UNIVERSIDAD DE MÁLAGA
Dpt. Lenguajes y CC. Computación
E.T.S.I. Informática
Ingeniería Informática

Fundamentos de Programación con el Lenguaje de Programación C++

Vicente Benjumea y Manuel Roldán

17 de mayo de 2013

Índice general

Prólogo

Este manual pretende ser una guía de referencia para la utilización del lenguaje de programación C++ en el desarrollo de programas. Está orientada a alumnos de primer curso de programación de Ingeniería Informática.

Este manual se concibe como *material de apoyo a la docencia*, y requiere de las explicaciones impartidas en clase por el profesor para su aprendizaje. Así mismo, este manual no pretende *"enseñar a programar"*, supone que el lector posee los fundamentos necesarios relativos a la programación, y simplemente muestra como aplicarlos utilizando el *Lenguaje de Programación C++*.

El lenguaje de programación C++ es un lenguaje muy flexible y versátil, y debido a ello, si se utiliza sin rigor puede dar lugar a construcciones y estructuras de programación complejas, difíciles de comprender y propensas a errores. Debido a ello, restringiremos tanto las estructuras a utilizar como la forma de utilizarlas.

No pretende ser una guía extensa del lenguaje de programación C++. Además, dada la amplitud del lenguaje, hay características del mismo que no han sido contempladas por exceder lo que entendemos que es un curso de programación elemental.

Este manual ha sido elaborado en el Dpto. de Lenguajes y Ciencias de la Computación de la Universidad de Málaga.

La última versión de este documento puede ser descargada desde la siguiente página web:

`http://www.lcc.uma.es/%7Evicente/docencia/index.html`

o directamente desde la siguiente dirección:

`http://www.lcc.uma.es/%7Evicente/docencia/cppdoc/programacion_cxx.pdf`

Parte I

Programación Básica

Capítulo 1

Un Programa C++

En principio, un *programa C++* se almacena en un fichero cuya extensión será una de las siguientes: ".cpp", ".cxx", ".cc", etc. Más adelante consideraremos programas complejos cuyo código se encuentra distribuido entre varios ficheros (véase 10).

Dentro de este fichero, normalmente, aparecerán al principio unas líneas para incluir las definiciones de los módulos de biblioteca que utilice nuestro programa. Posteriormente, se realizarán declaraciones y definiciones de tipos, de constantes (véase 2) y de subprogramas (véase 5) cuyo ámbito de visibilidad (véase 4.2) será global a todo el fichero (desde el punto donde ha sido declarado hasta el final del fichero).

De entre las definiciones de subprogramas, debe definirse una función principal, denominada *main*, que indica donde comienza la ejecución del programa. Al finalizar, dicha función devolverá un número entero que indica al Sistema Operativo el estado de terminación tras la ejecución del programa (un número **0** indica terminación normal). En caso de no aparecer explícitamente el valor de retorno de `main`, el sistema recibirá por defecto un valor indicando terminación normal.

Ejemplo de un programa que convierte una cantidad determinada de *euros* a su valor en *pesetas*.

```
//- fichero: euros.cpp ----------------------------------------------
#include <iostream>
using namespace std ;
const double EUR_PTS = 166.386 ;
int main()
{
   cout << "Introduce la cantidad (en euros): " ;
   double euros ;
   cin >> euros ;
   double pesetas = euros * EUR_PTS ;
   cout << euros << " Euros equivalen a " << pesetas << " Pts" << endl ;
   // return 0 ;
}
//- fin: euros.cpp --------------------------------------------------
```

Se deberá compilar el programa (código fuente) contenido en el fichero `euros.cpp` para traducirlo a un programa ejecutable mediante un compilador. En caso de utilizar el compilador *GNU GCC*, la compilación se realizará de la siguiente forma:

```
g++ -ansi -Wall -Werror -o euros euros.cpp
```

Una vez compilado correctamente el programa, su ejecución podrá ser como se indica a continuación, donde el texto enmarcado corresponde a una entrada de datos del usuario:

```
Introduce la cantidad (en euros): 3.5 ENTER
3.5 Euros equivalen a 582.351 Pts
```

En algunos entornos de programación, por ejemplo Dev-C++ en Windows, puede ser necesario pausar el programa antes de su terminación, para evitar que desaparezca la ventana de ejecución. En este caso el programa anterior quedaría:

```
//- fichero: euros.cpp -----------------------------------------
#include <iostream>
#include <cstdlib>
using namespace std ;
const double EUR_PTS = 166.386 ;
int main()
{
   cout << "Introduce la cantidad (en euros): " ;
   double euros ;
   cin >> euros ;
   double pesetas = euros * EUR_PTS ;
   cout << euros << " Euros equivalen a " << pesetas << " Pts" << endl ;
   system("pause") ;  // llamada para que el S.O. Windows pause el programa
   // return 0 ;
}
//- fin: euros.cpp -------------------------------------------------
```

Ejemplo de un programa que imprime los números menores que uno dado por teclado.

```
//- fichero: numeros.cpp ---------------------------------------
#include <iostream>   // biblioteca de entrada/salida
using namespace std ; // utilización del espacio de nombres de la biblioteca
// ------------------------------------
// Imprime los números menores a 'n'
// ------------------------------------
void imprimir_numeros(int n)
{
   for (int i = 0; i < n; ++i) {
       cout << i << " " ; // escribe el valor de 'i'
   }
   cout << endl ;   // escribe 'fin de línea'
}
// ------------------------------------
// Imprime los números menores a 'n'
// ------------------------------------
int main()
{
   int maximo ;
   cout << "Introduce un número: " ;
   cin >> maximo ;
   imprimir_numeros(maximo) ;
   // return 0 ;
}
//- fin: numeros.cpp -----------------------------------------------
```

En un programa C++ podemos distinguir los siguientes elementos básicos, considerando que las letras minúsculas se consideran diferentes de las letras mayúsculas:

- *Palabras reservadas*

  Son un conjunto de palabras que tienen un significado predeterminado para el compilador, y sólo pueden ser utilizadas con dicho sentido. Por ejemplo: `using`, `namespace`, `const`, `double`, `int`, `char`, `bool`, `void`, `for`, `while`, `do`, `if`, `switch`, `case`, `default`, `return`, `typedef`, `enum`, `struct`, etc.

- *Identificadores*

  Son nombres elegidos por el programador para representar entidades (tipos, constantes, variables, funciones, etc) en el programa.

  Se construyen mediante una secuencia de letras y dígitos, siendo el primer carácter una letra. El carácter `'_'` se considera como una letra, sin embargo, los nombres que comienzan con dicho carácter se reservan para situaciones especiales, por lo que no deberían utilizarse en programas.

  En este manual, seguiremos la siguiente convención para los identificadores:

  Constantes Simbólicas: Sólo se utilizarán letras mayúsculas, dígitos y el carácter `'_'`. Ejemplo: `EUR_PTS`

  Tipos: Comenzarán por una letra mayúscula seguida por letras mayúsculas, minúsculas, dígitos o `'_'`. Deberá contener al menos una letra minúscula. Ejemplo: `Persona`

  Variables: Sólo se utilizarán letras minúsculas, dígitos y el carácter `'_'`. Ejemplo: `euros`, `pesetas`, `n`, `i1`, etc.

  Funciones: Sólo se utilizarán letras minúsculas, dígitos y el carácter `'_'`. Ejemplo: `imprimir_numeros`

  Campos de Registros: Sólo se utilizarán letras minúsculas, dígitos y el carácter `'_'`. Ejemplo: `nombre`

- *Constantes literales*

  Son valores que aparecen explícitamente en el programa, y podrán ser lógicos, numéricos, caracteres y cadenas. Ejemplo: `true`, `false`, `0`, `25`, `166.386`, `" Pts"`, `' '`, etc.

- *Operadores*

  Símbolos con significado propio según el contexto en el que se utilicen. Ejemplo: `= << >> * / % + - < > <= >= == != ++ -- . ,` etc.

- *Delimitadores*

  Símbolos que indican comienzo o fin de una entidad. Ejemplo: `( ) { } ; , < >`

- *Comentarios y espacios en blanco*

  Los espacios en blanco, tabuladores, nueva línea, retorno de carro, avance de página y los comentarios son ignorados por el compilador, excepto en el sentido en que separan elementos.

  Los comentarios en un programa es texto que el programador escribe para facilitar la comprensión, o remarcar algún hecho importante a un lector humano, y son, por lo tanto, ignorados por el compilador.

  Los comentarios en C++ se expresan de dos formas diferentes:

  - Comentarios hasta fin de línea: los símbolos `//` marcan el comienzo del comentario, que se extiende hasta el final de la línea.

    ```
    // acumula el siguiente número
    suma = suma + n ; // acumula el valor de 'n'
    ```

  - Comentarios enmarcados: los símbolos `/*` marcan el comienzo del comentario, que se extiende hasta los símbolos del fin del comentario `*/`

    ```
    /*
     * acumula el siguiente número
     */
    suma = suma + n ; /* acumula el valor de 'n' */
    ```

Capítulo 2

Tipos Simples

El *tipo* define las características que tiene una determinada entidad, de tal forma que toda entidad manipulada por un programa lleva asociado un determinado tipo. Las características que el tipo define son:

- El rango de posibles valores que la entidad puede tomar.
- El conjunto de operaciones y manipulaciones aplicables a la entidad.
- El espacio de almacenamiento necesario para almacenar dichos valores.
- La interpretación del valor almacenado.

Los tipos se pueden clasificar en tipos simples y tipos compuestos. Los *tipos simples* se caracterizan porque sus valores son indivisibles, es decir, no se puede acceder o modificar parte de ellos (aunque ésto se pueda realizar indirectamente mediante operaciones de bits) y los *tipos compuestos* se caracterizan por estar formados como un agregado o composición de otros tipos, ya sean simples o compuestos.

2.1. Declaración Vs. Definición

Con objeto de clarificar la terminología, en C++ una *declaración* "presenta" un identificador para el cual la entidad a la que hace referencia deberá ser definida posteriormente.

Una *definición* "establece las características" de una determinada entidad para el identificador al cual se refiere. Toda definición es a su vez también una declaración.

Es obligatorio que por cada entidad, sólo exista **una única definición** en la unidad de compilación, aunque pueden existir varias declaraciones. Así mismo, también es obligatorio la declaración de las entidades que se manipulen en el programa, especificando su tipo, identificador, valores, etc. antes de que sean utilizados.

2.2. Tipos Simples Predefinidos

Los *tipos simples predefinidos* en el lenguaje de programación C++ son:

```
bool char int float double
```

El tipo `bool` se utiliza para representar valores lógicos (o booleanos), es decir, los valores "Verdadero" o "Falso" o las constantes lógicas `true` y `false`. Suele almacenarse en el tamaño de palabra más pequeño posible direccionable (normalmente 1 byte).

El tipo `char` se utiliza para representar los caracteres, es decir, símbolos alfanuméricos (dígitos y letras mayúsculas y minúsculas), de puntuación, espacios, control, etc. Normalmente utiliza un espacio de almacenamiento de 1 byte (8 bits) y puede representar 256 valores diferentes (véase 2.8).

El tipo `int` se utiliza para representar los números Enteros. Su representación suele coincidir con la definida por el tamaño de palabra del procesador sobre el que va a ser ejecutado, hoy día suele ser de 4 bytes (32 bits), aunque en determinados ordenadores puede ser de 8 bytes (64 bits).

> Puede ser modificado para representar un rango de valores menor mediante el modificador `short` (normalmente 2 bytes [16 bits]) o para representar un rango de valores mayor mediante el modificador `long` (normalmente 4 bytes [32 bits] u 8 bytes [64 bits]) y `long long` (normalmente 8 bytes [64 bits]).
>
> También puede ser modificado para representar solamente números Naturales (enteros positivos) utilizando el modificador `unsigned`.

Tanto el tipo `float` como el `double` se utilizan para representar números reales en formato de punto flotante diferenciándose en el rango de valores que representan, utilizándose el tipo `double` (normalmente 8 bytes [64 bits]) para representar números de punto flotante en "doble precisión" y el tipo `float` (normalmente 4 bytes [32 bits]) para representar la "simple precisión". El tipo `double` también puede ser modificado con `long` para representar "cuádruple precisión" (normalmente 12 bytes [96 bits]).

Todos los tipos simples tienen la propiedad de ser indivisibles y además mantener una relación de orden entre sus elementos (se les pueden aplicar los operadores relacionales 2.5). Se les conoce también como tipos ***Escalares***. Todos ellos, salvo los de punto flotante (`float` y `double`), tienen también la propiedad de que cada posible valor tiene un único antecesor y un único sucesor. A éstos se les conoce como tipos ***Ordinales*** (en terminología C++, también se les conoce como tipos integrales, o enteros).

Ⓐ Valores Límites de los Tipos Predefinidos

Tanto la biblioteca estándar `climits` como `cfloat` definen constantes, accesibles por los programas, que proporcionan los valores límites que pueden contener las entidades (constantes y variables) de tipos simples. Para ello, si un programa necesita acceder a algún valor definido, deberá incluir la biblioteca correspondiente, y utilizar las constantes adecuadas.

Biblioteca `climits`

```
#include <climits>
```

		`char`	`short`	`int`	`long`	`long long`
unsigned	*máximo*	`UCHAR_MAX`	`USHRT_MAX`	`UINT_MAX`	`ULONG_MAX`	`ULLONG_MAX`
signed	*máximo*	`SCHAR_MAX`	`SHRT_MAX`	`INT_MAX`	`LONG_MAX`	`LLONG_MAX`
	mínimo	`SCHAR_MIN`	`SHRT_MIN`	`INT_MIN`	`LONG_MIN`	`LLONG_MIN`
	máximo	`CHAR_MAX`				
	mínimo	`CHAR_MIN`				
	n bits	`CHAR_BIT`				

Biblioteca `cfloat`

```
#include <cfloat>
```

`FLT_EPSILON`	Menor número `float` x tal que $1{,}0 + x \neq 1{,}0$
`FLT_MAX`	Máximo número `float` de punto flotante
`FLT_MIN`	Mínimo número `float` normalizado de punto flotante
`DBL_EPSILON`	Menor número `double` x tal que $1{,}0 + x \neq 1{,}0$
`DBL_MAX`	Máximo número `double` de punto flotante
`DBL_MIN`	Mínimo número `double` normalizado de punto flotante
`LDBL_EPSILON`	Menor número `long double` x tal que $1{,}0 + x \neq 1{,}0$
`LDBL_MAX`	Máximo número `long double` de punto flotante
`LDBL_MIN`	Mínimo número `long double` normalizado de punto flotante

Para ver el tamaño (en bytes) que ocupa un determinado tipo/entidad en memoria, podemos aplicarle el siguiente operador:

- `unsigned sz = sizeof(tipo) ;`
- `unsigned sz = sizeof(variable) ;`

Por definición, `sizeof(char)` es 1. El número de bits que ocupa un determinado tipo se puede calcular de la siguiente forma:

```
#include <iostream>
#include <climits>
using namespace std ;
int main()
{
    unsigned nbytes = sizeof(int) ;
    unsigned nbits = sizeof(int) / sizeof(char) * CHAR_BIT ;
    cout << "int: "<<nbytes<<" "<<nbits<<" "<<INT_MIN<<" "<<INT_MAX<<endl ;
}
```

Veamos un cuadro resumen con los tipos predefinidos, su espacio de almacenamiento y el rango de valores para una máquina de 32 bits, donde para una representación de n bits, en caso de ser un tipo entero con signo, sus valores mínimo y máximo vienen especificados por el rango $[(-2^{n-1}) \cdots (+2^{n-1}-1)]$, y en caso de ser un tipo entero sin signo, sus valores mínimo y máximo vienen especificados por el rango $[0 \cdots (+2^{n}-1)]$:

Tipo	Bytes	Bits	Min.Valor	Max.Valor
`bool`	1	8	`false`	`true`
`char`	1	8	-128	127
`short`	2	16	-32768	32767
`int`	4	32	-2147483648	2147483647
`long`	4	32	-2147483648	2147483647
`long long`	8	64	-9223372036854775808	9223372036854775807
`unsigned char`	1	8	0	255
`unsigned short`	2	16	0	65535
`unsigned`	4	32	0	4294967295
`unsigned long`	4	32	0	4294967295
`unsigned long long`	8	64	0	18446744073709551615
`float`	4	32	1.17549435e-38	3.40282347e+38
`double`	8	64	2.2250738585072014e-308	1.7976931348623157e+308
`long double`	12	96	3.36210314311209350626e-4932	1.18973149535723176502e+4932

2.3. Tipos Simples Enumerados

Además de los tipos simples predefinidos, el programador puede definir nuevos tipos simples que expresen mejor las características de las entidades manipuladas por el programa. Así, dicho tipo se definirá en base a una enumeración de los posibles valores que pueda tomar la entidad asociada. A dicho tipo se le denomina *tipo enumerado*, es un tipo simple ordinal, y se define de la siguiente forma:

```
enum Color {
    ROJO,
    AZUL,
    AMARILLO
} ;
```

De esta forma definimos el tipo `Color`, que definirá una entidad (constante o variable) que podrá tomar cualquiera de los diferentes valores enumerados. Los tipos enumerados, al ser tipos definidos por el programador, no tiene entrada ni salida predefinida por el lenguaje, sino que deberá ser el programador el que especifique (programe) como se realizará la entrada y salida de datos en caso de ser necesaria. Otro ejemplo de enumeración:

```
enum Meses {
    Enero, Febrero, Marzo, Abril, Mayo, Junio, Julio,
    Agosto, Septiembre, Octubre, Noviembre, Diciembre
} ;
```

2.4. Constantes y Variables

Podemos dividir las entidades que nuestro programa manipula en dos clases fundamentales: aquellos cuyo valor no varía durante la ejecución del programa (*constantes*) y aquellos otros cuyo valor puede ir cambiando durante la ejecución del programa (*variables*).

Constantes

Las constantes pueden aparecer a su vez como *constantes literales*, son aquellas cuyo valor aparece directamente en el programa, y como *constantes simbólicas*, aquellas cuyo valor se asocia a un identificador, a través del cual se representa.

Ejemplos de constantes literales:

- Constantes lógicas (bool):

  ```
  false, true
  ```

- Constantes carácter (`char`), el símbolo constante aparece entre comillas simples:

  ```
  'a', 'b', ..., 'z',
  'A', 'B', ..., 'Z',
  '0', '1', ..., '9',
  ' ', '.', ',', ':', ';', ...
  ```

  Así mismo, ciertos caracteres constantes tienen un significado especial (caracteres de escape):

  - `'\n'`: fin de línea (newline)
  - `'\r'`: retorno de carro (carriage-return)
  - `'\b'`: retroceso (backspace)
  - `'\t'`: tabulador horizontal
  - `'\v'`: tabulador vertical
  - `'\f'`: avance de página (form-feed)
  - `'\a'`: sonido (audible-bell)
  - `'\0'`: fin de cadena
  - `'\137'`, `'\x5F'`: carácter correspondiente al valor especificado en notación octal y hexadecimal respectivamente

- Constantes cadenas de caracteres literales, la secuencia de caracteres aparece entre comillas dobles (puede contener caracteres de escape):

  ```
  "Hola Pepe"
  "Hola\nJuan\n"
  "Hola " "María"
  ```

- Constantes enteras, pueden ser expresadas en decimal (base 10), hexadecimal (base 16) y octal (base 8). El sufijo `L` se utiliza para especificar `long`, el sufijo `LL` se utiliza para especificar `long long`, el sufijo `U` se utiliza para especificar `unsigned`, el sufijo `UL` especifica `unsigned long`, y el sufijo `ULL` especifica `unsigned long long`:

  ```
  123, -1520, 2345U, 30000L, 50000UL, 0x10B3FC23 (hexadecimal), 0751 (octal)
  ```

- Constantes reales, números en punto flotante. El sufijo `F` especifica `float`, y el sufijo `L` especifica `long double`:

  ```
  3.1415, -1e12, 5.3456e-5, 2.54e-1F, 3.25e200L
  ```

Constantes Simbólicas

Las constantes simbólicas se declaran indicando la palabra reservada `const` seguida por su tipo, el nombre simbólico (o identificador) con el que nos referiremos a ella y el valor asociado tras el símbolo (=). Usualmente se definen al principio del programa (después de la inclusión de las cabeceras de las bibliotecas), y serán visibles desde el punto de declaración, hasta el final del programa. Ejemplos de constantes simbólicas:

```
const bool OK = true ;
const char SONIDO = '\a' ;
const short ELEMENTO = 1000 ;
const int MAXIMO = 5000 ;
const long ULTIMO = 100000L ;
const long long TAMANO = 1000000LL ;
const unsigned short VALOR = 100U ;
const unsigned FILAS = 200U ;
const unsigned long COLUMNAS = 200UL ;
const unsigned long long NELMS = 2000ULL ;
const float N_E = 2.7182F ;
const double LOG10E = log(N_E) ;
const long double N_PI = 3.141592L ;
const Color COLOR_DEFECTO = ROJO ;
```

Variables

Las *variables* se definen, dentro de un bloque de sentencias (véase 4.1), especificando su tipo y el identificador con el que nos referiremos a ella, y serán visibles desde el punto de declaración hasta el final del cuerpo (bloque) donde han sido declaradas. Se les podrá asignar un valor inicial en la definición (mediante el símbolo =), si no se les asigna ningún valor inicial, entonces tendrán un valor **inespecificado**. Su valor podrá cambiar mediante la sentencia de asignación (véase 4.3) o mediante una sentencia de entrada de datos (véase 3.2).

```
{
    char letra ; // valor inicial inespecificado
    int contador = 0 ;
    double total = 5.0 ;
    ...
}
```

2.5. Operadores

Los siguientes *operadores* se pueden aplicar a los datos, donde las siguiente tabla los muestra ordenados de mayor a menor orden de precedencia, así como también muestra su asociatividad:

Operador	Tipo de Operador	Asociatividad
`[] -> .`	Binarios	Izq. a Dch.
`! ~ - *`	Unarios	Dch. a Izq.
`* / %`	Binarios	Izq. a Dch.
`+ -`	Binarios	Izq. a Dch.
`<< >>`	Binarios	Izq. a Dch.
`< <= > >=`	Binarios	Izq. a Dch.
`== !=`	Binarios	Izq. a Dch.
`&`	Binario	Izq. a Dch.
`^`	Binario	Izq. a Dch.
`|`	Binario	Izq. a Dch.
`&&`	Binario	Izq. a Dch.
`||`	Binario	Izq. a Dch.
`?:`	Ternario	Dch. a Izq.

Significado de los operadores:

- Aritméticos. El resultado es del mismo tipo que los operandos (véase 2.6):

`- valor`	Menos unario
`valor * valor`	Producto (multiplicación)
`valor / valor`	División (entera o real según el tipo de operandos)
`valor % valor`	Módulo (resto de la división) (sólo tipos enteros)
`valor + valor`	Suma
`valor - valor`	Resta

- Relacionales/Comparaciones. El resultado es de tipo `bool`

`valor < valor`	Comparación menor
`valor <= valor`	Comparación menor o igual
`valor > valor`	Comparación mayor
`valor >= valor`	Comparación mayor o igual
`valor == valor`	Comparación igualdad
`valor != valor`	Comparación desigualdad

- Operadores de Bits, sólo aplicable a operandos de tipos enteros. El resultado es del mismo tipo que los operandos (véase 2.6):

`~ valor`	Negación de bits (complemento)
`valor << despl`	Desplazamiento de bits a la izq.
`valor >> despl`	Desplazamiento de bits a la dch.
`valor & valor`	AND de bits
`valor ^ valor`	XOR de bits
`valor | valor`	OR de bits

- Lógicos, sólo aplicable operandos de tipo booleano. Tanto el operador `&&` como el operador `||` se evalúan en cortocircuito. El resultado es de tipo `bool`:

`! valor`	Negación lógica	(Si `valor` es `true` entonces `false`, en otro caso `true`)
`valor1 && valor2`	AND lógico	(Si `valor1` es `false` entonces `false`, en otro caso `valor2`)
`valor1 || valor2`	OR lógico	(Si `valor1` es `true` entonces `true`, en otro caso `valor2`)

x	! x
F	T
T	F

x	y	x && y	x || y
F	F	F	F
F	T	F	T
T	F	F	T
T	T	T	T

- Condicional. El resultado es del mismo tipo que los operandos:

`cond ? valor1 : valor2`	Si `cond` es `true` entonces `valor1`, en otro caso `valor2`

2.6. Conversiones Automáticas (Implícitas) de Tipos

Es posible que nos interese realizar operaciones en las que se mezclen datos de tipos diferentes. El lenguaje de programación C++ realiza *conversiones de tipo automáticas* ("castings"), de tal forma que el resultado de la operación sea del tipo más amplio de los implicados en ella. *Siempre que sea posible*, los valores se convierten de tal forma que no se pierda información.

Promociones Enteras

Son conversiones *implícitas* que preservan valores. Antes de realizar una operación aritmética, se utiliza *promoción a entero* para crear `int` a partir de otros tipos integrales mas cortos. Para los siguientes tipos origen: `bool`, `char`, `signed char`, `unsigned char`, `short`, `unsigned short`

- Si `int` puede representar todos los valores posibles del tipo origen, entonces sus valores promocionan a `int`.
- En otro caso, promocionan a `unsigned int`

Conversiones Enteras

Si el tipo destino es `unsigned`, el valor resultante es el menor `unsigned` congruente con el valor origen módulo 2^n, siendo n el número de bits utilizados en la representación del tipo destino (en representación de complemento a dos, simplemente tiene tantos bits del origen como quepan en el destino, descartando los de mayor orden).

Si el tipo destino es `signed`, el valor no cambia si se puede representar en el tipo destino, si no, viene definido por la implementación.

Conversiones Aritméticas Implícitas Habituales

Se realizan sobre los operandos de un operador binario para convertirlos a un tipo común que será el tipo del resultado:

1. Si algún operando es de tipo de punto flotante (real):
   a) Si algún operando es de tipo `long double`, el otro se convierte a `long double`.
   b) En otro caso, si algún operando es `double` el otro se convierte a `double`.
   c) En otro caso, si algún operando es `float` el otro se convierte a `float`.
2. En otro caso, se realizan promociones enteras (véase sec. 2.6) sobre ambos operandos:
   a) Si algún operando es de tipo `unsigned long`, el otro se convierte a `unsigned long`.
   b) En otro caso, si algún operando es `long int` y el otro es `unsigned int`, si un `long int` puede representar todos los valores de un `unsigned int`, el `unsigned int` se convierte a `long int`; en caso contrario, ambos se convierten a `unsigned long int`.
   c) En otro caso, si algún operando es `long` el otro se convierte a `long`.
   d) En otro caso, si algún operando es `unsigned` el otro se convierte a `unsigned`.
   e) En otro caso, ambos operandos son `int`

2.7. Conversiones Explícitas de Tipos

También es posible realizar *conversiones de tipo explícitas*. Para ello, se escribe el tipo al que queremos convertir y entre paréntesis la expresión cuyo valor queremos convertir. Por ejemplo:

```
char x = char(65) ;        produce el carácter 'A'
int x = int('a') ;         convierte el carácter 'a' a su valor entero (97)
int x = int(ROJO) ;        produce el entero 0
int x = int(AMARILLO) ;    produce el entero 2
int x = int(3.7) ;         produce el entero 3
double x = double(2) ;     produce el real (doble precisión) 2.0
Color x = Color(1) ;       produce el Color AZUL
Color x = Color(c+1) ;     si c es de tipo Color, produce el siguiente valor de la enumeración
```

El tipo enumerado se convierte automáticamente a entero, aunque la conversión inversa no se realiza de forma automática. Así, para incrementar una variable de tipo color se realizará de la siguiente forma:

```
enum Color {
    ROJO, AZUL, AMARILLO
} ;
int main()
{
    Color c = ROJO ;
    c = Color(c + 1) ;
    // ahora c tiene el valor AZUL
}
```

2.8. Tabla ASCII

La tabla ASCII es comúnmente utilizada como base para la representación de los caracteres, donde los números del 0 al 31 se utilizan para representar caracteres de control, y los números del 128 al 255 se utilizan para representar caracteres extendidos.

Rep	Simb	Rep	Simb	Rep	Simb	Rep	Simb
0	\0	32	SP	64	@	96	‘
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	\a	39	’	71	G	103	g
8	\b	40	(	72	H	104	h
9	\t	41	)	73	I	105	i
10	\n	42	*	74	J	106	j
11	\v	43	+	75	K	107	k
12	\f	44	,	76	L	108	l
13	\r	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL

2.9. Algunas Consideraciones Respecto a Operaciones con Números Reales

La representación de los números reales es limitada en cuanto al rango de valores y a la precisión de los mismos, ya que el número de dígitos decimales que puede almacenar es finito. Además, la representación de números reales en base 2 hace que la representación de determinados números en base 10 sea **inexacta**, por lo tanto la realización de operaciones aritméticas en punto flotante puede

dar lugar a pérdidas de precisión que den lugar a resultados inesperados, distintas operaciones que matemáticamente son equivalentes pueden ser computacionalmente diferentes. Así, la comparación directa de igualdad o desigualdad de números reales debe ser **evitada**. Por ejemplo, el siguiente programa:

```
#include <iostream>
using namespace std ;
int main()
{
    bool ok = (3.0 * (0.1 / 3.0)) == ((3.0 * 0.1) / 3.0) ;
    cout << "Resultado de (3.0 * (0.1 / 3.0)) == ((3.0 * 0.1) / 3.0): "
         << boolalpha << ok << endl ;
}
```

produce un resultado distinto a lo que cabría esperar:

```
Resultado de (3.0 * (0.1 / 3.0)) == ((3.0 * 0.1) / 3.0): false
```

El siguiente programa:

```
#include <iostream>
using namespace std ;
int main()
{
    double x, y ;
    double a, b ;
    cout << "Introduce 3 números reales: " ;
    cin >> a >> b >> y ;
    x = a * b ;
    bool cmp = (x == y) ;
    cout << a << " * " << b << " == " << y << " "
         << boolalpha << cmp
         << endl ;
}
```

produce los siguientes resultados en una máquina de 32 bits para los valores de `a`, `b` y `c`:

- a: 1, b: 1, c: 1 ⇒ `true`
- a: 2, b: 2, c: 4 ⇒ `true`
- a: 3, b: 3, c: 9 ⇒ `true`
- a: 3, b: 0.1, c: 0.3 ⇒ `false`
- a: 2, b: 0.2, c: 0.4 ⇒ `true`
- a: 3, b: 0.3, c: 0.9 ⇒ `false`
- a: 0.1, b: 0.1, c: 0.01 ⇒ `false`
- a: 0.2, b: 0.2, c: 0.04 ⇒ `false`
- a: 0.3, b: 0.3, c: 0.09 ⇒ `false`

Así, la comparación de igualdad entre números reales se debería sustituir por una comparación de proximidad, de tal forma que consideraremos que dos números reales (`x`, `y`) son *"iguales"* si están lo suficientemente próximos:

```
#include <iostream>
#include <cmath>
using namespace std ;
int main()
{
    double x, y ;
    double a, b ;
    cout << "Introduce 3 números reales: " ;
    cin >> a >> b >> y ;
    x = a * b ;
    bool cmp = fabs(x - y) < 1e-6 ;
    cout << a << " * " << b << " == " << y << " "
         << boolalpha << cmp
         << endl ;
}
```

Capítulo 3

Entrada y Salida de Datos Básica

La entrada y salida de datos permite a un programa recibir información desde el exterior (usualmente el teclado), la cual será transformada mediante un determinado procesamiento, y posteriormente permitirá mostrar al exterior (usualmente la pantalla del monitor) el resultado de la computación.

Para poder realizar entrada y salida de datos básica es necesario incluir la biblioteca `iostream` que contiene las declaraciones de tipos y operaciones que la realizan. Todas las definiciones y declaraciones de la biblioteca estándar se encuentran bajo el espacio de nombres `std` (ver capítulo 10), por lo que para utilizarlos adecuadamente habrá que utilizar la *directiva* `using` al comienzo del programa.

```
#include <iostream>   // inclusión de la biblioteca de entrada/salida
using namespace std ; // utilización del espacio de nombres de la biblioteca
const double EUR_PTS = 166.386 ;
int main()
{
   cout << "Introduce la cantidad (en euros): " ;
   double euros ;
   cin >> euros ;
   double pesetas = euros * EUR_PTS ;
   cout << euros << " Euros equivalen a " << pesetas << " Pts" << endl ;
}
```

3.1. Salida de Datos

La *salida* de datos permite mostrar información al exterior, y se realiza a través de los *flujos de salida*, así el flujo de salida asociado a la *salida estándar* (usualmente la pantalla o terminal de la consola) se denomina `cout`. De esta forma, la salida de datos a pantalla se realiza utilizando el operador `<<` sobre el flujo `cout` especificando el dato cuyo valor se mostrará. Por ejemplo:

```
cout << "Introduce la cantidad (en euros): " ;
```

escribirá en la salida estándar el mensaje correspondiente a la cadena de caracteres especificada. El siguiente ejemplo escribe en la salida estándar el valor de las variables `euros` y `pesetas`, así como un mensaje para interpretarlos adecuadamente. El símbolo `endl` indica que la sentencia deberá escribir un *fin de línea* (lo que se muestre a continuación se realizará en una nueva línea).

```
cout << euros << " Euros equivalen a " << pesetas << " Pts" << endl ;
```

Salida de Datos Formateada

Es posible especificar el formato bajo el que se realizará la salida de datos. Para ello se debe incluir la biblioteca estándar `iomanip`. Por ejemplo:

```
#include <iostream>
#include <iomanip>
using namespace std ;
int main()
{
    bool x = true ;
    cout << boolalpha << x ; // escribe los booleanos como 'false' o 'true'

    cout << dec << 27 ; // escribe 27 (decimal)
    cout << hex << 27 ; // escribe 1b (hexadecimal)
    cout << oct << 27 ; // escribe 33 (octal)

    cout << setprecision(2) << 4.567 ;       // escribe 4.6
    cout << setw(5) << 234 ;                 // escribe "  234"
    cout << setfill('#') << setw(5) << 234 ; // escribe "##234"
}
```

donde el manipulador `boolalpha` especifica que los valores lógicos se mostrarán mediante los valores `false` y `true`, y los manipuladores `dec`, `hex`, `oct` especifican que la salida se realizará utilizando el sistema de numeración *decimal*, *hexadecimal* o *octal* respectivamente. Por otra parte, el manipulador `setprecision(...)` especifica la cantidad de dígitos significativos (precisión) que se mostrará en la salida de números reales, el manipulador `setw(...)` especifica la anchura (*width*) que como mínimo ocupará la salida da datos (permite mostrar la información de forma tabulada), y el manipulador `setfill(...)` especifica el carácter de relleno (*fill*) que se utilizará en caso de ser necesario para ocupar toda la anchura del campo de salida.

3.2. Entrada de Datos

La *entrada* de datos permite recibir información desde el exterior, y se realiza a través de los *flujos de entrada*, así el flujo de entrada asociado a la *entrada estándar* (usualmente el teclado) se denomina `cin`. De esta forma, la entrada de datos desde el teclado se realiza mediante el operador `>>` sobre el flujo `cin` especificando la variable donde almacenar el valor de entrada leído desde el teclado:

```
cin >> euros ;
```

incluso es posible leer varios valores consecutivamente en la misma sentencia de entrada:

```
cin >> minimo >> maximo ;
```

Dicho operador de entrada se comporta de la siguiente forma: elimina los espacios en blanco que hubiera al principio de la entrada de datos, y lee dicha entrada hasta que encuentre algún carácter no válido para dicha entrada, que no será leído y permanecerá disponible en el buffer de entrada (véase 3.3) hasta la próxima operación de entrada. En caso de que durante la entrada surja alguna situación de error, dicha entrada se detiene y el flujo de entrada se pondrá en un estado erróneo. Se consideran espacios en blanco los siguientes caracteres: espacio en blanco, tabuladores, retorno de carro y nueva línea (`' '`, `'\t'`, `'\v'`, `'\f'`, `'\r'`, `'\n'`).

Ⓐ Entrada de Datos Avanzada

A diferencia del comportamiento especificado anteriormente, también es posible leer un carácter, desde el flujo de entrada, sin eliminar los espacios iniciales:

```
{
    char c ;
    cin.get(c) ; // lee un carácter sin eliminar espacios en blanco iniciales
    ...
}
```

En caso de querer eliminar los espacios iniciales explícitamente:

```
{
    char c ;
    cin >> ws ;  // elimina los espacios en blanco iniciales
    cin.get(c) ; // lee sin eliminar espacios en blanco iniciales
    ...
}
```

Es posible también eliminar un número determinado de caracteres del flujo de entrada, o hasta que se encuentre un determinado carácter:

```
{
    cin.ignore() ;           // elimina el próximo carácter
    cin.ignore(5) ;          // elimina los 5 próximos caracteres
    cin.ignore(1000, '\n') ; // elimina 1000 caracteres o hasta nueva-línea
}
```

La entrada y salida de cadenas de caracteres se puede ver en los capítulos correspondientes (cap. 6.2).

Ⓐ Control del Estado del Flujo

Cuando se realiza una entrada de datos errónea, el flujo de entrada se pondrá en un *estado de error*, de tal forma que cualquier operación de entrada de datos sobre un flujo de datos en estado erróneo también fallará. Por ejemplo, la ejecución del siguiente programa entrará en un "bucle sin fin" en caso de que se introduzca una letra en vez de un número, ya que el valor que tome la variable `n` no estará en el rango adecuado, y cualquier otra operación de entrada también fallará, por lo que el valor de `n` nunca podrá tomar un valor válido dentro del rango espcificado:

```
int main()
{
    int n = 0;
    do {
        cout << "Introduzca un numero entre [1..9]: ";
        cin >> n;
    } while (! (n > 0 && n < 10));
    cout << "Valor: " << n << endl;
}
```

Sin embargo, es posible comprobar el estado de un determinado flujo de datos, y en caso de que se encuentre en un estado de error, es posible restaurarlo a un estado correcto, por ejemplo:

```
int main()
{
    int n = 0;
    do {
        cout << "Introduzca un numero [1..9]: ";
        cin >> n;
        while (cin.fail()) {                    // ¿ Estado Erróneo ?
            cin.clear();                        // Restaurar estado
            cin.ignore(1000, '\n');             // Eliminar la entrada de datos anterior
            cout << "Error: Introduzca un numero [1..9]: ";
            cin >> n;
        }
    } while (! (n > 0 && n < 10));
    cout << "Valor: " << n << endl;
}
```

3.3. El "Buffer" de Entrada y el "Buffer" de Salida Ⓐ

Ningún dato de entrada o de salida en un programa C++ se obtiene o envía directamente del/al hardware, sino que se realiza a través de unas zonas de memoria intermedia (*"buffers"*) de entrada y salida respectivamente controlados por el Sistema Operativo y son independientes de nuestro programa.

Así, cuando se pulsa alguna tecla, el Sistema Operativo almacena en secuencia las teclas pulsadas en una zona de memoria intermedia: *el "buffer" de entrada*. Cuando un programa realiza una operación de entrada de datos (`cin >> valor`), accede al "buffer" de entrada y obtiene los valores allí almacenados si los hubiera, o esperará hasta que los haya (se pulsen una serie de teclas seguidas por la tecla "enter"). Una vez obtenidos las teclas pulsadas (caracteres), se convertirán a un valor del tipo especificado por la operación de entrada, y dicho valor se asignará a la variable especificada.

De igual forma, cuando se va a mostrar alguna información de salida dichos datos no van directamente a la pantalla, sino que se convierten a un formato adecuado para ser impresos (caracteres) y se almacenan en una zona de memoria intermedia denominada *"buffer" de salida*, desde donde el Sistema Operativo tomará la información para ser mostrada por pantalla.

```
cout << "Valor: " << val << endl ;
```

Capítulo 4

Estructuras de Control

Las estructuras de control en el lenguaje de programación C++ son muy flexibles, sin embargo, la excesiva flexibilidad puede dar lugar a estructuras complejas. Por ello sólo veremos algunas de ellas y utilizadas en contextos y situaciones restringidas.

4.1. Sentencia, Secuencia y Bloque

En C++ la unidad básica de acción es la sentencia, y expresamos la composición de sentencias como una *secuencia de sentencias* terminadas cada una de ellas por el carácter "punto y coma" (;), de tal forma que su flujo de *ejecución es secuencial*, es decir, se ejecuta una sentencia, y cuando ésta termina, entonces se ejecuta la siguiente sentencia, y así sucesivamente.

Un *bloque* es una unidad de ejecución mayor que la sentencia, y permite agrupar una secuencia de sentencias como una unidad. Para ello enmarcamos la secuencia de sentencias entre dos llaves para formar un bloque. Es posible el anidamiento de bloques.

```
int main()
{
    <sentencia_1> ;
    <sentencia_2> ;
    {
        <sentencia_3> ;
        <sentencia_4> ;
        . . .
    }
    <sentencia_n> ;
}
```

4.2. Declaraciones Globales y Locales

Como ya se vio en el capítulo anterior, es obligatoria la declaración de las entidades manipuladas en el programa. Distinguiremos dos clases de declaraciones: globales y locales.

Entidades globales son aquellas que han sido definidas fuera de cualquier bloque. Su *ámbito de visibilidad* comprende desde el punto en el que se definen hasta el final del fichero. Respecto a su tiempo de vida, se crean al principio de la ejecución del programa y se destruyen al finalizar éste. Normalmente serán constantes simbólicas, definiciones de tipos, declaración de prototipos de subprogramas y definiciones de subprogramas.

