
         fin = medio-1;
      } else {
         ini = medio+1;
      }
   }
   return -1;
}

int main () {
   int v[5] = {1, 3, 5, 7, 9};
   int dato;
   std::cout << "Introduzca dato: ";
   std::cin >> dato;
   int pos = busqueda_ite (v, 5, dato);
   if (pos == -1)
      std::cout << "El dato no está en el vector\n";
   else
      std::cout << "El dato ocupa la posición " << pos << '\n';
  return 0;
}
```

Figura 9.7: Búsqueda binaria iterativa.

9.4. Ejercicios

1. Realice una función recursiva que sume los primeros n enteros positivos. Nota: para plantear la función recursiva tenga en cuenta que la suma puede expresarse mediante la siguiente recurrencia:

$$suma(n) = \begin{cases} 1 & \text{si } n = 1 \\ n + suma(n-1) & \text{si } n > 1 \end{cases}$$

2. Escriba una función recursiva que calcule un número elevado a una potencia entera mayor o igual que cero: x^y. Exprese el cálculo mediante una recurrencia y después escriba la función recursiva.

3. Realice una función recursiva que diga si una cadena de caracteres es un palíndromo. Un palíndromo es una frase o palabra que se lee igual de delante hacia atrás que de atrás hacia delante, por ejemplo: reconocer o anilina. Para simplificar suponga que la cadena no contiene ni mayúsculas, ni signos de puntuación, ni espacios en blanco ni tildes.

4. En su libro *Elementos* el matemático griego Euclides describió un método para calcular el máximo común divisor de dos enteros. El método se puede describir con la siguiente recurrencia:

$$mcd(x, y) = \begin{cases} x & \text{si } y = 0 \\ mcd(y, resto(x, y)) & \text{si } x \geq y \text{ y } y > 0 \end{cases}$$

   Realice una función recursiva y otra iterativa que calculen el máximo común divisor de dos enteros.

5. Escriba una función **void** escribeNumeros(**int** ini, **int** fin) que muestre en la salida estándar los números del ini al fin. Escriba una versión que escriba los números en orden ascendente y otra en orden descendente.

Tema 10
Flujos

Un *flujo*—del inglés *stream*—es una abstracción utilizada en los lenguajes de programación para representar una secuencia de datos que puede ser leída o ser escrita. Un flujo del que sólo se pueden leer datos es conocido como un *flujo de entrada*; un flujo en el que sólo se pueden escribir datos es llamado *flujo de salida* y, por último, si se puede tanto leer como escribir en un flujo entonces el flujo se denomina de entrada y salida.

Un flujo está asociado a algún dispositivo físico o virtual que se puede modelar como una secuencia de datos como, por ejemplo, el teclado, el monitor, la red, la impresora, un fichero en disco, un string o una interconexión. La abstracción de flujo hace posible que se pueda trabajar con estos dispositivos de una manera uniforme: a través de la interfaz de trabajo de los flujos.

Para comprender en su totalidad la interfaz de trabajo de los flujos en C++ es preciso tener unos conocimientos de C++ superiores a los obtenidos tras la lectura de estos apuntes. En este tema se verá una versión simplificada—aunque no demasiado—del trabajo con flujos en C++. Nos centraremos en el estudio de flujos de entrada y de salida asociados a ficheros y orientados a texto. El ser orientados a texto significa que se lee y escribe información de tipo texto, es decir, secuencias de caracteres.

10.1. cin y cout

Antes de empezar a trabajar con flujos asociados a ficheros se indica que tanto cin como cout son dos variables globales de tipo flujo definidas en la biblioteca estándar iostream. cin es un flujo de entrada que lee datos de la entrada estándar que, por defecto, está asociada al teclado. cout es un flujo de salida que escribe datos en la salida estándar que, por defecto, está asociada al monitor. Es posible realizar ciertos cambios al ejecutar un programa de forma que la entrada o la salida estándar estén asociadas a otros dispositivos. De esta forma el programa puede obtener su entrada y enviar su salida a otros dispositivos. Por ejemplo,

podría obtener la entrada de un fichero de texto y enviar la salida a la impresora, todo ello sin tener que modificar el programa.

Se indica a continuación, y como curiosidad, cómo redireccionar la salida estándar de un programa para que vaya a un fichero en lugar de ir al monitor. En un sistema operativo Linux suponga que ha compilado el primer programa del Tema 1 y que el ejecutable se llama *saludo*. Escriba en una consola:

```
$ ./saludo >fich
```

y observe cómo en el directorio de trabajo se ha creado un fichero denominado *fich* cuyo contenido es el texto `Hola a todos`, que en lugar de aparecer en el monitor se ha enviado al fichero. En un sistema operativo Windows, y suponiendo que el ejecutable se llama *saludo.exe*, puede obtener lo mismo ejecutando el "programa" *Accesorios–>Símbolo del sistema* y escribiendo en ese programa:

```
C:> saludo >fich
```

10.2. Flujos de salida

Un flujo de salida es un flujo que se utiliza para escribir información. La siguiente instrucción permite escribir información en formato texto en un flujo de salida:

```
f << expresión1 << expresión2 << ... << expresiónn;
```

donde f es una variable de tipo flujo de salida y las distintas expresiones producen como resultado un valor de un tipo básico o bien de tipo string. En el flujo se escribirá el resultado de las distintas expresiones convertidas a una secuencia de caracteres. El resultado de las expresiones se escribe en el flujo según el orden—de izquierda a derecha—en que aparecen escritas en la instrucción.

Se puede escribir en un flujo de salida de la misma forma en que se escribe información en la salida estándar a través de la variable cout—cambiando cout por el nombre de la variable de tipo flujo asociada al flujo en el que se quiere escribir. Por lo tanto, todo lo comentado en estos apuntes para escribir en la salida estándar a través de cout es aplicable a la escritura en un flujo de salida.

En estos apuntes nos limitaremos a escribir información de tipo texto en un flujo. Los flujos también permiten escribir información binaria, es decir, información que puede ser de cualquier tipo. De esta forma, es posible escribir en un flujo un conjunto de enteros almacenados en complemento a dos, o sea, como se almacenan en la memoria principal del ordenador.

10.2.1. Asociación de un fichero a un flujo de salida

En esta sección se estudia cómo se asocia un fichero a un flujo de salida y cómo se escribe información en el fichero a través del flujo. El fichero residirá en algún dispositivo de memoria secundaria como un disco duro.

Para trabajar con un flujo de datos hay que utilizar un *objeto*. Un objeto es una instancia de una *clase*. Una clase es un tipo de dato definido por el usuario, prácticamente igual a una estructura. Las clases tienen una forma de definición y uso relativamente distinta a la forma de definición y uso de los tipos básicos. En la asignatura "Programación orientada a objetos" se estudiará la forma de definir una clase en C++. En este tema nos veremos obligados a trabajar con objetos que pertenecen a clases definidas en la biblioteca estándar de C++, pues los flujos de datos se implementan mediante clases de C++. El tipo string también es una clase de C++. Dos peculiariadades del trabajo con objetos se ilustran en el siguiente fragmento de código:

