

# **C++ PASO A PASO**

Sergio Luján Mora



TEXTOS DOCE

PUBLICACIONES  
**UNIVERSIDAD DE ALICANTE**

# Índice resumido

Índice resumido	III
Índice general	v
Índice de cuadros	XI
Índice de figuras	XIII
1. Introducción	1
2. Clases y objetos	7
3. Constructor y destructor	31
4. Funciones y clases amigas y reserva de memoria	55
5. Sobrecarga de operadores	81
6. Composición y herencia	119
7. Otros temas	135
8. Errores más comunes	141
9. Ejercicios	185
A. Palabras clave	191
B. Operadores	197
C. Sentencias	201
D. Herramientas	207

---

E. Código de las clases	229
Bibliografía recomendada	249
Índice alfabético	253

# Índice general

Índice resumido	III
Índice general	v
Índice de cuadros	xI
Índice de figuras	xIII
<b>1. Introducción</b>	<b>1</b>
1.1. Introducción . . . . .	1
1.2. Ventajas de C++ . . . . .	2
1.3. Objetivos de este libro . . . . .	2
1.4. Contenido de los capítulos . . . . .	3
1.5. Sistema operativo y compilador . . . . .	5
1.6. Convenciones tipográficas . . . . .	5
<b>2. Clases y objetos</b>	<b>7</b>
2.1. Introducción . . . . .	8
2.2. Declaración de una clase . . . . .	10
2.3. Acceso a los miembros de una clase . . . . .	11
2.4. Control de acceso . . . . .	12
2.5. Visualización de un objeto . . . . .	16
2.6. Empleo de punteros . . . . .	18
2.7. Separación de la interfaz y la implementación . . . . .	19
2.8. La herramienta make . . . . .	23
2.9. Ficheros de encabezado . . . . .	24
2.10. Uso de espacios de nombres . . . . .	25
2.11. Ejercicios de autoevaluación . . . . .	26
2.12. Ejercicios de programación . . . . .	28
2.12.1. Clase TVector . . . . .	28
2.12.2. Clase TCalendario . . . . .	28

2.13. Respuesta a los ejercicios de autoevaluación . . . . .	28
2.14. Respuesta a los ejercicios de programación . . . . .	30
2.14.1. Clase TVector . . . . .	30
2.14.2. Clase TCalendario . . . . .	30
<b>3. Constructor y destructor</b>	<b>31</b>
3.1. Sobrecarga de funciones . . . . .	32
3.2. Constructor . . . . .	34
3.3. Constructor por defecto . . . . .	34
3.4. Otros constructores . . . . .	36
3.5. Constructor de copia . . . . .	37
3.6. ¿Un constructor en la parte privada? . . . . .	42
3.7. Destructor . . . . .	43
3.8. Forma canónica de una clase . . . . .	44
3.9. Ejercicios de autoevaluación . . . . .	45
3.10. Ejercicios de programación . . . . .	47
3.10.1. Clase TCoordenada . . . . .	47
3.10.2. Clase TVector . . . . .	47
3.10.3. Clase TCalendario . . . . .	48
3.11. Respuesta a los ejercicios de autoevaluación . . . . .	48
3.12. Respuesta a los ejercicios de programación . . . . .	50
3.12.1. Clase TVector . . . . .	50
3.12.2. Clase TCalendario . . . . .	52
<b>4. Funciones y clases amigas y reserva de memoria</b>	<b>55</b>
4.1. Introducción . . . . .	56
4.2. Declaración de amistad . . . . .	56
4.3. Guardas de inclusión . . . . .	62
4.4. Administración de memoria dinámica . . . . .	64
4.5. Administración de memoria dinámica y arrays de objetos . . . . .	66
4.6. Compilación condicional . . . . .	70
4.7. Directivas #warning y #error . . . . .	72
4.8. Ejercicios de autoevaluación . . . . .	73
4.9. Ejercicios de programación . . . . .	75
4.9.1. Clase TVector . . . . .	75
4.9.2. Clase TCalendario . . . . .	75
4.10. Respuesta a los ejercicios de autoevaluación . . . . .	75
4.11. Respuesta a los ejercicios de programación . . . . .	77
4.11.1. Clase TVector . . . . .	77
4.11.2. Clase TCalendario . . . . .	78

<b>5. Sobrecarga de operadores</b>	<b>81</b>
5.1. Introducción . . . . .	82
5.2. Puntero this . . . . .	82
5.3. Modificador const . . . . .	83
5.4. Paso por referencia . . . . .	85
5.5. Sobrecarga de operadores . . . . .	87
5.6. Restricciones al sobrecargar un operador . . . . .	88
5.7. ¿Función miembro o función no miembro? . . . . .	88
5.8. Consejos . . . . .	89
5.9. Operador asignación . . . . .	90
5.10. Constructor de copia y operador asignación . . . . .	93
5.11. Operadores aritméticos . . . . .	94
5.12. Operadores de incremento y decremento . . . . .	98
5.13. Operadores abreviados . . . . .	99
5.14. Operadores de comparación . . . . .	100
5.15. Operadores de entrada y salida . . . . .	101
5.16. Operador corchete . . . . .	102
5.17. Ejercicios de autoevaluación . . . . .	105
5.18. Ejercicios de programación . . . . .	107
5.18.1. Clase TCoordenada . . . . .	107
5.18.2. Clase TLinea . . . . .	107
5.18.3. Clase TVector . . . . .	108
5.18.4. Clase TCalendario . . . . .	108
5.19. Respuesta a los ejercicios de autoevaluación . . . . .	109
5.20. Respuesta a los ejercicios de programación . . . . .	111
5.20.1. Clase TCoordenada . . . . .	111
5.20.2. Clase TLinea . . . . .	112
5.20.3. Clase TVector . . . . .	112
5.20.4. Clase TCalendario . . . . .	115
<b>6. Composición y herencia</b>	<b>119</b>
6.1. Introducción . . . . .	119
6.2. Composición . . . . .	120
6.3. Inicialización de los objetos miembro . . . . .	122
6.4. Herencia . . . . .	126
6.5. Ejercicios de autoevaluación . . . . .	129
6.6. Ejercicios de programación . . . . .	130
6.6.1. Clase TLinea . . . . .	130
6.6.2. Clase TCoordenadaV . . . . .	131
6.6.3. Clase TAgenda . . . . .	131
6.7. Respuesta a los ejercicios de autoevaluación . . . . .	132

<b>7. Otros temas</b>	<b>135</b>
7.1. Forma canónica de una clase . . . . .	135
7.2. Funciones de cero parámetros . . . . .	137
7.3. Valores por omisión de una función . . . . .	137
7.4. Funciones inline . . . . .	139
<b>8. Errores más comunes</b>	<b>141</b>
8.1. Introducción . . . . .	141
8.2. Sobre el fichero makefile y la compilación . . . . .	144
8.3. Sobre las directivas de inclusión . . . . .	146
8.4. Sobre las clases . . . . .	149
8.5. Sobre la sobrecarga de los operadores . . . . .	161
8.6. Sobre la memoria . . . . .	166
8.7. Sobre las cadenas . . . . .	172
8.8. Varios . . . . .	178
<b>9. Ejercicios</b>	<b>185</b>
9.1. Mentiras arriesgadas . . . . .	185
9.2. La historia interminable . . . . .	186
9.3. Pegado a ti . . . . .	187
9.4. Clase TComplejo . . . . .	188
<b>A. Palabras clave</b>	<b>191</b>
A.1. Lista de palabras clave . . . . .	191
<b>B. Operadores</b>	<b>197</b>
B.1. Lista de operadores . . . . .	197
<b>C. Sentencias</b>	<b>201</b>
C.1. Introducción . . . . .	201
C.1.1. Asignación . . . . .	202
C.1.2. Sentencia compuesta (bloque de código) . . . . .	202
C.1.3. Sentencia condicional . . . . .	202
C.1.4. Sentencia condicional múltiple . . . . .	203
C.1.5. Sentencia de selección . . . . .	203
C.1.6. Bucle con contador . . . . .	204
C.1.7. Bucle con condición inicial . . . . .	204
C.1.8. Bucle con condición final . . . . .	205
<b>D. Herramientas</b>	<b>207</b>
D.1. Editor JOE . . . . .	208
D.1.1. Comandos básicos . . . . .	208
D.1.2. Bloques de texto . . . . .	208

D.1.3. Movimiento . . . . .	210
D.1.4. Ayuda . . . . .	210
D.2. Editor vim . . . . .	212
D.2.1. Salir de vim . . . . .	212
D.2.2. Introducción de nuevo texto . . . . .	212
D.2.3. Movimientos del cursor . . . . .	214
D.2.4. Posicionamiento del cursor sobre palabras . . . . .	214
D.2.5. Deshacer . . . . .	214
D.2.6. Adiciones, cambios y supresiones simples de texto . . . . .	214
D.2.7. Búsquedas . . . . .	215
D.2.8. Opciones del editor . . . . .	215
D.3. Depurador gdb . . . . .	215
D.3.1. Ejemplo de depuración . . . . .	216
D.4. Depurador Valgrind . . . . .	223
D.4.1. Memcheck . . . . .	224
D.5. Compresor/descompresor tar . . . . .	227
<b>E. Código de las clases</b>	<b>229</b>
E.1. La clase TCoordenada . . . . .	229
E.2. La clase TLinea . . . . .	235
E.3. La clase TVector . . . . .	237
E.4. La clase TCalendario . . . . .	242
<b>Bibliografía recomendada</b>	<b>249</b>
<b>Índice alfabético</b>	<b>253</b>

# Índice de cuadros

2.1. Especificadores de acceso . . . . .	14
B.1. Nivel de precedencia 1 . . . . .	197
B.2. Nivel de precedencia 2 . . . . .	198
B.3. Nivel de precedencia 3 . . . . .	198
B.4. Nivel de precedencia 4 . . . . .	198
B.5. Nivel de precedencia 5 . . . . .	198
B.6. Nivel de precedencia 6 . . . . .	198
B.7. Nivel de precedencia 7 . . . . .	198
B.8. Nivel de precedencia 8 . . . . .	199
B.9. Nivel de precedencia 9 . . . . .	199
B.10. Nivel de precedencia 10 . . . . .	199
B.11. Nivel de precedencia 11 . . . . .	199
B.12. Nivel de precedencia 12 . . . . .	199
B.13. Nivel de precedencia 13 . . . . .	199
B.14. Nivel de precedencia 14 . . . . .	199
B.15. Nivel de precedencia 15 . . . . .	200
B.16. Nivel de precedencia 16 . . . . .	200

# Índice de figuras

3.1. Situación de error por no proporcionar un constructor de copia adecuado	38
4.1. Inclusión de la misma definición de clase dos veces	62
D.1. Página web del editor JOE	209
D.2. JOE: ficha de ayuda 1	210
D.3. JOE: ficha de ayuda 2	210
D.4. JOE: ficha de ayuda 3	210
D.5. JOE: ficha de ayuda 4	211
D.6. JOE: ficha de ayuda 5	211
D.7. JOE: ficha de ayuda 6	211
D.8. Página web del editor vim	213
D.9. Principales opciones del editor vim	216
D.10. Página web de gdb	217
D.11. Ejemplo de sesión de depuración con gdb	219
D.12. Página web de Valgrind	224

# Capítulo 1

## Introducción

En este capítulo se realiza una pequeña introducción del libro, se explican sus objetivos y se presenta el contenido de cada uno de los capítulos. Además, también se comentan las convenciones tipográficas empleadas.

### Índice General

---

1.1. Introducción . . . . .	1
1.2. Ventajas de C++ . . . . .	2
1.3. Objetivos de este libro . . . . .	2
1.4. Contenido de los capítulos . . . . .	3
1.5. Sistema operativo y compilador . . . . .	5
1.6. Convenciones tipográficas . . . . .	5

---

### 1.1. Introducción

El lenguaje de programación C++ es uno de los más empleados en la actualidad. Se puede decir que C++ es un lenguaje híbrido, ya que permite programar tanto en estilo procedimental (como si fuese C), como en estilo orientado a objetos, como en ambos a la vez. Además, también se puede emplear mediante programación basada en eventos para crear programas que usen interfaz gráfico de usuario.

El nacimiento de C++ se sitúa en el año 1980, cuando Bjarne Stroustrup, de los laboratorios Bell, desarrolló una extensión de C llamada "C with Classes" que permitía aplicar los conceptos de la programación orientada a objetos con el lenguaje C. Stroustrup se basó en las características de orientación a objetos del lenguaje de

programación Simula, aunque también tomó ideas de otros lenguajes importantes de la época como ALGOL68 o ADA.

Durante los siguientes años, Stroustrup continuó el desarrollo del nuevo lenguaje y en 1983 se acuñó el término C++.

En 1985, Bjarne Stroustrup publicó la primera versión de “The C++ Programming Language” (Addison-Wesley, 1985), que a partir de entonces se convirtió en el libro de referencia del lenguaje.

En la actualidad, este lenguaje se encuentra estandarizado a nivel internacional con el estándar ISO/IEC 14882:1998 con el título “Information Technology - Programming Languages - C++”, publicado el 1 de septiembre de 1998. En el año 2003 se publicó una versión corregida del estándar (ISO/IEC 14882:2003). Finalmente, en la actualidad se está preparando una nueva versión del lenguaje, llamada “C++0X”, que se espera que esté preparada para el año 2010.

## 1.2. Ventajas de C++

Las principales ventajas que presenta el lenguaje C++ son:

- Difusión: al ser uno de los lenguajes más empleados en la actualidad, posee un gran número de usuarios y existe una gran cantidad de libros, cursos, páginas web, etc. dedicados a él.
- Versatilidad: C++ es un lenguaje de propósito general, por lo que se puede emplear para resolver cualquier tipo de problema.
- Portabilidad: el lenguaje está estandarizado y un mismo código fuente se puede compilar en diversas plataformas.
- Eficiencia: C++ es uno de los lenguajes más rápidos en cuanto ejecución.
- Herramientas: existe una gran cantidad de compiladores, depuradores, librerías, etc.

## 1.3. Objetivos de este libro

Este libro no es, ni mucho menos, una referencia del lenguaje C++. Tampoco es un libro sobre programación orientada a objetos. Entonces, ¿qué es? El objetivo de este libro es proporcionar un material de aprendizaje breve y sencillo, donde se presenten los conceptos básicos del lenguaje C++ relacionados con la orientación a objetos conforme se vayan necesitando y paso por paso. Siguiendo el principio de Pareto de que “el 20 % de algo es el responsable del 80 % del resultado”, en este libro sólo se explica una pequeña parte de C++, pero que se puede considerar que es la más importante.

El libro está estructurado como soporte de un curso de 10 horas de duración de introducción al lenguaje C++. Los capítulos principales, del 2 al 6, constituyen las 5 sesiones del curso, con una duración de 2 horas por sesión. Todas las explicaciones van acompañadas de ejemplos, seguidos de ejecuciones que muestran la entrada/salida del ejemplo para afianzar los conceptos. Es aconsejable que el lector lea este libro delante del ordenador, pruebe los ejemplos y los modifique para comprender mejor su funcionamiento. Además, al final de cada capítulo se proponen ejercicios de autoevaluación y de programación, todos ellos con sus correspondientes soluciones.

En este libro, suponemos que los lectores (alumnos del curso) poseen unos conocimientos mínimos sobre programación en general. Además, también suponemos que los alumnos conocen la sintaxis básica de C, C++, Java o JavaScript, por lo que este libro no explica las sentencias básicas del lenguaje (sentencias condicionales, bucles, etc.) o los diferentes tipos de datos que se pueden emplear (entero, carácter, etc.). De todos modos, en los apéndices del libro se incluyen las palabras clave, los operadores y las sentencias del lenguaje C++.

La principal aportación de este libro, frente a otros libros similares, es que en este libro hemos querido reflejar los problemas a los que se enfrenta un lector cuando aprende un lenguaje de programación nuevo. La mayoría de los libros suponen que el lector no va a cometer errores, por lo que no hacen ninguna referencia a los posibles problemas de compilación del código o de comprensión de los conceptos explicados. Sin embargo, en este libro hemos optado por incluir algunos ejemplos con errores para mostrar los mensajes que genera el compilador. Además, en algunos apartados hemos dejado planteadas preguntas para que el lector pruebe y descubra la respuesta por sí mismo.

Por último, este libro posee un capítulo dedicado en su totalidad a los errores más comunes que se cometan al programar con C++. Este capítulo, que es toda una novedad frente a otros libros similares, se puede emplear de dos formas: como material de aprendizaje o como referencia para buscar la solución frente a un error que se resiste a nuestros intentos por solucionarlo.

## 1.4. Contenido de los capítulos

Este libro se compone de 9 capítulos y 5 apéndices, además de varios índices (cuadros, figuras, etc.) que facilitan la búsqueda de información y la bibliografía recomendada.

El el Capítulo 2 (**Clases y objetos**) se introducen los conceptos de clase y objeto, los elementos principales de la programación orientada a objetos. Se explica cómo declarar una clase y cómo crear objetos a partir de ella. Además, se explica la separación de la interfaz y la implementación, lo que posibilita estructurar el código en varios ficheros para permitir la compilación separada. Por último, se comentan algunos aspectos relacionados con los ficheros de encabezado y los espacios de nombres.

En el Capítulo 3 (**Constructor y destructor**) se introducen los conceptos de constructor y destructor, funciones miembro especiales de una clase que se invocan automáticamente al crear o eliminar un objeto. Como en C++ una clase puede tener varios constructores con el mismo nombre de función, primero se explica la sobrecarga de funciones, que permite crear varias funciones con el mismo nombre.

En el Capítulo 4 (**Funciones y clases amigas y reserva de memoria**) se presenta el concepto de amistad entre funciones y clases. Aunque el uso de funciones y clases amigas va en contra del principio de ocultación de la información, puede ser apropiado su uso cuando no haya otra solución, la solución posible sea demasiado compleja o tenga un impacto negativo en el rendimiento del programa. Además, se explica la reserva y eliminación de memoria dinámica y la compilación condicional que facilita el proceso de depuración.

En el Capítulo 5 (**Sobrecarga de operadores**) se explica cómo se pueden redefinir (sobrecargar) los operadores del lenguaje C++ para que funcionen correctamente con las clases creadas por el usuario. Para ello, antes hace falta explicar el puntero `this`, el modificador `const` y el paso por referencia.

En el Capítulo 6 (**Composición y herencia**) se introducen dos tipos de relaciones entre objetos: la relación “tiene-un” (composición) y “es-un” (herencia). Estos dos mecanismos básicos de la programación orientada a objetos permiten la reutilización de código, con las ventajas que ello conlleva.

En el Capítulo 7 (**Otros temas**) se comentan una serie de características del lenguaje de programación C++ que puntualizan algunos aspectos que se han presentando en los capítulos anteriores: la forma canónica de una clase, las funciones de cero parámetros, los valores por omisión de una función y las funciones `inline`.

En el Capítulo 8 (**Errores más comunes**) se recogen los errores más habituales que comete una persona cuando comienza a programar con el lenguaje de programación C++.

En el Capítulo 9 (**Ejercicios**) se proponen una serie de ejercicios complementarios a los propuestos a lo largo del libro.

Además, el libro también posee una serie de apéndices que complementan la información tratada a lo largo de los capítulos. En concreto, en el Apéndice A (**Palabras clave**) se listan y describen brevemente las palabras clave del lenguaje C++.

En el Apéndice B (**Operadores**) se explican los diferentes operadores del lenguaje C++, con su precedencia y asociatividad.

En el Apéndice C (**Sentencias**) se incluye un resumen de la sintaxis de las sentencias del lenguaje C++. El objetivo de este apéndice es que sirva como una guía rápida de búsqueda en caso de duda.

En el Apéndice D (**Herramientas**) se comentan algunas de las herramientas que pueden ayudar a la hora de programar con el lenguaje C++: el editor JOE, el editor vim, el depurador gdb, el depurador Valgrind y el compresor/descompresor tar.

En el Apéndice E (**Código de las clases**) se incluye el código completo de las cuatro clases desarrolladas a lo largo del libro: TCoordenada, TLinea, TVector y

TFecha.

Finalmente, este libro termina con la (**Bibliografía recomendada**), donde se incluye una serie de libros sobre el lenguaje C++. Además, también se incluyen algunos enlaces a páginas de Internet donde se puede encontrar más información sobre el lenguaje C++ o sobre herramientas de programación.

## 1.5. Sistema operativo y compilador

Todo el código que se muestra en este libro ha sido compilado en Linux con el compilador g++ de GNU versión 3.3.2. Los mensajes de error que se recogen en este documento han sido generados por dicho compilador y pueden variar entre distintas versiones y, evidentemente, entre distintos compiladores.

Los ejemplos de código están basados en C++ estándar, por lo que se pueden compilar sin problemas con otros compiladores que cumplan el estándar.

## 1.6. Convenciones tipográficas

Con el fin de mejorar la legibilidad del texto, distintas convenciones tipográficas se han empleado a lo largo de todo el libro.

Los ejemplos, que normalmente están completos y por tanto se pueden escribir y probar, aparecen destacados y numerados dentro de una caja de la siguiente forma (el texto de los ejemplos emplea un tipo de letra de paso fijo como Courier):

---

Ejemplo 1.1

---

```
1 #include <iostream>
2
3 using namespace std;
4
5 int
6 main(void)
7 {
8     int i;
9     TCoordenada p1;
10
11    return 0;
12 }
```

---

Los números de línea permiten hacer referencia a una línea concreta del código.

Los ejemplos parciales, que por sí solos no se pueden compilar y que normalmente complementan o corrigen un ejemplo completo que ha aparecido anteriormente, también aparecen destacados y numerados de la siguiente forma:

```
1 class NombreClase {  
2     // Contenido de la clase  
3 };
```

También se ha empleado la notación anterior para indicar las ordenes que se pueden ejecutar desde la línea de comandos del sistema operativo. Por ejemplo:

```
1 | g++ -o ejem1 ejem1.cc
```

La salida que genera un código de ejemplo o un programa como el compilador cuando un código presenta errores, se muestra destacada con el siguiente formato:

Salida ejemplo 1.1

```
1 ejem1.cc: In function ‘int main()’:  
2 ejem1.cc:9: ‘TCoordenada’ undeclared (first use this function)  
3 ejem1.cc:9: (Each undeclared identifier is reported only once for  
4     each function it appears in.)  
5 ejem1.cc:9: syntax error before ‘;’ token
```

El título de la salida hace referencia al ejemplo que lo produce.

Por último, los estilos empleados a lo largo del texto son:

- Los nombres de los programas se muestran con un tipo de letra sin palo (*sans serif*). Ejemplo: *Linux*, *g++*, *gdb*, etc.
- Las palabras no escritas en español aparecen destacadas en *cursiva*. Ejemplo: *layering*, *link*, etc.
- Las extensiones de los ficheros, las palabras clave de los lenguajes de programación y el código incluido dentro del texto se muestra con un tipo de letra de *paso fijo* como *Courier*. Ejemplo: *.cc*, *g++*, *int a = 10;*, etc.
- Los términos importantes, cuando aparecen por primera, se destacan con un tipo de letra **sin palo (sans serif) y en negrita**. Ejemplo: **clase, ocultación de información, operador de resolución de alcance**, etc.

# Capítulo 2

## Clases y objetos

En este capítulo se introducen los conceptos de clase y objeto, los elementos principales de la programación orientada a objetos. Se explica cómo declarar una clase y cómo crear objetos a partir de ella. Además, se explica la separación de la interfaz y la implementación, lo que posibilita estructurar el código en varios ficheros para permitir la compilación separada. Por último, se comentan algunos aspectos relacionados con los ficheros de encabezado y los espacios de nombres.

### Índice General

---

2.1. Introducción . . . . .	8
2.2. Declaración de una clase . . . . .	10
2.3. Acceso a los miembros de una clase . . . . .	11
2.4. Control de acceso . . . . .	12
2.5. Visualización de un objeto . . . . .	16
2.6. Empleo de punteros . . . . .	18
2.7. Separación de la interfaz y la implementación . . . . .	19
2.8. La herramienta make . . . . .	23
2.9. Ficheros de encabezado . . . . .	24
2.10. Uso de espacios de nombres . . . . .	25
2.11. Ejercicios de autoevaluación . . . . .	26
2.12. Ejercicios de programación . . . . .	28
2.12.1. Clase TVector . . . . .	28
2.12.2. Clase TCalendario . . . . .	28
2.13. Respuesta a los ejercicios de autoevaluación . . . . .	28
2.14. Respuesta a los ejercicios de programación . . . . .	30

---

2.14.1. Clase TVector . . . . .	30
2.14.2. Clase TCalendario . . . . .	30

---

## 2.1. Introducción

La programación orientada a objetos se basa en encapsular datos (atributos) y funciones o métodos (comportamientos) juntos en estructuras llamadas **clases**. Las clases se emplean para modelar objetos del mundo real.

Una clase se puede **instanciar** para crear diversos **objetos**. Los objetos se declaran como las variables de los tipos básicos del lenguaje C++ (`int`, `float`, `bool`, etc.). Por ejemplo, suponiendo que existe una clase llamada `TCoordenada`, las declaraciones:

```

1 // Declaración de un objeto
2 TCoordenada objetoP1;
3 // Declaración de un array
4 TCoordenada arrayP2[10];
5 // Declaración de un puntero
6 TCoordenada *ptrP3 = &objetoP1;
7 // Declaración de una referencia
8 TCoordenada &refP4 = objetoP1;
```

declaran `objetoP1` como una variable de tipo `TCoordenada`, `arrayP2` como un array de 10 elementos de tipo `TCoordenada`, `ptrP3` como un puntero a un objeto de tipo `TCoordenada` y `refP4` como una referencia a un objeto de tipo `TCoordenada`.

Por ejemplo, el siguiente código incluye la declaración de un objeto llamado `p1` que es una instancia de la clase `TCoordenada` que representa coordenadas o puntos en el espacio:

---

Ejemplo 2.1

---

```

1 #include <iostream>
2
3 using namespace std;
4
5 int
6 main(void)
7 {
8     int i;
9     TCoordenada p1;
10
11     return 0;
12 }
```

---

En este ejemplo, la instrucción `#include <iostream>` incluye el contenido del archivo de encabezado<sup>1</sup> de flujo de entrada/salida (*input output stream*) en el código del programa. La sentencia `using namespace std;` permite el acceso a todos los miembros definidos en el espacio de nombres<sup>2</sup> `std` que se emplea en el archivo `iostream`.

En C++, el **código fuente** de los programas suele tener las extensiones `.cpp`, `.cxx`, `.cc` o `.c`, según el compilador que se emplee. El código fuente en C++ se tiene que traducir a **código máquina (código objeto)** para que se pueda ejecutar. Para ello, se tiene que **compilar** el código fuente mediante una serie de programas que realizan distintas funciones:

1. Preprocesador.
2. Compilador.
3. Enlazador.

En el entorno Linux, se suele emplear la extensión `.cc` para los ficheros con el código fuente y para compilar se emplea el programa `g++`. Si suponemos que el código del ejemplo anterior está almacenado en un archivo llamado `ejem1.cc`, el comando para compilarlo es<sup>3</sup>:

```
1 | g++ -o ejem1 ejem1.cc
```

que debería generar un fichero ejecutable llamado `ejem1`. Sin embargo, si se compila el código anterior, se producen los siguientes mensajes de error:

■ Salida ejemplo 2.1 ■

```
1 ejem1.cc: In function ‘int main()’:
2 ejem1.cc:9: ‘TCoordenada’ undeclared (first use this function)
3 ejem1.cc:9: (Each undeclared identifier is reported only once for
4     each function it appears in.)
5 ejem1.cc:9: syntax error before ‘;’ token
```

¿Qué está pasando? El compilador no reconoce `TCoordenada` como algo válido del lenguaje. Aunque el código sea correcto desde un punto de vista sintáctico, desde un punto de vista semántico no es correcto porque `TCoordenada` no tiene significado: falta la declaración de la clase `TCoordenada`.

<sup>1</sup>Para más información sobre archivos de encabezado, consultar la Sección 2.9.

<sup>2</sup>Para más información sobre espacios de nombres, consultar la Sección 2.10.

<sup>3</sup>Para más información sobre la compilación, consultar la Sección 2.8.

## 2.2. Declaración de una clase

En C++, se emplea la palabra reservada `class`<sup>4</sup> para crear una clase. La construcción

```
1 class NombreClase {
2     // Contenido de la clase
3 };
```

se denomina **definición de clase** o **declaración de clase**. Una clase se compone de datos y funciones miembro, que se conocen en general como **miembros** de la clase. Por ejemplo, el siguiente código declara la clase `TCoordenada` que contiene tres datos miembro de tipo entero que representan las coordenadas (x, y, z) de un punto en el espacio:

```
1 class TCoordenada {
2     int x, y, z;
3 };
```

Una vez que se define una clase, el nombre de la clase se vuelve un nombre de un tipo nuevo y se puede emplear para declarar variables que representan objetos de dicha clase. Por ejemplo, si tomamos el código del apartado anterior que producía un error y le añadimos la declaración que acabamos de ver, el código compilará sin problemas:

---

### Ejemplo 2.2

---

```
1 #include <iostream>
2
3 using namespace std;
4
5 class TCoordenada {
6     int x, y, z;
7 };
8
9 int
10 main(void)
11 {
12     int i;
13     TCoordenada p1;
14
15     return 0;
16 }
```

---

El código anterior se compila sin errores. Si se ejecuta, no produce resultados visibles. ¿Cómo podemos acceder a los miembros (datos y funciones) de una clase?

<sup>4</sup>También se puede emplear `struct`, aunque más adelante se verá que existe una diferencia en el control de acceso a los datos o funciones miembro.

## 2.3. Acceso a los miembros de una clase

El acceso a los miembros de una clase se realiza mediante los **operadores de acceso a miembros**: el **operador punto “.”** y el **operador flecha “->”** (el signo menos y el símbolo mayor que, sin espacio intermedio).

El operador punto accede a un miembro de una clase mediante el nombre de la variable del objeto o mediante una referencia al objeto. Por ejemplo, para mostrar por pantalla los datos que contiene un objeto **TCoordenada**, se puede emplear:

```
1 TCoordenada p1;
2
3 cout << "Componente x: " << p1.x << endl;
4 cout << "Componente y: " << p1.y << endl;
5 cout << "Componente z: " << p1.z << endl;
```

El operador flecha accede a un miembro de una clase mediante un puntero al objeto. Por ejemplo, para mostrar por pantalla los datos del objeto **p1** mediante el puntero **ptr1**:

```
1 TCoordenada p1;
2 TCoordenada *ptr1 = &p1;
3
4 cout << "Componente x: " << ptr1->x << endl;
5 cout << "Componente y: " << ptr1->y << endl;
6 cout << "Componente z: " << ptr1->z << endl;
```

Por ejemplo, el siguiente código crea un objeto llamado **p1** a partir de la clase **TCoordenada** e inicializa sus datos a los valores 1, 2 y 3:

Ejemplo 2.3

```
1 #include <iostream>
2
3 using namespace std;
4
5 class TCoordenada {
6     int x, y, z;
7 };
8
9 int
10 main(void)
11 {
12     int i;
13     TCoordenada p1;
14
15     p1.x = 1;
16     p1.y = 2;
```

```

17     p1.z = 3;
18
19     return 0;
20 }
```

Si se compila el código anterior, se producen los siguientes mensajes de error:

■ Salida ejemplo 2.3 ■

```

1 ejem3.cc: In function 'int main()':
2 ejem3.cc:6: 'int TCoordenada::x' is private
3 ejem3.cc:15: within this context
4 ejem3.cc:6: 'int TCoordenada::y' is private
5 ejem3.cc:16: within this context
6 ejem3.cc:6: 'int TCoordenada::z' is private
7 ejem3.cc:17: within this context
```

El compilador nos informa de que los datos `x`, `y` y `z` de la clase `TCoordenada` no son accesibles desde el contexto de la función `main()`. ¿Qué está pasando? ¿Qué significa `is private`?

## 2.4. Control de acceso

Una de las características básicas de la programación orientada a objetos es la **ocultación de información**. La idea es ocultar (impedir el acceso) a la implementación de una clase (básicamente, los datos que se utilizan en una clase) a los posibles usuarios de una clase. ¿Para qué? De este modo, los usuarios de la clase la pueden utilizar y obtener los mismos resultados sin darse cuenta de que la implementación ha cambiado.

¿Por qué puede cambiar la implementación de una clase? Por diversas razones. Por ejemplo, una clase que represente un hora, la puede almacenar internamente como el número de segundos transcurridos a partir de medianoche, como una cadena donde se almacena la hora con un formato textual, como el número de *beats* de la hora de Internet<sup>5</sup> o como horas, minutos y segundos almacenados por separado. Por ejemplo, en el siguiente código aparecen tres posibles implementaciones de la clase hora y cómo se almacenarían las 12 horas, 15 minutos y 20 segundos:

```

1 // nSegundos = 920
2 class THora {
3     int nSegundos;
4 };
5 // hora = "00:15:20"
```

<sup>5</sup>Véase <http://www.swatch.com/internettime/>.

```

7  class THora {
8      char *hora;
9  };
10
11 // beat = 7.1
12 class THora {
13     float beat;
14 };
15
16 // horas = 0, minutos = 15, segundos = 20
17 class THora {
18     int horas, minutos, segundos;
19 };

```

En C++, el modo de acceso predeterminado a los miembros de una clase se llama **privado** (*private*). Sólo se puede acceder a la parte privada de una clase mediante funciones miembros y funciones y clases amigas<sup>6</sup> de dicha clase. Por otro lado, en C++, una estructura (**struct**) es una clase (**class**) cuyo modo de acceso es **público** por defecto.

El modo de acceso, tanto de una clase como de una estructura, se puede modificar mediante las etiquetas **public:**, **private:** y **protected:** que se denominan **especificadores de acceso a miembros**. El modo de acceso predeterminado para una clase es **private:**, por tanto, todos los miembros después del inicio de la declaración de la clase y antes del primer especificador de acceso a miembros son privados. A continuación de cada especificador de acceso a miembros, se mantiene el modo especificado hasta el siguiente especificador de acceso o hasta que finalice la declaración de la clase con la llave de cierre. Se pueden mezclar los especificadores de acceso a miembros, pero no es lo común, ya que puede resultar confuso. En el Cuadro 2.1 se resume el significado de los tres modos de acceso. El siguiente código muestra la estructura básica de una clase en C++:

```

1  class UnaClase {
2      public:
3          // Parte pública
4          // Normalmente, sólo funciones
5          .
6      protected:
7          // Parte protegida
8          // Funciones y datos
9
10     private:
11         // Parte privada
12         // Normalmente, datos y funciones auxiliares

```

---

<sup>6</sup>Las funciones y clases amigas se explican en el Capítulo 4.

	Descripción
<b>public:</b>	Accesible tanto desde la propia clase como desde funciones ajenas a la clase
<b>private:</b>	Accesible exclusivamente desde las funciones miembros y funciones y clases amigas
<b>protected:</b>	Se emplea para limitar el acceso a las clases derivadas, su funcionamiento depende del tipo de herencia que se realice

Cuadro 2.1: Especificadores de acceso

13  
14     };

Por tanto, para poder acceder directamente a los datos de una clase desde la función `main()` se tiene que incluir el especificador de acceso a miembros `public:`, tal como se muestra en el siguiente ejemplo:

## Ejemplo 2.4

```

1 #include <iostream>
2
3 using namespace std;
4
5 class TCoordenada {
6     public:
7         int x, y, z;
8 };
9
10 int
11 main(void)
12 {
13     int i;
14     TCoordenada p1;
15
16     p1.x = 1;
17     p1.y = 2;
18     p1.z = 3;
19
20     cout << "(" << p1.x << ", " << p1.y << ", " << p1.z << ")" << endl;
```

```
21  
22     return 0;  
23 }
```

El código anterior produce como salida:

Salida ejemplo 2.4

```
1 (1, 2, 3)
```

Sin embargo, como se ha dicho al principio de este apartado, una de las características básicas de la programación orientada a objetos es la ocultación de información. ¿Para qué? Para limitar su empleo a lo estrictamente deseado por el programador de la clase. Por ejemplo, en la clase **TCoordenada** del ejemplo anterior, el programador puede desear que las componentes no tomen valores negativos. Por tanto, los datos del ejemplo anterior deberían de declararse como privado. En ese caso, ¿cómo se puede acceder a ellos?

La solución es proporcionar una serie de funciones que permitan acceder a los datos de una clase, para su inicialización, consulta y modificación. Estas funciones normalmente llevan los prefijos **set** o **get** según se empleen para inicializar/modificar o consultar (leer) el valor de los datos, aunque no necesitan que específicamente se les llame así. Aunque puede parecer que emplear las funciones **set** y **get** es lo mismo que hacer públicos los datos miembro, el empleo de estas funciones permite controlar su utilización.

En el siguiente ejemplo, se han añadido las funciones **setX()**, **setY()** y **setZ()** para inicializar y modificar los datos de la clase y las funciones **getX()**, **getY()** y **getZ()** para recuperar los datos:

Ejemplo 2.5

```
1 #include <iostream>  
2  
3 using namespace std;  
4  
5 class TCoordenada {  
6     public:  
7         void setX(int xx) {x = xx;}  
8         void setY(int yy) {y = yy;}  
9         void setZ(int zz) {z = zz;}  
10  
11         int getX(void) {return x;}  
12         int getY(void) {return y;}  
13         int getZ(void) {return z;}  
14  
15     private:
```

```

16     int x, y, z;
17 }
18
19 int
20 main(void)
21 {
22     int i;
23     TCoordenada p1;
24
25     p1.setX(1);
26     p1.setY(2);
27     p1.setZ(3);
28
29     cout << "(" << p1.getX();
30     cout << ", " << p1.getY();
31     cout << ", " << p1.getZ();
32     cout << ")" << endl;
33
34     return 0;
35 }
```

---

Cuando el código de una función miembro se incluye en la propia declaración de la clase, la función se considera `inline`<sup>7</sup>. Las funciones `getX()`, `getY()` y `getZ()` se denominan de cero parámetros<sup>8</sup> porque no reciben argumentos.

El código anterior produce otra vez como salida:

— Salida ejemplo 2.5 —

1 (1, 2, 3)

## 2.5. Visualización de un objeto

El lenguaje C++ proporciona un extenso conjunto de funciones de entrada/salida. C++ es capaz de visualizar los tipos básicos (`int`, `float`, etc.). Sin embargo, no es capaz de visualizar los tipos creados por el usuario, como por ejemplo un objeto de una clase.

Hasta ahora, para visualizar el contenido de un objeto `TCoordenada` se han empleado instrucciones como el siguiente fragmento de código:

```

1 | cout << "(" << p1.getX();
2 | cout << ", " << p1.getY();
```

<sup>7</sup>Para más información sobre las funciones `inline`, consultar la Sección 7.4.

<sup>8</sup>Para más información sobre las funciones de cero parámetros, consultar la Sección 7.2.

```
3 |     cout << ", " << p1.getZ();  
4 |     cout << ")" << endl;
```

Sin embargo, C++ permite que el usuario pueda definir como realizar la entrada/salida para los objetos definidos por él mismo. Esta característica, que se conoce como sobrecarga de los operadores de entrada/salida, se explicará en la Sección 5.15. Hasta entonces, podemos añadir a la clase **TCoordenada** una función miembro, que hemos llamado **Imprimir()**, que muestra por la salida estándar (**cout**) el contenido del objeto sobre la que se invoca:

---

#### Ejemplo 2.6

---

```
1 #include <iostream>  
2  
3 using namespace std;  
4  
5 class TCoordenada {  
6     public:  
7         void setX(int xx) {x = xx;}  
8         void setY(int yy) {y = yy;}  
9         void setZ(int zz) {z = zz;}  
10  
11        int getX(void) {return x;}  
12        int getY(void) {return y;}  
13        int getZ(void) {return z;}  
14  
15        void Imprimir(void) {  
16            cout << "(" << x;  
17            cout << ", " << y;  
18            cout << ", " << z;  
19            cout << ")";  
20        }  
21  
22    private:  
23        int x, y, z;  
24 };  
25  
26 int  
27 main(void)  
28 {  
29     int i;  
30     TCoordenada p1, p2;  
31  
32     p1.setX(1);  
33     p1.setY(2);  
34     p1.setZ(3);  
35
```

```

36     p2.setX(4);
37     p2.setY(5);
38     p2.setZ(6);

39
40     p1.Imprimir();
41     cout << endl;
42     p2.Imprimir();
43     cout << endl;

44
45     return 0;
46 }
```

El código anterior produce como salida:

■ Salida ejemplo 2.6 ■

```

1   (1, 2, 3)
2   (4, 5, 6)
```

## 2.6. Empleo de punteros

Como se ha comentado en la Sección 2.3, el acceso a los miembros de una clase se realiza mediante el operador punto “.” y el operador flecha “->”.

En el caso de que la variable sea un puntero a un objeto, se puede emplear cualquiera de los operadores. Por ejemplo:

```

1 TCoordenada p1;
2 TCoordenada *ptr1 = &p1;
3
4 ptr1->x = 1;
5 (*ptr1).x = 1;
```

Las expresiones `ptr1->x = 1` y `(*ptr1).x = 1` son equivalentes. En el segundo caso, se emplea el **operador de desreferencia \*** para obtener el objeto al que apunta el puntero y se accede al miembro `x` mediante el operador punto. Los paréntesis son necesarios, ya que el operador punto tiene un nivel de precedencia<sup>9</sup> más alto que el operador de desreferencia del puntero. La omisión de los paréntesis produce que la expresión se evalúe como si los paréntesis estuviesen como `*(ptr1.x)`, que sería un error de sintaxis.

Por ejemplo, el siguiente código produce un error al ser compilado:

<sup>9</sup>Para más información sobre los operadores y los niveles de precedencia, consultar el Apéndice B.

## Ejemplo 2.7

```
1 #include <iostream>
2
3 using namespace std;
4
5 class TCoordenada {
6     public:
7         int x, y, z;
8 };
9
10 int
11 main(void)
12 {
13     int i;
14     TCoordenada p1;
15     TCoordenada *ptr1 = &p1;
16
17     ptr1->x = 1;
18     *ptr1.x = 1;
19
20     return 0;
21 }
```

Si se compila el código anterior, se produce el siguiente mensaje de error:

## Salida ejemplo 2.7

```
1 ejem6.cc: In function ‘int main()’:
2 ejem6.cc:18: request for member ‘x’ in ‘ptr1’, which is of
3     non-aggregate type ‘TCoordenada*’
```

El error de compilación se resuelve modificando la línea 18:

```
1 |     (*ptr1).x = 1;
```

## 2.7. Separación de la interfaz y la implementación

La separación de la **interfaz** (el aspecto de una clase) y su **implementación** facilitan el mantenimiento de la misma. De este modo, se puede proporcionar al usuario final de la clase la interfaz y la implementación por separado y modificar la implementación sin que afecte al usuario.

La declaración de una clase se debe colocar en un archivo de cabecera o encabezado (extensión .h), para que los usuarios lo incluyan en su código. El código de las funciones miembro de la clase se debe de colocar en un archivo fuente (extensión .cc)

o .cpp según el compilador). A los usuarios de la clase no es necesario proporcionarles el archivo fuente, sino el código una vez compilado que genera un archivo objeto (extensión .o o .obj). Por tanto<sup>10</sup>:

- .h: Archivo de cabecera. Contiene la declaración de la clase: las estructuras de datos y los prototipos de las funciones. En este fichero es necesario incluir los ficheros de cabecera necesarios para el funcionamiento de la clase, como por ejemplo, `iostream`, `string`, `cmath`, etc.
- .cc: Archivo fuente. Contiene el código de cada uno de los métodos que aparecen en el fichero .h. En este fichero es necesario incluir el fichero de cabecera que contiene la declaración de la clase.
- .o: Archivo objeto. Este fichero se crea automáticamente al compilar el archivo fuente.