Entidades locales son aquellas que se definen dentro de un bloque. Su *ámbito de visibilidad* comprende desde el punto en el que se definen hasta el final de dicho bloque. Respecto a su tiempo de vida, se crean en el punto donde se realiza la definición, y se destruyen al finalizar el bloque. Normalmente serán constantes simbólicas y variables locales.

```
#include <iostream>
using namespace std ;
const double EUR_PTS = 166.386 ;       // Declaración de constante GLOBAL
int main()
{
   cout << "Introduce la cantidad (en euros): " ;
   double euros ;                      // Declaración de variable LOCAL
   cin >> euros ;
   double pesetas = euros * EUR_PTS ;  // Declaración de variable LOCAL
   cout << euros << " Euros equivalen a " << pesetas << " Pts" << endl ;
}
```

Respecto al ámbito de visibilidad de una entidad, en caso de declaraciones de diferentes entidades con el mismo identificador en diferentes niveles de anidamiento, la entidad visible será aquella que se encuentre declarada en el bloque de nivel de anidamiento más interno. Es decir, cuando se solapa el ámbito de visibilidad de dos entidades con el mismo identificador, en dicha zona de solapamiento será visible el identificador declarado o definido en el bloque más interno. Sin embargo, no es una buena práctica de programación ocultar identificadores al redefinirlos en niveles de anidamiento mas internos, ya que conduce a programas difíciles de leer y propensos a errores, por lo que procuraremos evitar esta práctica.

```
int main()
{
    int x = 3 ;
    int z = x * 2 ;        // x es vble de tipo int con valor 3
    {
        double x = 5.0 ;
        double n = x * 2 ;  // x es vble de tipo double con valor 5.0
    }
    int y = x + 4 ;        // x es vble de tipo int con valor 3
}
```

4.3. Sentencias de Asignación

La sentencia de asignación permite asignar a una variable el resultado de evaluar una expresión aritmética expresada en notación infija, de tal forma que primero se evalúa la expresión, considerando las reglas de precedencia y asociatividad de los operadores (véase 2.5), y a continuación el valor resultante se asigna a la variable, que perderá el valor anterior que tuviese.

```
{
    ...
    <variable> = <expresión_aritmética> ;
    ...
}
```

Por ejemplo:

```
const int MAXIMO = 15 ;
int main()
{
    int cnt ;               // valor inicial inespecificado
    cnt = 30 * MAXIMO + 1 ; // asigna a cnt el valor 451
    cnt = cnt + 10 ;        // cnt pasa ahora a contener el valor 461
}
```

Además, se definen las siguientes sentencias de incremento y decremento, así como su equivalencia en la siguiente tabla :

Sentencia	Equivalencia
`++variable;`	`variable = variable + 1;`
`--variable;`	`variable = variable - 1;`
`variable++;`	`variable = variable + 1;`
`variable--;`	`variable = variable - 1;`
`variable += expresion;`	`variable = variable + (expresion);`
`variable -= expresion;`	`variable = variable - (expresion);`
`variable *= expresion;`	`variable = variable * (expresion);`
`variable /= expresion;`	`variable = variable / (expresion);`
`variable %= expresion;`	`variable = variable % (expresion);`
`variable &= expresion;`	`variable = variable & (expresion);`
`variable ^= expresion;`	`variable = variable ^ (expresion);`
`variable |= expresion;`	`variable = variable | (expresion);`
`variable <<= expresion;`	`variable = variable << (expresion);`
`variable >>= expresion;`	`variable = variable >> (expresion);`

Nota: las sentencias de asignación vistas anteriormente se pueden utilizar en otras formas muy diversas, pero nosotros restringiremos su utilización a la expresada anteriormente, debido a que otras utilizaciones pueden dificultar la legibilidad y aumentar las posibilidades de cometer errores de programación.

4.4. Sentencias de Selección

Las sentencias de selección alteran el flujo secuencial de ejecución de un programa, de tal forma que permiten seleccionar flujos de ejecución alternativos y excluyentes dependiendo del valor que tomen determinadas expresiones lógicas. La más simple de todas es la sentencia de selección condicional `if` cuya sintaxis es la siguiente:

```
int main()
{
    if ( <expresión_lógica> ) {
        <secuencia_de_sentencias> ;
    }
}
```

y cuya semántica consiste en evaluar la expresión lógica, y si su resultado es Verdadero (`true`) entonces se ejecuta la secuencia de sentencias entre las llaves. Ejemplo de programa que imprime el valor mayor de tres números:

```
#include <iostream>
using namespace std ;
int main ()
{
    int a, b, c ;
    cin >> a >> b >> c ;
    int mayor = a ;
    if (b > mayor) {
        mayor = b ;
    }
    if (c > mayor) {
        mayor = c ;
    }
    cout << mayor << endl ;
}
```

Otra posibilidad es la sentencia de selección condicional compuesta, que tiene la siguiente sintaxis:

```
int main()
{
    if ( <expresión_lógica> ) {
        <secuencia_de_sentencias_v> ;
    } else {
        <secuencia_de_sentencias_f> ;
    }
}
```

y cuya semántica consiste en evaluar la expresión lógica, y si su resultado es Verdadero (`true`) entonces se ejecuta la *<secuencia_de_sentencias_v>*. Sin embargo, si el resultado de evaluar la expresión lógica es Falso (`false`) entonces se ejecuta la *<secuencia_de_sentencias_f>*.

La sentencia de selección condicional se puede encadenar de la siguiente forma con el flujo de control esperado:

```
#include <iostream>
using namespace std ;
int main ()
{
    double nota ;
    cin >> nota ;
    if ( ! ((nota >= 0.0) && (nota <= 10.0))) {
        cout << "Error: 0 <= n <= 10" << endl ;
    } else if (nota >= 9.5) {
        cout << "Matrícula de Honor" << endl ;
    } else if (nota >= 9.0) {
        cout << "Sobresaliente" << endl ;
    } else if (nota >= 7.0) {
        cout << "Notable" << endl ;
    } else if (nota >= 5.0) {
        cout << "Aprobado" << endl ;
    } else {
        cout << "Suspenso" << endl ;
    }
}
```

La sentencia `switch` es otro tipo de sentencia de selección en la cual la secuencia de sentencias alternativas a ejecutar no se decide en base a expresiones lógicas, sino en función del valor que tome una determinada expresión de tipo **ordinal** (véase 2.2), es decir, una relación de igualdad entre el valor de una expresión y unos determinados valores **constantes** de tipo ordinal especificados. Su sintaxis es la siguiente:

```
int main()
{
    switch ( <expresión> ) {
    case <valor_cte_1>:
        <secuencia_de_sentencias_1> ;
        break ;
    case <valor_cte_2>:
    case <valor_cte_3>:
        <secuencia_de_sentencias_2> ;
        break ;
    case <valor_cte_4>:
        <secuencia_de_sentencias_3> ;
        break ;
    . . .
    default:
        <secuencia_de_sentencias_d> ;
        break ;
    }
```

```
}
```

en la cual se evalúa la expresión, y si su valor coincide con *<valor_cte_1>* entonces se ejecuta la *<secuencia_de_sentencias_1>*. Si su valor coincide con *<valor_cte_2>* o con *<valor_cte_3>* se ejecuta la *<secuencia_de_sentencias_2>* y así sucesivamente. Si el valor de la expresión no coincide con ningún valor especificado, se ejecuta la secuencia de sentencias correspondiente a la etiqueta `default` (si es que existe). Nótese que la sentencia `break;` termina la secuencia de sentencias a ejecutar para cada caso, en caso de que falte, el comportamiento no será el deseado. Ejemplo:

```
#include <iostream>
using namespace std ;
int main ()
{
    int dia ;
    ...
    // 'dia' tiene algún valor válido
    switch (dia) {
    case 1:
        cout << "Lunes" << endl ;
        break ;
    case 2:
        cout << "Martes" << endl ;
        break ;
    case 3:
        cout << "Miércoles" << endl ;
        break ;
    case 4:
        cout << "Jueves" << endl ;
        break ;
    case 5:
        cout << "Viernes" << endl ;
        break ;
    case 6:
        cout << "Sábado" << endl ;
        break ;
    case 7:
        cout << "Domingo" << endl ;
        break ;
    default:
        cout << "Error" << endl ;
        break ;
    }
}
```

4.5. Sentencias de Iteración. Bucles

Vamos a utilizar tres tipos de sentencias diferentes para expresar la repetición de la ejecución de un grupo de sentencias: `while`, `for` y `do-while`.

La sentencia `while` es la más utilizada y su sintaxis es la siguiente:

```
int main()
{
    while ( <expresión_lógica> ) {
        <secuencia_de_sentencias> ;
    }
}
```

en la cual primero se evalúa la expresión lógica, y si es cierta, se ejecuta la secuencia de sentencias entre llaves completamente. Posteriormente se vuelve a evaluar la expresión lógica y si vuelve a ser cierta se vuelve a ejecutar la secuencia de sentencias entre llaves. Este ciclo iterativo consistente en evaluar la condición y ejecutar las sentencias se realizará *MIENTRAS* que la condición se evalúe a Verdadera y finalizará cuando la condición se evalúe a Falsa. Ejemplo:

```
#include <iostream>
using namespace std ;
int main ()
{
    int num, divisor ;
    cin >> num ;
    if (num <= 1) {
        divisor = 1 ;
    } else {
        divisor = 2 ;
        while ((num % divisor) != 0) {
            ++divisor ;
        }
    }
    cout << "El primer divisor de " << num << " es " << divisor << endl ;
}
```

La sentencia `for` es semejante a la estructura `FOR` de Pascal o Modula-2, aunque en C++ toma una dimensión más amplia y flexible. En realidad se trata de la misma construcción `while` vista anteriormente pero con una sintaxis diferente para hacer más explícito los casos en los que la iteración está controlada por los valores que toma una determinada variable de control, de tal forma que existe una clara inicialización y un claro incremento de la variable de control hasta llegar al caso final. La sintaxis es la siguiente:

```
int main()
{
    for ( <inicialización> ; <expresión_lógica> ; <incremento> ) {
        <secuencia_de_sentencias> ;
    }
}
```

y es equivalente a:

```
int main()
{
    <inicialización> ;
    while ( <expresión_lógica> ) {
        <secuencia_de_sentencias> ;
        <incremento> ;
    }
}
```

Nota: es posible y adecuado declarar e inicializar la variable de control del bucle en el lugar de la inicialización. En este caso especial, el ámbito de visibilidad de la variable de control del bucle es solamente hasta el final del bloque de la estructura `for`.

```
#include <iostream>
using namespace std ;
int main ()
{
    int n ;
    cin >> n ;
    for (int i = 0 ; i < n ; ++i) {
        cout << i << " " ;
```

```
    }
    // i ya no es visible aquí
    cout << endl ;
}
```

La sentencia `do-while` presenta la siguiente estructura

```
int main()
{
    do {
        <secuencia_de_sentencias> ;
    } while ( <expresión_lógica> ) ;
}
```

también expresa la iteración en la ejecución de la secuencia de sentencias, pero a diferencia de la primera estructura iterativa ya vista, donde primero se evalúa la expresión lógica y después, en caso de ser cierta, se ejecuta la secuencia de sentencias, en esta estructura se ejecuta primero la secuencia de sentencias y posteriormente se evalúa la expresión lógica, y si ésta es cierta se repite el proceso.

```
#include <iostream>
using namespace std;
int main ()
{
    int num ;
    do {
        cin >> num ;
    } while ((num % 2) != 0) ;
    cout << "El número par es " << num << endl ;
}
```

4.6. Programación Estructurada

Un programa sigue una metodología de programación estructurada si todas las estructuras de control que se utilizan (secuencia, selección, iteración y modularización) tienen un único punto de entrada y un único punto de salida. Esta característica hace posible que se pueda aplicar la abstracción para su diseño y desarrollo. La abstracción se basa en la identificación de los elementos a un determinado nivel, ignorando los detalles especificados en niveles inferiores. Un algoritmo que use tan sólo las estructuras de control tratadas en este tema, se denomina estructurado.

Bohm y Jacopini demostraron que todo problema computable puede resolverse usando únicamente estas estructuras de control. Ésta es la base de la programación estructurada. La abstracción algorítmica va un paso más allá y considera que cada nivel de refinamiento corresponde con un subprograma independiente

4.7. Ejemplos

Ejemplo 1

Programa que multiplica dos números mediante sumas acumulativas:

```
#include <iostream>
using namespace std ;
int main ()
{
    cout << "Introduzca dos números: " ;
    int m, n ;
    cin >> m >> n ;
    // Sumar: m+m+m+...+m (n veces)
```

```
    int total = 0 ;
    for (int i = 0 ; i < n ; ++i) {
        // Proceso iterativo: acumular el valor de 'm' al total
        total = total + m ;     // total += m ;
    }
    cout << total << endl ;
}
```

Ejemplo 2

Programa que calcula el factorial de un número dado:

```
#include <iostream>
using namespace std ;
int main ()
{
    cout << "Introduzca un número: " ;
    int n ;
    cin >> n ;
    // Multiplicar: 1 2 3 4 5 6 7 ... n
    int fact = 1 ;
    for (int i = 2 ; i <= n ; ++i) {
        // Proceso iterativo: acumular el valor de 'i' al total
        fact = fact * i ;       // fact *= i ;
    }
    cout << fact << endl ;
}
```

Ejemplo 3

Programa que divide dos números mediante restas sucesivas:

```
#include <iostream>
using namespace std ;
int main ()
{
    cout << "Introduzca dos números: " ;
    int dividendo, divisor ;
    cin >> dividendo >> divisor ;
    if (divisor == 0) {
        cout << "El divisor no puede ser cero" << endl ;
    } else {
        int resto = dividendo ;
        int cociente = 0 ;
        while (resto >= divisor) {
            resto -= divisor ;
            ++cociente ;
        }
        cout << cociente << " " << resto << endl ;
    }
}
```

Capítulo 5

Subprogramas. Funciones y Procedimientos

La abstracción es una herramienta mental que nos permite analizar, comprender y construir sistemas complejos. Así, identificamos y denominamos conceptos abstractos y aplicamos refinamientos sucesivos hasta comprender y construir el sistema completo.

Los subprogramas constituyen una herramienta del lenguaje de programación que permite al programador aplicar explícitamente la abstracción al diseño y construcción del software, proporcionando abstracción algorítmica (procedimental) a la construcción de programas.

Los subprogramas pueden ser vistos como un *mini* programa encargado de resolver algorítmicamente un subproblema que se encuentra englobado dentro de otro mayor. En ocasiones también pueden ser vistos como una ampliación o elevación del conjunto de operaciones básicas (acciones primitivas) del lenguaje de programación, proporcionándole nuevos mecanismos para resolver nuevos problemas.

5.1. Funciones y Procedimientos

Los subprogramas codifican la solución algorítmica a un determinado problema, de tal forma que cuando es necesario resolver dicho problema en un determinado momento de la computación, se invocará a una instancia del subprograma (mediante una llamada al subprograma) para que resuelva el problema para unos determinados parámetros. Así, en la invocación al subprograma, se transfiere la información que necesita para resolver el problema, entonces se ejecuta el código del subprograma que resolverá el problema y produce unos resultados que devuelve al lugar donde ha sido requerido. Podemos distinguir dos tipos de subprogramas:

- ***Procedimientos***: encargados de resolver un problema computacional general. En el siguiente ejemplo el procedimiento `ordenar` ordena los valores de las variables pasadas como parámetros (`x` e `y`), de tal forma que cuando termine el procedimiento, las variables `x` e `y` tendrán el menor y el mayor valor respectivamente de los valores originales:

```
int main()
{
    int x = 8 ;
    int y = 4 ;
    ordenar(x, y) ;
    cout << x << " " << y << endl ;
}
```

- ***Funciones***: encargadas de realizar un cálculo computacional y generar un único resultado, normalmente calculado en función de los datos recibidos. En el siguiente ejemplo, la función `calcular_menor` calcula (y devuelve) el menor valor de los dos valores recibidos como parámetros:

```
int main()
{
    int x = 8 ;
    int y = 4 ;
    int z = calcular_menor(x, y) ;
    cout << "Menor: " << z << endl ;
}
```

La llamada (invocación) a un subprograma se realiza mediante el nombre seguido por los parámetros actuales entre paréntesis, considerando que:

- La llamada a un procedimiento constituye por sí sola una sentencia independiente que puede ser utilizada como tal en el cuerpo de otros subprogramas (y del programa principal).
- La llamada a una función **no** constituye por sí sola una sentencia, por lo que debe aparecer dentro de alguna sentencia que utilice el valor resultado de la función.

5.2. Definición de Subprogramas

Los subprogramas codifican la solución algorítmica parametrizada a un determinado problema, es decir, especifica la secuencia de acciones a ejecutar para resolver un determinado problema dependiendo de unos determinados parámetros formales. Donde sea necesaria la resolución de dicho problema, se invocará a una instancia del subprograma para unos determinados parámetros actuales.

Considerando la norma de C++ de que antes de utilizar una determinada entidad en un programa, esta entidad deberá estar previamente declarada, normalmente deberemos definir los subprogramas en una posición previa a donde sean utilizados. No obstante, esta disposición de los subprogramas puede ser alterada como se indica en la sección 5.7. La definición de los subprogramas presentados anteriormente podría ser como se indica a continuación:

```
#include <iostream>
using namespace std ;
int calcular_menor(int a, int b)
{
    int menor ;
    if (a < b) {
        menor = a ;
    } else {
        menor = b ;
    }
    return menor ;
}
void ordenar(int& a, int& b)
{
    if (a > b) {
        int aux = a ;
        a = b ;
        b = aux ;
    }
}
int main()
{
    int x = 8 ;
    int y = 4 ;
    int z = calcular_menor(x, y) ;
    cout << "Menor: " << z << endl ;
    ordenar(x, y) ;
    cout << x << " " << y << endl ;
}
```

La definición de un subprograma comienza con un encabezamiento en la que se especifica en primer lugar el tipo del valor devuelto por éste si es una función, o `void` en caso de ser un procedimiento. A continuación vendrá el nombre del subprograma seguido por la declaración de sus parámetros formales entre paréntesis.

Los parámetros formales especifican como se realiza la transferencia de información entre el (sub)programa llamante y el subprograma llamado (véase 5.4). Normalmente la solución de un subproblema dependerá del valor de algunos datos, modificará el valor de otros datos, y posiblemente generará nuevos valores. Todo este flujo de información se realiza a través de los parámetros del subprograma.

El cuerpo del subprograma especifica la secuencia de acciones a ejecutar necesarias para resolver el subproblema especificado, y podrá definir tantas variables locales como necesite para desempeñar su misión. En el caso de una función, el valor que devuelve (el valor que toma tras la llamada) vendrá dado por el resultado de evaluar la expresión de la sentencia `return`. Aunque C++ es más flexible, nosotros sólo permitiremos una única utilización de la sentencia `return` y deberá ser al final del cuerpo de la función. Así mismo, un procedimiento no tendrá ninguna sentencia `return` en su cuerpo.

5.3. Ejecución de Subprogramas

Cuando se produce una llamada (invocación) a un subprograma:

1. Se establecen las vías de comunicación entre los algoritmos llamante y llamado por medio de los parámetros.
2. Posteriormente el flujo de ejecución pasa a ejecutar la primera sentencia del cuerpo del subprograma llamado, ejecutándose éste.
3. Cuando sea necesario, se crean las variables locales especificadas en el cuerpo de la definición del subprograma.
4. Cuando finaliza la ejecución del subprograma, las variables locales y parámetros previamente creados se destruyen, el flujo de ejecución retorna al (sub)programa llamante, y continúa la ejecución por la sentencia siguiente a la llamada realizada.

5.4. Paso de Parámetros. Parámetros por Valor y por Referencia

Todo el intercambio y transferencia de información entre el programa llamante y el subprograma llamado se debe realizar a través de los parámetros. Los *parámetros formales* son los que aparecen en la definición del subprograma, mientras que los *parámetros actuales* son los que aparecen en la llamada (invocación) al subprograma.

- Denominamos *parámetros de entrada* a aquellos parámetros que se utilizan para recibir la información necesaria para realizar una computación. Por ejemplo los parámetros `a` y `b` de la función `calcular_menor` anterior.
  - Los parámetros de entrada se definen mediante *paso por valor* (cuando son de tipos simples), que significa que los parámetros formales son variables independientes que toman sus valores iniciales como *copias* de los valores de los parámetros actuales de la llamada en el momento de la invocación al subprograma. Se declaran especificando el tipo y el identificador asociado.

```
int calcular_menor(int a, int b)
{
    int menor ;
    if (a < b) {
```

```
        menor = a ;
    } else {
        menor = b ;
    }
    return menor ;
}
```

- Cuando los parámetros de entrada son de tipos compuestos (véase 6), entonces se definen mediante *paso por referencia constante* que será explicado más adelante.

- Denominamos *parámetros de salida* a aquellos parámetros que se utilizan para transferir al programa llamante información producida como parte de la computación/solución realizada por el subprograma.

  - Los parámetros de salida se definen mediante *paso por referencia* que significa que el parámetro formal es una referencia a la variable que se haya especificado como parámetro actual de la llamada en el momento de la invocación al subprograma. Es decir, cualquier acción dentro del subprograma que se haga sobre el parámetro formal es equivalente a que se realice sobre la variable referenciada que aparece como parámetro actual en la llamada al subprograma. Se declaran especificando el tipo, el símbolo "ampersand" (`&`) y el identificador asociado.

    En el siguiente ejemplo, el procedimiento dividir recibe dos valores sobre los cuales realizará la operación de división (`dividendo` y `divisor` son parámetros de entrada y son pasados por valor), y devuelve dos valores como resultado de la división (`cociente` y `resto` son parámetros de salida y son pasados por referencia):

```
void dividir(int dividendo, int divisor, int& coc, int& resto)
{
    coc = dividendo / divisor ;
    resto = dividendo % divisor ;
}
int main()
{
    int cociente ;
    int resto ;

    dividir(7, 3, cociente, resto) ;
    // ahora 'cociente' valdrá 2 y 'resto' valdrá 1
}
```

- Denominamos *parámetros de entrada/salida* a aquellos parámetros que se utilizan para recibir información necesaria para realizar la computación, y que tras ser modificada se transfiere al lugar de llamada como parte de la información producida resultado de la computación del subprograma. Por ejemplo los parámetros `a` y `b` del procedimiento `ordenar` anterior.

  Los parámetros de entrada/salida se definen mediante *paso por referencia* y se declaran como se especificó anteriormente.

```
void ordenar(int& a, int& b)
{
    if (a > b) {
        int aux = a ;
        a = b ;
        b = aux ;
    }
}
```

	Tipos	
	Simples	Compuestos
(⇓) Entrada	P.Valor `(int x)`	P.Ref.Cte `(const Persona& x)`
(⇑) Salida, (⇕) E/S	P.Ref `(int& x)`	P.Ref `(Persona& x)`

En la llamada a un subprograma, se deben cumplir los siguientes requisitos:

- El número de parámetros actuales debe coincidir con el número de parámetros formales.
- La correspondencia entre parámetros actuales y formales es posicional.
- El tipo del parámetro actual debe coincidir con el tipo del correspondiente parámetro formal.
- Un parámetro formal de salida o entrada/salida (paso por referencia) requiere que el parámetro actual sea una variable.
- Un parámetro formal de entrada (paso por valor o paso por referencia constante) requiere que el parámetro actual sea una variable, constante o expresión.

	Tipos Simples		Tipos Compuestos	
Parámetro Formal	(⇓) Ent P.Valor `(int x)`	(⇑) Sal (⇕) E/S P.Referencia `(int& x)`	(⇓) Ent P.Ref.Constante `(const Persona& x)`	(⇑) Sal (⇕) E/S P.Referencia `(Persona& x)`
Parámetro Actual	Constante Variable Expresión	Variable	Constante Variable Expresión	Variable

5.5. Criterios de Modularización

No existen métodos objetivos para determinar como descomponer la solución de un problema en subprogramas, es una labor subjetiva. No obstante, se siguen algunos criterios que pueden guiarnos para descomponer un problema y modularizar adecuadamente. El diseñador de software debe buscar un **bajo acoplamiento** entre los subprogramas y una **alta cohesión** dentro de cada uno.

- Acoplamiento: Un objetivo en el diseño descendente es crear subprogramas aislados e independientes. Sin embargo, debe haber alguna conexión entre los subprogramas para formar un sistema coherente. A dicha conexión se conoce como acoplamiento. Por lo tanto, maximizar la independencia entre subprogramas será minimizar el acoplamiento.
- Cohesión: Hace referencia al grado de relación entre las diferentes partes internas dentro de un mismo subprograma. Si la cohesión es muy débil, la diversidad entre las distintas tareas realizadas dentro del subprograma es tal que posteriores modificaciones podrán resultar complicadas. Se busca maximizar la cohesión dentro de cada subprograma

Si no es posible analizar y comprender un subprograma de forma aislada e independiente del resto, entonces podemos deducir que la división modular no es la más adecuada.

5.6. Subprogramas "en Línea" Ⓐ

La llamada a un subprograma conlleva un pequeño coste debido al control y gestión de la misma que ocasiona cierta pérdida de tiempo de ejecución.

Hay situaciones en las que el subprograma es tan pequeño que el coste asociado a la invocación es superior al coste asociado a computar la solución del mismo, de tal forma que en estas situaciones interesa eliminar el coste asociado a su invocación. En ese caso se puede especificar que el subprograma se traduzca como *código en línea* en vez de como una llamada a un subprograma. Para ello

se especificará la palabra reservada `inline` justo antes del tipo. De esta forma, se mantiene los beneficios proporcionados por la abstracción, pero se eliminan los costes asociados a la invocación.

```
inline int calcular_menor(int a, int b)
{
    int menor ;
    if (a < b) {
        menor = a ;
    } else {
        menor = b ;
    }
    return menor ;
}
```

Este mecanismo sólo es adecuado cuando el cuerpo del subprograma es muy pequeño, de tal forma que el coste asociado a la invocación dominaría respecto a la ejecución del cuerpo del mismo.

5.7. Declaración de Subprogramas. Prototipos Ⓐ

Los subprogramas, al igual que los tipos, constantes y variables, deben ser declarados antes de ser utilizados. Dicha declaración se puede realizar de dos formas: una de ellas consiste simplemente en definir el subprograma antes de utilizarlo. La otra posibilidad consiste en declarar el *prototipo* del subprograma antes de su utilización, y definirlo posteriormente. El ámbito de visibilidad del subprograma será global al fichero, es decir, desde el lugar donde ha sido declarado hasta el final del fichero. Para declarar un subprograma habrá que especificar el tipo del valor devuelto (o void si es un procedimiento) seguido por el nombre y la declaración de los parámetros formales igual que en la definición del subprograma, pero sin definir el cuerpo del mismo. En lugar de ello se terminará la declaración con el carácter "punto y coma" (`;`).

```
int calcular_menor(int a, int b) ;  // prototipo de 'calcular_menor'
int main()
{
    int x = 8 ;
    int y = 4 ;
    int z = calcular_menor(x, y) ;
}
int calcular_menor(int a, int b)   // definición de 'calcular_menor'
{
    int menor ;
    if (a < b) {
        menor = a ;
    } else {
        menor = b ;
    }
    return menor ;
}
```

5.8. Sobrecarga de Subprogramas Ⓐ

Se denomina sobrecarga cuando distintos subprogramas se denominan con el mismo identificador u operador, pero se aplican a parámetros distintos. En el lenguaje de programación C++ es posible sobrecargar tanto subprogramas como operadores siempre y cuando tengan parámetros diferentes, y el compilador pueda discriminar entre ellos por la especificación de la llamada.

```
void imprimir(int x)
{
    cout << "entero: " << x << endl ;
```

```
}
void imprimir(double x)
{
    cout << "real: " << x << endl ;
}

inline double media(int x, int y, int z)
{
    return double(x + y + z) / 3.0 ;
}
inline double media(int x, int y)
{
    return double(x + y) / 2.0 ;
}
```

5.9. Pre-Condiciones y Post-Condiciones Ⓐ

- Pre-condición es un enunciado que debe ser cierto antes de la llamada a un subprograma. Especifica las condiciones bajo las cuales se ejecutará dicho subprograma.

- Post-condición es un enunciado que debe ser cierto tras la ejecución de un subprograma. Especifica el comportamiento de dicho subprograma.

- Codificar las pre/post-condiciones mediante asertos proporciona una valiosa documentación, y tiene varias ventajas:

  - Hace al programador explícitamente consciente de los prerrequisitos y del objetivo del subprograma.
  - Durante la depuración, las pre-condiciones comprueban que la llamada al subprograma se realiza bajo condiciones validas.
  - Durante la depuración, las post-condiciones comprueban que el comportamiento del subprograma es adecuado.
  - Sin embargo, a veces no es posible codificarlas fácilmente.

En C++, las pre-condiciones y post-condiciones se pueden especificar mediante asertos, para los cuales es necesario incluir la biblioteca `cassert`. Por ejemplo:

```
#include <iostream>
#include <cassert>
using namespace std ;
//---------------------------
void dividir(int dividendo, int divisor, int& cociente, int& resto)
{
    assert(divisor != 0) ;                              // PRE-CONDICIÓN
    cociente = dividendo / divisor ;
    resto = dividendo % divisor ;
    assert(dividendo == (divisor * cociente + resto)) ; // POST-CONDICIÓN
}
```

Nota: en *GNU GCC* es posible desactivar la comprobación de asertos mediante la siguiente directiva de compilación:

```
g++ -DNDEBUG -ansi -Wall -Werror -o programa programa.cpp
```

5.10. Ejemplos

Ejemplo 1

Ejemplo de un programa que imprime los números primos existentes entre dos valores leídos por teclado:

```
//- fichero: primos.cpp -----------------------------------------
#include <iostream>
using namespace std ;
void ordenar(int& menor, int& mayor)
{
   if (mayor < menor) {
       int aux = menor ;
       menor = mayor ;
       mayor = aux ;
   }
}
inline bool es_divisible(int x, int y)
{
   return ( x % y == 0 ) ;
}
bool es_primo(int x)
{
   int i ;
   for (i = 2 ; ((i <= x/2) && ( ! es_divisible(x, i))) ; ++i) {
       // vacío
   }
   return (i == x/2+1) ;
}
void primos(int min, int max)
{
   cout << "Números primos entre " << min << " y " << max << endl ;
   for (int i = min ; i <= max ; ++i) {
       if (es_primo(i)) {
           cout << i << " " ;
       }
   }
   cout << endl ;
}
int main()
{
   int min, max ;
   cout << "Introduzca el rango de valores " ;
   cin >> min >> max ;
   ordenar(min, max) ;
   primos(min, max) ;
}
//- fin: primos.cpp ------------------------------------------------
```

Ejemplo 2

Ejemplo de un programa que convierte grados sexagesimales a radianes:

```
//- fichero: gradrad.cpp ------------------------------------------
#include <iostream>
#include <string>
using namespace std ;
// -- Constantes -------
const double PI = 3.1416 ;
```

```
const int PI_GRAD = 180 ;
const int MIN_GRAD = 60 ;
const int SEG_MIN = 60 ;
const int SEG_GRAD = SEG_MIN * MIN_GRAD ;
// -- Subalgoritmos ----
void leer_grados (int& grad, int& min, int& seg)
{
    cout << "Grados, minutos y segundos " ;
    cin >> grad >> min >> seg ;
}
//---------------------------
void escribir_radianes (double rad)
{
    cout << "Radianes: " << rad << endl ;
}
//---------------------------
double calc_rad (double grad_tot)
{
    return (grad_tot * PI) / double(PI_GRAD) ;
}
//---------------------------
double calc_grad_tot (int grad, int min, int seg)
{
    return double(grad) + (double(min) / double(MIN_GRAD)) + (double(seg) / double(SEG_GRAD)) ;
}
//---------------------------
double transf_gr_rad (int grad, int min, int seg)
{
    double gr_tot = calc_grad_tot(grad, min, seg) ;
    return calc_rad(gr_tot) ;
}
// -- Principal --------
int main ()
{
    int grad, min, seg ;
    leer_grados(grad, min, seg) ;
    double rad = transf_gr_rad(grad, min, seg) ;
    escribir_radianes(rad) ;
}
```

Capítulo 6

Tipos Compuestos

Los *tipos compuestos* surgen de la composición y/o agregación de otros tipos para formar nuevos tipos de mayor entidad. Existen dos formas fundamentales para crear tipos de mayor entidad: la composición de elementos, que denominaremos "Registros" o "Estructuras" y la agregación de elementos del mismo tipo, y se conocen como "Agregados", "Arreglos" o mediante su nombre en inglés "Arrays". Además de los tipos compuestos definidos por el programador mencionados anteriormente, los lenguajes de programación suelen proporcionar algún tipo adicional para representar las "cadenas de caracteres".

6.1. Paso de Parámetros de Tipos Compuestos

Los lenguajes de programación normalmente utilizan el *paso por valor* y el *paso por referencia* para implementar la transferencia de información entre subprogramas descrita en el interfaz. Para la transferencia de información de *entrada*, el paso por valor supone *duplicar y copiar el valor* del parámetro actual en el formal. En el caso de tipos *simples*, el paso por valor es adecuado para la transferencia de información de *entrada*, sin embargo, si el tipo de dicho parámetro es *compuesto*, es posible que dicha copia implique una *alta sobrecarga*, tanto en espacio de memoria como en tiempo de ejecución. El lenguaje de programación *C++* permite realizar de forma eficiente la transferencia de información de **entrada** para tipos compuestos mediante el *paso por referencia constante*.

Así, en el *paso por referencia constante* el parámetro formal es una referencia al parámetro actual especificado en la llamada, tomando así su valor, pero no puede ser modificado al ser una referencia constante, evitando de esta forma la semántica de salida asociada al paso por referencia. El paso por referencia constante suele utilizarse para el paso de parámetros de entrada con tipos compuestos, ya que evita la duplicación de memoria y la copia del valor, que en el caso de tipos compuestos suele ser costosa. Para ello, los parámetros se declaran como se especificó anteriormente para el paso por referencia, pero anteponiendo la palabra reservada `const`.

```
void imprimir(const Fecha& fech)
{
    cout << fech.dia << (int(fech.mes)+1) << fech.anyo << endl ;
}
```

	Tipos	
	Simples	Compuestos
(⇓) Entrada	P.Valor `(int x)`	P.Ref.Cte `(const Persona& p)`
(⇑) Salida, (⇕) E/S	P.Ref `(int& x)`	P.Ref `(Persona& p)`

Funciones que Retornan Tipos Compuestos

Por la misma razón y como *norma general*, salvo excepciones, tampoco es adecuado que una *función* retorne un valor de tipo compuesto, debido a la sobrecarga que generalmente ésto conlleva. En estos casos, suele ser más adecuado que el subprograma devuelva el valor de tipo compuesto como un parámetro de salida mediante el paso por referencia.

6.2. Cadenas de Caracteres en C++: el Tipo String

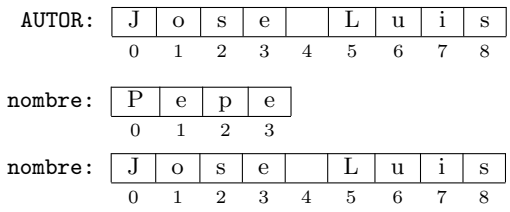
Las cadenas de caracteres representan una sucesión o secuencia de caracteres. Es un tipo de datos muy versátil, y es útil para representar información muy diversa:

- Información textual (caracteres)
- Entrada de datos y salida de resultados en forma de secuencia de caracteres.
- Información abstracta por medio de una secuencia de caracteres

Es posible utilizar el tipo `string` de la biblioteca estándar para representar cadenas de caracteres de *longitud finita* limitada por la implementación. Para ello, se debe incluir la biblioteca estándar `<string>`, así como utilizar el espacio de nombres de `std`. La definición de cadenas de caracteres mediante el tipo `string` permite definir cadenas de caracteres mas robustas y con mejores características que las cadenas de caracteres predefinidas al estilo-C (arrays de caracteres).

Es posible definir tanto constantes simbólicas como variables y parámetros de tipo `string`. Una cadena de caracteres literal se representa mediante una sucesión de caracteres entre comillas dobles. Así mismo, también es posible la asignación de cadenas de caracteres:

```
#include <iostream>
#include <string>
using namespace std ;
const string AUTOR = "José Luis" ;
int main()
{
    string nombre = "Pepe" ;
    // ...
    nombre = AUTOR ;
}
```



Si no se le asigna un valor inicial a una variable de tipo `string`, entonces la variable tendrá como valor por defecto la cadena vacía (`""`).

Entrada y Salida de Cadenas de Caracteres

El operador `<<` aplicado a un flujo de salida (`cout` para el flujo de salida estándar, usualmente el terminal) permite mostrar el contenido de las cadenas de caracteres, tanto constantes como variables. Por ejemplo:

```
#include <iostream>
#include <string>
using namespace std ;
const string AUTOR = "José Luis" ;
int main()
{
    string nombre = "Pepe" ;
    cout << "Nombre: " << nombre << " " << AUTOR << endl ;
}
```

El operador `>>` aplicado a un flujo de entrada (`cin` para el flujo de entrada estándar, usualmente el teclado) permite leer secuencias de caracteres y almacenarlas en variables de tipo `string`. Por ejemplo:

```
#include <iostream>
#include <string>
using namespace std ;
int main()
{
    cout << "Introduzca el nombre: " ;
    string nombre ;
    cin >> nombre ;
    cout << "Nombre: " << nombre << endl ;
}
```

Este operador de entrada (`>>`) se comporta (como se especificó en el capítulo 3.2 dedicado a la Entrada y Salida básica) de la siguiente forma: elimina los espacios en blanco que hubiera al principio de la entrada de datos, y lee dicha entrada hasta que encuentre algún carácter de espacio en blanco, que no será leído y permanecerá en el buffer de entrada (véase 3.3) hasta la próxima operación de entrada. En caso de que durante la entrada surja alguna situación de error, dicha entrada se detiene y el flujo de entrada se pondrá en un estado erróneo. Se consideran espacios en blanco los siguientes caracteres: espacio en blanco, tabuladores, retorno de carro y nueva línea (`' '`, `'\t'`, `'\v'`, `'\f'`, `'\r'`, `'\n'`).

También es posible leer una línea completa, hasta leer el carácter de fin de línea (ENTER), desde el flujo de entrada, sin eliminar los espacios iniciales:

```
#include <iostream>
#include <string>
using namespace std ;
int main()
{
    cout << "Introduzca el nombre: " ;
    string nombre ;
    getline(cin, nombre) ;
    cout << "Nombre: " << nombre << endl ;
}
```

También es posible leer una línea completa, hasta leer un delimitador especificado, desde el flujo de entrada, sin eliminar los espacios iniciales:

```
#include <iostream>
#include <string>
using namespace std ;
const char DELIMITADOR = '.' ;
int main()
{
    cout << "Introduzca el nombre: " ;
    string nombre ;
    getline(cin, nombre, DELIMITADOR) ;
    cout << "Nombre: " << nombre << endl ;
}
```

Nótese que realizar una operación `getline` después de una operación con `>>` puede tener complicaciones, ya que `>>` dejara los espacios en blanco (y fin de línea) en el buffer, que serán leídos por `getline`. Por ejemplo:

```
#include <iostream>
#include <string>
using namespace std ;
int main()
{
    //-----------------------------------------------------
    cout << "Introduzca número: " ;
    int n ;
```

```
    cin >> n ;
    //-----------------------------------------------------
    cout << "Introduzca el nombre: " ;
    string nombre ;
    getline(cin, nombre) ;
    //-----------------------------------------------------
    cout << "Número: " << n << " Nombre: [" << nombre << "]" << endl ;
}
```

Para evitar este problema, eliminaremos los caracteres de espacios en blanco (y fin de línea) del buffer de entrada antes de leer la entrada de datos con `getline(...)`, de tal forma que leerá una secuencia de caracteres que sea distinta de la vacía:

```
#include <iostream>
#include <string>
using namespace std ;
int main()
{
    //-----------------------------------------------------
    cout << "Introduzca número: " ;
    int n ;
    cin >> n ;
    //-----------------------------------------------------
    cout << "Introduzca el nombre (NO puede ser vacío): " ;
    string nombre ;
    ent >> ws ;            // salta los espacios en blanco y fin de línea
    getline(cin, nombre) ; // leerá la primera línea no vacía
    //-----------------------------------------------------
    cout << "Número: " << n << " Nombre: " << nombre << endl ;
}
```

Por el contrario, en caso de que la cadena vacía sea una entrada válida posible, entonces será necesario eliminar el resto de caracteres (incluyendo los espacios en blanco y fin de línea) del buffer de entrada, después de leer un dato con `>>`, de tal forma que el buffer esté limpio antes de realizar la entrada de la cadena de caracteres con `getline`. Por ejemplo:

```
#include <iostream>
#include <string>
#include <limits>
using namespace std ;
int main()
{
    //-----------------------------------------------------
    cout << "Introduzca número: " ;
    int n ;
    cin >> n ;
    cin.ignore(10000, '\n') ; // elimina todos los caracteres del buffer hasta '\n'
    //-----------------------------------------------------
    cout << "Introduzca el nombre (puede ser vacío): " ;
    string nombre ;
    getline(cin, nombre) ;
    //-----------------------------------------------------
    cout << "Número: " << n << " Nombre: " << nombre << endl ;
}
```

Operaciones con Cadenas de Caracteres

- Las cadenas de caracteres se pueden asignar a variables de dicho tipo. Por ejemplo:

```
#include <iostream>
```

```
#include <string>
using namespace std ;
const string AUTOR = "José Luis" ;
int main()
{
    string nombre = "Pepe" ;
    // ...
    nombre = AUTOR ;
}
```

- Es posible realizar la comparación lexicográfica[1] entre cadenas de caracteres del tipo `string` mediante los operadores relacionales (`==`, `!=`, `>`, `>=`, `<`, `<=`). Por ejemplo:
  - `if (nombre >= AUTOR) { /*...*/ }`
- Es posible la concatenación de cadenas y caracteres mediante los operadores de concatenación (`+`, `+=`):

```
#include <iostream>
#include <string>
using namespace std ;
const string AUTOR = "José Luis" ;
int main ()
{
    string nombre = AUTOR + "López" ;
    nombre += "Vázque" ;
    nombre += 'z' ;
    nombre = AUTOR + 's' ;
}
```

- Para acceder al número de caracteres que componen la cadena:
  - `unsigned ncar = nombre.size();`
- Comprobar si la cadena de caracteres está vacía:
  - `if (nombre.size() == 0) { /*...*/ }`
- Para acceder al i-ésimo carácter de la cadena (de tipo `char`):
  - `char c = nombre[i];` donde `i` ∈ `[0..nombre.size()-1]`
  - `nombre[i] = 'z';` donde `i` ∈ `[0..nombre.size()-1]`
- Para acceder al i-ésimo carácter de la cadena (de tipo `char`), comprobando que el valor del índice (`i`) es adecuado, de tal forma que si el índice (`i`) se encuentra fuera de rango, entonces lanza la excepción `out_of_range` (abortará la ejecución del programa):
  - `char c = nombre.at(i);` donde `i` ∈ `[0..nombre.size()-1]`.
  - `nombre.at(i) = 'z';` donde `i` ∈ `[0..nombre.size()-1]`.
- Obtener una *nueva* subcadena (de tipo `string`) a partir del índice `i`, con un tamaño especificado por `sz`. Si no se especifica el tamaño, o `(sz > nombre.size()-i)`, entonces se toma la subcadena desde el índice hasta el final. Si el índice (`i`) se encuentra fuera de rango, entonces lanza la excepción `out_of_range` (abortará la ejecución del programa):
  - `string sb = nombre.substr(i);` donde `i` ∈ `[0..nombre.size()]`
  - `string sb = nombre.substr(i, sz);` donde `i` ∈ `[0..nombre.size()]`
  - Nótese que **no** es válida la asignación a una subcadena: `nombre.substr(i, sz) = "...";`
- En *GNU G++*, la opción de compilación `-D_GLIBCXX_DEBUG` permite comprobar los índices de acceso.