```
string s ("hola");
cout << s.length () << endl;
```

La primera línea define una variable llamada s que almacena un objeto de la clase string. En los ejemplos anteriores que trabajaban con strings se utilizaba la sintaxis string s = ''hola''; para iniciar una cadena en su definición. Sin embargo, las variables que almacenan objetos se suelen iniciar utilizando la sintaxis:

```
tipo var (lista de parámetros);
```

donde tipo es el tipo o clase de la variable var y la lista de parámetros es análoga a la lista de parámetros reales con que se invoca a una función y sirve para suministrar información para la iniciación del objeto asociado a la variable. La clase string tiene la peculiaridad de que permite la iniciación de los objetos de la clase mediante las dos sintaxis.

La segunda línea del fragmento de código incluye la expresión s.length (). Dado un objeto obj la sintaxis:

```
obj.func (lista de parámetros);
```

invoca a la función func asociada al objeto obj utilizando una lista de parámetros reales. A una función asociada a un objeto se le llamada *función miembro* de la clase del objeto.

Tras este inciso para la introducción de los conceptos de objeto y clase, se retoma el estudio de los flujos de salida asociados a ficheros. El programa de la Figura 10.1 permite escribir una serie de enteros en un fichero en formato texto, almacenando un entero por línea. A continuación se describe el programa:

- Para trabajar con flujos asociados a ficheros es preciso incluir el archivo de cabecera fstream. Este archivo de cabecera de la biblioteca estándar incluye las declaraciones necesarias para el trabajo con flujos asociados a ficheros.

- La primera línea de la función main define la variable salida, cuyo tipo es ofstream. El nombre ofstream viene de *Output File STREAM*, que se puede traducir como flujo de fichero de salida. Una variable de tipo ofstream representa a un flujo de salida al que se le asocia un fichero. La variable salida se inicia con el valor ''enteros''. Esta cadena es el nombre del fichero que se asocia al flujo. Como parte de la iniciación de una variable de tipo flujo de fichero se intenta *abrir* el fichero asociado. La apertura de un fichero es una operación en la que se comprueba si se puede trabajar con el fichero y, en caso de que sea posible, se inicia cierta información para el trabajo con el fichero. Existen diversos modos de abrir un fichero asociado a un flujo de fichero. Si no se especifica nada, como el programa de la Figura 10.1, se utiliza el modo de apertura por defecto para un fichero de salida, que *trunca* el fichero en caso de que exista. Truncar un fichero significa que su contenido previo se pierde. Si el fichero no existía entonces se crea vacío.

- En la segunda línea de la función main se utiliza el nombre de la variable de tipo flujo salida como condición de la sentencia **if**. Cuando se utiliza un objeto de tipo flujo como expresión lógica se realiza una conversión implícita del objeto al tipo **bool**. Esta conversión refleja el estado del objeto tipo flujo. Tras la apertura de un fichero el estado del flujo refleja si se ha podido abrir correctamente el fichero. En el programa, la expresión salida produce **true** si se abrió con éxito el fichero y **false** si el fichero no pudo ser abierto.

- Si el fichero no pudo ser abierto con éxito, el programa no realiza nada más. En otro caso el programa escribe una serie de enteros introducidos por el usuario en el fichero, almacenando cada entero en una línea distinta. El programa termina de escribir enteros en el fichero cuando el usuario introduce un entero negativo, este entero negativo también se introduce en el fichero.

- Existe una operación sobre un flujo de fichero que consiste en el *cierre* del fichero. Esta operación se lleva a cabo cuando no se quiere trabajar más con el fichero asociado al flujo. El cierre de un fichero implica la liberación de los recursos utilizados por el fichero y la actualización del fichero con todos los datos que se han escrito en él. Se puede cerrar de manera explícita el fichero asociado a un objeto flujo f realizando la operación f.close (); . Esta operación podría aparecer tras la sentencia **while** del programa de la Figura 10.1, pues sólo tiene sentido cerrar un fichero si se ha abierto correctamente. Sin embargo, en C++ los programas no suelen cerrar explícitamente los ficheros ya que cuando un objeto de tipo flujo de fichero deja de estar en ámbito su fichero asociado se cierra implícitamente. No obstante, el programador siempre puede cerrar explícitamente los ficheros si así lo precisa.

```
#include <iostream>
#include <fstream>
using namespace std;

int main () {
   ofstream salida ("enteros");
   if (salida) {
      int x;
      do {
         cout << "Introduzca un entero (negativo para terminar): ";
         cin >> x;
         salida << x << '\n';
      } while (x >= 0);
   }
   return 0;
}
```

Figura 10.1: Programa que escribe enteros en formato de texto en un fichero.

Codifique y ejecute el programa de la Figura 10.1 y compruebe que crea un fichero en el disco. Abra el fichero de texto creado e inspeccione su contenido. Vuelva a ejecutar el programa y observe cómo se trunca el contenido previo del fichero.

10.3. Flujos de entrada

Un flujo de entrada se utiliza para representar una secuencia de datos que puede ser leída. La siguiente instrucción permite leer información de tipo texto de un flujo de entrada:

```
f >> variable1 >> variable2 >> ... >> variablen;
```

donde f es una variable de tipo flujo de entrada y variable1, variable2, ..., variablen son variables de tipos básicos o de tipo string. Esta instrucción provoca la lectura de datos desde el flujo y su almacenamiento en las variables especificadas en la instrucción. La instrucción lee los datos en formato de texto y los convierte a la representación interna adecuada al tipo de las variables.

La forma en que se realiza la lectura de cada variable es la siguiente:

- En primer lugar, se salta cualquier secuencia consecutiva de caracteres blancos que exista en la posición de lectura del flujo.
- A continuación se leen caracteres del flujo hasta que se encuentre un carácter *C* que no puede formar parte de un dato del tipo asociado a la variable que se está leyendo. Por

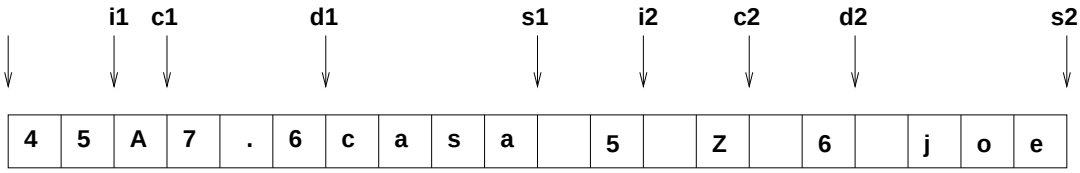

Figura 10.2: Posición de lectura del flujo.

ejemplo, si se lee un entero cuando se lee un carácter que no es un dígito—salvo un carácter + o - situado al principio.

- Se realiza una conversión de los caracteres leídos a la representación interna de la variable que se está leyendo y se almacena el resultado de la conversión en la variable. La posición de lectura del flujo queda como la posición del último carácter leído—el carácter *C*.
- Si se está leyendo un dato de tipo string se procede de la siguiente manera: 1) se salta una posible secuencia inicial de caracteres blancos, 2) se lee en la variable una secuencia consecutiva de caracteres no blancos, 3) la posición de lectura del flujo queda como la posición del primer carácter blanco encontrado tras la secuencia de caracteres no blancos.

Por ejemplo, suponiendo que un flujo de texto contiene el texto:

```
45A7.6casa 5 Z 6 joe
```

el siguiente fragmento de programa:

```
int i1, i2;
char c1, c2;
double d1, d2;
string s1, s2;
f >> i1 >> c1 >> d1 >> s1 >> i2 >> c2 >> d2 >> s2;
```

almacenaría los valores 45, 'A', 7.6, "casa", 5, 'Z', 6 y "joe" en las variables i1, c1, d1, s1, i2, c2, d2 y s2 respectivamente. La Figura 10.2 indica mediante una flecha la posición de lectura del flujo tras la lectura de cada uno de los datos asociados a las variables. La flecha situada más a la izquierda indica la posición inicial de lectura del flujo.

La lectura de la entrada estándar mediante cin presenta el mismo comportamiento que el comportamiento explicado en esta sección para la lectura de un flujo de entrada.

Los flujos de entrada también ofrecen funciones para la lectura de información binaria de un flujo. Esto permite, por ejemplo, leer de un flujo un conjunto de enteros representados en complemento a dos. Este tipo de lectura no se estudiará en estos apuntes.

```
#include <iostream>
#include <fstream>
using namespace std;

int main () {
   ifstream entrada ("enteros");
   if (entrada) {
      int x;
      entrada >> x;
      while (x >= 0) {
         cout << x << endl;
         entrada >> x;
      }
   } else {
      cout << "No se pudo abrir el fichero\n";
   }
   return 0;
}
```

Figura 10.3: Programa que lee enteros de un fichero de texto.

10.3.1. Asociación de un fichero a un flujo de entrada

Se puede asociar un fichero a un flujo de entrada de manera análoga a como se asocia un fichero a un flujo de salida—Sección 10.2.1—, pero utilizando un objeto de la clase ifstream—de *Input File STREAM*. Una vez realizada la asociación se puede utilizar el flujo, como se explicó en la sección previa, para leer el contenido del fichero.

El programa de la Figura 10.3 lee y muestra en la salida estándar los enteros positivos almacenados en un fichero, de nombre enteros, creado en una ejecución del programa de la Figura 10.1. El programa define la variable entrada de tipo flujo de fichero de entrada—ifstream—y la asocia al fichero de nombre enteros. Si el fichero se abre con éxito, el programa lee enteros del fichero hasta que encuentra un valor negativo; todos los valores se muestran en la salida estándar, salvo el valor negativo que hace de centinela para terminar la lectura.

La apertura de un fichero asociado a un flujo de entrada tiene éxito si el fichero existe y se puede leer—el fichero no se trunca. Si el fichero no existe o existe pero no se puede leer la apertura fracasa.

10.3.2. Detección del fin de un flujo

El programa de la Figura 10.3 lee una serie de enteros positivos de un fichero. Para determinar cuándo termina la lectura del fichero se ha utilizado la estrategia de colocar un valor negativo, que hace de centinela, como último entero del fichero. Si un fichero puede conte-

```
#include <iostream>
#include <fstream>
using namespace std;

int main () {
   ifstream entrada ("ficheroEntrada");
   if (! entrada) {
      cout << "No se pudo abrir el fichero\n";
      return 0;
   }
   // Fichero abierto con éxito
   int ndatos, x;
   entrada >> ndatos;
   for (int i = 0; i < ndatos; i++) {
      entrada >> x;
      cout << x << endl;
   }
   return 0;
}
```

Figura 10.4: Programa que lee enteros. El primer entero del fichero indica cuántos enteros se leen.

ner cualquier entero entonces no se puede utilizar un valor centinela—por lo menos de tipo entero—para determinar el fin de la entrada del fichero. En este caso, una solución consiste en incluir en el fichero un entero inicial que indique el número de enteros que guarda el fichero. Por ejemplo, un fichero que quiere almacenar los números del uno al cuatro podría ser el siguiente:

```
4 1 2 3 4
```

Un fichero que utiliza esta estrategia de almacenamiento puede ser leído mediante el programa de la Figura 10.4. Este programa comienza la lectura del fichero leyendo el número de enteros que contiene el fichero en la variable ndatos. Una vez conocido este número utiliza el bucle **for** para leer el resto de enteros que contiene el fichero.

Los flujos de C++ proporcionan mecanismos para determinar si se ha intentado leer de un flujo cuando no quedaban más datos en el flujo. Esto permite leer un fichero, por ejemplo de enteros, sin necesidad de que el fichero incluya un centinela o un entero adicional que indique el número de datos que contiene el fichero. El programa de la Figura 10.5 lee un fichero de enteros hasta que encuentra el fin de fichero. Este programa se basa en las siguientes características para determinar cuándo se han leído todos los enteros del fichero:

- Anteriormente se ha comentado que un código del tipo: entrada >> x es una instrucción. Realmente es una expresión que produce como resultado el flujo, es decir, entrada.
- En la Sección 10.2.1 se indicó que es posible utilizar como condición lógica un objeto de tipo flujo, en cuyo caso se realiza una conversión implícita del objeto de tipo flujo a un valor de tipo lógico que refleja el estado del flujo.
- El estado de un flujo refleja, entre otras cosas, si se ha intentado leer del flujo cuando no quedaban más datos en el flujo. Un flujo en el que se sabe que la próxima operación de lectura sobre el flujo fallará produce el valor **false** al ser convertido a un valor de tipo lógico. Un flujo en el que no se sabe si la próxima operación de lectura fallará o tendrá éxito es convertido al valor **true**. Cuando se ha intentado leer de un flujo y no quedaban más datos se sabe que la próxima operación sobre el flujo fallará y, por tanto, al ser convertido el flujo a un valor de tipo lógico se produce el valor **false**—consulte la Sección 10.6 para una descripción detallada del estado de un flujo.

El programa de la Figura 10.5 utiliza la expresión entrada >> x en la condición del ciclo **while** para leer los datos del fichero y, a la vez, determinar si se ha alcanzado el final del fichero. entrada >> x intenta leer un dato del flujo entrada. Si la lectura tiene éxito—porque había datos en el fichero—se almacena en la variable x el valor leído y el flujo entrada se convierte al valor **true**. La condición del ciclo es verdadera y se procede a ejecutar su cuerpo, que envía el valor del entero leído a la salida estándar. Si al ejecutar entrada >> x no existen más datos a leer, entonces no se almacena nada en x, el estado de entrada refleja que se ha intentado leer del flujo sin existir datos y entrada se convierte al valor **false**, lo que propicia el término de la ejecución del ciclo **while** y de la lectura del fichero.

10.3.3. Detección de errores de lectura

Cuando se lee o escribe en un fichero pueden producirse errores. Un error de escritura ocurre, por ejemplo, cuando se intenta escribir en un fichero que reside en un dispositivo que está lleno o protegido contra la escritura o cuando se intenta escribir en una zona defectuosa de un dispositivo. Los errores de escritura ocurren con muy poca frecuencia, por lo que en estos apuntes no se estudiará su detección y tratamiento—un programa comercial sí debería hacerlo.

La biblioteca de flujos de C++ ofrece diversos mecanismos para la detección y tratamiento de errores en la lectura o escritura en ficheros. A continuación se estudian algunos de estos mecanismos, otros, como los basados en *excepciones*, no se estudiarán en estos apuntes.

Nos vamos a centrar en el estudio de los errores al leer de un flujo. El error de lectura más frecuente ocurre cuando se intenta leer un dato de un determinado tipo y el valor que aparece en el flujo no es de ese tipo. Por ejemplo, suponga que se ejecuta el programa de la Figura 10.5 y el fichero leído tiene el siguiente contenido:

```
#include <iostream>
#include <fstream>
using namespace std;

int main () {
   ifstream entrada ("ficheroEntrada");
   if (entrada) {
      int x;
      while (entrada >> x)
         cout << x << endl;
   } else {
      cout << "No se pudo abrir el fichero\n";
   }
   return 0;
}
```

Figura 10.5: Programa que lee enteros hasta encontrar el fin del fichero.

```
7 6 4z 8 10
```

El programa leería los enteros 7, 6 y 4. En ese momento la posición de lectura del fichero estaría situada en el valor z. Al intentar leer el siguiente valor se produce un error, porque una secuencia de caracteres que empieza por z no puede ser interpretada como un entero. Como consecuencia la lectura falla, no se almacena nada en la variable x y el estado del flujo entrada se actualiza registrándose el hecho de que se ha producido un error de lectura. Cualquier operación de lectura sobre un flujo sobre el que se ha producido un error de lectura fallará. Luego, el flujo entrada se convierte al valor **false** al ser empleado en una condición lógica. Por lo tanto, la ejecución del ciclo **while**, y la lectura del fichero, termina.

El programa de la Figura 10.5, pues, deja de leer enteros si se intenta leer del fichero y no hay más datos o si se produce un error en la lectura. Sería interesante poder consultar si la lectura se ha terminado porque se llegó al fin del fichero o porque se produjo un error de lectura. La función miembro eof de los flujos permite realizar esta consulta. Dado un flujo f la llamada `f.eof ()` devuelve un valor de tipo lógico que indica si f tiene activado el estado de que se ha intentado leer en f y no había más datos a leer—véase la Sección 10.6.

El programa de la Figura 10.6 utiliza la función miembro eof para determinar la causa del fin de la lectura del flujo. Este programa controla el hecho de que la entrada del flujo no sea correcta. Los programas de las Figuras 10.3 y 10.4 no lo hacen. Piense cómo se comportarían estos programas si la entrada no fuese correcta. Por ejemplo, si en la entrada existe un dato que no es un entero o si el número de enteros que dice tener el fichero es superior al que realmente tiene. Cree algunos ficheros de entrada que reflejen estas entradas