Los archivos de cabecera se incluyen en el código del usuario mediante sentencias `#include`. Los archivos de cabecera propios se encierran entre comillas ("") en lugar de los símbolos de menor y mayor que (<>). Normalmente, se tienen que situar los archivos de cabecera en el mismo directorio que los archivos que emplean los archivos de cabecera. Si se emplea (<>) el compilador asumirá que el archivo de cabecera es parte de la biblioteca estándar de C++ y no buscará el archivo en el directorio actual.

Por ejemplo, a continuación se incluye el fichero `tcoordenada.h`:

---

#### Ejemplo 2.8

---

```

1 #include <iostream>
2
3 using namespace std;
4
5 class TCoordenada {
6     public:
7         void setX(int);
8         void setY(int);
9         void setZ(int);
10
11        int getX(void);
12        int getY(void);
13        int getZ(void);
14
15        void Imprimir(void);
16
17    private:
18        int x, y, z;
19 };

```

---

<sup>10</sup>Extensiones usadas con g++ en Linux.

En la declaración de la clase hemos eliminado el código de las funciones miembro que contiene. A continuación mostramos el código del archivo fuente **tcoordenada.cc**; notar como en la primera línea del código se incluye el archivo de cabecera **tcoordenada.h**:

---

Ejemplo 2.9

---

```
1 #include "tcoordenada.h"
2
3 void
4 TCoordenada::setX(int xx) {
5     x = xx;
6 }
7
8 void
9 TCoordenada::setY(int yy) {
10    y = yy;
11 }
12
13 void
14 TCoordenada::setZ(int zz) {
15    z = zz;
16 }
17
18 int
19 TCoordenada::getX(void) {
20     return x;
21 }
22
23 int
24 TCoordenada::getY(void) {
25     return y;
26 }
27
28 int
29 TCoordenada::getZ(void) {
30     return z;
31 }
32
33 void
34 TCoordenada::Imprimir(void) {
35     cout << "(" << x;
36     cout << ", " << y;
37     cout << ", " << z;
38     cout << ")";
39 }
```

---

Cuando se definen las funciones miembro de una clase fuera de ésta, el nombre

de la función es precedido por el nombre de la clase y el **operador de resolución de alcance**<sup>11</sup> u **operador de ámbito** “`::`”. De este modo, se identifica de manera única a las funciones de una clase en particular.

Una vez que se ha separado la interfaz de la implementación, se necesita crear un archivo principal que contenga el código de la función `main()` que emplee la clase. En este fichero se tiene que incluir el archivo de cabecera de la clase para poderla usar. Por ejemplo, a continuación se incluye el código de un fichero llamado `main.cc`:

---

Ejemplo 2.10

---

```

1 #include <iostream>
2
3 using namespace std;
4
5 #include "tcoordenada.h"
6
7 int
8 main(void)
9 {
10     int i;
11     TCoordenada p1;
12
13     p1.setX(1);
14     p1.setY(2);
15     p1.setZ(3);
16
17     p1.Imprimir();
18     cout << endl;
19
20     return 0;
21 }
```

---

Para compilar el ejemplo anterior, es necesario ejecutar los siguientes comandos:

```

1 g++ -c tcoordenada.cc
2 g++ -c main.cc
3 g++ -o main main.o tcoordenada.o
```

Mucho trabajo para tan poca cosa. ¿Qué ocurre en un proyecto real con cientos o miles de ficheros? ¿Cómo nos acordamos de la forma de compilar un proyecto? Para facilitar esta labor, existe la herramienta `make`, que automatiza el proceso de compilación.

---

<sup>11</sup>El operador de resolución de alcance también se emplea con los espacios de nombres (ver Sección 2.10) y para acceder a los miembros estáticos de una clase (información propia de una clase que es compartida por todos los objetos creados a partir de la clase).

## 2.8. La herramienta make

El propósito de la herramienta **make** es determinar de forma automática qué trozos de un programa necesitan recompilarse y qué comandos se tienen que ejecutar para ello. **make** recompila un fichero fuente (objetivo) si depende de una serie de ficheros (prerrequisitos) que se han modificado desde la última actualización o si el objetivo no existe. Las relaciones entre los distintos ficheros objetivo y los prerrequisitos se definen mediante reglas en un fichero **makefile**<sup>12</sup>. Un ejemplo de fichero **makefile** para compilar los ficheros anteriores es:

Ejemplo 2.11

```

1 main: main.o tcoordenada.o
2         g++ -o main main.o tcoordenada.o
3
4 main.o: main.cc tcoordenada.h
5         g++ -c main.cc
6
7 tcoordenada.o: tcoordenada.h tcoordenada.cc
8         g++ -c tcoordenada.cc

```

En un fichero **makefile** se pueden declarar variables para representar aquellas partes que se repitan varias veces. Por ejemplo, el fichero anterior se podría escribir como:

Ejemplo 2.12

```

1 OBJ=main.o tcoordenada.o
2 COMP=g++
3 OPC=-g
4
5 main: $(OBJ)
6     $(COMP) $(OPC) $(OBJ) -o main
7
8 main.o: main.cc tcoordenada.h
9     $(COMP) $(OPC) -c main.cc
10
11 tcoordenada.o: tcoordenada.h tcoordenada.cc
12     $(COMP) $(OPC) -c tcoordenada.cc

```

En el fichero **makefile** anterior:

- **OBJ**, **COMP** y **OPC** son tres variables.

<sup>12</sup>Este fichero puede tener cualquier nombre, pero si no se emplea el nombre por defecto, se tiene que indicar a la herramienta **make** con el parámetro **-f**.

- La variable **OBJ** define una serie de ficheros objetivo.
- La variable **COMP** indica el compilador que se utiliza.
- La variable **OPC** con el valor **-g** indica el nivel de depuración: si no se desea incluir información de depuración, se puede anular dejando esta variable en blanco.
- **-c** compila los ficheros fuente, pero no los enlaza (*link*). Si no se indica un fichero de salida (con **-o**), el compilador automáticamente sustituye la extensión **.cc** por **.o**.
- **main, main.o** y **tcoordenada.o** son ficheros objetivo (aparecen a la izquierda de una regla).
- **main.cc, main.o, tcoordenada.h, tcoordenada.cc** y **tcoordenada.o** son ficheros prerequisito (aparecen a la derecha de una regla). Notar que hay dos ficheros, **main.o** y **tcoordenada.o**, que son a la vez fichero objetivo y fichero prerequisito.
- En el proceso de creación del ejecutable, si no se indica un nombre de fichero de salida con **-o**, automáticamente se crea el fichero **a.out**.
- Las líneas que aparecen separadas del margen izquierdo están separadas mediante tabuladores y no mediante espacios en blanco.

## 2.9. Ficheros de encabezado

En C++ se distinguen dos tipos de ficheros de encabezado: C y C++. Los de C no hacen uso de espacios de nombre y llevan el prefijo “c”. Por ejemplo, **cstdio**, **cstdlib** o **cstring**. Los ficheros de encabezado de C++ hacen uso de espacios de nombre (el espacio de nombres estándar **std**). En ambos casos, no llevan la extensión **.h**.

Temporalmente, es posible que estén disponibles los ficheros de cabecera antiguos (con la extensión **.h**), pero no se aconseja su uso. Por tanto, a partir de ahora hay que escribir:

```

1 | #include <iostream> // Para usar: cin, cout, ...
2 | #include <cstring> // Para usar: strcpy(), strcmp(), ...
3 | #include<string>   // Para usar la clase string

```

Hay que llevar cuidado y no confundir **string.h** (librería de C), **cstring** (librería de C adaptada para ser usada en C++) y **string** (librería de C++).

## 2.10. Uso de espacios de nombres

Cuando un proyecto alcanza un gran tamaño, con miles o millones de líneas, se pueden producir problemas de colisión de nombres (identificadores): variables globales, clases o funciones con el mismo nombre. La colisión de nombres también ocurre frecuentemente cuando se emplean librerías de terceras partes.

En algunos lenguajes de programación (C, Basic, etc.) no existe una solución (el programador la tiene que articular de algún modo). En C++ existen los **espacios de nombres**<sup>13</sup>, que permiten dividir el espacio general de nombres en subespacios distintos e independientes. Sin embargo, aún así puede persistir el problema de la colisión de nombres, ya que no existe ningún mecanismo para garantizar que dos espacios de nombres sean únicos.

El proceso consiste en declarar un espacio de nombres asignándole un identificador y delimitándolo por un bloque entre llaves. Dentro de este bloque pueden declararse los elementos correspondientes al mismo: variables, clases, funciones, etc. A diferencia de la declaración de una clase o estructura, un espacio de nombres no termina con punto y coma.

Los elementos declarados dentro de un espacio de nombres son accesibles de diversos modos:

- Mediante `espacioNombre::miembro` cada vez que se necesite, que utiliza el operador de resolución de alcance “`::`”. Por ejemplo:  
`std::cout << "Algo";`
- Mediante `using espacioNombre::miembro;` para permitir el acceso a un miembro individual de un espacio de nombres, sin permitir un acceso general a todos los miembros del espacio de nombres. Por ejemplo:  
`using std::cout;`
- Mediante `using namespace espacioNombre;`, que permite el acceso a todos los miembros del espacio de nombres. Por ejemplo:  
`using namespace std;`

Por ejemplo, en el siguiente código se declaran dos espacios de nombres y se muestra cómo se pueden emplear:

```
1  namespace Espacio1 {  
2      int a;  
3  }  
4  
5  namespace Espacio2 {  
6      int a;
```

<sup>13</sup>Se incorporó al lenguaje en julio de 1998, por lo que existen muchos desarrollos previos que no lo emplean.

```
7 }  
8  
9 using namespace Espacio2;  
10  
11 Espacio1::a = 3; // Hace falta indicar su namespace  
12 a = 5; // Se refiere a Espacio2::a
```

Las librerías estándar de C++ están definidas dentro de un espacio de nombres llamado `std`. Por tanto, si se quiere evitar el tener que emplear el operador de ámbito constantemente, hay que añadir la sentencia `using namespace std;` justo después de la inclusión (`#include`) de las librerías en un fichero.

## 2.11. Ejercicios de autoevaluación

1. La palabra clave `struct`:

- a) Introduce la definición de una estructura
- b) Introduce la definición de una clase
- c) Introduce la definición de una estructura o una clase
- d) No es una palabra clave de C++

2. Los miembros de una clase especificados como `private`:

- a) Sólo son accesibles por las funciones miembro de la clase
- b) Son accesibles por las funciones miembro de la clase y las funciones amigas de la clase
- c) Son accesibles por las funciones miembro de la clase, las funciones amigas de la clase y las clases que heredan
- d) Las anteriores respuestas no son correctas

3. El acceso predeterminado para los miembros de una clase es:

- a) `private`
- b) `public`
- c) `protected`
- d) No está definido

4. Si se tiene un puntero a un objeto, para acceder a los miembros de la clase se emplea:

- a) `“.”`
- b) `“->”`

- c) “&”
  - d) Las anteriores respuestas no son correctas
5. En el fichero .h de una clase se almacena:
- a) La declaración de la clase
  - b) El código de cada una de las funciones miembro de una clase
  - c) El programa principal de una clase
  - d) Las anteriores respuestas no son correctas
6. ¿Para qué sirve una clase?
- a) Para encapsular datos
  - b) Para modelar objetos del mundo real
  - c) Para simplificar la reutilización de código
  - d) Todas las respuestas son correctas
7. ¿Cuál no es un nivel de visibilidad en C++?
- a) protected
  - b) hidden
  - c) private
  - d) public
8. ¿Cuál es una declaración correcta de una clase?
- a) class A {int x;};
  - b) class B { }
  - c) public class A { }
  - d) object A {int x;};
9. Un espacio de nombres se emplea para:
- a) Definir una función miembro fuera de la definición de su clase.
  - b) Evitar la colisión de nombres de los identificadores (nombres de variables, funciones, etc.)
  - c) Lograr un aumento de la velocidad de ejecución del código.
  - d) Todas las respuestas son correctas.
10. El operador de ámbito se emplea para:
- a) Identificar una función miembro cuando se define fuera de su clase.
  - b) Acceder a un elemento definido en un espacio de nombres.
  - c) Para acceder a los miembros estáticos de una clase.
  - d) Todas las respuestas son correctas.

## 2.12. Ejercicios de programación

### 2.12.1. Clase TVector

Definid en C++ la clase **TVector** que contiene un vector dinámico de números enteros y un número entero que contiene la dimensión del vector.

### 2.12.2. Clase TCalendario

Definid en C++ la clase **TCalendario** que contiene una fecha (representada mediante tres variables enteras para el día, mes y año) y un mensaje (representado mediante un vector dinámico de caracteres).

## 2.13. Respuesta a los ejercicios de autoevaluación

1. La palabra clave struct:

- a) **(✓)** Introduce la definición de una estructura
- b) Introduce la definición de una clase
- c) Introduce la definición de una estructura o una clase
- d) No es una palabra clave de C++

2. Los miembros de una clase especificados como private:

- a) Sólo son accesibles por las funciones miembro de la clase
- b) **(✓)** Son accesibles por las funciones miembro de la clase y las funciones amigas de la clase
- c) Son accesibles por las funciones miembro de la clase, las funciones amigas de la clase y las clases que heredan
- d) Las anteriores respuestas no son correctas

3. El acceso predeterminado para los miembros de una clase es:

- a) **(✓)** **private**
- b) public
- c) protected
- d) No está definido

4. Si se tiene un puntero a un objeto, para acceder a los miembros de la clase se emplea:

- a) “.”

- b) (✓) “->”  
c) “&”  
d) Las anteriores respuestas no son correctas
5. En el fichero .h de una clase se almacena:
- a) (✓) La declaración de la clase  
b) El código de cada una de las funciones miembro de una clase  
c) El programa principal de una clase  
d) Las anteriores respuestas no son correctas
6. ¿Para qué sirve una clase?
- a) Para encapsular datos  
b) Para modelar objetos del mundo real  
c) Para simplificar la reutilización de código  
d) (✓) Todas las respuestas son correctas
7. ¿Cuál no es un nivel de visibilidad en C++?
- a) protected  
b) (✓) hidden  
c) private  
d) public
8. ¿Cuál es una declaración correcta de una clase?
- a) (✓) class A {int x;};  
b) class B {}  
c) public class A {}  
d) object A {int x;};
9. Un espacio de nombres se emplea para:
- a) Definir una función miembro fuera de la definición de su clase.  
b) (✓) Evitar la colisión de nombres de los identificadores (nombres de variables, funciones, etc.)  
c) Lograr un aumento de la velocidad de ejecución del código.  
d) Todas las respuestas son correctas.
10. El operador de ámbito se emplea para:

- a) Identificar una función miembro cuando se define fuera de su clase.
- b) Acceder a un elemento definido en un espacio de nombres.
- c) Para acceder a los miembros estáticos de una clase.
- d) (✓) Todas las respuestas son correctas.

## 2.14. Respuesta a los ejercicios de programación

### 2.14.1. Clase TVector

---

Ejemplo 2.13

---

```
1 class TVector {  
2     private:  
3         int dimension;  
4         int *datos;  
5     };
```

---

### 2.14.2. Clase TCalendario

---

Ejemplo 2.14

---

```
1 class TCalendario {  
2     private:  
3         int dia, mes, anyo;  
4         char *mensaje;  
5     };
```

---

# Capítulo 3

# Constructor y destructor

En este capítulo se introducen los conceptos de constructor y destructor, funciones miembro especiales de una clase que se invocan automáticamente al crear o eliminar un objeto. Como en C++ una clase puede tener varios constructores con el mismo nombre de función, primero se explica la sobrecarga de funciones, que permite crear varias funciones con el mismo nombre.

## Índice General

---

<b>3.1.</b>	<b>Sobrecarga de funciones . . . . .</b>	<b>32</b>
<b>3.2.</b>	<b>Constructor . . . . .</b>	<b>34</b>
<b>3.3.</b>	<b>Constructor por defecto . . . . .</b>	<b>34</b>
<b>3.4.</b>	<b>Otros constructores . . . . .</b>	<b>36</b>
<b>3.5.</b>	<b>Constructor de copia . . . . .</b>	<b>37</b>
<b>3.6.</b>	<b>¿Un constructor en la parte privada? . . . . .</b>	<b>42</b>
<b>3.7.</b>	<b>Destructor . . . . .</b>	<b>43</b>
<b>3.8.</b>	<b>Forma canónica de una clase . . . . .</b>	<b>44</b>
<b>3.9.</b>	<b>Ejercicios de autoevaluación . . . . .</b>	<b>45</b>
<b>3.10.</b>	<b>Ejercicios de programación . . . . .</b>	<b>47</b>
3.10.1.	Clase TCoordenada . . . . .	47
3.10.2.	Clase TVector . . . . .	47
3.10.3.	Clase TCalendario . . . . .	48
<b>3.11.</b>	<b>Respuesta a los ejercicios de autoevaluación . . . . .</b>	<b>48</b>
<b>3.12.</b>	<b>Respuesta a los ejercicios de programación . . . . .</b>	<b>50</b>
3.12.1.	Clase TVector . . . . .	50
3.12.2.	Clase TCalendario . . . . .	52

---

### 3.1. Sobrecarga de funciones

La **sobrecarga de funciones** consiste en definir varias funciones con el mismo nombre. Pero, ¿cómo sabe el compilador qué función invocar en cada caso? Para ello, las funciones se tienen que distinguir por el número, orden o tipo de sus parámetros.

Un caso típico de sobrecarga de funciones son las funciones matemáticas, que se sobrecargan para diferentes tipos de datos numéricos. Por ejemplo, el siguiente código muestra tres usos de la función `min()`<sup>1</sup>, sobrecargada para números enteros (`int`), enteros largos (`long`) y reales (`double`):

---

Ejemplo 3.1

---

```

1 #include <iostream>
2
3 using namespace std;
4
5 int
6 main(void)
7 {
8     cout << min(1, 2) << endl;
9
10    cout << min(1L, 2L) << endl;
11
12    cout << min(1.1, 2.2) << endl;
13
14    return 0;
15 }
```

---

En el código anterior, el modificador L indica al compilador que el número se tiene que tratar como un entero largo (`long`) y no como un entero (`int`).

En C++ se puede sobrecargar cualquier función, incluso las funciones de la librería estándar. Por ejemplo, el siguiente código sobrecarga la función `min()` para que acepte tres parámetros:

---

Ejemplo 3.2

---

```

1 #include <iostream>
2
3 using namespace std;
4
5
6 int
7 min(int a, int b, int c)
8 {
9     return min(min(a, b), c);
```

---

<sup>1</sup>La biblioteca estándar de C++ incluye las funciones `min()` y `max()`.

```
10 }
11
12 long
13 min(long a, long b, long c)
14 {
15     return min(min(a, b), c);
16 }
17
18 double
19 min(double a, double b, double c)
20 {
21     return min(min(a, b), c);
22 }
23
24 int
25 main(void)
26 {
27     cout << min(1, 2, 3) << endl;
28
29     cout << min(1L, 2L, 3L) << endl;
30
31     cout << min(1.1, 2.2, 3.3) << endl;
32
33     return 0;
34 }
```

---

Un error muy común cuando se hace uso de la sobrecarga de funciones es definir funciones sobrecargadas con parámetros idénticos y tipos de retorno diferentes. Por ejemplo, la siguiente sobrecarga de la función `calcula()` no es correcta:

---

Ejemplo 3.3

---

```
1 #include <iostream>
2
3 using namespace std;
4
5 int
6 calcula(int a, float b) {
7     return (int) (a * b);
8 }
9
10 float
11 calcula(int a, float b) {
12     return (float) (a * b);
13 }
14
```

```

15 int
16 main(void)
17 {
18     cout << calcula(1, 2.2) << endl;
19
20     cout << calcula(1, 2.2) << endl;
21
22     return 0;
23 }
```

---

Si se compila el código anterior, se produce el siguiente mensaje de error:

■ Salida ejemplo 3.3 ■

```

1 ejem7.cc: In function ‘float calcula(int, float)’:
2 ejem7.cc:9: new declaration ‘float calcula(int, float)’
3 ejem7.cc:5: ambiguates old declaration ‘int calcula(int, float)’
```

## 3.2. Constructor

Un **constructor** es una función miembro especial de una clase que se invoca automáticamente cada vez que se crea un objeto de dicha clase. El objetivo de un constructor es inicializar los datos miembro, reservar memoria dinámica, abrir un fichero o realizar cualquier otra tarea necesaria para el correcto funcionamiento del objeto.

En C++, un constructor tiene el mismo nombre de la clase y no devuelve ningún tipo de dato (ni `void`). Suele estar situado en la parte pública de la clase, aunque también se puede situar en la parte privada, como veremos más adelante.

Una clase puede tener varios constructores gracias a la sobrecarga de funciones. Si no se proporciona un constructor, el compilador genera automáticamente uno que no hace nada.

## 3.3. Constructor por defecto

El **constructor por defecto** es un constructor sin parámetros, una función de cero parámetros<sup>2</sup>.

En nuestra clase `TCoordenada` de ejemplo, cuando se crea un objeto a partir de ella, se desea que sus tres componentes se inicialicen a 0. Para ello, la definición de la clase `TCoordenada` en el fichero `tcoordenada.h` se tiene que modificar para añadir el correspondiente constructor por defecto:

<sup>2</sup>Para más información sobre las funciones de cero parámetros, consultar la Sección 7.2.

---

Ejemplo 3.4

---

```
1 class TCoordenada {  
2     public:  
3         TCoordenada();  
4  
5         void setX(int);  
6         void setY(int);  
7         void setZ(int);  
8  
9         int getX(void);  
10        int getY(void);  
11        int getZ(void);  
12  
13        void Imprimir(void);  
14  
15    private:  
16        int x, y, z;  
17 };
```

---

En el fichero `tcoordenada.cc` añadimos la definición del constructor:

---

Ejemplo 3.5

---

```
1 TCoordenada::TCoordenada() {  
2     x = y = z = 0;  
3 }
```

---

El siguiente ejemplo permite verificar el funcionamiento del constructor por defecto:

---

Ejemplo 3.6

---

```
1 #include <iostream>  
2  
3 using namespace std;  
4  
5 #include "tcoordenada.h"  
6  
7 int main(void) {  
8     int i;  
9     TCoordenada p1;  
10  
11     p1.Imprimir();  
12     cout << endl;  
13  
14     return 0;  
15 }
```

El código anterior produce como salida:

Salida ejemplo 3.6

1 (0, 0, 0)

### 3.4. Otros constructores

En C++, una clase puede tener tantos constructores como se desee gracias a la sobrecarga de funciones. Todo depende de las necesidades del programador.

Por ejemplo, para que se pueda crear un objeto de tipo TCoordenada con sus componentes inicializadas a un valor cualquiera, se tiene que añadir el constructor TCoordenada(int, int, int) en el fichero tcoordenada.h:

Ejemplo 3.7

```

1 class TCoordenada {
2     public:
3         TCoordenada();
4         TCoordenada(int, int, int);
5
6         void setX(int);
7         void setY(int);
8         void setZ(int);
9
10        int getX(void);
11        int getY(void);
12        int getZ(void);
13
14        void Imprimir(void);
15
16    private:
17        int x, y, z;
18 }
```

Y en el fichero tcoordenada.cc:

Ejemplo 3.8

```

1 TCoordenada::TCoordenada(int a, int b, int c) {
2     x = a;
3     y = b;
4     z = c;
5 }
```

Los dos constructores de la clase TCoordenada se invocan de la siguiente forma:

```
1 // Constructor por defecto
2 TCoordenada p1;
3
4 // Constructor sobrecargado a partir de tres enteros
5 TCoordenada p2(10, 20, 30);
```

### 3.5. Constructor de copia

El **constructor de copia** es un constructor especial que permite crear un objeto a partir de otro objeto de la misma clase. El constructor de copia es similar a la asignación de objetos, pero ni la sustituye ni la implementa.

Este constructor se invoca automáticamente en los siguientes casos:

- Cuando se realiza una inicialización explícita de un objeto a partir de otro objeto del mismo tipo:  
`TCoordenada a; TCoordenada b(a);`  
En la declaración de un objeto también se puede emplear la asignación, pero se invoca al constructor de copia<sup>3</sup>:  
`TCoordenada a; TCoordenada b = a;`
- Al pasar un objeto como parámetro por valor<sup>4</sup> a una función:  
`int UnaFuncion(TCoordenada c);`
- Al devolver una función un objeto por valor:  
`TCoordenada OtraFuncion(void);`

Si no se proporciona un constructor de copia, el compilador proporciona un constructor predeterminado de copia, que realiza una copia miembro a miembro (*bit a bit*) del objeto original en el nuevo objeto. Este constructor puede ocasionar problemas cuando un objeto emplea memoria dinámica. Si no se proporciona, se pueden tener dos o más objetos que comparten la misma zona de memoria dinámica: cuando se destruya el primer objeto se eliminará la memoria dinámica empleada y, por tanto, dejará a los otros objetos con una zona de memoria dinámica indefinida.

En la Figura 3.1 mostramos un ejemplo de la situación de error que se puede producir si no se proporciona un constructor de copia adecuado:

**Paso 1:** El objeto `obj1` contiene un puntero llamado `ptr` que apunta a la zona de memoria `0x1234` donde se almacena el valor `xxxxx`.

<sup>3</sup>En la Sección 5.9 se explica la sobrecarga del operador asignación.

<sup>4</sup>Es preferible que los argumentos de las funciones se pasen por referencia, para ahorrar tiempo y espacio, ya que así se evita la invocación del constructor de copia.

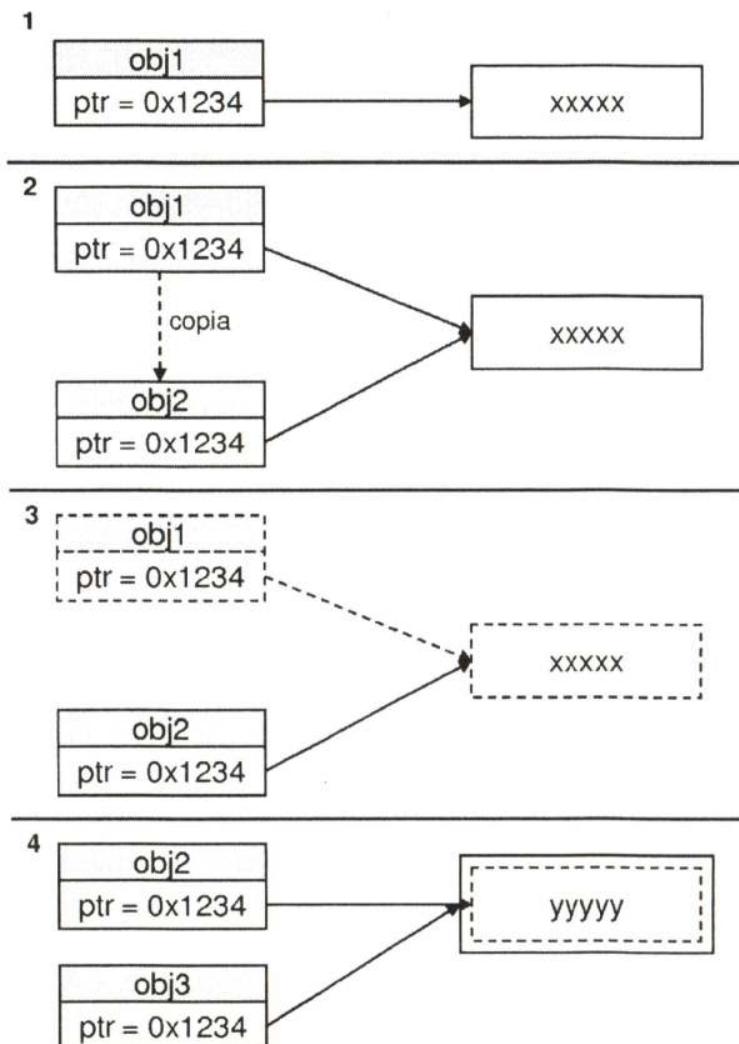


Figura 3.1: Situación de error por no proporcionar un constructor de copia adecuado

**Paso 2:** El compilador crea el objeto **obj2** como copia del objeto **obj1**: el puntero **ptr** de ambos objetos apunta a la misma zona de memoria.

**Paso 3:** El objeto **obj1** se destruye y libera la zona de memoria apuntada por su puntero **ptr**. El puntero del objeto **obj2** apunta ahora a una zona de memoria no válida.

**Paso 4:** Se crea un nuevo objeto **obj3** cuyo puntero **ptr** apunta a la zona de memoria **0x1234** que había quedado libre al destruirse el objeto **obj1**. Existe una situación de error porque los objetos **obj2** y **obj3** comparten la misma zona de memoria.

Por tanto, se recomienda siempre proporcionar un constructor de copia que copie correctamente los objetos de una clase.

Por ejemplo, a continuación se ha añadido a la clase **TCoordenada** el correspondiente constructor de copia:

---

Ejemplo 3.9

---

```
1 class TCoordenada {
2     public:
3         TCoordenada();
4         TCoordenada(int, int, int);
5         TCoordenada(const TCoordenada &);
6
7         void setX(int);
8         void setY(int);
9         void setZ(int);
10
11        int getX(void);
12        int getY(void);
13        int getZ(void);
14
15        void Imprimir(void);
16
17    private:
18        int x, y, z;
19 }
```

---

Y en el fichero **tcoordenada.cc** tenemos que añadir:

---

Ejemplo 3.10

---

```
1 TCoordenada::TCoordenada(const TCoordenada & c) {
2     x = c.x;
3     y = c.y;
4     z = c.z;
5 }
```

---

El siguiente código muestra como crear tres objetos p1, p2 y p3 a partir de la clase TCoordenada mediante el constructor por defecto, el constructor a partir de tres números enteros y el constructor de copia:

---

Ejemplo 3.11

---

```
1 #include <iostream>
2
3 using namespace std;
4
5 #include "tcoordenada.h"
6
7 int
8 main(void)
9 {
10     int i;
11     TCoordenada p1;
12     TCoordenada p2(10, 20, 30);
13     TCoordenada p3(p2);
14
15     p1.Imprimir();
16     cout << endl;
17
18     p1.setX(1);
19     p1.setY(2);
20     p1.setZ(3);
21
22     p1.Imprimir();
23     cout << endl;
24
25     p2.Imprimir();
26     cout << endl;
27
28     p3.Imprimir();
29     cout << endl;
30
31     return 0;
32 }
```

---

El código anterior produce como salida:

---

Salida ejemplo 3.11

---

```
1 (0, 0, 0)
2 (1, 2, 3)
3 (10, 20, 30)
4 (10, 20, 30)
```

El constructor de copia tiene un único parámetro que es un objeto del mismo tipo que el objeto que se desea crear. Este único parámetro se tiene que pasar por referencia, nunca por valor. ¿Por qué? Como se ha indicado al principio de esta sección, el constructor de copia se invoca automáticamente al pasar un objeto por valor a una función. Por tanto, si el objeto que recibe el constructor de copia como parámetro se pasase por valor, se invocaría automáticamente al constructor de copia para hacer una copia de ese objeto y se entraría en un bucle infinito.

Si el compilador es “algo inteligente” puede detectar la situación anterior y avisar al programador durante la compilación. Por ejemplo, en el código siguiente se ha modificado el constructor de copia de la clase TCoordenada y se ha suprimido el paso por referencia:

---

Ejemplo 3.12

---

```
1 class TCoordenada {  
2     public:  
3         TCoordenada();  
4         TCoordenada(int, int, int);  
5         TCoordenada(const TCoordenada);  
6  
7         void setX(int);  
8         void setY(int);  
9         void setZ(int);  
10  
11        int getX(void);  
12        int getY(void);  
13        int getZ(void);  
14  
15        void Imprimir(void);  
16  
17     private:  
18         int x, y, z;  
19 };
```

---

Y en el fichero tcoordenada.cc:

---

Ejemplo 3.13

---

```
1 TCoordenada::TCoordenada(const TCoordenada c) {  
2     x = c.x;  
3     y = c.y;  
4     z = c.z;  
5 }
```

---

Si se compila el código anterior, se produce el siguiente mensaje de error:

## Salida ejemplo 3.13

```

1 In file included from tcoordenada.cc:1:
2 tcoordenada.h:5: invalid constructor; you probably meant 'TCoordenada
3     (const TCoordenada&)'
4 tcoordenada.cc:13: prototype for 'TCoordenada::TCoordenada(TCoordenada)'
5     does not match any in class 'TCoordenada'
6 tcoordenada.h:1: candidates are: TCoordenada::TCoordenada(const
7                         TCoordenada&)
8 tcoordenada.cc:7:                         TCoordenada::TCoordenada(int, int, int)
9 tcoordenada.cc:3:                         TCoordenada::TCoordenada()
10 tcoordenada.cc:13: invalid constructor; you probably meant 'TCoordenada
11     (const TCoordenada&)'
12 tcoordenada.cc:13: syntax error before '{' token
13 tcoordenada.cc:15: ISO C++ forbids declaration of 'y' with no type
14 tcoordenada.cc:15: 'c' was not declared in this scope
15 tcoordenada.cc:16: ISO C++ forbids declaration of 'z' with no type
16 tcoordenada.cc:16: 'c' was not declared in this scope
17 tcoordenada.cc:17: syntax error before '}' token

```

Aunque parezca que hay muchos errores, si se corrige el código del constructor de copia y se vuelve a indicar el paso por referencia, todos los errores desaparecen.

¿Qué significa el modificador `const` que precede al parámetro que recibe el constructor de copia? Más adelante, en la Sección 5.3 y 5.4 se explicará porque es necesario ponerlo siempre.

### 3.6. ¿Un constructor en la parte privada?

Aunque puede parecer que no tiene sentido, en algunas situaciones puede interesar declarar un constructor en la parte privada de una clase para evitar que sea invocado. Por ejemplo, si se coloca el constructor por defecto en la parte privada de una clase, se impide que el usuario lo pueda usar y, por tanto, se obliga al usuario a construir un objeto a partir de otro constructor(por ejemplo, a partir de una serie de valores). Del mismo modo, si el constructor de copia se declara en la parte privada de una clase, se evita que un objeto se pueda construir a partir de un objeto del mismo tipo. El siguiente código muestra esta situación para la clase `TCoordenada`:

## Ejemplo 3.14

```

1 class TCoordenada {
2     public:
3         TCoordenada();
4         TCoordenada(int, int, int);
5

```

```
6     void setX(int);
7     void setY(int);
8     void setZ(int);
9
10    int getX(void);
11    int getY(void);
12    int getZ(void);
13
14    void Imprimir(void);
15
16 private:
17     TCoordenada(const TCoordenada &);
18
19     int x, y, z;
20 }
```

Si se compila el código anterior, se produce el siguiente mensaje de error:

Salida ejemplo 3.14

```
1 tcoordenada.h: In function ‘int main()’:
2 tcoordenada.h:17: ‘TCoordenada::TCoordenada(const TCoordenada& )’ is
3     private
4 main.cc:13: within this context
```

### 3.7. Destructor

El **destructor** es una función miembro de una clase que se invoca automáticamente cada vez que un objeto se destruya y se libere la memoria de dicho objeto. El objetivo del destructor es finalizar correctamente un objeto: liberar memoria dinámica, cerrar un fichero o, en general, liberar cualquier tipo de recurso que tenga en uso el objeto.

En C++, un destructor tiene el mismo nombre de la clase, pero precedido por el carácter virgulilla<sup>5</sup> (~). Un destructor no recibe ningún parámetro ni devuelve ningún valor (ni **void**).

Una clase sólo puede tener un destructor: el destructor no puede sobrecargarse, por la sencilla razón de que no admite argumentos. Si no se proporciona un destructor, el compilador genera automáticamente uno que no hace nada. Si se destruye un objeto que emplea memoria dinámica y no posee destructor que la libere, la memoria que emplea permanecerá ocupada y sin posibilidad de volver a ser usada hasta que finalice el programa.

<sup>5</sup>La vírgula o virgulilla es el carácter tilde que aparece en la letra “ñ”. En el teclado se puede obtener mediante la combinación de teclas **AltGr+4** o **Alt+126**.

Por ejemplo, el siguiente código añade a la clase TCoordenada un destructor que inicializa a 0 las tres componentes de un objeto:

---

Ejemplo 3.15

---

```

1 class TCoordenada {
2     public:
3         TCoordenada();
4         TCoordenada(int, int, int);
5         TCoordenada(const TCoordenada &);
6         ~TCoordenada();
7
8         void setX(int);
9         void setY(int);
10        void setZ(int);
11
12        int getX(void);
13        int getY(void);
14        int getZ(void);
15
16        void Imprimir(void);
17
18    private:
19        int x, y, z;
20 };

```

---

Y en el fichero tcoordenada.cc:

---

Ejemplo 3.16

---

```

1 TCoordenada::~TCoordenada() {
2     x = y = z = 0;
3 }

```

---

### 3.8. Forma canónica de una clase

La forma canónica u ortodoxa de una clase en C++ define el diseño correcto de una clase. Respetar este diseño es importante, ya que permite usar la clase de la misma manera que cualquier tipo básico de C++.

Una clase en su forma canónica tiene que contener, como mínimo, los siguientes métodos: constructor por defecto, constructor de copia, destructor y sobrecarga del operador asignación. En la Sección 7.1 se explicará con más detalle la forma canónica de una clase.

### 3.9. Ejercicios de autoevaluación

1. En la sobrecarga de funciones, para que sea correcta, las funciones se tienen que distinguir por:
  - a) El número de los parámetros
  - b) El orden de los parámetros
  - c) El tipo de los parámetros
  - d) Todas las respuestas son correctas
2. Una clase puede tener:
  - a) Todos los constructores que se deseé
  - b) Sólo el constructor por defecto y el constructor de copia
  - c) Un destructor por cada constructor
  - d) Las anteriores respuestas no son correctas
3. Respecto a los constructores de una clase:
  - a) Todos tienen el mismo nombre
  - b) Su nombre coincide con el nombre de la clase
  - c) Pueden tener cualquier tipo y cualquier número de parámetros
  - d) Todas las respuestas son correctas
4. ¿Qué valor debe devolver un destructor?
  - a) Un puntero a la clase
  - b) Un objeto de la clase
  - c) Un código de estado que indica si la clase se ha destruido correctamente
  - d) Los destructores no devuelven un valor
5. ¿Qué funciones debe de tener toda clase?
  - a) Ninguna
  - b) Constructor
  - c) Destructor
  - d) Constructor y destructor
6. Para inicializar los datos miembro de una clase, se emplea:
  - a) Constructor

- b) Destructor
- c) Se inicializan automáticamente
- d) Las anteriores respuestas no son correctas

7. El constructor de copia:

- a) Crea un objeto a partir de un objeto cualquiera
- b) Crea un objeto a partir de un objeto de la misma clase
- c) Crea un objeto que es un puntero al objeto copiado
- d) Las anteriores respuestas no son correctas

8. El constructor de copia se invoca:

- a) Cuando se pasa un objeto como parámetro a una función por referencia
- b) Cuando una función devuelve un objeto por valor
- c) Al asignar un objeto a otro objeto
- d) Las anteriores respuestas no son correctas

9. Después de invocar al destructor de un objeto:

- a) No se puede acceder a los miembros de dicho objeto
- b) Se puede acceder a los miembros de dicho objeto siempre que el objeto sea constante
- c) Se puede acceder a los miembros de dicho objeto sin problema
- d) Las anteriores respuestas no son correctas

10. Cuál de las siguientes declaraciones es verdadera:

- a) El constructor de copia recibe como argumento un objeto del mismo tipo pasado por referencia o por valor
- b) Si un objeto se sale de ámbito entonces se invoca automáticamente al destructor de ese objeto
- c) El constructor de copia sustituye al operador asignación cuando no se ha definido
- d) Si no se proporciona un destructor, el compilador crea automáticamente uno que contiene las instrucciones del constructor por defecto pero en orden inverso

## 3.10. Ejercicios de programación

### 3.10.1. Clase TCoordenada

Comprobar cómo se invocan los constructores y el destructor en cada situación. Para ello, modificar la clase **TCoordenada** y colocar instrucciones como la siguiente en cada constructor y destructor:

```
1 TCoordenada::TCoordenada(const TCoordenada & c) {  
2     cout << "Constructor de copia" << endl;  
3     x = c.x;  
4     y = c.y;  
5     z = c.z;  
6 }
```

Crear un array de 10 objetos **TCoordenada**. ¿Cuántas veces se invoca al constructor? ¿Y el destructor?

Añadir una función que reciba un objeto **TCoordenada** por valor. Comprobar que se invoca automáticamente al constructor de copia al llamar a la función.

### 3.10.2. Clase TVector

Dada la clase **TVector** que contiene un vector dinámico de números enteros y un número entero que contiene la dimensión del vector, definid en C++:

- Constructor por defecto (dimensión 10 y componentes inicializadas a -1).
- Constructor a partir de una dimensión (componentes inicializadas a -1).
- Constructor de copia.
- Destructor.
- **Dimension()**: devuelve la dimensión del vector.
- **Almacenar(int posicion, int valor)**: acceso a las componentes del vector para almacenar un valor, las posiciones tienen que empezar desde 1.
- **Recuperar(int posicion)**: acceso a las componentes del vector para recuperar un valor, las posiciones tienen que empezar desde 1.
- **Imprimir()**: muestra el contenido del vector componente a componente, separadas por un espacio en blanco y todo el vector encerrado entre corchetes. Por ejemplo:  
[3 1 16 5]

¿Cómo se invocan los constructores?

### 3.10.3. Clase TCalendario

Dada la clase `TCalendario` que contiene una fecha (representada mediante tres variables enteras para el día, mes y año) y un mensaje (representado mediante un vector dinámico de caracteres), definid en C++:

- Constructor por defecto (fecha a 01/01/01) y ningún mensaje.
- Constructor a partir de una fecha.
- Constructor a partir de una fecha y un mensaje.
- Constructor de copia.
- Destructor.
- `Imprimir()`: muestra la fecha con el formato estándar d/m/a y el mensaje.

¿Cómo se invocan los constructores?