[1] Comparación lexicográfica se basa en la ordenación alfabética, y es comúnmente utilizada en los diccionarios.

Ejemplos

Ejemplo 1

Programa que convierte una cadena de caracteres a mayúsculas:

```
#include <iostream>
#include <string>
using namespace std ;
// -- Subalgoritmos ----
void mayuscula (char& letra)
{
    if ((letra >= 'a') && (letra <= 'z')) {
        letra = char(letra - 'a' + 'A') ;
    }
}
void mayusculas (string& palabra)
{
    for (int i = 0 ; i < int(palabra.size()) ; ++i) {
        mayuscula(palabra[i]) ;
    }
}
// -- Principal --------
int main ()
{
    string palabra ;
    cin >> palabra ;
    mayusculas(palabra) ;
    cout << palabra << endl ;
}
```

Ejemplo 2

Programa que lee una palabra (formada por letras minúsculas), y escribe su plural según las siguientes reglas:

- Si acaba en vocal se le añade la letra `'s'`.
- Si acaba en consonante se le añaden las letras `'es'`. Si la consonante es la letra `'z'`, se sustituye por la letra `'c'`
- Suponemos que la palabra introducida es correcta y está formada por letras minúsculas.

```
#include <iostream>
#include <string>
using namespace std ;
// -- Subalgoritmos ----
bool es_vocal (char c)
{
    return (c == 'a') || (c == 'e') || (c == 'i') || (c == 'o') || (c == 'u') ;
}
void plural_1 (string& palabra)
{
    if (palabra.size() > 0) {
        if (es_vocal(palabra[palabra.size() - 1])) {
            palabra += 's' ;
        } else {
            if (palabra[palabra.size() - 1] == 'z') {
                palabra[palabra.size() - 1] = 'c' ;
            }
            palabra += "es" ;
        }
```

```
    }
}
void plural_2 (string& palabra)
{
    if (palabra.size() > 0) {
        if (es_vocal(palabra[palabra.size() - 1])) {
            palabra += 's' ;
        } else if (palabra[palabra.size() - 1] == 'z') {
            palabra = palabra.substr(0, palabra.size() - 1) + "ces" ;
        } else {
            palabra += "es" ;
        }
    }
}
// -- Principal --------
int main ()
{
    string palabra ;
    cin >> palabra ;
    plural_1(palabra) ;
    cout << palabra << endl ;
}
```

Ejemplo 3

Diseñe una función que devuelva verdadero si la palabra recibida como parámetro es "palíndromo" y falso en caso contrario.

```
bool es_palindromo (const string& palabra)
{
    bool ok = false ;
    if (palabra.size() > 0) {
        int i = 0 ;
        int j = int(palabra.size()) - 1 ;
        while ((i < j) && (palabra[i] == palabra[j])) {
            ++i ;
            --j ;
        }
        ok = i >= j ;
    }
    return ok ;
}
```

Ejemplo 4

Diseñe un subprograma que reemplace una parte de la cadena, especificada por un índice y una longitud, por otra cadena.

```
void reemplazar (string& str, unsigned i, unsigned sz, const string& nueva)
{
    if (i + sz < str.size()) {
        str = str.substr(0, i) + nueva + str.substr(i + sz, str.size() - (i + sz)) ;
    } else if (i <= str.size()) {
        str = str.substr(0, i) + nueva ;
    }
}
```

- Este subprograma es equivalente a la operación `str.replace(i, sz, nueva)`
- `str.replace(i, 0, nueva)` es equivalente a `str.insert(i, nueva)`
- `str.replace(i, sz, "")` es equivalente a `str.erase(i, sz)`.

6.3. Registros o Estructuras

El tipo *registro* se utiliza para la definición de un nuevo tipo mediante la composición de un número determinado de elementos que pueden ser de distintos tipos (simples y compuestos).

Un tipo registro se especifica enumerando los elementos (campos) que lo componen, indicando su tipo y su identificador con el que referenciarlo. Una vez definido el tipo, podremos utilizar la entidad (constante o variable) de dicho tipo como un todo o acceder a los diferentes elementos que lo componen. Por ejemplo, podemos definir un nuevo tipo que represente el concepto de *Fecha* como composición de *día*, *mes* y *año*.

```
struct Fecha {
    unsigned dia ;
    unsigned mes ;
    unsigned anyo ;
} ;
```

y posteriormente utilizarlo para definir constantes:

```
const Fecha f_nac = { 20 , 2, 2001} ;
```

o utilizarlo para definir variables:

```
Fecha f_nac ;
```

Los valores del tipo `Fecha` se componen de tres elementos concretos (el día de tipo `unsigned`, el mes de tipo `unsigned` y el año de tipo `unsigned`). Los identificadores `dia`, `mes` y `anyo` representan los nombres de sus elementos componentes, denominados **campos**, y su ámbito de visibilidad se restringe a la propia definición del registro. Los campos de un registro pueden ser de cualquier tipo de datos, simple o estructurado. Por ejemplo:

```
// -- Tipos ------------
struct Empleado {
    string   nombre ;
    unsigned codigo ;
    unsigned sueldo ;
    Fecha    fecha_ingreso ;
} ;
// -- Principal --------
int main ()
{
    Empleado e ;
    // ...
}
```

Una vez declarada una entidad (constante o variable) de tipo registro, por ejemplo la variable `f_nac`, podemos referirnos a ella en su globalidad (realizando asignaciones y pasos de parámetros) o acceder a sus componentes (campos) especificándolos tras el operador punto (`.`), donde un determinado componente podrá utilizarse en cualquier lugar en que resulten válidas las variables de su mismo tipo.

```
int main ()
{
    Fecha f_nac, hoy ;
    f_nac.dia = 18 ;
    f_nac.mes = 10 ;
    f_nac.anyo = 2001 ;

    hoy = f_nac ;
}
```

Ejemplo

```
#include <iostream>
#include <string>
using namespace std ;
// -- Constantes -------
const unsigned SEGMIN = 60 ;
const unsigned MINHOR = 60 ;
const unsigned MAXHOR = 24 ;
const unsigned SEGHOR = SEGMIN * MINHOR ;
// -- Tipos ------------
struct Tiempo {
    unsigned horas ;
    unsigned minutos ;
    unsigned segundos ;
} ;
// -- Subalgoritmos ----
unsigned leer_rango (unsigned inf, unsigned sup)
{
    unsigned num ;
    do {
        cin >> num ;
    } while ( ! ((num >= inf) && (num < sup))) ;
    return num ;
}
void leer_tiempo (Tiempo& t)
{
    t.horas = leer_rango(0, MAXHOR) ;
    t.minutos = leer_rango(0, MINHOR) ;
    t.segundos = leer_rango(0, SEGMIN) ;
}
void escribir_tiempo (const Tiempo& t)
{
    cout << t.horas << ":" << t.minutos << ":" << t.segundos ;
}
unsigned tiempo_a_seg (const Tiempo& t)
{
    return (t.horas * SEGHOR) + (t.minutos * SEGMIN) + (t.segundos) ;
}
void seg_a_tiempo (unsigned sg, Tiempo& t)
{
    t.horas = sg / SEGHOR ;
    t.minutos = (sg % SEGHOR) / SEGMIN ;
    t.segundos = (sg % SEGHOR) % SEGMIN ;
}
void diferencia (const Tiempo& t1, const Tiempo& t2, Tiempo& dif)
{
    seg_a_tiempo(tiempo_a_seg(t2) - tiempo_a_seg(t1), dif) ;
}
// -- Principal --------
int main ()
{
    Tiempo t1, t2, dif ;
    leer_tiempo(t1) ;
    leer_tiempo(t2) ;
    diferencia(t1, t2, dif) ;
    escribir_tiempo(dif) ;
    cout << endl ;
}
```

6.4. Agregados: el Tipo Array

El tipo *array* se utiliza para la definición de un nuevo tipo mediante la **agregación** de entidades menores del *mismo tipo*, es decir, se define como una colección de un número determinado (definido en tiempo de compilación) de elementos de un mismo tipo de datos, de tal forma que se puede acceder a cada elemento individual de la colección de forma *parametrizada* mediante índices.

Los arrays son útiles en todas aquellas circunstancias en que necesitamos tener almacenados una colección de valores (un número fijo predeterminado en tiempo de compilación) a los cuales pretendemos acceder de forma parametrizada, normalmente para aplicar un proceso iterativo.

Es posible utilizar el tipo `array` de la biblioteca estándar para definir agregados. Para ello, se debe incluir la biblioteca `<tr1/array>`, así como utilizar el espacio de nombres de `std::tr1`.[2] La definición de agregados de este tipo permite definir agregados mas robustos y con mejores características que los agregados predefinidos.

Un tipo *agregado* se especifica declarando el tipo base de los elementos que lo componen y el número de elementos (constante especificada en tiempo de compilación) de que consta dicha agregación. Así, por ejemplo, podemos definir un nuevo tipo `Vector` como un agregado de 5 elementos, cada uno del tipo `int`, y definir variables y constantes de dicho tipo (nótese que los elementos constantes del tipo array se especifican entre *llaves dobles*[3]):

```
#include <tr1/array>
using namespace std::tr1 ;
// -- Constantes -------
const int NELMS = 5 ;
// -- Tipos ------------
typedef array<int, NELMS> Vector ;
// -- Constantes -------
const Vector PRIMOS = {{ 2, 3, 5, 7, 11 }} ;
// -- Principal --------
int main ()
{
    Vector v ;
}
```

PRIMOS:

2	3	5	7	11
0	1	2	3	4

v:

?	?	?	?	?
0	1	2	3	4

El tipo base (de los elementos) del array puede ser de tipo simple o compuesto, así, por ejemplo, podemos definir un nuevo tipo `Citas` como un agregado de 4 elementos, cada uno del tipo `Fecha`, y definir variables y constantes de dicho tipo:

```
#include <tr1/array>
using namespace std::tr1 ;
struct Fecha {
    unsigned dia ;
    unsigned mes ;
    unsigned anyo ;
} ;
const int N_CITAS = 4 ;
typedef array<Fecha, N_CITAS> Citas ;
const Citas CUMPLEANYOS = {{
    { 1, 1, 2001 },
    { 2, 2, 2002 },
    { 3, 3, 2003 },
    { 4, 4, 2004 }
}} ;
int main()
{
    Citas cit ;
    // ...
    cit = CUMPLEANYOS ;
}
```

CUMPLEANYOS:

1	2	3	4
1	2	3	4
2001	2002	2003	2004
0	1	2	3

[2] Cuando los compiladores se adapten al nuevo estándar de C++ (2011), entonces se deberá incluir la biblioteca `<array>`, utilizar el espacio de nombres `std` y será suficiente especificar los elementos entre *llaves simples*.

Un agregado de tipo `array<...>` acepta la asignación (`=`) entre variables de dicho tipo. También se le pueden aplicar los operadores relacionales (`==`, `!=`, `>`, `>=`, `<`, `<=`) a entidades del mismo tipo `array<...>` siempre y cuando a los elementos del agregado (del tipo base del array) se le puedan aplicar dichos operadores.

Para conocer el número de elementos que componen un determinado agregado, la operación `cit.size()` proporciona dicho valor, que en este ejemplo es `4`.

Para acceder a un elemento concreto del agregado, especificaremos entre corchetes (`[` y `]`) el índice de la posición que ocupa el mismo, teniendo en cuenta que el primer elemento ocupa la posición 0 (cero) y el último elemento ocupa la posición `a.size()-1`. Por ejemplo `cit[0]` y `cit[cit.size()-1]` aluden al primer y último elemento del agregado respectivamente. Un determinado elemento puede utilizarse en cualquier lugar donde sea válido una variable de su mismo tipo base.

El lenguaje de programación C++ no comprueba que los accesos a los elementos de un agregado sean correctos y se encuentren dentro de los límites válidos del array, por lo que será responsabilidad del programador comprobar que así sea.

Sin embargo, en *GNU G++*, la opción de compilación `-D_GLIBCXX_DEBUG` permite comprobar los índices de acceso (descargar la biblioteca de página web de la asignatura).

También es posible acceder a un determinado elemento mediante la operación `at(i)`, de tal forma que si el valor del índice `i` está fuera del rango válido, entonces se lanzará una excepción `out_of_range`. Se puede tanto utilizar como modificar el valor de este elemento.

```
#include <tr1/array>
using namespace std::tr1 ;
struct Fecha {
    unsigned dia ;
    unsigned mes ;
    unsigned anyo ;
} ;
const int N_CITAS = 4 ;
typedef array<Fecha, N_CITAS> Citas ;
int main()
{
    Citas cit ;
    cit[0].dia = 18 ;
    cit[0].mes = 10 ;
    cit[0].anyo = 2001 ;
    for (int i = 0 ; i < int(cit.size()) ; ++i) {
        cit[i].dia = 1 ;
        cit[i].mes = 1 ;
        cit[i].anyo = 2002 ;
    }
    cit[N_CITAS] = { 1, 1, 2002 } ;    // ERROR. Acceso fuera de los límites
    cit.at(N_CITAS) = { 1, 1, 2002 } ; // ERROR. Lanza excepción out_of_range
    // ...
}
```

Ejemplo 1

```
#include <iostream>
#include <tr1/array>
using namespace std ;
using namespace std::tr1 ;
// -- Constantes -------
const int NELMS = 5 ;
// -- Tipos ------------
typedef array<int, NELMS> Vector ;
// -- Subalgoritmos ----
void leer (Vector& v)
```

```
{
    for (int i = 0 ; i < int(v.size()) ; ++i) {
        cin >> v[i] ;
    }
}
int sumar (const Vector& v)
{
    int suma = 0 ;
    for (int i = 0 ; i < int(v.size()) ; ++i) {
        suma += v[i] ;
    }
    return suma ;
}
// -- Principal --------
int main ()
{
    Vector v1, v2 ;
    leer(v1) ;
    leer(v2) ;
    if (sumar(v1) == sumar(v2)) {
        cout << "Misma suma" << endl ;
    }
    if (v1 < v2) {
        cout << "Vector Menor" << endl ;
    }
    v1 = v2 ;   // Asignación
    if (v1 == v2) {
        cout << "Vectores Iguales" << endl ;
    }
}
```

Ejemplo 2

Programa que lee las ventas de cada "agente" e imprime su sueldo que se calcula como una cantidad fija (1000 €) más un incentivo que será un 10 % de las ventas que ha realizado. Dicho incentivo sólo será aplicable a aquellos agentes cuyas ventas superen los 2/3 de la media de ventas del total de los agentes.

```
#include <iostream>
#include <tr1/array>
using namespace std ;
using namespace std::tr1 ;
// -- Constantes -------
const int NAGENTES = 20 ;
const double SUELDO_FIJO = 1000.0 ;
const double INCENTIVO = 10.0 ;
const double PROMEDIO = 2.0 / 3.0 ;
// -- Tipos ------------
typedef array<double, NAGENTES> Ventas ;
// -- Subalgoritmos ----
double calc_media (const Ventas& v)
{
    double suma = 0.0 ;
    for (int i = 0 ; i < int(v.size()) ; ++i) {
        suma += v[i] ;
    }
    return suma / double(v.size()) ;
}
inline double porcentaje (double p, double valor)
```

```
{
    return (p * valor) / 100.0 ;
}
void leer_ventas (Ventas& v)
{
    for (int i = 0 ; i < int(v.size()) ; ++i) {
        cout << "Introduzca ventas del Agente " << i << ": " ;
        cin >> v[i] ;
    }
}
void imprimir_sueldos (const Ventas& v)
{
    double umbral = PROMEDIO * calc_media(ventas) ;
    for (int i = 0 ; i < int(v.size()) ; ++i) {
        double sueldo = SUELDO_FIJO ;
        if (v[i] >= umbral) {
            sueldo += porcentaje(INCENTIVO, v[i]) ;
        }
        cout << "Agente: " << i << " Sueldo: " << sueldo << endl ;
    }
}
// -- Principal --------
int main ()
{
    Ventas ventas ;
    leer_ventas(ventas) ;
    imprimir_sueldos(ventas) ;
}
```

Agregados Incompletos

Hay situaciones donde un array se define en tiempo de compilación con un tamaño mayor que el número de elementos actuales válidos que contendrá durante el Tiempo de Ejecución.

- Gestionar el array con huecos durante la ejecución del programa suele ser, en la mayoría de los casos, complejo e ineficiente.
- Mantener los elementos actuales válidos consecutivos al comienzo del array suele ser más adecuado:
  - Marcar la separación entre los elementos actuales válidos de los elementos vacíos con algún valor de adecuado suele ser, en la mayoría de los casos, complejo e ineficiente.
  - Definir un registro que contenga tanto el array, como el número de elementos actuales válidos consecutivos que contiene suele ser más adecuado.

Ejemplo

```
#include <iostream>
#include <tr1/array>
using namespace std ;
using namespace std::tr1 ;
// -- Constantes -------
const int MAX_AGENTES = 20 ;
const double SUELDO_FIJO = 1000.0 ;
const double INCENTIVO = 10.0 ;
const double PROMEDIO = 2.0 / 3.0 ;
// -- Tipos ------------
typedef array<double, MAX_AGENTES> Datos ;
struct Ventas {
    int nelms ;
    Datos elm ;
```

```
} ;
// -- Subalgoritmos ----
double calc_media (const Ventas& v)
{
    double suma = 0.0 ;
    for (int i = 0 ; i < v.nelms ; ++i) {
        suma += v.elm[i] ;
    }
    return suma / double(v.nelms) ;
}
inline double porcentaje (double p, double valor)
{
    return (p * valor) / 100.0 ;
}
void leer_ventas_ag (int i, double& v)
{
    cout << "Introduzca ventas Agente " << i << ": " ;
    cin >> v ;
}
// ----------------------------------
// Dos métodos diferentes de leer un
// vector incompleto:
// ----------------------------------
// Método-1: cuando se conoce a priori el número
//           de elementos que lo componen
// ----------------------------------
void leer_ventas_2 (Ventas& v)
{
    int nag ;
    cout << "Introduzca total de agentes: " ;
    cin >> nag ;
    if (nag > int(v.elm.size())) {
        v.nelms = 0 ;
        cout << "Error" << endl ;
    } else {
        v.nelms = nag ;
        for (int i = 0 ; i < v.nelms ; ++i) {
            leer_ventas_ag(i, v.elm[i]) ;
        }
    }
}
// ----------------------------------
// Método-2: cuando NO se conoce a priori el número
//           de elementos que lo componen, y este
//           número depende de la propia lectura de
//           los datos
// ----------------------------------
void leer_ventas_1 (Ventas& v)
{
    double vent_ag ;
    v.nelms = 0 ;
    leer_ventas_ag(v.nelms+1, vent_ag) ;
    while ((v.nelms < int(v.elm.size()))&&(vent_ag > 0)) {
        v.elm[v.nelms] = vent_ag ;
        ++v.nelms ;
        leer_ventas_ag(v.nelms+1, vent_ag) ;
    }
}
// ----------------------------------
```

```
void imprimir_sueldos (const Ventas& v)
{
    double umbral = PROMEDIO * calc_media(ventas) ;
    for (int i = 0 ; i < v.nelms ; ++i) {
        double sueldo = SUELDO_FIJO ;
        if (v.elm[i] >= umbral) {
            sueldo += porcentaje(INCENTIVO, v.elm[i]) ;
        }
        cout << "Agente: " << i << " Sueldo: " << sueldo << endl ;
    }
}
// -- Principal --------
int main ()
{
    Ventas ventas ;
    leer_ventas(ventas) ;
    imprimir_sueldos(ventas) ;
}
```

Agregados Multidimensionales

El *tipo Base* de un array puede ser tanto simple como compuesto, por lo tanto puede ser otro array, dando lugar a *arrays con múltiples dimensiones*. Así, cada elemento de un array puede ser a su vez otro array.

Los agregados anteriormente vistos se denominan de *una dimensión*. Así mismo, es posible declarar agregados de varias dimensiones. Un ejemplo de un agregado de dos dimensiones:

```
#include <iostream>
#include <tr1/array>
using namespace std ;
using namespace std::tr1 ;
// -- Constantes -------
const int NFILAS = 3 ;
const int NCOLUMNAS = 5 ;
// -- Tipos ------------
typedef array<int, NCOLUMNAS> Fila ;
typedef array<Fila, NFILAS> Matriz ;
// -- Principal --------
int main ()
{
    Matriz m ;
    for (int f = 0 ; f < int(m.size()) ; ++f) {
        for (int c = 0 ; c < int(m[f].size()) ; ++c) {
            m[f][c] = (f * 10) + c ;
        }
    }
    Matriz mx = m ;        // asigna a mx los valores de la Matriz m
    Fila fil = m[0] ;      // asigna a fil el array con valores {{ 00, 01, 02, 03, 04 }}
    int n = m[2][4] ; // asigna a n el valor 24
}
```

m:

	0	1	2	3	4
0	00	01	02	03	04
1	10	11	12	13	14
2	20	21	22	23	24

Donde `m` hace referencia a una variable de tipo `Matriz`, `m[f]` hace referencia a la fila `f` de la matriz `m` (que es de tipo `Fila`), y `m[f][c]` hace referencia al elemento `c` de la fila `f` de la matriz `m` (que es de tipo `int`). Del mismo modo, el número de filas de la matriz `m` es igual a `m.size()`, y el número de elementos de la fila `f` de la matriz `m` es igual a `m[f].size()`.

Ejemplo 1

Diseñar un programa que lea una matriz de 3×5 de números enteros (fila a fila), almacenándolos en un array bidimensional, finalmente imprima la matriz según el siguiente formato:

a a a a a b
a a a a a b
a a a a a b
c c c c c

donde *a* representa los elementos de la matriz leída desde el teclado, *b* representa el resultado de sumar todos los elementos de la fila correspondiente, y *c* representa el resultado de sumar todos los elementos de la columna donde se encuentran. Nótese en el ejemplo como es posible pasar como parámetro una única *fila*, y sin embargo no es posible pasar como parámetro una única *columna*.

```
#include <iostream>
#include <tr1/array>
using namespace std ;
using namespace std::tr1 ;
// -- Constantes -------
const int NFILAS = 3 ;
const int NCOLUMNAS = 5 ;
// -- Tipos ------------
typedef array<int, NCOLUMNAS> Fila ;
typedef array<Fila, NFILAS> Matriz ;
// -- Subalgoritmos ----
int sumar_fila (const Fila& fil)
{
    int suma = 0 ;
    for (int c = 0 ; c < int(fil.size()) ; ++c) {
        suma += fil[c] ;
    }
    return suma ;
}
int sumar_columna (const Matriz& m, int c)
{
    int suma = 0 ;
    for (int f = 0 ; f < int(m.size()) ; ++f) {
        suma += m[f][c] ;
    }
    return suma ;
}
void escribir_fila (const Fila& fil)
{
    for (int c = 0 ; c < int(fil.size()) ; ++c) {
        cout << fil[c] << " " ;
    }
}
void escribir_matriz_formato (const Matriz& m)
{
    for (int f = 0 ; f < int(m.size()) ; ++f) {
        escribir_fila(m[f]) ;
        cout << sumar_fila(m[f]) ;
        cout << endl ;
    }
    for (int c = 0 ; c < int(m[0].size()) ; ++c) {
        cout << sumar_columna(m, c) << " " ;
    }
    cout << endl ;
}
void leer_matriz (Matriz& m)
{
    cout << "Escribe fila a fila" << endl ;
    for (int f = 0 ; f < int(m.size()) ; ++f) {
        for (int c = 0 ; c < int(m[f].size()) ; ++c) {
```

```
            cin >> m[f][c] ;
        }
    }
}
// -- Principal --------
int main ()
{
    Matriz m ;
    leer_matriz(m) ;
    escribir_matriz_formato(m) ;
}
```

Ejemplo 2

Diseñe un programa que realice el producto de 2 matrices de máximo 10×10 elementos:

```
#include <iostream>
#include <cassert>
#include <tr1/array>
using namespace std ;
using namespace std::tr1 ;
// -- Constantes -------
const int MAX = 10 ;
// -- Tipos ------------
typedef array<double, MAX> Fila ;
typedef array<Fila, MAX> Tabla ;
struct Matriz {
    int n_fil ;
    int n_col ;
    Tabla datos ;
} ;
// -- Subalgoritmos ----
void leer_matriz (Matriz& m)
{
    cout << "Dimensiones?: " ;
    cin >> m.n_fil >> m.n_col ;
    assert(m.n_fil <= int(m.datos.size()) && m.n_col <= int(m.datos[0].size())) ;
    cout << "Escribe valores fila a fila:" << endl ;
    for (int f = 0 ; f < m.n_fil ; ++f) {
        for (int c = 0 ; c < m.n_col ; ++c) {
            cin >> m.datos[f][c] ;
        }
    }
}
void escribir_matriz (const Matriz& m)
{
    for (int f = 0 ; f < m.n_fil ; ++f) {
        for (int c = 0 ; c < m.n_col ; ++c) {
            cout << m.datos[f][c] << " " ;
        }
        cout << endl ;
    }
}
double suma_fila_por_col (const Matriz& x, const Matriz& y, int f, int c)
{
    assert(x.n_col == y.n_fil) ; // PRE-COND
    double suma = 0.0 ;
    for (int k = 0 ; k < x.n_col ; ++k) {
        suma += x.datos[f][k] * y.datos[k][c] ;
    }
```

```
    return suma ;
}
void mult_matriz (Matriz& m, const Matriz& a, const Matriz& b)
{
    assert(a.n_col == b.n_fil) ; // PRE-COND
    m.n_fil = a.n_fil ;
    m.n_col = b.n_col ;
    for (int f = 0 ; f < m.n_fil ; ++f) {
        for (int c = 0 ; c < m.n_col ; ++c) {
            m.datos[f][c] = suma_fil_por_col(a, b, f, c) ;
        }
    }
}
// -- Principal --------
int main ()
{
    Matriz a,b,c ;
    leer_matriz(a) ;
    leer_matriz(b) ;
    if (a.n_col != b.n_fil) {
        cout << "No se puede multiplicar." << endl ;
    } else {
        mult_matriz(c, a, b) ;
        escribir_matriz(c) ;
    }
}
```

6.5. Resolución de Problemas Utilizando Tipos Compuestos

Diseñe un programa para gestionar una agenda personal que contenga la siguiente información: Nombre, Teléfono, Dirección, Calle, Número, Piso, Código Postal y Ciudad, y las siguientes operaciones:

- Añadir los datos de una persona.
- Acceder a los datos de una persona a partir de su nombre.
- Borrar una persona a partir de su nombre.
- Modificar los datos de una persona a partir de su nombre.
- Listar el contenido completo de la agenda.

```
#include <iostream>
#include <string>
#include <cassert>
#include <tr1/array>
using namespace std ;
using namespace std::tr1 ;
// -- Constantes -------
const int MAX_PERSONAS = 50 ;
// -- Tipos ------------
struct Direccion {
    unsigned num ;
    string calle ;
    string piso ;
    string cp ;
    string ciudad ;
} ;
struct Persona {
    string nombre ;
```

```
    string tel ;
    Direccion direccion ;
} ;
// -- Tipos ------------
typedef array<Persona, MAX_PERSONAS> Personas ;
struct Agenda {
    int n_pers ;
    Personas pers ;
} ;
enum Cod_Error {
    OK, AG_LLENA, NO_ENCONTRADO, YA_EXISTE
} ;
// -- Subalgoritmos ----
void Inicializar (Agenda& ag)
{
    ag.n_pers = 0 ;
}
//--------------------------
void Leer_Direccion (Direccion& dir)
{
    cin >> dir.calle ;
    cin >> dir.num ;
    cin >> dir.piso ;
    cin >> dir.cp ;
    cin >> dir.ciudad ;
}
//--------------------------
void Escribir_Direccion (const Direccion& dir)
{
    cout << dir.calle << " " ;
    cout << dir.num << " " ;
    cout << dir.piso << " " ;
    cout << dir.cp << " " ;
    cout << dir.ciudad << " " ;
}
//--------------------------
void Leer_Persona (Persona& per)
{
    cin >> per.nombre ;
    cin >> per.tel ;
    Leer_Direccion(per.direccion) ;
}
//--------------------------
void Escribir_Persona (const Persona& per)
{
    cout << per.nombre << " " ;
    cout << per.tel << " " ;
    Escribir_Direccion(per.direccion) ;
    cout << endl ;
}
//--------------------------
// Busca una Persona en la Agenda
// Devuelve su posición si se encuentra, o bien >= ag.n_pers en otro caso
int Buscar_Persona (const string& nombre, const Agenda& ag)
{
    int i = 0 ;
    while ((i < ag.n_pers) && (nombre != ag.pers[i].nombre)) {
        ++i ;
    }
```

```
    return i ;
}
//---------------------------
void Anyadir (Agenda& ag, const Persona& per)
{
    assert(ag.n_pers < int(ag.pers.size())) ;
    ag.pers[ag.n_pers] = per ;
    ++ag.n_pers ;
}
//---------------------------
void Eliminar (Agenda& ag, int pos)
{
    assert(pos < ag.n_pers) ;
    if (pos < ag.npers-1) {
        ag.pers[pos] = ag.pers[ag.n_pers - 1] ;
    }
    --ag.n_pers ;
}
//---------------------------
void Anyadir_Persona (const Persona& per, Agenda& ag, Cod_Error& ok)
{
    int i = Buscar_Persona(per.nombre, ag) ;
    if (i < ag.n_pers) {
        ok = YA_EXISTE ;
    } else if (ag.n_pers >= int(ag.pers.size())) {
        ok = AG_LLENA ;
    } else {
        ok = OK ;
        Anyadir(ag, per) ;
    }
}
//---------------------------
void Borrar_Persona (const string& nombre, Agenda& ag, Cod_Error& ok)
{
    int i = Buscar_Persona(nombre, ag) ;
    if (i >= ag.n_pers) {
        ok = NO_ENCONTRADO ;
    } else {
        ok = OK ;
        Eliminar(ag, i) ;
    }
}
//---------------------------
void Modificar_Persona (const string& nombre, const Persona& nuevo, Agenda& ag, Cod_Error& ok)
{
    int i = Buscar_Persona(nombre, ag) ;
    if (i >= ag.n_pers) {
        ok = NO_ENCONTRADO ;
    } else {
        Eliminar(ag, i) ;
        Anyadir_Persona(nuevo, ag, ok) ;
    }
}
//---------------------------
void Imprimir_Persona (const string& nombre, const Agenda& ag, Cod_Error& ok)
{
    int i = Buscar_Persona(nombre, ag) ;
    if (i >= ag.n_pers) {
        ok = NO_ENCONTRADO ;
```

```
    } else {
        ok = OK ;
        Escribir_Persona(ag.pers[i]) ;
    }
}
//---------------------------
void Imprimir_Agenda (const Agenda& ag, Cod_Error& ok)
{
    for (int i = 0 ; i < ag.n_pers ; ++i) {
        Escribir_Persona(ag.pers[i]) ;
    }
    ok = OK ;
}
//---------------------------
char Menu ()
{
    char opcion ;
    cout << endl ;
    cout << "a. - Añadir Persona" << endl ;
    cout << "b. - Buscar Persona" << endl ;
    cout << "c. - Borrar Persona" << endl ;
    cout << "d. - Modificar Persona" << endl ;
    cout << "e. - Imprimir Agenda" << endl ;
    cout << "x. - Salir" << endl ;
    do {
        cout << "Introduzca Opción: " ;
        cin >> opcion ;
    } while ( ! (((opcion >= 'a') && (opcion <= 'e')) || (opcion == 'x'))) ;
    return opcion ;
}
//---------------------------
void Escribir_Cod_Error (Cod_Error cod)
{
    switch (cod) {
    case OK:
        cout << "Operación correcta" << endl ;
        break ;
    case AG_LLENA:
        cout << "Agenda llena" << endl ;
        break ;
    case NO_ENCONTRADO:
        cout << "La persona no se encuentra en la agenda" << endl ;
        break ;
    case YA_EXISTE:
        cout << "La persona ya se encuentra en la agenda" << endl ;
        break ;
    }
}
// -- Principal --------
int main ()
{
    Agenda ag ;
    char opcion ;
    Persona per ;
    string nombre ;
    Cod_Error ok ;
    Inicializar(ag) ;
    do {
        opcion = Menu() ;
```

```
            switch (opcion) {
            case 'a':
                cout << "Introduzca los datos de la Persona" << endl ;
                cout << "(nombre, tel, calle, num, piso, cod_postal, ciudad)" << endl ;
                Leer_Persona(per) ;
                Anyadir_Persona(per, ag, ok) ;
                Escribir_Cod_Error(ok) ;
                break ;
            case 'b':
                cout << "Introduzca Nombre" << endl ;
                cin >> nombre ;
                Imprimir_Persona(nombre, ag, ok) ;
                Escribir_Cod_Error(ok) ;
                break ;
            case 'c':
                cout << "Introduzca Nombre" << endl ;
                cin >> nombre ;
                Borrar_Persona(nombre, ag, ok) ;
                Escribir_Cod_Error(ok) ;
                break ;
            case 'd':
                cout << "Introduzca Nombre" << endl ;
                cin >> nombre ;
                cout << "Nuevos datos de la Persona" << endl ;
                cout << "(nombre, tel, calle, num, piso, cod_postal, ciudad)" << endl ;
                Leer_Persona(per) ;
                Modificar_Persona(nombre, per, ag, ok) ;
                Escribir_Cod_Error(ok) ;
                break ;
            case 'e':
                Imprimir_Agenda(ag, ok) ;
                Escribir_Cod_Error(ok) ;
                break ;
            }
        } while (opcion != 'x' ) ;
    }
```

Capítulo 7

Búsqueda y Ordenación

Los algoritmos de búsqueda de un elemento en una colección de datos, así como los algoritmos de ordenación, son muy utilizados comúnmente, por lo que merecen un estudio explícito. En el caso de los algoritmos de búsqueda, éstos normalmente retornan la posición, dentro de la colección de datos, del elemento buscado, y en caso de no ser encontrado, retornan una posición *no válida*. Por otra parte, los algoritmos de ordenación organizan una colección de datos de acuerdo con algún criterio de ordenación. Los algoritmos de ordenación que se verán en este capítulo son los más fáciles de programar, pero sin embargo son los más ineficientes.