```
#include <iostream>
#include <fstream>
using namespace std;

int main () {
   ifstream entrada ("ficheroEntrada");
   if (entrada) {
      int x;
      while (entrada >> x)
         cout << x << endl;
      if (entrada.eof ())
         cout << "Datos leídos correctamente\n";
      else
         cout << "Hubo un error en la lectura\n";
   } else {
      cout << "No se pudo abrir el fichero\n";
   }
   return 0;
}
```

Figura 10.6: Programa que lee enteros y comprueba la causa del final de la lectura.

erróneas y compruebe si el programa se comporta como usted había pensado. Modifique estos programas para que detecten errores de lectura e informen sobre ellos.

10.4. Lectura de flujos sin formato

La lectura de un flujo mediante la expresión `f >> x` permite leer datos en un formato esperado—el de la variable `x`. Sin embargo, este tipo de lectura no sirve cuando lo que se quiere es la lectura sin interpretación de los caracteres del flujo. En las siguientes subsecciones se estudia dos maneras de leer un flujo de texto sin que se realicen interpretaciones sobre los datos leídos del flujo.

10.4.1. Lectura de flujos carácter a carácter

Dado un flujo `f` es posible leer un carácter del flujo utilizando la expresión `f >> x`, donde `x` es una variable de tipo carácter. Sin embargo, esta forma de lectura realiza cierta interpretación: sólo lee los caracteres no blancos del flujo, los caracteres blancos se "saltan". Si se quieren leer todos los caracteres se puede utilizar la función miembro `get` de los flujos:

```
istream& get (char& c);
```

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main () {
   string nombre;
   cout << "Nombre del archivo: ";
   cin >> nombre;
   ifstream entrada (nombre.c_str ());
   if (entrada) {
      char c;
      while (entrada.get (c))
         cout << c;
   }
   return 0;
}
```

Figura 10.7: Programa que lee un fichero carácter a carácter.

Esta función toma como parámetro por referencia una variable de tipo **char** y guarda en esa variable el siguiente carácter encontrado en el flujo de entrada—ya sea un carácter blanco o no blanco. La función devuelve una referencia al propio flujo.

El programa de la Figura 10.7 ejemplifica la lectura de un fichero carácter a carácter. Este programa lee de la entrada estándar el nombre de un fichero de texto y a continuación muestra su contenido en la salida estándar. Observe la definición de la variable entrada; para especificar el nombre del fichero que se asociada al flujo se utiliza la expresión nombre.c_str () en lugar de utilizar nombre. Esto es necesario porque el nombre del fichero hay que pasarlo mediante un parámetro del tipo cadena de caracteres al estilo C—un vector de caracteres terminado en el carácter '\0'. La llamada s.c_str (), donde s es un objeto de la clase string, devuelve s convertida a una cadena de caracteres al estilo C.

El cuerpo del ciclo **while** se encarga de leer el fichero carácter a carácter utilizando la función miembro get. Como get devuelve el flujo, éste se utiliza como condición lógica del ciclo y de esta forma se sale del ciclo cuando se alcanza el fin de la entrada en el flujo.

Tras el ciclo **while** no se ejecuta ningún código para comprobar si se ha producido un error de lectura. Salvo que el fichero esté corrupto, la ejecución de este programa no generará ningún error, ya que no se realiza ninguna interpretación de los caracteres leídos.

10.4.2. Lectura de flujos línea a línea

La otra forma típica de lectura sin formato de un fichero de texto consiste en leer el fichero línea a línea. La función getline:

```
istream& getline (istrema& is, string& str);
```

que se estudió en el Tema 5, Sección 5.3.2, sirve para leer de un flujo una secuencia de caracteres desde la posición actual de lectura del flujo hasta que se encuentre un carácter de nueva línea. La secuencia de caracteres leída se almacena en una variable de tipo string; el carácter de nueva línea no se inserta en el string, pero se elimina del flujo de entrada. La función getline devuelve una referencia al flujo de lectura con el que trabaja y se encuentra declarada en el archivo de cabecera string.

El programa de la Figura 10.8 ilustra la lectura de un flujo línea a línea. Este programa lee un fichero de texto y lo muestra en la salida estándar precediendo cada línea con un número de línea. En la condición del ciclo **while** se realiza la lectura del fichero. Como puede ver los parámetros de getline son, por este orden, el flujo de entrada y el objeto de tipo string en el que se almacena la línea leída. Como getline devuelve el flujo, éste se utiliza en la condición del ciclo **while** para comprobar si se ha llegado al fin de la entrada.

Como curiosidad, indicar que un programador experimentando en C++ seguramente escribiría el ciclo **while** de una forma más compacta. Sería así:

```
while (getline (entrada, s))
   cout << ++n << ": " << s << '\n';
```

Observe que la instrucción cout envía a la salida estándar un carácter de salto de línea tras la cadena s. Esto es necesario puesto que la función getline no almacena el carácter de salto de línea en s.

Al igual que ocurre con la lectura carácter a carácter de un flujo, la probabilidad de errores en la lectura línea a línea de un flujo pasa por problemas de corrupción del flujo.

10.5. Lectura de flujos con campos heterogéneos

En esta sección se muestra un ejemplo de lectura de un flujo que consta de campos heterogéneos. Con la expresión "campos heterogéneos" se indica que, aunque toda la información del flujo sea de tipo texto, los datos del flujo serán interpretados como valores de tipos de datos distintos.

Por ejemplo, suponga un fichero de texto que contiene información sobre los resultados de una convocatoria oficial de una asignatura. Cada línea del fichero contiene información de un alumno que se ha presentado al examen. La información guardada en el fichero consta de tres campos: número de convocatoria, DNI del alumno y nota. Los campos se separan por espacios en blanco. Una línea del fichero podría ser la siguiente:

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main () {
   string nombre;
   cout << "Nombre de archivo: ";
   getline (cin, nombre);
   ifstream entrada (nombre.c_str ());
   if (entrada) {
      string s;
      int n = 0;
      while (getline (entrada, s)) {
         n++;
         cout << n << ": " << s << '\n';
      }
   }
   return 0;
}
```

Figura 10.8: Programa que lee un fichero línea a línea.