## 3.11. Respuesta a los ejercicios de autoevaluación

1. En la sobrecarga de funciones, para que sea correcta, las funciones se tienen que distinguir por:
  - a) El número de los parámetros
  - b) El orden de los parámetros
  - c) El tipo de los parámetros
  - d) (✓) Todas las respuestas son correctas
2. Una clase puede tener:
  - a) (✓) Todos los constructores que se deseé
  - b) Sólo el constructor por defecto y el constructor de copia
  - c) Un destructor por cada constructor
  - d) Las anteriores respuestas no son correctas
3. Respecto a los constructores de una clase:
  - a) Todos tienen el mismo nombre
  - b) Su nombre coincide con el nombre de la clase
  - c) Pueden tener cualquier tipo y cualquier número de parámetros
  - d) (✓) Todas las respuestas son correctas

4. ¿Qué valor debe devolver un destructor?
  - a) Un puntero a la clase
  - b) Un objeto de la clase
  - c) Un código de estado que indica si la clase se ha destruido correctamente
  - d) **(✓) Los destructores no devuelven un valor**
5. ¿Qué funciones debe de tener toda clase?
  - a) Ninguna
  - b) Constructor
  - c) Destructor
  - d) **(✓) Constructor y destructor**
6. Para inicializar los datos miembro de una clase, se emplea:
  - a) **(✓) Constructor**
  - b) Destructor
  - c) Se inicializan automáticamente
  - d) Las anteriores respuestas no son correctas
7. El constructor de copia:
  - a) Crea un objeto a partir de un objeto cualquiera
  - b) **(✓) Crea un objeto a partir de un objeto de la misma clase**
  - c) Crea un objeto que es un puntero al objeto copiado
  - d) Las anteriores respuestas no son correctas
8. Después de invocar al destructor de un objeto:
  - a) No se puede acceder a los miembros de dicho objeto
  - b) Se puede acceder a los miembros de dicho objeto siempre que el objeto sea constante
  - c) **(✓) Se puede acceder a los miembros de dicho objeto sin problema**
  - d) Las anteriores respuestas no son correctas
9. Cuál de las siguientes declaraciones es verdadera:
  - a) El constructor de copia recibe como argumento un objeto del mismo tipo pasado por referencia o por valor

- b) (✓) Si un objeto se sale de ámbito entonces se invoca automáticamente al destructor de ese objeto
- c) El constructor de copia sustituye al operador asignación cuando no se ha definido
- d) Si no se proporciona un destructor, el compilador crea automáticamente uno que contiene las instrucciones del constructor por defecto pero en orden inverso

## 3.12. Respuesta a los ejercicios de programación

### 3.12.1. Clase TVector

El constructor por defecto y el constructor a partir de una dimensión se pueden juntar en uno solo si se emplean los valores por omisión<sup>6</sup>:

Fichero tvector.h:

Ejemplo 3.17

```

1 #include <iostream>
2
3 using namespace std;
4
5 class TVector {
6     public:
7         TVector(int = 10);
8         TVector(const TVector &);
9         ~TVector();
10
11     int Dimension(void);
12     void Almacenar(int, int);
13     int Recuperar(int);
14     void Imprimir(void);
15
16     private:
17         int dimension;
18         int *datos;
19     };

```

Fichero tvector.cc:

Ejemplo 3.18

```

1 #include "tvector.h"
2

```

<sup>6</sup>Para más información sobre los valores por omisión, consultar la Sección 7.3.

```
3 TVector::TVector(int dim) {
4     dimension = dim;
5     datos = new int[dimension];
6     if(datos == NULL) {
7         dimension = 0;
8         return;
9     }
10    for(int i = 0; i < dimension; i++)
11        datos[i] = -1;
12 }
13
14 TVector::TVector(const TVector &origen) {
15     dimension = origen.dimension;
16     datos = new int[dimension];
17     if(datos == NULL) {
18         dimension = 0;
19         return;
20     }
21     for(int i = 0; i < dimension; i++)
22         datos[i] = origen.datos[i];
23 }
24
25 TVector::~TVector() {
26     dimension = 0;
27     if(datos != NULL) {
28         delete datos;
29         datos = NULL;
30     }
31 }
32
33 int
34 TVector::Dimension(void) {
35     return dimension;
36 }
37
38 void
39 TVector::Almacenar(int posicion, int valor) {
40     if(posicion >= 1 && posicion <= dimension)
41         datos[posicion - 1] = valor;
42 }
43
44 int
45 TVector::Recuperar(int posicion) {
46     if(posicion >= 1 && posicion <= dimension)
47         return datos[posicion - 1];
48     else
```

---

```

49     return -1;
50 }
51
52 void
53 TVector::Imprimir(void) {
54     int i;
55
56     cout << "[";
57     if(dimension > 0)
58         cout << datos[0];
59     for(i = 1; i < dimension; i++)
60         cout << " " << datos[i];
61     cout << "]";
62 }
```

---

En el código anterior se puede ver que después de cada reserva de memoria se tiene que comprobar que se ha podido realizar correctamente.

Los constructores se pueden invocar de la siguiente forma:

1	TVector a;
2	TVector b(20);
3	TVector c(b);

### 3.12.2. Clase TCalendario

Fichero tcalendario.h:

---

Ejemplo 3.19

---

```

1 class TCalendario {
2     public:
3         TCalendario();
4         TCalendario(int, int, int);
5         TCalendario(int, int, int, char *);
6         TCalendario(const TCalendario &);
7         ~TCalendario();
8
9     void Imprimir(void);
10
11    private:
12        int dia, mes, anyo;
13        char *mensaje;
14 }
```

---

Fichero tcalendario.cc:

---

Ejemplo 3.20

---

```
1 #include <cstring>
2
3 #include "tcalendario.h"
4
5 TCalendario::TCalendario() {
6     dia = mes = anyo = 1;
7     mensaje = NULL;
8 }
9
10 TCalendario::TCalendario(int d, int m, int a) {
11     dia = d;
12     mes = m;
13     anyo = a;
14     mensaje = NULL;
15 }
16
17 TCalendario::TCalendario(int d, int m, int a, char *ms) {
18     dia = d;
19     mes = m;
20     anyo = a;
21     mensaje = new char[strlen(ms) + 1];
22     if(mensaje == NULL)
23         return;
24     strcpy(mensaje, ms);
25     // También funciona
26     // strdup(mensaje, ms);
27 }
28
29 TCalendario::TCalendario(const TCalendario &cal) {
30     dia = cal.dia;
31     mes = cal.mes;
32     anyo = cal.anyo;
33     mensaje = new char[strlen(cal.mensaje) + 1];
34     if(mensaje == NULL)
35         return;
36     strcpy(mensaje, cal.mensaje);
37     // También funciona
38     // strdup(mensaje, cal.mensaje);
39 }
40
41 TCalendario::~TCalendario() {
42     dia = mes = anyo = 1;
43     if(mensaje != NULL) {
44         delete mensaje;
45         mensaje = NULL;
```

```
46     }
47 }
48
49 void
50 TCalendario::Imprimir(void) {
51     cout << dia << "/" << mes << "/" << anyo << ":" ;
52     if(mensaje != NULL)
53         cout << mensaje;
54 }
```

---

En el código anterior, se incluye el archivo `cstring` porque contiene la definición de las funciones `strlen`, `strcpy` y `strdup` que se emplean para copiar cadenas del tipo `char*`.

Los constructores se pueden invocar de la siguiente forma:

```
1 | TCalendario a;
2 | TCalendario b(1, 10, 2005, "Inicio clases");
3 | TCalendario c(b);
```

# Capítulo 4

## Funciones y clases amigas y reserva de memoria

En este capítulo se presenta el concepto de amistad entre funciones y clases. Aunque el uso de funciones y clases amigas va en contra del principio de ocultación de la información, puede ser apropiado su uso cuando no haya otra solución, la solución posible sea demasiado compleja o tenga un impacto negativo en el rendimiento del programa. Además, se explica la reserva y eliminación de memoria dinámica y la compilación condicional que facilita el proceso de depuración.

### Índice General

---

<b>4.1. Introducción . . . . .</b>	<b>56</b>
<b>4.2. Declaración de amistad . . . . .</b>	<b>56</b>
<b>4.3. Guardas de inclusión . . . . .</b>	<b>62</b>
<b>4.4. Administración de memoria dinámica . . . . .</b>	<b>64</b>
<b>4.5. Administración de memoria dinámica y arrays de objetos . . . . .</b>	<b>66</b>
<b>4.6. Compilación condicional . . . . .</b>	<b>70</b>
<b>4.7. Directivas #warning y #error . . . . .</b>	<b>72</b>
<b>4.8. Ejercicios de autoevaluación . . . . .</b>	<b>73</b>
<b>4.9. Ejercicios de programación . . . . .</b>	<b>75</b>
4.9.1. Clase TVector . . . . .	75
4.9.2. Clase TCalendario . . . . .	75
<b>4.10. Respuesta a los ejercicios de autoevaluación . . . . .</b>	<b>75</b>
<b>4.11. Respuesta a los ejercicios de programación . . . . .</b>	<b>77</b>
4.11.1. Clase TVector . . . . .	77

4.11.2. Clase TCalendario . . . . .	78
-------------------------------------	----

---

## 4.1. Introducción

En la Sección 2.4 se introdujo el concepto de ocultación de información. Como se recordará, todo lo que se declara en la parte protegida o privada de una clase no es accesible desde fuera de ella. Sin embargo, en algunas situaciones puede ser conveniente o imprescindible saltarse esta restricción y permitir que desde fuera de la clase se pueda acceder a la parte privada.

En C++, se puede declarar una función o incluso una clase entera como **amiga** (*friend*) de una clase para permitirle tener derecho a acceder a la parte no pública de una clase. En el Capítulo 5 se verá que la amistad es necesaria para lograr la sobrecarga de algunos operadores.

## 4.2. Declaración de amistad

Para declarar una función o clase como amiga de una clase, se tiene que colocar la palabra reservada **friend** antes del prototipo de la función o la declaración de la clase en la definición de la clase. Por ejemplo, en el siguiente código se declara la clase **TLinea** y la función **Distancia** como amigas de la clase **TCoordenada**:

---

Ejemplo 4.1

---

```
1 class TCoordenada {  
2     friend class TLinea;  
3     friend float Distancia(TCoordenada, TCoordenada);  
4  
5     public:  
6         TCoordenada();  
7         TCoordenada(int, int, int);  
8         TCoordenada(const TCoordenada &);  
9         ~TCoordenada();  
10  
11        void setX(int);  
12        void setY(int);  
13        void setZ(int);  
14  
15        int getX(void);  
16        int getY(void);  
17        int getZ(void);  
18  
19        void Imprimir(void);  
20
```

```
21 private:  
22     int x, y, z;  
23 };
```

---

La clase **TLinea** representa una línea en el espacio y está definida por dos objetos de tipo **TCoordenada**. La función **Distancia()** calcula la distancia euclídea entre dos objetos de tipo **TCoordenada**.

Algunas características básicas de la amistad que hay que tener en cuenta son:

- Las funciones amigas no son funciones miembro de la clase, aunque aparezcan en la declaración de la clase.
- Al declarar una clase A como amiga de otra clase B, todas las funciones miembro de la clase A son amigas de la clase B.
- Si se desea únicamente permitir que unas funciones concretas de una clase sean amigas de una clase, se tiene que emplear el operador de ámbito `::` para indicar una por una las funciones que son amigas.
- Las funciones y clases amigas se pueden colocar en cualquier parte de la declaración de la clase (los modificadores de acceso no se les aplica).
- Las funciones y clases amigas se suelen colocar al principio de la declaración de la clase.
- La amistad se otorga, no se toma: para que la clase B sea amiga de la clase A, la clase A lo debe de permitir de forma explícita en su declaración.
- La amistad no es una propiedad simétrica: si la clase B es amiga de la clase A, eso no significa que se cumpla al contrario.
- La amistad no es una propiedad transitiva: si la clase A es amiga de la clase B y la clase B es amiga de la clase C, eso no significa que la clase A sea amiga de la clase C.

El código de la función **Distancia()** lo añadimos al fichero **tcoordenada.cc**. También se podría almacenar en un fichero independiente y hacer referencia a él con una instrucción `#include`. Para calcular la distancia entre dos puntos necesitamos las funciones **pow()** y **sqrt()** que se encuentran en el fichero **cmath** de la librería estándar de C++. Por tanto, se tiene que añadir al fichero **tcoordenada.cc** la instrucción `#include <cmath>`. El código de la función **Distancia()** queda como:

---

#### Ejemplo 4.2

---

```
1 #include <cmath>  
2
```

```

3  using namespace std;
4
5  float
6  Distancia(TCoordenada a, TCoordenada b) {
7      float d;
8
9      d = pow((a.x - b.x), 2);
10     d += pow((a.y - b.y), 2);
11     d += pow((a.z - b.z), 2);
12
13
14     return sqrt(d);
15 }
```

Si compilamos el fichero `tcoordenada.cc` con la función `Distancia()`, obtenemos los siguientes mensajes de error:

Salida ejemplo 4.2

```

1  tcoordenada.cc: In function ‘float Distancia(TCoordenada, TCoordenada)’:
2  tcoordenada.cc:72: call of overloaded ‘pow(int, int)’ is ambiguous
3  /usr/include/bits/mathcalls.h:154: candidates are: double pow(double,
4      double)
5  /usr/include/c++/3.2.2/cmath:427:           long double std::pow(long
6      double, int)
7  /usr/include/c++/3.2.2/cmath:423:           float std::pow(float, int)
8  /usr/include/c++/3.2.2/cmath:419:           double std::pow(double, int)
9  /usr/include/c++/3.2.2/cmath:410:           long double std::pow(long
10     double, long double)
11 /usr/include/c++/3.2.2/cmath:401:           float std::pow(float, float)
12 tcoordenada.cc:73: call of overloaded ‘pow(int, int)’ is ambiguous
13 /usr/include/bits/mathcalls.h:154: candidates are: double pow(double,
14     double)
15 /usr/include/c++/3.2.2/cmath:427:           long double std::pow(long
16     double, int)
17 /usr/include/c++/3.2.2/cmath:423:           float std::pow(float, int)
18 /usr/include/c++/3.2.2/cmath:419:           double std::pow(double, int)
19 /usr/include/c++/3.2.2/cmath:410:           long double std::pow(long
20     double, long double)
21 /usr/include/c++/3.2.2/cmath:401:           float std::pow(float, float)
22 tcoordenada.cc:74: call of overloaded ‘pow(int, int)’ is ambiguous
23 /usr/include/bits/mathcalls.h:154: candidates are: double pow(double,
24     double)
25 /usr/include/c++/3.2.2/cmath:427:           long double std::pow(long
26     double, int)
27 /usr/include/c++/3.2.2/cmath:423:           float std::pow(float, int)
28 /usr/include/c++/3.2.2/cmath:419:           double std::pow(double, int)
```

```
29 /usr/include/c++/3.2.2/cmath:410:           long double std::pow(long
30     double, long double)                         float std::pow(float, float)
31 /usr/include/c++/3.2.2/cmath:401:
```

El error está en la llamada a la función `pow()`, porque no está definida para dos parámetros de tipo `int` y el compilador no sabe cuál elegir de las posibles funciones `pow()` sobrecargadas: `pow(long double, int)`, `pow(float, int)`, `pow(double, int)`, etc. Este error se soluciona con una conversión de tipo explícita, por ejemplo, a `float`. Notad que cuando se escribe el código de una función amiga, no se tiene que colocar el modificador `friend`.

Por tanto, el código una vez corregido queda como:

---

#### Ejemplo 4.3

---

```
1 #include <cmath>
2
3 using namespace std;
4
5 float
6 Distancia(TCoordenada a, TCoordenada b) {
7     float d;
8
9     d = pow((float) (a.x - b.x), 2);
10    d += pow((float) (a.y - b.y), 2);
11    d += pow((float) (a.z - b.z), 2);
12
13
14    return sqrt(d);
15 }
```

---

La clase `TLinea` se declara y define en ficheros independientes a `TCoordenada`. A continuación se incluye el código de la declaración de la clase que se almacena en el fichero `tlinea.h`:

---

#### Ejemplo 4.4

---

```
1 #include "tcoordenada.h"
2
3 class TLinea {
4     public:
5         TLinea();
6         TLinea(const TCoordenada &, const TCoordenada &);
7         TLinea(const TLinea &);
8         ~TLinea();
9         float Longitud(void);
```

```
10
11     private:
12         TCoordenada p1, p2;
13     };
```

---

La clase **TLinea** representa una línea definida mediante dos puntos representados por dos objetos **p1** y **p2** de tipo **TCoordenada**. Por ello, se necesita incluir la clase **TCoordenada** con la instrucción `#include "tcoordenada.h"`.

El constructor **TLinea(const TCoordenada &, const TCoordenada &)** construye un objeto línea a partir de los dos puntos que lo definen. La función miembro **Longitud()** calcula la longitud de la línea. En la declaración de esta clase se emplea la composición de objetos (un objeto que contiene otro objeto), que se explicará con detalle en el Capítulo 6.

El código de la clase se almacena en el fichero **tlinea.cc**:

---

Ejemplo 4.5

---

```
1 #include "tlinea.h"
2
3 TLinea::TLinea() {
4     p1.x = 0;
5     p1.y = 0;
6     p1.z = 0;
7
8     p2.x = 0;
9     p2.y = 0;
10    p2.z = 0;
11 }
12
13 TLinea::TLinea(const TCoordenada & a, const TCoordenada & b) {
14     p1.x = a.x;
15     p1.y = a.y;
16     p1.z = a.z;
17
18     p2.x = b.x;
19     p2.y = b.y;
20     p2.z = b.z;
21 }
22
23 TLinea::TLinea(const TLinea & l) {
24     p1.x = l.p1.x;
25     p1.y = l.p1.y;
26     p1.z = l.p1.z;
27
28     p2.x = l.p2.x;
29     p2.y = l.p2.y;
```

```
30     p2.z = l.p2.z;
31 }
32
33 TLinea::~TLinea() {
34     p1.x = 0;
35     p1.y = 0;
36     p1.z = 0;
37
38     p2.x = 0;
39     p2.y = 0;
40     p2.z = 0;
41 }
42
43 float
44 TLinea::Longitud(void) {
45     return Distancia(p1, p2);
46 }
```

---

Un posible programa `main.cc` que usa la clase `TLinea`:

---

Ejemplo 4.6 —

```
1 #include <iostream>
2
3 using namespace std;
4
5 #include "tcoordenada.h"
6 #include "tlinea.h"
7
8 int
9 main(void)
10 {
11     TCoordenada p1;
12     TCoordenada p2(1, 2, 3);
13     TCoordenada p3(p2);
14
15     TLinea l1, l2(p1, p2);
16
17     cout << Distancia(p1, p2) << endl;
18
19     return 0;
20 }
```

---

Al compilar el código anterior, se produce el siguiente mensaje de error:

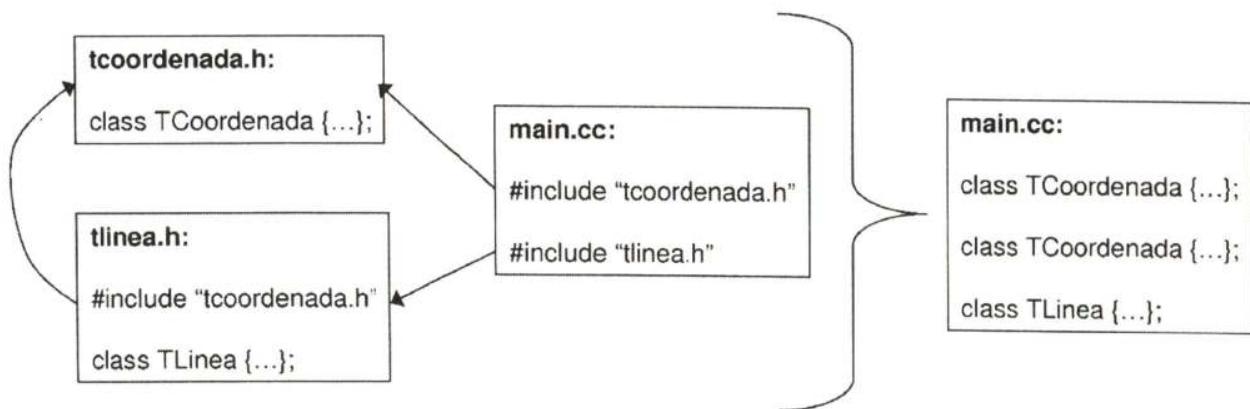


Figura 4.1: Inclusión de la misma definición de clase dos veces

#### Salida ejemplo 4.6

```

1 In file included from tlinea.h:1,
      from main.cc:6:
2 tcoordenada.h:2: redefinition of 'class TCoordenada'
3 tcoordenada.h:2: previous definition of 'class TCoordenada'
4 tcoordenada.h:4: warning: 'float Distancia(TCoordenada, TCoordenada)'
5     is already a friend of class 'TCoordenada'
6 tcoordenada.h:4: warning: previous friend declaration of 'float
7     Distancia(TCoordenada, TCoordenada)'
8

```

¿Qué está pasando? ¿Qué significa **redefinition of 'class TCoordenada'**?

### 4.3. Guardas de inclusión

¿Cuál es el problema del ejemplo anterior? En C++, una clase o función debe de definirse en un programa una sola vez. Si la definición de una clase o función se incluye más de una vez, el compilador producirá un mensaje de error.

En el código de la sección anterior, se produce un error al compilar el programa principal `main.cc` porque la clase `TCoordenada` se incluye dos veces. ¿Cómo puede ser que se incluya dos veces si en el programa principal sólo hay una instrucción `#include "tcoordenada.h"`? En la Figura 4.1 se muestra de forma esquemática el contenido de los ficheros `tcoordenada.h`, `tlinea.h` y `main.cc`. Como el fichero `tlinea.h` posee una instrucción `#include "tcoordenada.h"`, al compilar el fichero `main.cc` se incluye dos veces la clase `TCoordenada`.

¿Cómo se puede resolver este problema? La solución más sencilla es emplear las **guardas de inclusión**, que emplean las instrucciones del preprocesador para incluir de

forma condicional la definición de una clase sólo en el caso de que no se haya incluido previamente.

El preprocesador admite las siguientes instrucciones<sup>1</sup>:

- **#define**: define constantes simbólicas y macros.
- **#undef**: descarta una constante simbólica o una macro.
- **#if** y **#endif**: construcción condicional.
- **#ifdef** y **#ifndef**: se proporcionan como abreviaturas de **#if defined(nombre)** y **#if !defined(nombre)**.
- **#elif** y **#else**: permite definir varias partes de una construcción condicional.

En una guarda de inclusión, se tiene que asociar a cada clase una constante simbólica única (cuanto más larga y “fea”, mejor). Cuando se incluye la definición de una clase, se comprueba si existe la constante simbólica asociada: si no existe, se crea y se incluye el contenido de la clase; si ya existe, no se incluye el contenido de la clase.

Por ejemplo, a continuación se muestra el contenido del fichero **tcoordenada.h** una vez que se ha incluido una guarda de inclusión:

---

Ejemplo 4.7

---

```
1 #ifndef __TCOORDENADA__  
2 #define __TCOORDENADA__  
3  
4 class TCoordenada {  
5     friend class TLinea;  
6     friend float Distancia(TCoordenada, TCoordenada);  
7  
8     public:  
9         TCoordenada();  
10        TCoordenada(int, int, int);  
11        TCoordenada(const TCoordenada &);  
12  
13        void setX(int);  
14        void setY(int);  
15        void setZ(int);  
16  
17        int getX(void);  
18        int getY(void);  
19        int getZ(void);  
20  
21        void Imprimir(void);  
22
```

---

<sup>1</sup>Hay otras directivas, como **#line** y **#pragma**, que no las vamos a estudiar.

---

```

23     private:
24         int x, y, z;
25     };
26 #endif

```

---

Y lo mismo para el fichero **tlinea.h**:

---

Ejemplo 4.8

---

```

1 #ifndef __TLINEA__
2 #define __TLINEA__
3
4 #include "tcoordenada.h"
5
6 class TLinea {
7     public:
8         TLinea();
9         TLinea(const TCoordenada &, const TCoordenada &);
10        TLinea(const TLinea &);
11        ~TLinea();
12        float Longitud(void);
13
14     private:
15         TCoordenada p1, p2;
16    };
17
18 #endif

```

---

## 4.4. Administración de memoria dinámica

La **administración de memoria dinámica** (**reserva** y **liberación**) se realiza con los operadores **new** y **delete** respectivamente.

El operador **new** se puede emplear para reservar memoria dinámica para cualquier tipo base de C++ (**int**, **float**, etc.) o para objetos definidos por el usuario. Por ejemplo:

```

1 | int *ptrInt = new int;
2 | // Constructor por defecto
3 | TCoordenada *ptrCoor = new TCoordenada;

```

En el momento de reservar memoria se puede inicializar el elemento recién creado. En el caso de objetos, se invoca al constructor correspondiente. Por ejemplo:

```

1 | int *ptrInt = new int(123);
2 | // Constructor a partir de las componentes
3 | TCoordenada *ptrCoor = new TCoordenada(7, 5, 3);

```

Cuando el operador `new` no puede realizar la reserva de memoria, se lanza una excepción y devuelve `NULL`. Después de cada reserva de memoria, se debe de comprobar que la reserva se ha podido realizar completamente, ya que emplear un puntero con valor `NULL` produce el temido mensaje de error `Segmentation fault`. Por ejemplo:

```
1 int *ptrInt = new int(123);
2 if(ptrInt == NULL)
3     cout << "Error" << endl;
4
5 TCoordenada *ptrCoor = new TCoordenada(7, 5, 3);
6 // Otra forma de comprobar ptrCoor == NULL
7 if(!ptrCoor)
8     cout << "Error" << endl;
```

Si el programador ha reservado memoria dinámica, el programador tiene que liberar esa memoria dinámica. De no hacerse así, esa memoria estará ocupada hasta que finalice el programa. Para liberar la memoria dinámica se emplea el operador `delete`. En el caso de objetos, antes de liberar la memoria que ocupa el objeto se invoca automáticamente al destructor. Por ejemplo:

```
1 int *ptrInt = new int(123);
2 TCoordenada *ptrCoor = new TCoordenada(7, 5, 3);
3
4 /*
5     Algo de código
6 */
7
8 // Se libera la memoria que ocupa
9 delete ptrInt;
10
11 // Se libera la memoria que ocupa y se invoca
12 // al destructor de la clase TCoordenada
13 delete ptrCoor;
```

Después de liberar la memoria dinámica a la que apunta un puntero, se debe de inicializar a `NULL` para que no se pueda volver a usar, ya que en caso de usarse se produciría un error (`Segmentation fault`). Esto es necesario hacerlo en el caso de que el puntero pueda volver a usarse más tarde; si se trata de un puntero de ámbito local que no va a usarse más, no hace falta hacerlo. Además, antes de liberar la memoria se debería de comprobar que el puntero realmente contiene una dirección de memoria (no apunta a `NULL`). Por ejemplo:

```
1 int *ptrInt = new int(123);
2 TCoordenada *ptrCoor = new TCoordenada(7, 5, 3);
3
4 /*
```

```

5   Algo de código
6   */
7
8   // Se libera la memoria que ocupa
9   if(ptrInt) {
10    delete ptrInt;
11    ptrInt = NULL;
12 }
13
14 // Se libera la memoria que ocupa y se invoca
15 // al destructor de la clase TCoordenada
16 if(ptrCoor) {
17    delete ptrCoor;
18    ptrCoor = NULL;
19 }

```

## 4.5. Administración de memoria dinámica y arrays de objetos

Cuando un array de objetos se elimina, el destructor de cada objeto del array se tiene que invocar. Esto se hace implícitamente para los arrays estáticos, cuando el array no se ha creado con `new`, por ejemplo:

```

1 void
2 UnaFuncion(void) {
3     TCoordenada a[10];
4
5     /*
6      Algo de código
7     */
8
9     // Se invoca 10 veces al destructor de TCoordenada
10    // para los 10 objetos de a[10]
11 }

```

Sin embargo, cuando el array se ha creado mediante reserva manual de memoria con `new`, el programador debe de indicarlo explícitamente. A continuación, se incluyen varios ejemplos que muestran esta situación. Para comprobar su funcionamiento, vamos a añadir al constructor por defecto y al destructor de la clase `TCoordenada` unas instrucciones que muestren por la salida estándar mensajes de aviso:

---

Ejemplo 4.9

---

```

1 TCoordenada::TCoordenada() {
2     cout << "Constructor por defecto" << endl;

```

```
3     x = y = z = 0;
4 }
5
6 TCoordenada::~TCoordenada() {
7     cout << "Destructor" << endl;
8 }
```

El siguiente ejemplo declara dos arrays, el primero es estático y el segundo es un puntero al que se asigna memoria dinámica mediante `new`:

Ejemplo 4.10

```
1 #include <iostream>
2
3 using namespace std;
4
5 #include "tcoordenada.h"
6
7 int
8 main(void)
9 {
10     TCoordenada array[3];
11     TCoordenada *ptr;
12
13     cout << "Reserva memoria" << endl;
14     ptr = new TCoordenada[3];
15     if(ptr == NULL)
16         return;
17
18     cout << "Fin" << endl;
19 }
```

Si se ejecuta el código anterior, se obtiene la siguiente salida:

Salida ejemplo 4.10

```
1 Constructor por defecto
2 Constructor por defecto
3 Constructor por defecto
4 Reserva memoria
5 Constructor por defecto
6 Constructor por defecto
7 Constructor por defecto
8 Fin
9 Destructor
10 Destructor
11 Destructor
```

En la salida, aparece seis veces el mensaje **Constructor por defecto**, tres veces para **array** y otras tres veces para **ptr**. Sin embargo, sólo aparece tres veces el mensaje **Destructor**. ¿Qué está pasando? Los objetos de **array** si que se están destruyendo, pero para los objetos de **ptr** no se invoca el destructor. Como ha sido el programador el que ha reservado la memoria para **ptr**, es también el programador el que tiene que liberar esa memoria mediante **delete**:

---

Ejemplo 4.11

---

```
1 #include <iostream>
2
3 using namespace std;
4
5 #include "tcoordenada.h"
6
7 int
8 main(void)
9 {
10     TCoordenada array[3];
11     TCoordenada *ptr;
12
13     cout << "Reserva memoria" << endl;
14     ptr = new TCoordenada[3];
15     if(ptr == NULL)
16         return;
17
18     cout << "Libera memoria" << endl;
19     delete ptr;
20     ptr = NULL;
21
22     cout << "Fin" << endl;
23 }
```

---

La salida que genera el código anterior es:

---

Salida ejemplo 4.11

---

```
1 Constructor por defecto
2 Constructor por defecto
3 Constructor por defecto
4 Reserva memoria
5 Constructor por defecto
6 Constructor por defecto
7 Constructor por defecto
8 Libera memoria
9 Destructor
10 Fin
```

```
11 Destructor  
12 Destructor  
13 Destructor
```

Ahora aparece el mensaje **Destructor** una vez más, pero no aparecen las tres veces que debería de aparecer al destruir `ptr`. ¿Qué está pasando ahora? C++ no distingue entre un puntero a un objeto individual y un puntero al elemento inicial de un array. Por tanto, sólo se está llamando al destructor para el primer objeto del array. En estas situaciones, el programador debe indicar que se va a destruir un array de objetos con el operador `delete []`, tal como se muestra en el siguiente ejemplo:

---

Ejemplo 4.12

---

```
1 #include <iostream>  
2  
3 using namespace std;  
4  
5 #include "tcoordenada.h"  
6  
7 int  
8 main(void)  
9 {  
10    TCoordenada array[3];  
11    TCoordenada *ptr;  
12  
13    cout << "Reserva memoria" << endl;  
14    ptr = new TCoordenada[3];  
15    if(ptr == NULL)  
16        return;  
17  
18    cout << "Libera memoria" << endl;  
19    delete [] ptr;  
20    ptr = NULL;  
21  
22    cout << "Fin" << endl;  
23 }
```

---

Como se ve en la salida que genera el código anterior, ahora sí que aparece seis veces el mensaje **Destructor**, tres veces para `array` y otras tres veces para `ptr` (mensajes entre **Libera memoria** y **Fin**):

---

Salida ejemplo 4.12

---

```
1 Constructor por defecto  
2 Constructor por defecto  
3 Constructor por defecto
```

```

4 Reserva memoria
5 Constructor por defecto
6 Constructor por defecto
7 Constructor por defecto
8 Libera memoria
9 Destructor
10 Destructor
11 Destructor
12 Fin
13 Destructor
14 Destructor
15 Destructor

```

Por tanto, existen dos formas de usar el operador `delete` que no se deben de confundir entre sí, ya que el uso incorrecto puede originar errores (depende de la implementación del compilador). El siguiente código muestra varias situaciones de uso correcto o incorrecto:

```

1 void
2 UnaFuncion(void) {
3     TCoordenada *a = new TCoordenada;
4     TCoordenada *b = new TCoordenada[5];
5     TCoordenada *c = new TCoordenada;
6     TCoordenada *d = new TCoordenada[5];
7
8     // Puntero a un solo objeto
9     delete a;          // Correcto
10    // Puntero a un array de objetos
11    delete [] b;      // Correcto
12    // Puntero a un solo objeto
13    delete [] c;      // Incorrecto
14    // Puntero a un array de objetos
15    delete d;         // Incorrecto
16 }

```

## 4.6. Compilación condicional

Además de para las guardas de inclusión (ver Sección 4.3), las instrucciones del preprocesador también se pueden emplear para lograr una **compilación condicional** de cualquier parte del código.

En la sección anterior, se ha modificado el código de la clase `TCoordenada` para mostrar una serie de mensajes que permitan verificar la ejecución del constructor por defecto y el destructor. Estos mensajes vienen muy bien en la fase de desarrollo y

depuración<sup>2</sup> del código, pero deberían de desaparecer en la versión final del código. ¿Qué podemos hacer? ¿Tenemos que eliminar esas instrucciones una a una al final? ¿Y si nos olvidamos de eliminar alguna o eliminamos alguna que sí que hace falta?

Para solucionar este problema, podemos emplear la compilación condicional, que permite controlar el proceso de compilación para que se compilen distintas partes del código según deseé el programador. De este modo, se pueden activar o desactivar diferentes partes del código.

En primer lugar, se tienen que definir una o varias constantes simbólicas (se pueden establecer distintos niveles de compilación) al principio del código con la instrucción `#define`:

```

1 | #define DEPURA1
2 | #define DEPURA2
3 | #define DEPURA3

```

Las instrucciones de salida que se emplean para mostrar los valores de las variables y comprobar el flujo de ejecución del código se tienen que encerrar en directivas condicionales que dependan de las constantes simbólicas definidas, de manera que las instrucciones sólo se compilen en el caso que el programador lo deseé. Por ejemplo:

```

1 | #ifdef DEPURA1
2 |     cout << "Un mensaje" << endl;
3 | #elif defined(DEPURA2)
4 |     cout << "Otro mensaje" << endl;
5 | #elif defined(DEPURA3)
6 |     cout << "El último mensaje" << endl;
7 | #else
8 |     // No hacemos nada
9 | #endif

```

Cuando no se deseé incluir las instrucciones de salida en el código, simplemente se comenta la definición de las constantes simbólicas.

Por ejemplo, a continuación se muestra parte del código de la clase TCoordenada, en el que se ha definido la constante simbólica DEPURACION que permite activar o desactivar los mensajes que se muestran por la salida estándar:

---

Ejemplo 4.13

---

```

1 #include "tcoordenada.h"
2
3 #define DEPURACION
4
5 TCoordenada::TCoordenada() {
6

```

<sup>2</sup>Para más información sobre la depuración, consultar la Sección D.3.

```

7 #ifdef DEPURACION
8     cout << "Constructor por defecto" << endl;
9 #endif
10
11     x = y = z = 0;
12 }
13
14 TCoordenada::~TCoordenada() {
15 #ifdef DEPURACION
16     cout << "Destructor" << endl;
17 #endif
18 }
```

---

## 4.7. Directivas `#warning` y `#error`

Otra alternativa a la compilación condicional explicada en la sección anterior es emplear la directiva `#warning` para marcar zonas del código que no queremos que se incluyan en la versión final. Cuando se compila el código, el compilador genera un mensaje de advertencia por cada directiva `#warning` que encuentre y continúa con el proceso de compilación de forma normal.

Por ejemplo, en el siguiente fragmento de código se han incluido dos directivas `#warning`:

Ejemplo 4.14

```

1 #include "tcoordenada.h"
2
3 TCoordenada::TCoordenada() {
4 #warning Eliminar esta instrucción
5     cout << "Constructor por defecto" << endl;
6
7     x = y = z = 0;
8 }
9
10 TCoordenada::~TCoordenada() {
11 #warning Eliminar esta instrucción
12     cout << "Destructor" << endl;
13 }
```

---

Cuando se compila el código anterior, el compilador genera la siguiente salida:

Salida ejemplo 4.14

```

1 tcoordenada.cc:4:2: warning: #warning Eliminar esta instrucción
2 tcoordenada.cc:11:2: warning: #warning Eliminar esta instrucción
```

Cuando no se deseé incluir las instrucciones de salida en el código, se toma nota de los números de línea en los mensajes de advertencia y se comentan las líneas.

Por otro lado, la directiva `#error` produce que el compilador genere un mensaje de error e impide la generación del correspondiente fichero objeto.

## 4.8. Ejercicios de autoevaluación

1. Respecto a la amistad:

- a) Al declarar una función como amiga de una clase, la función pasa a ser miembro de la clase
- b) Una función amiga sólo puede acceder a la parte protegida y nunca a la privada
- c) Una función amiga no puede tener como parámetro un objeto de la clase de la que es amiga
- d) Las anteriores respuestas no son correctas

2. Respecto a la amistad entre clases:

- a) Si la clase A es amiga de la clase B, entonces las funciones miembro de B pueden acceder a la parte privada de A
- b) Si la clase A es amiga de la clase B, entonces B también es amiga de A
- c) Si la clase A es amiga de la clase B y B es amiga de la clase C, entonces A también es amiga de C
- d) Las anteriores respuestas no son correctas

3. Respecto a la amistad y la ocultación de información:

- a) Las funciones y clases amigas sólo pueden acceder a la parte protegida de una clase
- b) La parte privada de una clase sólo es accesible por las funciones miembro de la propia clase
- c) La amistad sólo se puede definir si la clase no posee parte privada
- d) Las anteriores respuestas no son correctas

4. Respecto a las directivas del preprocesador, cuál de las siguientes afirmaciones es falsa:

- a) Todas las directivas empiezan con `#`
- b) Las directivas se procesan después de que el programa se compile
- c) Cada construcción `#if` termina con `#endif`

- d) La directiva `#undef` descarta una constante simbólica
5. ¿Existe alguna diferencia entre las dos siguientes líneas de código?:  
`myObj *x = new myObj[100]; delete x;`  
`myObj *x = new myObj[100]; delete [] x;`
- a) Son equivalentes
  - b) `delete x` invoca el destructor para todos los objetos del array
  - c) `delete [] x` invoca el destructor para todos los objetos del array
  - d) Las anteriores respuestas no son correctas
6. Cuál es el operador empleado para reservar memoria:
- a) `new`
  - b) `create`
  - c) `reserve`
  - d) Las anteriores respuestas no son correctas
7. Cuando en el código se reserva memoria para un array de objetos:
- a) El programador tiene que liberar la memoria con `delete`
  - b) El programador tiene que liberar la memoria con `delete[]`
  - c) El programador tiene que liberar la memoria invocando al destructor de cada objeto de forma individual
  - d) Las anteriores respuestas no son correctas
8. Cuál es el operador empleado para liberar memoria:
- a) `remove`
  - b) `clear`
  - c) `delete`
  - d) Las anteriores respuestas no son correctas
9. Las guardas de inclusión se emplean para:
- a) Incluir de forma condicional la definición de una clase
  - b) Evitar incluir la definición de una clase dos o más veces
  - c) Evitar el problema de la redefinición de una clase
  - d) Todas las respuestas son correctas
10. La compilación condicional permite:

- a) Incluir diferentes partes del código en tiempo de compilación
- b) Incluir diferentes partes del código en tiempo de ejecución
- c) Incluir diferentes partes del código en tiempo de depuración
- d) Todas las respuestas son correctas

## 4.9. Ejercicios de programación

### 4.9.1. Clase TVector

Dada la clase **TVector** que contiene un vector dinámico de números enteros y un número entero que contiene la dimensión del vector, definid en C++ las siguientes funciones amigas:

- **Suma(TVector, TVector)**: suma de dos vectores componente a componente, tiene que comprobar que los dos vectores tienen la misma dimensión, en caso contrario devolverá un vector vacío de dimensión 0.
- **Resta(TVector, TVector)**: resta de dos vectores componente a componente, tiene que comprobar que los dos vectores tienen la misma dimensión, en caso contrario devolverá un vector vacío de dimensión 0.

### 4.9.2. Clase TCalendario

Dada la clase **TCalendario** que contiene una fecha (representada mediante tres variables enteras para el día, mes y año) y un mensaje (representado mediante un vector dinámico de caracteres), definid en C++ la siguiente función amiga:

- **Bisiesto(TCalendario)**: devuelve **true** si la fecha corresponde a una año bisiesto y **false** en caso contrario.

## 4.10. Respuesta a los ejercicios de autoevaluación

1. Respecto a la amistad:

- a) Al declarar una función como amiga de una clase, la función pasa a ser miembro de la clase
- b) Una función amiga sólo puede acceder a la parte protegida y nunca a la privada
- c) Una función amiga no puede tener como parámetro un objeto de la clase de la que es amiga
- d) **(✓) Las anteriores respuestas no son correctas**

2. Respecto a la amistad entre clases:

- a) Si la clase A es amiga de la clase B, entonces las funciones miembro de B pueden acceder a la parte privada de A
- b) Si la clase A es amiga de la clase B, entonces B también es amiga de A
- c) Si la clase A es amiga de la clase B y B es amiga de la clase C, entonces A también es amiga de C
- d) **(✓) Las anteriores respuestas no son correctas**

3. Respecto a la amistad y la ocultación de información:

- a) Las funciones y clases amigas sólo pueden acceder a la parte protegida de una clase
- b) La parte privada de una clase sólo es accesible por las funciones miembro de la propia clase
- c) La amistad sólo se puede definir si la clase no posee parte privada
- d) **(✓) Las anteriores respuestas no son correctas**

4. Respecto a las directivas del preprocesador, cuál de las siguientes afirmaciones es falsa:

- a) Todas las directivas empiezan con #
- b) **(✓) Las directivas se procesan después de que el programa se compile**
- c) Cada construcción #if termina con #endif
- d) La directiva #undef descarta una constante simbólica

5. ¿Existe alguna diferencia entre las dos siguientes líneas de código?:

myObj \*x = new myObj[100]; delete x;  
myObj \*x = new myObj[100]; delete [] x;

- a) Son equivalentes
- b) delete x invoca el destructor para todos los objetos del array
- c) **(✓) delete [] x invoca el destructor para todos los objetos del array**
- d) Las anteriores respuestas no son correctas

6. Cuál es el operador empleado para reservar memoria:

- a) **(✓) new**
- b) create

- c) reserve
  - d) Las anteriores respuestas no son correctas
7. Cuando en el código se reserva memoria para un array de objetos:
- a) El programador tiene que liberar la memoria con delete
  - b) **(✓) El programador tiene que liberar la memoria con delete[]**
  - c) El programador tiene que liberar la memoria invocando al destructor de cada objeto de forma individual
  - d) Las anteriores respuestas no son correctas
8. Cuál es el operador empleado para liberar memoria:
- a) remove
  - b) clear
  - c) **(✓) delete**
  - d) Las anteriores respuestas no son correctas
9. Las guardas de inclusión se emplean para:
- a) Incluir de forma condicional la definición de una clase
  - b) Evitar incluir la definición de una clase dos o más veces
  - c) Evitar el problema de la redefinición de una clase
  - d) **(✓) Todas las respuestas son correctas**
10. La compilación condicional permite:
- a) **(✓) Incluir diferentes partes del código en tiempo de compilación**
  - b) Incluir diferentes partes del código en tiempo de ejecución
  - c) Incluir diferentes partes del código en tiempo de depuración
  - d) Todas las respuestas son correctas

## 4.11. Respuesta a los ejercicios de programación

### 4.11.1. Clase TVector

En `tvector.h`:

---

Ejemplo 4.15

---

```
1 friend TVector Suma(const TVector &, const TVector &);  
2 friend TVector Resta(const TVector &, const TVector &);
```

---

En tvector.cc:

Ejemplo 4.16

```

1 TVector
2 Suma(const TVector &iz, const TVector &de) {
3     if(iz.dimension == de.dimension) {
4         TVector temp(iz);
5
6         for(int i = 0; i < iz.dimension; i++)
7             temp.datos[i] += de.datos[i];
8
9         return temp;
10    }
11    else {
12        TVector temp(0);
13
14        return temp;
15    }
16 }
17
18 TVector
19 Resta(const TVector &iz, const TVector &de) {
20     if(iz.dimension == de.dimension) {
21         TVector temp(iz);
22
23         for(int i = 0; i < iz.dimension; i++)
24             temp.datos[i] -= de.datos[i];
25
26         return temp;
27    }
28    else {
29        TVector temp(0);
30
31        return temp;
32    }
33 }
```

#### 4.11.2. Clase TCalendario

Un año es bisiesto si es divisible por 4 pero, si es divisible por 100, no es año bisiesto, pero si es divisible por 400, sí es año bisiesto.