7.1. Búsqueda Lineal (Secuencial)

La búsqueda lineal es adecuada como mecanismo de búsqueda general en colecciones de datos *sin organización* conocida. Consiste en ir recorriendo secuencialmente la colección de datos hasta encontrar el elemento buscado, o en última instancia recorrer toda la colección completa, en cuyo caso el elemento buscado no habrá sido encontrado. En los siguientes ejemplos se presentan los algoritmos básicos, los cuales pueden ser adaptados según las circunstancias.

```
//--------------------------------
typedef array<int, MAXIMO> Vector ;
//--------------------------------
// busca la posición del primer elemento == x
// si no encontrado, retorna v.size()
//-------------
int buscar(int x, const Vector& v)
{
    int i = 0 ;
    while ((i < int(v.size()))&&(x != v[i])) {
        ++i ;
    }
    return i ;
}
//--------------------------------
```

7.2. Búsqueda Binaria

La búsqueda binaria es adecuada como mecanismo de búsqueda cuando las colecciones de datos se encuentran *ordenadas* por algún criterio. Consiste en comprobar si el elemento buscado es igual, menor o mayor que el elemento que ocupa la posición central de la colección de datos, en caso de ser mayor o menor que dicho elemento, se descartan los elementos no adecuados de la colección de datos, y se repite el proceso hasta encontrar el elemento o hasta que no queden elementos adecuados

en la colección, en cuyo caso el elemento no habrá sido encontrado. En los siguientes ejemplos se presentan los algoritmos básicos, los cuales pueden ser adaptados según las circunstancias.

```
//--------------------------------
typedef array<int, MAXIMO> Vector ;
//--------------------------------
// busca la posición del primer elemento == x
// si no encontrado, retorna v.size()
//-------------
int buscar_bin(int x, const Vector& v)
{
    int i = 0 ;
    int f = int(v.size()) ;
    int res = int(v.size()) ;
    while (i < f) {
        int m = (i + f) / 2 ;
        if (x == v[m]) {
            res = i = f = m ;
        } else if (x < v[m]) {
            f = m ;
        } else {
            i = m + 1 ;
        }
    }
    return res ;
}
//--------------------------------
```

7.3. Ordenación por Intercambio (Burbuja)

Se hacen múltiples recorridos sobre la *zona no ordenada* del array, ordenando los elementos *consecutivos*, trasladando en cada uno de ellos al elemento más pequeño hasta el inicio de dicha zona.

```
//--------------------------------
typedef array<int, MAXIMO> Vector ;
//--------------------------------
inline void intercambio(int& x, int& y)
{
    int a = x ;
    x = y ;
    y = a ;
}
//--------------------------------
void subir_menor(Vector& v, int pos)
{
    for (int i = int(v.size())-1 ; i > pos ; --i) {
        if (v[i] < v[i-1]) {
            intercambio(v[i], v[i-1]) ;
        }
    }
}
//--------------------------------
void burbuja(Vector& v)
{
    for (int pos = 0 ; pos < int(v.size())-1 ; ++pos) {
        subir_menor(v, pos) ;
    }
}
```

```
//--------------------------------
```

7.4. Ordenación por Selección

Se busca el elemento más pequeño de la *zona no ordenada* del array, y se traslada al inicio dicha zona, repitiendo el proceso hasta ordenar completamente el array.

```
//--------------------------------
typedef array<int, MAXIMO> Vector ;
//--------------------------------
inline void intercambio(int& x, int& y)
{
    int a = x ;
    x = y ;
    y = a ;
}
//--------------------------------
int posicion_menor(const Vector& v, int pos)
{
    int pos_menor = pos ;
    for (int i = pos_menor+1 ; i < int(v.size()) ; ++i) {
        if (v[i] < v[pos_menor]) {
            pos_menor = i ;
        }
    }
    return pos_menor ;
}
//--------------------------------
inline void subir_menor(Vector& v, int pos)
{
    int pos_menor = posicion_menor(v, pos) ;
    if (pos != pos_menor) {
        intercambio(v[pos], v[pos_menor]) ;
    }
}
//--------------------------------
void seleccion(Vector& v)
{
    for (int pos = 0 ; pos < int(v.size())-1 ; ++pos) {
        subir_menor(v, pos) ;
    }
}
//--------------------------------
```

7.5. Ordenación por Inserción

Se toma el primer elemento de la *zona no ordenada* del array, y se inserta en la posición adecuada de la *zona ordenada* del array, repitiendo el proceso hasta ordenar completamente el array.

```
//--------------------------------
typedef array<int, MAXIMO> Vector ;
//--------------------------------
int buscar_posicion(const Vector& v, int posicion)
{
    int i = 0 ;
    while (/*(i < posicion)&&*/ (v[posicion] > v[i])) {
        ++i ;
```

```
    }
    return i ;
}
//--------------------------------
void abrir_hueco(Vector& v, int p_hueco, int p_elm)
{
    for (int i = p_elm ; i > p_hueco ; --i) {
        v[i] = v[i-1] ;
    }
}
//--------------------------------
void insercion(Vector& v)
{
    for (int pos = 1 ; pos < int(v.size()) ; ++pos) {
        int p_hueco = buscar_posicion(v, pos) ;
        if (p_hueco != pos) {
            int aux = v[pos] ;
            abrir_hueco(v, p_hueco, pos) ;
            v[p_hueco] = aux ;
        }
    }
}
//--------------------------------
```

7.6. Aplicación de los Algoritmos de Búsqueda y Ordenación

Diseñe un programa para gestionar una agenda personal *ordenada* que contenga la siguiente información: Nombre, Teléfono, Dirección, Calle, Número, Piso, Código Postal y Ciudad, y las siguientes operaciones:

- Añadir los datos de una persona.
- Acceder a los datos de una persona a partir de su nombre.
- Borrar una persona a partir de su nombre.
- Modificar los datos de una persona a partir de su nombre.
- Listar el contenido completo de la agenda.

```
#include <iostream>
#include <string>
#include <cassert>
#include <tr1/array>
using namespace std ;
using namespace std::tr1 ;
// -- Constantes -------
const int MAX_PERSONAS = 50 ;
// -- Tipos ------------
struct Direccion {
    unsigned num ;
    string calle ;
    string piso ;
    string cp ;
    string ciudad ;
} ;
struct Persona {
    string nombre ;
    string tel ;
    Direccion direccion ;
} ;
```

```
// -- Tipos ------------
typedef array<Persona, MAX_PERSONAS> Personas ;
struct Agenda {
    int n_pers ;
    Personas pers ;
} ;
enum Cod_Error {
    OK, AG_LLENA, NO_ENCONTRADO, YA_EXISTE
} ;
// -- Subalgoritmos ----
void Inicializar (Agenda& ag)
{
    ag.n_pers = 0 ;
}
//---------------------------
void Leer_Direccion (Direccion& dir)
{
    cin >> dir.calle ;
    cin >> dir.num ;
    cin >> dir.piso ;
    cin >> dir.cp ;
    cin >> dir.ciudad ;
}
//---------------------------
void Escribir_Direccion (const Direccion& dir)
{
    cout << dir.calle << " " ;
    cout << dir.num << " " ;
    cout << dir.piso << " " ;
    cout << dir.cp << " " ;
    cout << dir.ciudad << " " ;
}
//---------------------------
void Leer_Persona (Persona& per)
{
    cin >> per.nombre ;
    cin >> per.tel ;
    Leer_Direccion(per.direccion) ;
}
//---------------------------
void Escribir_Persona (const Persona& per)
{
    cout << per.nombre << " " ;
    cout << per.tel << " " ;
    Escribir_Direccion(per.direccion) ;
    cout << endl ;
}
//---------------------------
// Busca una Persona en la Agenda Ordenada
// Devuelve su posición si se encuentra, o bien >= ag.n_pers en otro caso
int Buscar_Persona (const string& nombre, const Agenda& ag)
{
    int i = 0 ;
    int f = ag.n_pers ;
    int res = ag.n_pers ;
    while (i < f) {
        int m = (i + f) / 2 ;
        int cmp = nombre.compare(ag.pers[m].nombre) ;
        if (cmp == 0) {
```

```
            res = i = f = m ;
        } else if (cmp < 0) {
            f = m ;
        } else {
            i = m + 1 ;
        }
    }
    return res ;
}
//---------------------------
int Buscar_Posicion (const string& nombre, const Agenda& ag)
{
    int i = 0 ;
    while ((i < ag.n_pers) && (nombre > ag.pers[i].nombre)) {
        ++i ;
    }
    return i ;
}
//---------------------------
void Anyadir_Ord (Agenda& ag, int pos, const Persona& per)
{
    for (int i = ag.n_pers ; i > pos ; --i) {
        ag.pers[i] = ag.pers[i - 1] ;
    }
    ag.pers[pos] = per ;
    ++ag.n_pers ;
}
//---------------------------
void Eliminar_Ord (Agenda& ag, int pos)
{
    --ag.n_pers ;
    for (int i = pos ; i < ag.n_pers ; ++i) {
        ag.pers[i] = ag.pers[i + 1] ;
    }
}
//---------------------------
void Anyadir_Persona (const Persona& per, Agenda& ag, Cod_Error& ok)
{
    int pos = Buscar_Posicion(per.nombre, ag) ;
    if ((pos < ag.n_pers) && (per.nombre == ag.pers[pos].nombre)) {
        ok = YA_EXISTE ;
    } else if (ag.n_pers >= int(ag.pers.size())) {
        ok = AG_LLENA ;
    } else {
        ok = OK ;
        Anyadir_Ord(ag, pos, per) ;
    }
}
//---------------------------
void Borrar_Persona (const string& nombre, Agenda& ag, Cod_Error& ok)
{
    int i = Buscar_Persona(nombre, ag) ;
    if (i >= ag.n_pers) {
        ok = NO_ENCONTRADO ;
    } else {
        ok = OK ;
        Eliminar_Ord(ag, i) ;
    }
}
```

```
//---------------------------
void Modificar_Persona (const string& nombre, const Persona& nuevo, Agenda& ag, Cod_Error& ok)
{
    int i = Buscar_Persona(nombre, ag) ;
    if (i >= ag.n_pers) {
        ok = NO_ENCONTRADO ;
    } else {
        ok = OK ;
        Eliminar_Ord(ag, i) ;
        Anyadir_Persona(nuevo, ag, ok) ;
    }
}
//---------------------------
void Imprimir_Persona (const string& nombre, const Agenda& ag, Cod_Error& ok)
{
    int i = Buscar_Persona(nombre, ag) ;
    if (i >= ag.n_pers) {
        ok = NO_ENCONTRADO ;
    } else {
        ok = OK ;
        Escribir_Persona(ag.pers[i]) ;
    }
}
//---------------------------
void Imprimir_Agenda (const Agenda& ag, Cod_Error& ok)
{
    for (int i = 0 ; i < ag.n_pers ; ++i) {
        Escribir_Persona(ag.pers[i]) ;
    }
    ok = OK ;
}
//---------------------------
char Menu ()
{
    char opcion ;
    cout << endl ;
    cout << "a. - Añadir Persona" << endl ;
    cout << "b. - Buscar Persona" << endl ;
    cout << "c. - Borrar Persona" << endl ;
    cout << "d. - Modificar Persona" << endl ;
    cout << "e. - Imprimir Agenda" << endl ;
    cout << "x. - Salir" << endl ;
    do {
        cout << "Introduzca Opción: " ;
        cin >> opcion ;
    } while ( ! (((opcion >= 'a') && (opcion <= 'e')) || (opcion == 'x'))) ;
    return opcion ;
}
//---------------------------
void Escribir_Cod_Error (Cod_Error cod)
{
    switch (cod) {
    case OK:
        cout << "Operación correcta" << endl ;
        break ;
    case AG_LLENA:
        cout << "Agenda llena" << endl ;
        break ;
    case NO_ENCONTRADO:
```

```
            cout << "La persona no se encuentra en la agenda" << endl ;
            break ;
        case YA_EXISTE:
            cout << "La persona ya se encuentra en la agenda" << endl ;
            break ;
        }
    }
    // -- Principal --------
    int main ()
    {
        Agenda ag ;
        char opcion ;
        Persona per ;
        string nombre ;
        Cod_Error ok ;
        Inicializar(ag) ;
        do {
            opcion = Menu() ;
            switch (opcion) {
            case 'a':
                cout << "Introduzca los datos de la Persona"<<endl ;
                cout << "(nombre, tel, calle, num, piso, cod_postal, ciudad)" << endl ;
                Leer_Persona(per) ;
                Anyadir_Persona(per, ag, ok) ;
                Escribir_Cod_Error(ok) ;
                break ;
            case 'b':
                cout << "Introduzca Nombre" << endl ;
                cin >> nombre ;
                Imprimir_Persona(nombre, ag, ok) ;
                Escribir_Cod_Error(ok) ;
                break ;
            case 'c':
                cout << "Introduzca Nombre" << endl ;
                cin >> nombre ;
                Borrar_Persona(nombre, ag, ok) ;
                Escribir_Cod_Error(ok) ;
                break ;
            case 'd':
                cout << "Introduzca Nombre" << endl ;
                cin >> nombre ;
                cout << "Nuevos datos de la Persona" << endl ;
                cout << "(nombre, tel, calle, num, piso, cod_postal, ciudad)" << endl ;
                Leer_Persona(per) ;
                Modificar_Persona(nombre, per, ag, ok) ;
                Escribir_Cod_Error(ok) ;
                break ;
            case 'e':
                Imprimir_Agenda(ag, ok) ;
                Escribir_Cod_Error(ok) ;
                break ;
            }
        } while (opcion != 'x' ) ;
    }
```

Capítulo 8

Algunas Bibliotecas Útiles

En este capítulo se muestra superficialmente algunas funciones básicas de la biblioteca estándar.

cmath

La biblioteca `<cmath>` proporciona principalmente algunas funciones matemáticas útiles:

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

`double sin(double r) ;`	seno, $\sin r$ (en radianes)
`double cos(double r) ;`	coseno, $\cos r$ (en radianes)
`double tan(double r) ;`	tangente, $\tan r$ (en radianes)
`double asin(double x) ;`	arco seno, $\arcsin x, x \in [-1, 1]$
`double acos(double x) ;`	arco coseno, $\arccos x, x \in [-1, 1]$
`double atan(double x) ;`	arco tangente, $\arctan x$
`double atan2(double y, double x) ;`	arco tangente, $\arctan y/x$
`double sinh(double r) ;`	seno hiperbólico, $\sinh r$
`double cosh(double r) ;`	coseno hiperbólico, $\cosh r$
`double tanh(double r) ;`	tangente hiperbólica, $\tanh r$
`double sqrt(double x) ;`	$\sqrt{x}, x \geq 0$
`double pow(double x, double y) ;`	x^y
`double exp(double x) ;`	e^x
`double log(double x) ;`	logaritmo neperiano, $\ln x, x > 0$
`double log10(double x) ;`	logaritmo decimal, $\log x, x > 0$
`double ceil(double x) ;`	menor entero $\geq x$, $\lceil x \rceil$
`double floor(double x) ;`	mayor entero $\leq x$, $\lfloor x \rfloor$
`double fabs(double x) ;`	valor absoluto de x, $\lvert x \rvert$
`double ldexp(double x, int n) ;`	$x2^n$
`double frexp(double x, int* exp) ;`	inversa de ldexp
`double modf(double x, double* ip) ;`	parte entera y fraccionaria
`double fmod(double x, double y) ;`	resto de x/y

cctype

La biblioteca `<cctype>` proporciona principalmente características sobre los valores de tipo `char`:

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

`bool isalnum(char ch) ;`	`(isalpha(ch) || isdigit(ch))`
`bool isalpha(char ch) ;`	`(isupper(ch) || islower(ch))`
`bool iscntrl(char ch) ;`	caracteres de control
`bool isdigit(char ch) ;`	dígito decimal
`bool isgraph(char ch) ;`	caracteres imprimibles excepto espacio
`bool islower(char ch) ;`	letra minúscula
`bool isprint(char ch) ;`	caracteres imprimibles incluyendo espacio
`bool ispunct(char ch) ;`	carac. impr. excepto espacio, letra o dígito
`bool isspace(char ch) ;`	espacio, `'\r'`, `'\n'`, `'\t'`, `'\v'`, `'\f'`
`bool isupper(char ch) ;`	letra mayúscula
`bool isxdigit(char ch) ;`	dígito hexadecimal
`char tolower(char ch) ;`	retorna la letra minúscula correspondiente a `ch`
`char toupper(char ch) ;`	retorna la letra mayúscula correspondiente a `ch`

ctime

La biblioteca `<ctime>` proporciona principalmente algunas funciones generales relacionadas con el tiempo:

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

`clock_t clock() ;`	retorna el tiempo de CPU utilizado (`CLOCKS_PER_SEC`)
`time_t time(0) ;`	retorna el tiempo de calendario (en segundos)

```
#include <iostream>
#include <ctime>
using namespace std ;
// ------------------------------------
int main()
{
    time_t t1 = time(0) ;
    clock_t c1 = clock() ;
    // ... procesamiento ...
    clock_t c2 = clock() ;
    time_t t2 = time(0) ;
    cout << "Tiempo de CPU: " << double(c2 - c1)/double(CLOCKS_PER_SEC) << " seg" << endl ;
    cout << "Tiempo total: " << (t2 - t1) << " seg" << endl ;
}
// ------------------------------------
```

cstdlib

La biblioteca `<cstdlib>` proporciona principalmente algunas funciones generales útiles:

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

`int abs(int n) ;`	retorna el valor absoluto del número `int n`
`long labs(long n) ;`	retorna el valor absoluto del número `long n`
`int system(const char orden[]) ;`	orden a ejecutar por el sistema operativo
`void exit(int estado) ;`	termina la ejecución del programa actual (`EXIT_SUCCESS`, `EXIT_FAILURE`)
`void abort() ;`	aborta la ejecución del programa actual
`void srand(unsigned semilla) ;`	inicializa el generador de números aleatorios
`int rand() ;`	retorna un aleatorio entre `0` y `RAND_MAX` (ambos inclusive)

```
#include <cstdlib>
#include <ctime>
using namespace std ;
```

```
// -------------------------------------
// inicializa el generador de números aleatorios
inline void ini_aleatorio()
{
    srand(time(0)) ;
}
// -------------------------------------
// Devuelve un número aleatorio entre 0 y max (exclusive)
inline int aleatorio(int max)
{
    return int(max*double(rand())/(RAND_MAX+1.0)) ;
}
// -------------------------------------
// Devuelve un número aleatorio entre min y max (ambos inclusive)
inline int aleatorio(int min, int max)
{
    return min + aleatorio(max-min+1) ;
}
// -------------------------------------
```

Parte II

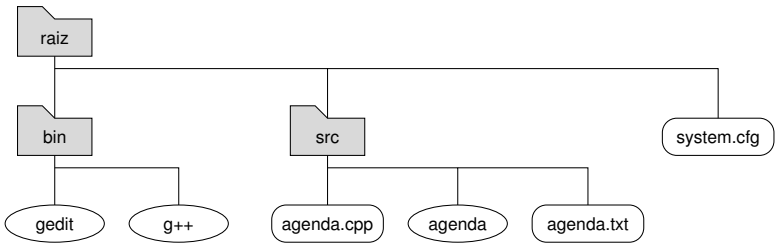
Programación Intermedia

Capítulo 9

Almacenamiento en Memoria Secundaria: Ficheros

Los programas de ordenador usualmente trabajan con datos almacenados en la *memoria principal* (RAM). Esta memoria principal tiene como principales características que tiene un tiempo de acceso (para lectura y escritura) muy eficiente, sin embargo este tipo de memoria es *volátil*, en el sentido de que los datos almacenados en ella desaparecen cuando termina la ejecución del programa o se apaga el ordenador. Los ordenadores normalmente almacenan su información de manera *permanente* en dispositivos de almacenamiento de *memoria secundaria*, tales como dispositivos magnéticos (discos duros, cintas), discos ópticos (CDROM, DVD), memorias permanentes de estado sólido (memorias flash USB), etc.

Estos dispositivos suelen disponer de gran capacidad de almacenamiento, por lo que es necesario alguna organización que permita gestionar y acceder a la información allí almacenada. A esta organización se la denomina el *sistema de ficheros*, y suele estar organizado jerárquicamente en directorios (a veces denominados también carpetas) y ficheros (a veces denominados también archivos), donde los directorios permiten organizar jerárquicamente[1] y acceder a los ficheros, y estos últimos almacenan de forma permanente la información, que puede ser tanto programas (software) como datos que serán utilizados por los programas. Así, los programas acceden y almacenan la información de manera permanente por medio de los ficheros, que son gestionados por el Sistema Operativo dentro de la jerarquía del sistema de ficheros.



Tipos de Ficheros

Los ficheros se pueden clasificar de múltiples formas dependiendo de los criterios seleccionados. En nuestro caso, nos centraremos en la clasificación por la codificación o formato en el que almacenan la información que contienen. Así, podemos distinguir los *ficheros de texto* y los *ficheros binarios*. En los ficheros de texto, la información se almacena utilizando una codificación y formato adecuados para que puedan ser procesados (y leídos), además de por un programa de ordenador, por un *ser humano*. Por lo tanto, los ficheros de texto almacenan la información utilizando una codificación textual como *secuencia de caracteres* (usualmente basada en la codificación ASCII,

[1] Un directorio puede contener múltiples ficheros y, a su vez, también múltiples directorios.

UTF-8, etc), y en un formato que permita su legibilidad y procesamiento. Por otra parte, los ficheros binarios son procesados automáticamente por la ejecución de programas, sin la intervención humana, por lo que no necesitan representar la información en un formato legible para el ser humano. Por ello, suelen codificar la información en un formato orientado a ser procesado eficientemente por los ordenadores, y en ese caso utilizan la representación en el *código binario* que utilizan internamente los ordenadores para representar los datos. En este caso, pueden surgir problemas de compatibilidad en aquellos casos en los que estos ficheros son procesados por programas ejecutándose en ordenadores que utilizan distintas representaciones internas de los datos.

Así por ejemplo, un fichero de texto denominado `fechas.txt` podría estar almacenado en una determinada posición en la jerarquía del sistema de ficheros (`/home/alumno/documentos/fechas.txt`) y contener información sobre las fechas de nacimiento de determinadas personas según el siguiente formato, donde cada línea se encuentra terminada por un carácter terminador de fin de línea:[2]

```
Juan López 12 3 1992
Lola Martínez 23 7 1987
Pepe Jiménez 17 8 1996
```

Téngase en cuenta que el ordenador almacena los números internamente en formato binario, y que éstos han sido convertidos a su representación textual como secuencia de caracteres cuando fueron escritos al fichero de texto, y que la conversión inversa se produce cuando se leen desde el fichero de texto para su procesamiento. Por ejemplo, el número `1992` se almacena en una variable de tipo `unsigned` y se representa internamente como el siguiente número binario de 32 bits (`00000000000000000000011111001000`), sin embargo, su representación textual en el fichero de texto se compone de la siguiente secuencia de cuatro caracteres: `'1' '9' '9' '2'`.

El contenido de un fichero de texto, además de ser procesado por un programa específico, diseñado para su procesamiento considerando su formato, también puede ser visualizado y editado mediante un programa de edición de textos de propósito general, tales como `gedit`, `kate`, `gvim`, `emacs`, etc. en Linux, `textedit` en MacOS-X y `notepad` en Windows, entre otros.

En el caso del software, los programas en código fuente codificados en un lenguaje de programación suelen ser almacenados como ficheros de texto. Sin embargo, el resultado de compilar estos programas fuente a programas ejecutables se almacenan en ficheros binarios (ejecutables por el Sistema Operativo). Así mismo, los ficheros que contienen imágenes, vídeo y música suelen estar, en su mayoría, almacenados en formato binario.

9.1. Flujos de Entrada y Salida Asociados a Ficheros

En el capítulo 3 se explicó que un programa codificado en C++ realiza la entrada y salida de información (con el entorno exterior del programa) a través de los flujos (*stream* en inglés) de entrada y salida respectivamente.

En dicho capítulo se muestra como se realiza la entrada y salida de datos a través de los flujos estándares de entrada y salida respectivamente (`cin` y `cout`), usualmente conectados con el teclado y la pantalla de la consola. Todo lo explicado anteriormente respecto a la entrada y salida básica con los flujos estándares (cap 3), y entrada y salida de cadenas de caracteres (cap 6.2) es también aplicable a los flujos de entrada y salida vinculados a ficheros que veremos en este capítulo.

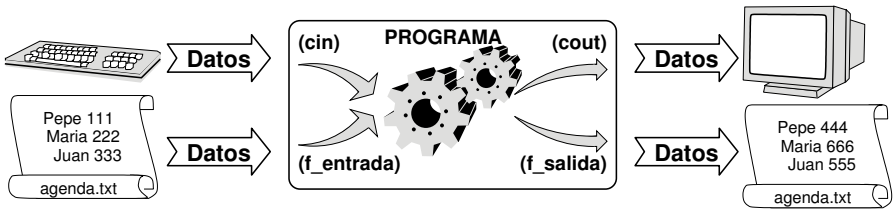
Un flujo de entrada de datos en modo texto actúa como una *fuente* que proporciona una *secuencia de caracteres* (usualmente a través de un buffer de almacenamiento intermedio) desde el que se extraen los caracteres que representan a los datos de entrada, que posteriormente serán convertidos a la representación interna adecuada.

```
Juan López 12 3 1992 ↩ Lola Martínez 23 7 1987 ↩ Pepe Jiménez 17 8 1996 ↩
```

Por el contrario, un flujo de salida de datos en modo texto actúa como un *sumidero* que recibe una *secuencia de caracteres* (usualmente a través de un buffer de almacenamiento intermedio) a

[2] El carácter terminador de fin de línea *no es visible*, aunque se aprecian sus efectos al mostrarse los siguientes caracteres en la siguiente línea.

la que se envían los caracteres que representan a los datos de salida, que previamente han sido convertidos al formato de texto adecuado.



En el caso de entrada y salida a ficheros, el lenguaje de programación C++ posee mecanismos para asociar y *vincular* estos flujos con ficheros almacenados en memoria secundaria en la jerarquía del sistema de ficheros. Así, toda la entrada y salida de información se realiza a través de estos flujos vinculados a ficheros, denominados *manejadores de ficheros*. De este modo, cuando un programa quiere realizar una entrada o salida de datos con un determinado fichero, debe realizar las siguientes acciones:

1. *Incluir* la biblioteca `<fstream>` donde se definen los tipos correspondientes, y utilizar el espacio de nombres `std`.
2. *Declarar* variables del tipo de flujo adecuado (para entrada o salida) para que actúen como manejadores de fichero.
3. *Abrir* el flujo de datos vinculando la variable correspondiente con el fichero especificado. Esta operación establece un vínculo entre la variable (manejador de fichero) definida en nuestro programa con el fichero gestionado por el sistema operativo, de tal forma que toda transferencia de información que el programa realice con el fichero, se realizará a través de la variable manejador de fichero vinculada con el mismo.
4. *Comprobar* que la apertura del fichero del paso previo se realizó correctamente. Si la vinculación con el fichero especificado no pudo realizarse por algún motivo (por ejemplo, el fichero no existe, en el caso de entrada de datos, o no es posible crear el fichero, en el caso de salida de datos), entonces la operación de apertura fallaría.
5. *Realizar* la transferencia de información (de entrada o de salida) con el fichero a través de la variable de flujo vinculada al fichero. Para esta transferencia de información (entrada y salida) se pueden utilizar los mecanismos vistos en los capítulos anteriores (3, 6.2). En el caso de salida de datos, éstos deberán escribirse siguiendo un formato adecuado que permita su posterior lectura, por ejemplo escribiendo los separadores adecuados para ello entre los diferentes valores almacenados.

   Normalmente, tanto la entrada como la salida de datos implican un proceso *iterativo*, que en el caso de entrada se suele realizar hasta leer y procesar todo el contenido del fichero.
6. *Comprobar* que el procesamiento del fichero del paso previo se realizó correctamente, de tal forma que si el procesamiento consistía en entrada de datos, el estado de la variable vinculada al fichero se encuentre en un estado indicando que se ha alcanzado el final del fichero, y si el procesamiento era de salida, el estado de la variable vinculada al fichero se deberá encontrar en un estado correcto.
7. Finalmente *cerrar* el flujo liberando la variable de su vinculación con el fichero. Si no se cierra el flujo de fichero, cuando termine el ámbito de vida de la variable vinculada, el flujo será cerrado automáticamente, y su vinculación liberada.
8. **Nota:** es importante tener en cuenta que cuando un flujo pasa al estado erróneo (`fail()`), entonces cualquier operación de entrada o salida que se realice sobre él también fallará. La operación `clear()` restaura el estado del flujo.

Cuando sea necesario, una variable de tipo flujo, tanto de entrada, como de salida, puede ser pasada como *parámetro por referencia* (**no constante**) a cualquier subprograma.

9.2. Entrada de Datos desde Ficheros de Texto

Para realizar la entrada de datos desde un fichero de texto, el programa debe:

1. Incluir la biblioteca `<fstream>` donde se definen los tipos correspondientes, y utilizar el espacio de nombres `std`.

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

2. Definir una variable manejador de fichero del tipo de flujo de entrada (`ifstream` –*input file stream*).

   ```
   ifstream f_ent ;
   ```

3. Abrir el flujo vinculando la variable correspondiente con el fichero especificado.

   ```
   f_ent.open(nombre_fichero.c_str()) ;
   ```

4. Comprobar que la apertura del fichero se realizó correctamente.

   ```
   if (f_ent.fail()) { ... }
   ```

5. Realizar la entrada de datos con los operadores y subprogramas correspondientes, así como procesar la información leída.

   ```
   f_ent >> nombre >> apellidos >> dia >> mes >> anyo ;
   f_ent.ignore(1000, '\n') ;
   f_ent >> ws ;
   getline(f_ent, linea) ;
   f_ent.get(c) ;
   ```

   Usualmente es un proceso iterativo que se realiza hasta que la operación de entrada de datos falla, usualmente debido a haber alcanzado el final del fichero. Este proceso iterativo usualmente consiste en la iteración del siguiente proceso:

   - Lectura de datos
   - Si la lectura no ha sido correcta, entonces terminar el proceso iterativo.
   - En otro caso, procesamiento de los datos leídos, y vuelta al proceso iterativo, leyendo nuevos datos

   ```
   {
       ...
       leer(f_ent, datos) ;
       while (! f_ent.fail() ... ) {
           procesar(datos, ...) ;
           leer(f_ent, datos) ;
       }
   }
   ```

6. Comprobar que el procesamiento del fichero se realizó correctamente, es decir, el fichero se leyó completamente hasta el final de mismo (`eof` representa *end-of-file*).

   ```
   if (f_ent.eof()) { ... }
   ```

7. Finalmente cerrar el flujo liberando la variable de su vinculación.

   ```
   f_ent.close() ;
   ```

Por ejemplo, un programa que lee números desde un fichero de texto y los procesa (en este caso simplemente los muestra por pantalla):

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std ;
enum Codigo {
    OK, ERROR_APERTURA, ERROR_FORMATO
} ;
void procesar(int num)
{
```

```
    cout << num << endl ;
}
void leer_fich(const string& nombre_fichero, Codigo& ok)
{
    ifstream f_ent ;
    f_ent.open(nombre_fichero.c_str()) ;
    if (f_ent.fail()) {
        ok = ERROR_APERTURA ;
    } else {
        int numero ;
        f_ent >> numero ;
        while (! f_ent.fail()) {
            procesar(numero) ;
            f_ent >> numero ;
        }
        if (f_ent.eof()) {
            ok = OK ;
        } else {
            ok = ERROR_FORMATO ;
        }
        f_ent.close() ;
    }
}
void codigo_error(Codigo ok)
{
    switch (ok) {
    case OK:
        cout << "Fichero procesado correctamente" << endl ;
        break ;
    case ERROR_APERTURA:
        cout << "Error en la apertura del fichero" << endl ;
        break ;
    case ERROR_FORMATO:
        cout << "Error de formato en la lectura del fichero" << endl ;
        break ;
    }
}
int main()
{
    Codigo ok ;
    string nombre_fichero ;
    cout << "Introduzca el nombre del fichero: " ;
    cin >> nombre_fichero ;
    leer_fich(nombre_fichero, ok) ;
    codigo_error(ok) ;
}
```

9.3. Salida de Datos a Ficheros de Texto

Para realizar la salida de datos a un fichero de texto, el programa debe:

1. Incluir la biblioteca `<fstream>` donde se definen los tipos correspondientes, y utilizar el espacio de nombres `std`.

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

2. Definir una variable manejador de fichero del tipo de flujo de salida (`ofstream` –*output file stream*).

   ```
   ofstream f_sal ;
   ```

3. Abrir el flujo vinculando la variable correspondiente con el fichero especificado.

```
f_sal.open(nombre_fichero.c_str()) ;
```

4. Comprobar que la apertura del fichero se realizó correctamente.

```
if (f_sal.fail()) { ... }
```

5. Realizar la salida de datos con los operadores y subprogramas correspondientes, teniendo en cuenta los separadores que se deben escribir para que puedan ser leídos adecuadamente.

```
f_sal << nombre << " " << apellidos << " " << dia << " " << mes << " " << anyo << endl ;
```

   Usualmente éste es un proceso iterativo que se realiza hasta que se escriben en el fichero todos los datos apropiados y mientras el estado del flujo sea correcto.

```
while ( ... ! f_sal.fail() ) { ... }
```

6. Comprobar que el procesamiento del fichero se realizó correctamente, es decir, el fichero se encuentra en buen estado.

```
if ( ! f_sal.fail()) { ... }
```

7. Finalmente cerrar el flujo liberando la variable de su vinculación.

```
f_sal.close() ;
```

Por ejemplo, un programa que lee números de teclado (hasta introducir un cero) y los escribe a un fichero de texto:

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std ;
enum Codigo {
    OK, ERROR_APERTURA, ERROR_FORMATO
} ;
void escribir_fich(const string& nombre_fichero, Codigo& ok)
{
    ofstream f_sal ;
    f_sal.open(nombre_fichero.c_str()) ;
    if (f_sal.fail()) {
        ok = ERROR_APERTURA ;
    } else {
        int numero ;
        cin >> numero ;
        while ((numero > 0) && ! cin.fail() && ! f_sal.fail()) {
            f_sal << numero << endl ;
            cin >> numero ;
        }
        if ( ! f_sal.fail()) {
            ok = OK ;
        } else {
            ok = ERROR_FORMATO ;
        }
        f_sal.close() ;
    }
}
void codigo_error(Codigo ok)
{
    switch (ok) {
    case OK:
        cout << "Fichero guardado correctamente" << endl ;
        break ;
    case ERROR_APERTURA:
```

```
            cout << "Error en la apertura del fichero" << endl ;
            break ;
        case ERROR_FORMATO:
            cout << "Error de formato al escribir al fichero" << endl ;
            break ;
        }
    }
    int main()
    {
        Codigo ok ;
        string nombre_fichero ;
        cout << "Introduzca el nombre del fichero: " ;
        cin >> nombre_fichero ;
        escribir_fich(nombre_fichero, ok) ;
        codigo_error(ok) ;
    }
```

9.4. Ejemplos

Ejemplo 1

Ejemplo de un programa que copia el contenido de un fichero a otro, carácter a carácter:

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std ;
enum Codigo {
    OK, ERROR_APERTURA_ENT, ERROR_APERTURA_SAL, ERROR_FORMATO
} ;
void copiar_fichero(const string& salida, const string& entrada, Codigo& ok)
{
    ifstream f_ent ;
    f_ent.open(entrada.c_str()) ;
    if (f_ent.fail()) {
        ok = ERROR_APERTURA_ENT ;
    } else {
        ofstream f_sal ;
        f_sal.open(salida.c_str()) ;
        if (f_sal.fail()) {
            ok = ERROR_APERTURA_SAL ;
        } else {
            char ch ;
            f_ent.get(ch) ;
            while (! f_ent.fail() && ! f_sal.fail()) {
                f_sal.put(ch) ;
                f_ent.get(ch) ;
            }
            if (f_ent.eof() && ! f_sal.fail()) {
                ok = OK ;
            } else {
                ok = ERROR_FORMATO ;
            }
            f_sal.close() ;  // no es necesario
        }
        f_ent.close() ;  // no es necesario
    }
}
void codigo_error(Codigo ok)
```

```
{
    switch (ok) {
    case OK:
        cout << "Fichero procesado correctamente" << endl ;
        break ;
    case ERROR_APERTURA_ENT:
        cout << "Error en la apertura del fichero de entrada" << endl ;
        break ;
    case ERROR_APERTURA_SAL:
        cout << "Error en la apertura del fichero de salida" << endl ;
        break ;
    case ERROR_FORMATO:
        cout << "Error de formato en la lectura del fichero" << endl ;
        break ;
    }
}
int main()
{
    Codigo ok ;
    string entrada, salida ;
    cout << "Introduzca el nombre del fichero de entrada: " ;
    cin >> entrada ;
    cout << "Introduzca el nombre del fichero de salida: " ;
    cin >> salida ;
    copiar_fichero(salida, entrada, ok) ;
    codigo_error(ok) ;
}
```

Ejemplo 2

Ejemplo de un programa que crea, guarda y carga una agenda personal.

```
//-------------------------------------------------------------------------
#include <iostream>
#include <fstream>
#include <string>
#include <tr1/array>
#include <cctype>
using namespace std ;
using namespace std::tr1 ;
//-------------------------------------------------------------------------
struct Fecha {
    unsigned dia ;
    unsigned mes ;
    unsigned anyo ;
} ;
struct Persona {
    string nombre ;
    string tfn ;
    Fecha  fnac ;
} ;
const int MAX = 100 ;
typedef array<Persona, MAX> APers ;
struct Agenda {
    int nelms ;
    APers elm ;
} ;
//-------------------------------------------------------------------------
void inic_agenda(Agenda& ag)
{
```

```
    ag.nelms = 0 ;
}
void anyadir_persona(Agenda& ag, const Persona& p, bool& ok)
{
    if (ag.nelms < int(ag.elm.size())) {
        ag.elm[ag.nelms] = p ;
        ++ag.nelms ;
        ok = true ;
    } else {
        ok = false ;
    }
}
//------------------------------------------------------------------------
void leer_fecha(Fecha& f)
{
    cout << "Introduza fecha de nacimiento (dia mes año): " ;
    cin >> f.dia >> f.mes >> f.anyo ;
}
void leer_persona(Persona& p)
{
    cout << "Introduza nombre: " ;
    cin >> ws ;
    getline(cin, p.nombre) ;
    cout << "Introduza teléfono: " ;
    cin >> p.tfn ;
    leer_fecha(p.fnac) ;
}
void nueva_persona(Agenda& ag)
{
    bool ok ;
    Persona p ;
    leer_persona(p) ;
    if (! cin.fail()) {
        anyadir_persona(ag, p, ok) ;
        if (!ok) {
            cout << "Error al introducir la nueva persona" << endl ;
        }
    } else {
        cout << "Error al leer los datos de la nueva persona" << endl ;
        cin.clear() ;
        cin.ignore(1000, '\n') ;
    }
}
//------------------------------------------------------------------------
void escribir_fecha(const Fecha& f)
{
    cout << f.dia << '/' << f.mes << '/' << f.anyo ;
}
void escribir_persona(const Persona& p)
{
    cout << "Nombre: " << p.nombre << endl ;
    cout << "Teléfono: " << p.tfn << endl ;
    cout << "Fecha nac: " ;
    escribir_fecha(p.fnac) ;
    cout << endl ;
}
void escribir_agenda(const Agenda& ag)
{
    for (int i = 0 ; i < ag.nelms ; ++i) {
```

```
        cout << "----------------------------------------" << endl ;
        escribir_persona(ag.elm[i]) ;
    }
    cout << "----------------------------------------" << endl ;
}
//------------------------------------------------------------------------
// FORMATO DEL FICHERO DE ENTRADA:
//
//  <nombre> <RC>
//  <teléfono> <dia> <mes> <año> <RC>
//  <nombre> <RC>
//  <teléfono> <dia> <mes> <año> <RC>
//  ...
//------------------------------------------------------------------------
void leer_fecha(ifstream& fich, Fecha& f)
{
    fich >> f.dia >> f.mes >> f.anyo ;
}
void leer_persona(ifstream& fich, Persona& p)
{
    fich >> ws ;
    getline(fich, p.nombre) ;
    fich >> p.tfn ;
    leer_fecha(fich, p.fnac) ;
}
//--------------------------------------------
// Otra posible implementación
// void leer_persona(ifstream& fich, Persona& p)
// {
//     getline(fich, p.nombre) ;
//     fich >> p.tfn ;
//     leer_fecha(fich, p.fnac) ;
//     fich.ignore(1000, '\n') ;
// }
//--------------------------------------------
void leer_agenda(const string& nombre_fich, Agenda& ag, bool& ok)
{
    ifstream fich ;
    Persona p ;

    fich.open(nombre_fich.c_str()) ;
    if (fich.fail()) {
        ok = false ;
    } else {
        ok = true ;
        inic_agenda(ag) ;
        leer_persona(fich, p) ;
        while (!fich.fail() && ok) {
            anyadir_persona(ag, p, ok) ;
            leer_persona(fich, p) ;
        }
        ok = ok && fich.eof() ;
        fich.close() ;
    }
}
void cargar_agenda(Agenda& ag)
{
    bool ok ;
    string nombre_fich ;
```

```
    cout << "Introduce el nombre del fichero: " ;
    cin >> nombre_fich ;
    leer_agenda(nombre_fich, ag, ok) ;
    if (!ok) {
        cout << "Error al cargar el fichero" << endl ;
    }
}
//--------------------------------------------------------------------------
// FORMATO DEL FICHERO DE SALIDA:
//
//  <nombre> <RC>
//  <teléfono> <dia> <mes> <año> <RC>
//  <nombre> <RC>
//  <teléfono> <dia> <mes> <año> <RC>
//  ...
//--------------------------------------------------------------------------
void escribir_fecha(ofstream& fich, const Fecha& f)
{
    fich << f.dia << ' ' << f.mes << ' ' << f.anyo ;
}
void escribir_persona(ofstream& fich, const Persona& p)
{
    fich << p.nombre << endl ;
    fich << p.tfn << ' ' ;
    escribir_fecha(fich, p.fnac) ;
    fich << endl ;
}
void escribir_agenda(const string& nombre_fich, const Agenda& ag, bool& ok)
{
    ofstream fich ;

    fich.open(nombre_fich.c_str()) ;
    if (fich.fail()) {
        ok = false ;
    } else {
        int i = 0 ;
        while ((i < ag.nelms) && (! fich.fail())) {
            escribir_persona(fich, ag.elm[i]) ;
            ++i ;
        }
        ok = ! fich.fail() ;
        fich.close() ;
    }
}
void guardar_agenda(const Agenda& ag)
{
    bool ok ;
    string nombre_fich ;
    cout << "Introduce el nombre del fichero: " ;
    cin >> nombre_fich ;
    escribir_agenda(nombre_fich, ag, ok) ;
    if (!ok) {
        cout << "Error al guardar el fichero" << endl ;
    }
}
//--------------------------------------------------------------------------
char menu()
{
    char op ;
```

```
    cout << endl ;
    cout << "C. Cargar Agenda" << endl ;
    cout << "M. Mostrar Agenda" << endl ;
    cout << "N. Nueva Persona" << endl ;
    cout << "G. Guardar Agenda" << endl ;
    cout << "X. Fin" << endl ;
    do {
        cout << endl << "    Opción: " ;
        cin >> op ;
        op = char(toupper(op)) ;
    } while (!((op == 'C')||(op == 'M')||(op == 'N')||(op == 'G')||(op == 'X'))) ;
    cout << endl ;
    return op ;
}
//-------------------------------------------------------------------------
int main()
{
    Agenda ag ;
    char op ;
    inic_agenda(ag) ;
    do {
        op = menu() ;
        switch (op) {
        case 'C':
            cargar_agenda(ag) ;
            break ;
        case 'M':
            escribir_agenda(ag) ;
            break ;
        case 'N':
            nueva_persona(ag) ;
            break ;
        case 'G':
            guardar_agenda(ag) ;
            break ;
        }
    } while (op != 'X') ;
}
//-------------------------------------------------------------------------
```

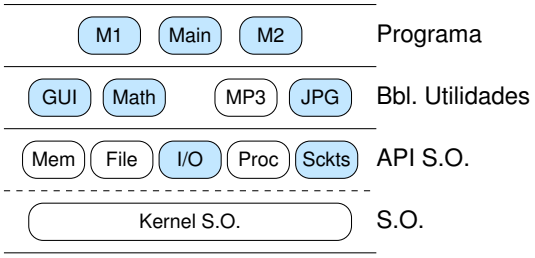
Capítulo 10

Módulos y Bibliotecas

Cuando se desarrollan programas de complejidad media/alta, el código fuente normalmente no se encuentra en un único fichero, sino que se encuentra distribuido entre varios módulos. Una primera ventaja de la existencia de módulos es que permiten aumentar la localidad y cohesión del código y aislarlo del exterior, es decir, poner todo el código encargado de resolver un determinado problema en un módulo nos permite aislarlo del resto, con lo que futuras modificaciones serán más fáciles de realizar.

Otra ventaja adicional de los módulos es el hecho de que si se modifica algo interno en un determinado módulo, sólo será necesario volver a compilar dicho módulo, y no todo el programa completo, lo que se convierte en una ventaja indudable en caso de programas grandes (*compilación separada*).

Además, esta división modular es una pieza fundamental para la *reutilización del código*, ya que permite la utilización de bibliotecas del sistema, así como la creación y distribución de bibliotecas de utilidades que podrán ser utilizadas por múltiples programas. Esta distribución de bibliotecas se puede hacer en *código objeto*, por lo que no es necesario distribuir el código fuente de la misma.



Así, vemos que en la figura un determinado programa se compone de varios módulos de programa (Main, M1 y M2) en los cuales está dividida la solución principal del problema, varios módulos de bibliotecas proporcionan utilidades gráficas, matemáticas y tratamiento de imágenes ; así como varios módulos de biblioteca dan acceso a servicios de entrada/salida y comunicaciones por Internet proporcionados por el sistema operativo.

10.1. Interfaz e Implementación del Módulo

En el lenguaje de programación C++, normalmente un módulo se compone de dos ficheros: uno donde aparece el código que resuelve un determinado problema o conjunto de problemas (implementación – parte privada), y un fichero que contiene las definiciones de tipos, constantes y prototipos de subprogramas que el módulo ofrece (interfaz – parte pública). Así, se denomina la *implementación* del módulo al fichero que contiene la parte privada del módulo, y se denomina la *interfaz* del módulo al fichero que contiene la parte pública del mismo. A este fichero también se le denomina "fichero de encabezamiento" o "fichero de cabecera" (*header file* en inglés).

main.cpp (Principal)
```
#include "complejo.hpp"
// utilización de complejo
int main()
{
   ...
}
```

complejo.hpp (Interfaz)
```
#ifndef _complejo_hpp_
#define _complejo_hpp_
// interfaz de complejo
// público
...
#endif
```

complejo.cpp (Implementación)
```
#include "complejo.hpp"
// implementación de complejo
// privado
...
...
...
```

Por lo tanto, un programa completo normalmente se compone de varios módulos, cada uno con su fichero de encabezamiento (interfaz) y de implementación, y de un módulo principal donde reside la función principal `main`. Para que un determinado módulo pueda hacer uso de las utilidades que proporciona otro módulo, deberá incluir el fichero de encabezamiento (interfaz) del módulo que se vaya a utilizar, de tal forma que tenga acceso a las declaraciones públicas de éste. Así mismo, el fichero de implementación de un determinado módulo también deberá especificar *al comienzo* del mismo la inclusión del fichero de encabezamiento de su propio módulo, con objeto de obtener y contrastar las definiciones allí especificadas, es decir, el fichero de encabezamiento del propio módulo debe ser el primer fichero que se incluya en el fichero de implementación del mismo módulo. Normalmente los ficheros de implementación tendrán una extensión ".cpp" (también suelen utilizarse otras extensiones como ".cxx" y ".cc") y los ficheros de encabezamiento tendrán una extensión ".hpp" (también suelen utilizarse otras extensiones como ".hxx", ".hh" y ".h"). Así, para incluir el fichero de encabezamiento (interfaz) de un módulo `complejo` se utiliza la siguiente directiva:

```
#include "complejo.hpp"
```

Nótese que cuando se incluyen ficheros de encabezamiento (interfaz) de la biblioteca estándar (o del sistema), el nombre del fichero se especifica entre los siguientes símbolos `<...>`, pero cuando se incluyen ficheros de encabezamiento de módulos y bibliotecas locales (no estándares), entonces el nombre del fichero se especifica entre los siguientes símbolos `"..."`. De esta forma, los ficheros de la biblioteca estándar y del sistema se buscan en directorios del sistema, pero los ficheros y bibliotecas locales se buscan en el directorio local de trabajo.

Guardas en un Fichero de Encabezamiento

Las definiciones en los ficheros de encabezamiento (interfaz) serán especificadas entre las *guardas* (directivas de compilación condicional) para evitar la inclusión duplicada de las definiciones allí contenidas. El nombre de la guarda usualmente se deriva del nombre del fichero, como se indica en el siguiente ejemplo donde el módulo `complejo` tendrá los siguientes ficheros de encabezamiento y de implementación (en determinadas circunstancias, puede ser conveniente que al nombre de la guarda se le añada también el nombre del espacio de nombres que se explicará en la siguiente sección):

Fichero: `complejo.hpp` (Interfaz)
```
// Guarda para evitar inclusión duplicada
#ifndef _complejo_hpp_
#define _complejo_hpp_

// Definiciones Públicas de:
//  * Constantes
//  * Tipos (Enum, Registros, Clases)
//  * Prototipos de Subprogramas

#endif // Fin de guarda
```

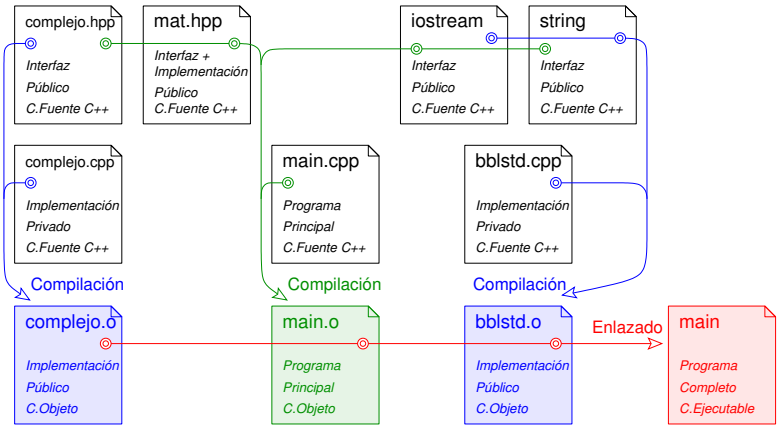
Fichero: `complejo.cpp` (Implementación)
```
#include "complejo.hpp"



// Implementaciones Privadas de:
//  * Constantes Privadas
//  * Tipos Privados
//  * Subprogramas
//  * Clases
```

Directrices para el Diseño de Ficheros de Encabezamiento

Las siguientes directrices deben ser tenidas en cuenta con objeto de organizar adecuadamente el diseño de los ficheros de encabezamiento de los módulos:

- Un fichero de encabezamiento sólo deberá contener definiciones de constantes, definiciones de tipos y prototipos de subprogramas que exporta (parte pública) el propio módulo. No deberá contener definiciones de variables globales, ni la implementación de código (de subprogramas y métodos). Esto último salvo algunas excepciones, tales como la definición de subprogramas simples "en línea" (véase 5.6) y la definición de subprogramas y clases genéricas (véase 12).
- El mecanismo de inclusión de ficheros de encabezamiento debe ser robusto ante posibles inclusiones duplicadas. Para ello siempre se utilizará el mecanismo de guardas explicado anteriormente.
- Un fichero de encabezamiento debe incluir todos los ficheros de encabezamiento de otros módulos que necesite para su propia definición, de forma tal que el orden de inclusión de los ficheros de encabezamiento no sea importante.

10.2. Compilación Separada y Enlazado



Cuando se compila un módulo de forma independiente (compilación separada), se compila su fichero de implementación, por ejemplo `complejo.cpp`, y produce como resultado un fichero en *código objeto*, por ejemplo `complejo.o`, considerando que el código fuente en C++ compilado es el contenido del fichero de implementación junto con el contenido de todos los ficheros de encabezamiento incluidos durante el proceso de compilación. Por ejemplo, mediante el siguiente comando se compila un módulo de implementación de código fuente en C++ utilizando el compilador *GNU GCC* para generar el correspondiente código objeto:

```
g++ -ansi -Wall -Werror -c complejo.cpp
g++ -ansi -Wall -Werror -c main.cpp
```

y el enlazado de los códigos objeto para generar el código ejecutable:

```
g++ -ansi -Wall -Werror -o main main.o complejo.o
```

Aunque también es posible realizar la compilación y enlazado en el mismo comando:

```
g++ -ansi -Wall -Werror -o main main.cpp complejo.cpp
```

o incluso mezclar compilación de código fuente y enlazado de código objeto:

```
g++ -ansi -Wall -Werror -o main main.cpp complejo.o
```

Hay que tener en cuenta que el compilador enlaza automáticamente el código generado con las bibliotecas estándares de C++, y por lo tanto no es necesario que éstas se especifiquen explícitamente. Sin embargo, en caso de ser necesario, también es posible especificar el enlazado con bibliotecas externas:

```
g++ -ansi -Wall -Werror -o main main.cpp complejo.cpp -ljpeg
```

Estas bibliotecas no son más que una agregación de módulos compilados a código objeto, y organizadas adecuadamente para que puedan ser reutilizados por muy diversos programas.

10.3. Espacios de Nombre

Cuando se trabaja con múltiples módulos y bibliotecas, es posible que se produzcan *colisiones* en la definición de entidades diferentes con los mismos identificadores proporcionadas por diferentes módulos y bibliotecas. Este hecho no está permitido por el lenguaje de programación C++. Para evitar estas posibles colisiones existen los *espacios de nombre* (*namespace* en inglés), que permiten agrupar bajo una misma denominación (jerarquía) un conjunto de declaraciones y definiciones, de tal forma que dicha denominación será necesaria para identificar y diferenciar cada entidad declarada. Así, para definir un espacio de nombres se utiliza la palabra reservada `namespace` seguida por el *identificador* del espacio de nombres, y entre *llaves* las declaraciones y definiciones que deban estar bajo dicha jerarquía del espacio de nombres.

Estos espacios de nombre pueden ser únicos para un determinado módulo, o por el contrario pueden abarcar múltiples módulos y bibliotecas gestionados por el mismo proveedor, por ejemplo todas las entidades definidas en la biblioteca estándar se encuentran bajo el espacio de nombres `std`. Así, el identificador del espacio de nombres puede ser derivado del propio nombre del fichero, puede incluir una denominación relativa al proveedor del módulo, o alguna otra denominación más compleja que garantice que no habrá colisiones en el identificador del espacio de nombres. Por ejemplo, podemos definir el módulo *complejo* dentro del espacio de nombres *umalcc*, que haría referencia a un proveedor del departamento de Lenguajes y Ciencias de la Computación de la Universidad de Málaga.

main.cpp (Principal)

```
#include <iostream>
#include "complejo.hpp"
using namespace std ;
using namespace umalcc ;
// utilización de complejo
int main()
{
   ...
}
```

complejo.hpp (Interfaz)

```
#ifndef _complejo_hpp_
#define _complejo_hpp_
#include <...otros...>
// interfaz de complejo
namespace umalcc {
   ...
   ...
}
#endif
```

complejo.cpp (Implementación)

```
#include "complejo.hpp"
#include <...otros...>
// implementación de complejo
namespace umalcc {
   ...
   ...
   ...
   ...
}
```

Nótese que la inclusión de ficheros de encabezamiento se debe realizar externamente a la definición de los espacios de nombre.

Utilización de Espacios de Nombre

Una vez que las entidades han sido definidas dentro de un espacio de nombres, éstas no pueden ser utilizadas directamente, sino que es necesario algún tipo de *cualificación* que permita referenciar e identificar a las entidades dentro de los espacios de nombre en los que han sido definidas. Ello se puede realizar de varias formas, dependiendo de las circunstancias donde se produzca esta utilización:

- Todos los identificadores definidos dentro de un espacio de nombres determinado son visibles y accesibles directamente desde dentro del mismo espacio de nombres, sin necesidad de cualificación.
- En la implementación de los módulos (ficheros de implementación `.cpp`), mediante la directiva `using namespace` se ponen disponibles (accesibles) todos los identificadores de dicho espacio de nombres completo, que podrán ser accedidos directamente, sin necesidad de cualificación explícita. Por ejemplo:

```
using namespace std ;
using namespace umalcc ;
```

- En los ficheros de encabezamiento (`.hpp`) cada identificador externo (perteneciente a otro espacio de nombres) se debe utilizar cualificado con el espacio de nombres al que pertenece *(cualificación explícita)* utilizando para ello el *identificador* del espacio de nombres, seguido por el operador `::` y del *identificador* de la entidad que se esté utilizando, como en el siguiente ejemplo para utilizar el tipo `string` del espacio de nombres `std`, o el tipo `array` del espacio de nombres `std::tr1`.

```
namespace umalcc {
    struct Persona {
        std::string nombre ;
        int edad ;
    } ;
    typedef std::tr1::array<int, 20> Vector ;
    void leer(std::string& nombre) ;
}
```

Si se utilizan (mediante *using namespace*) varios espacios de nombre simultáneamente y ambos definen el mismo identificador, si dicho identificador no se utiliza, entonces no se produce *colisión*. Sin embargo en caso de que se utilice dicho identificador, entonces se produce una *colisión* ya que el mismo identificador se encuentra definido (y es accesible) en dos espacios de nombre diferentes, por lo que el compilador no puede discernir por sí mismo a que entidad se refiere. En este último caso, el programador debe utilizar la *cualificación explícita* para este identificador y eliminar de esta forma la ambigüedad en su utilización.

main.cpp

```
#include <iostream>
#include <string>
#include "datos.hpp"
using namespace std ;
using namespace umalcc ;
// utilización de datos
int main()
{
   string colision ;
   std::string nombre_1 ;
   umalcc::string nombre_2 ;
   ...
}
```

datos.hpp

```
#ifndef _datos_hpp_
#define _datos_hpp_
#include <tr1/array>
#include <...otros...>
// interfaz de datos
namespace umalcc {
   struct string {
      std::tr1::array<char, 50> datos ;
      int size ;
   } ;
   ...
}
#endif
```

datos.cpp

```
#include "datos.hpp"
#include <...otros...>
using namespace std ;
namespace umalcc {
   ...
   ...
   ...
   ...
   ...
   ...
   ...
   ...
}
```

Es importante remarcar que no es adecuado aplicar la directiva `using namespace` dentro de ficheros de encabezamiento, ya que si es utilizada en un fichero de encabezamiento que se incluye por múltiples módulos, entonces pondría disponible (accesible) todos los identificadores de dicho espacio de nombres para todos los ficheros que incluyan (`include`) dicho fichero de encabezamiento, algo que podría provocar colisiones inesperadas, y esto sería un efecto colateral no deseado para aquellos que utilizasen dicho módulo (incluyeran dicho fichero de encabezamiento).

Espacios de Nombre Anónimos

Los *espacios de nombre anónimos* permiten definir entidades privadas internas a los módulos de implementación, de tal forma que no puedan producir colisiones con las entidades públicas del sistema completo. De esta forma, cualquier declaración y definición realizada dentro de un espacio de nombres anónimo será únicamente visible en el módulo de implementación donde se encuentre (privada), pero no será visible en el exterior del módulo.

Adicionalmente, también es posible definir espacios de nombre anidados (dentro de otros espacios de nombre), pudiendo, de esta forma, definir jerarquías de espacios de nombre.

Ejemplo

Módulo Números Complejos

Ejemplo de módulo para definir operaciones con números complejos, donde el fichero de encabezamiento podría ser:

```
//- fichero: complejos.hpp ------------------------------------------
#ifndef _complejos_hpp_
#define _complejos_hpp_
namespace umalcc {
    //---------------------------------
    const double ERROR_PRECISION = 1e-6 ;
    //---------------------------------
    struct Complejo {
        double real ; // parte real del numero complejo
        double imag ; // parte imaginaria del numero complejo
    } ;
    //---------------------------------
    void sumar(Complejo& r, const Complejo& a, const Complejo& b) ;
    // Devuelve un numero complejo (r) que contiene el resultado de
    // sumar los numeros complejos (a) y (b).
    //---------------------------------
    void restar(Complejo& r, const Complejo& a, const Complejo& b) ;
    // Devuelve un numero complejo (r) que contiene el resultado de
    // restar los numeros complejos (a) y (b).
    //---------------------------------
    void multiplicar(Complejo& r, const Complejo& a, const Complejo& b) ;
    // Devuelve un numero complejo (r) que contiene el resultado de
    // multiplicar los numeros complejos (a) y (b).
    //---------------------------------
    void dividir(Complejo& r, const Complejo& a, const Complejo& b) ;
    // Devuelve un numero complejo (r) que contiene el resultado de
    // dividir los numeros complejos (a) y (b).
    //---------------------------------
    bool iguales(const Complejo& a, const Complejo& b) ;
    // Devuelve true si los numeros complejos (a) y (b) son iguales.
    //---------------------------------
    void escribir(const Complejo& a) ;
    // muestra en pantalla el numero complejo (a)
    //---------------------------------
    void leer(Complejo& a) ;
    // lee de teclado el valor del numero complejo (a).
    // lee la parte real y la parte imaginaria del numero
    //---------------------------------
}
#endif
//- fin: complejos.hpp -------------------------------------------
```

La implementación del módulo se realiza en un fichero independiente, donde además se puede apreciar la utilización de un espacio de nombres anónimo:

```
//- fichero: complejos.cpp ------------------------------------------
#include "complejos.hpp"
#include <iostream>
using namespace std ;
using namespace umalcc ;
//-----------------------------------------------------------------------------
// Espacio de nombres anonimo. Es una parte privada de la
// implementacion. No es accesible desde fuera del modulo
```

```
//------------------------------------------------------------------------------
namespace {
    //---------------------------------
    //-- Subprogramas Auxiliares -------
    //---------------------------------
    // cuadrado de un numero (a^2)
    inline double sq(double a)
    {
        return a*a ;
    }
    //---------------------------------
    // Valor absoluto de un numero
    inline double abs(double a)
    {
        if (a < 0) {
            a = -a ;
        }
        return a ;
    }
    //---------------------------------
    // Dos numeros reales son iguales si la distancia que los
    // separa es lo suficientemente pequenya
    inline bool iguales(double a, double b)
    {
        return abs(a-b) <= ERROR_PRECISION ;
    }
}
//------------------------------------------------------------------------------
// Espacio de nombres umalcc.
// Aqui reside la implementacion de la parte publica del modulo
//------------------------------------------------------------------------------
namespace umalcc {
    //---------------------------------
    //-- Implementación ----------------
    //---------------------------------
    // Devuelve un numero complejo (r) que contiene el resultado de
    // sumar los numeros complejos (a) y (b).
    void sumar(Complejo& r, const Complejo& a, const Complejo& b)
    {
        r.real = a.real + b.real ;
        r.imag = a.imag + b.imag ;
    }
    //---------------------------------
    // Devuelve un numero complejo (r) que contiene el resultado de
    // restar los numeros complejos (a) y (b).
    void restar(Complejo& r, const Complejo& a, const Complejo& b)
    {
        r.real = a.real - b.real ;
        r.imag = a.imag - b.imag ;
    }
    //---------------------------------
    // Devuelve un numero complejo (r) que contiene el resultado de
    // multiplicar los numeros complejos (a) y (b).
    void multiplicar(Complejo& r, const Complejo& a, const Complejo& b)
    {
        r.real = (a.real * b.real) - (a.imag * b.imag) ;
        r.imag = (a.real * b.imag) + (a.imag * b.real) ;
    }
    //---------------------------------
```

```
    // Devuelve un numero complejo (r) que contiene el resultado de
    // dividir los numeros complejos (a) y (b).
    void dividir(Complejo& r, const Complejo& a, const Complejo& b)
    {
        double divisor = sq(b.real) + sq(b.imag) ;
        if (::iguales(0.0, divisor)) {
            r.real = 0 ;
            r.imag = 0 ;
        } else {
            r.real = ((a.real * b.real) + (a.imag * b.imag)) / divisor ;
            r.imag = ((a.imag * b.real) - (a.real * b.imag)) / divisor ;
        }
    }
    //---------------------------------
    // Devuelve true si los numeros complejos (a) y (b) son iguales.
    bool iguales(const Complejo& a, const Complejo& b)
    {
        return ::iguales(a.real, b.real) && ::iguales(a.imag, b.imag) ;  }
    //---------------------------------
    // muestra en pantalla el numero complejo (a)
    void escribir(const Complejo& a)
    {
        cout << "{ " << a.real << ", " << a.imag << " }" ;
    }
    //---------------------------------
    // lee de teclado el valor del numero complejo (a).
    // lee la parte real y la parte imaginaria del numero
    void leer(Complejo& a)
    {
        cin >> a.real >> a.imag ;
    }
    //---------------------------------
}
//- fin: complejos.cpp -------------------------------------------------
```

Un ejemplo de utilización del módulo de números complejos podría ser:

```
//- fichero: main.cpp ----------------------------------------
#include <iostream>
#include "complejos.hpp"
using namespace std ;
using namespace umalcc ;
//---------------------------------
void leer(Complejo& c)
{
    cout << "Introduzca un numero complejo { real img }: " ;
    // cualificacion explícita del espacio de nombres uamalcc
    // para evitar colisión en invocación a subprograma
    // leer(Complejo& c) del espacio de nombres umalcc
    umalcc::leer(c) ;
}
//---------------------------------
void prueba_suma(const Complejo& c1, const Complejo& c2)
{
    Complejo c0 ;
    sumar(c0, c1, c2) ;
    escribir(c1) ;
    cout <<" + " ;
    escribir(c2) ;
    cout <<" = " ;
```

```
    escribir(c0) ;
    cout << endl ;
    Complejo aux ;
    restar(aux, c0, c2) ;
    if (! iguales(c1, aux)) {
        cout << "Error en operaciones de suma/resta"<< endl ;
    }
}
//-----------------------------------
void prueba_resta(const Complejo& c1, const Complejo& c2)
{
    Complejo c0 ;
    restar(c0, c1, c2) ;
    escribir(c1) ;
    cout <<" - " ;
    escribir(c2) ;
    cout <<" = " ;
    escribir(c0) ;
    cout << endl ;
    Complejo aux ;
    sumar(aux, c0, c2) ;
    if (! iguales(c1, aux)) {
        cout << "Error en operaciones de suma/resta"<< endl ;
    }
}
//-----------------------------------
void prueba_mult(const Complejo& c1, const Complejo& c2)
{
    Complejo c0 ;
    multiplicar(c0, c1, c2) ;
    escribir(c1) ;
    cout <<" * " ;
    escribir(c2) ;
    cout <<" = " ;
    escribir(c0) ;
    cout << endl ;
    Complejo aux ;
    dividir(aux, c0, c2) ;
    if (! iguales(c1, aux)) {
        cout << "Error en operaciones de mult/div"<< endl ;
    }
}
//-----------------------------------
void prueba_div(const Complejo& c1, const Complejo& c2)
{
    Complejo c0 ;
    dividir(c0, c1, c2) ;
    escribir(c1) ;
    cout <<" / " ;
    escribir(c2) ;
    cout <<" = " ;
    escribir(c0) ;
    cout << endl ;
    Complejo aux ;
    multiplicar(aux, c0, c2) ;
    if (! iguales(c1, aux)) {
        cout << "Error en operaciones de mult/div"<< endl ;
    }
}
```

```
//-----------------------------------
int main()
{
    Complejo c1, c2 ;
    // cualificación explícita del espacio de nombres global
    // para evitar colisión en invocación al subprograma
    // leer(Complejo& c) del espacio de nombres global
    ::leer(c1) ;
    ::leer(c2) ;
    //--------------------------------
    prueba_suma(c1, c2) ;
    prueba_resta(c1, c2) ;
    prueba_mult(c1, c2) ;
    prueba_div(c1, c2) ;
    //--------------------------------
}
//- fin: main.cpp -----------------------------------------------
```

Su compilación separada y enlazado en *GNU GCC*:

```
g++ -ansi -Wall -Werror -c complejos.cpp
g++ -ansi -Wall -Werror -c main.cpp
g++ -ansi -Wall -Werror -o main main.o complejos.o
```

Alternativamente se puede realizar en dos pasos:

```
g++ -ansi -Wall -Werror -c complejos.cpp
g++ -ansi -Wall -Werror -o main main.cpp complejos.o
```

o incluso en un único paso:

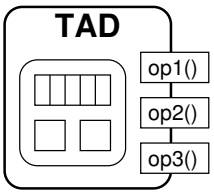
```
g++ -ansi -Wall -Werror -o main main.cpp complejos.cpp
```

Capítulo 11

Tipos Abstractos de Datos

A medida que aumenta la complejidad del problema a resolver, del mismo modo deben aumentar los niveles de abstracción necesarios para diseñar y construir su solución algorítmica. Así, la abstracción procedimental permite aplicar adecuadamente técnicas de *diseño descendente* y *refinamientos sucesivos* en el desarrollo de algoritmos y programas. La programación modular permite aplicar la abstracción a mayor escala, permitiendo abstraer sobre conjuntos de operaciones y los datos sobre los que se aplican. De esta forma, a medida que aumenta la complejidad del problema a resolver, aumenta también la complejidad de las estructuras de datos necesarias para su resolución, y este hecho requiere, así mismo, la aplicación de la abstracción a las estructuras de datos.

La aplicación de la abstracción a las estructuras de datos da lugar a los *Tipos Abstractos de Datos* (TAD), donde se especifica el concepto que representa un determinado tipo de datos, y la semántica (el significado) de las operaciones que se le pueden aplicar, pero donde su representación e implementación internas permanecen ocultas e inaccesibles desde el exterior, de tal forma que no son necesarias para su utilización. Así, podemos considerar que un tipo abstracto de datos *encapsula* una determinada estructura abstracta de datos, impidiendo su manipulación directa, permitiendo solamente su manipulación a través de las operaciones especificadas. De este modo, los tipos abstractos de datos proporcionan un mecanismo adecuado para el diseño y reutilización de software fiable y robusto.



Para un determinado tipo abstracto de datos, se pueden distinguir tres niveles:

- Nivel de utilización, donde se utilizan objetos de un determinado tipo abstracto de datos, basándose en la especificación del mismo, de forma independiente a su implementación y representación concretas. Así, estos objetos se manipulan mediante la invocación a las operaciones especificadas en el TAD.

- Nivel de especificación, donde se especifica el tipo de datos, el concepto abstracto que representa y la semántica y restricciones de las operaciones que se le pueden aplicar. Este nivel representa el *interfaz* público del tipo abstracto de datos.

Utilización de TAD
Especificación de TAD
Implementación de TAD

- Nivel de implementación, donde se define e implementa tanto las estructuras de datos que soportan la abstracción, como las operaciones que actúan sobre ella según la semántica especificada. Este nivel interno permanece privado, y no es accesible desde el exterior del tipo abstracto de datos.

Nótese que para una determinada especificación de un tipo abstracto de datos, su implementación puede cambiar sin que ello afecte a la utilización del mismo.

11.1. Tipos Abstractos de Datos en C++: Clases

En el lenguaje de programación C++, las clases dan la posibilidad al programador de definir tipos abstractos de datos, de tal forma que permiten definir su representación interna (compuesta por sus atributos miembros), la forma en la que se crean y se destruyen, como se asignan y se pasan como parámetros, y las operaciones que se pueden aplicar (denominadas funciones miembro o simplemente métodos). De esta forma se hace el lenguaje extensible. Así mismo, la definición de tipos abstractos de datos mediante clases puede ser combinada con la definición de módulos (véase 10), haciendo de este modo posible la reutilización de estos nuevos tipos de datos.

Así, en C++ una determinada *clase* define un determinado *tipo abstracto de datos*, y un *objeto* se corresponde con una determinada *instancia de una clase*, de igual forma que una *variable* se corresponde con una determinada *instancia de un tipo de datos*.

Aunque C++ permite implementar las clases utilizando una definición *en línea*, en este capítulo nos centraremos en la implementación separada de los métodos de las clases. Además, combinaremos la especificación de la clase y su implementación con los conceptos de programación modular vistos en el capítulo anterior (véase 10), de tal forma que la definición de la clase se realizará en el fichero de cabecera (`hpp`) de un determinado módulo, y la implementación de la clase se realizará en el fichero de implementación (`cpp`) del módulo.

11.1.1. Definición de Clases

La definición de la clase se realizará en el fichero de cabecera (`hpp`) de un determinado módulo, dentro de las guardas y espacio de nombres adecuado. Para ello, se especifica la palabra reservada `class` seguida por el identificador de la nueva clase (tipo) que se está definiendo, y entre llaves la definición de los atributos (miembros) que lo componen y de los métodos (funciones miembros) que se le pueden aplicar directamente a los objetos de la clase. Finalmente el delimitador *punto y coma* (`;`) debe seguir al delimitador *cierra-llaves* (`}`).

```
//- fichero: complejos.hpp ------------------------------------------
#ifndef _complejos_hpp_
#define _complejos_hpp_
namespace umalcc {
    class Complejo {
        // ...
    } ;
}
#endif
//- fin: complejos.hpp ----------------------------------------------
```

Zona Privada y Zona Pública

En la definición de una clase, se pueden distinguir dos ámbitos de visibilidad (accesibilidad), la parte *privada*, cuyos miembros sólo serán accesibles desde un ámbito *interno* a la propia clase, y la parte *pública*, cuyos miembros son accesibles tanto desde un ámbito *interno* como desde un ámbito *externo* a la clase.

La parte privada comprende desde el principio de la definición de la clase hasta la etiqueta `public:`, y la parte pública comprende desde esta etiqueta hasta que se encuentra otra etiqueta `private:`. Cada vez que se especifica una de las palabras reservadas `public:` o `private:`, las declaraciones que la siguen adquieren el atributo de visibilidad dependiendo de la etiqueta especificada.

```
class Complejo {
public:
    // ... zona pública ...
private:
    // ... zona privada ...
} ;
```

Atributos

Los atributos componen la representación interna de la clase, y se definen usualmente en la zona de visibilidad privada de la clase, con objeto de proteger el acceso a dicha representación interna. Su definición se realiza de igual forma a los campos de los registros (véase 6.3).

De igual modo a los registros y sus campos, cada *objeto* que se defina, almacenará su propia representación interna de los atributos de forma independiente a los otros objetos (instancias de la misma clase).

```
class Complejo {
public:
    // ...
private:
    double real ; // parte real del numero complejo
    double imag ; // parte imaginaria del numero complejo
} ;
```

El Constructor por Defecto

El *constructor* de una clase permite construir e inicializar un *objeto*. El *constructor por defecto* es el mecanismo por defecto utilizado para construir objetos de este tipo cuando no se especifica ninguna forma explícita de construcción. Así, será el encargado de construir el objeto con los valores iniciales adecuados en el momento en que sea necesaria dicha construcción, por ejemplo cuando el flujo de ejecución alcanza la declaración de una variable de dicho tipo (véase 11.1.2).

Los constructores se declaran con el mismo identificador de la clase, seguidamente se especifican entre paréntesis los parámetros necesarios para la construcción, que en el caso del constructor por defecto, serán vacíos.

```
class Complejo {
public:
    Complejo() ;                              // Constructor por Defecto
    // ...
private:
    // ...
} ;
```

Métodos Generales y Métodos Constantes

Los métodos se corresponden con las operaciones que permiten manipular de muy diversa forma el estado interno de un determinado objeto como instancia de una determinada clase. Puede haber métodos definidos en el ámbito público de la clase, en cuyo caso podrán ser invocados tanto desde métodos internos de la clase, como desde el exterior de la clase, y métodos definidos en el ámbito privado de la clase, en cuyo caso sólo podrán ser invocados desde métodos internos de la clase. Estos métodos definidos en el ámbito privado de la clase suelen ser definidos como métodos auxiliares que facilitan la implementación de otros métodos más complejos.

Los métodos se declaran como los prototipos de los subprogramas (véase 5.7), pero teniendo en cuenta son aplicados a un objeto instancia de la clase a la que pertenece, y que por lo tanto no es necesario que sea recibido como parámetro.

Los métodos de una clase pueden tener el cualificador `const` especificado después de los parámetros, en cuyo caso indica que el método no modifica el estado interno del objeto, por lo que se puede aplicar tanto a objetos constantes como variables. En otro caso, si dicho cualificador no aparece, entonces significa que el método si modifica el estado interno del objeto, por lo que sólo podrá ser aplicado a objetos variables, y por el contrario no podrá ser aplicado a objetos constantes.

```
class Complejo {
public:
    // ...
```

```
    //----------------------------
    void sumar(const Complejo& a, const Complejo& b) ;
    // asigna al numero complejo (actual) el resultado de
    // sumar los numeros complejos (a) y (b).
    //----------------------------
    bool igual(const Complejo& b) const ;
    // Devuelve true si el numero complejo (actual) es
    // igual al numero complejo (b)
    //----------------------------
    void escribir() const ;
    // muestra en pantalla el numero complejo (actual)
    //----------------------------
    void leer() ;
    // lee de teclado el valor del numero complejo (actual).
    // lee la parte real y la parte imaginaria del numero
    //----------------------------
    // ...
private:
    // ...
} ;
```

11.1.2. Utilización de Clases

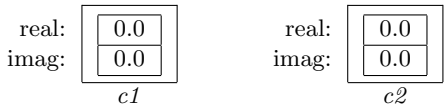
Un tipo abstracto de datos, definido como una clase encapsulada dentro de un módulo, puede ser utilizado por cualquier otro módulo que lo necesite. Para ello, deberá incluir el fichero de cabecera donde se encuentra la definición de la clase, y podrá definir tantos objetos (instancias de dicha clase) como sean necesarios, para ello, deberá utilizar cualificación explícita o implícita dependiendo del contexto de su utilización (ficheros de encabezamiento o de implementación respectivamente).

Instancias de Clase: Objetos

Un objeto es una instancia de una clase, y podremos definir tantos objetos cuyo tipo sea de una determinada clase como sea necesario, de tal modo que cada objeto contiene su propia representación interna de forma independiente del resto.

La definición de un objeto de una determinada clase se realiza de igual forma a la definición de una variable (o constante) de un determinado tipo, de tal forma que cada objeto será una instancia independiente de una determinada clase (tipo abstracto de datos).

```
//- fichero: main.cpp -----------------------------------------------
#include <iostream>
#include "complejos.hpp"
using namespace std ;
using namespace umalcc ;

int main()
{
    Complejo c1, c2 ;
    // ...
}
//- fin: main.cpp ---------------------------------------------------
```



Es importante remarcar que cada objeto, definido de una determinada clase, es una instancia independiente de los otros objetos definidos de la misma clase, con su propia memoria para contener de forma independiente el estado de su representación interna.

Tiempo de Vida de los Objetos

Durante la ejecución del programa, cuando el flujo de ejecución llega a la sentencia donde se define un determinado objeto, entonces se reserva espacio en memoria para contener a dicho objeto,

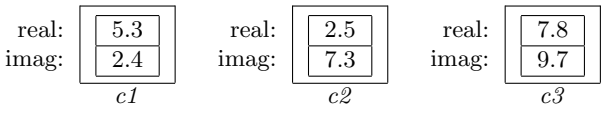
y se invoca al constructor especificado (si no se especifica ningún constructor, entonces se invoca al constructor por defecto) para construir adecuadamente al objeto, siendo de esta forma accesible desde este punto de construcción hasta que el flujo de ejecución alcanza el final de bloque donde el objeto ha sido definido, en cuyo caso el objeto se destruye (invocando a su *destructor*) y se libera la memoria que ocupaba, pasando de este modo a estar inaccesible.

```
//- fichero: main.cpp -----------------------------------------------
#include <iostream>
#include "complejos.hpp"
using namespace std ;
using namespace umalcc ;

int main()
{
    Complejo c1 ;                    // construcción de c1 (1 vez)
    for (int i = 0 ; i < 3 ; ++i) {
        Complejo c2 ;                // construcción de c2 (3 veces)
        // ...
    }                                // destrucción de c2 (3 veces)
    // ...
}                                    // destrucción de c1 (1 vez)
//- fin: main.cpp ---------------------------------------------------
```

Manipulación de los Objetos

Una vez que un objeto es accesible, se puede manipular invocando a los métodos públicos definidos en su interfaz. Esta invocación de los métodos se aplica sobre un determinado objeto en concreto, y se realiza especificando el identificador del objeto sobre el que recae la invocación al método, seguido por el símbolo punto (`.`) y por la invocación al método en cuestión, es decir, el identificador del método y los parámetros actuales necesarios entre paréntesis.

```
//- fichero: main.cpp -----------------------------------------------
#include <iostream>
#include "complejos.hpp"
using namespace std ;
using namespace umalcc ;

int main()
{
    Complejo c1, c2, c3 ;            // construcción de c1, c2, c3
    c1.leer() ;
    c2.leer() ;
    c3.sumar(c1, c2) ;
    c3.escribir() ;
}                                    // destrucción de c1, c2, c3
//- fin: main.cpp ---------------------------------------------------
```



Paso de Parámetros de Objetos

Es importante considerar que las clases (tipos abstractos de datos) son *tipos compuestos*, y por lo tanto deben seguir las mismas convenciones para el paso de parámetros de tipos compuestos (véase 6.1), es decir, los parámetros de salida o entrada/salida se pasan por *referencia*, y los parámetros de entrada se pasan por *referencia constante*.

11.1.3. Implementación de Clases

La implementación de los métodos de la clase se realizará en el fichero de implementación (`cpp`) del módulo correspondiente, dentro del mismo espacio de nombres en el que fue realizada la definición de la clase en el fichero de cabecera.

En el fichero de implementación se podrán definir, dentro del espacio de nombres adecuado, las constantes, tipos y subprogramas auxiliares necesarios que nos faciliten la implementación de los métodos de la clase.

Para implementar un determinado constructor o método de la clase, dentro del mismo espacio de nombres que la definición de la clase, se *cualificará explícitamente* el identifidor del método con el identificador de la clase a la que pertenece.

```
//- fichero: complejos.cpp ----------------------------------------
#include "complejos.hpp"
namespace umalcc {
    // ...
    //----------------------------
    Complejo::Complejo()                        // Constructor por Defecto
        // ...
    //----------------------------
    void Complejo::sumar(const Complejo& a, const Complejo& b)
        // ...
    //----------------------------
    bool Complejo::igual(const Complejo& b) const
        // ...
    //----------------------------
    void Complejo::escribir() const
        // ...
    //----------------------------
    void Complejo::leer()
        // ...
    //----------------------------
    // ...
}
//- fin: complejos.cpp ----------------------------------------------
```

Métodos

En la implementación de un determinado método de una clase, éste método puede invocar directamente a cualquier otro método de la clase sin necesidad de aplicar el operador punto (`.`). Así mismo, un método de la clase puede *acceder directamente a los atributos del objeto* sobre el que se invoque dicho método, sin necesidad de aplicar el operador punto (`.`), ni necesidad de recibirlo como parámetro. Por ejemplo:

```
void Complejo::sumar(const Complejo& a, const Complejo& b)
{
    real = a.real + b.real ;
    imag = a.imag + b.imag ;
}
```

Sin embargo, para acceder a los atributos de los objetos recibidos como parámetros, si son accesibles desde la implementación de una determinada clase, es necesario especificar el objeto (mediante su identificador) seguido por el operador punto (`.`) y a continuación el identificador del atributo en cuestión.

Así, podemos ver como para calcular la suma de números complejos, se asigna a las partes real e imaginaria del número complejo que estamos calculando (el objeto sobre el que se aplica el método `sumar`) la suma de las partes real e imaginaria respectivamente de los números complejos que recibe como parámetros. Por ejemplo, cuando se ejecuta la sentencia:

```
c3.sumar(c1, c2) ;
```

la sentencia correspondiente a la implementación del método `sumar(...)`:

```
    real = a.real + b.real ;
```

almacenará en el atributo `real` del número complejo `c3` el resultado de sumar los valores del atributo `real` de los números complejos `c1` y `c2`. De igual modo sucederá con el atributo `imag` del número complejo `c3`, que almacenará el resultado de sumar los valores del atributo `imag` de los números complejos `c1` y `c2`.

Constructores

En la implementación de los constructores de la clase, también será cualificado explícitamente con el identificador de la clase correspondiente. Después de la definición de los parámetros, a continuación del delimitador (`:`), se especifica la *lista de inicialización*, donde aparecen, separados por comas y según el orden de declaración, todos los atributos miembros del objeto, así como los valores con los que serán inicializados especificados entre paréntesis (se invoca al constructor adecuado según los parámetros especificados entre paréntesis, de tal forma que los paréntesis vacíos representan la construcción por defecto). A continuación se especifican entre llaves las sentencias pertenecientes al cuerpo del constructor para realizar las acciones adicionales necesarias para la construcción del objeto. Si no es necesario realizar ninguna acción adicional, entonces el cuerpo del constructor se dejará vacío.

Por ejemplo, implementaremos el constructor por defecto de la clase `Complejo` para que asigne el valor cero a cada componente (parte real e imaginaria) del objeto que se construya.