```
1 23564789J 6.8
```

El programa de la Figura 10.9 lee un fichero con información sobre una convocatoria. El programa define una estructura, llamada Presentado, que representa la información relacionada con la presentación de un alumno a una convocatoria. Al leer los datos del fichero se lee en primer lugar el número de convocatoria, de tal modo que si no hay más datos en el fichero se termina la lectura. En caso de que el campo número de convocatoria sea leído con éxito se pasa a leer el resto de los campos y se muestran sus valores en la salida estándar.

10.5.1. Lectura de campos de tipo cadena con espacios en blanco

Siguiendo con el ejemplo de la sección anterior, suponga que la información de una convocatoria incluye el nombre del alumno en lugar de su DNI. La información de una convocatoria se guarda en un fichero, donde cada línea del fichero contiene información sobre un alumno que se ha presentado al examen en el siguiente formato:

```
1 Onofre Bouvila Saavedra 6.8
```

Para leer la información de un fichero como éste no sirve el programa de la Figura 10.9, pues una expresión del tipo `f >> s`, donde `f` es un flujo y `s` una variable de tipo `string` lee

```
#include <iostream>
#include <string>
#include <fstream>
using namespace std;

struct Presentado {
   int convocatoria;
   string dni;
   double nota;
};

int main () {
   ifstream entrada ("ficheroEntrada");
   if (entrada) {
      Presentado pre;
      while (entrada >> pre.convocatoria) {
         entrada >> pre.dni >> pre.nota;
         cout << pre.convocatoria << " -- " << pre.dni << " -- "
              << pre.nota << endl;
      }
      if (!entrada.eof ())
         cout << "La lectura terminó con un error\n";
   }
   return 0;
}
```

Figura 10.9: Programa que lee un fichero de campos heterogéneos.

una cadena hasta que encuentra un carácter blanco—en el ejemplo sólo leería `Onofre`. Una posible solución consistiría en reorganizar la información del fichero, situando la cadena con posibles espacios en blanco como último campo de la línea. Es decir:

```
1 6.8 Onofre Bouvila Saavedra
```

De esta forma el fichero podría leerse mediante un código como el siguiente:

```
while (entrada >> pre.convocatoria) {
   entrada >> pre.nota;
   getline (entrada >> ws, pre.nombre);
   cout << ... << endl;
}
```

El efecto de la instrucción getline (entrada >> ws, pre.nombre) es leer una secuencia de caracteres de entrada hasta que se encuentre el carácter nueva línea y almacenar los caracteres leídos en pre.nombre. Observe que se utiliza la expresión entrada >> ws para saltar los caracteres blancos iniciales. Por lo tanto, cambiado el orden de los campos se ha resuelto el problema. Sin embargo, ¿qué se puede hacer si hay más de un campo de tipo cadena de caracteres que puede contener espacios en blanco? Una solución consiste en incluir cada uno de estos campos en una línea aparte del fichero. Por ejemplo, si a la información sobre la presentación de un alumno a una convocatoria se le añade un campo que contenga la dirección del alumno, el fichero con información sobre una convocatoria se puede formatear así:

```
1 6.8
Onofre Bouvila Saavedra
C/ Palencia, n 23, 4B
```

Es decir, la información sobre el número de convocatoria y la nota se almacena en una línea y la información que implica una cadena con posibles espacios en blanco se almacena en líneas separadas. Un fichero de este tipo podría leerse mediante un fragmento de código como el siguiente:

```
while (entrada >> pre.convocatoria) {
   entrada >> pre.nota;
   getline (entrada >> ws, pre.nombre); // lee el nombre
   getline (entrada >> ws, pre.dir);    // lee la dirección
   cout << ... << endl;
}
```

Un enfoque distinto permite almacenar la información relativa a un alumno en una sola línea. Consiste en utilizar un carácter especial para separar los campos del fichero. Este carácter debe ser tal que no se pueda utilizar como parte de un valor de ninguno de los

campos del fichero. En nuestro ejemplo este carácter podría ser el punto y coma. Una línea del fichero de convocatorias quedaría así:

```
1;6.8;Onofre Bouvila Saavedra;C/ Palencia, n 23, 4B;
```

El programa de la Figura 10.10 permite la lectura de un fichero de convocatorias cuyos campos vienen separados por el carácter punto y coma. De este programa se destacan dos expresiones. La primera es:

```
entrada >> pre.convocatoria >> c
```

Con esta expresión se lee un dato de tipo entero en el campo pre.convocatoria. En el caso de que el entero se lea con éxito se lee también un campo de tipo **char** en la variable c. El objeto de la lectura del carácter es "saltar" del flujo el punto y coma que viene tras el número de convocatoria. La siguiente expresión a analizar es:

```
getline (entrada, pre.nombre, ';')
```

Esta expresión tiene un comportamiento equivalente a la expresión:

```
getline (entrada, pre.nombre)
```

pero esta última lee del flujo hasta encontrar el carácter nueva línea, mientras que la expresión getline (entrada, pre.nombre, ';') lee del flujo hasta encontrar el carácter especificado como último parámetro en la función getline. El carácter punto y coma se salta del flujo y no se almacena en el campo pre.nombre.

10.6. Estado de un flujo

En las Secciones 10.2.1, 10.3.2 y 10.3.3 se ha hablado del estado de un flujo y de la conversión de un flujo a un valor de tipo lógico para poder consultar el estado del flujo. En esta sección se analiza con más detenimiento el estado de un flujo y su consulta.

El estado de un flujo se registra mediante tres banderas. Una *bandera* es un tipo de información de naturaleza lógica que se guarda sobre un objeto. Una bandera se encuentra en uno de los siguientes estados: activada o desactivada. Las banderas asociadas al estado de un flujo son las siguientes: **eof**, **fail** y **bad**. Su significado es el siguiente:

- **eof**: se encuentra activada si se ha intentado leer del flujo cuando no quedaban más datos en el flujo—**eof** viene de *End Of File*.
- **fail**: cuando se encuentra activada la siguiente operación de lectura o escritura sobre el flujo fallará. Si se encuentra desactivada, entonces la siguiente operación de lectura o escritura sobre el flujo puede tener éxito.