En tcalendario.h:

Ejemplo 4.17

```
1 friend bool Bisiento(const TCalendario &);
```

---

En `tcalendario.cc`:

---

Ejemplo 4.18

---

```
1 bool
2 Bisiesto(const TCalendario &cal) {
3     if(cal.anyo % 4 == 0)
4     {
5         if(cal.anyo % 100 == 0)
6             if(cal.anyo % 400 != 0)
7                 return false;
8
9     return true;
10 }
11
12 return false;
13 }
```

---

# Capítulo 5

## Sobrecarga de operadores

En este capítulo se explica cómo se pueden redefinir (sobrecargar) los operadores del lenguaje C++ para que funcionen correctamente con las clases creadas por el usuario. Para ello, antes hace falta explicar el puntero this, el modificador const y el paso por referencia.

### Índice General

---

5.1. Introducción . . . . .	82
5.2. Puntero this . . . . .	82
5.3. Modificador const . . . . .	83
5.4. Paso por referencia . . . . .	85
5.5. Sobrecarga de operadores . . . . .	87
5.6. Restricciones al sobrecargar un operador . . . . .	88
5.7. ¿Función miembro o función no miembro? . . . . .	88
5.8. Consejos . . . . .	89
5.9. Operador asignación . . . . .	90
5.10. Constructor de copia y operador asignación . . . . .	93
5.11. Operadores aritméticos . . . . .	94
5.12. Operadores de incremento y decremento . . . . .	98
5.13. Operadores abreviados . . . . .	99
5.14. Operadores de comparación . . . . .	100
5.15. Operadores de entrada y salida . . . . .	101
5.16. Operador corchete . . . . .	102
5.17. Ejercicios de autoevaluación . . . . .	105
5.18. Ejercicios de programación . . . . .	107

---

5.18.1. Clase TCoordenada . . . . .	107
5.18.2. Clase TLinea . . . . .	107
5.18.3. Clase TVector . . . . .	108
5.18.4. Clase TCalendario . . . . .	108
<b>5.19. Respuesta a los ejercicios de autoevaluación . . . . .</b>	<b>109</b>
<b>5.20. Respuesta a los ejercicios de programación . . . . .</b>	<b>111</b>
5.20.1. Clase TCoordenada . . . . .	111
5.20.2. Clase TLinea . . . . .	112
5.20.3. Clase TVector . . . . .	112
5.20.4. Clase TCalendario . . . . .	115

---

## 5.1. Introducción

En este capítulo se introduce la sobrecarga de los operadores del lenguaje C++. Cada operador del lenguaje tiene asociada una representación en forma de función. Mediante la sobrecarga de funciones que se explicó en la Sección 3.1, se puede sobre cargar la función asociada a un operador para que se pueda invocar con diferentes parámetros.

Antes de ver la sobrecarga de operadores, veremos el empleo del puntero `this`, el uso del modificador `const` y el paso de parámetros a una función por referencia, ya que estos tres conceptos son esenciales en la sobrecarga de operadores.

## 5.2. Puntero `this`

Cada objeto tiene acceso a su propia dirección de memoria mediante un puntero llamado `this` (palabra reservada de C++). Este puntero no es parte del propio objeto, sino que se pasa de forma implícita como un parámetro oculto a cada una de las llamadas a las funciones miembro de los objetos. Por tanto, no forma parte del propio objeto y no aumenta su tamaño en memoria.

Este puntero sólo se puede usar dentro de las funciones miembro de un objeto. Este puntero puede ser desreferenciado con el operador `*` como cualquier otro puntero para acceder al objeto. Por ejemplo, en el siguiente código se accede a los datos miembros del objeto de tipo `TCoordenada` directamente (uso implícito del puntero `this`), explícitamente mediante el puntero `this` y el operador flecha `->` y explícitamente mediante el puntero `this` y el operador punto `.` (los paréntesis son necesarios porque el operador punto tiene un nivel de precedencia mayor que el operador de desreferencia):

---

Ejemplo 5.1

---

```

1 TCoordenada::TCoordenada(int a, int b, int c) {
2     x = a;

```

---

```

3     this->y = b;
4     (*this).z = c;
5 }
```

---

Normalmente, el puntero `this` no se emplea ya que no es necesario para acceder a los datos o funciones miembro de un objeto. Los principales usos del puntero `this` son:

- En una función miembro, devolver una instancia del propio objeto.
- Evitar que un objeto se asigne a sí mismo (autoasignación), tal como veremos en la sobrecarga del operador asignación (ver Sección 5.9).
- Permitir la llamada en cascada de varias funciones miembro sobre un mismo objeto, como veremos a continuación con la sobrecarga de algunos operadores.

El puntero `this` es de sólo lectura y no se puede modificar. Por ejemplo, en el siguiente fragmento de código perteneciente a la clase `TCoordenada`, se modifica el valor de `this` en el constructor de copia:

---

#### Ejemplo 5.2

---

```

1
2 TCoordenada::TCoordenada(const TCoordenada &c) {
3     x = c.x;
4     y = c.y;
5     z = c.z;
6     this = &c;
7 }
```

---

El compilador detecta que se está modificando un valor que no puede estar en la parte izquierda de una asignación (no es modificable) y muestra el siguiente mensaje de error:

---

#### Salida ejemplo 5.2

---

```

1 tcoordenada.cc: In copy constructor 'TCoordenada::TCoordenada(
2         const TCoordenada&)' :
3 tcoordenada.cc:17: non-lvalue in assignment
```

---

### 5.3. Modificador `const`

El modificador `const` permite al programador definir **constants**, valores que no pueden cambiar una vez han sido declarados. Normalmente, este modificador se emplea en las siguientes situaciones:

- Para crear constantes simbólicas, como constantes numéricas o de literales.
- Para indicar que un argumento de una función no se va a modificar.

Una vez creado un objeto como constante, no se puede modificar su valor. Por tanto, una variable u objeto declarado como constante se tiene que inicializar en su declaración.

Un objeto constante no puede ser modificado. Por tanto, no se pueden invocar sobre un objeto métodos que lo modifiquen. ¿Cómo sabe el compilador qué métodos se pueden invocar sobre un objeto constante? Un método se puede declarar como constante para indicar que únicamente consulta el objeto, pero no lo modifica. Los métodos constantes sí que se puede invocar sobre objetos constantes.

Para declarar un método de una clase como constante, se tiene que añadir el modificador `const` después de la lista de parámetros del método en los ficheros `.h` y `.cc`. Por ejemplo:

```

1 // En .h
2 void algo() const;
3
4 // En .cc
5 void
6 UnaClase::algo() const
7 {
8     ...
9 }
```

Desde un método constante de una clase únicamente se pueden invocar otros métodos que también sean constantes. De este modo, se asegura que el objeto sobre el que se invoca el método no va a ser cambiado. Para ello, el compilador verifica que no se modifica ningún miembro de dato de la clase (esta verificación depende de las capacidades de cada compilador). Por ejemplo, la siguiente clase contiene un método declarado como constante, pero que modifica un miembro de dato:

```

1 // En el .h
2 class UnaClase
3 {
4     public:
5         UnaClase();
6         void Modifica() const;
7
8     private:
9         int a;
10    };
11
12 // En el .cc
```

```

13 UnaClase::UnaClase()
14 {
15     a = 0;
16 }
17
18 void
19 UnaClase::Modifica() const
20 {
21     a++;
22 }
```

Al compilar esta clase, se genera el siguiente mensaje de error, ya que si el método es `const` no puede modificar el objeto:

Salida ejemplo 5.2

```

1 unaclase.cc: In member function ‘void UnaClase::Modifica() const’:
2 unaclase.cc:23: increment of data-member ‘UnaClase::a’ in read-only
3 structure
```

¿Qué funciones miembro se tienen que declarar como constantes? En principio, todas las que se pueda, es decir, todas las que no modifiquen el objeto, ya que declarar una función miembro como constante “no hace daño”: una función miembro constante se puede invocar sobre objetos tanto constantes como no constantes.

## 5.4. Paso por referencia

Una **referencia** es un nombre alternativo para una variable u objeto. El principal uso de las referencias es la especificación de argumentos y valores de retorno de funciones en general y de los operadores sobrecargados en particular.

El **iniciador de una referencia** tiene que ser un valor del cual se pueda obtener su dirección. De una constante o de un objeto temporal no se puede obtener su dirección. En esos casos se puede indicar con `const` que se quiere crear una referencia a una constante. Por ejemplo:

```

1 // Error: no se puede obtener la referencia
2 int& a = 1;
3 // Correcto
4 const int& a = 1;
```

En el primer caso, el compilador genera el siguiente mensaje de error:

Salida ejemplo 5.2

```

1 int.cc: In function ‘int main()’:
2 int.cc:20: initialization of non-const reference type ‘int&’ from rvalue
3 of type ‘int’
```

En el último caso, se emplea una variable temporal “invisible” para almacenar el valor constante:

1. Primero, se crea una variable temporal de tipo `int`.
2. Después, el valor 1 se almacena en la variable temporal.
3. Por último, se utiliza esa variable temporal como iniciador de la referencia.

Una variable temporal creada para almacenar el iniciador de una referencia perdura hasta el final del ámbito de su referencia.

Si una función tiene un argumento por referencia, no se podrá invocar con un argumento constante o con un valor de retorno por valor. Por ejemplo, el siguiente código produce un error al compilarse:

---

#### Ejemplo 5.3

---

```
1 #include <iostream>
2
3 using namespace std;
4
5 int
6 algo()
7 {
8     int a;
9     // ...
10    return a;
11 }
12
13 void
14 otra(int& a)
15 {
16     // ...
17 }
18
19 int
20 main(void)
21 {
22     // Error
23     otra(algo());
24 }
```

---

El mensaje de error que genera el compilador es:

## Salida ejemplo 5.3

```

1 int.cc: In function 'int main()':
2 int.cc:23: could not convert 'algo()()' to 'int&'
3 int.cc:15: in passing argument 1 of 'void otra(int&)',
```

Tal como indica el mensaje de error, el compilador no puede convertir el valor de retorno de la función `algo()`, que es de tipo `int` a `int&` que es el tipo del parámetro de la función `otra()`. Este error se resuelve definiendo la referencia como constante:

```

1 void
2 otra(const int& a)
3 {
4     ...
5 }
```

## 5.5. Sobre carga de operadores

C++ no permite la creación de nuevos operadores, pero si permite la sobre carga de la mayoría de los operadores que posee. De este modo, los operadores sobre cargados se pueden utilizar con objetos y tener el significado apropiado para esos objetos.

En C++ algunas operadores ya están sobre cargados. Por ejemplo, el operador `+` está sobre cargado con variables de tipo `int`, `float`, `double` y con punteros.

En vez de sobre cargar un operador, se podría crear una función para representar la operación asociada. Pero la notación de operadores es más clara y común para los programadores que el uso de funciones explícitas.

Los operadores se sobre cargan definiendo una función especial cuyo nombre es la palabra reservada `operator` seguida por el símbolo del operador que se va a sobre cargar. Por ejemplo:

```

1 ... operator+(...) {
2     ...
3 }
4 ...
5 ... operator=(...) {
6
7 }
```

Cuando se emplea un operador sobre cargado, el compilador invoca automáticamente a la función asociada que lo define. Por ejemplo:

```

1 TCoordenada a, b, c;
2
3 a = b + c;
```

```

4 | // Equivale a:
5 | a.operator=(b.operator+(c));

```

## 5.6. Restricciones al sobrecargar un operador

Se tienen que respetar las siguientes restricciones:

- La precedencia de un operador no se puede modificar. Si se necesita una precedencia distinta, se pueden utilizar paréntesis en las expresiones para forzar un orden de evaluación de los operadores sobrecargados distinta.
- La asociatividad de un operador (izquierda o derecha) no se puede modificar.
- No es posible modificar el número de operandos de un operador.
- No es posible modificar el comportamiento de un operador con tipos predefinidos del lenguaje. Por ejemplo, no es posible modificar la manera en que el operador suma + realiza la suma de dos enteros.
- Los operadores abreviados de asignación (+=, -=, etc.) se tienen que sobrecargar independientemente. Por ejemplo, sobrecargar el operador suma + y el operador asignación = no implica que el operador += se encuentre sobrecargado.

Además, existen restricciones respecto a los operadores que pueden o no pueden sobrecargarse. Los siguientes operadores se pueden sobrecargar<sup>1</sup>:

+	-	*	/	%	^	&	
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	=	<<	>>	>>=
<<=	==	!=	<=	>=	&&		++
--	->*	,	->	[]	()	new	delete
new[]	delete[]						

Los siguientes operadores no se pueden sobrecargar:

.

.\*

::

?:

## 5.7. ¿Función miembro o función no miembro?

La sobrecarga de los operadores se puede realizar mediante funciones miembro o funciones no miembro (sólo los operadores () , [] , -> y cualquiera de los operadores de asignación requieren que se realice como funciones miembro). Entonces, ¿cuál de las dos opciones elegir?:

---

<sup>1</sup>Para más información sobre el significado de cada operador, consultar el Apéndice B.

- En funciones miembro el operando de la izquierda siempre es un objeto de la clase:  
`TCoordenada a; a + 3; // a.operator+(3);`
- En funciones no miembro, el operando de la izquierda puede ser de un tipo distinto a la clase. Por ejemplo:  
`TCoordenada a; 3 + a; // operator+(3, a);`
- Las funciones no miembro, si necesitan acceder a la parte privada de la clase, se implementan como funciones amigas (`friend`) por razones de rendimiento.

## 5.8. Consejos

- La versión `friend` siempre tendrá un parámetro más:  
`TCoordenada a; 3 + a; // operator+(3, a);`
- Se debe respetar el significado original de los operadores para no confundir al usuario. Por ejemplo, no emplear el operador suma para implementar una resta y vice versa.
- Cuando se modifica al objeto (operando de la izquierda) (ej: `a = b`):
  - Se almacena el resultado en el propio objeto.
  - Se devuelve el propio objeto para permitir su posterior uso en expresiones complejas:  
`return *this; // a = b = c`
  - Se devuelve referencia al objeto (no es obligatorio pero sí más rápido).
- Cuando no se modifica al objeto (operando de la izquierda) (ej: `a + b`):
  - Se crea un objeto temporal que almacena el resultado de la operación.
  - Se devuelve el objeto temporal como resultado de la operación:  
`return temp;`
  - Se devuelve el objeto temporal por valor, nunca por referencia ya que es memoria local.
- El parámetro de la función se declara como referencia (ahorro de tiempo).
- Se devuelve una referencia cuando el operador sobrecargado se quiere utilizar como parte izquierda de una expresión.

## 5.9. Operador asignación

El operador de asignación = puede utilizarse con cada clase sin sobrecargarse. Pero el comportamiento predeterminado del operador de asignación es realizar una asignación de los datos miembro a miembro (*bit a bit*), al igual que el constructor de copia predeterminado. Este comportamiento puede ocasionar problemas cuando un objeto emplea memoria dinámica. Por tanto, se recomienda siempre sobrecargar el operador asignación para que copie correctamente los objetos de una clase.

Por ejemplo, el siguiente código muestra la sobrecarga del operador asignación para la clase TCoordenada:

---

Ejemplo 5.4

---

```

1 // En el .h
2 TCoordenada& operator=(TCoordenada &);
3
4
5 // En el .cc
6 // Asignación: a = b
7 TCoordenada&
8 TCoordenada::operator=(TCoordenada &op2) {
9     (*this).~TCoordenada();
10
11    x = op2.x;
12    y = op2.y;
13    z = op2.z;
14
15    return *this;
16 }
```

---

En el código anterior, vemos que el parámetro de la sobrecarga del operador asignación se pasa por referencia para evitar que se invoque el constructor de copia<sup>2</sup>. Además, antes de realizar la asignación, se llama al destructor del objeto para que se liberen todos los recursos que pueda tener en uso, como por ejemplo, memoria dinámica.

El siguiente ejemplo permite verificar el funcionamiento de la sobrecarga del operador asignación:

---

Ejemplo 5.5

---

```

1 #include <iostream>
2
3 using namespace std;
```

---

<sup>2</sup> El parámetro también se debería de definir como constante, tal como veremos en la sobrecarga de los operadores aritméticos en la Sección 5.11.

```
4
5 #include "tcoordenada.h"
6
7 int main(void) {
8     TCoordenada p1(1, 2, 3), p2(4, 5, 6);
9
10    p1.Imprimir();
11    cout << endl;
12    p2.Imprimir();
13    cout << endl;
14
15    cout << "p1 = p2" << endl;
16    p1 = p2;
17
18    p1.Imprimir();
19    cout << endl;
20    p2.Imprimir();
21    cout << endl;
22
23    return 0;
24 }
```

El código anterior produce como salida:

Salida ejemplo 5.5

```
1 (1, 2, 3)
2 (4, 5, 6)
3 p1 = p2
4 (4, 5, 6)
5 (4, 5, 6)
```

Pero, ¿qué pasaría si se estuviese asignando a un objeto el propio objeto? El siguiente fichero de prueba comprueba esa situación:

Ejemplo 5.6

```
1 #include <iostream>
2
3 using namespace std;
4
5 #include "tcoordenada.h"
6
7 int main(void) {
8     TCoordenada p1(1, 2, 3);
9
```

```

10    p1.Imprimir();
11    cout << endl;
12
13    cout << "p1 = p1" << endl;
14    p1 = p1;
15
16    p1.Imprimir();
17    cout << endl;
18
19    return 0;
20 }

```

El código anterior produce como salida:

Salida ejemplo 5.6

```

1 (1, 2, 3)
2 p1 = p1
3 (0, 0, 0)

```

Cuando se produzca esta situación, que se conoce como **autoasignación**, la sobrecarga del operador asignación no tiene que hacer nada. Para detectar la autoasignación, el mejor método es comparar las direcciones de memoria de los dos objetos, tal como se muestra en el siguiente código:

Ejemplo 5.7

```

1 TCoordenada&
2 TCoordenada::operator=(TCoordenada &op2) {
3     if(this != &op2)
4     {
5         (*this).~TCoordenada();
6
7         x = op2.x;
8         y = op2.y;
9         z = op2.z;
10    }
11
12    return *this;
13 }

```

Como curiosidad, se puede impedir que un objeto de una clase se asigne a otro objeto de la misma clase. Para ello, el operador asignación se tiene que declarar en la parte privada de la clase.

En resumen, los pasos que debe realizar el operador asignación son:

1. Proteger contra la autoasignación.
2. Eliminar la memoria que tenga en uso el objeto de la parte izquierda de la asignación.
3. Inicializar el objeto y reserva memoria para los nuevos elementos.
4. Copiar los elementos nuevos del objeto de la parte derecha al objeto de la parte izquierda.

## 5.10. Constructor de copia y operador asignación

En la mayoría de las clases, el constructor de copia y la sobrecarga del operador asignación realizan una función similar. Por ello, comparten muchas líneas de código que se pueden aislar en una función auxiliar que sea invocada por ambos. Por ejemplo, si observamos el código del constructor de copia y del operador asignación de TCoordenada, vemos que hay tres líneas repetidas:

---

Ejemplo 5.8

---

```
1 // Constructor de copia
2 TCoordenada::TCoordenada(const TCoordenada &c) {
3     x = c.x;
4     y = c.y;
5     z = c.z;
6 }
7
8 // Sobre carga del operador asignación
9 TCoordenada&
10 TCoordenada::operator=(TCoordenada &op2) {
11     if(this != &op2)
12     {
13         (*this).~TCoordenada();
14
15         x = op2.x;
16         y = op2.y;
17         z = op2.z;
18     }
19
20     return *this;
21 }
```

---

El código común del constructor de copia y el operador asignación lo podemos aislar en una función auxiliar que llamamos `Copia(const TCoordenada &)` y que definimos en la parte privada de la clase `TCoordenada` en `tcoordenada.h`:

---

Ejemplo 5.9

---

```
1 void Copia(const TCoordenada &);
```

---

Y realizamos las siguientes modificaciones en **tcoordenada.cc**:

---

Ejemplo 5.10

---

```
1 // Constructor de copia
2 TCoordenada::TCoordenada(const TCoordenada &c) {
3     Copia(c);
4 }
5
6 // Sobrecarga del operador asignación
7 TCoordenada&
8 TCoordenada::operator=(TCoordenada &op2) {
9     if(this != &op2)
10    {
11        (*this).~TCoordenada();
12
13        Copia(op2);
14    }
15
16    return *this;
17 }
18
19 void
20 TCoordenada::Copia(const TCoordenada &c) {
21     x = c.x;
22     y = c.y;
23     z = c.z;
24 }
```

---

## 5.11. Operadores aritméticos

Normalmente, los operadores aritméticos no modifican el objeto sobre el que actúa. Por tanto, se necesita un objeto temporal para realizar el cálculo. Por ejemplo, la sobrecarga del operador suma para dos objetos de tipo **TCoordenada** se puede definir como:

---

Ejemplo 5.11

---

```
1 // En el .h
2 TCoordenada operator+(TCoordenada &);
```

---

```
5 // En el .cc
6 // Suma: a + b
7 TCoordenada
8 TCoordenada::operator+(TCoordenada &op2) {
9     TCoordenada temp;
10
11     temp.x = x + op2.x;
12     temp.y = y + op2.y;
13     temp.z = z + op2.z;
14
15     return temp;
16 }
```

---

Y la sobrecarga del operador resta para dos objetos de tipo TCoordenada:

---

Ejemplo 5.12

---

```
1 // En el .h
2 TCoordenada operator-(TCoordenada &);
3
4
5 // En el .cc
6 // Resta: a - b
7 TCoordenada
8 TCoordenada::operator-(TCoordenada &op2) {
9     TCoordenada temp;
10
11     temp.x = x - op2.x;
12     temp.y = y - op2.y;
13     temp.z = z - op2.z;
14
15     return temp;
16 }
```

---

En la sobrecarga de los operadores es muy importante llevar cuidado con el orden en el que se realizan las operaciones. En el operador anterior, no se obtendría el mismo resultado si el código fuese:

```
1 |     temp.x = op2.x - x;
2 |     temp.y = op2.y - y;
3 |     temp.z = op2.z - z;
```

El siguiente ejemplo comprueba la sobrecarga del operador suma y el operador resta:

## Ejemplo 5.13

```
1 #include <iostream>
2
3 using namespace std;
4
5 #include "tcoordenada.h"
6
7 int main(void) {
8     TCoordenada p1(1, 2, 3), p2(4, 5, 6), p3(7, 8, 9);
9
10    p1.Imprimir();
11    cout << endl;
12    p2.Imprimir();
13    cout << endl;
14    p3.Imprimir();
15    cout << endl;
16
17    cout << "p1 = p2 + p3" << endl;
18    p1 = p2 + p3;
19
20    p1.Imprimir();
21    cout << endl;
22    p2.Imprimir();
23    cout << endl;
24    p3.Imprimir();
25    cout << endl;
26
27    cout << "p1 = p2 - p3" << endl;
28    p1 = p2 - p3;
29
30    p1.Imprimir();
31    cout << endl;
32    p2.Imprimir();
33    cout << endl;
34    p3.Imprimir();
35    cout << endl;
36
37    return 0;
38 }
```

---

Si se compila el código anterior, se produce el siguiente mensaje de error:

## Salida ejemplo 5.13

```
1 main.cc: In function 'int main()':
2 main.cc:18: no match for 'TCoordenada& = TCoordenada' operator
3 tcoordenada.h:12: candidates are: TCoordenada&
```

```
4     TCoordenada::operator=(TCoordenada&)
5 main.cc:28: no match for 'TCoordenada& = TCoordenada' operator
6 tcoordenada.h:12: candidates are: TCoordenada&
7     TCoordenada::operator=(TCoordenada&)
```

El compilador indica que hay un error en las líneas 18 (`p1 = p2 + p3;`) y 28 (`p1 = p2 - p3;`). ¿Por qué? Por el operador asignación espera un parámetro pasado por referencia, pero los operadores suma y resta devuelven su resultado por valor, a partir del cual no se puede obtener una referencia. Este problema se resuelve si se define el parámetro del operador asignación como constante, tal como se ha explicado en la Sección 5.4.

Siempre que se pueda, es mejor definir los parámetros de las funciones como `const`. Por tanto, en nuestro ejemplo, la clase `TCoordenada` queda definida en `tcoordenada.h` como:

---

Ejemplo 5.14

---

```
1 class TCoordenada {
2     public:
3         TCoordenada();
4         TCoordenada(int, int, int);
5         TCoordenada(const TCoordenada &);
6         ~TCoordenada();
7
8         TCoordenada& operator=(const TCoordenada &);
9         TCoordenada operator+(const TCoordenada &);
10        TCoordenada operator-(const TCoordenada &);
11
12        void setX(int);
13        void setY(int);
14        void setZ(int);
15
16        int getX(void);
17        int getY(void);
18        int getZ(void);
19
20        void Imprimir(void);
21
22    private:
23        int x, y, z;
24 }
```

---

Y el fichero `tcoordenada.cc` también se tiene que modificar para que la sobrecarga de los operadores incluya el modificador `const`.

Una vez corregido y compilado el código, la salida que genera el programa de prueba es:

Salida ejemplo 5.14

```

1  (1, 2, 3)
2  (4, 5, 6)
3  (7, 8, 9)
4  p1 = p2 + p3
5  (11, 13, 15)
6  (4, 5, 6)
7  (7, 8, 9)
8  p1 = p2 - p3
9  (-3, -3, -3)
10 (4, 5, 6)
11 (7, 8, 9)

```

Recordemos que cuando se emplea un operador sobrecargado, se invoca automáticamente a la función que define la sobrecarga. Por tanto, en el ejemplo anterior, las instrucciones:

```

1 |     p1 = p2 + p3;
2 |
3 |     p1 = p2 - p3;

```

equivalen a las siguientes instrucciones:

```

1 |     p1.operator=(p2.operator+(p3));
2 |
3 |     p1.operator=(p2.operator-(p3));

```

## 5.12. Operadores de incremento y decremento

La sobrecarga de los operadores de incremento y decremento con sus distintas formas (pre y post) presenta un problema. ¿Cómo puede el compilador distinguir entre la versión pre y la versión post? La solución que se adoptó en C++ es que la versión post reciba un parámetro ficticio que permita al compilador distinguirla de la versión pre. El valor de este parámetro ficticio es irrelevante y no aporta ninguna información.

Por ejemplo, la sobrecarga del operador preincremento de la clase TCoordenada es:

Ejemplo 5.15

```

1 // En el .h
2 TCoordenada& operator++(void);

```

```
3
4
5 // En el .cc
6 // Preincremento: ++a
7 TCoordenada&
8 TCoordenada::operator++(void) {
9     x++;
10    y++;
11    z++;
12
13    return *this;
14 }
```

---

En el caso del operador post se necesita un objeto temporal para almacenar una copia del objeto, ya que el incremento se debe de realizar después de haber utilizado el objeto en la expresión donde aparezca. Por ejemplo, la sobrecarga del operador postincremento de la clase TCoordenada es:

---

Ejemplo 5.16

---

```
1 // En el .h
2 TCoordenada operator++(int);
3
4
5 // En el .cc
6 // Postincremento: a++
7 TCoordenada
8 TCoordenada::operator++(int op2) {
9     TCoordenada temp(*this);
10
11    x++;
12    y++;
13    z++;
14
15    return temp;
16 }
```

---

Como el operador postincremento devuelve un objeto temporal, se devuelve por valor y no por referencia como en el caso del operador preincremento.

### 5.13. Operadores abreviados

También se pueden sobrecargar los métodos abreviados del operador asignación combinado con un operador aritmético: `+=`, `-=`, `*=`, etc. En este caso, el resultado de

la operación se almacena en el propio objeto (el objeto se modifica), por lo que no es necesario un objeto temporal para realizar el cálculo.

Por ejemplo, el siguiente código muestra la sobrecarga del operador `+=` y `-=` para la clase `TCoordenada`:

---

Ejemplo 5.17

---

```

1 // En el .h
2 TCoordenada& operator+=(const TCoordenada &);
3 TCoordenada& operator-=(const TCoordenada &);

4

5
6 // En el .cc
7 // a += b
8 TCoordenada&
9 TCoordenada::operator+=(const TCoordenada &op2) {
10    x += op2.x;
11    y += op2.y;
12    z += op2.z;

13
14    return *this;
15 }
16
17 // a -= b
18 TCoordenada&
19 TCoordenada::operator-=(const TCoordenada &op2) {
20    x -= op2.x;
21    y -= op2.y;
22    z -= op2.z;

23
24    return *this;
25 }
```

---

## 5.14. Operadores de comparación

La sobrecarga de los operadores de comparación (`==`, `!=`, `<`, `>`, `<=` y `>=`) tiene que devolver un valor booleano. Por ejemplo, el siguiente código muestra la sobrecarga del operador igualdad (`==`) para la clase `TCoordenada`:

---

Ejemplo 5.18

---

```

1 // En el .h
2 bool operator==(const TCoordenada &);
3
4
5 // En el .cc
```

---

```

6 // a == b
7 bool
8 TCoordenada::operator==(const TCoordenada &op2) {
9     bool temp;
10    temp = (x==op2.x && y==op2.y && z==op2.z) ? true : false;
11
12    return temp;
13 }
14 }
```

---

Como los operadores de comparación están relacionados entre sí, se pueden implementar unos a partir de otros. Por ejemplo, el operador desigualdad ( $\neq$ ) se puede definir como la negación del operador igualdad ( $\equiv$ ):

---

Ejemplo 5.19

---

```

1 // En el .h
2 bool operator!=(const TCoordenada &);
3
4
5 // En el .cc
6 // a != b
7 bool
8 TCoordenada::operator!=(const TCoordenada &op2) {
9     return !(*this == op2);
10 }
```

---

## 5.15. Operadores de entrada y salida

La sobrecarga de los operadores de entrada ( $>>$ ) y salida ( $<<$ ) se tiene que realizar obligatoriamente como funciones amigas, porque el operando izquierdo es un objeto de una clase diferente (`istream` y `ostream`).

Por ejemplo, el siguiente código muestra la sobrecarga del operador de entrada para la clase `TCoordenada`:

---

Ejemplo 5.20

---

```

1 // En el .h
2 friend istream& operator>>(istream &, TCoordenada &);
3
4
5 // En el .cc
6 // cin >> a
7 istream&
8 operator>>(istream &s, TCoordenada &obj) {
```

---

```

9   cout << "Introducir coordenada x:";
10  s >> obj.x;
11
12  cout << "Introducir coordenada y:";
13  s >> obj.y;
14
15  cout << "Introducir coordenada z:";
16  s >> obj.z;
17
18  return s;
19 }
```

---

Y la sobrecarga para el operador de salida de la clase TCoordenada que sustituye al método `Imprimir()` empleado hasta ahora:

---

Ejemplo 5.21

---

```

1 // En el .h
2 friend ostream& operator<<(ostream &, const TCoordenada &);
3
4
5 // En el .cc
6 // cout << a
7 ostream&
8 operator<<(ostream &s, const TCoordenada &obj) {
9   s << "(" << obj.x << ", ";
10  s << obj.y << ", ";
11  s << obj.z << ")";
12
13  return s;
14 }
```

---

En ambos casos, se tiene que devolver el objeto de flujo de entrada o salida que se recibe para que expresiones como la siguiente funcionen correctamente:

1	TCoordenada a, b;
2	
3	cout << "Objeto a: " << a << " Objeto b: << b << endl;

---

## 5.16. Operador corchete

El operador corchete es especial, porque unas veces se emplea para leer y otras veces para escribir (modificar) el objeto. Como podemos tener objetos constantes que no se pueden modificar, necesitamos dos sobrecargas del operador corchete. La primera sobrecarga permite tanto la lectura como escritura sobre objetos no constantes,

mientras que la segunda sobrecarga sólo permite la lectura y se emplea con los objetos constantes. Como en el primer caso se permite la escritura, se tiene que devolver una referencia al valor que se va a leer/escribir, mientras que en el segundo caso se retorna por valor, ya que sólo se permite la lectura.

La sobrecarga del operador corchete debe de comprobar que la posición que recibe como argumento se encuentra en el rango correcto. De no ser así, estamos ante una situación de error, y se puede optar por varias estrategias:

- Olvidarse de la comprobación del rango y devolver el contenido de la “supuesta” posición en el vector (comportamiento de C++ sobre los vectores de tipos básicos: no comprueba nada, es misión del programador asegurarse que una posición en un vector es correcta).
- Generar un mensaje de error y detener la ejecución del programa.
- Devolver un valor que indique una situación errónea. En la primera sobrecarga del operador corchete, como se tiene que devolver una referencia, hace falta un miembro de la parte privada de la clase que se emplea en las situaciones de error en las que hay que devolver una referencia.

Por ejemplo, para la clase **TCoordenada**, hemos definido la sobrecarga del operador corchete de forma que permite acceder directamente a cada una de las componentes. Para ello, el operador corchete recibe como índice un carácter que indica la componente que se desea acceder. La sobrecarga del operador corchete realiza la misma función que las funciones **setX(int)**, **setY(int)**, **setZ(int)** y **getX()**, **getY()**, **getZ()** que hemos empleado hasta ahora para acceder a las componentes de una coordenada.

En el fichero **tcoordenada.h** añadimos a la declaración de la clase la sobrecarga del operador corchete:

---

Ejemplo 5.22

---

```
1 int& operator[](char);  
2 int operator[](char) const;
```

---

Tal como se ha explicado previamente, se tiene que definir la sobrecarga dos veces, una para lectura/escritura y la otra para sólo lectura. Además, tenemos que añadir a la parte privada de la clase un dato miembro para devolver cuando el índice no sea correcto:

---

Ejemplo 5.23

---

```
1 int error;
```

---

En el fichero **tcoordenada.cc** añadimos el código de la función que sobrecarga el operador corchete:

## Ejemplo 5.24

```

1 int&
2 TCoordenada::operator[](char c) {
3     if(c == 'x' || c == 'X')
4         return x;
5     if(c == 'y' || c == 'Y')
6         return y;
7     if(c == 'z' || c == 'Z')
8         return z;
9
10    error = 0;
11    return error;
12 }
13
14 int
15 TCoordenada::operator[](char c) const {
16     if(c == 'x' || c == 'X')
17         return x;
18     if(c == 'y' || c == 'Y')
19         return y;
20     if(c == 'z' || c == 'Z')
21         return z;
22
23     return 0;
24 }
```

Por último, a continuación incluimos un fichero de ejemplo para probar la sobre-carga del operador corchete:

## Ejemplo 5.25

```

1 #include <iostream>
2
3 using namespace std;
4
5 #include "tcoordenada.h"
6
7 int main(void) {
8     TCoordenada p1(1, 2, 3);
9
10    cout << p1 << endl;
11
12    p1['x'] = 4;
13    p1['y'] = 5;
14    p1['z'] = 6;
15
```

```
16     cout << p1 << endl;
17
18     cout << "x: " << p1['x'] << endl;
19     cout << "y: " << p1['y'] << endl;
20     cout << "z: " << p1['z'] << endl;
21
22     return 0;
23 }
```

La salida que genera el código anterior es:

Salida ejemplo 5.25

```
1 (1, 2, 3)
2 (4, 5, 6)
3 x: 4
4 y: 5
5 z: 6
```

## 5.17. Ejercicios de autoevaluación

1. Respecto el puntero this:

- a) Sólo se puede usar dentro de las funciones miembro de la clase
- b) Es un puntero al propio objeto donde se emplea
- c) Se pasa de forma implícita como un parámetro oculto cuando se invoca una función miembro
- d) Todas las respuestas son correctas

2. Cuando se emplea el paso por referencia para un parámetro de una función:

- a) Se invoca al constructor de copia
- b) Se invoca al constructor de copia sólo si el parámetro también se ha definido como const
- c) No se invoca al constructor de copia
- d) No se invoca al constructor de copia sólo si el parámetro también se ha definido como const

3. Respecto a la sobrecarga de los operadores:

- a) En las funciones no miembro, el operando de la izquierda no puede ser de un tipo distinto a la clase

- b) En las funciones miembro, no se puede acceder a la parte privada de la clase y se tienen que declarar como friend
- c) En las funciones miembro, el operando de la izquierda no puede ser un objeto de la clase
- d) Las anteriores respuestas no son correctas
4. Respecto al constructor de copia y la sobrecarga del operador asignación:
- a) Son equivalentes
- b) Si uno no existe, se invoca automáticamente al otro en su lugar
- c) Tienen que definirse con el mismo código
- d) Las anteriores respuestas no son correctas
5. Para que la sobrecarga del operador corchete pueda aparecer a ambos lados de una asignación, tiene que definirse como:
- a) TItem operator[](int);
- b) TItem& operator[](int);
- c) const TItem operator[](int);
- d) Las anteriores respuestas no son correctas
6. El prototipo TCoordenada& operator++(void) corresponde a la sobrecarga del operador:
- a) Preincremento
- b) Postincremento
- c) Cualquiera de los dos
- d) Las anteriores respuestas no son correctas
7. La sentencia TCoordenada a=b; invoca a:
- a) El operador asignación
- b) El constructor por defecto
- c) El constructor de copia
- d) Es una sentencia incorrecta
8. ¿Cuál es la sobrecarga correcta?:
- a) void operator=(const TCoordenada &);
- b) TCoordenada operator=(const TCoordenada &);
- c) TCoordenada& operator=(const TCoordenada &);

- d) Las anteriores respuestas no son correctas
9. Respecto a la sobrecarga del operador incremento:
- operator++(void) define la sobrecarga del operador preincremento
  - operator++(void) define la sobrecarga del operador postincremento
  - No se pueden sobrecargar el operador preincremento y postincremento a la vez para una misma clase
  - Las anteriores respuestas no son correctas
10. Respecto a la sobrecarga de los operadores:
- Una vez definida la sobrecarga del operador de comparación (==), el compilador define automáticamente el operador desigualdad (!=)
  - La sobrecarga de los operadores de entrada (>>) y salida (<<) se puede realizar como función miembro o no miembro
  - Si el programador no define la sobrecarga del operador asignación =, el compilador proporciona automáticamente uno
  - Las anteriores respuestas no son correctas

## 5.18. Ejercicios de programación

### 5.18.1. Clase TCoordenada

Dada la clase **TCoordenada** que representa puntos en el espacio, definid en C++ las siguientes sobrecargas de operadores:

- Operador +: suma de un objeto coordenada y un número entero por la derecha. El número entero se suma a las tres componentes del objeto coordenada.
- Operador +: suma de un objeto coordenada y un número entero por la izquierda. El número entero se suma a las tres componentes del objeto coordenada.

### 5.18.2. Clase TLinea

Dada la clase **TLinea** que representa una línea definida mediante dos puntos representados por dos objetos de tipo **TCoordenada**, definid en C++ la siguiente sobrecarga de operador salida:

- Operador <<: muestra los dos objetos **TCoordenada** que definen la línea, separados por una coma y encerrados entre paréntesis. Por ejemplo:  
((1, 2, 3), (4, 5, 6))

### 5.18.3. Clase TVector

Dada la clase **TVector** que contiene un vector dinámico de números enteros y un número entero que contiene la dimensión del vector, definid en C++ las siguientes sobrecargas de operadores:

- Operador **+**: suma de dos vectores componente a componente, tiene que comprobar que los dos vectores tienen la misma dimensión, en caso contrario devolverá un vector vacío de dimensión 0. Esta sobrecarga sustituye a la función **Suma(TVector, TVector)** propuesta en el capítulo anterior.
- Operador **-**: resta de dos vectores componente a componente, tiene que comprobar que los dos vectores tienen la misma dimensión, en caso contrario devolverá un vector vacío de dimensión 0. Esta sobrecarga sustituye a la función **Resta(TVector, TVector)** propuesta en el capítulo anterior.
- Operador **=**: asignación de vectores.
- Operador **[]**: acceso a las componentes del vector, los índices tienen que empezar desde 1.
- Operador **==**: dos vectores son iguales si poseen la misma dimensión y los mismos valores en todas las posiciones.
- Operador **!=**: dos vectores son distintos si no poseen la misma dimensión o en alguna posición posee valores distintos.
- Operador **<<**: muestra el contenido del vector componente a componente, separadas por un espacio en blanco y todo el vector encerrado entre corchetes. Por ejemplo:  
[3 1 16 5]

### 5.18.4. Clase TCalendario

Dada la clase **TCalendario** que contiene una fecha (representada mediante tres variables enteras para el día, mes y año) y un mensaje (representado mediante un vector dinámico de caracteres), definid en C++ las siguientes sobrecargas de operadores:

- Operador **++** (preincremento y postincremento): suma un día a la fecha.
- Operador **--** (predecremento y postdecremento): resta un día a la fecha.
- Operador **=**: asignación de fechas.
- Operador **==**: dos objetos son iguales si poseen la misma fecha y el mismo mensaje.

- Operador !=: dos objetos son distintos si no poseen la misma fecha o el mismo mensaje.
- Operador <<: muestra la fecha con el formato estándar d/m/a y el mensaje.