```
Complejo::Complejo()                          // Constructor por Defecto
    : real(0.0), imag(0.0) { }
```

11.1.4. Ejemplo

Por ejemplo, el *TAD número complejo* representa el siguiente concepto matemático de número complejo:

> Un número complejo representa un punto en el plano complejo, compuesto por dos componentes que representan la parte real y la parte imaginaria del número (abcisa y ordenada respectivamente en el plano cartesiano), al cual se le pueden aplicar las operaciones de suma, resta, multiplicación y división, así como la comparación de igualdad.

Definición

```
//- fichero: complejos.hpp ----------------------------------------
#ifndef _complejos_hpp_
#define _complejos_hpp_
namespace umalcc {
    //---------------------------------
    const double ERROR_PRECISION = 1e-6 ;
    //---------------------------------
    class Complejo {
    public:
        //-------------------------------------------------------
        //-- Métodos Públicos -----------------------------------
        //-------------------------------------------------------
        Complejo() ;                              // Constructor por Defecto
        //----------------------------
        double parte_real() const ;
        // devuelve la parte real del numero complejo
        //----------------------------
        double parte_imag() const ;
        // devuelve la parte imaginaria del numero complejo
        //----------------------------
        void sumar(const Complejo& a, const Complejo& b) ;
        // asigna al numero complejo (actual) el resultado de
        // sumar los numeros complejos (a) y (b).
```

```
        //----------------------------
        void restar(const Complejo& a, const Complejo& b) ;
        // asigna al numero complejo (actual) el resultado de
        // restar los numeros complejos (a) y (b).
        //----------------------------
        void multiplicar(const Complejo& a, const Complejo& b) ;
        // asigna al numero complejo (actual) el resultado de
        // multiplicar los numeros complejos (a) y (b).
        //----------------------------
        void dividir(const Complejo& a, const Complejo& b) ;
        // asigna al numero complejo (actual) el resultado de
        // dividir los numeros complejos (a) y (b).
        //----------------------------
        bool igual(const Complejo& b) const ;
        // Devuelve true si el numero complejo (actual) es
        // igual al numero complejo (b)
        //----------------------------
        void escribir() const ;
        // muestra en pantalla el numero complejo (actual)
        //----------------------------
        void leer() ;
        // lee de teclado el valor del numero complejo (actual).
        // lee la parte real y la parte imaginaria del numero
        //----------------------------
    private:
        //-------------------------------------------------------
        //-- Atributos Privados ---------------------------------
        //-------------------------------------------------------
        double real ; // parte real del numero complejo
        double imag ; // parte imaginaria del numero complejo
        //-------------------------------------------------------
    } ;
}
#endif
//- fin: complejos.hpp -------------------------------------------
```

Implementación

```
//- fichero: complejos.cpp ---------------------------------------
#include "complejos.hpp"
#include <iostream>
using namespace std ;
using namespace umalcc ;
//-----------------------------------------------------------------------------
// Espacio de nombres anonimo. Es una parte privada de la
// implementacion. No es accesible desde fuera del modulo
//-----------------------------------------------------------------------------
namespace {
    //-------------------------------------------------------
    //-- Subprogramas Auxiliares ----------------------------
    //-------------------------------------------------------
    // cuadrado de un numero (a^2)
    inline double sq(double a)
    {
        return a*a ;
    }
    //-------------------------
    // Valor absoluto de un numero
    inline double abs(double a)
```

```
    {
        return (a >= 0) ? a : -a ;
    }
    //-------------------------
    // Dos numeros reales son iguales si la distancia que los
    // separa es lo suficientemente pequenya
    inline bool iguales(double a, double b)
    {
        return abs(a-b) <= ERROR_PRECISION ;
    }
}
//-----------------------------------------------------------------------------
// Espacio de nombres umalcc.
// Aqui reside la implementacion de la parte publica del modulo
//-----------------------------------------------------------------------------
namespace umalcc {
    //-----------------------------------------------------------
    //-- Métodos Públicos -------------------------------------
    //-----------------------------------------------------------
    Complejo::Complejo()                        // Constructor por Defecto
        : real(0.0), imag(0.0) { }
    //----------------------------
    // devuelve la parte real del numero complejo
    double Complejo::parte_real() const
    {
        return real ;
    }
    //----------------------------
    // devuelve la parte imaginaria del numero complejo
    double Complejo::parte_imag() const
    {
        return imag ;
    }
    //----------------------------
    // asigna al numero complejo (actual) el resultado de
    // sumar los numeros complejos (a) y (b).
    void Complejo::sumar(const Complejo& a, const Complejo& b)
    {
        real = a.real + b.real ;
        imag = a.imag + b.imag ;
    }
    //----------------------------
    // asigna al numero complejo (actual) el resultado de
    // restar los numeros complejos (a) y (b).
    void Complejo::restar(const Complejo& a, const Complejo& b)
    {
        real = a.real - b.real ;
        imag = a.imag - b.imag ;
    }
    //----------------------------
    // asigna al numero complejo (actual) el resultado de
    // multiplicar los numeros complejos (a) y (b).
    void Complejo::multiplicar(const Complejo& a, const Complejo& b)
    {
        real = (a.real * b.real) - (a.imag * b.imag) ;
        imag = (a.real * b.imag) + (a.imag * b.real) ;
    }
    //----------------------------
    // asigna al numero complejo (actual) el resultado de
```

```
    // dividir los numeros complejos (a) y (b).
    void Complejo::dividir(const Complejo& a, const Complejo& b)
    {
        double divisor = sq(b.real) + sq(b.imag) ;
        if (iguales(0.0, divisor)) {
            real = 0.0 ;
            imag = 0.0 ;
        } else {
            real = ((a.real * b.real) + (a.imag * b.imag)) / divisor ;
            imag = ((a.imag * b.real) - (a.real * b.imag)) / divisor ;
        }
    }
    //----------------------------
    // Devuelve true si el numero complejo (actual) es
    // igual al numero complejo (b)
    bool Complejo::igual(const Complejo& b) const
    {
        return iguales(real, b.real) && iguales(imag, b.imag) ;
    }
    //----------------------------
    // muestra en pantalla el numero complejo (actual)
    void Complejo::escribir() const
    {
        cout << "{ " << real << ", " << imag << " }" ;
    }
    //----------------------------
    // lee de teclado el valor del numero complejo (actual).
    // lee la parte real y la parte imaginaria del numero
    void Complejo::leer()
    {
        cin >> real >> imag ;
    }
    //----------------------------
}
//- fin: complejos.cpp -------------------------------------------
```

Utilización

```
//- fichero: main.cpp --------------------------------------------
#include <iostream>
#include "complejos.hpp"
using namespace std ;
using namespace umalcc ;
//-----------------------------------
void leer(Complejo& c)
{
    cout << "Introduzca un numero complejo { real img }: " ;
    c.leer() ;
}
//-----------------------------------
void prueba_suma(const Complejo& c1, const Complejo& c2)
{
    Complejo c0 ;
    c0.sumar(c1, c2) ;
    c1.escribir() ;
    cout <<" + " ;
    c2.escribir() ;
    cout <<" = " ;
    c0.escribir() ;
```

```
    cout << endl ;
    Complejo aux ;
    aux.restar(c0, c2) ;
    if ( ! c1.igual(aux)) {
        cout << "Error en operaciones de suma/resta"<< endl ;
    }
}
//-----------------------------------
void prueba_resta(const Complejo& c1, const Complejo& c2)
{
    Complejo c0 ;
    c0.restar(c1, c2) ;
    c1.escribir() ;
    cout <<" - " ;
    c2.escribir() ;
    cout <<" = " ;
    c0.escribir() ;
    cout << endl ;
    Complejo aux ;
    aux.sumar(c0, c2) ;
    if ( ! c1.igual(aux)) {
        cout << "Error en operaciones de suma/resta"<< endl ;
    }
}
//-----------------------------------
void prueba_mult(const Complejo& c1, const Complejo& c2)
{
    Complejo c0 ;
    c0.multiplicar(c1, c2) ;
    c1.escribir() ;
    cout <<" * " ;
    c2.escribir() ;
    cout <<" = " ;
    c0.escribir() ;
    cout << endl ;
    Complejo aux ;
    aux.dividir(c0, c2) ;
    if ( ! c1.igual(aux)) {
        cout << "Error en operaciones de mult/div"<< endl ;
    }
}
//-----------------------------------
void prueba_div(const Complejo& c1, const Complejo& c2)
{
    Complejo c0 ;
    c0.dividir(c1, c2) ;
    c1.escribir() ;
    cout <<" / " ;
    c2.escribir() ;
    cout <<" = " ;
    c0.escribir() ;
    cout << endl ;
    Complejo aux ;
    aux.multiplicar(c0, c2) ;
    if ( ! c1.igual(aux)) {
        cout << "Error en operaciones de mult/div"<< endl ;
    }
}
//-----------------------------------
```

```
int main()
{
    Complejo c1, c2 ;
    leer(c1) ;
    leer(c2) ;
    //--------------------------------
    prueba_suma(c1, c2) ;
    prueba_resta(c1, c2) ;
    prueba_mult(c1, c2) ;
    prueba_div(c1, c2) ;
    //--------------------------------
}
//- fin: main.cpp -------------------------------------------------
```

11.2. Tipos Abstractos de Datos en C++: Más sobre Clases

Constantes de Ámbito de Clase

Las constantes de ámbito de clase se definen especificando los cualificadores `static const`, seguidos por el tipo, el identificador y el valor de la constante. Estas constantes serán comunes y accesibles a todas las instancias (objetos) de la clase. Por ejemplo, para definir la constante `MAX` con un valor de 256 en la zona privada de la clase:

```
class ListaInt {
public:
    // ...
private:
    static const int MAX = 256 ;
    // ...
} ;
```

Usualmente las constantes se definen en la zona privada de la clase, por lo que usualmente sólo serán accesibles internamente desde dentro de la clase. Sin embargo, en algunas situaciones puede ser conveniente definir la constante en la zona pública de la clase, entonces en este caso la constante podrá ser accedida desde el exterior de la clase, y será utilizada mediante cualificación explícita utilizando el identificador de la clase. Por ejemplo:

```
class ListaInt {
public:
    static const int MAX = 256 ;
    // ...
private:
    // ...
} ;
// ...
int main()
{
    int x = ListaInt::MAX ;
    // ...
}
```

Tipos de Ámbito de Clase

También se pueden definir tipos internos de ámbito de clase de igual forma a como se hace externamente a la clase, pero en este caso su ámbito de visibilidad estará restringido a la clase donde se defina. Estos tipos serán útiles en la definición de los atributos miembros de la clase, o para definir elementos auxiliares en la implementación del tipo abstracto de datos. Por ejemplo, para definir un tipo `Datos` como un array de 256 números enteros:

```
#include <tr1/array>
// ...
    class ListaInt {
    public:
        // ...
    private:
        static const int MAX = 256 ;
        typedef tr1::std::array<int, MAX> Datos ;
        struct Elemento {
            // ...
        } ;
        // ...
    } ;
```

Usualmente los tipos se definen en la zona privada de la clase, por lo que usualmente sólo serán accesibles internamente desde dentro de la clase. Sin embargo, en algunas situaciones puede ser conveniente definir el tipo en la zona pública de la clase, entonces en este caso el tipo podrá ser accedido desde el exterior de la clase, y será utilizado mediante cualificación explícita utilizando el identificador de la clase. Por ejemplo:

```
#include <tr1/array>
// ...
    class ListaInt {
    public:
        static const int MAX = 256 ;
        typedef tr1::std::array<int, MAX> Datos ;
        // ...
    private:
        // ...
    } ;
    // ...
int main()
{
    ListaInt::Datos d ;
    // ...
}
```

Constructores Específicos

Los *constructores* de una clase permiten construir e inicializar un *objeto*. Anteriormente se ha explicado el *constructor por defecto*, el cual se invoca cuando se crea un determinado objeto, y no se especifica que tipo de construcción se debe realizar. C++ permite, además, la definición e implementación de tantos constructores específicos como sean necesarios, para ello, se debe especificar en la lista de parámetros, aquellos que sean necesarios para poder construir el objeto adecuadamente en cada circunstancia específica, de tal forma que será la lista de parámetros formales la que permita discriminar que constructor será invocado dependiendo de los parámetros actuales utilizados en la invocación al constructor.

Por ejemplo, podemos definir un constructor específico para que reciba dos números reales como parámetros (parte real e imaginaria respectivamente de un número complejo), los cuales serán utilizados para dar los valores iniciales a cada atributo correspondiente del objeto que se construya. Así, su definición podría ser:

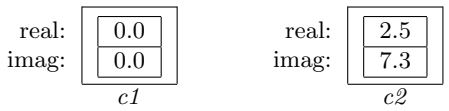
```
class Complejo {
public:
    Complejo(double p_real, double p_imag) ;  // Constructor Específico
} ;
```

A continuación se puede ver como sería la implementación de este constructor específico:

```
Complejo::Complejo(double p_real, double p_imag)  // Constructor específico
    : real(p_real), imag(p_imag) { }
```

Finalmente, a continuación podemos ver un ejemplo de como sería una posible invocación a dicho constructor específico (para `c2`), junto a una invocación al constructor por defecto (para `c1`):

```
//- fichero: main.cpp -----------------------------------------------
#include <iostream>
#include "complejos.hpp"
using namespace std ;
using namespace umalcc ;

int main()
{
    Complejo c1 ;
    Complejo c2(2.5, 7.3) ;
    // ...
}
//- fin: main.cpp ---------------------------------------------------
```



Constructor por Defecto

Como se explicó anteriormente (véase 11.1.1 y 11.1.3), el *constructor por defecto* es el mecanismo por defecto utilizado para construir objetos de este tipo cuando no se especifica ninguna forma explícita de construcción. Así, será invocado automáticamente cuando se deba construir un determinado objeto, sin especificar explícitamente el tipo de construcción requerido, en el momento en que sea necesaria dicha construcción, por ejemplo cuando el flujo de ejecución alcanza la declaración de una variable de dicho tipo (véase 11.1.2).

El constructor por defecto es un método especial de la clase, ya que si el programador no define **ningún** constructor para una determinada clase, entonces el compilador generará e implementará **automáticamente** dicho constructor con el comportamiento por defecto de invocar automáticamente al constructor por defecto para cada atributo de *tipo compuesto* miembro de la clase. Nótese, sin embargo, que en el caso atributos de *tipo simple*, la implementación automática del compilador los dejará sin inicializar.

No obstante, el programador puede definir el constructor por defecto para una determinada clase cuando el comportamiento generado automáticamente por el compilador no sea el deseado. Para ello, la definición del constructor por defecto se corresponde con la definición de un constructor que no recibe ningún parámetro, y la implementación dependerá de las acciones necesarias para inicializar por defecto el estado interno del objeto que se está creando. Por ejemplo, para la clase `Complejo`:

```
class Complejo {
public:
    Complejo() ;              // Constructor por Defecto
} ;
```

A continuación se puede ver como sería la implementación del constructor por defecto:

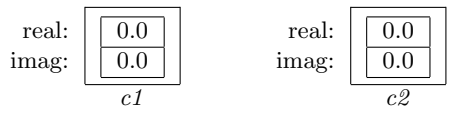
```
Complejo::Complejo()          // Constructor por Defecto
    : real(0.0), imag(0.0) { }
```

Otra posible implementación podría ser la siguiente, que invoca explícitamente al constructor por defecto para cada atributo miembro de la clase (que en este caso se incializará a cero):

```
Complejo::Complejo()          // Constructor por Defecto
    : real(), imag() { }
```

Finalmente, a continuación podemos ver un ejemplo de como sería una invocación a dicho constructor por defecto:

```
//- fichero: main.cpp -----------------------------------------------
#include <iostream>
#include "complejos.hpp"
using namespace std ;
using namespace umalcc ;

int main()
{
    Complejo c1, c2 ;
    // ...
}
//- fin: main.cpp ---------------------------------------------------
```



Constructor de Copia

El constructor de copia es el constructor que permite inicializar un determinado objeto como una copia de otro objeto de su misma clase. Así, se invoca automaticamente al inicializar el contenido de un objeto con el valor de otro objeto de su misma clase, y también es invocado automáticamente cuando un objeto de dicho tipo se pasa como *parámetro por valor* a subprogramas, aunque esto último, como se ha explicado previamente, está desaconsejado, ya que lo usual es pasar los tipos compuestos por referencia o por referencia constante.

El constructor de copia es un método especial de la clase, ya que si el programador no define dicho constructor de copia para una determinada clase, entonces el compilador generará e implementará **automáticamente** dicho constructor de copia con el comportamiento por defecto de invocar automáticamente al constructor de copia para cada atributo miembro de la clase, en este caso, tanto para atributos de *tipo simple* como de *tipo compuesto*.

No obstante, el programador puede definir el constructor de copia para una determinada clase cuando el comportamiento generado automáticamente por el compilador no sea el deseado. Para ello, la definición del constructor de copia se corresponde con la definición de un constructor que recibe como único parámetro *por referencia constante* un objeto del mismo tipo que la clase del constructor, y la implementación dependerá de las acciones necesarias para copiar el estado interno del objeto recibido como parámetro al objeto que se está creando. Por ejemplo, para la clase `Complejo`:

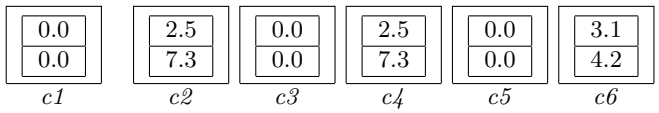
```
class Complejo {
public:
    Complejo(const Complejo& c) ;  // Constructor de Copia
} ;
```

y su implementación podría ser la siguiente, que en este caso coincide con la implementación que generaría automaticamente el compilador en caso de que no fuese implementado por el programador:

```
Complejo::Complejo(const Complejo& o)  // Constructor de Copia
    : real(o.real), imag(o.imag) { }   // Implementación automática
```

Finalmente, a continuación podemos ver un ejemplo de como sería una invocación al constructor de copia (para `c3` y `c4`), junto a una invocación a un constructor específico (para `c2`) y una invocación al constructor por defecto (para `c1`), así como la construcción por copia (para `c5` y `c6`) de objetos construidos invocando explícitamente a los constructores adecuados:

```
//- fichero: main.cpp -----------------------------------------------
#include <iostream>
#include "complejos.hpp"
using namespace std ;
using namespace umalcc ;

int main()
{
    Complejo c1 ;              // Construcción por defecto
    Complejo c2(2.5, 7.3) ;    // Construcción específica
    Complejo c3(c1) ;          // Construcción de copia (de c1)
    Complejo c4 = c2 ;         // Construcción de copia (de c2)
    Complejo c5 = Complejo(); // Construcción de copia de Complejo por Defecto
    Complejo c6 = Complejo(3.1, 4.2); // Construcción de copia de Complejo Específico
    // ...
}
//- fin: main.cpp ---------------------------------------------------
```



Destructor

El *destructor* de una clase será invocado automáticamente (sin parámetros actuales) para una determinada instancia (objeto) de esta clase cuando dicho objeto deba ser destruido, normalmente ésto sucederá cuando el flujo de ejecución del programa salga del ámbito de visibilidad de dicho objeto (véase 11.1.2).

El destructor es un método especial de la clase, ya que si el programador no define dicho destructor para una determinada clase, entonces el compilador generará e implementará **automáticamente** dicho destructor con el comportamiento por defecto de invocar automáticamente al destructor para cada atributo de *tipo compuesto* miembro de la clase.

No obstante, el programador puede definir el destructor para una determinada clase cuando el comportamiento generado automáticamente por el compilador no sea el deseado. Para ello, el destructor de la clase se define mediante el símbolo `~` seguido del identificador de la clase y una lista de parámetros vacía, y la implementación dependerá de las acciones necesarias para destruir y liberar los recursos asociados al estado interno del objeto que se está destruyendo. Posteriormente, el destructor invoca automáticamente a los destructores de los atributos miembros del objeto para que éstos sean destruidos. Por ejemplo, para la clase `Complejo`:

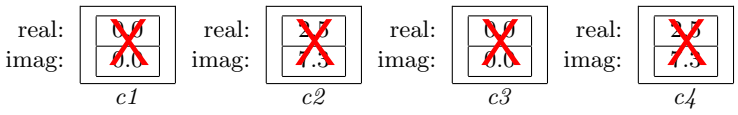
```
class Complejo {
public:
    ~Complejo() ;          // Destructor
} ;
```

y su implementación podría ser la siguiente, que en este caso coincide con la implementación que generaría automaticamente el compilador en caso de que no fuese implementado por el programador:

```
Complejo::~Complejo() { }   // Destructor:  Implementación automática
```

Finalmente, a continuación podemos ver un ejemplo de como se invoca automáticamente al destructor de los objetos cuando termina su tiempo de vida (para `c1`, `c2`, `c3` y `c4`):

```
//- fichero: main.cpp -----------------------------------------------
#include <iostream>
#include "complejos.hpp"
using namespace std ;
using namespace umalcc ;

int main()
{
    Complejo c1 ;              // Construcción por defecto
    Complejo c2(2.5, 7.3) ;    // Construcción específica
    Complejo c3(c1) ;          // Construcción de copia (de c1)
    Complejo c4 = c2 ;         // Construcción de copia (de c2)
    // ...
}                              // Destrucción automática de c4, c3, c2 y c1
//- fin: main.cpp --------------------------------------------------
```



Operador de Asignación

El operador de asignación define como se realiza la asignación (`=`) para objetos de esta clase. No se debe confundir el operador de asignación con el constructor de copia, ya que el constructor de copia construye un nuevo objeto que no tiene previamente ningún valor, mientras que en el caso del operador de asignación, el objeto ya tiene previamente un valor que deberá ser sustituido por el nuevo valor. Este valor previo deberá, en ocasiones, ser destruido antes de realizar la asignación del nuevo valor.

El operador de asignación (`=`) es un método especial de la clase, ya que si el programador no define dicho operador de asignación para una determinada clase, entonces el compilador generará e implementará **automáticamente** dicho operador de asignación con el comportamiento por defecto de invocar automáticamente al operador de asignación para cada atributo miembro de la clase, tanto para atributos de *tipo simple* como de *tipo compuesto*.

No obstante, el programador puede definir el operador de asignación para una determinada clase cuando el comportamiento generado automáticamente por el compilador no sea el deseado. Para ello, la definición del operador de asignación se corresponde con la definición de un operador `=` que recibe como único parámetro *por referencia constante* un objeto del mismo tipo que la clase del constructor, devuelve una referencia al propio objeto que recibe la asignación, y la implementación dependerá de las acciones necesarias para destruir el estado interno del objeto que recibe la asignación y para asignar el estado interno del objeto recibido como parámetro al objeto que se está creando. Por ejemplo, para la clase `Complejo`:

```
class Complejo {
public:
    Complejo& operator=(const Complejo& o) ;  // Operador de Asignación
} ;
```

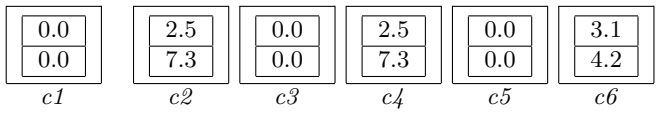
y su implementación podría ser la siguiente, que en este caso coincide con la implementación que generaría automaticamente el compilador en caso de que no fuese implementado por el programador:

```
Complejo& Complejo::operator=(const Complejo& o)  // Operador de Asignacion
{                                                 // Implementación automática
    real = o.real ;
    imag = o.imag ;
    return *this ;
}
```

El operador de asignación debe devolver el objeto actual (`return *this`) sobre el que recae la asignación.

Finalmente, a continuación podemos ver un ejemplo de como sería una invocación al operador de asignación (para `c3` y `c4`), junto a una invocación a un constructor específico (para `c2`) y una invocación al constructor por defecto (para `c1`), así como la asignación (para `c5` y `c6`) de objetos construidos invocando explícitamente a los constructores adecuados:

```
//- fichero: main.cpp -----------------------------------------------
#include <iostream>
#include "complejos.hpp"
using namespace std ;
using namespace umalcc ;

int main()
{
    Complejo c1, c3, c4, c5, c6; // Construcción por defecto de c1, c3, c4
    Complejo c2(2.5, 7.3) ;      // Construcción específica
    c3 = c1 ;                    // Asignación de c1 a c3
    c4 = c2 ;                    // Asignación de c2 a c4
    c5 = Complejo();             // Asignación de Complejo por Defecto
    c6 = Complejo(3.1, 4.2);     // Asignación de Complejo Específico
   // ...
}
//- fin: main.cpp ---------------------------------------------------
```



Hay situaciones en las que los objetos que se asignan tienen representaciones internas complejas, y en estos casos puede ser necesario destruir el estado interno del objeto que recibe la asignación antes de asignar el nuevo valor. En este caso, es conveniente comprobar que no se está produciendo una *auto-asignación* del mismo objeto (`x = x`), ya que en este caso se destruiría la representación interna del objeto antes de haberla asignado, con los errores que ello trae asociado. Por lo tanto, suele ser habitual que el operador de asignación implemente una condición para evitar la asignación en el caso de que se produzca una *auto-asignación*, de la siguiente forma:

```
Complejo& Complejo::operator=(const Complejo& o)  // Operador de Asignacion
{
    if (this != &o) {
        // destruir el valor anterior (en este caso no es necesario)
        real = o.real ;
        imag = o.imag ;
    }
    return *this ;
}
```

Así, `this` representa la dirección en memoria del objeto que recibe la asignación, y `&o` representa la dirección en memoria del objeto que se recibe como parámetro. Si ambas direcciones son diferentes, entonces significa que son variables diferentes y se puede realizar la asignación.

11.2.1. Ejemplo

Veamos un Tipo Abstracto de Datos `Lista` de enteros, la cual permite almacenar una secuencia de número enteros, permitiendo insertar, eliminar, acceder y modificar elementos según la posición que ocupen en la secuencia de números.

Definición

```
//- fichero: lista.hpp ----------------------------------------
#ifndef _lista_hpp_
#define _lista_hpp_
#include <tr1/array>
namespace umalcc {
    class ListaInt {
    public:
        //-----------------------------------------------------------
        //-- Métodos Públicos -------------------------------------
        //-----------------------------------------------------------
        // ~ListaInt() ;                    // Destructor Automático
        //------------------------------
```

```
        ListaInt() ;
        ListaInt(const ListaInt& o) ;
        ListaInt& operator = (const ListaInt& o) ;
        //-----------------------------
        bool llena() const ;
        int size() const ;
        void clear() ;
        //-----------------------------
        void insertar(int pos, int dato) ;
            // PRECOND: ( ! llena() && pos >= 0 && pos <= size())
        void eliminar(int pos) ;
            // PRECOND: (pos >= 0 && pos < size())
        //-----------------------------
        int acceder(int pos) const ;
            // PRECOND: (pos >= 0 && pos < size())
        void modificar(int pos, int dato);
            // PRECOND: (pos >= 0 && pos < size())
        //-------------------------------------------------------
    private:
        //-------------------------------------------------------
        //-- Ctes y Tipos Privados ------------------------------
        //-------------------------------------------------------
        static const int MAX = 100;
        typedef std::tr1::array<int, MAX> Datos;
        //-------------------------------------------------------
        //-- Metodos Privados -----------------------------------
        //-------------------------------------------------------
        void abrir_hueco(int pos) ;
        void cerrar_hueco(int pos) ;
        //-------------------------------------------------------
        //-- Atributos Privados ---------------------------------
        //-------------------------------------------------------
        int sz;         // numero de elementos de la lista
        Datos v;        // contiene los elementos de la lista
        //-------------------------------------------------------
    };
}
#endif
//- fin: lista.hpp -----------------------------------------
```

Implementación

```
//- fichero: lista.cpp -------------------------------------
#include "lista.hpp"
#include <cassert>
namespace umalcc {
    //-------------------------------------------------------
    //-- Métodos Públicos -----------------------------------
    //-------------------------------------------------------
    // ListaInt::~ListaInt() { }               // Destructor Automático
    //---------------------------------
    ListaInt::ListaInt() : sz(0), v() { }      // Constructor por Defecto
    //---------------------------------
    ListaInt::ListaInt(const ListaInt& o)      // Constructor de Copia
        : sz(o.sz), v()
    {
        for (int i = 0; i < sz; ++i) {
            v[i] = o.v[i] ;
        }
```

```
}
//----------------------------------
ListaInt& ListaInt::operator = (const ListaInt& o)  // Op. de Asignación
{
    if (this != &o) {
        sz = o.sz ;
        for (int i = 0; i < sz; ++i) {
            v[i] = o.v[i] ;
        }
    }
    return *this ;
}
//----------------------------------
bool ListaInt::llena() const
{
    return sz == int(v.size());
}
//----------------------------------
int ListaInt::size() const
{
    return sz ;
}
//----------------------------------
void ListaInt::clear()
{
    sz = 0 ;
}
//----------------------------------
void ListaInt::insertar(int pos, int dato)
{
    assert( ! llena() && pos >= 0 && pos <= size()) ;
    abrir_hueco(pos) ;
    v[pos] = dato ;
}
//----------------------------------
void ListaInt::eliminar(int pos)
{
    assert(pos >= 0 && pos < size()) ;
    cerrar_hueco(pos) ;
}
//----------------------------------
int ListaInt::acceder(int pos) const
{
    assert(pos >= 0 && pos < size()) ;
    return v[pos] ;
}
//----------------------------------
void ListaInt::modificar(int pos, int dato)
{
    assert(pos >= 0 && pos < size()) ;
    v[pos] = dato;
}
//-------------------------------------------------------
//-- Metodos Privados -----------------------------------
//-------------------------------------------------------
void ListaInt::abrir_hueco(int pos)
{
    assert(sz < int(v.size())) ;
    for (int i = sz; i > pos; --i) {
```

```
            v[i] = v[i-1];
        }
        ++sz;                                // Ahora hay un elemento más
    }
    //---------------------------------
    void ListaInt::cerrar_hueco(int pos)
    {
        assert(sz > 0) ;
        --sz;                                // Ahora hay un elemento menos
        for (int i = pos; i < sz; ++i) {
            v[i] = v[i+1];
        }
    }
    //---------------------------------
}
//- fin: lista.cpp -----------------------------------------------
```

Utilización

```
//- fichero: main.cpp --------------------------------------------
#include <iostream>
#include <cctype>
#include <cassert>
#include "lista.hpp"
using namespace std ;
using namespace umalcc ;
//---------------------------------------------------------------
void leer_pos(int& pos, int limite)
{
    assert(limite > 0);
    do {
        cout << "Introduzca posicion ( < " << limite << " ): " ;
        cin >> pos;
    } while (pos < 0 || pos >= limite);
}
//---------------------------------
void leer_dato(int& dato)
{
    cout << "Introduzca un dato: " ;
    cin >> dato;
}
//---------------------------------
void leer(ListaInt& lista)
{
    int dato ;
    lista.clear() ;
    cout << "Introduzca datos (0 -> FIN): " << endl ;
    cin >> dato ;
    while ((dato != 0)&&( ! lista.llena())) {
        lista.insertar(lista.size(), dato) ;
        cin >> dato ;
    }
}
//---------------------------------
void escribir(const ListaInt& lista)
{
    cout << "Lista: " ;
    for (int i = 0 ; i < lista.size() ; ++i) {
        cout << lista.acceder(i) << " " ;
```

```
    }
    cout << endl ;
}
//---------------------------------
void prueba_asg(const ListaInt& lista)
{
    cout << "Constructor de Copia" << endl ;
    ListaInt lst(lista) ;
    escribir(lst) ;
    cout << "Operador de Asignacion" << endl ;
    lst = lista ;
    escribir(lst) ;
}
//-----------------------------------------------------------------------
char menu()
{
    char op ;
    cout << endl ;
    cout << "X. Fin" << endl ;
    cout << "A. Leer Lista" << endl ;
    cout << "B. Borrar Lista" << endl ;
    cout << "C. Insertar Posicion" << endl ;
    cout << "D. Eliminar Posicion" << endl ;
    cout << "E. Acceder Posicion" << endl ;
    cout << "F. Modificar Posicion" << endl ;
    cout << "G. Prueba Copia y Asignacion" << endl ;
    do {
        cout << endl << "    Opcion: " ;
        cin >> op ;
        op = char(toupper(op)) ;
    } while (!((op == 'X')||((op >= 'A')&&(op <= 'G')))) ;
    cout << endl ;
    return op ;
}
//-----------------------------------------------------------------------
int main()
{
    ListaInt lista ;
    int dato ;
    int pos ;
    char op = ' ' ;
    do {
        op = menu() ;
        switch (op) {
        case 'A':
            leer(lista) ;
            escribir(lista) ;
            break ;
        case 'B':
            lista.clear() ;
            escribir(lista) ;
            break ;
        case 'C':
            if (lista.llena()) {
                cout << "Error: Lista llena" << endl ;
            } else {
                leer_pos(pos, lista.size()+1) ;
                leer_dato(dato) ;
                lista.insertar(pos, dato) ;
```

```
                escribir(lista) ;
            }
            break ;
        case 'D':
            if (lista.size() == 0) {
                cout << "Error: lista vacia" << endl ;
            } else {
                leer_pos(pos, lista.size()) ;
                lista.eliminar(pos) ;
                escribir(lista) ;
            }
            break ;
        case 'E':
            if (lista.size() == 0) {
                cout << "Error: lista vacia" << endl ;
            } else {
                leer_pos(pos, lista.size()) ;
                cout << "Lista[" << pos << "]: " << lista.acceder(pos) << endl ;
                escribir(lista) ;
            }
            break ;
        case 'F':
            if (lista.size() == 0) {
                cout << "Error: lista vacia" << endl ;
            } else {
                leer_pos(pos, lista.size()) ;
                leer_dato(dato) ;
                lista.modificar(pos, dato) ;
                escribir(lista) ;
            }
            break ;
        case 'G':
            prueba_asg(lista) ;
            break ;
        }
    } while (op != 'X') ;
}
//- fin: main.cpp -------------------------------------------------
```

Capítulo 12

Introducción a la Programación Genérica. Plantillas

El lenguaje de programación C++ proporciona soporte a la programación genérica mediante las plantillas (*"templates"* en inglés). Las plantillas proporcionan un mecanismo eficaz para definir código (constantes, tipos y subprogramas) genéricos parametrizados, que puedan ser instanciados en *"tiempo de compilación"*. Estos parámetros genéricos de las plantillas podrán ser instanciados con tipos y valores constantes concretos especificados en tiempo de compilación.

Las definiciones genéricas deberán, por lo general, estar visibles en el lugar donde sean instanciadas, por lo que en el caso de definirse en módulos diferentes, deberán estar definidas en los ficheros de encabezamiento, para que puedan ser incluidas por todos aquellos módulos que las necesiten.

La definición de plantillas, tanto de subprogramas como de tipos comienza con la palabra reservada `template`, seguida entre delimitadores `< ... >` por los parámetros genéricos de la definición. Estos parámetros genéricos pueden ser tanto tipos (precedidos por la palabra reservada `typename`), como constantes de tipos integrales (`char`, `short`, `int`, `unsigned`, `long`) o de tipos genéricos parametrizados con anterioridad (que deben ser instanciados a tipos integrales).

12.1. Subprogramas Genéricos

Los subprogramas genéricos son útiles cuando definen procesamientos genéricos que son independientes de los tipos concretos sobre los que se aplican. En este caso, simplemente se define el subprograma utilizando tanto los tipos como constantes genéricas mediante sus identificadores declarados en la directiva `template <...>`. Posteriormente, cuando se utilicen estos subprogramas genéricos, tanto los tipos como constantes genéricas serán instanciadas según los tipos y constantes actuales utilizados en la invocación a dichos subprogramas.

Veamos algunos ejemplos de definición de subprogramas genéricos:

```
template <typename Tipo>
inline Tipo maximo(const Tipo& x, const Tipo& y)
{
    Tipo max;
    if (x > y) {
        max = x ;
    } else {
        max = y ;
    }
    return max ;
}

template <typename Tipo>
```

```
inline void intercambio(Tipo& x, Tipo& y)
{
    Tipo aux = x ;
    x = y ;
    y = aux ;
}

int main()
{
    int x = 4 ;
    int y = maximo(x, 8) ;
    intercambio(x, y) ;

    double a = 7.5 ;
    double b = maximo(a, 12.0) ;
    intercambio(a, b) ;

    double c = maximo(a, 12) ; // Error: maximo(double, int) no esta definido
}
```

En el ejemplo se puede ver que los parámetros de entrada a los subprogramas se pasan por referencia (constante o variable), ya que al ser un tipo genérico podría ser tanto un tipo simple como un tipo estructurado.

También puede apreciarse que la instanciación de subprogramas genéricos a tipos concretos se realiza automáticamente a partir de la invocación a los mismos, de tal forma que la instanciación de los parámetros genéricos se realiza por deducción a partir del tipo de los parámetros especificados en la invocación a los subprogramas.

Sin embargo, hay situaciones donde los parámetros genéricos no pueden ser deducidos de la propia invocación al subprograma. En este caso, los parámetros genéricos involucrados deben ser especificados explícitamente en la llamada.

```
int main()
{
    double a = 7.5 ;
    double b = maximo(a, 12.0) ;

    double c = maximo<double>(a, 12) ;
}
```

El siguiente ejemplo de subprograma genérico muestra la utilización de parámetros genéricos constantes (de tipo integral).

```
#include <iostream>
#include <string>
#include <tr1/array>
using namespace std ;
using namespace std::tr1 ;
//-------------------------------------
template <typename TipoBase, unsigned SIZE>
void asignar(array<TipoBase, SIZE>& v, const TipoBase& x)
{
    for (int i = 0 ; i < int(v.size()) ; ++i) {
        v[i] = x ;
    }
}
//-------------------------------------
// Requiere que el TipoBase se pueda escribir con <<
template <typename TipoBase, unsigned SIZE>
void escribir(const array<TipoBase, SIZE>& v)
{
```

```
    for (int i = 0 ; i < int(v.size()) ; ++i) {
        cout << v[i] << " ";
    }
    cout << endl ;
}
//-------------------------------------
typedef array<int, 5> AInt ;
void prueba1()
{
    AInt a = {{ 1, 2, 3, 4, 5 }};
    escribir(a);
    asignar(a, 5) ;
    escribir(a);
}
//-------------------------------------
typedef array<string, 3> APers ;
void prueba2()
{
    APers a = {{ "pepe", "juan", "maría" }} ;
    escribir(a);
    asignar(a, "lola") ;
    escribir(a);
}
//-------------------------------------
```

Errores de Instanciación de Parámetros Genéricos

En el caso de que la definición de un determinado subprograma sea incorrecta para una determinada instanciación concreta de los parámetros genéricos, se producirá un error de compilación indicando el tipo de error. Por ejemplo, si compilamos el siguiente código, se produce un error de compilación, ya que el tipo `Persona` no tiene definido el operador de salida (`<<`).

```
#include <iostream>
#include <string>
#include <tr1/array>
using namespace std ;
using namespace std::tr1 ;
//-------------------------------------
// Requiere que el TipoBase se pueda escribir con <<
template <typename TipoBase, unsigned SIZE>
void escribir(const array<TipoBase, SIZE>& v)
{
    for (int i = 0 ; i < int(v.size()) ; ++i) {
        cout << v[i] << " ";                          // línea 12
    }
    cout << endl ;
}
//-------------------------------------
struct Persona {
    string nombre ;
    string telefono ;
} ;
typedef array<Persona, 4> APersona ;
void prueba3()
{
    APersona a = {{
        { "pepe",   "00" },
        { "juan",   "11" },
        { "maría",  "22" },
```

```
        { "carmen", "33" }
    }} ;
    escribir(a);                                    // línea 30
}
//-------------------------------------
```

Produce el siguiente mensaje de error de compilación (con *GNU GCC*):

```
main.cpp: In function 'void escribir(const std::tr1::array<_Tp, _Nm>&)
                          [with TipoBase = Persona, unsigned int SIZE = 4u]':
main.cpp:30:   instantiated from here
main.cpp:12: error: no match for 'operator<<' in '...'
             [......]
```

Cuando se trabaja con plantillas en C++, hay que tener presente que, a veces, los mensajes de error pueden ser bastante complicados de interpretar en el caso de errores producidos por instanciaciones de parámetros genéricos. En estos casos, el primer mensaje de error suele ser el más útil para guiarnos en su corrección.

12.2. Tipos Abstractos de Datos Genéricos

Las plantillas también pueden ser utilizadas para la definición de tipos genéricos, tanto registros como clases genéricas (TADs genéricos).

Como se explicó al comienzo del capítulo, las definiciones genéricas deben, por lo general, estar visibles en el lugar donde sean instanciadas, y por lo tanto, en el caso de tipos abstractos genéricos tanto la definición como la implementación se realizará completamente dentro de los ficheros de encabezamiento. Además, en esta sección introductoria a la programación genérica sólo veremos los tipos abstractos de datos genéricos definidos e implementados *"en línea"*, y en su versión más simple.

La definición de una clase genérica se realiza de forma similar a la definición de una clase no genérica, pero con algunas diferencias:

- Se precede la definición de la clase con la palabra reservada `template` seguida entre delimitadores `< ... >` por los parámetros genéricos de la definición. Estos parámetros genéricos pueden ser tanto tipos (precedidos por la palabra reservada `typename`), como constantes de tipos integrales (`char`, `short`, `int`, `unsigned`, `long`) o de tipos genéricos parametrizados con anterioridad (que deben ser instanciados a tipos integrales).

- Los tipos y constantes genéricas de la plantilla pueden ser utilizados en la definición de la clase mediante sus identificadores declarados en la directiva `template <...>`.

- La implementación de los métodos se debe realizar *"en-línea"*, es decir, el cuerpo del método también se definirá dentro de la definición de la clase, a diferencia de como se ha visto hasta ahora, en la cual los métodos se implementaban de forma independiente en un fichero de implementación aparte.

En la instanciación de tipos abstractos genéricos, será necesaria la instanciación explícita de los parámetros de los mismos, a diferencia de la instanciación de subprogramas genéricos vista anteriormente, en la cual los parámetros genéricos se instancian automáticamente a través de la invocación a los subprogramas.

Veamos algunos ejemplos de definición y utilización de tipos abstractos genéricos.

Ejemplo 1

Consideremos el tipo abstracto genérico denominado `Par`, que almacena dos elementos de tipos genéricos, y proporciona métodos tanto para conocer sus valores, como para su modificación.