```
#include <iostream>
#include <string>
#include <fstream>
using namespace std;

struct Presentado {
   int convocatoria;
   string nombre;
   double nota;
   string dir;
};

int main () {
   ifstream entrada ("ficheroEntrada");
   if (entrada) {
      Presentado pre;
      char c;
      while (entrada >> pre.convocatoria >> c) {
         entrada >> pre.nota >> c;
         getline (entrada, pre.nombre, ';');
         getline (entrada, pre.dir, ';');
         cout << pre.convocatoria << " -- " << pre.nombre << " -- "
              << pre.nota << " -- " << pre.dir << endl;
      }
      if (!entrada.eof ())
         cout << "La lectura terminó con un error\n";
   }
   return 0;
}
```

Figura 10.10: Programa que lee un fichero de campos heterogéneos separados por puntos y comas.

- **bad**: se encuentra activada si el flujo está corrompido. Esto ocurre rara vez, por lo que no se prestará más atención a esta bandera.

Cuando se crea un objeto de tipo flujo, éste tiene sus tres banderas desactivadas. A continuación se analizan algunas operaciones que pueden modificar el estado del flujo:

- Cuando se asocia un fichero a un flujo, se intenta abrir el fichero. Si no se puede abrir el fichero, entonces se activa la bandera **fail** del flujo.
- Cuando se intenta leer de un flujo y no existen más datos en el flujo se activan las banderas **eof** y **fail** del flujo.
- Cuando se intenta leer de un flujo mediante una operación del tipo: `f >> x` y los datos encontrados en el flujo no pueden interpretarse como datos válidos para el tipo de la variable `x` se activa la bandera **fail** del flujo.

Cuando un objeto de tipo flujo aparece en una condición lógica es convertido implícitamente a un valor de tipo lógico que indica si está desactivada la bandera **fail**. Es decir, produce el valor **true** si la siguiente operación de lectura o escritura sobre el flujo podría tener éxito y el valor **false** si es seguro que la siguiente operación de lectura o escritura sobre el flujo fallará. Se puede consultar el estado de las banderas de un flujo mediante las siguientes funciones miembro:

- `good ()` devuelve un valor de tipo lógico que indica si la bandera **fail** está desactivada.
- `fail ()` devuelve un valor de tipo lógico que indica si la bandera **fail** está activada.
- `eof ()` devuelve un valor de tipo lógico que indica si la bandera **eof** está activada.
- `bad ()` devuelve un valor de tipo lógico que indica si la bandera **bad** está activada.

Por ejemplo, la expresión `f.good ()` devuelve el valor **true** si la siguiente operación de lectura o escritura sobre el flujo `f` puede tener éxito, en otro caso devuelve el valor **false**.

Observe que siempre que se intente leer de un flujo y no haya más datos a leer o se produzca un fallo en la lectura del flujo se activa la bandera **fail**, lo que imposibilita cualquier intento posterior de lectura del flujo. Existen funciones miembro de un flujo que permiten modificar el valor de las banderas del flujo. Por ejemplo, dado un flujo `f`, la invocación `f.clear ()` desactiva las tres banderas de estado del flujo. Esto es interesante porque permite, por ejemplo, intentar recuperarse de un error en la lectura de un flujo—como que haya aparecido un carácter alfabético en un flujo que debería contener enteros.

La biblioteca de flujos posee mecanismos adicionales a los estudiados en esta sección para consultar y establecer el estado de las banderas de un flujo.

10.7. Otras funcionalidades de los flujos

En esta sección se comentan algunas funcionalidades de los flujos que no se estudian en estos apuntes. Son las siguientes:

Modos de apertura. En este tema se han utilizado los modos de apertura por defecto de los ficheros asociados a flujos. El modo de apertura por defecto de un fichero de salida trunca al fichero si existe y lo crea si no existe. Sin embargo, la biblioteca de flujos de C++ admite otros modos de apertura que permiten, por ejemplo, no truncar un fichero de salida existente o abrir un fichero de salida para añadirle información al final del fichero.

Acceso directo o aleatorio. En este tema se ha trabajado mediante *el acceso secuencial al flujo*. En un flujo de entrada el acceso secuencial significa que los datos del flujo se leen uno detrás de otro, según el orden en que aparecen en el flujo; en un flujo de salida el acceso secuencial quiere decir que los datos se escriben inmediatamente después de los últimos datos escritos. Un flujo con acceso aleatorio permite escribir o leer en cualquier posición del flujo. Para que un flujo pueda utilizarse de manera aleatoria debe estar asociado a un dispositivo que permita el acceso aleatorio. Por ejemplo, un fichero almacenado en un disco permite el acceso aleatorio, pero una impresora o el teclado no lo permiten.

Flujos de entrada y salida. En estos apuntes se ha trabajado con flujos de entrada o con flujos de salida. Los primeros permiten leer información y los segundos enviar información. Existen también flujos de entrada y salida en los que se puede tanto leer como escribir información. Un flujo de entrada y salida tiene que estar asociado a un dispositivo que permita tanto la lectura como la escritura de información, como un fichero almacenado en un disco.

Tratamiento de errores mediante excepciones. Las *excepciones* son un mecanismo soportado por algunos lenguajes de programación. Las excepciones permiten tratar los errores poco comunes de una forma elegante. C++ soporta excepciones y la biblioteca de flujos permite que los errores que ocurren al trabajar con flujos puedan ser tratados mediante excepciones.

Flujos binarios. En este tema se ha estudiado la lectura y escritura en flujos que contienen información de tipo texto. Sin embargo, un flujo puede contener información almacenada en cualquier formato. Los flujos poseen funciones miembro para procesar flujos binarios, o sea, flujos que contienen información almacenada en cualquier formato.

Estructura de la biblioteca de flujos. La biblioteca de flujos está formada por una serie de clases que están relacionadas mediante herencia y que pueden ser utilizadas polimórfi-

camente. Los conceptos de *herencia* y *polimorfismo* se estudiarán en la asignatura "Programación orientada a objetos".

Flujos de cadena. Un flujo de cadena es un flujo que está asociado a un string. Los flujos de entrada de cadena permiten extraer el texto almacenado en un string en formato numérico. Los flujos de salida de cadena permiten convertir información almacenada en formato binario a formato texto y almacenarla en un string.

10.8. Ejercicios

1. Realice un programa que permita al usuario introducir datos de tipo **double** positivos y los vaya guardando en un fichero.

2. Realice un programa que lea los datos almacenados en un fichero generado por un programa como el del ejercicio anterior y los envíe a la salida estándar.

   Nota: Observe que para poder realizar la lectura es preciso que los datos del fichero estén separados por algún carácter delimitador—por ejemplo, por caracteres blancos.

3. Realice una función que, dada una cadena de caracteres que contiene el nombre de un fichero generado por un programa como el del ejercicio 1, lea los valores almacenados en el fichero y los guarde en un vector de elementos de tipo **double** recibido como parámetro. La función debe devolver también el número de datos leídos del fichero.

4. Realice un programa que permita al usuario introducir datos de tipo **int** y los vaya guardando en un fichero separados por el carácter punto y coma.

5. Realice una función que reciba como parámetro de entrada una cadena de caracteres que contiene el nombre de un fichero generado mediante el programa del ejercicio anterior y devuelva como parámetros de salida:

   - un vector de enteros con los valores leídos del fichero.
   - el número de enteros leídos del fichero.
   - un valor de tipo lógico indicando si se abrió con éxito el fichero.
   - un valor de tipo lógico que indique si la lectura del fichero terminó porque se llegó al fin de fichero o por otra causa.

6. Realice un programa que permita escribir en un fichero datos de tipo **char**, **int**, **float** y cadenas de caracteres sin espacios en blanco, en cualquier orden, y hasta que el usuario decida. Realice también un programa que permita visualizar los datos almacenados en un fichero creado con el programa anterior.

   Sugerencia: Para que el fichero pueda ser leído correctamente preceda cada dato almacenado en el fichero con un carácter que indique el tipo del siguiente elemento

guardado en el fichero. Por ejemplo, 'c' para **char**, 'i' para **int**, 'f' para **float** y 's' para string. Un fichero de este tipo podría ser el siguiente:

```
i 66 f 7.5 s hola s Luis c a c b
```

Este fichero contiene un dato de tipo **int**—66—, seguido de un dato de tipo **float**—7.5—, seguido de dos datos de tipo string— hola y Luis—, seguidos por dos datos de tipo **char**—a y b.

7. Realice un programa que trabaje con una matriz bidimensional de enteros, implemente funciones que permitan:
   - guardar una matriz de enteros en un fichero de texto, almacenando cada fila en una línea distinta.
   - recuperar los datos almacenados en un fichero de texto en una matriz.
8. Realice un programa que lea el contenido de un fichero de texto y lo envíe a la salida estándar. Haga dos versiones, una que lea el fichero carácter a carácter y otro que lea el fichero línea a línea.
9. Realice una función que tome como parámetros de entrada el nombre de un fichero de texto y un carácter *c* y devuelva el número de ocurrencias en el fichero del carácter.
10. Realice una función que tome como parámetros dos nombres de ficheros, siendo el primero fichero un fichero de texto, y copie el contenido del primer fichero en el segundo.
11. Realice una función que tome como parámetros un vector con los nombres de ficheros de texto y el nombre de un fichero de salida. La función debe guardar en el fichero de salida la concatenación de todos los ficheros cuyos nombres aparecen en el vector.
12. Realice una función que tome como parámetro de entrada el nombre de un fichero de texto y devuelva la línea del fichero de mayor longitud.
13. Desarrolle un programa que trabaje con un vector de personas. Por cada persona hay que almacenar su nombre, sexo y edad. Realice las siguientes funciones:
    - Función que lea de la entrada estándar una nueva persona y la inserte al final de un vector de personas.
    - Función que inserte al final de un vector de personas las personas almacenadas en un fichero de texto—el fichero tendrá un formato adecuado que debe pensar. La función recibe como parámetros de entrada y salida el vector de personas y su número de elementos y como parámetro de entrada el nombre del fichero de texto con las personas.

- Función que envíe a la salida estándar el contenido de un vector de personas.
- Función que dado un vector de personas y el nombre de un fichero almacene a las personas en el fichero—utilizando un formato que permita al programa leer su contenido.
- Función *main* que contenga un menú que permita probar el resto de funciones.

14. Suponga un fichero de texto con el siguiente formato: cada línea contiene una palabra en inglés separada mediante un espacio en blanco de la misma palabra en castellano. Por ejemplo:

```
kitchen cocina
apple manzana
```

Realice las siguientes funciones:

- Función que tome como parámetro de entrada el nombre de un fichero de texto con el formato comentado y devuelva un vector de estructuras con las palabras almacenadas en el fichero. Cada estructura debe contener dos campos que almacenan la palabra en inglés y en castellano respectivamente.
- Función main que solicite al usuario un nombre de fichero de texto con palabras en inglés y castellano y utilice la función previa para leer las palabras del fichero. A continuación debe mostrar en la salida estándar cada palabra en inglés y debe solicitar al usuario que escriba la palabra en castellano. Tras ello debe mostrar en la salida estándar el número de aciertos y fallos que ha tenido el usuario.

15. Dado un fichero como el siguiente (del que sólo se muestran las primeras cuatro líneas, el resto tienen el mismo formato):

```
Nombre: Luis Montejo Aguilera
Hijos: 3 8 6 2
Nombre: Ignacio Molina Guerrero
Hijos: 1 20
```

En el fichero, el primer entero que aparece tras el texto `Hijos:` indica el número de hijos que tiene la persona cuyo nombre aparece en la línea anterior. Defina una estructura que conste de dos campos: nombre y número de hijos. Realice una función con las siguientes características:

- Entradas: el nombre de un fichero de texto con el formato como el explicado.

- Salidas: Un valor lógico indicando si se pudo abrir el fichero correctamente. Un vector de estructuras—y su tamaño—del tipo que ha tenido que definir. El vector contiene a los pares (nombre, número de hijos) que contiene el fichero. También se debe devolver un valor de tipo lógico indicando si se detectó algún error al leer del fichero.

16. Realice una función cuyos parámetros sean los nombres de dos ficheros de texto. La función hará una copia en el segundo fichero de aquellas frases contenidas en el primer fichero que tengan más de 10 palabras. Las frases en el primer fichero están separadas por el carácter punto y coma. Las frases deben quedar en el fichero destino cada una en una línea. La función debe devolver un valor de tipo lógico indicando si lo dos ficheros se han abierto correctamente. Las palabras se separan por un único espacio en blanco o carácter de nueva línea.

    Si suponemos que el contenido del fichero origen es:

    Si buscas resultados distintos, no hagas siempre lo mismo; Lo que sabemos es una gota de agua, lo que ignoramos es el océano;¡Oh envidia, raíz de infinitos males y carcoma de las virtudes!;El que lee mucho y anda mucho, ve mucho y sabe mucho;

    El contenido del fichero destino debe ser:

    Lo que sabemos es una gota de agua, lo que ignoramos es el océano
    ¡Oh envidia, raíz de infinitos males y carcoma de las virtudes!
    El que lee mucho y anda mucho, ve mucho y sabe mucho