## 5.19. Respuesta a los ejercicios de autoevaluación

1. Respecto el puntero this:
  - a) Sólo se puede usar dentro de las funciones miembro de la clase
  - b) Es un puntero al propio objeto donde se emplea
  - c) Se pasa de forma implícita como un parámetro oculto cuando se invoca una función miembro
  - d) **(✓) Todas las respuestas son correctas**
2. Cuando se emplea el paso por referencia para un parámetro de una función:
  - a) Se invoca al constructor de copia
  - b) Se invoca al constructor de copia sólo si el parámetro también se ha definido como const
  - c) **(✓) No se invoca al constructor de copia**
  - d) No se invoca al constructor de copia sólo si el parámetro también se ha definido como const
3. Respecto a la sobrecarga de los operadores:
  - a) En las funciones no miembro, el operando de la izquierda no puede ser de un tipo distinto a la clase
  - b) En las funciones miembro, no se puede acceder a la parte privada de la clase y se tienen que declarar como friend
  - c) En las funciones miembro, el operando de la izquierda no puede ser un objeto de la clase
  - d) **(✓) Las anteriores respuestas no son correctas**
4. Respecto al constructor de copia y la sobrecarga del operador asignación:
  - a) Son equivalentes
  - b) Si uno no existe, se invoca automáticamente al otro en su lugar
  - c) Tienen que definirse con el mismo código
  - d) **(✓) Las anteriores respuestas no son correctas**

- 
5. Para que la sobrecarga del operador corchete pueda aparecer a ambos lados de una asignación, tiene que definirse como:
- a) TItem operator[](int);
  - b) **(✓) TItem& operator[](int);**
  - c) const TItem operator[](int);
  - d) Las anteriores respuestas no son correctas
6. El prototipo TCoordenada& operator++(void) corresponde a la sobrecarga del operador:
- a) **(✓) Preincremento**
  - b) Postincremento
  - c) Cualquiera de los dos
  - d) Las anteriores respuestas no son correctas
7. La sentencia TCoordenada a=b; invoca a:
- a) El operador asignación
  - b) El constructor por defecto
  - c) **(✓) El constructor de copia**
  - d) Es una sentencia incorrecta
8. ¿Cuál es la sobrecarga correcta?:
- a) void operator=(const TCoordenada &);
  - b) TCoordenada operator=(const TCoordenada &);
  - c) **(✓) TCoordenada& operator=(const TCoordenada &);**
  - d) Las anteriores respuestas no son correctas
9. Respecto a la sobrecarga del operador incremento:
- a) **(✓) operator++(void) define la sobrecarga del operador preincremento**
  - b) operator++(void) define la sobrecarga del operador postincremento
  - c) No se pueden sobrecargar el operador preincremento y postincremento a la vez para una misma clase
  - d) Las anteriores respuestas no son correctas
10. Respecto a la sobrecarga de los operadores:

- a) Una vez definida la sobrecarga del operador de comparación (==), el compilador define automáticamente el operador desigualdad (!=)
- b) La sobrecarga de los operadores de entrada (>>) y salida (<<) se puede realizar como función miembro o no miembro
- c) **(✓) Si el programador no define la sobrecarga del operador asignación =, el compilador proporciona automáticamente uno**
- d) Las anteriores respuestas no son correctas

## 5.20. Respuesta a los ejercicios de programación

### 5.20.1. Clase TCoordenada

La sobrecarga del operador + cuando se suma el número entero por la derecha se puede realizar como función miembro o función no miembro declarada como función amiga. Sin embargo, cuando se suma por la izquierda, sólo se puede realizar como función no miembro.

En `tcoordenada.h`:

---

#### Ejemplo 5.26

---

```

1 TCoordenada operator+(int);
2
3 friend TCoordenada operator+(int, const TCoordenada &);
```

---

En `tcoordenada.cc`:

---

#### Ejemplo 5.27

---

```

1 TCoordenada
2 TCoordenada::operator+(int n) {
3     TCoordenada temp(*this);
4
5     temp.x += n;
6     temp.y += n;
7     temp.z += n;
8
9     return temp;
10 }
11
12 TCoordenada
13 operator+(int n, const TCoordenada &obj) {
14     TCoordenada temp(n, n, n);
15
16     return temp + obj;
17 }
```

La implementación de estas dos sobrecargas se puede realizar de varias formas. En la primera sobrecarga, se ha optado por realizar la suma del número entero componente a componente. En la segunda sobrecarga, se crea un objeto TCoordenada inicializado con el número que se desea sumar y se aprovecha la sobrecarga del operador suma para dos objetos de tipo TCoordenada.

### 5.20.2. Clase TLinea

En tlinea.h:

Ejemplo 5.28

```
1 friend ostream& operator<<(ostream &, const TLinea &);
```

---

En tlinea.cc:

Ejemplo 5.29

```
1 ostream&
2 operator<<(ostream &s, const TLinea &obj) {
3     s << "(" << obj.p1 << ", ";
4     s << obj.p2 << ")";
5
6     return s;
7 }
```

---

### 5.20.3. Clase TVector

En tvector.h:

Ejemplo 5.30

```
1 friend ostream & operator<<(ostream &, const TVector &);
2
3 TVector operator+(const TVector &);
4 TVector operator-(const TVector &);
5 TVector& operator=(const TVector &);
6 int& operator[](int);
7 bool operator==(const TVector &);
8 bool operator!=(const TVector &);
```

---

En tvector.cc:

---

Ejemplo 5.31

---

```
1 TVector
2 TVector::operator+(const TVector &v) {
3     if(dimension != v.dimension)
4     {
5         TVector temp(0);
6
7         return temp;
8     }
9     else
10    {
11        TVector temp(dimension);
12
13        for(int i = 0; i < dimension; i++)
14            temp.datos[i] = datos[i] + v.datos[i];
15
16        return temp;
17    }
18 }
19
20 TVector
21 TVector::operator-(const TVector &v) {
22     if(dimension != v.dimension)
23     {
24         TVector temp(0);
25
26         return temp;
27     }
28     else
29     {
30        TVector temp(dimension);
31
32        for(int i = 0; i < dimension; i++)
33            temp.datos[i] = datos[i] - v.datos[i];
34
35        return temp;
36    }
37 }
38
39 TVector&
40 TVector::operator=(const TVector &v) {
41     if(this != &v)
42     {
43         if(dimension != v.dimension)
44         {
45             dimension = v.dimension;
```

```
46     if(datos != NULL)
47         delete datos;
48     datos = new int[dimension];
49     if(datos == NULL)
50     {
51         dimension = 0;
52         return *this;
53     }
54 }
55 for(int i = 0; i < dimension; i++)
56     datos[i] = v.datos[i];
57 }
58
59     return *this;
60 }
61
62 int&
63 TVector::operator[](int indice) {
64     if(indice >= 1 && indice <= dimension)
65         return datos[indice - 1];
66
67     error = -1;
68
69     return error;
70 }
71
72 bool
73 TVector::operator==(const TVector &v) {
74     if(dimension != v.dimension)
75         return false;
76     else
77         for(int i = 0; i < dimension; i++)
78             if(datos[i] != v.datos[i])
79                 return false;
80
81     return true;
82 }
83
84 bool
85 TVector::operator!=(const TVector &v) {
86     return !(*this == v);
87 }
88
89 ostream&
90 operator<<(ostream &oo, const TVector &cc)
91 {
```

```
92     oo << "[";  
93  
94     if(cc.dimension > 0)  
95         oo << cc.datos[0];  
96     for(int i = 1; i < cc.dimension; i++)  
97         oo << " " << cc.datos[i];  
98     oo << "]";  
99  
100    return oo;  
101 }
```

---

#### 5.20.4. Clase TCalendario

En `tcalendario.h`:

---

Ejemplo 5.32

---

```
1 friend ostream& operator<<(ostream &, const TCalendario &);  
2  
3 TCalendario& operator++(void);  
4 TCalendario operator++(int);  
5 TCalendario& operator--(void);  
6 TCalendario operator--(int);  
7 TCalendario& operator=(const TCalendario &);  
8 bool operator==(const TCalendario &);  
9 bool operator!=(const TCalendario &);
```

---

En `tcalendario.cc`:

---

Ejemplo 5.33

---

```
1 TCalendario&  
2 TCalendario::operator++(void) {  
3     dia++;  
4     if(mes == 2 && dia == 29 && !Bisiesto(*this))  
5     {  
6         dia = 1;  
7         mes = 3;  
8     }  
9     else if((mes == 1 || mes == 3 || mes == 5 || mes == 7 ||  
10        mes == 8 || mes == 10 || mes == 12) && dia == 32)  
11     {  
12         dia = 1;  
13         mes++;  
14         if(mes == 13)  
15         {
```

```
16     mes = 1;
17     anyo++;
18 }
19 }
20 else if(dia == 31)
21 {
22     dia = 1;
23     mes++;
24 }
25
26 return *this;
27 }

28
29 TCalendario
30 TCalendario::operator++(int a) {
31     TCalendario temp(*this);
32
33     dia++;
34     if(mes == 2 && dia == 29 && !Bisiesto(*this))
35     {
36         dia = 1;
37         mes = 3;
38     }
39     else if((mes == 1 || mes == 3 || mes == 5 || mes == 7 ||
40             mes == 8 || mes == 10 || mes == 12) && dia == 32)
41     {
42         dia = 1;
43         mes++;
44         if(mes == 13)
45         {
46             mes = 1;
47             anyo++;
48         }
49     }
50     else if(dia == 31)
51     {
52         dia = 1;
53         mes++;
54     }
55
56     return temp;
57 }

58
59 TCalendario&
60 TCalendario::operator--(void) {
61     dia--;
```

```
62     if(dia == 0)
63     {
64         mes--;
65         if(mes == 0)
66         {
67             mes = 12;
68             anyo--;
69         }
70         if(mes == 2 && !Bisiesto(*this))
71             dia = 28;
72         else if(mes == 2 && Bisiesto(*this))
73             dia = 29;
74         else if(mes == 1 || mes == 3 || mes == 5 || mes == 7 ||
75                 mes == 8 || mes == 10 || mes == 12)
76             dia = 31;
77         else
78             dia = 30;
79     }
80
81     return *this;
82 }
83
84 TCalendario
85 TCalendario::operator--(int a) {
86     TCalendario temp(*this);
87
88     dia--;
89     if(dia == 0)
90     {
91         mes--;
92         if(mes == 0)
93         {
94             mes = 12;
95             anyo--;
96         }
97         if(mes == 2 && !Bisiesto(*this))
98             dia = 28;
99         else if(mes == 2 && Bisiesto(*this))
100            dia = 29;
101        else if(mes == 1 || mes == 3 || mes == 5 || mes == 7 ||
102                  mes == 8 || mes == 10 || mes == 12)
103            dia = 31;
104        else
105            dia = 30;
106    }
107 }
```

```
108     return temp;
109 }
110
111 TCalendario&
112 TCalendario::operator=(const TCalendario &cal) {
113     dia = cal.dia;
114     mes = cal.mes;
115     anyo = cal.anyo;
116     if(mensaje != NULL)
117         delete mensaje;
118     if(cal.mensaje == NULL)
119         mensaje == NULL;
120     else
121         mensaje = new char[strlen(cal.mensaje) + 1];
122     if(mensaje == NULL)
123         return *this;
124     strcpy(mensaje, cal.mensaje);
125     // También funciona
126     // strdup(mensaje, cal.mensaje);
127
128     return *this;
129 }
130
131 bool
132 TCalendario::operator==(const TCalendario &cal) {
133     if(dia == cal.dia && mes == cal.mes && anyo == cal.anyo &&
134         !strcmp(mensaje, cal.mensaje))
135         return true;
136
137     return false;
138 }
139
140 bool
141 TCalendario::operator!=(const TCalendario &cal) {
142     return !(*this == cal);
143 }
144
145 ostream&
146 operator<<(ostream &os, const TCalendario &cal) {
147     os << cal.dia << "/" << cal.mes << "/" << cal.anyo << ":" ;
148     if(cal.mensaje != NULL)
149         os << cal.mensaje;
150
151     return os;
152 }
```

---

# Capítulo 6

# Composición y herencia

En este capítulo se introducen dos tipos de relaciones entre objetos: la relación “tiene-un” (composición) y “es-un” (herencia). Estos dos mecanismos básicos de la programación orientada a objetos permiten la reutilización de código, con las ventajas que ello conlleva.

## Índice General

---

6.1. Introducción . . . . .	119
6.2. Composición . . . . .	120
6.3. Inicialización de los objetos miembro . . . . .	122
6.4. Herencia . . . . .	126
6.5. Ejercicios de autoevaluación . . . . .	129
6.6. Ejercicios de programación . . . . .	130
6.6.1. Clase TLinea . . . . .	130
6.6.2. Clase TCoordenadaV . . . . .	131
6.6.3. Clase TAgenda . . . . .	131
6.7. Respuesta a los ejercicios de autoevaluación . . . . .	132

---

## 6.1. Introducción

La composición y herencia son dos mecanismos de abstracción de la programación orientada a objetos que permiten que el programador comparta código común entre distintos objetos del sistema. La reutilización de código ahorra costes (tiempo y dinero) durante el desarrollo de un programa. Además, ayuda a mejorar la calidad del

código, reducir el número de errores y aumentar la portabilidad y mantenibilidad del código.

La composición representa una relación “tiene-un” (*has-a*). En este tipo de relación, un objeto contiene uno o más objetos de otras clases como miembros y hace uso de las funciones públicas que proporciona. Por ejemplo, una clase que representa un cliente puede contener un objeto de la clase dirección.

La herencia representa una relación “es-un” (*is-a*). En este tipo de relación, una clase llamada derivada (o subclase) es una especialización de una o varias clases base (o superclase) y un objeto de un clase derivada puede tratarse como un objeto de sus clases base. Una clase derivada hereda los componentes (datos y funciones miembro) de su clase base, pero además puede añadir los suyos propios. Por ejemplo, una clase que representa a un estudiante puede derivar de una clase base que representa a una persona en general. De la clase base persona hereda sus datos y funciones miembro, como nombre, apellidos y fecha de nacimiento, pero además puede añadir los suyos propios como expediente y curso.

## 6.2. Composición

En la **composición**, una clase tiene objetos de otras clases como datos miembros. Desde la clase compuesta, la clase que contiene los objetos miembro, sólo se puede acceder a la parte pública de los objetos que contiene. La composición es una forma de reutilización de código. Una clase puede contener objetos de otras clases por valor o por referencia (mediante punteros). En C++, la composición por valor se conoce como ***layering***.

Recordemos que cuando se crea un objeto, se invoca automáticamente a alguno de sus constructores, según los parámetros que se indiquen en el momento de su creación. Si un objeto contiene objetos de otras clases, ¿cómo y cuando se construyen los objetos que contiene? ¿Cómo se le pueden pasar parámetros?

Para indicar cómo construir un objeto miembro se emplea un **inicializador de miembro**. Los objetos miembro se construyen en el mismo orden en que se declararon en la definición de la clase y antes de que se ejecute el código del constructor del objeto que los contiene.

El orden de invocación de los destructores es el inverso al de inicialización: primero se ejecuta el código del destructor del objeto compuesto y a continuación se ejecutan los destructores de los objetos miembro. Si se invoca directamente al destructor del objeto compuesto, también se invocan los destructores de los objetos miembro, tal como se muestra en el siguiente código de ejemplo donde se definen dos clases llamadas A y B:

---

Ejemplo 6.1

---

```
1 #include <iostream>
```

<sup>2</sup>

```
3 using namespace std;
4
5 class A {
6     public:
7     A() {
8         cout << "A()" << endl;
9     }
10
11    ~A() {
12        cout << "~A()" << endl;
13    }
14 };
15
16 class B {
17     public:
18     B() {
19         cout << "B()" << endl;
20     }
21
22    ~B() {
23        cout << "~B()" << endl;
24    }
25
26     private:
27     A a;
28 };
29
30 int main(void) {
31     B b;
32
33     cout << "Llama destructor" << endl;
34     b.~B();
35     cout << "Fin" << endl;
36
37     return 0;
38 }
```

El código anterior produce como salida:

Salida ejemplo 6.1

```
1     A()
2     B()
3     Llama destructor
4     ~B()
5     ~A()
6     Fin
```

```

7  ~B()
8  ~A()

```

En la salida anterior podemos comprobar el orden de ejecución de los constructores y destructores. Cuando se crea el objeto `b` de la clase `B`, primero se ejecuta el código del constructor por defecto del objeto `a` que contiene y a continuación el código de su propio constructor. Sin embargo, cuando se invoca el destructor, el orden es inverso: primero se ejecuta el código del destructor del propio objeto y a continuación el destructor `~A()` para el objeto `a` que contiene.

## 6.3. Inicialización de los objetos miembro

Los inicializadores de miembro sólo se pueden emplear en los constructores. Aparecen entre la lista de parámetros de un constructor y la llave izquierda (`{`) que indica el comienzo del código del constructor. La lista de inicializadores de miembro se separa de la lista de parámetros del constructor mediante dos puntos (`:`). Los inicializadores se separan entre sí mediante comas (uno para cada objeto que contenga). En el inicializador se indica el nombre del objeto y los valores iniciales que se le pasan, no se indica el nombre de la clase o del constructor: el compilador sabe a qué clase pertenece cada objeto y según los valores iniciales que se indiquen, se invocará al constructor apropiado. Un objeto miembro no necesita inicializarse siempre a través de un inicializador: si no se proporciona un inicializador para un objeto, se invoca automáticamente al constructor por defecto.

En el Capítulo 4 se introdujo la clase `TLinea`, que representa una línea en el espacio definida por dos objetos de tipo `TCoordenada`, tal como se puede observar en la declaración de la clase que se almacena en el fichero `tlinea.h`<sup>1</sup>:

---

Ejemplo 6.2

---

```

1 #ifndef __TLINEA__
2 #define __TLINEA__
3
4 #include <iostream>
5
6 using namespace std;
7
8 #include "tcoordenada.h"
9
10 class TLinea {
11     friend ostream& operator<<(ostream &, const TLinea &);
12
13 public:

```

<sup>1</sup>Se ha añadido la sobrecarga del operador salida respecto al código que aparece en el Capítulo 4.

```
14     TLinea();
15     TLinea(const TCoordenada &, const TCoordenada &);
16     TLinea(const TLinea &);
17     ~TLinea();
18     float Longitud(void);
19
20     private:
21         TCoordenada p1, p2;
22     };
23
24 #endif
```

La clase **TLinea** define una línea mediante dos puntos representados por dos objetos **p1** y **p2** de tipo **TCoordenada**. El código de la clase se almacena en el fichero **tlinea.cc**:

---

Ejemplo 6.3

---

```
1 #include "tlinea.h"
2
3 TLinea::TLinea() {
4     p1.x = 0;
5     p1.y = 0;
6     p1.z = 0;
7
8     p2.x = 0;
9     p2.y = 0;
10    p2.z = 0;
11 }
12
13 TLinea::TLinea(const TCoordenada & a, const TCoordenada & b) {
14     p1.x = a.x;
15     p1.y = a.y;
16     p1.z = a.z;
17
18     p2.x = b.x;
19     p2.y = b.y;
20     p2.z = b.z;
21 }
22
23 TLinea::TLinea(const TLinea & l) { copiar los datos
24     p1.x = l.p1.x;
25     p1.y = l.p1.y;
26     p1.z = l.p1.z;
27
28     p2.x = l.p2.x;
```

```

29     p2.y = l.p2.y;
30     p2.z = l.p2.z;
31 }
32
33 TLinea::~TLinea() {
34     p1.x = 0;
35     p1.y = 0;
36     p1.z = 0;
37
38     p2.x = 0;
39     p2.y = 0;
40     p2.z = 0;
41 }
42
43 float
44 TLinea::Longitud(void) {
45     return Distancia(p1, p2);
46 }
47
48 ostream&
49 operator<<(ostream &s, const TLinea &obj) {
50     s << "(" << obj.p1 << ", ";
51     s << obj.p2 << ")";
52
53     return s;
54 }
```

---

El constructor `TLinea(const TCoordenada &, const TCoordenada &)` construye un objeto línea a partir de los dos puntos que lo definen, mientras que el constructor de copia `TLinea(const TLinea &)` construye un objeto línea a partir de otro objeto línea. En ambos casos, los objetos `p1` y `p2` se construyen mediante una copia de sus datos miembro a miembro. Sin embargo, no es necesario realizar una copia de los objetos miembro a miembro, se puede construir un objeto `TCoordenada` a partir de otro objeto del mismo tipo mediante su constructor de copia. ¿Cómo lo podemos indicar en el código anterior?

En el siguiente código, se emplean los inicializadores de miembro para que los objetos `p1` y `p2` se construyan mediante su constructor de copia:

---

Ejemplo 6.4

---

```

1 TLinea::TLinea() {
2     // No hace nada
3 }
4
5 TLinea::TLinea(const TCoordenada & a, const TCoordenada & b): p1(a), p2(b) {
6     // No hace nada
```

```
7 }
8
9 TLinea::TLinea(const TLinea & l): p1(l.p1), p2(l.p2) {
10 // No hace nada
11 }
12
13 TLinea::~TLinea() {
14 // No hace nada
15 }
```

---

En el constructor por defecto de `TLinea` no se indica ningún inicializador. En ese caso, se invoca automáticamente al constructor por defecto para los objetos `p1` y `p2`.

En el caso del destructor, también se ha eliminado el código que contenía porque se invoca automáticamente al destructor para los objetos `p1` y `p2`.

Gracias a los inicializadores de miembro, el código necesario es menor y más sencillo de entender. Además, se produce una mejoría en el rendimiento del código, ya que evitamos unas invocaciones innecesarias a unas funciones miembro de `TCoordenada`. ¿Qué nos hemos ahorrado?

Un posible programa que usa la clase `TLinea` es:

---

#### Ejemplo 6.5

---

```
1 #include <iostream>
2
3 using namespace std;
4
5 #include "tcoordenada.h"
6 #include "tlinea.h"
7
8 int
9 main(void)
10 {
11     TCoordenada p1;
12     TCoordenada p2(0, 3, 4);
13     TCoordenada p3(p2);
14
15     TLinea l1, l2(p1, p2);
16
17     cout << l1 << endl;
18     cout << l1.Longitud() << endl;
19     cout << l2 << endl;
20     cout << l2.Longitud() << endl;
21
22     return 0;
23 }
```

---

El código anterior produce como salida:

```
1 ((0, 0, 0), (0, 0, 0))
2 0
3 ((0, 0, 0), (0, 3, 4))
4 5
```

Salida ejemplo 6.5

## 6.4. Herencia

La **herencia** es una de las principales características de la programación orientada a objetos. La herencia es un mecanismo de programación que permite la reutilización de código. La herencia es un tema complejo y extenso, por lo que en esta sección únicamente se indican algunas de sus características en C++.

La herencia consiste en definir una clase nueva a partir de otra u otras que ya existan. La clase nueva, llamada **clase derivada** o **subclase**, hereda los miembros (datos y funciones) de las clases a partir de las que se definen (la **clase base** o **superclase**). Además, la clase derivada puede ser a su vez clase base de otras clases derivadas, lo que origina una **jerarquía de clases**. De este modo, el programador no se ve obligado a escribir todo el código desde cero.

Cuando una clase se deriva a partir de una única clase base, se habla de **herencia simple**. Cuando una clase derivada hereda de múltiples clases base se denomina **herencia múltiple**.

En C++ existen tres tipos de herencia: pública (**public**), protegida (**protected**) y privada (**private**).

En la **herencia pública**, cada objeto de la clase derivada es también un objeto de la clase base de dicha clase derivada; sin embargo, al revés no se cumple: los objetos de la clase base no son objetos de sus clases derivadas. En este tipo de herencia, los miembros de la clase base conservan su nivel de acceso (visibilidad): los miembros **public** de la clase base quedan como **public** en la clase derivada, los miembros **protected** quedan como **protected** en la clase derivada y los miembros **private** quedan como **private**.

La **herencia protegida** rara vez se emplea. En este tipo de herencia, los miembros **public** y **protected** de la clase base se convierten en miembros **protected** de la clase derivada.

La **herencia privada** se puede emplear como una alternativa a la composición. En este tipo de herencia, todos los miembros de la clase base quedan como **private** en la clase derivada y, por ello, no son accesibles desde fuera de la clase derivada. Por tanto, la herencia protegida y la herencia privada no son relaciones del tipo “es-un” (*is-a*), sino más bien del tipo “tiene-un” (*has-a*).

Desde las clases derivadas no es posible acceder a la parte privada de las clases base, pero sí que se puede acceder a la parte pública y protegida: los miembros protegidos son privados para el exterior de la clase, pero permiten el acceso a las clases derivadas.

El siguiente código muestra un ejemplo de herencia pública. A partir de la clase **TCoordenada** se ha creado la clase **TCoordenadaV** que se comporta como la clase **TCoordenada** pero añade un dato miembro adicional (**valor**) que representa un valor asociado a cada coordenada (por ejemplo, el potencial eléctrico o la temperatura en un punto del espacio). La definición de la clase se almacena en el fichero **tcoordenadav.h**:

---

Ejemplo 6.6

---

```
1 #ifndef __TCOORDENADAV__
2 #define __TCOORDENADAV__
3
4 #include "tcoordenada.h"
5
6 class TCoordenadaV: public TCoordenada {
7     friend ostream& operator<<(ostream &, const TCoordenadaV &);
8
9     public:
10         TCoordenadaV();
11         TCoordenadaV(int, int, int);
12         TCoordenadaV(int, int, int, int);
13         TCoordenadaV(const TCoordenadaV &);
14         ~TCoordenadaV();
15
16     private:
17         int valor;
18 };
19
20#endif
```

---

El código de la clase se almacena en el fichero **tcoordenadav.cc**:

---

Ejemplo 6.7

---

```
1 #include "tcoordenadav.h"
2
3 TCoordenadaV::TCoordenadaV(): TCoordenada() {
4     valor = 0;
5 }
6
7 TCoordenadaV::TCoordenadaV(int a, int b, int c): TCoordenada(a, b, c) {
8     valor = 0;
9 }
10
11 TCoordenadaV::TCoordenadaV(int a, int b, int c, int v):
```

```

12         TCoordenada(a, b, c) {
13     valor = v;
14 }
15
16 TCoordenadaV::TCoordenadaV(const TCoordenadaV &obj): TCoordenada(obj) {
17     valor = obj.valor;
18 }
19
20 TCoordenadaV::~TCoordenadaV() {
21     // Se invoca automáticamente al destructor de TCoordenada
22     valor = 0;
23 }
24
25 ostream&
26 operator<<(ostream &s, const TCoordenadaV &obj) {
27     s << (TCoordenada) obj << ": " << obj.valor;
28
29     return s;
30 }
```

---

En el código anterior, podemos observar en los cuatro constructores de la clase derivada `TCoordenadaV` como se invoca a un constructor concreto de la clase base `TCoordenada` en cada caso. Si no se especifica un constructor de la clase base, el compilador invoca al constructor por defecto.

La sobrecarga del operador salida muestra una característica importante de la herencia: mediante una **conversión de tipo (casting)**, indicamos que un objeto `TCoordenadaV` se puede tratar como un objeto `TCoordenada` y, por tanto, se puede emplear la sobrecarga del operador salida de `TCoordenada` para visualizarlo.

A continuación incluimos un fichero de prueba para la clase `TCoordenadaV`:

---

#### Ejemplo 6.8

---

```

1 #include <iostream>
2
3 using namespace std;
4
5 #include "tcoordenada.h"
6 #include "tcoordenadav.h"
7
8 int
9 main(void)
10 {
11     TCoordenadaV p1;
12     TCoordenadaV p2(10, 20, 30);
13     TCoordenadaV p3(40, 50, 60, -100);
14     TCoordenadaV p4(p3);
```

```
15     cout << p1 << endl;
16     cout << p2 << endl;
17     cout << p3 << endl;
18     cout << p4 << endl;
19
20
21     return 0;
22 }
```

El código anterior produce como salida:

Salida ejemplo 6.8

```
1 (0, 0, 0): 0
2 (10, 20, 30): 0
3 (40, 50, 60): -100
4 (40, 50, 60): -100
```

## 6.5. Ejercicios de autoevaluación

1. Respecto al layering o composición:

- a) Los métodos de la clase compuesta pueden acceder a la parte pública de la clase base
- b) Los métodos de la clase compuesta pueden acceder a la parte pública y protegida de la clase base
- c) Los métodos de la clase compuesta pueden acceder a la parte pública, protegida y privada de la clase base
- d) Las anteriores respuestas no son correctas

2. Respecto al layering o composición:

- a) Los objetos miembro se construyen antes que el objeto que los contiene
- b) Los objetos miembro se construyen después que el objeto que los contiene
- c) Los objetos miembro se construyen al mismo tiempo que el objeto que los contiene
- d) Las anteriores respuestas no son correctas

3. Respecto a la herencia:

- a) La parte privada no es heredada por las clases derivadas

- b) La parte protegida de una clase sólo es accesible por los métodos de la propia clase
- c) Una relación “tiene-un” se implementa mediante la herencia pública
- d) Las anteriores respuestas no son correctas

4. Respecto a la herencia:

- a) En la herencia pública, la parte privada de la clase base es accesible desde los métodos de la clase derivada.
- b) En la herencia protegida, la parte privada de la clase base es accesible desde los métodos de la clase derivada.
- c) En la herencia privada, la parte privada de la clase base es accesible desde los métodos de la clase derivada.
- d) Las anteriores respuestas no son correctas

5. Respecto a la herencia:

- a) Si definimos un método en la parte privada de una clase, éste será accesible desde los métodos de la propia clase y desde todas sus clases derivadas
- b) Un método público de la clase derivada es accesible desde los métodos de la clase base
- c) Los métodos de la clase derivada pueden acceder a la parte pública de la clase base
- d) Las anteriores respuestas no son correctas

## 6.6. Ejercicios de programación

### 6.6.1. Clase TLinea

Comprobar cómo se invocan los constructores y destructores en cada situación. Para ello, modificar las clases **TCoordenada** y **TLinea**: colocar instrucciones como la siguiente en cada constructor y destructor de las dos clases:

```

1  TCoordenada::TCoordenada(const TCoordenada & c) {
2      cout << "Constructor de copia de TCoordenada" << endl;
3      x = c.x;
4      y = c.y;
5      z = c.z;
6  }
7
8  TLinea::~TLinea() {
9      // No hace nada
10     cout << "Destructor de TLinea" << endl;
11 }
```

¿Cuántas mensajes aparecen cuando se construye un objeto **TLinea**? ¿Y cuando se destruye?

### 6.6.2. Clase **TCoordenadaV**

Comprobar cómo se invocan los constructores y destructores en cada situación. Para ello, modificar las clases **TCoordenada** y **TCoordenadaV**: colocar instrucciones como la siguiente en cada constructor y destructor de las dos clases:

```
1 | TCoordenadaV::TCoordenadaV(): TCoordenada() {
2 |     cout << "Constructor por defecto de TCoordenadaV" << endl;
3 |     valor = 0;
4 | }
```

¿Cuántas mensajes aparecen cuando se construye un objeto **TCoordenadaV**? ¿Y cuando se destruye?

### 6.6.3. Clase **TAgenda**

Dada la clase **TCalendario** que contiene una fecha (representada mediante tres variables enteras para el día, mes y año) y un mensaje (representado mediante un vector dinámico de caracteres), definid en C++ la clase **TAgenda** que representa una agenda de citas para todos los días de un año. La clase **TAgenda** contiene un array de objetos **TCalendario**, uno para cada día del año (se tiene que definir un array de 366 posiciones, para poder almacenar un año bisiesto). Se pide:

- Constructor por defecto (año 2006).
- Constructor a partir de un año.
- Constructor de copia.
- Destructor.
- Operador +: une las citas de dos agendas del mismo año en una agenda nueva. Si en un mismo día hay dos citas, se concatenan los mensajes de las dos citas con un guión de separación.
- Operador =: asignación de agendas.
- Operador ==: devuelve cierto si dos agendas contienen las mismas citas y falso en caso contrario.
- Operador !=: devuelve cierto si dos agendas no contienen las mismas citas y falso en caso contrario.

- Operador [int]: accede a un día de la agenda para lectura y escritura mediante un índice que representa el número de día en el año.
- Operador [TCalendario]: accede a un día de la agenda para lectura y escritura.
- Operador <<: muestra la agenda, un día por línea.

## 6.7. Respuesta a los ejercicios de autoevaluación

1. Respecto al layering o composición:

- a) (✓) Los métodos de la clase compuesta pueden acceder a la parte pública de la clase base
- b) Los métodos de la clase compuesta pueden acceder a la parte pública y protegida de la clase base
- c) Los métodos de la clase compuesta pueden acceder a la parte pública, protegida y privada de la clase base
- d) Las anteriores respuestas no son correctas

2. Respecto al layering o composición:

- a) (✓) Los objetos miembro se construyen antes que el objeto que los contiene
- b) Los objetos miembro se construyen después que el objeto que los contiene
- c) Los objetos miembro se construyen al mismo tiempo que el objeto que los contiene
- d) Las anteriores respuestas no son correctas

3. Respecto a la herencia:

- a) La parte privada no es heredada por las clases derivadas
- b) La parte protegida de una clase sólo es accesible por los métodos de la propia clase
- c) Una relación “tiene-un” se implementa mediante la herencia pública
- d) (✓) Las anteriores respuestas no son correctas

4. Respecto a la herencia:

- a) En la herencia pública, la parte privada de la clase base es accesible desde los métodos de la clase derivada.
- b) En la herencia protegida, la parte privada de la clase base es accesible desde los métodos de la clase derivada.

- c) En la herencia privada, la parte privada de la clase base es accesible desde los métodos de la clase derivada.
  - d) **(✓) Las anteriores respuestas no son correctas**
5. Respecto a la herencia:
- a) Si definimos un método en la parte privada de una clase, éste será accesible desde los métodos de la propia clase y desde todas sus clases derivadas
  - b) Un método público de la clase derivada es accesible desde los métodos de la clase base
  - c) **(✓) Los métodos de la clase derivada pueden acceder a la parte pública de la clase base**
  - d) Las anteriores respuestas no son correctas

# Capítulo 7

## Otros temas

En este capítulo se comentan una serie de características del lenguaje de programación C++ que puntuilan algunos aspectos que se han presentado en los capítulos anteriores.

### Índice General

---

7.1. Forma canónica de una clase . . . . .	135
7.2. Funciones de cero parámetros . . . . .	137
7.3. Valores por omisión de una función . . . . .	137
7.4. Funciones inline . . . . .	139

---

### 7.1. Forma canónica de una clase

La **forma canónica** o **forma ortodoxa** de una clase en C++ define el diseño correcto de una clase. Respetar este diseño es importante, ya que permite usar la clase de la misma manera que cualquier tipo básico de C++.

Una clase en su forma canónica tiene que contener, como mínimo, los siguientes métodos<sup>1</sup>:

- Constructor por defecto: crea objetos sin parámetros. Se encarga de inicializar los atributos, reservar recursos (reservar memoria, abrir ficheros, etc.) y otras tareas que preparan al objeto para ser usado.
- Constructor de copia: crea objetos a partir de otros objetos. Se emplea para crear un objeto que es una copia de otro objeto de la misma clase.

<sup>1</sup>Una clase se compone de métodos (también llamados funciones miembro) y de atributos (también llamados miembros de dato o miembros).

- Operador asignación: limpia y copia objetos. Similar al constructor de copia, pero se diferencia en que el objeto sobre el que se realiza la copia ya ha sido creado previamente.
- Destructor: limpieza de objetos. El método complementario a los constructores: libera todos los recursos que posea un objeto.

El constructor por defecto permite crear correctamente un objeto cuando en el momento de crearlo no se inicializa. Si no se proporciona un constructor por defecto, el compilador genera uno. Un constructor por defecto generado por el compilador invoca implícitamente a los constructores por defecto de los miembros de la clase (objetos contenidos por composición o *layering*); aparte de eso, no hace nada más.

El constructor de copia crea un objeto a partir de otro objeto. Este constructor se puede invocar explícita o implícitamente. El primer caso se produce cuando se invoca directamente en la declaración de un objeto. Por ejemplo:

```

1 // Se invoca al constructor por defecto
2 UnaClase a;
3 // Se invoca al constructor de copia
4 UnaClase b(a);
5 // Se invoca al constructor de copia
6 UnaClase c = a; // Equivale a UnaClase c(a);

```

Por otro lado, el constructor de copia se invoca de forma implícita cuando se llama a un método y se pasa como argumento un objeto por valor o cuando un método devuelve un objeto por valor. Por ejemplo, en las siguientes situaciones se invoca el constructor de copia automáticamente para crear el objeto **a** y para crear una copia del objeto **b**:

```

1 UnaClase algo(UnaClase a) {
2     UnaClase b;
3     ...
4     return b;
5 }

```

Si no se proporciona un constructor de copia, el compilador genera uno que realiza una copia miembro a miembro. La copia miembro a miembro tiene resultados no deseados (normalmente desastrosos) en la mayoría de las ocasiones y especialmente cuando se utiliza en una clase con miembros de tipo puntero, ya que si los miembros no poseen un constructor de copia definido, se realiza una copia bit a bit.

El operador asignación permite copiar un objeto ya construido sobre otro ya construido. Otra vez, si el programador no proporciona el operador asignación, el compilador se encarga de generar uno por su cuenta que únicamente realiza una copia miembro a miembro.

Por último, el destructor se encarga de liberar aquellos recursos (memoria, fichero, etc.) que haya podido adquirir un objeto. Los destructores se llaman implícitamente cuando una variable que representa un objeto sale de su ámbito, se borra un objeto de forma explícita (`delete`), etc. Sólo en ocasiones muy poco habituales es necesario que el programador llame al destructor de forma explícita; además, se desaconseja la invocación explícita del destructor.

Si el programador no proporciona un destructor, el compilador genera uno que únicamente invoca a los destructores de los miembros que contiene la clase.

Como se ha visto, el compilador genera de forma automática una serie de funciones miembro en el caso que el programador no las defina. Como las funciones miembro puede ser que generen errores, es importante que el programador siempre las defina. De todos modos, con definirlas no están resueltos todos los problemas: el programador tiene que asignar el comportamiento apropiado a estas funciones miembro.

## 7.2. Funciones de cero parámetros

En C++, una **función de cero parámetros** o **función de lista de parámetros vacía** se especifica mediante la palabra reservada `void` o con paréntesis vacíos. Por ejemplo, el siguiente fragmento de código declara dos funciones de cero parámetros:

```
1 | void AbreFichero();  
2 |  
3 | void CierraFichero(void);
```

Ambas formas son correctas y válidas en C++ e indican que las funciones declaradas `AbreFichero()` y `CierraFichero()` no reciben argumentos.

El significado de la lista de parámetros vacía difiere en C y C++. En C, significa que la verificación de los argumentos no es necesaria (se puede para cualquier argumento que se deseé). Sin embargo, en C++ significa que la función no toma argumentos.

## 7.3. Valores por omisión de una función

C++ nos ofrece la posibilidad de declarar **valores por omisión** para los parámetros de una función. Con esta técnica, también conocida como **argumentos predeterminados** o **parámetros por defecto**, cuando se omite un argumento predeterminado en una llamada a una función, el compilador inserta el valor por defecto de dicho argumento. Los argumentos predeterminados tienen que estar en el extremo derecho de la lista de parámetros de la función.

El uso de valores por omisión puede ocasionar errores: si los valores por omisión de una función se modifican a posteriori, la ejecución del programa podría no ser la

esperada, ya que los valores por omisión que emplea la función no son los especificados inicialmente.

El siguiente ejemplo muestra la función `convertir()` que posee dos parámetros, uno de ellos con un valor por omisión:

---

Ejemplo 7.1

---

```

1 #include <iostream>
2
3 using namespace std;
4
5 // Convierte de euros a pesetas por defecto
6 float
7 convertir(float cantidad, float factor = 166.386)
8 {
9     return cantidad * factor;
10}
11
12 int
13 main(void)
14 {
15     cout << "2 euros --> pesetas: " << convertir(2) << endl;
16
17     cout << "1000 pesetas --> euros: " << convertir(1000, 0.00601) << endl;
18
19     cout << "2 euros --> dólares: " << convertir(2, 1.15) << endl;
20 }
```

---

El código anterior produce como salida:

---

Salida ejemplo 7.1

---

```

1 2 euros --> pesetas: 332.772
2 1000 pesetas --> euros: 6.01
3 2 euros --> dólares: 2.3
```

El efecto de un parámetro por defecto se puede lograr de forma alternativa mediante la sobrecarga de funciones, pero es menos evidente. Por ejemplo, la función `convertir()` del ejemplo anterior se podría reescribir como:

```

1 // Convierte entre dos unidades monetarias
2 float
3 convertir(float cantidad, float factor)
4 {
5     return cantidad * factor;
6 }
```

```
7 // Convierte de euros a pesetas por defecto
8 float
9 convertir(float cantidad)
10 {
11     return convertir(cantidad, 166.386);
12 }
```

## 7.4. Funciones inline

Las llamadas a funciones involucra tiempo de ejecución al llamar y al volver de una función por el cambio de contexto que se produce. C++ proporciona el modificador **inline** que permite reducir el número de llamadas a las funciones al insertar el código de las funciones en el sitio donde son llamadas. Sin embargo, este método presenta dos inconvenientes. Por un lado, al insertar una copia del código de la función, el tamaño del código final aumenta. Por otro lado, aunque una función lleve el modificador **inline**, el compilador puede decidir no aplicarlo; en realidad, este modificador únicamente funciona con las funciones muy sencillas y pequeñas.

Por ejemplo, el siguiente código declara una función que calcula el mayor de dos números enteros como **inline**:

```
1 inline int max(int a, int b) {
2     if(a > b) return a;
3
4     return b;
5 }
```

# Capítulo 8

## Errores más comunes

En este capítulo se recogen los errores más habituales que comete una persona cuando comienza a programar con el lenguaje de programación C++.

### Índice General

---

8.1. Introducción . . . . .	141
8.2. Sobre el fichero makefile y la compilación . . . . .	144
8.3. Sobre las directivas de inclusión . . . . .	146
8.4. Sobre las clases . . . . .	149
8.5. Sobre la sobrecarga de los operadores . . . . .	161
8.6. Sobre la memoria . . . . .	166
8.7. Sobre las cadenas . . . . .	172
8.8. Varios . . . . .	178

---

### 8.1. Introducción

Uno de los aspectos más difíciles de la programación es la depuración de errores. Como dice el aforismo, “errar es de humanos”: los errores de programación son muy comunes incluso en los productos comerciales que se “supone” que se ven sometidos a controles de calidad exhaustivos. Por ello, no es de extrañar que cuando se comience a programar por primera vez o se aprenda un nuevo lenguaje de programación se cometan muchos errores, lo que suele ser origen de muchas frustraciones. Sin embargo, aunque se trata de un aspecto importante de la programación, no se le presta mucha atención tanto en la bibliografía sobre programación (existen pocos libros cuyo tema central sea la depuración de errores) como en la docencia.

En este capítulo se muestran los errores más habituales que comete una persona cuando comienza a programar con el lenguaje de programación C++. Los errores se han clasificado en las siguientes categorías:

- Sobre el fichero makefile y la compilación.
- Sobre las directivas de inclusión.
- Sobre las clases.
- Sobre la sobrecarga de los operadores.
- Sobre la memoria.
- Sobre las cadenas.
- Varios.

Los mensajes de error pueden cambiar de una versión de compilador a otro. Por ejemplo, el siguiente código contiene dos errores, porque se confunde el manejo de objetos y punteros a objetos:

---

#### Ejemplo 8.1

---

```

1 UnaClase a, *b;
2
3 b = &a;
4 cout << a->propiedad;
5 cout << b.propiedad;
```

---

Si se compila con el compilador g++ 2.95.4 de GNU se obtiene el siguiente mensaje de error:

#### Salida ejemplo 8.1

```

1 prueba.cc: In function ‘int main(...):’
2 prueba.cc:12: base operand of ‘->’ has non-pointer type ‘UnaClase’
3 prueba.cc:13: request for member ‘propiedad’ in ‘b’, which is of
4           non-aggregate type ‘UnaClase *’
```

Si se compila con el compilador bcc32 5.5.1 de Borland se obtiene el siguiente mensaje de error:

#### Salida ejemplo 8.1

```

1 Error E2288 prueba.cc 12: Pointer to structure required on left side
2   of -> or ->* in function main()
3 Error E2294 prueba.cc 13: Structure required on left side of . or .*
4   in function main()
```

Como se ha comentado previamente, el formato de los mensajes de error cambia de un compilador a otro, aunque suelen tener un estructura similar. En general, cada mensaje de error consta de:

- El nombre del archivo donde se ha detectado el error.
- El número de línea dentro del archivo donde se ha detectado el error.
- Un número de referencia de un tipo de error.
- Una breve descripción del error.