```
//- par.hpp ----------------------------------------------------------------
#ifndef _par_hpp_
#define _par_hpp_
namespace umalcc {
    template <typename Tipo_1, typename Tipo_2>
    class Par {
    public:
        //------------------------------
        // Destructor Automático
        // ~Par() { }
        //------------------------------
        // Constructor Copia Automático
        // Par(const Par& o) : elem_1(o.elem_1), elem_2(o.elem_2) { }
        //------------------------------
        // Operador Asignación Automático
        // Par& operator = (const Par& o)
        // {
        //    if (this != &o) {
        //         elem_1 = o.elem_1;
        //         elem_2 = o.elem_2;
        //     }
        //     return *this;
        // }
        //------------------------------
        Par() : elem_1(), elem_2() { }
        //------------------------------
        Par(const Tipo_1& valor_1, const Tipo_2& valor_2)
            : elem_1(valor_1), elem_2(valor_2) { }
        //------------------------------
        Tipo_1 primero() const
        {
            // Devuelve el valor del primer elemento del par
            return elem_1;
        }
        //------------------------------
        Tipo_2 segundo() const
        {
            // Devuelve el valor del segundo elemento del par
            return elem_2;
        }
        //------------------------------
        void asg_primero(const Tipo_1& valor)
        {
            // Asigna un valor al primer elemento del par
            elem_1 = valor;
        }
        //------------------------------
        void asg_segundo(const Tipo_2& valor)
        {
            // Asigna un valor al segundo elemento del par
            elem_2 = valor;
        }
        //------------------------------
    private:
        Tipo_1 elem_1;
        Tipo_2 elem_2;
    };
}
#endif
```

A continuación podemos ver un ejemplo de su utilización:

```
//- main.hpp ----------------------------------------------------------
#include <iostream>
#include <string>
#include "par.hpp"
using namespace std;
using namespace umalcc;
//--------------------------------------
typedef Par<int, int>    ParInt;
typedef Par<string, int> ParPer;
//--------------------------------------
int main()
{
    ParInt x(3, 7) ;        // Crea un Par de enteros con los valores 3 y 7
    ParPer p("pepe", 23) ;  // Crea un Par de Persona con los valores "pepe" y 23
    ParPer q = p;           // Copia el par de p a q
    cout << x.primero() << " " << x.segundo() << endl;
    p.asg_primero("juan");
    cout << p.primero() << " " << p.segundo() << endl;
}
//--------------------------------------
```

Ejemplo 2

En el capítulo anterior (véase 11.2.1) se vio un ejemplo de un TAD Lista de número enteros.

Definición e Implementación

Podemos definir una TAD genérico `Lista` que permita almacenar elementos homogéneos de un tipo de datos genérico, y donde su capacidad máxima también esté parametrizada de forma genérica:

```
//- fichero: lista.hpp ------------------------------------------
#ifndef _lista_hpp_
#define _lista_hpp_
#include <tr1/array>
#include <cassert>
namespace umalcc {
    template <typename TipoBase, unsigned SIZE>
    class Lista {
    public:
        //------------------------------------------------------------
        //-- Métodos Públicos ----------------------------------------
        //------------------------------------------------------------
        // ~Lista() { }                    // Destructor Automático
        //-----------------------------
        Lista() : sz(0), v() { }           // Constructor por Defecto
        //-----------------------------
        Lista(const Lista& o)              // Constructor de Copia
            : sz(o.sz), v()
        {
            for (int i = 0; i < sz; ++i) {
                v[i] = o.v[i] ;
            }
        }
        //-----------------------------
        Lista& operator = (const Lista& o)  // Operador de Asignación
        {
            if (this != &o) {
```

```
            sz = o.sz ;
            for (int i = 0; i < sz; ++i) {
                v[i] = o.v[i] ;
            }
        }
        return *this ;
    }
    //------------------------------
    bool llena() const
    {
        return sz == int(v.size());
    }
    //------------------------------
    int size() const
    {
        return sz ;
    }
    //------------------------------
    void clear()
    {
        sz = 0 ;
    }
    //------------------------------
    // PRECOND: ( ! llena() && pos >= 0 && pos <= size())
    void insertar(int pos, const TipoBase& dato)
    {
        assert( ! llena() && pos >= 0 && pos <= size()) ;
        abrir_hueco(pos) ;
        v[pos] = dato ;
    }
    //------------------------------
    // PRECOND: (pos >= 0 && pos < size())
    void eliminar(int pos)
    {
        assert(pos >= 0 && pos < size()) ;
        cerrar_hueco(pos) ;
    }
    //------------------------------
    // PRECOND: (pos >= 0 && pos < size())
    TipoBase acceder(int pos) const
    {
        assert(pos >= 0 && pos < size()) ;
        return v[pos] ;
    }
    //------------------------------
    // PRECOND: (pos >= 0 && pos < size())
    void modificar(int pos, const TipoBase& dato)
    {
        assert(pos >= 0 && pos < size()) ;
        v[pos] = dato;
    }
    //-----------------------------------------------------------
private:
    //-----------------------------------------------------------
    //-- Ctes y Tipos Privados --------------------------------
    //-----------------------------------------------------------
    typedef std::tr1::array<TipoBase, SIZE> Datos;
    //-----------------------------------------------------------
    //-- Metodos Privados -------------------------------------
```

```
        //-------------------------------------------------------
        void abrir_hueco(int pos)
        {
            assert(sz < int(v.size())) ;
            for (int i = sz; i > pos; --i) {
                v[i] = v[i-1];
            }
            ++sz;                               // Ahora hay un elemento más
        }
        //------------------------------
        void cerrar_hueco(int pos)
        {
            assert(sz > 0) ;
            --sz;                               // Ahora hay un elemento menos
            for (int i = pos; i < sz; ++i) {
                v[i] = v[i+1];
            }
        }
        //-------------------------------------------------------
        //-- Atributos Privados ---------------------------------
        //-------------------------------------------------------
        int sz;         // numero de elementos de la lista
        Datos v;        // contiene los elementos de la lista
        //-------------------------------------------------------
    };
}
#endif
//- fin: lista.hpp -------------------------------------------
```

Utilización

Veamos a continuación un ejemplo de instanciación y utilización del TAD Lista genérica visto anteriormente:

```
//- fichero: main.cpp ----------------------------------------------
#include <iostream>
#include <cctype>
#include <cassert>
#include "lista.hpp"
using namespace std ;
using namespace umalcc ;
//-----------------------------------------------------------------
// Instanciación de la Lista genérica para almacenar hasta 100 números enteros
typedef Lista<int, 100> ListaInt;
//-----------------------------------------------------------------
void leer_pos(int& pos, int limite)
{
    assert(limite > 0);
    do {
        cout << "Introduzca posicion ( < " << limite << " ): " ;
        cin >> pos;
    } while (pos < 0 || pos >= limite);
}
//--------------------------------
void leer_dato(int& dato)
{
    cout << "Introduzca un dato: " ;
    cin >> dato;
}
//--------------------------------
```

```
void leer(ListaInt& lista)
{
    int dato ;
    lista.clear() ;
    cout << "Introduzca datos (0 -> FIN): " << endl ;
    cin >> dato ;
    while ((dato != 0)&&( ! lista.llena())) {
        lista.insertar(lista.size(), dato) ;
        cin >> dato ;
    }
}
//---------------------------------
void escribir(const ListaInt& lista)
{
    cout << "Lista: " ;
    for (int i = 0 ; i < lista.size() ; ++i) {
        cout << lista.acceder(i) << " " ;
    }
    cout << endl ;
}
//---------------------------------
void prueba_asg(const ListaInt& lista)
{
    cout << "Constructor de Copia" << endl ;
    ListaInt lst(lista) ;
    escribir(lst) ;
    cout << "Operador de Asignacion" << endl ;
    lst = lista ;
    escribir(lst) ;
}
//-------------------------------------------------------------------------
char menu()
{
    char op ;
    cout << endl ;
    cout << "X. Fin" << endl ;
    cout << "A. Leer Lista" << endl ;
    cout << "B. Borrar Lista" << endl ;
    cout << "C. Insertar Posicion" << endl ;
    cout << "D. Eliminar Posicion" << endl ;
    cout << "E. Acceder Posicion" << endl ;
    cout << "F. Modificar Posicion" << endl ;
    cout << "G. Prueba Copia y Asignacion" << endl ;
    do {
        cout << endl << "    Opcion: " ;
        cin >> op ;
        op = char(toupper(op)) ;
    } while (!((op == 'X')||((op >= 'A')&&(op <= 'G')))) ;
    cout << endl ;
    return op ;
}
//-------------------------------------------------------------------------
int main()
{
    ListaInt lista ;
    int dato ;
    int pos ;
    bool ok ;
    char op = ' ' ;
```

```
    do {
        op = menu() ;
        switch (op) {
        case 'A':
            leer(lista) ;
            escribir(lista) ;
            break ;
        case 'B':
            lista.clear() ;
            escribir(lista) ;
            break ;
        case 'C':
            if (lista.llena()) {
                cout << "Error: Lista llena" << endl ;
            } else {
                leer_pos(pos, lista.size()+1) ;
                leer_dato(dato) ;
                lista.insertar(pos, dato) ;
                escribir(lista) ;
            }
            break ;
        case 'D':
            if (lista.size() == 0) {
                cout << "Error: lista vacia" << endl ;
            } else {
                leer_pos(pos, lista.size()) ;
                lista.eliminar(pos) ;
                escribir(lista) ;
            }
            break ;
        case 'E':
            if (lista.size() == 0) {
                cout << "Error: lista vacia" << endl ;
            } else {
                leer_pos(pos, lista.size()) ;
                cout << "Lista[" << pos << "]: " << lista.acceder(pos) << endl ;
                escribir(lista) ;
            }
            break ;
        case 'F':
            if (lista.size() == 0) {
                cout << "Error: lista vacia" << endl ;
            } else {
                leer_pos(pos, lista.size()) ;
                leer_dato(dato) ;
                lista.modificar(pos, dato) ;
                escribir(lista) ;
            }
            break ;
        case 'G':
            prueba_asg(lista) ;
            break ;
        }
    } while (op != 'X') ;
}
//- fin: main.cpp -------------------------------------------------
```

Capítulo 13

Memoria Dinámica. Punteros

Hasta ahora, todos los programas que se han visto en capítulos anteriores almacenan su estado interno por medio de variables que son automáticamente gestionadas por el compilador. Las variables son *creadas* cuando el flujo de ejecución entra en el ámbito de su definición (se reserva espacio en memoria y se crea el valor de su estado inicial), posteriormente se *manipula* el estado de la variable (accediendo o modificando su valor almacenado), y finalmente se *destruye* la variable cuando el flujo de ejecución sale del ámbito donde fue declarada la variable (liberando los recursos asociados a ella y la zona de memoria utilizada). A este tipo de variables gestionadas automáticamente por el compilador se las suele denominar *variables automáticas* (también variables locales), y residen en una zona de memoria gestionada automáticamente por el compilador, la *pila* de ejecución, donde se alojan y desalojan las variables locales (automáticas) pertenecientes al ámbito de ejecución de cada subprograma.

Así, el tiempo de vida de una determinada variable está condicionado por el ámbito de su declaración. Además, el número de variables automáticas utilizadas en un determinado programa está especificado explícitamente en el propio programa, y por lo tanto su capacidad de almacenamiento está también especificada y predeterminada por lo especificado explícitamente en el programa. Es decir, con la utilización única de variables automáticas, la capacidad de almacenamiento de un determinado programa está predeterminada desde el momento de su programación (tiempo de compilación), y no puede adaptarse a las necesidades reales de almacenamiento surgidas durante la ejecución del programa (tiempo de ejecución).[1]

La gestión de *memoria dinámica* surge como un mecanismo para que el propio programa, durante su ejecución (tiempo de ejecución), pueda solicitar (alojar) y liberar (desalojar) memoria según las necesidades surgidas durante una determinada ejecución, dependiendo de las circunstancias reales de cada momento de la ejecución del programa en un determinado entorno. Esta ventaja adicional viene acompañada por un determinado coste asociado a la mayor complejidad que requiere su gestión, ya que en el caso de las variables automáticas, es el propio compilador el encargado de su gestión, sin embargo en el caso de las *variables dinámicas* es el propio programador el que debe, mediante código software, gestionar el tiempo de vida de cada variable dinámica, cuando debe ser alojada y creada, como será utilizada, y finalmente cuando debe ser destruida y desalojada. Adicionalmente, como parte de esta gestión de la memoria dinámica por el propio programador, la memoria dinámica pasa a ser un *recurso* que debe gestionar el programador, y se debe preocupar de su alojo y de su liberación, poniendo especial cuidado y énfasis en no perder recursos (perder zonas de memoria sin liberar y sin capacidad de acceso).

[1] En realidad esto no es completamente cierto, ya que en el caso de subprogramas recursivos, cada invocación recursiva en tiempo de ejecución tiene la capacidad de alojar nuevas variables que serán posteriormente desalojadas automáticamente cuando la llamada recursiva finaliza.

13.1. Punteros

El *tipo puntero* es un tipo *simple* que permite a un determinado programa acceder a posiciones concretas de memoria, y más específicamente a determinadas zonas de la memoria dinámica. Aunque el lenguaje de programación C++ permite otras utilizaciones más diversas del tipo puntero, en este capítulo sólo se utilizará el tipo puntero para acceder a zonas de memoria dinámica.

Así, una determinada variable de tipo puntero apunta (o referencia) a una determinada entidad (variable) de un determinado tipo alojada en la zona de memoria dinámica. Por lo tanto, para un determinado tipo puntero, se debe especificar también el tipo de la variable (en memoria dinámica) a la que apunta, el cual define el espacio que ocupa en memoria y las operaciones (y métodos) que se le pueden aplicar, entre otras cosas.

De este modo, cuando un programa gestiona la memoria dinámica a través de punteros, debe manejar y gestionar por una parte la propia variable de tipo puntero, y por otra parte la variable dinámica apuntada por éste.

Un tipo puntero se define utilizando la palabra reservada `typedef` seguida del tipo de la variable dinámica apuntada, un asterisco para indicar que es un **puntero** a una variable de dicho tipo, y el identificador que denomina al tipo. Por ejemplo:

```
typedef int* PInt ;          // Tipo Puntero a Entero

struct Persona {             // Tipo Persona
    string nombre ;
    string telefono ;
    int edad ;
} ;
typedef Persona* PPersona ; // Tipo Puntero a Persona
```

Así, el tipo `PInt` es el tipo de una variable que apunta a una variable dinámica de tipo `int`. Del mismo modo, el tipo `PPersona` es el tipo de una variable que apunta a una variable dinámica de tipo `Persona`.

Es importante remarcar que el *tipo puntero*, en sí mismo, es un **tipo simple**, aunque el tipo apuntado puede ser tanto un tipo simple, como un tipo compuesto.

Es posible definir variables de los tipos especificados anteriormente. Nótese que estas variables (`p1` y `p2` en el siguiente ejemplo) son variables automáticas (gestionadas automáticamente por el compilador), es decir, se crean automáticamente (con un valor indeterminado) al entrar el flujo de ejecución en el ámbito de visibilidad de la variable, y posteriormente se destruyen automáticamente cuando el flujo de ejecución sale del ámbito de visibilidad de la variable. Por otra parte, las variables apuntadas por ellos son variables dinámicas (gestionadas por el programador), es decir el programador se encargará de solicitar la memoria dinámica cuando sea necesaria y de liberarla cuando ya no sea necesaria, durante la ejecución del programa. En el siguiente ejemplo, si las variables se definen sin inicializar, entonces tendrán un valor inicial inespecificado:

```
int main()
{
    PInt p1 ;
    PPersona p2 ;
}
```

p1: ? *p2:* ?

La constante `NULL` es una constante especial de tipo puntero que indica que una determinada variable de tipo puntero no apunta a nada, es decir, especifica que la variable de tipo puntero que contenga el valor `NULL` no apunta a ninguna zona de la memoria dinámica. Para utilizar la constante `NULL` se debe incluir la biblioteca estándar `<cstddef>`. Así, se pueden definir las variables `p1` y `p2` e inicializarlas a un valor indicando que no apuntan a nada.

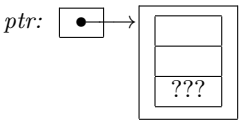
```
#include <cstddef>
int main()
{
    PInt p1 = NULL ;
    PPersona p2 = NULL ;
}
```

p1: ⁄ *p2:* ⁄

13.2. Gestión de Memoria Dinámica

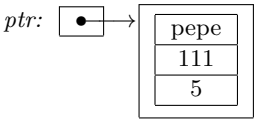
La memoria dinámica la debe gestionar el propio programador, por lo que cuando necesite crear una determinada variable dinámica, debe *solicitar memoria dinámica* con el operador `new` seguido por el tipo de la variable dinámica a crear. Este operador (`new`) realiza dos acciones principales, primero **aloja** (reserva) espacio en memoria dinámica para albergar a la variable, y después **crea** (invocando al constructor especificado) el contenido de la variable dinámica. Finalmente, a la variable `ptr` se le asigna el valor del puntero (una dirección de memoria) que apunta a la variable dinámica creada por el operador `new`. Por ejemplo, para crear una variable dinámica del tipo `Persona` definido anteriormente utilizando el constructor por defecto de dicho tipo.

```
int main()
{
    PPersona ptr = new Persona ;
}
```



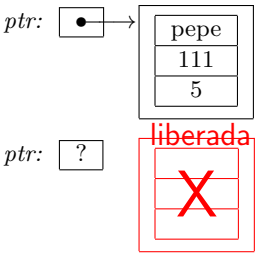
En caso de que el tipo de la variable dinámica tenga otros constructores definidos, es posible utilizarlos en la construcción del objeto en memoria dinámica. Por ejemplo, suponiendo que el tipo `Persona` tuviese un constructor que reciba el nombre, teléfono y edad de la persona:

```
int main()
{
    PPersona ptr = new Persona("pepe", "111", 5) ;
}
```



Posteriormente, tras manipular adecuadamente, según las características del programa, la memoria dinámica alojada, llegará un momento en que dicha variable dinámica ya no sea necesaria, y su tiempo de vida llegue a su fin. En este caso, el programador debe *liberar* explícitamente dicha variable dinámica mediante el operador `delete` de la siguiente forma:

```
int main()
{
    PPersona ptr = new Persona("pepe", "111", 5) ;

    // manipulación ...

    delete ptr ;
}
```

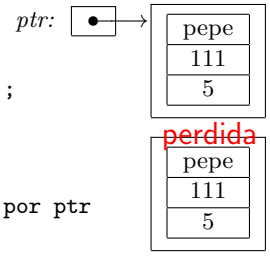


La sentencia `delete ptr` realiza dos acciones principales, primero **destruye** la variable dinámica (invocando a su destructor), y después **desaloja** (libera) la memoria dinámica reservada para dicha variable. Finalmente la variable local `ptr` queda con un valor inespecificado, y será destruida automáticamente por el compilador cuando el flujo de ejecución salga de su ámbito de declaración.

Si se ejecuta la operación `delete` sobre una variable de tipo puntero que tiene el valor `NULL`, entonces esta operación no hace nada.

En caso de que no se libere (mediante el operador `delete`) la memoria dinámica apuntada por la variable `ptr`, y esta variable sea destruida al terminar su tiempo de vida (su ámbito de visibilidad), entonces se *perderá* la memoria dinámica a la que apunta, con la consiguiente **pérdida de recursos** que ello conlleva.

```
int main()
{
    PPersona ptr = new Persona("pepe", "111", 5) ;

    // manipulación ...

    // no se libera la memoria dinámica apuntada por ptr
    // se destruye la variable local ptr
}
```



13.3. Operaciones con Variables de Tipo Puntero

Desreferenciación de una Variable de Tipo Puntero

Para acceder a una variable dinámica apuntada por una variable de tipo puntero, se utiliza el operador unario *asterisco* (`*`) precediendo al nombre de la variable de tipo puntero a través de la cual es apuntada. Por ejemplo, si `ptr` es una variable local de tipo puntero que apunta a una variable dinámica de tipo `Persona`, entonces `*ptr` es la variable dinámica apuntada, y se trata de igual forma que cualquier otra variable de tipo `Persona`.

```
int main()
{
    PPersona ptr = new Persona("pepe", "111", 5) ;

    Persona p = *ptr ;  // Asigna el contenido de la variable dinámica a la variable p

    *ptr = p ;          // Asigna el contenido de la variable p a la variable dinámica

    delete ptr ;        // destruye la variable dinámica y libera su espacio de memoria
}
```

Sin embargo, si una variable de tipo puntero tiene el valor `NULL`, entonces *desreferenciar* la variable produce un **error** en tiempo de ejecución que aborta la ejecución del programa.

Es posible, así mismo, acceder a los elementos de la variable apuntada mediante el operador de desreferenciación. Por ejemplo:

```
int main()
{
    PPersona ptr = new Persona ;

    (*ptr).nombre = "pepe" ;
    (*ptr).telefono = "111" ;
    (*ptr).edad = 5 ;

    delete ptr ;
}
```

Nótese que el uso de los paréntesis es obligatorio debido a que el operador punto (`.`) tiene mayor precedencia que el operador de desreferenciación (`*`). Por ello, en el caso de acceder a los campos de un registro en memoria dinámica a través de una variable de tipo puntero, es más adecuado utilizar el operador de desreferenciación (`->`). Por ejemplo:

```
int main()
{
    PPersona ptr = new Persona ;

    ptr->nombre = "pepe" ;
    ptr->telefono = "111" ;
    ptr->edad = 5 ;

    delete ptr ;
}
```

Este operador también se utiliza para invocar a métodos de un objeto si éste se encuentra alojado en memoria dinámica. Por ejemplo:

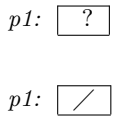
```
#include <iostream>
using namespace std ;
class Numero {
public:
```

```
    Numero(int v) : val(v) {}
    int valor() const { return val ; }
private:
    int val ;
} ;
typedef Numero* PNumero ;
int main()
{
    PNumero ptr = new Numero(5) ;

    cout << ptr->valor() << endl ;

    delete ptr ;
}
```

Asignación de Variables de Tipo Puntero

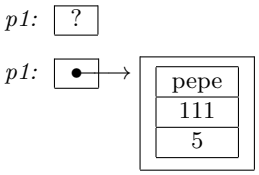
El puntero nulo (`NULL`) se puede asignar a cualquier variable de tipo puntero. Por ejemplo:

```
int main()
{
    PPersona p1 ;
    // ...
    p1 = NULL ;
    // ...
}
```



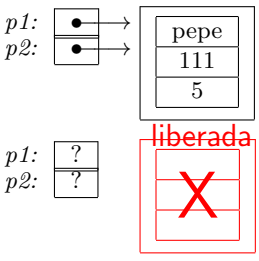
El resultado de crear una variable dinámica con el operador `new` se puede asignar a una variable de tipo puntero al tipo de la variable dinámica creada. Por ejemplo:

```
int main()
{
    PPersona p1 ;
    // ...
    p1 = new Persona("pepe", "111", 5) ;
    // ...
}
```



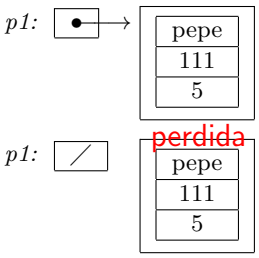
Así mismo, a una variable de tipo puntero se le puede asignar el valor de otra variable puntero. En este caso, ambas variables de tipo puntero apuntarán a la misma variable dinámica, que será compartida por ambas. Si se libera la variable dinámica apuntada por una de ellas, la variable dinámica compartida se destruye, su memoria se desaloja y ambas variables locales de tipo puntero quedan con un valor inespecificado.

```
int main()
{
    PPersona p1 = new Persona("pepe", "111", 5) ;
    PPersona p2 ;
    // ...
    p2 = p1 ;
    // ...
    delete p1 ;
}
```



En la operación de asignación, el valor anterior que tuviese la variable de tipo puntero se pierde, por lo que habrá que tener especial cuidado de que no se pierda la variable dinámica que tuviese asignada, si tuviese alguna.

```
int main()
{
    PPersona p1 = new Persona("pepe", "111", 5) ;
    // ...
    p1 = NULL ; // se pierde el valor anterior
    // ...
    delete p1 ;
}
```



Comparación de Variables de Tipo Puntero

Las variables del mismo tipo puntero se pueden comparar entre ellas por igualdad (==) o desigualdad (!=), para comprobar si apuntan a la misma variable dinámica. Así mismo, también se pueden comparar por igualdad o desigualdad con el puntero nulo (NULL) para saber si apunta a alguna variable dinámica, o por el contrario no apunta a nada. Por ejemplo:

```
int main()
{
    PPersona p1, p2 ;
    // ...
    if (p1 == p2) {
        // ...
    }
    if (p1 != NULL) {
        // ...
    }
}
```

13.4. Paso de Parámetros de Variables de Tipo Puntero

El tipo puntero es un *tipo simple*, y por lo tanto se tratará como tal. En caso de paso de parámetros de tipo puntero, si es un parámetro de entrada, entonces se utilizará el *paso por valor*, y si es un parámetro de salida o de entrada/salida, entonces se utilizará el *paso por referencia*.

Hay que ser consciente de que un parámetro de tipo puntero puede apuntar a una variable dinámica, y en este caso, a partir del parámetro se puede acceder a la variable apuntada.

Así, si el parámetro se pasa *por valor*, entonces se copia el valor del puntero del parámetro actual (en la invocación) al parámetro formal (en el subprograma), por lo que ambos apuntarán a la misma variable dinámica compartida, y en este caso, si se modifica el valor almacenado en la variable dinámica, este valor se verá afectado, así mismo, en el exterior del subprograma, aunque el parámetro haya sido pasado por valor.

Por otra parte, las funciones también pueden devolver valores de tipo puntero.

```
void modificar(PPersona& p) ;

PPersona buscar(PPersona l, const string& nombre) ;
```

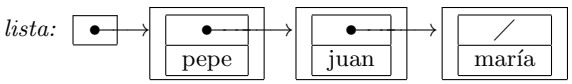
13.5. Listas Enlazadas Lineales

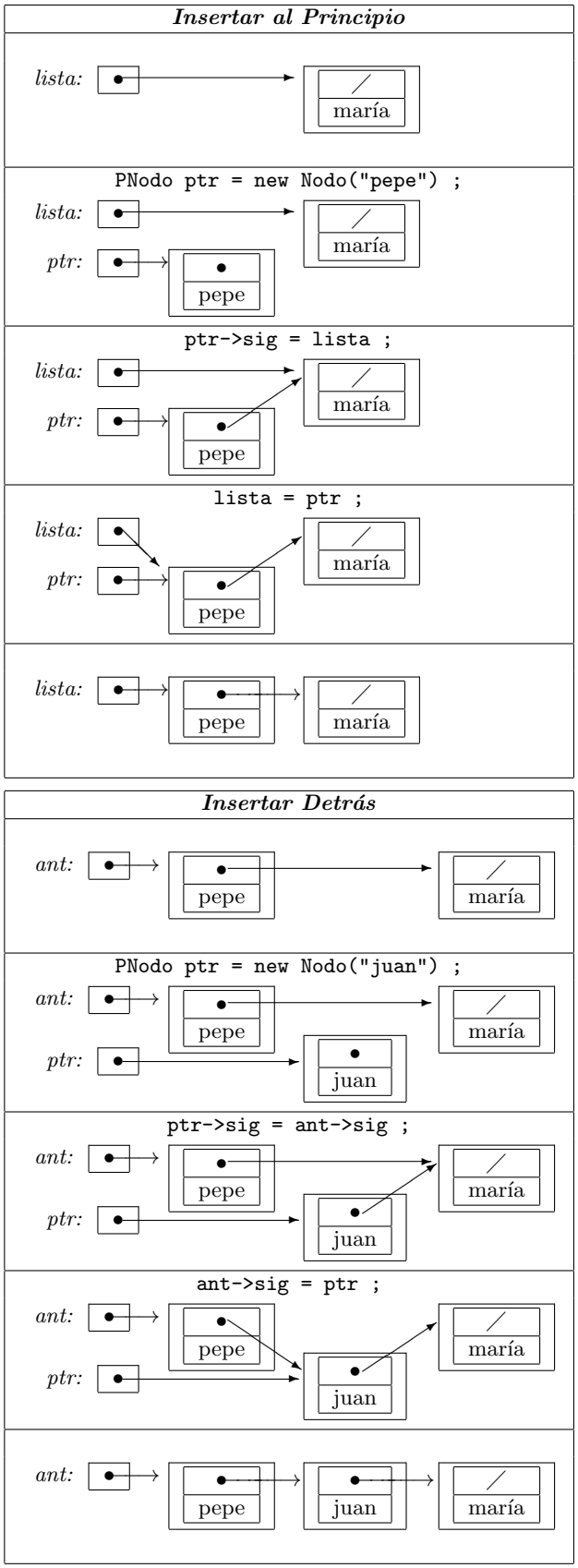
Una de las principales aplicaciones de la Memoria Dinámica es el uso de estructuras enlazadas, de tal forma que un campo o atributo de la variable dinámica es a su vez también de tipo puntero, por lo que puede apuntar a otra variable dinámica que también tenga un campo o atributo de tipo puntero, el cual puede volver a apuntar a otra variable dinámica, y así sucesivamente, tantas veces como sea necesario, hasta que un puntero con el valor NULL indique el final de la estructura enlazada (lista enlazada).

Así, en este caso, vemos que un campo de la estructura es de tipo puntero a la propia estructura, por lo que es necesario definir el tipo puntero antes de definir la estructura. Sin embargo, la

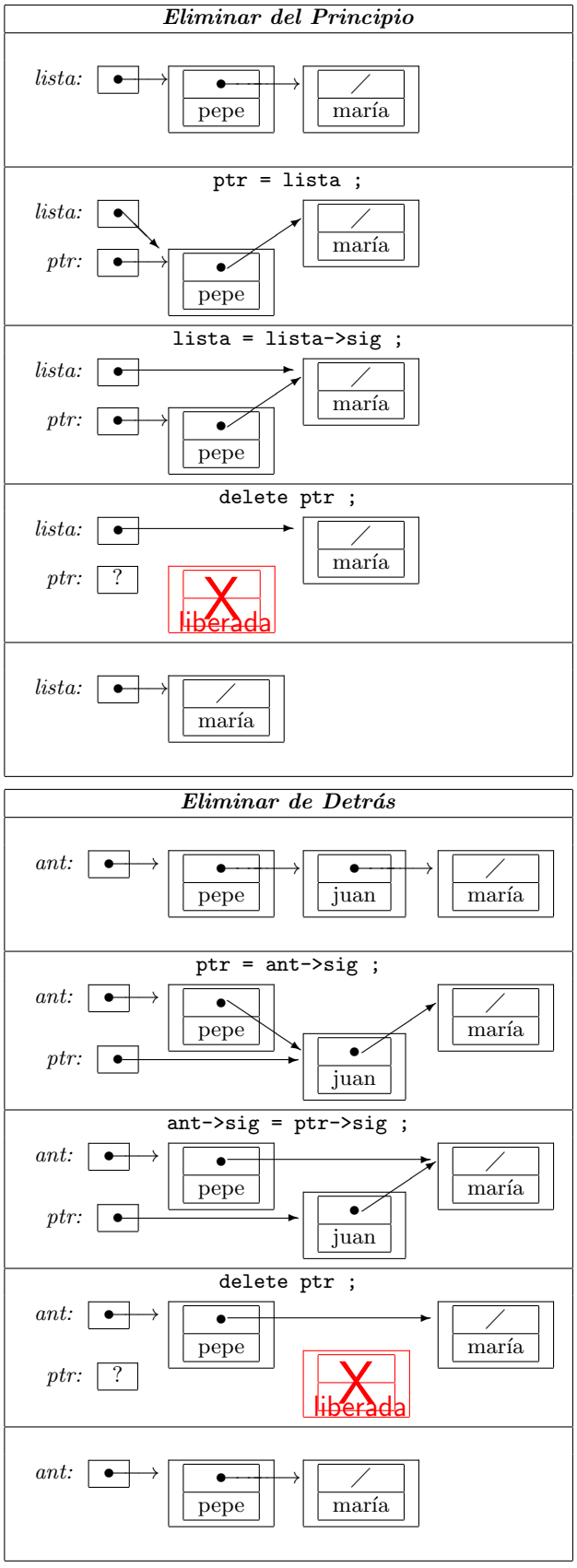
estructura todavía no ha sido definida, por lo que no se puede definir un puntero a ella. Por ello es necesario realizar una *declaración adelantada* de un *tipo incompleto* del tipo de la variable dinámica, donde se declara que un determinado identificador es una estructura o clase, pero no se definen sus componentes.

```
struct Nodo ;         // Declaración adelantada del tipo incompleto Nodo
typedef Nodo* PNodo ; // Definición de tipo Puntero a tipo incompleto Nodo
struct Nodo {         // Definición del tipo Nodo
    PNodo sig ;       // Enlace a la siguiente estructura dinámica
    string dato ;     // Dato almacenado en la lista
} ;
void escribir(PNodo lista)
{
    PNodo ptr = lista;
    while (ptr != NULL) {
        cout << ptr->dato << endl ;
        ptr = ptr->sig ;
    }
}
PNodo buscar(PNodo lista, const string& dt)
{
    PNodo ptr = lista ;
    while ((ptr != NULL)&&(ptr->dato != dt)) {
        ptr = ptr->sig ;
    }
    return ptr ;
}
PNodo leer_inversa()
{
    PNodo lista = NULL ;
    string dt ;
    cin >> dt ;
    while (dt != "fin") {
        PNodo ptr = new Nodo ;
        ptr->dato = dt ;
        ptr->sig = lista ;
        lista = ptr ;
        cin >> dt ;
    }
    return lista ;
}
void destruir(PNodo& lista)
{
    while (lista != NULL) {
        PNodo ptr = lista ;
        lista = lista->sig ;
        delete ptr ;
    }
}
int main()
{
    PNodo lista ;
    lista = leer_inversa() ;
    escribir(lista) ;
    PNodo ptr = buscar(lista, "juan");
    if (ptr != NULL) {
        cout << ptr->dato << endl;
    }
    destruir(lista) ;
}
```






```
struct Nodo ;
typedef Nodo* PNodo ;
struct Nodo {
    PNodo sig ;
    string dato ;
} ;

void insertar_principio(PNodo& lista, const string& dt)
{
    PNodo ptr = new Nodo ;
    ptr->dato = dt ;
    ptr->sig = lista ;
    lista = ptr ;
}

void insertar_final(PNodo& lista, const string& dt)
{
    PNodo ptr = new Nodo ;
    ptr->dato = dt ;
    ptr->sig = NULL ;
    if (lista == NULL) {
        lista = ptr ;
    } else {
        PNodo act = lista ;
        while (act->sig != NULL) {
            act = act->sig ;
        }
        act->sig = ptr ;
    }
}

PNodo situar(PNodo lista, int pos)
{
    PNodo ptr = lista;
    while ((ptr != NULL)&&(pos > 0)) {
        ptr = ptr->sig;
        --pos;
    }
    return ptr;
}

void insertar_pos(PNodo& lista, int pos, const string& dt)
{
    if (pos < 1) {
        PNodo ptr = new Nodo ;
        ptr->dato = dt ;
        ptr->sig = lista ;
        lista = ptr ;
    } else {
        PNodo ant = situar(lista, pos - 1);
        if (ant != NULL) {
            PNodo ptr = new Nodo ;
            ptr->dato = dt ;
            ptr->sig = ant->sig ;
            ant->sig = ptr ;
        }
    }
}
```




```
void eliminar_primero(PNodo& lista)
{
    if (lista != NULL) {
        PNodo ptr = lista ;
        lista = lista->sig ;
        delete ptr ;
    }
}

void eliminar_ultimo(PNodo& lista)
{
    if (lista != NULL) {
        if (lista->sig == NULL) {
            delete lista ;
            lista = NULL ;
        } else {
            PNodo ant = lista ;
            PNodo act = ant->sig ;
            while (act->sig != NULL) {
                ant = act ;
                act = act->sig ;
            }
            delete act ;
            ant->sig = NULL ;
        }
    }
}

void eliminar_pos(PNodo& lista, int pos)
{
    if (lista != NULL) {
        if (pos < 1) {
            PNodo ptr = lista ;
            lista = lista->sig ;
            delete ptr ;
        } else {
            PNodo ant = situar(lista, pos - 1) ;
            if ((ant != NULL)&&(ant->sig != NULL)) {
                PNodo ptr = ant->sig ;
                ant->sig = ptr->sig ;
                delete ptr ;
            }
        }
    }
}
```

```
void insertar_ord(PNodo& lista,
                  const string& dt)
{
    PNodo ptr = new Nodo ;
    ptr->dato = dt ;
    if ((lista==NULL)||(dt < lista->dato)) {
        ptr->sig = lista ;
        lista = ptr ;
    } else {
        PNodo ant = lista ;
        PNodo act = ant->sig ;
        while ((act!=NULL)&&(act->dato<=dt)){
            ant = act ;
            act = act->sig ;
        }
        ptr->sig = ant->sig ;
        ant->sig = ptr ;
    }
}
```

```
void eliminar_elem(PNodo& lista, const string& dt)
{
    if (lista != NULL) {
        if (lista->dato == dt) {
            PNodo ptr = lista ;
            lista = lista->sig ;
            delete ptr ;
        } else {
            PNodo ant = lista ;
            PNodo act = ant->sig ;
            while ((act != NULL)&&(act->dato != dt)) {
                ant = act ;
                act = act->sig ;
            }
            if (act != NULL) {
                ant->sig = act->sig ;
                delete act ;
            }
        }
    }
}
```

```
PNodo duplicar(PNodo lista)
{
    PNodo nueva = NULL;
    if (lista != NULL) {
        nueva = new Nodo ;
        nueva->dato = lista->dato ;
        PNodo u = nueva ;
        PNodo p = lista->sig ;
        while (p != NULL) {
            u->sig = new Nodo ;
            u->sig->dato = p->dato ;
            u = u->sig ;
            p = p->sig ;
        }
        u->sig = NULL ;
    }
    return nueva;
}
```

```
void purgar(PNodo& lista, const string& dt)
{
    while ((lista != NULL)&&(dt == lista->dato)) {
        PNodo ptr = lista ;
        lista = lista->sig ;
        delete ptr ;
    }
    if (lista != NULL) {
        PNodo ant = lista;
        PNodo act = lista->sig;
        while (act != NULL) {
            if (dt == act->dato) {
                ant->sig = act->sig ;
                delete act ;
            } else {
                ant = act;
            }
            act = ant->sig;
        }
    }
}
```

```
void insertar_ultimo(PNodo& lista,
                     PNodo& ult,
                     const string& dt)
{
    PNodo p = new Nodo ;
    p->dato = dt ;
    p->sig = NULL ;
    if (lista == NULL) {
        lista = p ;
    } else {
        ult->sig = p ;
    }
    ult = p ;
}
```

```
PNodo leer()
{
    PNodo lista = NULL ;
    PNodo ult = NULL ;
    string dt ;
    cin >> dt ;
    while (dt != "fin") {
        insertar_ultimo(lista, ult, dt) ;
        cin >> dt ;
    }
    return lista ;
}
```

13.6. Abstracción en la Gestión de Memoria Dinámica

La gestión de memoria dinámica por parte del programador se basa en estructuras de programación de *bajo nivel*, las cuales son propensas a errores de programación y pérdida de recursos de memoria. Además, entremezclar sentencias de gestión de memoria, de bajo nivel, con sentencias aplicadas al dominio de problema a resolver suele dar lugar a código no legible y propenso a errores.

Por lo tanto se hace necesario aplicar niveles de abstracción que aislen la gestión de memoria dinámica (de bajo nivel) del resto del código más directamente relacionado con la solución del problema. Para ello, los tipos abstractos de datos proporcionan el mecanismo adecuado para aplicar la abstracción a estas estructuras de datos basadas en la gestión de memoria dinámica, además de proporcionar una herramienta adecuada para la gestión de memoria dinámica, ya que los destructores se pueden encargar de liberar los recursos asociados a un determinado objeto.

Así, vemos que será necesario duplicar la información almacenada en la memoria dinámica al copiar y asignar objetos, así como también será necesario liberar la memoria dinámica antes de asignar nueva información o de destruir el objeto.

- El *Destructor* debe liberar la memoria dinámica antes de destruir el objeto.
- El *Ctor-Defecto* debe inicializar adecuadamente los atributos de tipo puntero.
- El *Ctor-Copia* debe duplicar la memoria dinámica del objeto a copiar.
- El *Op-Asignación* debe liberar la memoria dinámica actual y duplicar la memoria dinámica del objeto a asignar.
- Los *métodos* de la clase permiten manipular la estructura de datos, proporcionando abstracción sobre su complejidad y representación interna.
- El acceso restringido a la representación interna impide una manipulación externa propensa a errores.

```
template <typename Tipo>
class Lista {
public:
    ~Lista() { destruir(lista) ; }
    Lista() : sz(0), lista(NULL) { }
    Lista(const Lista& o)
        : sz(o.sz), lista(duplicar(o.lista)) { }
    Lista& operator = (const Lista& o)
    {
        if (this != &o) {
            destruir(lista) ;
            sz = o.sz ;
            lista = duplicar(o.lista) ;
        }
        return *this ;
    }
    void insertar(int pos, const Tipo& d) { ... }
    void eliminar(int pos) { ... }
    // ...
private:
    struct Nodo ;
    typedef Nodo* PNodo ;
    struct Nodo {
        PNodo sig ;
        Tipo dato ;
    } ;
    //-- Métodos privados --
    void destruir(PNodo& l) const { ... }
```

```
    PNodo duplicar(PNodo l) const { ... }
    // ...
    //-- Atributos privados --
    int sz ;
    PNodo lista ;
} ;
```

13.7. Tipo Abstracto de Datos Lista Enlazada Genérica

Aunque las listas enlazadas se pueden programar directamente entremezcladas con el código de resolución del problema en cuestión, es conveniente que su gestión se realice dentro de una *abstracción* que aisle su tratamiento y permita una mejor gestión de sus recursos.

Con objeto de facilitar su estudio, el siguiente ejemplo es una implementación *simplificada* del tipo abstracto de datos *lista genérica* de elementos homogéneos. Nótese, sin embargo, que otras implementaciones pueden mejorar notablemente su eficiencia.

```
//- lista.hpp ----------------------------------------------------------------
#ifndef _lista_hpp_
#define _lista_hpp_
#include <cstddef>
#include <cassert>
namespace umalcc {
    template <typename Tipo>
    class Lista {
    public:
        //-- Métodos Públicos ----------

        // Destructor
        ~Lista() { destruir(lista) ; }

        // Constructor por Defecto
        Lista() : sz(0), lista(NULL) { }

        // Constructor de Copia
        Lista(const Lista& o)
            : sz(o.sz), lista(duplicar(o.lista)) { }

        // Operador de Asignación
        Lista& operator = (const Lista& o)
        {
            if (this != &o) {
                destruir(lista) ;
                sz = o.sz ;
                lista = duplicar(o.lista) ;
            }
            return *this ;
        }

        // Elimina todos los elementos de la lista actual (queda vacia)
        void clear()
        {
            destruir(lista) ;
            sz = 0 ;
        }

        // Devuelve el numero de elementos almacenados
        int size() const
        {
```

```
        return sz ;
    }

    // Devuelve true si el numero de elementos almacenados
    // alcanza la capacidad maxima de almacenamiento
    bool llena() const
    {
        return false;
    }

    // PRECOND: ( ! llena() && 0 <= pos && pos <= size())
    // Inserta (dato) en la lista actual en la posicion (pos)
    void insertar(int pos, const Tipo& d)
    {
        assert(! llena()
               && 0 <= pos && pos <= size()) ;
        insertar_pos(lista, pos, d) ;
        ++sz ;
    }

    // PRECOND: (0 <= pos  && pos < size())
    // Elimina de la lista actual el elemento que ocupa la posicion (pos)
    void eliminar(int pos)
    {
        assert(0 <= pos && pos < size()) ;
        eliminar_pos(lista, pos) ;
        --sz ;
    }

    // PRECOND: (0 <= pos  && pos < size())
    // Devuelve el elemento de la lista actual que ocupa la posicion (pos)
    Tipo acceder(int pos) const
    {
        assert(0 <= pos && pos < size()) ;
        PNodo ptr = situar(lista, pos) ;
        assert(ptr != NULL) ;
        return ptr->dato ;
    }

    // PRECOND: (0 <= pos  && pos < size())
    // Asigna (dato) al elemento de la lista actual que ocupa la posicion (pos)
    void modificar(int pos, const Tipo& d)
    {
        assert(0 <= pos && pos < size()) ;
        PNodo ptr = situar(lista, pos) ;
        assert(ptr != NULL) ;
        ptr->dato = d ;
    }

private:
    //-- Tipos Privados ------

    struct Nodo ;
    typedef Nodo* PNodo ;
    struct Nodo {
        PNodo sig ;
        Tipo dato ;
    } ;
```

```
//-- Atributos privados --

int sz ;
PNodo lista ;

//-- Métodos Privados ----------

void destruir(PNodo& lst) const
{
    while (lst != NULL) {
        PNodo ptr = lst ;
        lst = lst->sig ;
        delete ptr ;
    }
}

PNodo situar(PNodo lst, int pos) const
{
    PNodo ptr = lst;
    while ((ptr != NULL)&&(pos > 0)) {
        ptr = ptr->sig;
        --pos;
    }
    return ptr;
}

void insertar_pos(PNodo& lst, int pos,
                  const Tipo& dt) const
{
    if (pos < 1) {
        PNodo ptr = new Nodo ;
        ptr->dato = dt ;
        ptr->sig = lst ;
        lst = ptr ;
    } else {
        PNodo ant = situar(lst, pos - 1);
        if (ant != NULL) {
            PNodo ptr = new Nodo ;
            ptr->dato = dt ;
            ptr->sig = ant->sig ;
            ant->sig = ptr ;
        }
    }
}

void eliminar_pos(PNodo& lst, int pos) const
{
    if (lst != NULL) {
        if (pos < 1) {
            PNodo ptr = lst ;
            lst = lst->sig ;
            delete ptr ;
        } else {
            PNodo ant = situar(lst, pos - 1) ;
            if ((ant != NULL)&&(ant->sig != NULL)) {
                PNodo act = ant->sig ;
                ant->sig = act->sig ;
                delete act ;
            }
```

```
                }
            }
        }

        PNodo duplicar(PNodo lst) const
        {
            PNodo nueva = NULL;
            if (lst != NULL) {
                nueva = new Nodo ;
                nueva->dato = lst->dato ;
                PNodo u = nueva ;
                PNodo p = lst->sig ;
                while (p != NULL) {
                    u->sig = new Nodo ;
                    u->sig->dato = p->dato ;
                    u = u->sig ;
                    p = p->sig ;
                }
                u->sig = NULL ;
            }
            return nueva;
        }

    } ;     // class
}           // namespace
#endif
//-----------------------------------------------------------------------
```

Se puede apreciar como tanto el constructor de copia, como el operador de asignación duplican la lista almacenada, y por el contrario tanto el destructor como el método `clear()` liberan todos los recursos que el objeto tenga asignados. Así mismo, el método `duplicar` invoca a la destrucción de los recursos que tuviese antes de duplicar y copiar la nueva lista.

A continuación se puede ver un ejemplo de utilización del tipo abstracto de datos *lista genérica* definido anteriormente.