Por otro lado, el mensaje de error también puede cambiar si previamente han aparecido otros errores durante el proceso de compilación (pueden aparecer errores donde realmente no hay errores o pueden desaparecer errores verdaderos). Por ejemplo, el siguiente fragmento de código contiene un error porque el parámetro de la sobrecarga del operador asignación se ha declarado como `UnaClase` cuando debería ser `UnaClase`:

---

#### Ejemplo 8.2

---

```
1   ...
2   UnaClase&
3   UnaClase::operator=(Unaclase& a)
4   {
5       ...
6   }
7   ...
```

---

produce los siguientes mensajes de error, de los cuales sólo el primero es válido (el resto son falsos errores inducidos por el primer error):

---

#### Salida ejemplo 8.2

---

```
1  unaclase.cc:43: 'Unaclase' was not declared in this scope
2  unaclase.cc:43: 'a' was not declared in this scope
3  unaclase.cc:44: declaration of 'operator =' as non-function
4  unaclase.cc:44: invalid declarator
5  unaclase.cc:44: syntax error before '{'
6  unaclase.cc:46: ANSI C++ forbids declaration 'dim' with no type
7  unaclase.cc:46: 'a' was not declared in this scope
8  unaclase.cc:47: ANSI C++ forbids declaration 'v' with no type
9  unaclase.cc:47: invalid use of member 'UnaClase::dim'
10 unaclase.cc:48: parse error before 'for'
11 unaclase.cc:48: invalid use of member 'UnaClase::dim'
12 unaclase.cc:48: parse error before ';'
13 unaclase.cc:48: syntax error before '++'
```

El error se produce porque **Unaclase** no está definido: cambiando **Unaclase** por **UnaClase** se resuelve el problema y desaparecen todos los mensajes de error. Conclusión: los errores hay que resolverlos desde el principio hasta el final del código, ya que al resolver un error pueden desaparecer mensajes de error “falsos”.

Los errores más difíciles de detectar son los relacionados con el manejo de memoria. En especial los de violación de segmento (**Segmentation fault**), ya que no siempre se produce el mensaje de error (lo que no significa que no exista un error). Como la memoria se reserva por trozos (por ejemplo, de 16 en 16 bytes), una operación incorrecta que invada una zona de memoria no válida puede pasar siempre inadvertida. Otras veces, el mensaje de error se produce después del punto donde existe el error (el error se manifiesta en un punto donde realmente no existe). Para resolver este tipo de errores, lo mejor es emplear un depurador y realizar una traza del código desde el principio.

## 8.2. Sobre el fichero makefile y la compilación

- Compilar código en C++ con el compilador de C (gcc). Ejemplo:

Ejemplo 8.3

```

1 gcc -c unaclase.cc
2 gcc -c prueba.cc
3 gcc -o prueba prueba.o unaclase.o

```

Mensaje de error: En el proceso de compilación no se genera un mensaje de error. En el proceso de enlazado (última ejecución de **gcc**) se genera un mensaje de error porque no se enlazan las librerías con el código de C++.

Salida ejemplo 8.3

```

1 prueba.o: In function ‘main’:
2 prueba.o(.text+0x28): undefined reference to ‘endl(ostream &)
3 prueba.o(.text+0x35): undefined reference to ‘cout’
4 prueba.o(.text+0x3a): undefined reference to ‘ostream::operator<<
5     (char const *)’

```

Solución: Emplear el compilador de C++ (g++).

```

1 g++ -c unaclase.cc
2 g++ -c prueba.cc
3 g++ -o prueba prueba.o unaclase.o

```

- Poner espacios en blanco en vez de emplear tabulador en el fichero **makefile**. Ejemplo: El subrayado (\_) representa un espacio en blanco.

## Ejemplo 8.4

```
1 prueba: prueba.o unaclase.o  
2 -----g++ -o prueba prueba.o unaclase.o
```

Mensaje de error:

## Salida ejemplo 8.4

```
1 makefile:2: *** missing separator (did you mean TAB instead of 8  
2 spaces?). Stop.
```

Solución: Emplear el tabulador en vez de espacios en blanco.

```
1 | prueba: prueba.o unaclase.o  
2 |         g++ -o prueba prueba.o unaclase.o
```

- No existe un fichero de una regla de dependencia (por ejemplo, porque se ha escrito mal el nombre del fichero). Ejemplo:

## Ejemplo 8.5

```
1 prueba.o: prueba.cc UnaClase.h  
2           g++ -c prueba.cc
```

Mensaje de error:

## Salida ejemplo 8.5

```
1 make: *** No rule to make target 'UnaClase.h', needed by 'prueba.o'.  
2 Stop.
```

Solución: Verificar que los nombres de los ficheros están bien escritos y que los ficheros existen.

```
1 | prueba.o: prueba.cc unaclase.h  
2 |         g++ -c prueba.cc
```

- Compilar un fichero cuyo nombre una vez compilado no coincide con el esperado. Ejemplo: El compilador genera el fichero **pruebaa.o**, pero lo que se quiere obtener es el fichero **prueba.o**.

## Ejemplo 8.6

```
1 prueba: prueba.o unaclase.o  
2           g++ -o prueba prueba.o unaclase.o  
3  
4 prueba.o: pruebaa.cc unaclase.h  
5           g++ -c pruebaa.cc
```

Mensaje de error:

Salida ejemplo 8.6

```
1 g++: prueba.o: No such file or directory
```

Solución: Verificar que los nombres de los ficheros están bien escritos y que los ficheros existen.

```
1 | prueba.o: prueba.cc unaclase.h
2 |         g++ -c prueba.cc
```

- Al enlazar, no incluir un fichero necesario. Ejemplo:

Ejemplo 8.7

```
1 g++ -c unaclase.cc
2 g++ -c prueba.cc
3 g++ -o prueba prueba.o
```

Mensaje de error:

Salida ejemplo 8.7

```
1 prueba.o: In function `main':
2 prueba.o(.text+0xe): undefined reference to
3   `UnaClase::UnaClase(void)'
4 prueba.o(.text+0x2f): undefined reference to
5   `UnaClase::~UnaClase(void)'
6 prueba.o(.text+0x4a): undefined reference to
7   `UnaClase::~UnaClase(void)'
```

Solución: Verificar que en el proceso de enlazado se tienen en cuenta todos los ficheros necesarios.

```
1 | g++ -c unaclase.cc
2 | g++ -c prueba.cc
3 | g++ -o prueba prueba.o unaclase.o
```

### 8.3. Sobre las directivas de inclusión

- Los espacios en blanco son significativos dentro de los < > o los " " de una directiva de inclusión. Ejemplo:

Ejemplo 8.8

```
1 #include <iostream >
```

Mensaje de error:

1    unaclase.cc:1: iostream : No such file or directory

Solución: Eliminar los espacios en blanco.

1 | #include <iostream>

- Emplear < > en vez de " " para un archivo de cabecera que no es estándar. Ejemplo:

Ejemplo 8.9

1 #include <unaclase>  
2 // O también  
3 #include <unaclase.h>

Mensaje de error:

1    unaclase.cc:3: unaclase: No such file or directory  
2    // O también  
3    unaclase.cc:3: unaclase.h: No such file or directory

Solución: Emplear " " y poner siempre el nombre completo del fichero de cabecera. Sólo para las cabeceras de la biblioteca estándar se puede emplear indistintamente < > o " " y con o sin .h.

1 | #include "unaclase.h"

- Incluir múltiples veces el mismo fichero de cabecera. Ejemplo:

Ejemplo 8.10

1 #include "unaclase.h"  
2 #include "unaclase.h"

Mensaje de error:

1    In file included from prueba.cc:3:  
2    unaclase.h:5: redefinition of 'class UnaClase'  
3    unaclase.h:16: previous definition here

Solución: Emplear las guardas de inclusión, que evita el problema de la redefinición cuando se incluye el mismo fichero de cabecera múltiples veces. Para evitar conflictos de nombres en las guardas, es conveniente elegir nombres largos y “extraños”. Todas las cabeceras estándar tienen guardas de inclusión, así que no hay que preocuparse en incluirlas varias veces.

```

1 | #ifndef __UNACLASE__
2 | #define __UNACLASE__
3 |
4 | class UnaClase
5 | {
6 |     ...
7 | };
8 |
9 | #endif

```

- Emplear un punto (.) en el nombre de una guarda de inclusión. Ejemplo:

Ejemplo 8.11

```

1 #ifndef __UNACLASE.H__
2 #define __UNACLASE.H__

```

Mensaje de error: No se produce un mensaje de error. Se trata de una advertencia: el identificador definido se trunca hasta el carácter no válido.

Salida ejemplo 8.11

```

1  unaclase.h:1: warning: garbage at end of '#ifndef' argument
2  unaclase.h:2: warning: missing white space after '#define'
   __UNACLASE'

```

Solución: Emplear únicamente los mismos caracteres que se emplean en el nombre de una función o de una variable.

```

1 | #ifndef __UNACLASE__
2 | #define __UNACLASE__

```

- No separar el nombre de una guarda de inclusión de las instrucciones de compilación condicional. Ejemplo:

Ejemplo 8.12

```

1 #ifndef__UNACLASEH__
2 #define__UNACLASEH__

```

Mensaje de error:

## Salida ejemplo 8.12

```

1 In file included from unaclase.cc:3:
2 unaclase.h:31: unbalanced '#endif'
```

Solución: Separar con un espacio en blanco como el nombre de la guarda de inclusión de las instrucciones de compilación condicional.

```

1 #ifndef __UNACLASE__
2 #define __UNACLASE__
```

## 8.4. Sobre las clases

- Emplear el nombre de la clase y el operador de ámbito (::) incorrectamente al definir el código de una función. Ejemplo:

## Ejemplo 8.13

```

1 UnaClase::UnaClase& operator=(UnaClase& a)
2 {
3     ...
4 }
```

Mensaje de error:

## Salida ejemplo 8.13

```

1 unaclase.cc:14: 'operator =(UnaClase &)' must be a nonstatic member
2     function
```

Solución: Primero se tiene que poner el tipo del valor de retorno de la función y a continuación el nombre de la clase y el operador de ámbito.

```

1 UnaClase& UnaClase::operator=(UnaClase& a)
2 {
3     ...
4 }
```

- No colocar el modificador de visibilidad **public:** antes de la declaración de los constructores, el destructor y otras funciones miembro. Ejemplo:

## Ejemplo 8.14

```

1 class UnaClase
2 {
3     UnaClase();
4     ~UnaClase();
```

```

5     ...
6 };

```

Mensaje de error: No se produce un mensaje de error. Se trata de una advertencia: una clase con todas sus funciones miembro privadas normalmente no tiene sentido.

Salida ejemplo 8.14

```

1 In file included from unaclase.cc:1:
2 unaclase.h:7: warning: all member functions in class 'UnaClase' are
3     private

```

Solución: Incluir el modificador de visibilidad **public:** donde haga falta.

```

1 class UnaClase
2 {
3     public:
4         UnaClase();
5         ~UnaClase();
6     ...
7 };

```

- Confundir la declaración de una función amiga (**friend**) con los modificadores de visibilidad. Ejemplo:

Ejemplo 8.15

```

1 class UnaClase
2 {
3     friend:
4         int funcionAmiga(void);
5
6     public:
7         UnaClase();
8         ~UnaClase();
9     ...
10 };

```

Mensaje de error:

Salida ejemplo 8.15

```

1 In file included from unaclase.cc:1:
2 unaclase.h:9: parse error before '('

```

Solución: El modificador **friend** se tiene que poner a cada función que sea amiga.

```

1  class UnaClase
2  {
3      friend int funcionAmiga(void);
4
5      public:
6          UnaClase();
7          ~UnaClase();
8          ...
9  };

```

- Declarar una función miembro y un atributo de una clase con el mismo nombre. Ejemplo:

---

Ejemplo 8.16

---

```

1  class UnaClase
2  {
3      public:
4          UnaClase();
5          int n();
6          ...
7
8      private:
9          int n;
10         ...
11 };

```

---

Mensaje de error:

Salida ejemplo 8.16

```

1  unaclase.h:12: declaration of 'int UnaClase::n'
2  unaclase.h:9: conflicts with previous declaration
3      'int UnaClase::n()'

```

Solución: Cambiar el nombre a la función miembro o al atributo. Normalmente, como el nombre de la función miembro está establecido de cara al exterior de la clase, se debe cambiar el nombre del atributo.

```

1  class UnaClase
2  {
3      public:
4          UnaClase();
5          int n();

```

```

6      ...
7
8  private:
9    int nn;
10   ...
11 };

```

- Múltiples definiciones de la misma función. Ejemplo: Se ha definido el constructor por defecto de una clase y al definir el destructor de la clase se han olvidado de poner el símbolo ~.

---

Ejemplo 8.17

---

```

1 UnaClase::UnaClase()
2 {
3   ...
4 }
5 ...
6 UnaClase::UnaClase()
7 {
8   ...
9 }

```

---

Mensaje de error:

Salida ejemplo 8.17

```

1  unaclase.cc:28: redefinition of ‘UnaClase::UnaClase()’
2  unaclase.cc:6: ‘UnaClase::UnaClase()’ previously defined here

```

Solución: Eliminar o modificar las definiciones múltiples.

```

1 UnaClase::UnaClase()
2 {
3   ...
4 }
5 ...
6 UnaClase::~UnaClase()
7 {
8   ...
9 }

```

- No escribir el mismo tipo de valor de retorno en la declaración y la definición de una función de una clase. Ejemplo: Se declara una función con un tipo de valor de retorno (**int**) y luego se define con otro tipo (**void**).

---

Ejemplo 8.18

---

```
1 class UnaClase
2 {
3     public:
4         UnaClase();
5         int Algo(int);
6     };
7
8 UnaClase::UnaClase()
9 {
10    ...
11 }
12
13 void UnaClase::Algo(int a)
14 {
15    ...
16 }
```

Mensaje de error:

Salida ejemplo 8.18

```
1 unaclase.cc:21: prototype for 'void UnaClase::Algo(int)', does not
2     match any in class 'UnaClase'
3 unaclase.cc:9: candidate is: int UnaClase::Algo(int)
4 unaclase.cc:21: 'void UnaClase::Algo(int)' and 'int
5     UnaClase::Algo(int)' cannot be overloaded
```

Solución: Verificar los tipos de retorno en la declaración y la definición de las funciones.

```
1 class UnaClase
2 {
3     public:
4         UnaClase();
5         int Algo(int);
6     };
7
8 UnaClase::UnaClase()
9 {
10    ...
11 }
12
13 int UnaClase::Algo(int a)
14 {
15    ...
16 }
```

- Olvidarse el punto y coma (;) al final de la declaración de una clase. Ejemplo:

---

Ejemplo 8.19

---

```

1 class UnaClase
2 {
3     ...
4 }
```

---

Mensaje de error:

Salida ejemplo 8.19

```

1 prueba.cc:5: semicolon missing after declaration of 'UnaClase'
2 prueba.cc:6: two or more data types in declaration of 'main'
3 prueba.cc:6: semicolon missing after declaration of 'class UnaClase'
```

En otros casos, el mensaje de error es confuso y no ayuda a encontrar el error:

Salida ejemplo 8.19

```
1 unaclase.cc:6: return type specification for constructor invalid
```

Solución: Poner el punto y coma al final de la declaración de la clase.

```

1 class UnaClase
2 {
3     ...
4 };
```

- Acceder a los miembros privados de una clase. Ejemplo:

---

Ejemplo 8.20

---

```

1 class UnaClase
2 {
3     public:
4         ...
5
6     private:
7         int dim;
8         ...
9 };
10
11 /**
12  * CÓDIGO DE EJEMPLO ***
13 */
14 UnaClase a;
15
16 cout << a.dim;
```

---

Mensaje de error:

Salida ejemplo 8.20

```
1 prueba.cc: In function 'int main(...)':
2     unaclase.h:16: 'int UnaClase::dim' is private
3     prueba.cc:10: within this context
```

Solución: Tener claro lo que se desea público o privado. Si se desea acceder a algún miembro declarado privado, se puede proporcionar un interfaz público.

```
1 class UnaClase
2 {
3     public:
4         ...
5         int Dim(void);
6
7     private:
8         int dim;
9         ...
10 }
11
12 /*** CÓDIGO DE EJEMPLO ***/
13 UnaClase a;
14
15 cout << a.Dim();
```

- En el destructor, liberar la memoria dinámica (`delete`), pero no modificar el valor de otros atributos que indican si el objeto contiene algo. Ejemplo:

Ejemplo 8.21

```
1 UnaClase::~UnaClase()
2 {
3     if(v != NULL)
4     {
5         delete v;
6         v = NULL;
7     }
8     // No se hace lo siguiente
9     // dim = 0;
10 }
11
12 /*** CÓDIGO DE EJEMPLO ***/
13 UnaClase a(5);
14 ...
15 a.~UnaClase();
16 ...
```

```

17 for(i = 0; i < a.dim; i++)
18     cout << a.v[i];

```

Mensaje de error:

Salida ejemplo 8.21

```
1 Segmentation fault
```

Solución: Definir correctamente el destructor de un objeto.

```

1 UnaClase::~UnaClase()
2 {
3     if(v != NULL)
4     {
5         delete v;
6         v = NULL;
7     }
8
9     dim = 0;
10 }

```

- No saber cuándo se trabaja con un objeto o con un puntero a un objeto: confundir los operadores . y ->. Ejemplo:

Ejemplo 8.22

```

1 UnaClase a, *b;
2
3 b = &a;
4 cout << a->propiedad;
5 cout << b.propiedad;

```

Mensaje de error:

Salida ejemplo 8.22

```

1 prueba.cc: In function 'int main(...)':
2 prueba.cc:12: base operand of '->' has non-pointer type 'UnaClase'
3 prueba.cc:13: request for member 'propiedad' in 'b', which is of
4     non-aggregate type 'UnaClase *'

```

Solución: Distinguir correctamente los objetos y los punteros a objeto.

```

1 UnaClase a, *b;
2
3 b = &a;

```

```

4   cout << a.propiedad;
5   cout << b->propiedad;
```

- No saber cuándo se trabaja con un objeto o con un puntero a un objeto: intentar asignar un objeto a un puntero o vice versa. Ejemplo:

---

Ejemplo 8.23

---

```

1 UnaClase a, *b;
2
3 b = a;
```

---

Mensaje de error:

---

Salida ejemplo 8.23

---

```

1 prueba.cc: In function ‘int main(...’:
2 prueba.cc:10: cannot convert ‘a’ from type ‘UnaClase’ to type
3   ‘UnaClase *’
```

Solución: Distinguir correctamente los objetos y los punteros a objeto.

```

1 UnaClase a, *b;
2
3 b = &a;
```

- Crear un objeto a partir de una clase que no está perfectamente definida (por ejemplo, declarar una clase que contiene un objeto de la propia clase). Ejemplo:

---

Ejemplo 8.24

---

```

1 class UnaClase {
2     public:
3         UnaClase();
4
5     private:
6         UnaClase a;
7 };
```

Mensaje de error:

---

Salida ejemplo 8.24

---

```

1 unaclase.cc:11: field ‘a’ has incomplete type
```

Solución: En C++ no se puede crear un objeto si su tamaño y contenido son desconocidos (tipo incompleto). Con un tipo incompleto sólo se pueden hacer cosas que no requieran conocer su tamaño y disposición en memoria, como declarar punteros o referencias.

```

1 class UnaClase {
2     public:
3         UnaClase();
4
5     private:
6         UnaClase *a;
7 }
```

- Crear un objeto a partir de una clase que aún no ha sido declarada. Ejemplo:

Ejemplo 8.25

```

1 class UnaClase
2 {
3     public:
4         UnaClase();
5
6     private:
7         OtraClase a;
8 }
9
10 class OtraClase
11 {
12     public:
13         OtraClase();
14 }
```

Mensaje de error:

Salida ejemplo 8.25

```

1 unaclase.cc:11: 'OtraClase' is used as a type, but is not defined
2 as a type.
```

Solución: Declarar las clases antes de crear objetos a partir de ellas.

```

1 class OtraClase
2 {
3     public:
4         OtraClase();
5 }
6
7 class UnaClase
8 {
9     public:
10        UnaClase();
```

```

12 |     private:
13 |         OtraClase a;
14 | };

```

- Establecer una referencia mutua entre dos clases: la clase A contiene un objeto de la clase B y la clase B contiene un objeto de la clase A. Ejemplo:

---

Ejemplo 8.26

---

```

1 class UnaClase
2 {
3     public:
4         UnaClase();
5
6     private:
7         OtraClase a;
8 };
9
10 class OtraClase
11 {
12     public:
13         OtraClase();
14
15     private:
16         UnaClase b;
17 };

```

---

Mensaje de error:

Salida ejemplo 8.26

```

1 unaclase.cc:11: 'OtraClase' is used as a type, but is not defined
2 as a type.

```

Solución: Esta situación no se puede solucionar modificando el orden de declaración de las clases (como en el ejemplo anterior), ya que existen referencias mutuas. Además, desde un punto de vista lógico no tiene sentido: un objeto de la clase `UnaClase` contiene un objeto de la clase `OtraClase` que a su vez contiene un objeto de la clase `UnaClase` y así sucesivamente hasta el infinito.

- Establecer una referencia mutua entre dos clases (la clase A contiene un objeto de la clase B y la clase B contiene un objeto de la clase A), con una declaración adelantada (*forward*) para evitar el problema planteado en la situación anterior. Ejemplo:

---

Ejemplo 8.27

---

```

1 class OtraClase;
2
3 class UnaClase
4 {
5     public:
6         UnaClase();
7
8     private:
9         OtraClase a;
10};
11
12 class OtraClase
13 {
14     public:
15         OtraClase();
16
17     private:
18         UnaClase b;
19 };

```

Mensaje de error:

Salida ejemplo 8.27

```
1 unaclase.cc:13: field 'a' has incomplete type
```

Solución: La declaración adelantada indica al compilador que OtraClase es de tipo **class**. Este tipo de declaración únicamente permite declarar punteros o referencias, pero no es suficiente para crear objetos, ya que no se conoce el tamaño del objeto o la disposición de su contenido en memoria.

```

1 class OtraClase;
2
3 class UnaClase
4 {
5     public:
6         UnaClase();
7
8     private:
9         OtraClase *a;
10};
11
12 class OtraClase
13 {
14     public:
15         OtraClase();

```

```

16
17     private:
18         UnaClase b;
19     };

```

## 8.5. Sobre la sobrecarga de los operadores

- Emplear la declaración **friend** cuando se está definiendo el código (en el fichero .cc) de una función que es amiga. Ejemplo:

Ejemplo 8.28

```

1 friend ostream& operator<<(ostream& os, UnaClase& uc)
2 {
3     ...
4 }

```

Mensaje de error:

Salida ejemplo 8.28

```

1 unaclase.cc:50: can't initialize friend function '<<'
2 unaclase.cc:50: friend declaration not in class definition

```

Solución: Eliminar la palabra clave **friend**. Una función se declara como **friend** dentro de la clase donde es amiga, pero cuando se define su código no se tiene que indicar.

```

1 ostream& operator<<(ostream& os, UnaClase& uc)
2 {
3     ...
4 }

```

- En el operador asignación, no borrar el elemento de la izquierda de la asignación (es decir, el objeto sobre el que se invoca el operador asignación). Ejemplo:

Ejemplo 8.29

```

1 UnaClase&
2 UnaClase::operator=(UnaClase& a)
3 {
4     dim = a.dim;
5     v = new int[dim];
6     for(int i = 0; i < dim; i++)
7         v[i] = a.v[i];
8

```

---

```

9     return *this;
10 }
```

---

Mensaje de error: No produce un mensaje de error. Este problema no se manifiesta en forma de error. El único efecto que produce es no liberar memoria reservada. Se pueden producir problemas de falta de memoria a largo plazo si esta operación se repite muchas veces.

Solución: Antes de realizar la copia en la asignación, hay que borrar el elemento de la izquierda de la asignación. Se puede realizar directamente o llamando al destructor de la clase.

```

1 UnaClase&
2 UnaClase::operator=(UnaClase& a)
3 {
4     this->~UnaClase();
5     dim = a.dim;
6     v = new int[dim];
7     for(int i = 0; i < dim; i++)
8         v[i] = a.v[i];
9
10    return *this;
11 }
```

- En el operador asignación, no se protege el código de la autoasignación. Ejemplo:

---

Ejemplo 8.30

---

```

1 // Sobre carga del operador asignación
2 UnaClase&
3 UnaClase::operator=(UnaClase& a)
4 {
5     this->~UnaClase();
6     dim = a.dim;
7     v = new int[dim];
8     for(int i = 0; i < dim; i++)
9         v[i] = a.v[i];
10
11    return *this;
12 }
13
14 /*** CÓDIGO DE EJEMPLO ***/
15 //
16 UnaClase a(5);
```

```
17 // Al realizar una autoasignación, se borra el objeto
18 a = a;
```

Mensaje de error: No se produce un mensaje de error durante la compilación. Este problema no se manifiesta en forma de error durante la ejecución. Únicamente se borra el objeto que realiza la autoasignación.

Solución: Comprobar antes de realizar la copia en la asignación que no se trata del mismo objeto. Si es así, la función no tiene que hacer nada.

```
1 UnaClase&
2 UnaClase::operator=(UnaClase& a)
3 {
4     // Más rápido que if(*this != a)
5     if(this != &a)
6     {
7         this->~UnaClase();
8         dim = a.dim;
9         v = new int[dim];
10        for(int i = 0; i < dim; i++)
11            v[i] = a.v[i];
12    }
13
14    return *this;
15 }
```

- En los operadores de entrada >> y salida <<, emplear los flujos de entrada (`cin`) o salida estándar (`cout`) en vez de los recibidos. Ejemplo:

---

Ejemplo 8.31

---

```
1 ostream& operator<<(ostream& os, UnaClase& uc)
2 {
3     cout << "Dimension: " << uc.dim << endl;
4
5     for(int i = 0; i < uc.dim; i++)
6         cout << "[" << uc.v[i] << "]";
7
8     return os;
9 }
```

Mensaje de error: No se produce un mensaje de error durante la compilación. Este problema no se manifiesta en forma de error durante la ejecución.

Solución: Para la entrada y salida se tienen que emplear los flujos que se reciben como argumento. El mismo código se puede emplear con distintos tipos de flujo (por ejemplo, para flujos de fichero).

```

1 ostream& operator<<(ostream& os, UnaClase& uc)
2 {
3     os << "Dimension: " << uc.dim << endl;
4
5     for(int i = 0; i < uc.dim; i++)
6         os << "[" << uc.v[i] << "]";
7
8     return os;
9 }
```

- En los operadores de entrada `>>` y salida `<<`, no devolver el flujo recibido (primera variante). Ejemplo:

---

Ejemplo 8.32

---

```

1 ostream& operator<<(ostream& os, UnaClase& uc)
2 {
3     os << "Dimension: " << uc.dim << endl;
4
5     for(int i = 0; i < uc.dim; i++)
6         os << "[" << uc.v[i] << "]";
7
8
9 /**
10  * *** CÓDIGO DE EJEMPLO ***
11 */
12 cout << a << endl;
```

---

Mensaje de error: Se produce un error al realizar operaciones de salida sobre el mismo flujo en una sola instrucción.

Salida ejemplo 8.32

1 Segmentation fault

Solución: La sobrecarga de los operadores de entrada y salida deben de devolver el flujo recibido.

```

1 ostream& operator<<(ostream& os, UnaClase& uc)
2 {
3     os << "Dimension: " << uc.dim << endl;
```

```

4
5     for(int i = 0; i < uc.dim; i++)
6         os << "[" << uc.v[i] << "]";
7
8     return os;
9 }
```

- En los operadores de entrada `>>` y salida `<<`, no devolver el flujo recibido (segunda variante). Ejemplo:

---

Ejemplo 8.33

---

```

1 void operator<<(ostream& os, UnaClase& uc)
2 {
3     os << "Dimension: " << uc.dim << endl;
4
5     for(int i = 0; i < uc.dim; i++)
6         os << "[" << uc.v[i] << "]";
7 }
8
9 /*** CÓDIGO DE EJEMPLO ***/
10 UnaClase a;
11
12 cout << a << endl;
```

---

Mensaje de error: Se está intentado emplear un valor de tipo `void` para realizar una operación.

Salida ejemplo 8.33

```

1 prueba.cc: In function ‘int main(...’:
2 prueba.cc:24: void value not ignored as it ought to be
```

Solución: La sobrecarga de los operadores de entrada y salida deben de devolver el flujo recibido.

```

1 ostream& operator<<(ostream& os, UnaClase& uc)
2 {
3     os << "Dimension: " << uc.dim << endl;
4
5     for(int i = 0; i < uc.dim; i++)
6         os << "[" << uc.v[i] << "]";
7
8     return os;
9 }
```

- En el operador corchete `[]`, devolver por valor en vez de por referencia. Ejemplo:

---

Ejemplo 8.34

---

```

1 int
2 UnaClase::operator[](int i)
3 {
4     ...
5 }
6
7 /* CÓDIGO DE EJEMPLO */
8 int i;
9 UnaClase a;
10
11 i = a[1];
12 a[1] = 10;

```

---

Mensaje de error: El error se produce cuando se emplea el operador corchete en la parte izquierda de una asignación, porque no se tiene una dirección de memoria (una referencia). Cuando se emplea en la parte derecha (`i = a[i]`) no se produce un error.

---

Salida ejemplo 8.34

---

```

1 prueba.cc: In function ‘int main(...’:
2 prueba.cc:14: non-lvalue in assignment

```

Solución: El operador corchete tiene que devolver por referencia.

```

1 int&
2 UnaClase::operator[](int i)
3 {
4     ...
5 }

```

## 8.6. Sobre la memoria

- Un objeto creado de forma dinámica con `new` existe hasta que se destruye explícitamente mediante `delete`. Ejemplo:

---

Ejemplo 8.35

---

```

1 void unaFuncion(void)
2 {
3     UnaClase *a;
4
5     a = new UnaClase();
6     ...
7 }

```

Mensaje de error: No se produce un mensaje de error durante la compilación. Este problema no se manifiesta en forma de error durante la ejecución. El único efecto que produce es no liberar memoria reservada. Se pueden producir problemas de falta de memoria a largo plazo si esta operación se repite muchas veces.

Solución: El programador se tiene que encargar de liberar toda la memoria que haya reservado de forma dinámica.

```
1 void unaFuncion(void)
2 {
3     UnaClase *a;
4
5     a = new UnaClase();
6     ...
7     delete a;
8 }
```

- Liberar la memoria reservada poniendo un puntero a NULL. Ejemplo:

---

Ejemplo 8.36

---

```
1 void unaFuncion(void)
2 {
3     UnaClase *a;
4
5     a = new UnaClase();
6     ...
7     a = NULL;
8 }
```

---

Mensaje de error: No se produce un mensaje de error durante la compilación. Este problema no se manifiesta en forma de error durante la ejecución. El único efecto que produce es no liberar memoria reservada. Se pueden producir problemas de falta de memoria a largo plazo si esta operación se repite muchas veces.

Solución: El programador se tiene que encargar de liberar toda la memoria que haya reservado de forma dinámica. Al poner un puntero a NULL no se libera memoria reservada: únicamente se cambia el valor de un puntero.

```

1 void unaFuncion(void)
2 {
3     UnaClase *a;
4
5     a = new UnaClase();
6     ...
7     delete a;
8 }
```

- Para destruir un objeto, llamar directamente a su destructor, pero luego no liberar la memoria con `delete`. Ejemplo:

Ejemplo 8.37

---

```

1 void unaFuncion(void)
2 {
3     UnaClase *a;
4
5     a = new UnaClase();
6     ...
7     a->~UnaClase();
8 }
```

---

Mensaje de error: No se produce un mensaje de error durante la compilación. Este problema no se manifiesta en forma de error durante la ejecución. El único efecto que produce es no liberar memoria reservada. Se pueden producir problemas de falta de memoria a largo plazo si esta operación se repite muchas veces.

Solución: El programador se tiene que encargar de liberar toda la memoria que haya reservado de forma dinámica. El destructor por sí solo no libera la memoria que ocupa un objeto: sólo libera la memoria que haya reservado el objeto. No hace falta llamar explícitamente al destructor, ya que al realizar `delete` se invoca automáticamente.

```

1 void unaFuncion(void)
2 {
3     UnaClase *a;
4
5     a = new UnaClase();
6     ...
7     a->~UnaClase();
8     delete a;
9     // O solamente (mejor)
```

```
10 }     delete a;  
11 }
```

- Liberar la memoria a la que apunta un puntero (`delete`), pero no poner el puntero a NULL. Ejemplo:

---

Ejemplo 8.38

---

```
1 char *a;  
2  
3 a = new char[20];  
4 ...  
5 delete a;  
6 ...  
7 if(a != NULL)  
8     strcpy(a, "Una cadena");
```

---

Mensaje de error: Como se está accediendo a una zona de memoria que ya no es válida, lo más seguro es que se produzca un mensaje de violación de segmento.

Salida ejemplo 8.38

```
1 Segmentation fault
```

Solución: Siempre que se libere la memoria a la que apunta un puntero, es recomendable ponerlo a NULL para evitar que se vuelva a usar esa zona de memoria.

```
1 char *a;  
2  
3 a = new char[20];  
4 ...  
5 delete a;  
6 a = NULL;  
7 ...  
8 if(a != NULL)  
9     strcpy(a, "Una cadena");
```

- Liberar un array de objetos con `delete` en vez de con `delete []` o liberar un objeto individual con `delete []` en vez de con `delete`. Ejemplo:

---

Ejemplo 8.39

---

```
1 UnaClase *a, *b;  
2  
3 a = new UnaClase;  
4 b = new UnaClase[5];  
5 ...
```

---

```
6 delete [] a;
7 delete b;
```

---

Mensaje de error: La manera exacta en la que se asigna la memoria de los arrays y de los objetos individuales es dependiente de la implementación. El uso incorrecto del operador `delete` puede producir mensajes de violación de segmento.

Salida ejemplo 8.39

```
1 Segmentation fault
```

Solución: El programador siempre debe decir si se va a destruir un array o un objeto individual.

```
1 UnaClase *a, *b;
2
3 a = new UnaClase;
4 b = new UnaClase[5];
5 ...
6 delete a;
7 delete [] b;
```

- Liberar la misma zona de memoria dos veces. Ejemplo:

Ejemplo 8.40

```
1 char *a, *b;
2
3 a = new char[20];
4 strcpy(a, "Una cadena");
5 b = a;
6 delete a;
7 a = NULL;
8 delete b;
9 b = NULL;
```

Mensaje de error: Como se está intentado liberar una zona de memoria que ya no es válida, lo más seguro es que se produzca un mensaje de violación de segmento.

Salida ejemplo 8.40

```
1 Segmentation fault
```

Solución: En este ejemplo, el error se ha producido por tener dos punteros apuntando a la misma zona de memoria. Hay que evitar situaciones de este estilo. En el caso de objetos, este error se suele producir al copiar un objeto que contiene punteros y no se realiza una copia correcta (reservando una zona de memoria y copiando el contenido). Por tanto, suele ocurrir si el constructor de copia o el operador asignación no están correctamente programados.

- No liberar toda la memoria cuando un objeto contiene punteros a otros objetos.  
Ejemplo:

---

Ejemplo 8.41

---

```
1 class OtraClase
2 {
3     public:
4         OtraClase();
5         ~OtraClase();
6         ...
7
8     private:
9         UnaClase *a;
10    ...
11 };
12
13 OtraClase::OtraClase()
14 {
15     a = new UnaClase();
16 }
17
18 OtraClase::~OtraClase()
19 {
20 }
```

---

Mensaje de error: No se produce un mensaje de error durante la compilación. Este problema no se manifiesta en forma de error durante la ejecución. El único efecto que produce es no liberar memoria reservada. Se pueden producir problemas de falta de memoria a largo plazo si esta operación se repite muchas veces.

Solución: En el destructor de un objeto hay que liberar toda la memoria que tenga reservada el objeto.

```
1 | OtraClase::~OtraClase()
2 | {
```

```

3   if(a != NULL)
4   {
5     delete a;
6     a = NULL;
7   }
8 }
```

## 8.7. Sobre las cadenas

- Realizar una copia de cadenas mediante una asignación de punteros. Ejemplo:

---

Ejemplo 8.42

---

```

1 char a[] = "Una cadena";
2 char *b;
3
4 b = a;
```

---

Mensaje de error: No se produce un mensaje de error durante la compilación. Se pueden producir errores del tipo **Segmentation fault** durante la ejecución, ya que existen dos punteros que hacen referencia a la misma zona de memoria.

Solución: Realizar una copia mediante **strdup** o mediante **new**, **strlen** y **strcpy**.

```

1 char a[] = "Una cadena";
2 char *b;
3
4 b = strdup(a);
5 // O también
6 b = new char[strlen(a) + 1];
7 strcpy(b, a);
```

- Realizar una copia de cadenas mediante **new** y **strcpy** y no obtener la longitud de la cadena. Ejemplo:

---

Ejemplo 8.43

---

```

1 char a[] = "Una cadena";
2 char *b;
3
4 b = new char[a + 1];
5 strcpy(b, a);
```

---

Mensaje de error:

## Salida ejemplo 8.43

```
1 prueba.cc: In function 'int main(...)':
2     prueba.cc:9: size in array new must have integral type
```

Solución: Emplear `strlen` para obtener la longitud de la cadena que se va a copiar.

```
1 char a[] = "Una cadena";
2 char *b;
3
4 b = new char[strlen(a) + 1];
5 strcpy(b, a);
```

- Realizar una copia de cadenas mediante `new`, `strlen` y `strcpy` y no reservar espacio para el carácter final de cadena '`\0`'. Ejemplo:

## Ejemplo 8.44

```
1 char a[] = "Una cadena";
2 char *b;
3
4 b = new char[strlen(a)];
5 strcpy(b, a);
```

Mensaje de error: No se produce un mensaje de error durante la compilación. Se pueden producir errores del tipo **Segmentation fault** durante la ejecución, ya que el carácter '`\0`' se ha copiado en una zona de memoria no válida.

Solución: Realizar una copia reservando una posición más para el carácter '`\0`'.

```
1 char a[] = "Una cadena";
2 char *b;
3
4 b = new char[strlen(a) + 1];
5 strcpy(b, a);
```

- Realizar una copia de cadenas mediante `new`, `strlen` y `strcpy` y reservar espacio para el carácter final de cadena '`\0`' de forma incorrecta. Ejemplo:

## Ejemplo 8.45

```
1 char a[] = "Una cadena";
2 char *b;
3
```

---

```

4 b = new char[strlen(a + 1)];
5 strcpy(b, a);

```

---

Mensaje de error: No se produce un mensaje de error durante la compilación. Se pueden producir errores del tipo **Segmentation fault** durante la ejecución, ya que no se ha reservado suficiente espacio para copiar la cadena (se ha reservado dos posiciones menos de las necesarias).

Solución: Realizar una copia reservando una posición más para el carácter '\0'.

```

1 char a[] = "Una cadena";
2 char *b;
3
4 b = new char[strlen(a) + 1];
5 strcpy(b, a);

```

- Realizar una copia de una cadena sobre otra que ya existía previamente, sin destruir la primera y reservar espacio para la nueva cadena. Ejemplo:

---

Ejemplo 8.46

---

```

1 char *a, *b;
2 a = strdup("Una cadena");
3 b = strdup("Otra cadena más larga");
4
5 strcpy(a, b);

```

---

Mensaje de error: No se produce un mensaje de error durante la compilación. Se pueden producir errores del tipo **Segmentation fault** durante la ejecución, ya que al copiar una cadena sobre otra de longitud menor se invaden zonas de memoria reservadas o utilizadas en otro sitio.

Solución: Liberar la memoria que ocupa la cadena sobre la que se va a copiar y realizar una copia mediante **strdup** o mediante **new**, **strlen** y **strcpy**.

```

1 char *a, *b;
2 a = strdup("Una cadena");
3 b = strdup("Otra cadena más larga");
4
5 delete a;
6 a = strdup(b);

```

- Reservar memoria para una cadena mediante el operador `new` y confundir los paréntesis `()` por los corchetes `[]`. Ejemplo:

---

Ejemplo 8.47

---

```

1 char *a;
2
3 a = new char(65);
4 strcpy(a, "Una cadena de caracteres");

```

---

Mensaje de error: No se produce un mensaje de error durante la compilación. Se pueden producir errores del tipo **Segmentation fault** durante la ejecución, ya que la cadena se ha copiado en una zona de memoria no válida (se ha reservado espacio para un carácter y se ha inicializado con el código ASCII 65).

Solución: Para reservar una cadena (un array de caracteres) se tienen que emplear los corchetes.

```

1 char *a;
2
3 a = new char[65];
4 strcpy(a, "Una cadena de caracteres");

```

- Intentar liberar una cadena que no ha sido creada mediante memoria dinámica (`new`). Ejemplo:

---

Ejemplo 8.48

---

```

1 char a[] = "Una cadena";
2
3 delete a;

```

---

Mensaje de error: El compilador es capaz de detectar esta situación y genera un mensaje de error.

Salida ejemplo 8.48

```

1 prueba.cc: In function 'int main(...)':
2 prueba.cc:15: warning: deleting array 'char a[11]'

```

Solución: No se puede liberar la memoria estática.

- Intentar liberar una cadena que no ha sido creada mediante memoria dinámica (`new`). Similar a la anterior situación, pero la cadena se pasa a una función. Ejemplo:

---

Ejemplo 8.49

---

```

1 void unaFuncion(char *c)
2 {
3     delete c;
4 }
5
6 /*** CÓDIGO DE EJEMPLO ***/
7 char a[] = "Una cadena";
8
9 unaFuncion(a);

```

---

Mensaje de error: En este caso, el compilador no es capaz de detectar esta situación. Durante la ejecución, lo más seguro es que se produzca un mensaje de violación de segmento.

---

Salida ejemplo 8.49

---

```
1 Segmentation fault
```

Solución: No se puede liberar la memoria estática.

- Emplear una cadena nula (el puntero apunta a NULL) con las funciones de manejo de cadena (**strlen**, **strcmp**, **strdup** y **strcpy**). Ejemplo:

---

Ejemplo 8.50

---

```

1 char *a = NULL, *b;
2
3 cout << strlen(a);
4 // O también
5 b = strdup(a);
6 // O también
7 strcpy(b, a);

```

---

Mensaje de error: Durante la ejecución, lo más seguro es que se produzca un mensaje de violación de segmento.

---

Salida ejemplo 8.50

---

```
1 Segmentation fault
```

Solución: Tener cuidado al manejar las cadenas. Comprobar siempre que no apuntan a NULL.

- Comparar dos cadenas comparando directamente los punteros. Ejemplo:

## Ejemplo 8.51

```

1 bool compara(char *a, char *b)
2 {
3     if(a == b)
4         return true;
5     return false;
6 }
7
8 /*** CÓDIGO DE EJEMPLO ***/
9 char a[] = "Esto es una cadena";
10 char b[] = "Esto es una cadena";
11
12 // Escribe false
13 cout << compara(a, b);

```

Mensaje de error: No produce un mensaje de error. Se trata de un error lógico, ya que no se están comparando las cadenas, sino los punteros. En el ejemplo anterior, la función `compara()` devuelve `false`, aunque las dos cadenas sean iguales.

Solución: Emplear la función `strcmp()`, que devuelve 0 cuando dos cadenas son iguales.

```

1 bool compara(char *a, char *b)
2 {
3     if(!strcmp(a, b))
4         return true;
5     return false;
6 }
7
8 /*** CÓDIGO DE EJEMPLO ***/
9 char a[] = "Esto es una cadena";
10 char b[] = "Esto es una cadena";
11
12 // Escribe true
13 cout << compara(a, b);

```

- Imprimir una cadena carácter a carácter. Ejemplo:

## Ejemplo 8.52

```

1 char a[] = "Esto es una cadena";
2

```

---

```

3 for(int i = 0; i < strlen(a); i++)
4     cout << a[i];

```

---

Mensaje de error: No produce un mensaje de error. Es ineficiente, ya que una cadena se puede imprimir directamente.

Solución: Se puede imprimir una cadena directamente.

```

1 | char a[] = "Esto es una cadena";
2 |
3 | cout << a;

```

## 8.8. Varios

- Confundir la asignación (=) con el operador igualdad (==). Ejemplo:

---

Ejemplo 8.53

---

```

1 void unaFuncion(char *a)
2 {
3     if(a = NULL)
4         cout << "La cadena está vacía" << endl;
5     else
6         cout << a << endl;
7 }

```

---

Mensaje de error: No se produce un mensaje de error. Siempre se va a ejecutar el código de la parte **else** (muestra `(null)`, que es la representación en pantalla de un puntero a cadena que vale `NULL`), porque se está realizando una asignación en vez de una comparación.

Solución: No confundir los dos operadores.

```

1 | void unaFuncion(char *a)
2 | {
3 |     if(a == NULL)
4 |         cout << "La cadena está vacía" << endl;
5 |     else
6 |         cout << a << endl;
7 |

```

- Confundir el preincremento (`++a`) con el postincremento (`a++`). Ejemplo:

---

Ejemplo 8.54

---

```
1 a = b++;
2 cout << "a y b tienen el mismo valor" << endl;
```

---

Mensaje de error: No se produce un mensaje de error. `a` y `b` no tienen el mismo valor porque primero se realiza la asignación y luego se incrementa `b`, ya que se está empleando el operador postincremento.

Solución: Cuando se emplea el operador preincremento, primero se realiza el incremento y luego el resto de operaciones. Cuando se emplea el operador postincremento, primero se realizan el resto de operaciones y luego el incremento.

```
1 | a = ++b;
2 | cout << "a y b tienen el mismo valor" << endl;
```

- Poner un punto y coma (`;`) después de la sentencia de un bucle. Ejemplo:

---

Ejemplo 8.55

---

```
1 for(i = 0; i < 10; i++);
2 {
3     ...
4 }
```

---

Mensaje de error: No se produce un mensaje de error. Las sentencias que se supone que se ejecutan dentro del bucle sólo se ejecutan una vez, ya que no están realmente dentro del bucle.

Solución: Tener cuidado con las sentencias de repetición (`for`, `do ... while` y `while`).

```
1 | for(i = 0; i < 10; i++)
2 | {
3 |     ...
4 | }
```

- Acceder a una posición no válida de un array. Ejemplo:

## Ejemplo 8.56

```

1 int vector[10];
2
3 for(int i = 0; i <= 10; i++)
4     vector[i] = 1;

```

Mensaje de error: Como se está accediendo a una zona de memoria que no es válida, lo más seguro es que se produzca un mensaje de violación de segmento.

## Salida ejemplo 8.56

```
1 Segmentation fault
```

Solución: Tener cuidado con los índices de los arrays. En C++ los arrays comienzan en 0 y terminan en dimensión - 1.

```

1 int vector[10];
2
3 for(int i = 0; i < 10; i++)
4     vector[i] = 1;

```

- Devolver una referencia a una variable u objeto local. Ejemplo:

## Ejemplo 8.57

```

1 UnaClase&
2 UnaClase::operator=(UnaClase& a)
3 {
4     UnaClase u;
5
6     ...
7     return u;
8 }

```

Mensaje de error:

## Salida ejemplo 8.57

```

1 unaclase.cc: In method ‘class UnaClase & UnaClase::operator
2     =(UnaClase & )’:
3     unaclase.cc:45: warning: reference to local variable ‘u’ returned

```

Solución: No devolver referencias a una variable u objeto local.

- Devolver un puntero a una variable local u objeto local. Ejemplo:

---

Ejemplo 8.58

---

```
1 UnaClase*
2 f(void)
3 {
4     UnaClase u;
5
6     ...
7     return &u;
8 }
```

---

Mensaje de error:

---

Salida ejemplo 8.58

---

```
1 prueba.cc: In function ‘class UnaClase * f()’:
2 prueba.cc:8: warning: address of local variable ‘a’ returned
```

---

Sólo en aquellos casos (como el anterior) donde el compilador puede detectar que se está devolviendo un puntero a una variable u objeto local se genera un mensaje de error. Por ejemplo, el siguiente código presenta el mismo problema, pero el compilador no lo detecta:

---

Ejemplo 8.59

---

```
1 UnaClase*
2 f(void)
3 {
4     UnaClase a, *b;
5
6     b = &a;
7
8     return b;
9 }
```

---

Solución: No devolver punteros a una variable u objeto local.

- Emplear para una variable de una función el mismo identificador que un argumento de la función. Ejemplo:

---

Ejemplo 8.60

---

```
1 UnaClase&
2 UnaClase::operator=(UnaClase& a)
3 {
4     UnaClase a;
5 }
```

```

6     ...
7 }

```

Mensaje de error:

Salida ejemplo 8.60

```

1  unaclase.cc: In method 'class UnaClase & UnaClase::operator
2      =(UnaClase &)' :
3  unaclase.cc:45: declaration of 'a' shadows a parameter

```

Solución: Usar identificadores distintos para los argumentos y las variables locales.

```

1  UnaClase&
2  UnaClase::operator=(UnaClase& a)
3  {
4      UnaClase a1;
5
6      ...
7 }

```

- Usar incorrectamente cualquier puntero y, en especial, el puntero `this`. Ejemplo:

Ejemplo 8.61

```

1 UnaClase::UnaClase()
2 {
3     (this*).dim=0;
4     (this*).v=NULL;
5 }

```

Mensaje de error:

Salida ejemplo 8.61

```

1  unaclase.cc: In method 'UnaClase::UnaClase()' :
2  unaclase.cc:7: parse error before ')'
3  unaclase.cc:8: parse error before ')'

```

Solución: El operador `*` se pone delante del puntero y no detrás.

```

1  UnaClase::UnaClase()
2  {
3      (*this).dim=0;
4      (*this).v=NULL;
5 }

```

También se puede emplear el operador `->`.

```

1 UnaClase::UnaClase()
2 {
3     this->dim=0;
4     this->v=NULL;
5 }
```

- No tener en cuenta la precedencia de los operadores. Ejemplo:

---

Ejemplo 8.62

---

```

1 UnaClase::UnaClase()
2 {
3     *this.dim=0;
4     *this.v=NULL;
5 }
```

---

Mensaje de error:

Salida ejemplo 8.62

```

1 unaclase.cc: In method ‘UnaClase::UnaClase()’:
2 unaclase.cc:7: request for member ‘dim’ in ‘this’, which is of
3     non-aggregate type ‘UnaClase *’
4 unaclase.cc:8: request for member ‘v’ in ‘this’, which is of
5     non-aggregate type ‘UnaClase *’
```

Solución: El operador `.` tiene mayor precedencia que el operador `*`. Si primero se tiene que aplicar el operador `*`, se tiene que encerrar la operación entre paréntesis.

```

1 UnaClase::UnaClase()
2 {
3     (*this).dim=0;
4     (*this).v=NULL;
5 }
```

- Suponer que el indicador de puntero (\*) se aplica a toda una lista de variables en una declaración. Ejemplo:

---

Ejemplo 8.63

---

```

1 char *a, b, c;
2
3 a = strdup("Una cadena");
4 b = strdup("Otra cadena");
5 c = strdup("La última cadena");
```

Mensaje de error:

Salida ejemplo 8.63

```
1 prueba.cc: In function 'int main(...)'::  
2 prueba.cc:10: assignment to 'char' from 'char *' lacks a cast  
3 prueba.cc:11: assignment to 'char' from 'char *' lacks a cast
```

Solución: El indicador de puntero tiene que acompañar a cada variable que se desee declarar como puntero.

```
1 char *a, *b, *c;  
2  
3 a = strdup("Una cadena");  
4 b = strdup("Otra cadena");  
5 c = strdup("La última cadena");
```

# Capítulo 9

## Ejercicios

En este capítulo se proponen una serie de ejercicios complementarios a los propuestos a lo largo del libro.

### Índice General

---

9.1. Mentiras arriesgadas . . . . .	185
9.2. La historia interminable . . . . .	186
9.3. Pegado a ti . . . . .	187
9.4. Clase TComplejo . . . . .	188

---

### 9.1. Mentiras arriesgadas

El siguiente código es muy sencillo, pero el resultado que se obtiene no es el que cabría esperar inicialmente. En principio parece obvio que el resultado del programa es mostrar el valor 2 por pantalla, pero esto no es así. ¿Qué es lo que pasa?

---

Ejemplo 9.1

---

```
1 #include <iostream>
2
3 using namespace std;
4
5 int
6 calcula(int x, int y, int z)
7 {
8     int a = z;
```

```

9
10    if(x >= 0)
11        if(y > 0) a = 1;
12    else if(y > 0) a = 2;
13
14    return a;
15 }
16
17 int
18 main(void)
19 {
20     cout << calcula(-1, 2, 0) << endl;
21
22     return 0;
23 }
```

---

## 9.2. La historia interminable

El siguiente código no presenta errores de compilación: se puede compilar y ejecutar sin problemas. Sin embargo, presenta un error lógico que origina un bucle infinito. Sin ejecutar el código, ¿sabrías decir dónde está el problema? ¿Cómo lo arreglarías?

---

Ejemplo 9.2

---

```

1 #include <iostream>
2
3 using namespace std;
4
5 class MiClase {
6     public:
7         MiClase()
8     {
9         cout << "MiClase()" << endl;
10        a = 0;
11    }
12
13    MiClase(MiClase &b)
14    {
15        cout << "MiClase(MiClase &)" << endl;
16        *this = b;
17    }
18
19    MiClase operator=(MiClase b)
20    {
21        cout << "operator=" << endl;
```

```
22         return *this;
23     }
24
25     private:
26     int a;
27 };
28
29
30 int main(void)
31 {
32     MiClase a;
33     MiClase b(a);
34
35     return 0;
36 }
```

---

### 9.3. Pegado a ti

El siguiente código no presenta errores de compilación.

---

Ejemplo 9.3

---

```
1 #include <iostream>
2
3 using namespace std;
4
5 class MiClase
6 {
7     friend void f(MiClase);
8
9     public:
10     MiClase(char *n)
11     {
12         a = new char[strlen(n) + 1];
13         if(a != NULL)
14             strcpy(a, n);
15     }
16
17     MiClase(MiClase &mc)
18     {
19         a = mc.a;
20     }
21
22
23     ~MiClase()
```

```
24  {
25      if(a != NULL)
26      {
27          delete a;
28          a = NULL;
29      }
30  }
31
32 private:
33     char *a;
34 };
35
36 void
37 f(MiClase mc)
38 {
39     cout << mc.a << endl;
40 }
41
42 int
43 main(void)
44 {
45     MiClase a("Programación y Estructuras de Datos");
46     f(a);
47
48     MiClase b(a);
49     f(b);
50
51     return 0;
52 }
```

---

Sin embargo, al ejecutarlo se obtiene el siguiente resultado por la salida estándar:

— Salida ejemplo 9.3 —

```
1 Programación y Estructuras de Datos
2 Programación y Estructuras de Datos
3 Segmentation fault
```

¿Qué está ocurriendo? ¿Dónde está el problema? ¿Cómo se puede resolver?

## 9.4. Clase TComplejo

Definid en C++ la clase **TComplejo** que representa números complejos y permite realizar aritmética compleja. Los números complejos tienen la forma:

```
parteReal + parteImaginaria * i
```

Se tienen que emplear datos de tipo `double` para representar la parte real e imaginaria. El prototipo de las funciones a implementar para esta clase es:

```
1 // Constructor por defecto: parte real e imaginaria
2 // inicializadas a 0.
3 TComplejo();
4
5 // Constructor a partir de la parte real: parte imaginaria
6 // inicializa a 0.
7 TComplejo(double);
8
9 // Constructor a partir de la parte real e imaginaria
10 TComplejo(double, double);
11
12 // Constructor de copia
13 TComplejo(TComplejo &);
14
15 // Destructor
16 ~TComplejo();
17
18 // Sobrecarga de operadores aritméticos
19 TComplejo operator+(TComplejo &);
20 TComplejo operator-(TComplejo &);
21 TComplejo operator*(TComplejo &);
22 TComplejo operator+(double);
23 TComplejo operator-(double);
24 TComplejo operator*(double);
25 friend TComplejo operator+(double, TComplejo &);
26 friend TComplejo operator-(double, TComplejo &);
27 friend TComplejo operator*(double, TComplejo &);
28
29 // Operador =: asignación de números complejos
30 TComplejo& operator=(TComplejo &);
31
32 // Operador ==: dos números complejos son iguales si
33 // la parte real es igual y la parte imaginaria
34 // es igual
35 bool operator==(TComplejo &);
36
37 // Operador !=: dos números complejos son distintos si
38 // la parte real no es igual o la parte imaginaria
39 // no es igual
40 bool operator!=(TComplejo &);
41
42 // Devuelve la parte real
```

```

43 double Re();
44
45 // Modifica la parte real
46 void Re(double);
47
48 // Devuelve la parte imaginaria
49 double Im();
50
51 // Modifica la parte imaginaria
52 void Im(double);
53
54 // Calcula el módulo
55 double Mod(void);
56
57 // Calcula el argumento, expresado en radianes
58 double Arg(void);
59
60 // Operador <<: muestra el número complejo, primero la
61 // parte real y después la parte imaginaria, separadas por
62 // un espacio en blanco y todo el número complejo
63 // encerrado entre paréntesis. Por ejemplo:
64 // (3 1.55)
65 friend ostream & operator<<(ostream &, TComplejo &);
```

En la parte privada contiene:

```

1 // Parte real
2 double re;
3 // Parte imaginaria
4 double im;
```

Algunas aclaraciones sobre el código que se tiene que implementar:

- El constructor por defecto, el constructor a partir de la parte real y el constructor a partir de la parte real e imaginaria se pueden agrupar en uno solo: `TComplejo(double = 0, double = 0)`.
- El destructor tiene que inicializar la parte real e imaginaria a 0.
- Para el cálculo del módulo y el argumento es necesario el empleo de funciones matemáticas auxiliares: raíz cuadrada, arcotangente, etc.
- La función que calcula el argumento debe de devolver un valor comprendido entre PI y -PI.

# Apéndice A

## Palabras clave

En este apéndice se listan y describen brevemente las palabras clave del lenguaje C++.

### Índice General

---

A.1. Lista de palabras clave . . . . .	191
--	-----

---

### A.1. Lista de palabras clave

#### A

**asm** Inserta una instrucción de ensamblador.

**auto** Declara una variable local.

#### B

**bool** Declara una variable booleana.

**break** Finaliza un bucle o una sentencia de selección (switch).

#### C

**case** Un bloque de código en una sentencia de selección (switch).

**catch** Captura una excepción

**char** Declara una variable carácter.

**class** Declara una clase.

**const** Declara una variable que no cambia su valor o una función que no modifica valores.

**const\_cast** Conversión a partir de variables const.

**continue** Salta una iteración en un bucle.

## D

**default** Caso por defecto en una sentencia de selección (switch).

**delete** Libera memoria reservada.

**do** Bucle con condición inicial.

**double** Declara una variable de punto flotante de doble precisión.

**dynamic\_cast** Realiza una conversión en tiempo de ejecución.

## E

**else** Caso alternativo en una sentencia condicional if.

**enum** Crea un tipo enumerado.

**explicit** Obliga a usar un constructor únicamente cuando la llamada coincide exactamente con el constructor.

**extern** Indica al compilador variables declaradas en un ámbito distinto al actual (por ejemplo, otro fichero).

## F

**false** El valor booleano falso.

**float** Declara una variable de punto flotante.

**for** Bucle con contador.

**friend** Concede acceso a la parte privada a una función o clase no miembro.

## G

**goto** Salta a una parte diferente en el código del programa.

**I**

**if** Sentencia condicional simple.

**inline** Optimiza la llamada a funciones cortas.

**int** Declara una variable entera.

**L**

**long** Declara una variable entera larga.

**M**

**mutable** Anula una declaración como constante.

**N**

**namespace** Define un espacio de nombres.

**new** Reserva memoria dinámica.

**O**

**operator** Declara una función que sobrecarga un operador.

**P**

**private** Declara miembros privados de una clase.

**protected** Declara miembros protegidos de una clase.

**public** Declara miembros públicos de una clase.

**R**

**register** Optimiza la velocidad de acceso a una variable.

**reinterpret\_cast** Cambia el tipo de una variable.

**return** Devuelve un valor de una función.

**S**

**short** Declara una variable entera corta.

**signed** Declara una variable con signo.

**sizeof** Devuelve el tamaño de una variable o un tipo.

**static** Crea almacenamiento permanente para una variable.

**static\_cast** Realiza una conversión de tipo.

**struct** Declara una estructura.

**switch** Sentencia de selección.

**T**

**template** Declara un tipo genérico.

**this** Puntero al objeto actual.

**throw** Lanza una excepción.

**true** El valor booleano verdadero.

**try** Ejecuta código que puede lanzar una excepción.

**typedef** Crea un nuevo nombre de tipo a partir de un tipo existente.

**typeid** Describe un objeto.

**typename** Declara una clase o un tipo no definido.

**U**

**union** Una estructura que asigna múltiples variables a la misma zona de memoria.

**unsigned** Declarara una variable sin signo.

**using** Importa un espacio de nombres en el ámbito actual.

**V**

**virtual** Crea una función miembro de una clase que puede ser sustituida por una clase derivada.

**void** Declara un valor sin un tipo de dato asociado.

**volatile** Declara una variable que puede verse modificada de forma inesperada.

## W

**wchar\_t** Declara una variable carácter amplia.

**while** Bucle con condición inicial o final.

# Apéndice B

## Operadores

En este apéndice se explican los diferentes operadores del lenguaje C++, con su precedencia y asociatividad.

### Índice General

---

B.1. Lista de operadores . . . . .	197
------------------------------------	-----

---

### B.1. Lista de operadores

A continuación se muestran los operadores de C++ agrupados por nivel de precedencia, de mayor a menor. Para cada operador se indica una pequeña descripción, un ejemplo y su asociatividad (ID: de izquierda a derecha, DI: de derecha a izquierda).

Operador	Descripción	Ejemplo	Asociatividad
()	Agrupamiento	a = (a + b) / 4;	ID
[]	Acceso a array	a[2] = 4;	ID
->	Acceso a miembros con un puntero	ptr->a = 4;	ID
.	Acceso a miembros con un objeto	obj.a = 4;	ID
::	Resolución de ámbito	Clase::a = 4;	ID
++	Postincremento	a = b++;	ID
--	Postdecremento	a = b--;	ID

Cuadro B.1: Nivel de precedencia 1

Operador	Descripción	Ejemplo	Asociatividad
!	Negación lógica	if(!a)	DI
<code>~</code>	Complemento a nivel de bits	a = ~b;	DI
<code>++</code>	Preincremento	a = ++b;	DI
<code>--</code>	Predecremento	a = --b;	DI
<code>-</code>	Menos unario	a = -4;	DI
<code>+</code>	Más unario	a = +4;	DI
<code>*</code>	Desreferencia	*ptr = 4;	DI
<code>&amp;</code>	Dirección de memoria	ptr = &a;	DI
<code>(type)</code>	Conversión de tipo	a = (int) 3.14;	DI
<code>sizeof</code>	Tamaño en bytes	a = sizeof(b);	DI

Cuadro B.2: Nivel de precedencia 2

Operador	Descripción	Ejemplo	Asociatividad
<code>-&gt;*</code>	Acceso a miembro puntero con un puntero	ptr->*a = 4;	ID
<code>.*</code>	Acceso a miembro puntero con un objeto	obj.a = 4;	ID

Cuadro B.3: Nivel de precedencia 3

Operador	Descripción	Ejemplo	Asociatividad
<code>*</code>	Multiplicación	a = b * 4;	ID
<code>/</code>	División	a = b / 4;	ID
<code>%</code>	Módulo	a = b % 4;	ID

Cuadro B.4: Nivel de precedencia 4

Operador	Descripción	Ejemplo	Asociatividad
<code>+</code>	Suma	a = b + 4;	ID
<code>-</code>	Resta	a = b - 4;	ID

Cuadro B.5: Nivel de precedencia 5

Operador	Descripción	Ejemplo	Asociatividad
<code>&lt;&lt;</code>	Desplazamiento de bits a la izquierda	a = b << 4;	ID
<code>&gt;&gt;</code>	Desplazamiento de bits a la derecha	a = b >> 4;	ID

Cuadro B.6: Nivel de precedencia 6

Operador	Descripción	Ejemplo	Asociatividad
<code>&lt;</code>	Menor	if(a < 4)	ID
<code>&lt;=</code>	Menor o igual	if(a <= 4)	ID
<code>&gt;</code>	Mayor	if(a > 4)	ID
<code>&gt;=</code>	Mayor o igual	if(a >= 4)	ID

Cuadro B.7: Nivel de precedencia 7

Operador	Descripción	Ejemplo	Asociatividad
<code>==</code>	Igual que	<code>if(a == 4)</code>	ID
<code>!=</code>	Distinto que	<code>if(a != 4)</code>	ID

Cuadro B.8: Nivel de precedencia 8

Operador	Descripción	Ejemplo	Asociatividad
<code>&amp;</code>	Y a nivel de bits	<code>a = b &amp; 4;</code>	ID

Cuadro B.9: Nivel de precedencia 9

Operador	Descripción	Ejemplo	Asociatividad
<code>^</code>	O exclusiva a nivel de bits	<code>a = b ^ 4;</code>	ID

Cuadro B.10: Nivel de precedencia 10

Operador	Descripción	Ejemplo	Asociatividad
<code> </code>	O a nivel de bits	<code>a = b   4;</code>	ID

Cuadro B.11: Nivel de precedencia 11

Operador	Descripción	Ejemplo	Asociatividad
<code>&amp;&amp;</code>	Y lógico	<code>if(a &lt; 4 &amp;&amp; b &lt; 4)</code>	ID

Cuadro B.12: Nivel de precedencia 12

Operador	Descripción	Ejemplo	Asociatividad
<code>  </code>	O lógico	<code>if(a &lt; 4    b &lt; 4)</code>	ID

Cuadro B.13: Nivel de precedencia 13

Operador	Descripción	Ejemplo	Asociatividad
<code>? :</code>	Condición (if..then..else)	<code>a = (a &lt; 4) ? 1 : 0;</code>	DI

Cuadro B.14: Nivel de precedencia 14

Operador	Descripción	Ejemplo	Asociatividad
=	Asignación	a = 4;	DI
+=	Suma y asignación	a += 4;	DI
-=	Resta y asignación	a -= 4;	DI
*=	Multiplicación y asignación	a *= 4;	DI
/=	División y asignación	a /= 4;	DI
%=	Módulo y asignación	a %= 4;	DI
&=	Y a nivel de bits y asignación	a &= 4;	DI
^=	O exclusiva a nivel de bits y asignación	a ^= 4;	DI
=	O a nivel de bits y asignación	a  = 4;	DI
<<=	Desplazamiento de bits a la izquierda y asignación	a <<= 4;	DI
>>=	Desplazamiento de bits a la derecha y asignación	a >>= 4;	DI

Cuadro B.15: Nivel de precedencia 15

Operador	Descripción	Ejemplo	Asociatividad
,	Evaluación secuencial	for(a = 0, b = 0; a < 4; a++)	ID

Cuadro B.16: Nivel de precedencia 16

# Apéndice C

## Sentencias

En este apéndice se incluye un resumen de la sintaxis de las sentencias de C++. El objetivo de este apéndice es que sirva como una guía rápida de búsqueda en caso de duda.

### Índice General

---

C.1. Introducción . . . . .	201
C.1.1. Asignación . . . . .	202
C.1.2. Sentencia compuesta (bloque de código) . . . . .	202
C.1.3. Sentencia condicional . . . . .	202
C.1.4. Sentencia condicional múltiple . . . . .	203
C.1.5. Sentencia de selección . . . . .	203
C.1.6. Bucle con contador . . . . .	204
C.1.7. Bucle con condición inicial . . . . .	204
C.1.8. Bucle con condición final . . . . .	205

---

### C.1. Introducción

Las sentencias del lenguaje C++ son:

- Asignación.
- Sentencia compuesta (bloque de código).
- Sentencia condicional.
- Sentencia condicional múltiple.

- Sentencia de selección.
- Bucle con contador.
- Bucle con condición inicial.
- Bucle con condición final.

En las siguientes secciones, para cada una de las sentencias, se presenta su sintaxis de forma informal junto con un pequeño ejemplo.

### C.1.1. Asignación

```
1 | variable = expresion;
```

Por ejemplo:

```
1 | int a, b;
2 |
3 | a = 5;
4 | b = a + 10;
```

### C.1.2. Sentencia compuesta (bloque de código)

```
1 | {
2 |   sentencia1
3 |   sentencia2
4 |   ...
5 |   ...
6 |   sentenciaN
7 | }
```

Por ejemplo:

```
1 | {
2 |   int a, b;
3 |
4 |   a = 5;
5 |   b = a + 10;
6 | }
```

### C.1.3. Sentencia condicional

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

Por ejemplo:

```
1 int a, b;  
2  
3 if(a <= 5)  
4     b = a + 10;
```

#### C.1.4. Sentencia condicional múltiple

```
1 if(expresion1)  
2     sentencia1  
3 else if(expresion2)  
4     sentencia2  
5 ...  
6 ...  
7 else  
8     sentenciaN
```

Por ejemplo:

```
1 int a, b;  
2  
3 if(a <= 5)  
4     b = a + 10;  
5 else if(a <= 10)  
6     b = a + 20;  
7 else  
8     b = a + 30;
```

#### C.1.5. Sentencia de selección

```
1 switch(expresion)  
2 {  
3     case constante1:  
4         sentencia1  
5         break;  
6  
7     case constante2:  
8         sentencia2  
9         break;  
10    ...  
11    ...  
12    default:  
13        sentenciaN  
14        break;  
15 }
```

Por ejemplo:

```

1 | int a, b;
2 |
3 | switch(a)
4 | {
5 |     case 1:
6 |     case 2:
7 |         b = a + 10;
8 |         break;
9 |
10 |    case 3:
11 |    case 4:
12 |        b = a + 20;
13 |        break;
14 |
15 |    case 5:
16 |    case 6:
17 |        b = a + 30;
18 |        break;
19 |
20 | default:
21 |     b = a + 40;
22 |     break;
23 |

```

### C.1.6. Bucle con contador

```

1 | for(inicializacion; condicion; actualizacion)
2 |     sentencia

```

Por ejemplo:

```

1 | int a, b;
2 |
3 | for(a = 0; a < 10; a++)
4 |     b = a + 10;

```

### C.1.7. Bucle con condición inicial

```

1 | while(expresion)
2 |     sentencia

```

Por ejemplo:

```

1 | int a, b;
2 |
3 | a = 0;

```

```
4 |     while(a < 10)
5 |     {
6 |         b = a + 10;
7 |         a++;
8 |     }
```

### C.1.8. Bucle con condición final

```
1 |     do
2 |         sentencia
3 |     while(expresion);
```

Por ejemplo:

```
1 |     int a, b;
2 |
3 |     a = 0;
4 |     do
5 |     {
6 |         b = a + 10;
7 |         a++;
8 |     }
9 |     while(a < 10);
```

# Apéndice D

## Herramientas

En este apéndice se comentan algunas de las herramientas que pueden ayudar a la hora de programar con el lenguaje C++: el editor JOE, el editor vim, el depurador gdb, el depurador Valgrind y el compresor/descompresor tar.

### Índice General

---

<b>D.1. Editor JOE</b> . . . . .	<b>208</b>
D.1.1. Comandos básicos . . . . .	208
D.1.2. Bloques de texto . . . . .	208
D.1.3. Movimiento . . . . .	210
D.1.4. Ayuda . . . . .	210
<b>D.2. Editor vim</b> . . . . .	<b>212</b>
D.2.1. Salir de vim . . . . .	212
D.2.2. Introducción de nuevo texto . . . . .	212
D.2.3. Movimientos del cursor . . . . .	214
D.2.4. Posicionamiento del cursor sobre palabras . . . . .	214
D.2.5. Deshacer . . . . .	214
D.2.6. Adiciones, cambios y supresiones simples de texto . . . . .	214
D.2.7. Búsquedas . . . . .	215
D.2.8. Opciones del editor . . . . .	215
<b>D.3. Depurador gdb</b> . . . . .	<b>215</b>
D.3.1. Ejemplo de depuración . . . . .	216
<b>D.4. Depurador Valgrind</b> . . . . .	<b>223</b>
D.4.1. Memcheck . . . . .	224
<b>D.5. Compresor/descompresor tar</b> . . . . .	<b>227</b>

---

## D.1. Editor JOE

JOE (*Joe's Own Editor*) es un editor de texto gratuito para sistemas Unix. Su curva de aprendizaje es muy baja comparada con otros editores típicos de Unix, como vi o emacs. Se basa en el empleo de secuencias de teclas similares a las empleadas en los programas WordStar o TurboC. JOE permite editar varios ficheros simultáneamente.

Para empezar a trabajar con este programa, tenemos que teclear desde la línea de comandos

joe

para empezar con un fichero en blanco o

joe fichero1 fichero2 ...

para continuar trabajando con ficheros creados previamente.

En la Figura D.1 aparece la página web<sup>1</sup> del editor JOE, donde se puede encontrar información sobre el programa y las últimas versiones disponibles.

A continuación incluimos los principales comandos de JOE organizados en tres categorías: comandos básicos, bloques de texto y movimiento. En la combinación de teclas que define cada comando, el símbolo ^ representa la pulsación de la tecla Control.

### D.1.1. Comandos básicos

- ^KE Editar un nuevo fichero
- ^KN Pasar a editar el siguiente fichero
- ^KH Ayuda
- ^KP Pasar a editar el anterior fichero
- ^KD Guardar archivo
- ^KX Guardar archivo y cerrarlo
- ^C Salir sin almacenar cambios

### D.1.2. Bloques de texto

- ^KB Principio de bloque
- ^KK Fin de bloque
- ^KC Copiar bloque
- ^KM Mover bloque
- ^KY Borrar bloque
- ^KW Guardar bloque en un fichero nuevo

---

<sup>1</sup><http://sourceforge.net/projects/joe-editor/>.

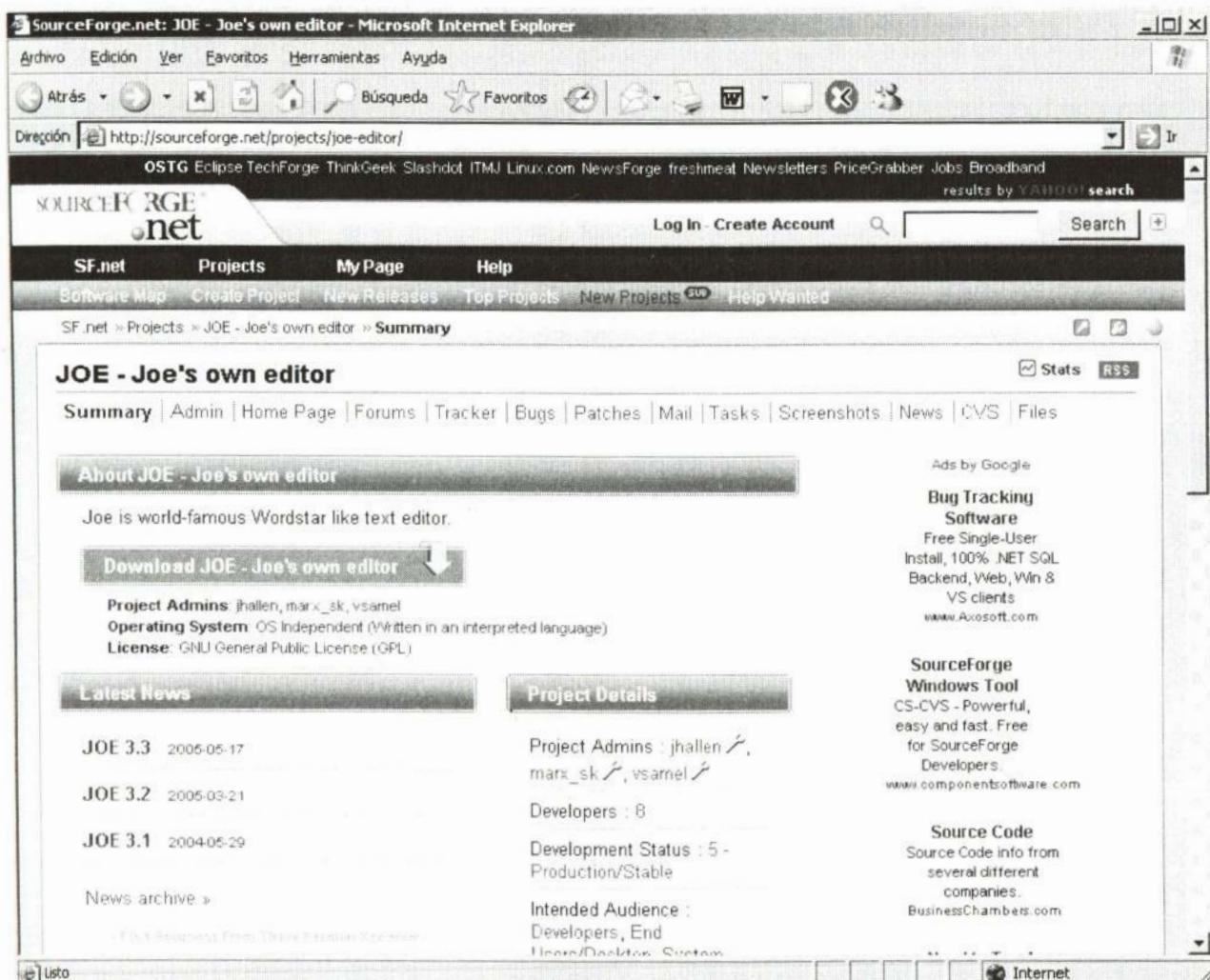


Figura D.1: Página web del editor JOE

CURSOR	GO TO	BLOCK	DELETE	MISC	EXIT
^B Left ^F right	^U prev. screen	^KB begin	^D char.	^KJ reformat	^KX save
^P up ^N down	^V next screen	^KK end	^Y line	^AT options	^AC abort
^Z previous word	^AA beg. of line	^KM move	^W >word	^AR refresh	^KZ shell
^X next word	^AE end of line	^KC copy	^O word<	^AQ insert	FILE
SEARCH	^KU top of file	^KW file	^J >line	SPELL	^KE edit
^KF Find text	^KV end of file	^KY delete	^_ undo	^IN word	^KR insert
^KL find next	^KL to line No.	^K/ filter	^A redo	^AL file	^KD save

Figura D.2: JOE: ficha de ayuda 1

^KO Split the window in half	^KE Load file into window
^KG Make current window bigger	^KT Make current window smaller
^KN Go to the window below	^KP Go to the window above
^KC Eliminate the current window	^KI Show all windows / show one window

Figura D.3: JOE: ficha de ayuda 2

### D.1.3. Movimiento

- ^A Inicio de línea
- ^E Fin de línea
- ^KU Inicio de documento
- ^KV Fin de documento
- ^U Retroceso página
- ^V Avance página
- ^KL Ir a la línea

### D.1.4. Ayuda

La ayuda es accesible con el comando ^KH y se compone de seis fichas, tal como mostramos en las figuras de la D.2 a la D.7.

MACROS	MISC	SCROLL	SHELL	GOTO	I-SEARCH
^K[ 0-9 Record	^K SPACE status	^W Up	^K Window	^B To	^R Backwards
^K]	^K\ Repeat	^Z Down	^![ Command	^K To	^KK A[S Forwards
^K 0-9 Play	^M Math	^K< Left	QUOTE	DELETE	BOOKMARKS
^K? Query	^KA Center line	^K> Right	Ctrl]-	^Y yank	^ 0-9 Goto
^D Dump	^H Message	^\\ Meta-	^O Line<	^O Line<	^A Set

Figura D.4: JOE: ficha de ayuda 3

```

GOTO      INDENT      COMPILING
^G Matching ( [ {     ^K, ^Tess ^[C Compile and parse errors
^K- Previous place   ^K. more ^[E Parse errors
^K= Next place        ^[= To next error
^K; Tags file search  ^[- To prev. error

```

Figura D.5: JOE: ficha de ayuda 4

```

Special search sequences:
  \^ $ matches beg./end of line    \?      match any single char
  \< > matches beg./end of word   \*      match 0 or more chars
  \c      matches balanced C expression  \\      matches a \
  \[..\] matches one of a set       \n      matches a newline
  \+      matches 0 or more of the character which follows the \+
special replace sequences:
  \&      replaced with text which matched search string
  \0 - 9 replaced with text which matched Nth \^*, \?, \c, \+, or \[..\]
  \\      replaced with \
                                              \n      replaced with newline

```

Figura D.6: JOE: ficha de ayuda 5

```

Hit TAB at file name prompts to generate menu of file names
Or use up/down keys to access history of previously entered names
Special file names:
  !command          Pipe in/out of a shell command
  >>filename        Append to a file
  -                Read/write to/from standard I/O
  filename,START,SIZE  Read/write a part of a file/device
                      Give START/SIZE in decimal (255), octal (0377) or hex (0xFF)

```

Figura D.7: JOE: ficha de ayuda 6

## D.2. Editor vim

**vim** es un editor de texto gratuito para sistemas Unix. **vim** es una mejora de **vi**, el editor de textos que se ha convertido en un estándar *de facto* en los sistemas Unix.

Para empezar a trabajar con este programa, tenemos que teclear desde la línea de comandos

**vim**

para empezar con un fichero en blanco o

**vim fichero1 fichero2 ...**

para continuar trabajando con ficheros creados previamente. En algunos sistemas, también se puede escribir **vi** en vez de **vim**.

Para ejecutar algún comando de **vim**, primero se tiene que pulsar la tecla **Esc** y a continuación la secuencia asociada al comando.

En la Figura D.8 aparece la página web<sup>2</sup> del editor **vim**, donde se pueden encontrar las últimas versiones del programa, documentación y consejos de uso.

### D.2.1. Salir de vim

- :wq<Enter>** Sale y guarda los cambios
- :x<Enter>** Sale y guarda los cambios
- :q!<Enter>** Sale y no guarda los cambios

### D.2.2. Introducción de nuevo texto

- a** Inserta texto después de la posición actual del cursor
- i** Inserta texto antes de la posición actual del cursor
- s** Reemplaza un carácter único por todos los caracteres que se escriban hasta pulsar **Esc**
- I** Inserta texto al comienzo de la línea
- A** Inserta texto al final de la línea
- o** Inserta una línea en blanco después de la línea actual y el cursor se coloca al comienzo de la nueva línea
- O** Inserta una línea en blanco antes de la línea actual y el cursor se coloca al comienzo de la nueva línea

---

<sup>2</sup><http://www.vim.org>.

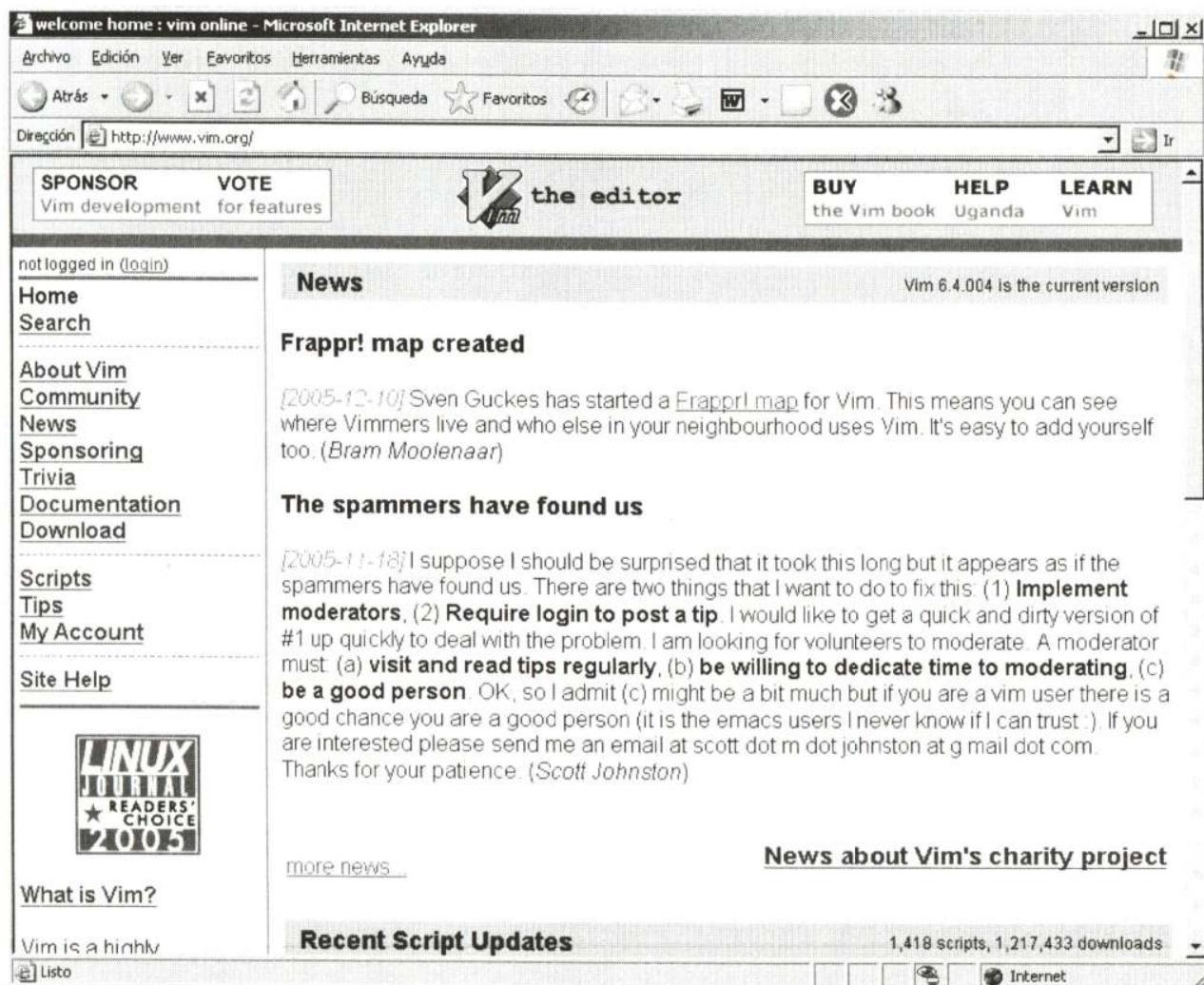


Figura D.8: Página web del editor vim

### D.2.3. Movimientos del cursor

- ~** Mueve el cursor al comienzo de la línea
- \$** Mueve el cursor al final de la línea
- <Enter>** Mueve el cursor al comienzo de la siguiente línea
- Mueve el cursor al comienzo de la línea anterior
- H** Mueve el cursor al primer carácter de la primera línea de la pantalla
- M** Mueve el cursor al primer carácter de la línea del medio de la pantalla
- L** Mueve el cursor al primer carácter de la última línea de la pantalla

### D.2.4. Posicionamiento del cursor sobre palabras

- w** Mueve el cursor hacia adelante al comienzo de la siguiente palabra
- e** Mueve el cursor al final de la siguiente palabra
- b** Mueve el cursor hacia atrás al comienzo de cada palabra

### D.2.5. Deshacer

- u** Deshace el último cambio realizado
- U** Anula todos los cambios que se han hecho en la línea actual

### D.2.6. Adiciones, cambios y supresiones simples de texto

- x** Borra un carácter
- X** Borra el carácter situado a la izquierda del cursor
- r** Reemplaza un carácter
- R** Reemplaza caracteres hasta que se le diga que pare al pulsar **Esc**
- dd** Borra una línea completa
- cc** Cambia una línea
- J** Une dos líneas, en la que está el cursor y la siguiente

### D.2.7. Búsquedas

- / Busca una cadena de texto
- :/ Coloca el cursor al comienzo de la línea que contiene la cadena a buscar
- ? Busca hacia atrás
- // Busca hacia adelante la siguiente ocurrencia de la misma cadena
- ?? Busca hacia atrás la siguiente ocurrencia de la misma cadena
- n Busca la siguiente ocurrencia de la misma cadena en la misma dirección
- N Busca la siguiente ocurrencia de la misma cadena en la dirección opuesta

### D.2.8. Opciones del editor

**vim** posee una serie de opciones que permiten configurar su funcionamiento. Cada opción tiene una abreviatura que se activa con la orden **set**. Las opciones se desactivan precediendo la abreviatura de la opción con la palabra **no**. Por ejemplo, para que las líneas de texto salgan numeradas, se emplea la opción:

```
:set number
```

y para desactivar esta opción:

```
:set nonumber
```

Para visualizar todas las opciones disponibles y su estado actual se emplea la orden **set all**, tal como se muestra en la Figura D.9.