```
//- fichero: main.cpp ------------------------------------------------
#include <iostream>
#include <cctype>
#include <cassert>
#include "lista.hpp"
using namespace std ;
using namespace umalcc ;
//-----------------------------------------------------------------------
// Instanciación de la Lista genérica para almacenar números enteros
typedef Lista<int> ListaInt;
//-----------------------------------------------------------------------
void leer_pos(int& pos, int limite)
{
    assert(limite > 0);
    do {
        cout << "Introduzca posicion ( < " << limite << " ): " ;
        cin >> pos;
    } while (pos < 0 || pos >= limite);
}
//---------------------------------
void leer_dato(int& dato)
{
    cout << "Introduzca un dato: " ;
    cin >> dato;
```

```
}
//---------------------------------
void leer(ListaInt& lista)
{
    int dato ;
    lista.clear() ;
    cout << "Introduzca datos (0 -> FIN): " << endl ;
    cin >> dato ;
    while ((dato != 0)&&( ! lista.llena())) {
        lista.insertar(lista.size(), dato) ;
        cin >> dato ;
    }
}
//---------------------------------
void escribir(const ListaInt& lista)
{
    cout << "Lista: " ;
    for (int i = 0 ; i < lista.size() ; ++i) {
        cout << lista.acceder(i) << " " ;
    }
    cout << endl ;
}
//---------------------------------
void prueba_asg(const ListaInt& lista)
{
    cout << "Constructor de Copia" << endl ;
    ListaInt lst(lista) ;
    escribir(lst) ;
    cout << "Operador de Asignacion" << endl ;
    lst = lista ;
    escribir(lst) ;
}
//-----------------------------------------------------------------------
char menu()
{
    char op ;
    cout << endl ;
    cout << "X. Fin" << endl ;
    cout << "A. Leer Lista" << endl ;
    cout << "B. Borrar Lista" << endl ;
    cout << "C. Insertar Posicion" << endl ;
    cout << "D. Eliminar Posicion" << endl ;
    cout << "E. Acceder Posicion" << endl ;
    cout << "F. Modificar Posicion" << endl ;
    cout << "G. Prueba Copia y Asignacion" << endl ;
    do {
        cout << endl << "    Opcion: " ;
        cin >> op ;
        op = char(toupper(op)) ;
    } while (!((op == 'X')||((op >= 'A')&&(op <= 'G')))) ;
    cout << endl ;
    return op ;
}
//-----------------------------------------------------------------------
int main()
{
    ListaInt lista ;
    int dato ;
    int pos ;
```

```
    char op = ' ' ;
    do {
        op = menu() ;
        switch (op) {
        case 'A':
            leer(lista) ;
            escribir(lista) ;
            break ;
        case 'B':
            lista.clear() ;
            escribir(lista) ;
            break ;
        case 'C':
            if (lista.llena()) {
                cout << "Error: Lista llena" << endl ;
            } else {
                leer_pos(pos, lista.size()+1) ;
                leer_dato(dato) ;
                lista.insertar(pos, dato) ;
                escribir(lista) ;
            }
            break ;
        case 'D':
            if (lista.size() == 0) {
                cout << "Error: lista vacia" << endl ;
            } else {
                leer_pos(pos, lista.size()) ;
                lista.eliminar(pos) ;
                escribir(lista) ;
            }
            break ;
        case 'E':
            if (lista.size() == 0) {
                cout << "Error: lista vacia" << endl ;
            } else {
                leer_pos(pos, lista.size()) ;
                cout << "Lista[" << pos << "]: " << lista.acceder(pos) << endl ;
                escribir(lista) ;
            }
            break ;
        case 'F':
            if (lista.size() == 0) {
                cout << "Error: lista vacia" << endl ;
            } else {
                leer_pos(pos, lista.size()) ;
                leer_dato(dato) ;
                lista.modificar(pos, dato) ;
                escribir(lista) ;
            }
            break ;
        case 'G':
            prueba_asg(lista) ;
            break ;
        }
    } while (op != 'X') ;
}
//- fin: main.cpp -------------------------------------------------
```

Capítulo 14

Introducción a los Contenedores de la Biblioteca Estándar (STL)

Los *contenedores* de la biblioteca estándar proporcionan un método general para almacenar y acceder a una colección de elementos homogéneos, proporcionando cada uno de ellos diferentes características que los hacen adecuados a diferentes necesidades.

En este capítulo introductorio se mostrarán las principales operaciones que se pueden realizar con los siguientes contenedores: el tipo `vector` y el tipo `deque` (para el tipo `array` véase 6.4). Así como con los siguientes *adaptadores de contenedores*: el tipo `stack` y el tipo `queue` de la biblioteca estándar, que implementan el *TAD Pila* y el *TAD Cola* respectivamente.

La biblioteca estándar también define otros tipos de contenedores optimizados para diferentes circunstancias, pero no serán explicados debido a que su estudio requiere mayores conocimientos que los obtenidos en un curso introductorio.

Contenedor	Tipo	Acceso	Inserción	Eliminación
`stack` (adaptador)	*TAD Pila*	Directo (al final)	Al final	Al final
`queue` (adaptador)	*TAD Cola*	Directo (al principio)	Al final	Al principio
`array`	Secuencia	Directo (pos)	–	–
`vector`	Secuencia	Directo (pos)	Al final	Al final
`deque`	Secuencia	Directo (pos)	Al final + al principio	Al final + Al principio
`list`	Secuencia	Secuencial (bidir)	Cualquier posición	Cualquier posición
`forward_list`	Secuencia	Secuencial (fw)	Cualquier posición	Cualquier posición
`map`	Asociativo	Binario por clave	Por Clave	Por Clave
`set`	Asociativo	Binario por clave	Por Clave	Por Clave
`multimap`	Asociativo	Binario por clave	Por Clave	Por Clave
`multiset`	Asociativo	Binario por clave	Por Clave	Por Clave
`unordered_map`	Asociativo	Hash por clave	Por Clave	Por Clave
`unordered_set`	Asociativo	Hash por clave	Por Clave	Por Clave
`unordered_multimap`	Asociativo	Hash por clave	Por Clave	Por Clave
`unordered_multiset`	Asociativo	Hash por clave	Por Clave	Por Clave

Paso de Parámetros de Contenedores

Los contenedores de la biblioteca estándar se pueden pasar como parámetros a subprogramas como cualquier otro tipo compuesto, y por lo tanto se aplican los mecanismos de paso de parámetros para tipos compuestos explicados en la sección 6.1. Es decir, los parámetros de entrada se pasarán por referencia constante, mientras que los parámetros de salida y entrada/salida se pasarán por referencia.

Así mismo, como norma general, salvo excepciones, no es adecuado que las funciones retornen valores de tipos de los contenedores, debido a la sobrecarga que generalmente conlleva dicha operación para el caso de los tipos compuestos. En estos casos suele ser más adecuado que el valor se devuelva como un parámetro por referencia.

14.1. Vector

El contenedor de tipo `vector<...>` representa una secuencia de elementos homogéneos optimizada para el acceso directo a los elementos según su posición, así como también para la inserción de elementos al final de la secuencia y para la eliminación de elementos del final de la secuencia. Para utilizar un contenedor de tipo `vector` se debe incluir la biblioteca estándar `<vector>`, de tal forma que sus definiciones se encuentran dentro del espacio de nombres `std`:

```
#include <vector>
```

El tipo `vector` es similar al tipo `array`, salvo en el hecho de que los vectores se caracterizan porque su tamaño puede crecer en tiempo de ejecución dependiendo de las necesidades surgidas durante la ejecución del programa. Por ello, a diferencia de los arrays, no es necesario especificar un tamaño fijo y predeterminado en tiempo de compilación respecto al número de elementos que pueda contener.

El número máximo de elementos que se pueden almacenar en una variable de tipo `vector` no está especificado, y se pueden almacenar elementos mientras haya capacidad suficiente en la memoria del ordenador donde se ejecute el programa.

Nótese que en los siguientes ejemplos, por simplicidad, tanto el número de elementos como el valor inicial de los mismos están especificados mediante valores constantes, sin embargo, también se pueden especificar como valores de variables y expresiones calculados en tiempo de ejecución.

Instanciación del Tipo Vector

Se pueden definir explícitamente instanciaciones del tipo `vector` para tipos de elementos concretos mediante la declaración `typedef`. Por ejemplo la siguiente definición declara el tipo `Vect_Int` como un tipo vector de números enteros.

```
typedef std::vector<int> Vect_Int ;
```

Las siguientes definiciones declaran el tipo `Matriz` como un vector de dos dimensiones de números enteros.

```
typedef std::vector<int>  Fila ;
typedef std::vector<Fila> Matriz ;
```

Construcción de un Objeto de Tipo Vector

Se pueden definir variables de un tipo vector previamente definido explícitamente, o directamente de la instanciación del tipo. Por ejemplo, el siguiente código define dos variables (`v1` y `v2`) de tipo vector de números enteros, así como la variable `m` de tipo vector de dos dimensiones de números enteros.

```
int main()
{
    Vect_Int v1 ;          // vector de enteros vacío
    std::vector<int> v2 ;  // vector de enteros vacío
    Matriz m ;             // vector de dos dimensiones de enteros vacío
    // ...
}
```

El constructor por defecto del tipo `vector` crea un objeto `vector` inicialmente vacío, sin elementos. Posteriormente se podrán añadir y eliminar elementos cuando sea necesario.

También es posible crear un objeto vector con un número inicial de elementos con un valor inicial por defecto, al que posteriormente se le podrán añadir nuevos elementos. Este número inicial de elementos puede ser tanto una constante, como el valor de una variable calculado en tiempo de ejecución.

```
int main()
{
    Vect_Int v1(10) ;       // vector con 10 enteros con valor inicial 0
    Matriz m(10, Fila(5)) ; // matriz de 10x5 enteros con valor inicial 0
    // ...
}
```

Así mismo, también se puede especificar el valor que tomarán los elementos creados inicialmente.

```
int main()
{
    Vect_Int v1(10, 3) ;       // vector con 10 enteros con valor inicial 3
    Matriz m(10, Fila(5, 3)) ; // matriz de 10x5 enteros con valor inicial 3
    // ...
}
```

También es posible inicializar un vector con el contenido de otro vector de igual tipo, invocando al constructor de copia:

```
int main()
{
    Vect_Int v1(10, 3) ;     // vector con 10 enteros con valor inicial 3
    Vect_Int v2(v1) ;        // vector con el mismo contenido de v1
    Vect_Int v3 = v1 ;       // vector con el mismo contenido de v1
    Vect_Int v4 = Vect_Int(7, 5) ; // vector con 7 elementos de valor 5
    // ...
}
```

Asignación de un Objeto de Tipo Vector

Es posible la asignación de vectores de igual tipo. En este caso, se destruye el valor anterior del vector destino de la asignación.

```
int main()
{
    Vect_Int v1(10, 3) ;    // vector con 10 enteros con valor inicial 3
    Vect_Int v2 ;           // vector de enteros vacío

    v2 = v1 ;               // asigna el contenido de v1 a v2
    v2.assign(5, 7) ;       // asigna 5 enteros con valor inicial 7
    v2 = Vect_Int(5, 7) ;   // asigna un vector con 5 elementos de valor 7
}
```

Así mismo, también es posible intercambiar (*swap* en inglés) de forma eficiente el contenido entre dos vectores utilizando el método `swap`. Por ejemplo:

```
int main()
{
    Vect_Int v1(10, 5) ;    // v1 = { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5 }
    Vect_Int v2(5, 7) ;     // v2 = { 7, 7, 7, 7, 7 }

    v1.swap(v2) ;           // v1 = { 7, 7, 7, 7, 7 }
                            // v2 = { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5 }
}
```

Control sobre los Elementos de un Vector

El número de elementos actualmente almacenados en un vector se obtiene mediante el método `size()`. Por ejemplo:

```
int main()
{
    Vect_Int v1(10, 3) ;    // vector con 10 enteros con valor inicial 3
    int n = v1.size() ;     // número de elementos de v1
}
```

Es posible tanto añadir un elemento al final de un vector mediante el método `push_back(...)`, como eliminar el último elemento del vector mediante el método `pop_back()` (en este caso el vector no debe estar vacío). Así mismo, el método `clear()` elimina todos los elementos del vector. Por ejemplo:

```
int main()
{
    Vect_Int v(5) ;                          // v = { 0, 0, 0, 0, 0 }
    for (int i = 1 ; i <= 3 ; ++i) {
        v.push_back(i) ;
    }                                        // v = { 0, 0, 0, 0, 0, 1, 2, 3 }
    for (int i = 0 ; i < int(v.size()) ; ++i) {
        cout << v[i] << " " ;
    }                                        // muestra: 0 0 0 0 0 1 2 3
    cout << endl ;
    while (v.size() > 3) {
        v.pop_back() ;
    }                                        // v = { 0, 0, 0 }
    v.clear() ;                              // v = { }
}
```

También es posible cambiar el tamaño del número de elementos almacenados en el vector. Así, el método `resize(...)` reajusta el número de elementos contenidos en un vector. Si el número especificado es menor que el número actual de elementos, se eliminarán del final del vector tantos elementos como sea necesario para reducir el vector hasta el número de elementos especificado. Si por el contrario, el número especificado es mayor que el número actual de elementos, entonces se añadirán al final del vector tantos elementos como sea necesario para alcanzar el nuevo número de elementos especificado (con el valor especificado o con el valor por defecto). Por ejemplo:

```
int main()
{
    Vect_Int v(10, 1) ;  // v = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 }
    v.resize(5) ;        // v = { 1, 1, 1, 1, 1 }
    v.resize(9, 2) ;     // v = { 1, 1, 1, 1, 1, 2, 2, 2, 2 }
    v.resize(7, 3) ;     // v = { 1, 1, 1, 1, 1, 2, 2 }
    v.resize(10) ;       // v = { 1, 1, 1, 1, 1, 2, 2, 0, 0, 0 }
}
```

Acceso a los Elementos de un Vector

Es posible acceder a cada elemento del vector individualmente, según el índice de la posición que ocupe, tanto para obtener su valor almacenado, como para modificarlo mediante el operador de indexación `[]`. El primer elemento ocupa la posición cero (0), y el último elemento almacenado en el vector `v` ocupa la posición `v.size()-1`. Por ejemplo:

```
int main()
{
    Vect_Int v(10) ;
    for (int i = 0 ; i < int(v.size()) ; ++i) {
        v[i] = i ;
    }
    for (int i = 0 ; i < int(v.size()) ; ++i) {
        cout << v[i] << " " ;
    }
```

```
    cout << endl ;
}
```

El lenguaje de programación C++ no comprueba que los accesos a los elementos de un vector sean correctos y se encuentren dentro de los límites válidos del vector, por lo que será responsabilidad del programador comprobar que así sea.

Sin embargo, en *GNU G++*, la opción de compilación `-D_GLIBCXX_DEBUG` permite comprobar los índices de acceso.

También es posible acceder a un determinado elemento mediante el método `at(i)`, de tal forma que si el valor del índice `i` está fuera del rango válido, entonces se lanzará una excepción `out_of_range` que abortará la ejecución del programa. Se puede tanto utilizar como modificar el valor de este elemento.

```
int main()
{
    Vect_Int v(10) ;
    for (int i = 0 ; i < int(v.size()) ; ++i) {
        v.at(i) = i ;
    }
    for (int i = 0 ; i < int(v.size()) ; ++i) {
        cout << v.at(i) << " " ;
    }
    cout << endl ;
}
```

Comparación Lexicográfica entre Vectores

Es posible realizar la comparación lexicográfica (`==`, `!=`, `>`, `>=`, `<`, `<=`) entre vectores del mismo tipo siempre y cuando los operadores de comparación estén definidos para el tipo de los componentes del vector. Por ejemplo:

```
int main()
{
    Vect_Int v1(10, 7) ;  // v1 = { 7, 7, 7, 7, 7, 7, 7, 7, 7, 7 }
    Vect_Int v2(5, 3) ;   // v2 = { 3, 3, 3, 3, 3 }
    if (v1 == v2) {
        cout << "Iguales" << endl ;
    } else {
        cout << "Distintos" << endl ;
    }
    if (v1 < v2) {
        cout << "Menor" << endl ;
    } else {
        cout << "Mayor o Igual" << endl ;
    }
}
```

14.2. Deque

El contenedor de tipo `deque<...>` representa una secuencia de elementos homogéneos optimizada para el acceso directo a los elementos según su posición, así como también para la inserción de elementos al principio y al final de la secuencia y para la eliminación de elementos del principio y del final de la secuencia. Para utilizar un contenedor de tipo `deque` se debe incluir la biblioteca estándar `<deque>`, de tal forma que sus definiciones se encuentran dentro del espacio de nombres `std`:

```
#include <deque>
```

El contenedor `deque` presenta el mismo interfaz público que el contenedor `vector`, pero añade dos métodos nuevos para facilitar la inserción y eliminación de elementos al principio de la secuencia (`push_front(...)` y `pop_front()`).

El número máximo de elementos que se pueden almacenar en una variable de tipo `deque` no está especificado, y se pueden almacenar elementos mientras haya capacidad suficiente en la memoria del ordenador donde se ejecute el programa.

Instanciación del Tipo Deque

Se pueden definir explícitamente instanciaciones del tipo `deque` para tipos de elementos concretos mediante la declaración `typedef`. Por ejemplo la siguiente definición declara el tipo `Deque_Int` como un tipo deque de números enteros.

```
typedef std::deque<int> Deque_Int ;
```

Construcción de un Objeto de Tipo Deque

Se pueden definir variables de un tipo deque previamente definido explícitamente, o directamente de la instanciación del tipo. Por ejemplo, el siguiente código define dos variables (`v1` y `v2`) de tipo deque de números enteros.

```
int main()
{
    Deque_Int v1 ;          // deque de enteros vacío
    std::deque<int> v2 ;    // deque de enteros vacío
    // ...
}
```

El constructor por defecto del tipo `deque` crea un objeto `deque` inicialmente vacío, sin elementos. Posteriormente se podrán añadir y eliminar elementos cuando sea necesario.

También es posible crear un objeto deque con un número inicial de elementos con un valor inicial por defecto, al que posteriormente se le podrán añadir nuevos elementos. Este número inicial de elementos puede ser tanto una constante, como el valor de una variable calculado en tiempo de ejecución.

```
int main()
{
    Deque_Int v1(10) ;       // deque con 10 enteros con valor inicial 0
    // ...
}
```

Así mismo, también se puede especificar el valor que tomarán los elementos creados inicialmente.

```
int main()
{
    Deque_Int v1(10, 3) ;       // deque con 10 enteros con valor inicial 3
    // ...
}
```

También es posible inicializar un deque con el contenido de otro deque de igual tipo, invocando al constructor de copia:

```
int main()
{
    Deque_Int v1(10, 3) ;     // deque con 10 enteros con valor inicial 3
    Deque_Int v2(v1) ;        // deque con el mismo contenido de v1
    Deque_Int v3 = v1 ;       // deque con el mismo contenido de v1
    Deque_Int v4 = Deque_Int(7, 5) ; // deque con 7 elementos de valor 5
    // ...
}
```

Asignación de un Objeto de Tipo Deque

Es posible la asignación de deques de igual tipo. En este caso, se destruye el valor anterior del deque destino de la asignación.

```
int main()
{
    Deque_Int v1(10, 3) ;    // deque con 10 enteros con valor inicial 3
    Deque_Int v2 ;           // deque de enteros vacío

    v2 = v1 ;                // asigna el contenido de v1 a v2
    v2.assign(5, 7) ;        // asigna 5 enteros con valor inicial 7
    v2 = Deque_Int(5, 7) ;   // asigna un deque con 5 elementos de valor 7
}
```

Así mismo, también es posible intercambiar (*swap* en inglés) de forma eficiente el contenido entre dos deques utilizando el método `swap`. Por ejemplo:

```
int main()
{
    Deque_Int v1(10, 5) ;    // v1 = { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5 }
    Deque_Int v2(5, 7) ;     // v2 = { 7, 7, 7, 7, 7 }

    v1.swap(v2) ;            // v1 = { 7, 7, 7, 7, 7 }
                             // v2 = { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5 }
}
```

Control sobre los Elementos de un Deque

El número de elementos actualmente almacenados en un deque se obtiene mediante el método `size()`. Por ejemplo:

```
int main()
{
    Deque_Int v1(10, 3) ;    // deque con 10 enteros con valor inicial 3
    int n = v1.size() ;      // número de elementos de v1
}
```

Es posible tanto añadir un elemento al final de un deque mediante el método `push_back(...)`, como eliminar el último elemento del deque mediante el método `pop_back()` (en este caso el deque no debe estar vacío). Así mismo, el método `clear()` elimina todos los elementos del deque. Por ejemplo:

```
int main()
{
    Deque_Int v(5) ;                          // v = { 0, 0, 0, 0, 0 }
    for (int i = 1 ; i <= 3 ; ++i) {
        v.push_back(i) ;
    }                                         // v = { 0, 0, 0, 0, 0, 1, 2, 3 }
    for (int i = 1 ; i <= 2 ; ++i) {
        v.push_front(i) ;
    }                                         // v = { 2, 1, 0, 0, 0, 0, 0, 1, 2, 3 }
    for (int i = 0 ; i < int(v.size()) ; ++i) {
        cout << v[i] << " " ;
    }                                         // muestra: 2 1 0 0 0 0 0 1 2 3
    cout << endl ;
    while (v.size() > 5) {
        v.pop_back() ;
    }                                         // v = { 2, 1, 0, 0, 0 }
    while (v.size() > 3) {
        v.pop_front() ;
```

```
    }                                       // v = { 0, 0, 0 }
    v.clear() ;                             // v = { }
}
```

También es posible cambiar el tamaño del número de elementos almacenados en el deque. Así, el método `resize(...)` reajusta el número de elementos contenidos en un deque. Si el número especificado es menor que el número actual de elementos, se eliminarán del final del deque tantos elementos como sea necesario para reducir el deque hasta el número de elementos especificado. Si por el contrario, el número especificado es mayor que el número actual de elementos, entonces se añadirán al final del deque tantos elementos como sea necesario para alcanzar el nuevo número de elementos especificado (con el valor especificado o con el valor por defecto). Por ejemplo:

```
int main()
{
    Deque_Int v(10, 1) ;  // v = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 }
    v.resize(5) ;         // v = { 1, 1, 1, 1, 1 }
    v.resize(9, 2) ;      // v = { 1, 1, 1, 1, 1, 2, 2, 2, 2 }
    v.resize(7, 3) ;      // v = { 1, 1, 1, 1, 1, 2, 2 }
    v.resize(10) ;        // v = { 1, 1, 1, 1, 1, 2, 2, 0, 0, 0 }
}
```

Acceso a los Elementos de un Deque

Es posible acceder a cada elemento del deque individualmente, según el índice de la posición que ocupe, tanto para obtener su valor almacenado, como para modificarlo mediante el operador de indexación `[]`. El primer elemento ocupa la posición cero (0), y el último elemento almacenado en el deque `v` ocupa la posición `v.size()-1`. Por ejemplo:

```
int main()
{
    Deque_Int v(10) ;
    for (int i = 0 ; i < int(v.size()) ; ++i) {
        v[i] = i ;
    }
    for (int i = 0 ; i < int(v.size()) ; ++i) {
        cout << v[i] << " " ;
    }
    cout << endl ;
}
```

El lenguaje de programación C++ no comprueba que los accesos a los elementos de un deque sean correctos y se encuentren dentro de los límites válidos del deque, por lo que será responsabilidad del programador comprobar que así sea.

Sin embargo, en *GNU G++*, la opción de compilación `-D_GLIBCXX_DEBUG` permite comprobar los índices de acceso.

También es posible acceder a un determinado elemento mediante el método `at(i)`, de tal forma que si el valor del índice `i` está fuera del rango válido, entonces se lanzará una excepción `out_of_range` que abortará la ejecución del programa. Se puede tanto utilizar como modificar el valor de este elemento.

```
int main()
{
    Deque_Int v(10) ;
    for (int i = 0 ; i < int(v.size()) ; ++i) {
        v.at(i) = i ;
    }
    for (int i = 0 ; i < int(v.size()) ; ++i) {
        cout << v.at(i) << " " ;
    }
    cout << endl ;
}
```

Comparación Lexicográfica entre Deques

Es posible realizar la comparación lexicográfica (==, !=, >, >=, <, <=) entre deques del mismo tipo siempre y cuando los operadores de comparación estén definidos para el tipo de los componentes del deque. Por ejemplo:

```
int main()
{
    Deque_Int v1(10, 7) ;  // v1 = { 7, 7, 7, 7, 7, 7, 7, 7, 7, 7 }
    Deque_Int v2(5, 3) ;   // v2 = { 3, 3, 3, 3, 3 }
    if (v1 == v2) {
        cout << "Iguales" << endl ;
    } else {
        cout << "Distintos" << endl ;
    }
    if (v1 < v2) {
        cout << "Menor" << endl ;
    } else {
        cout << "Mayor o Igual" << endl ;
    }
}
```

14.3. Stack

El adaptador de contenedor de tipo `stack<...>` representa el tipo abstracto de datos *Pila*, como una colección ordenada (según el orden de inserción) de elementos homogéneos donde se pueden introducir elementos (manteniendo el orden de inserción) y sacar elementos de ella (en orden inverso al orden de inserción), de tal forma que el primer elemento que sale de la pila es el último elemento que ha sido introducido en ella. Además, también es posible comprobar si la pila contiene elementos, de tal forma que no se podrá sacar ningún elemento de una pila vacía. Para utilizar un adaptador de contenedor de tipo `stack` se debe incluir la biblioteca estándar `<stack>`, de tal forma que sus definiciones se encuentran dentro del espacio de nombres `std`:

```
#include <stack>
```

El número máximo de elementos que se pueden almacenar en una variable de tipo `stack` no está especificado, y se pueden introducir elementos mientras haya capacidad suficiente en la memoria del ordenador donde se ejecute el programa.

Instanciación del Tipo Stack

Se pueden definir explícitamente instanciaciones del tipo `stack` para tipos de elementos concretos mediante la declaración `typedef`. Por ejemplo la siguiente definición declara el tipo `Stack_Int` como un tipo pila de números enteros.

```
typedef std::stack<int> Stack_Int ;
```

Construcción de un Objeto de Tipo Pila

Se pueden definir variables de un tipo pila previamente definido explícitamente, o directamente de la instanciación del tipo. Por ejemplo, el siguiente código define dos variables (`s1` y `s2`) de tipo pila de números enteros.

```
int main()
{
    Stack_Int s1 ;        // stack de enteros vacío
    std::stack<int> s2 ;  // stack de enteros vacío
    // ...
}
```

El constructor por defecto del tipo `stack` crea un objeto `stack` inicialmente vacío, sin elementos. Posteriormente se podrán añadir y eliminar elementos cuando sea necesario.

También es posible inicializar una pila con el contenido de otra pila de igual tipo:

```
int main()
{
    Stack_Int s1 ;       // stack de enteros vacío
    // ...
    Stack_Int s2(s1) ;   // stack con el mismo contenido de s1
    Stack_Int s3 = s1 ;  // stack con el mismo contenido de s1
    Stack_Int s4 = Stack_Int() ;  // copia el contenido de stack vacío
    // ...
}
```

Asignación de un Objeto de Tipo Pila

Es posible la asignación de pilas de igual tipo. En este caso, se destruye el valor anterior de la pila destino de la asignación.

```
int main()
{
    Stack_Int s1 ;       // stack de enteros vacío
    Stack_Int s2 ;       // stack de enteros vacío

    s2 = s1 ;            // asigna el contenido de s1 a s2
    s2 = Stack_Int() ;   // asigna el contenido de stack vacío
}
```

Control y Acceso a los Elementos de una Pila

Es posible tanto añadir un elemento una pila mediante el método `push(...)`, como eliminar el último elemento introducido en la pila mediante el método `pop()` (en este caso la pila no debe estar vacía).

Por otra parte, el método `empty()` indica si una pila está vacía o no, mientras que el número de elementos actualmente almacenados en una pila se obtiene mediante el método `size()`.

Así mismo, se puede acceder al último elemento introducido en la pila mediante el método `top()`. Se puede tanto utilizar como modificar el valor de este elemento (en este caso la pila no debe estar vacía).

Por ejemplo:

```
int main()
{
    Stack_Int s ;                              // s = { }
    for (int i = 1 ; i <= 3 ; ++i) {
        s.push(i) ;
    }                                         // s = { 1, 2, 3 }

    s.top() = 5 ;                              // s = { 1, 2, 5 }

    s.pop() ;                                  // s = { 1, 2 }
    s.pop() ;                                  // s = { 1 }
    s.push(7) ;                                // s = { 1, 7 }
    s.push(9) ;                                // s = { 1, 7, 9 }

    cout << s.size() << endl ;                 // muestra: 3

    while (! s.empty()) {
        cout << s.top() << " " ;               // muestra: 9 7 1
        s.pop() ;
```

```
    }                                          // s = { }
    cout << endl ;
}
```

Comparación Lexicográfica entre Pilas

Es posible realizar la comparación lexicográfica (`==`, `!=`, `>`, `>=`, `<`, `<=`) entre pilas del mismo tipo siempre y cuando los operadores de comparación estén definidos para el tipo de los componentes de la pila. Por ejemplo:

```
int main()
{
    Stack_Int s1 ;
    Stack_Int s2 ;
    // ...
    if (s1 == s2) {
        cout << "Iguales" << endl ;
    } else {
        cout << "Distintos" << endl ;
    }
    if (s1 < s2) {
        cout << "Menor" << endl ;
    } else {
        cout << "Mayor o Igual" << endl ;
    }
}
```

14.4. Queue

El adaptador de contenedor de tipo `queue<...>` representa el tipo abstracto de datos *Cola*, como una colección ordenada (según el orden de inserción) de elementos homogéneos donde se pueden introducir elementos (manteniendo el orden de inserción) y sacar elementos de ella (en el mismo orden al orden de inserción), de tal forma que el primer elemento que sale de la cola es el primer elemento que ha sido introducido en ella. Además, también es posible comprobar si la cola contiene elementos, de tal forma que no se podrá sacar ningún elemento de una cola vacía. Para utilizar un adaptador de contenedor de tipo `queue` se debe incluir la biblioteca estándar `<queue>`, de tal forma que sus definiciones se encuentran dentro del espacio de nombres `std`:

```
#include <queue>
```

El número máximo de elementos que se pueden almacenar en una variable de tipo `queue` no está especificado, y se pueden introducir elementos mientras haya capacidad suficiente en la memoria del ordenador donde se ejecute el programa.

Instanciación del Tipo Queue

Se pueden definir explícitamente instanciaciones del tipo `queue` para tipos de elementos concretos mediante la declaración `typedef`. Por ejemplo la siguiente definición declara el tipo `Queue_Int` como un tipo cola de números enteros.

```
typedef std::queue<int> Queue_Int ;
```

Construcción de un Objeto de Tipo Cola

Se pueden definir variables de un tipo cola previamente definido explícitamente, o directamente de la instanciación del tipo. Por ejemplo, el siguiente código define dos variables (`c1` y `c2`) de tipo cola de números enteros.

```
int main()
{
    Queue_Int c1 ;        // queue de enteros vacío
    std::queue<int> c2 ;  // queue de enteros vacío
    // ...
}
```

El constructor por defecto del tipo `queue` crea un objeto `queue` inicialmente vacío, sin elementos. Posteriormente se podrán añadir y eliminar elementos cuando sea necesario.

También es posible inicializar una cola con el contenido de otra cola de igual tipo:

```
int main()
{
    Queue_Int c1 ;       // queue de enteros vacío
    // ...
    Queue_Int c2(c1) ;   // queue con el mismo contenido de c1
    Queue_Int c3 = c1 ;  // queue con el mismo contenido de c1
    Queue_Int c4 = Stack_Int() ;  // copia el contenido de queue vacío
    // ...
}
```

Asignación de un Objeto de Tipo Cola

Es posible la asignación de colas de igual tipo. En este caso, se destruye el valor anterior de la cola destino de la asignación.

```
int main()
{
    Queue_Int c1 ;       // queue de enteros vacío
    Queue_Int c2 ;       // queue de enteros vacío

    c2 = c1 ;            // asigna el contenido de c1 a c2
    c2 = Queue_Int() ;   // asigna el contenido de queue vacío
}
```

Control y Acceso a los Elementos de una Cola

Es posible tanto añadir un elemento una cola mediante el método `push(...)`, como eliminar el primer elemento introducido en la cola mediante el método `pop()` (en este caso la cola no debe estar vacía).

Por otra parte, el método `empty()` indica si una cola está vacía o no, mientras que el número de elementos actualmente almacenados en una cola se obtiene mediante el método `size()`.

Así mismo, se puede acceder al último elemento introducido en la cola mediante el método `back()`, así como al primer elemento introducido en ella mediante el método `front()`. Se pueden tanto utilizar como modificar el valor de estos elementos (en este caso la cola no debe estar vacía).

Por ejemplo:

```
int main()
{
    Queue_Int c ;                              // c = { }
    for (int i = 1 ; i <= 3 ; ++i) {
        c.push(i) ;
    }                                         // c = { 1, 2, 3 }

    c.front() = 6 ;                            // c = { 6, 2, 3 }
    c.back() = 5 ;                             // c = { 6, 2, 5 }

    c.pop() ;                                  // c = { 2, 5 }
    c.pop() ;                                  // c = { 5 }
```

```
    c.push(7) ;                                  // c = { 5, 7 }
    c.push(9) ;                                  // c = { 5, 7, 9 }

    cout << c.size() << endl ;                   // muestra: 3

    while (! c.empty()) {
        cout << c.front() << " " ;               // muestra: 5 7 9
        c.pop() ;
    }                                           // c = { }
    cout << endl ;
}
```

Comparación Lexicográfica entre Colas

Es posible realizar la comparación lexicográfica (==, !=, >, >=, <, <=) entre colas del mismo tipo siempre y cuando los operadores de comparación estén definidos para el tipo de los componentes de la cola. Por ejemplo:

```
int main()
{
    Queue_Int c1 ;
    Queue_Int c2 ;
    // ...
    if (c1 == c2) {
        cout << "Iguales" << endl ;
    } else {
        cout << "Distintos" << endl ;
    }
    if (c1 < c2) {
        cout << "Menor" << endl ;
    } else {
        cout << "Mayor o Igual" << endl ;
    }
}
```

14.5. Resolución de Problemas Utilizando Contenedores

Ejemplo 1: Agentes de Ventas

Diseñe un programa que lea y almacene las ventas realizadas por unos *agentes de ventas*, de tal forma que se eliminen aquellos agentes cuyas ventas sean inferiores a la media de las ventas realizadas.

```
//-----------------------------------------------------------------
#include <iostream>
#include <vector>
#include <string>
using namespace std ;

struct Agente {
    string nombre ;
    double ventas ;
} ;

typedef vector<Agente> VAgentes ;

void leer (VAgentes& v)
{
    v.clear() ;
```

```
    Agente a ;
    cout << "Introduzca Nombre: " ;
    getline(cin, a.nombre) ;
    while (( ! cin.fail()) && (a.nombre.size() > 0)) {
        cout << "Introduzca Ventas: " ;
        cin >> a.ventas ;
        cin.ignore(1000, '\n') ;
        v.push_back(a) ;
        cout << "Introduzca Nombre: " ;
        getline(cin, a.nombre) ;
    }
}

double media(const VAgentes& v)
{
    double suma=0.0 ;
    for (int i = 0 ; i < int(v.size()) ; ++i) {
        suma += v[i].ventas ;
    }
    return suma/double(v.size()) ;
}

void purgar(VAgentes& v, double media)
{
    // altera el orden secuencial de los elementos
    int i = 0 ;
    while (i < int(v.size())) {
        if (v[i].ventas < media) {
            v[i] = v[v.size()-1] ;
            v.pop_back() ;
        } else {
            ++i ;
        }
    }
}

void purgar_ordenado(VAgentes& v, double media)
{
    // mantiene el orden secuencial de los elementos
    int k = 0 ;
    while ((k < int(v.size()))&&(v[k].ventas >= media)) {
        ++k;
    }
    for (int i = k ; i < int(v.size()) ; ++i) {
        if(v[i].ventas >= media) {
            v[k] = v[i] ;
            ++k ;
        }
    }
    v.resize(k) ;
}

void imprimir(const VAgentes& v)
{
    for (int i = 0 ; i < int(v.size()) ; ++i) {
        cout << v[i].nombre << " " << v[i].ventas << endl ;
    }
}
```

```
int main ()
{
    VAgentes v ;
    leer(v) ;
    purgar(v, media(v)) ;
    imprimir(v) ;
}
//----------------------------------------------------------------
```

Ejemplo 2: Multiplicación de Matrices

Diseñe un programa que lea dos matrices de tamaños arbitrarios y muestre el resultado de multiplicar ambas matrices.

```
//----------------------------------------------------------------
#include <vector>
#include <iostream>
#include <iomanip>
using namespace std ;

typedef vector <double> Fila ;
typedef vector <Fila> Matriz ;

void imprimir(const Matriz& m)
{
    for (int f = 0 ; f < int(m.size()) ; ++f) {
        for (int c = 0 ; c < int(m[f].size()) ; ++c) {
            cout << setw(10) << setprecision(4)
                 << m[f][c] << " " ;
        }
        cout << endl ;
    }
}

void leer(Matriz& m)
{
    int nf, nc ;
    cout << "Introduzca el numero de filas: " ;
    cin >> nf ;
    cout << "Introduzca el numero de columnas: " ;
    cin >> nc ;
    m = Matriz(nf, Fila (nc)) ; // copia de la matriz completa
    cout << "Introduzca los elementos: " << endl ;
    for (int f = 0 ; f < int(m.size()) ; ++f) {
        for (int c = 0 ; c < int(m[f].size()) ; ++c) {
            cin >> m[f][c] ;
        }
    }
}

// otra opción más eficiente para la lectura de vectores
void leer_2(Matriz& m)
{
    int nf, nc ;
    cout << "Introduzca el numero de filas: " ;
    cin >> nf ;
    cout << "Introduzca el numero de columnas: " ;
    cin >> nc ;
    Matriz aux(nf, Fila (nc)) ;
    cout << "Introduzca los elementos: " << endl ;
```

```
    for (int f = 0 ; f < int(aux.size()) ; ++f) {
        for (int c = 0 ; c < int(aux[f].size()) ; ++c) {
            cin >> aux[f][c] ;
        }
    }
    m.swap(aux) ;  // evita la copia de la matriz completa
}

void multiplicar(const Matriz& m1, const Matriz& m2, Matriz& m3)
{
    m3.clear() ;
    if ((m1.size() > 0) && (m2.size() > 0) && (m2[0].size() > 0)
        && (m1[0].size() == m2.size())){
        Matriz aux(m1.size(), Fila(m2[0].size())) ;
        for (int f = 0 ; f < int(aux.size()) ; ++f) {
            for (int c = 0 ; c < int(aux[f].size()) ; ++c) {
                double suma = 0.0 ;
                for (int k = 0 ; k < int(m2.size()) ; ++k) {
                    suma += m1[f][k] * m2[k][c] ;
                }
                aux[f][c] = suma ;
            }
        }
        m3.swap(aux) ;  // evita la copia de la matriz completa
    }
}

int main()
{
    Matriz m1, m2, m3 ;
    leer(m1) ;
    leer(m2) ;
    multiplicar(m1, m2, m3) ;
    if (m3.size() == 0) {
        cout << "Error en la multiplicación de Matrices" << endl ;
    } else {
        imprimir(m3) ;
    }
}
//----------------------------------------------------------------
```

Capítulo 15

Bibliografía

- El Lenguaje de Programación C. 2.Ed.
  B.Kernighan, D. Ritchie
  Prentice Hall 1991

- The C++ Programming Language. Special Edition
  B. Stroustrup
  Addison Wesley 2000

Índice alfabético