## D.3. Depurador **gdb**

**gdb** (*The GNU Project Debugger*) es el depurador estándar de los sistemas basados en el software de GNU. Se puede emplear con diferentes lenguajes de programación, como C, C++ y Fortran. En la Figura D.10 aparece la página web<sup>3</sup> del depurador **gdb**, donde se puede consultar la documentación y consejos de uso.

El propósito de un depurador como **gdb** es permitir que el programador pueda “ver” qué está ocurriendo dentro de un programa mientras se está ejecutando o “qué pasaba” cuando un programa falló.

Los comandos básicos de **gdb** son:

---

<sup>3</sup><http://www.gnu.org/software/gdb/gdb.html>

```

--- options ---
aleph=224      cscopetagorder=0      foldmethod=manual  nolinebreak      printoptions=
noallowrevins nocscopeverbose      foldminlines=1    lines=68          nosplitbelow
noaltkeymap     debug=              foldnestmax=20   nolisp           noreadonly
nodecombine     dictionary=        formatoptions=tco  listchars=eo:$       nosplitright
noautoread     nodiff=            formatprg=        loadplugins      remap
noautowrite    diffexpr=filler   nogdefault=      magic           report=2
noautowriteall helpheight=20     nohidden=        makeef=         suffixesadd=
back-ground=light history=50     nohlsearch=      makeprg=make   ruler
back-space=2    nodigraph=       iconstring=      scroll=33
noback-up      display=         noignorecase=   scrollbind=    scrolljump=1
back-upcopy=auto eadirection=both nohlsearch=   maxfuncdepth=100 scrolloff=0
back-upext=    noedcompatible   noicon=         maxmapdepth=1000 nosecure
nobinary       encoding=latin1  endofline=      maxmem=516874  selectmode=
nobomb=        equalalways=    iconstring=    menuitems=25   shell=/bin/bash
bufhidden=     noerrorbeep=    ignorecase=   modeline=5    shellcmdflag=-c
buflist=        noeskeys=       noinsearch=    modifiable=   shellquote=
buftype=       eventignore=   noinscase=     indentexpr=  more
charconvert=   noexec=         noinsertmode=  nonumber=    mouse=
noindent       fileencoding=   isprint=0x161-255 nosettime=500 noshowcmd
cinoptions=    filetype=       foldcolumn=0   key=         noshowmatch
cmdheight=1    foldenable=     foldexpr=0     keymodel=    pastetoggle=
cmdwinheight=7 foldexpr=0     keywordprg=man langmap=    previewheight=12 noshowtag
nocompatible   foldexpr=#
noconfirm=     foldexpr=#
coptions=aABceFS cscopepath=comp-o foldignore=#
foldlevel=0    foldlevelstart=-1 foldlevelstart=-1 laststatus=1 nosmartcase
nocscopeprg=cscope
noxcscopeprg=cscope

```

Figura D.9: Principales opciones del editor vim

<b>r</b> (run)	Inicia la ejecución de un programa. Permite pasarle parámetros al programa. Ejemplo: <b>r fichero.txt</b> .
<b>l</b> (list)	Lista el contenido del fichero con los números de línea.
<b>b</b> (breakpoint)	Fija un punto de parada. Ejemplo: <b>b 10</b> (breakpoint en la línea 10), <b>b main</b> (breakpoint en la función main).
<b>bt</b> (backtrace)	Muestra la pila de ejecución. Con el modificador <b>full</b> muestra también el valor de las variables locales.
<b>c</b> (continue)	Continúa la ejecución de un programa.
<b>n</b> (next)	Ejecuta la siguiente orden; si es una función la salta (no muestra las líneas de la función) y continúa con la siguiente orden.
<b>s</b> (step)	Ejecuta la siguiente orden; si es una función entra en ella y la podemos ejecutar línea a línea.
<b>p</b> (print)	Muestra el contenido de una variable. Ejemplo: <b>p auxiliar</b> , <b>p this</b> (muestra la dirección del objeto), <b>p *this</b> (muestra el objeto completo).
<b>where</b>	Muestra la pila de ejecución (como bt).
<b>h</b> (help)	Ayuda general.

### D.3.1. Ejemplo de depuración

Depurar es un arte y cada programador tiene que encontrar su “estilo”. Proporcionar un método que se pueda aplicar a todos los casos es imposible.

El siguiente código presenta varios errores lógicos que no impiden que se compile el

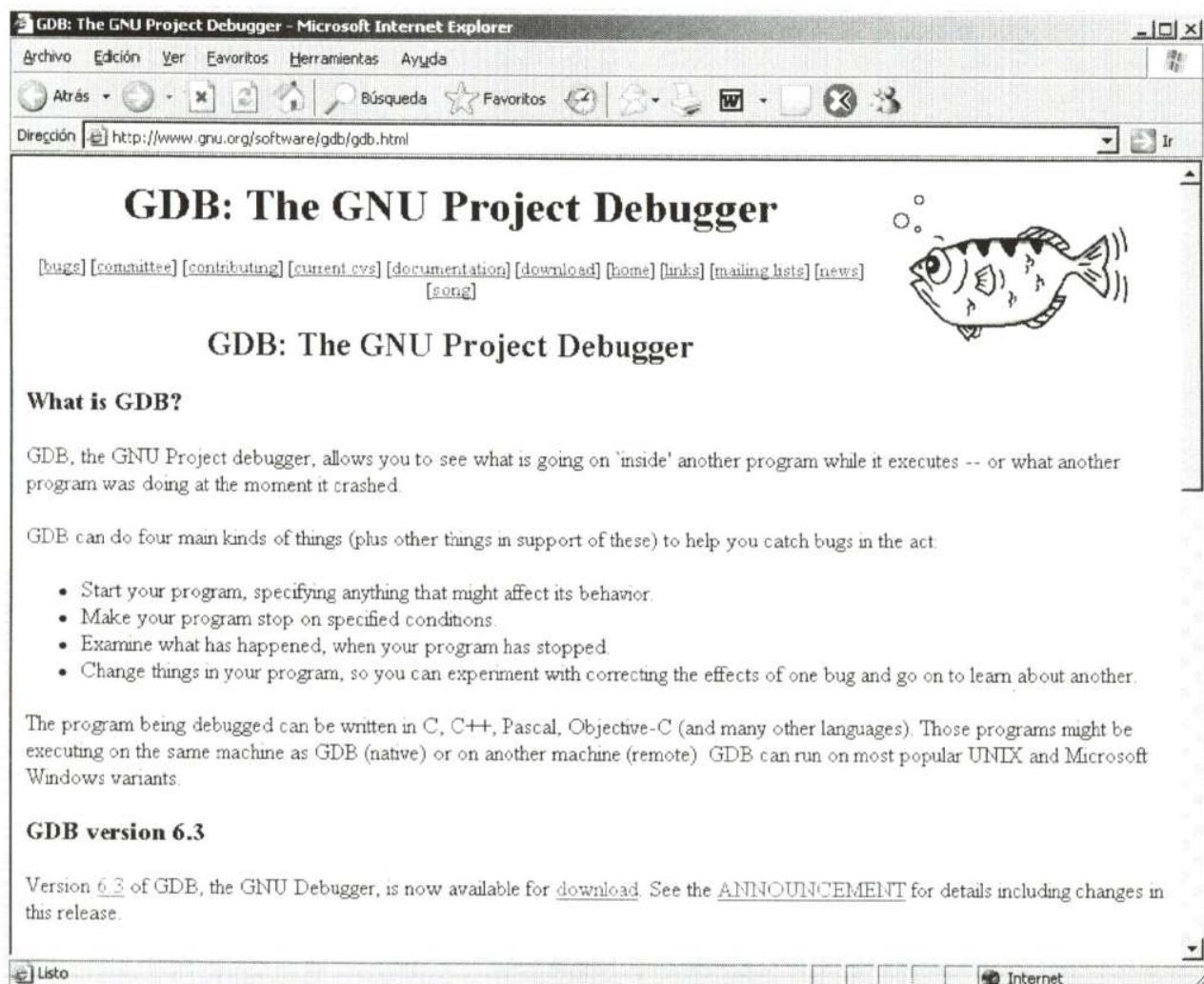


Figura D.10: Página web de gdb

código, pero que generan el temido Segmentation fault (core dumped). El código contiene una función `copia(char *, char *)` que tiene que copiar el contenido de la primera cadena sobre la segunda cadena que puede ya contener datos, por lo que se tiene que liberar la memoria que pueda estar ocupando:

---

Ejemplo D.1

---

```

1 #include <iostream>
2 #include <cstring>
3
4 using namespace std;
5
6 void copia(char *a, char *b) {
7     if(a != NULL)
8         delete a;
9
10    a = new char[strlen(b) + 1];
11    strcpy(a, b);
12 }
13
14 int main(void) {
15     char m[] = "Un mensaje";
16     char *n;
17
18     copia(m, n);
19 }
```

---

Para empezar a trabajar con este programa, tenemos que teclear desde la línea de comandos

`gdb`

Para empezar directamente con un fichero ejecutable cargado, tenemos que teclear  
`gdb fichero`

En la Figura D.11 mostramos la primera parte de una sesión de depuración con `gdb`. El primer paso es ejecutar el código directamente, sin ningún punto de parada, para tener una idea de qué y dónde se produjo la violación de segmento:

---

Salida ejemplo D.1

---

```

1 (gdb) r
2 Starting program: /home/tad/libro/gdb/error
3
4 Program received signal SIGSEGV, Segmentation fault.
5 0x4017c140 in _int_free () from /lib/libc.so.6
6 (gdb)
```

```
tad@localhost:~/libro/gdb$ g++ -g -o error error.cc
tad@localhost:~/libro/gdb$ error
Segmentation fault (core dumped)
tad@localhost:~/libro/gdb$ gdb error
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...
(gdb) r
Starting program: /home/tad/libro/gdb/error

Program received signal SIGSEGV, Segmentation fault.
0x4017c140 in _int_free () from /lib/libc.so.6
(gdb) b main
Breakpoint 1 at 0x80485fe: file error.cc, line 17.
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/tad/libro/gdb/error

Breakpoint 1, main () at error.cc:17
17      char m[] = "Un mensaje";
(gdb) s
20      copia(m, n);
(gdb)
copia(char*, char*) (a=0xbfffffa90 "Un mensaje", b=0x8048451 "ÉA") at error.cc:8
8      if(a != NULL)
(gdb)
9      delete a;
(gdb)

Program received signal SIGSEGV, Segmentation fault.
0x4017c140 in _int_free () from /lib/libc.so.6
(gdb) p a
No symbol "a" in current context.
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/tad/libro/gdb/error

Breakpoint 1, main () at error.cc:17
17      char m[] = "Un mensaje";
(gdb) s
20      copia(m, n);
(gdb)
copia(char*, char*) (a=0xbfffffa90 "Un mensaje", b=0x8048451 "ÉA") at error.cc:8
8      if(a != NULL)
(gdb) p a
$1 = 0xbfffffa90 "Un mensaje"
(gdb)
```

Figura D.11: Ejemplo de sesión de depuración con gdb

El problema se ha producido al llamar al método `_int_free()` que se emplea para liberar memoria. En el código del ejemplo se libera memoria en la sentencia `delete a;` de la línea 8. Para asegurarnos de que la violación de segmento se produjo en esa línea, vamos a realizar una ejecución paso a paso desde el principio. Para ello, colocamos un punto de parada en la función `main()` y ejecutamos el programa con la orden `step (s)`:

Salida ejemplo D.1

```

1  (gdb) b main
2  Breakpoint 1 at 0x80485fe: file error.cc, line 17.
3  (gdb) r
4  The program being debugged has been started already.
5  Start it from the beginning? (y or n) y
6  Starting program: /home/tad/libro/gdb/error
7
8  Breakpoint 1, main () at error.cc:17
9    17      char m[] = "Un mensaje";
10   (gdb) s
11   20      copia(m, n);
12   (gdb) s
13   copia(char*, char*) (a=0xbfffffa90 "Un mensaje", b=0x8048451 "ÉÃ")
14     at error.cc:8
15   8      if(a != NULL)
16   (gdb) s
17   9      delete a;
18   (gdb) s
19
20  Program received signal SIGSEGV, Segmentation fault.
21  0x4017c140 in _int_free () from /lib/libc.so.6
22  (gdb)

```

Efectivamente, la violación de segmento se produce en la línea `delete a;`. ¿Dónde está el problema? El siguiente paso es averiguar el valor de la variable `a`; para ello debemos de comenzar desde el principio otra vez y consultar el valor de `a` antes de que se produzca la violación de segmento:

Salida ejemplo D.1

```

1  (gdb) r
2  The program being debugged has been started already.
3  Start it from the beginning? (y or n) y
4  Starting program: /home/tad/libro/gdb/error
5
6  Breakpoint 1, main () at error.cc:17
7    17      char m[] = "Un mensaje";
8  (gdb) s

```

```
9      20      copia(m, n);
10     (gdb) s
11     copia(char*, char*) (a=0xbfffffa90 "Un mensaje", b=0x8048451 "ÉÃ")
12       at error.cc:8
13     8      if(a != NULL)
14     (gdb) p a
15     $1 = 0xbfffffa90 "Un mensaje"
16     (gdb)
```

¡La variable **a** es la cadena “Un mensaje” que queremos copiar! Esto también lo podríamos haber detectado en los parámetros de la llamada a la función **copia()**:

---

Salida ejemplo D.1

---

```
1 copia(char*, char*) (a=0xbfffffa90 "Un mensaje", b=0x8048451 "ÉÃ")
2   at error.cc:8
```

Además, vemos que la variable **b**, que representa la cadena destino de la copia, contiene un valor extraño, porque se nos ha olvidado inicializar el puntero **n** a **NULL** en **main()**. El código una vez corregido estos dos errores es:

---

Ejemplo D.2

---

```
1 #include <iostream>
2 #include <cstring>
3
4 using namespace std;
5
6 void
7 copia(char *a, char *b) {
8     if(b != NULL)          // Corregido
9         delete b;          // Corregido
10
11     a = new char[strlen(b) + 1];
12     strcpy(a, b);
13 }
14
15 int
16 main(void) {
17     char m[] = "Un mensaje";
18     char *n = NULL;        // Corregido
19
20     copia(m, n);
21 }
```

Sin embargo, si compilamos y ejecutamos este código, volvemos a obtener una violación de segmento. ¿Qué pasa ahora? Volvamos a hacer una traza para localizar el origen del error:

Salida ejemplo D.2

```

1 (gdb) r
2 Starting program: /home/tad/libro/gdb/error
3
4 Program received signal SIGSEGV, Segmentation fault.
5 0x40182203 in strlen () from /lib/libc.so.6
6 (gdb)

```

Ahora el problema está al llamar al método `strlen()` que se emplea para calcular la longitud de la cadena que queremos copiar. Veamos el valor que toman las variables en esa instrucción:

Salida ejemplo D.2

```

1 (gdb) b main
2 Breakpoint 1 at 0x80485fe: file error.cc, line 17.
3 (gdb) r
4 The program being debugged has been started already.
5 Start it from the beginning? (y or n) y
6 Starting program: /home/tad/libro/gdb/error
7
8 Breakpoint 1, main () at error.cc:17
9      17     char m[] = "Un mensaje";
10 (gdb) s
11      18     char *n = NULL;
12 (gdb)
13      20     copia(m, n);
14 (gdb)
15 copia(char*, char*) (a=0xbfffffa90 "Un mensaje", b=0x0) at error.cc:8
16      8     if(b != NULL)
17 (gdb)
18      11     a = new char[strlen(b) + 1];
19 (gdb) p a
20 $1 = 0xbfffffa90 "Un mensaje"
21 (gdb) p b
22 $2 = 0x0
23 (gdb)

```

¡Está cambiado el orden de las variables! Las funciones de tratamiento de cadenas (`strlen()`, `strcpy()`, etc.) generan un error cuando un puntero a cadena está a `NULL`. Tenemos que calcular la longitud de la cadena origen `a` y reservar memoria para la cadena destino `b`. El código una vez corregido es:

---

Ejemplo D.3

---

```
1 #include <iostream>
2 #include <cstring>
3
4 using namespace std;
5
6 void
7 copia(char *a, char *b) {
8     if(b != NULL)
9         delete b;
10
11     b = new char[strlen(a) + 1];
12     strcpy(a, b);
13 }
14
15 int
16 main(void) {
17     char m[] = "Un mensaje";
18     char *n = NULL;
19
20     copia(m, n);
21 }
```

---

¿Se podría mejorar este código? ¿Y si la cadena que queremos copiar apunta a NULL?

## D.4. Depurador Valgrind

Valgrind es un conjunto de herramientas para la depuración y mejora del rendimiento de los programas (aumento de la velocidad y menor uso de memoria). Valgrind se compone de cinco herramientas:

- **Memcheck:** detecta problemas con la gestión de memoria.
- **Addrcheck:** versión ligera de Memcheck (se ejecuta más rápido y necesita menos memoria), pero no es tan potente.
- **Cachegrind:** analizador de la caché, simula la caché del ordenador para detectar posibles puntos donde se pueda optimizar el código.
- **Massif:** analizador de la memoria *heap*.
- **Helgrind:** depurador de hilos de ejecución, busca condiciones de carrera en programas multihilo.



Figura D.12: Página web de Valgrind

En la Figura D.8 aparece la página web<sup>4</sup> de Valgrind, donde se puede encontrar el manual de usuario e información técnica sobre las distintas herramientas.

A continuación vamos a comentar la herramienta Memcheck, la más útil de cara a la depuración de errores.

#### D.4.1. Memcheck

Memcheck es la utilidad más importante de las que incluye Valgrind. Memcheck permite detectar:

- Empleo de memoria no inicializada.

<sup>4</sup><http://valgrind.org>.

- Lectura y escritura de memoria después de que haya sido liberada.
- Lectura y escritura de zonas de memoria incorrectas.
- Pérdidas de memoria (zonas de memoria que se han perdido para siempre).
- Otros tipos de errores.

Veamos el funcionamiento de Valgrind con un ejemplo. El siguiente código contiene dos errores, porque no libera la memoria reservada y accede a una zona de memoria no reservada:

---

Ejemplo D.4

---

```
1 #include <iostream>
2
3 using namespace std;
4
5 void f(void) {
6     int *x;
7
8     x = new int[10];
9     x[10] = 0;
10 }
11
12 int main(void) {
13     f();
14
15     return 0;
16 }
```

---

Para ejecutar este programa con Valgrind y Memcheck se tiene que ejecutar la siguiente orden:

```
valgrind --tool=memcheck --leak-check=yes prueba
```

donde *prueba* es el nombre del programa que queremos depurar.

Valgrind genera unos mensajes de error bastante extensos que hay que leer con cuidado, ya que incluye mucha información superflua<sup>5</sup>:

---

Salida ejemplo D.4

---

```
1 ==27647== Memcheck, a memory error detector for x86-linux.
2 ==27647== Copyright (C) 2002-2004, and GNU GPL'd, by Julian Seward et al.
3 ==27647== Using valgrind-2.2.0, a program supervision framework for
4     x86-linux.
```

---

<sup>5</sup>Si incluimos la opción *-v (verbose)*, se generan mensajes de error aún más largos.

```

5  ==27647== Copyright (C) 2000-2004, and GNU GPL'd, by Julian Seward et al.
6  ==27647== For more details, rerun with: -v
7  ==27647==
8  ==27647== Invalid write of size 4
9  ==27647==     at 0x8048520: f() (prueba.cc:11)
10 ==27647==     by 0x804853C: main (prueba.cc:17)
11 ==27647== Address 0x1BB38050 is 0 bytes after a block of size 40 alloc'd
12 ==27647==     at 0x1B9040DA: operator new[](unsigned)
13             (vg_replace_malloc.c:139)
14 ==27647==     by 0x8048513: f() (prueba.cc:10)
15 ==27647==     by 0x804853C: main (prueba.cc:17)
16 ==27647==
17 ==27647== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 17 from 1)
18 ==27647== malloc/free: in use at exit: 40 bytes in 1 blocks.
19 ==27647== malloc/free: 1 allocs, 0 frees, 40 bytes allocated.
20 ==27647== For counts of detected errors, rerun with: -v
21 ==27647== searching for pointers to 1 not-freed blocks.
22 ==27647== checked 2328396 bytes.
23 ==27647==
24 ==27647==
25 ==27647== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
26 ==27647==     at 0x1B9040DA: operator new[](unsigned)
27             (vg_replace_malloc.c:139)
28 ==27647==     by 0x8048513: f() (prueba.cc:10)
29 ==27647==     by 0x804853C: main (prueba.cc:17)
30 ==27647==
31 ==27647== LEAK SUMMARY:
32 ==27647==     definitely lost: 40 bytes in 1 blocks.
33 ==27647==     possibly lost: 0 bytes in 0 blocks.
34 ==27647==     still reachable: 0 bytes in 0 blocks.
35 ==27647==             suppressed: 0 bytes in 0 blocks.
36 ==27647== Reachable blocks (those to which a pointer was found) are not
37             shown.
38 ==27647== To see them, rerun with: --show-reachable=yes

```

En los mensajes de escritura en una zona de memoria, se indica el lugar y la cantidad de memoria que se ha intentado escribir. Por ejemplo, en el siguiente mensaje de error se nos indica que el programa ha intentado escribir (**Invalid write**) 4 bytes en una zona de memoria no reservada y se ha producido en la función **f()** que fue llamada desde **main**:

#### Salida ejemplo D.4

```

1  ==27647== Invalid write of size 4
2  ==27647==     at 0x8048520: f() (prueba.cc:11)
3  ==27647==     by 0x804853C: main (prueba.cc:17)

```

```

4 ==27647== Address 0x1BB38050 is 0 bytes after a block of size 40 alloc'd
5 ==27647== at 0x1B9040DA: operator new[](unsigned)
6   (vg_replace_malloc.c:139)
7 ==27647== by 0x8048513: f() (prueba.cc:10)
8 ==27647== by 0x804853C: main (prueba.cc:17)

```

En los mensaje de pérdida de memoria, se indica el lugar y la cantidad de memoria que se ha perdido, pero no la causa (no se puede saber). Por ejemplo, en el siguiente mensaje de error se nos indica que se han perdido 40 bytes (10 enteros por 4 bytes cada entero) en la función `f()`, que es la memoria del puntero `x` para el que se ha reservado un array de enteros de 10 posiciones, que no se ha liberado:

■ Salida ejemplo D.4 ■

```

1 ==27647== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
2 ==27647== at 0x1B9040DA: operator new[](unsigned)
3   (vg_replace_malloc.c:139)
4 ==27647== by 0x8048513: f() (prueba.cc:10)
5 ==27647== by 0x804853C: main (prueba.cc:17)

```

## D.5. Compresor/descompresor tar

El formato de ficheros `tar` se emplea para archivar varios fichero en uno solo. El nombre proviene de *Tape ARchive*, porque se empleaba antiguamente para almacenar datos en cinta. Estos ficheros se generan con el comando `tar` de Unix y su formato está estandarizado en la especificación POSIX.1-1998 y POSIX.1-2001. El formato `tar` conserva la estructura de ficheros, los permisos, los propietarios y demás información asociada a los ficheros que almacena.

El formato `tar` por si solo no comprime. Para ello se emplean otras herramientas como `gzip` o `bzip2` que comprimen el fichero `tar` a posteriori. Sin embargo, el `tar` de GNU disponible en Linux puede comprimir directamente. El funcionamiento básico de `tar` es:

- Usar el comando `tar` y `mcopy` para almacenar todos los ficheros en un único fichero y copiarlo en un disco:

```

1 | $ tar cvzf practica.tgz *
2 | $ mcopy practica.tgz a:/

```

- Para recuperarlo del disco posteriormente:

```

1 | $ mcopy a:/practica.tgz
2 | $ tar xvzf practica.tgz

```

Cuando se copien ficheros binarios (.tgz, .gif, .jpg, etc.) no se debe emplear el parámetro -t en mcopy, ya que sirve para convertir ficheros de texto de Linux a DOS y viceversa.

# Apéndice E

## Código de las clases

En este apéndice se incluye el código completo de las cuatro clases desarrolladas a lo largo del libro: TCoordenada, TLinea, TVector y TCalendario.

### Índice General

---

E.1. La clase TCoordenada . . . . .	229
E.2. La clase TLinea . . . . .	235
E.3. La clase TVector . . . . .	237
E.4. La clase TCalendario . . . . .	242

---

### E.1. La clase TCoordenada

Definid en C++ la clase TCoordenada que representa puntos en el espacio.  
Fichero tcoordenada.h:

---

Ejemplo E.1

---

```
1 #include <iostream>
2
3 using namespace std;
4
5 class TCoordenada {
6     friend istream& operator>>(istream &, TCoordenada &);
7     friend ostream& operator<<(ostream &, const TCoordenada &);
8
9     friend TCoordenada operator+(int, const TCoordenada &);
10
```

```
11 public:
12     TCoordenada();
13     TCoordenada(int, int, int);
14     TCoordenada(const TCoordenada &);
15     ~TCoordenada();
16
17     TCoordenada& operator=(const TCoordenada &);
18     TCoordenada operator+(const TCoordenada &);
19     TCoordenada operator-(const TCoordenada &);
20     TCoordenada& operator++(void);
21     TCoordenada operator++(int);
22     TCoordenada& operator+=(const TCoordenada &);
23     TCoordenada& operator-=(const TCoordenada &);
24     bool operator==(const TCoordenada &);
25     bool operator!=(const TCoordenada &);
26     int& operator[](char);
27     int operator[](char) const;
28
29     TCoordenada operator+(int);
30
31     void setX(int);
32     void setY(int);
33     void setZ(int);
34
35     int getX(void);
36     int getY(void);
37     int getZ(void);
38
39     void Imprimir(void);
40
41 private:
42     int x, y, z;
43     int error;
44 };
```

---

Fichero tcoordenada.cc:

---

Ejemplo E.2

---

```
1 #include "tcoordenada.h"
2
3 TCoordenada::TCoordenada() {
4     x = y = z = 0;
5 }
6
7 TCoordenada::TCoordenada(int a, int b, int c) {
```

```
8     x = a;
9     y = b;
10    z = c;
11 }
12
13 TCoordenada::TCoordenada(const TCoordenada &c) {
14     x = c.x;
15     y = c.y;
16     z = c.z;
17 }
18
19 TCoordenada::~TCoordenada() {
20     x = y = z = 0;
21 }
22
23 void
24 TCoordenada::setX(int xx) {
25     x = xx;
26 }
27
28 void
29 TCoordenada::setY(int yy) {
30     y = yy;
31 }
32
33 void
34 TCoordenada::setZ(int zz) {
35     z = zz;
36 }
37
38 int
39 TCoordenada::getX(void) {
40     return x;
41 }
42
43 int
44 TCoordenada::getY(void) {
45     return y;
46 }
47
48 int
49 TCoordenada::getZ(void) {
50     return z;
51 }
52
53 void
```

```
54 TCoordenada::Imprimir(void) {
55     cout << "(" << x;
56     cout << ", " << y;
57     cout << ", " << z;
58     cout << ")";
59 }
60
61 TCoordenada&
62 TCoordenada::operator=(const TCoordenada &op2) {
63     if(this != &op2)
64     {
65         (*this).~TCoordenada();
66
67         x = op2.x;
68         y = op2.y;
69         z = op2.z;
70     }
71
72     return *this;
73 }
74
75 TCoordenada
76 TCoordenada::operator+(const TCoordenada &op2) {
77     TCoordenada temp;
78
79     temp.x = x + op2.x;
80     temp.y = y + op2.y;
81     temp.z = z + op2.z;
82
83     return temp;
84 }
85
86 TCoordenada
87 TCoordenada::operator-(const TCoordenada &op2) {
88     TCoordenada temp;
89
90     temp.x = x - op2.x;
91     temp.y = y - op2.y;
92     temp.z = z - op2.z;
93
94     return temp;
95 }
96
97 TCoordenada&
98 TCoordenada::operator++(void) {
99     x++;

```

```
100     y++;
101     z++;
102
103     return *this;
104 }
105
106 TCoordenada
107 TCoordenada::operator++(int op2) {
108     TCoordenada temp(*this);
109
110     x++;
111     y++;
112     z++;
113
114     return temp;
115 }
116
117 TCoordenada&
118 TCoordenada::operator+=(const TCoordenada &op2) {
119     x += op2.x;
120     y += op2.y;
121     z += op2.z;
122
123     return *this;
124 }
125
126 TCoordenada&
127 TCoordenada::operator-=(const TCoordenada &op2) {
128     x -= op2.x;
129     y -= op2.y;
130     z -= op2.z;
131
132     return *this;
133 }
134
135 bool
136 TCoordenada::operator==(const TCoordenada &op2) {
137     bool temp;
138
139     temp = (x==op2.x && y==op2.y && z==op2.z) ? true : false;
140
141     return temp;
142 }
143
144 bool
145 TCoordenada::operator!=(const TCoordenada &op2) {
```

```
146     return !(*this == op2);
147 }
148
149 istream&
150 operator>>(istream &s, TCoordenada &obj) {
151     cout << "Introducir coordenada x:";
152     s >> obj.x;
153
154     cout << "Introducir coordenada y:";
155     s >> obj.y;
156
157     cout << "Introducir coordenada z:";
158     s >> obj.z;
159
160     return s;
161 }
162
163 ostream&
164 operator<<(ostream &s, const TCoordenada &obj) {
165     s << "(" << obj.x << ", ";
166     s << obj.y << ", ";
167     s << obj.z << ")";
168
169     return s;
170 }
171
172 int&
173 TCoordenada::operator[](char c) {
174     if(c == 'x' || c == 'X')
175         return x;
176     if(c == 'y' || c == 'Y')
177         return y;
178     if(c == 'z' || c == 'Z')
179         return z;
180
181     error = 0;
182     return error;
183 }
184
185 int
186 TCoordenada::operator[](char c) const {
187     if(c == 'x' || c == 'X')
188         return x;
189     if(c == 'y' || c == 'Y')
190         return y;
191     if(c == 'z' || c == 'Z')
```

```
192     return z;
193
194     return 0;
195 }
196
197 TCoordenada
198 TCoordenada::operator+(int n) {
199     TCoordenada temp(*this);
200
201     temp.x += n;
202     temp.y += n;
203     temp.z += n;
204
205     return temp;
206 }
207
208 TCoordenada
209 operator+(int n, const TCoordenada &obj) {
210     TCoordenada temp(n, n, n);
211
212     return temp + obj;
213 }
```

## E.2. La clase TLinea

Definid en C++ la clase **TLinea** que representa una línea definida mediante dos puntos representados por dos objetos de tipo **TCoordenada**.

Con el fin de reducir el acoplamiento entre las clases **TCoordenada** y **TLinea**, se ha eliminado la relación de amistad entre las dos clases y se ha sustituido en el método **Longitud()** la llamada al método **Distancia()** por el código de ese método.

Fichero **tlinea.h**:

---

### Ejemplo E.3

---

```
1 #ifndef __TLINEA__
2 #define __TLINEA__
3
4 #include <iostream>
5
6 using namespace std;
7
8 #include "tcoordenada.h"
9
10 class TLinea {
11     friend ostream& operator<<(ostream &, const TLinea &);
```

```
12
13     public:
14         TLinea();
15         TLinea(const TCoordenada &, const TCoordenada &);
16         TLinea(const TLinea &);
17         ~TLinea();
18         float Longitud(void);
19
20     private:
21         TCoordenada p1, p2;
22     };
23 #endif
```

---

Fichero **tlinea.cc**:

---

Ejemplo E.4

---

```
1 #include <cmath>
2
3 using namespace std;
4
5 #include "tlinea.h"
6
7 TLinea::TLinea() {
8     // No hace nada
9 }
10
11 TLinea::TLinea(const TCoordenada & a, const TCoordenada & b): p1(a), p2(b) {
12     // No hace nada
13 }
14
15 TLinea::TLinea(const TLinea & l): p1(l.p1), p2(l.p2) {
16     // No hace nada
17 }
18
19 TLinea::~TLinea() {
20     // No hace nada
21 }
22
23
24 float
25 TLinea::Longitud(void) {
26     float d;
27
28     d = pow((float) (p1.x - p2.x), 2);
29     d += pow((float) (p1.y - p2.y), 2);
```

```
30     d += pow((float) (p1.z - p2.z), 2);
31
32
33     return sqrt(d);
34 }
35
36 ostream&
37 operator<<(ostream &s, const TLinea &obj) {
38     s << "(" << obj.p1 << ", ";
39     s << obj.p2 << ")";
40
41     return s;
42 }
```

## E.3. La clase TVector

Definid en C++ la clase **TVector** que contiene un vector dinámico de números enteros y un número entero que contiene la dimensión del vector.

Respecto el código que ha aparecido a lo largo de varios capítulos en este libro:

- La sobrecarga para el operador salida de la clase sustituye al método **Imprimir()** empleado en algunos ejemplos.
- La sobrecarga para el operador corchete de la clase sustituye a los métodos **Almacenar()** y **Recuperar()** empleados en algunos ejemplos.

Fichero **tvector.h**:

---

### Ejemplo E.5

---

```
1 #include <iostream>
2
3 using namespace std;
4
5 #ifndef __TVECTOR__
6 #define __TVECTOR__
7
8 class TVector
9 {
10     friend ostream & operator<<(ostream &, const TVector &);
11
12 public:
13     TVector();
14     TVector(int);
15     TVector(const TVector &);
```

```
16     ~TVector();
17     int Dimension(void);
18     TVector operator+(const TVector &);
19     TVector operator-(const TVector &);
20     TVector& operator=(const TVector &);
21     int& operator[](int);
22     bool operator==(const TVector &);
23     bool operator!=(const TVector &);
24
25 private:
26     int dimension;
27     int *datos;
28     int error;
29 };
30
31 #endif
```

Fichero tvector.cc:

---

Ejemplo E.6

---

```
1 #include <iostream>
2
3 using namespace std;
4
5 #include "tvector.h"
6
7 TVector::TVector() {
8     dimension = 10;
9     datos = new int[dimension];
10    if(datos == NULL)
11        dimension = 0;
12    else
13        for(int i = 0; i < dimension; i++)
14            datos[i] = -1;
15 }
16
17 TVector::TVector(int dim) {
18     if(dim <= 0)
19     {
20         dimension = 0;
21         datos = NULL;
22     }
23     else
24     {
25         dimension = dim;
```

```
26     datos = new int[dimension];
27     if(datos == NULL)
28         dimension = 0;
29     else
30         for(int i = 0; i < dimension; i++)
31             datos[i] = -1;
32     }
33 }
34
35 TVector::TVector(const TVector &origen) {
36     dimension = origen.dimension;
37     datos = new int[dimension];
38     if(datos == NULL)
39         dimension = 0;
40     else
41         for(int i = 0; i < dimension; i++)
42             datos[i] = origen.datos[i];
43 }
44
45 TVector::~TVector() {
46     dimension = 0;
47     if(datos != NULL)
48     {
49         delete datos;
50         datos = NULL;
51     }
52 }
53
54 int
55 TVector::Dimension(void) {
56     return dimension;
57 }
58
59 TVector
60 TVector::operator+(const TVector &v) {
61     if(dimension != v.dimension)
62     {
63         TVector temp(0);
64
65         return temp;
66     }
67     else
68     {
69         TVector temp(dimension);
70
71         for(int i = 0; i < dimension; i++)
```

```
72     temp.datos[i] = datos[i] + v.datos[i];
73
74     return temp;
75 }
76 }
77
78 TVector
79 TVector::operator-(const TVector &v) {
80     if(dimension != v.dimension)
81     {
82         TVector temp(0);
83
84         return temp;
85     }
86     else
87     {
88         TVector temp(dimension);
89
90         for(int i = 0; i < dimension; i++)
91             temp.datos[i] = datos[i] - v.datos[i];
92
93         return temp;
94     }
95 }
96
97 TVector&
98 TVector::operator=(const TVector &v) {
99     if(this != &v)
100    {
101        if(dimension != v.dimension)
102        {
103            dimension = v.dimension;
104            if(datos != NULL)
105                delete datos;
106            datos = new int[dimension];
107            if(datos == NULL)
108            {
109                dimension = 0;
110                return *this;
111            }
112        }
113        for(int i = 0; i < dimension; i++)
114            datos[i] = v.datos[i];
115    }
116
117    return *this;
```

```
118 }
119
120 int&
121 TVector::operator[](int indice) {
122     if(indice >= 1 && indice <= dimension)
123         return datos[indice - 1];
124
125     error = -1;
126
127     return error;
128 }
129
130 bool
131 TVector::operator==(const TVector &v) {
132     if(dimension != v.dimension)
133         return false;
134     else
135         for(int i = 0; i < dimension; i++)
136             if(datos[i] != v.datos[i])
137                 return false;
138
139     return true;
140 }
141
142 bool
143 TVector::operator!=(const TVector &v) {
144     return !(*this == v);
145 }
146
147 ostream&
148 operator<<(ostream &oo, const TVector &cc)
149 {
150     oo << "[";
151
152     if(cc.dimension > 0)
153         oo << cc.datos[0];
154     for(int i = 1; i < cc.dimension; i++)
155         oo << " " << cc.datos[i];
156     oo << "]";
157
158     return oo;
159 }
```

---

## E.4. La clase TCalendario

Definid en C++ la clase **TCalendario** que contiene una fecha (representada mediante tres variables enteras para el día, mes y año) y un mensaje (representado mediante un vector dinámico de caracteres).

A continuación se incluye el código final de la clase **TCalendario**; la sobrecarga para el operador de salida de la clase sustituye al método **Imprimir()** empleado en algunos ejemplos.

Fichero **tcalendario.h**:

---

Ejemplo E.7

---

```

1 #include <iostream>
2
3 using namespace std;
4
5 class TCalendario {
6     friend ostream& operator<<(ostream &, const TCalendario &);
7     friend bool Bisiesto(const TCalendario &);
8
9 public:
10    TCalendario();
11    TCalendario(int, int, int);
12    TCalendario(int, int, int, char *);
13    TCalendario(const TCalendario &);
14    ~TCalendario();
15
16    TCalendario& operator++(void);
17    TCalendario operator++(int);
18    TCalendario& operator--(void);
19    TCalendario operator--(int);
20    TCalendario& operator=(const TCalendario &);
21    bool operator==(const TCalendario &);
22    bool operator!=(const TCalendario &);
23
24 private:
25    int dia, mes, anyo;
26    char *mensaje;
27 };

```

---

Fichero **tcalendario.cc**:

---

Ejemplo E.8

---

```

1 #include <cstring>
2
3 #include "tcalendario.h"

```

```
4
5 TCalendario::TCalendario() {
6     dia = mes = anyo = 1;
7     mensaje = NULL;
8 };
9
10 TCalendario::TCalendario(int d, int m, int a) {
11     dia = d;
12     mes = m;
13     anyo = a;
14     mensaje = NULL;
15 };
16
17 TCalendario::TCalendario(int d, int m, int a, char *ms) {
18     dia = d;
19     mes = m;
20     anyo = a;
21     mensaje = new char[strlen(ms) + 1];
22     if(mensaje == NULL)
23         return;
24     strcpy(mensaje, ms);
25     // También funciona
26     // strdup(mensaje, ms);
27 };
28
29 TCalendario::TCalendario(const TCalendario &cal) {
30     dia = cal.dia;
31     mes = cal.mes;
32     anyo = cal.anyo;
33     if(cal.mensaje == NULL)
34         mensaje == NULL;
35     else
36         mensaje = new char[strlen(cal.mensaje) + 1];
37     if(mensaje == NULL)
38         return;
39     strcpy(mensaje, cal.mensaje);
40     // También funciona
41     // strdup(mensaje, cal.mensaje);
42 };
43
44 TCalendario::~TCalendario() {
45     dia = mes = anyo = 1;
46     if(mensaje != NULL) {
47         delete mensaje;
48         mensaje = NULL;
49     }
```

```
50 }
51
52 TCalendario&
53 TCalendario::operator++(void) {
54     dia++;
55     if(mes == 2 && dia == 29 && !Bisiesto(*this))
56     {
57         dia = 1;
58         mes = 3;
59     }
60     else if((mes == 1 || mes == 3 || mes == 5 || mes == 7 ||
61             mes == 8 || mes == 10 || mes == 12) && dia == 32)
62     {
63         dia = 1;
64         mes++;
65         if(mes == 13)
66         {
67             mes = 1;
68             anyo++;
69         }
70     }
71     else if(dia == 31)
72     {
73         dia = 1;
74         mes++;
75     }
76
77     return *this;
78 }

79
80 TCalendario
81 TCalendario::operator++(int a) {
82     TCalendario temp(*this);
83
84     dia++;
85     if(mes == 2 && dia == 29 && !Bisiesto(*this))
86     {
87         dia = 1;
88         mes = 3;
89     }
90     else if((mes == 1 || mes == 3 || mes == 5 || mes == 7 ||
91             mes == 8 || mes == 10 || mes == 12) && dia == 32)
92     {
93         dia = 1;
94         mes++;
95         if(mes == 13)
```

```
96     {
97         mes = 1;
98         anyo++;
99     }
100 }
101 else if(dia == 31)
102 {
103     dia = 1;
104     mes++;
105 }
106
107 return temp;
108 }

109
110 TCalendario&
111 TCalendario::operator--(void) {
112     dia--;
113     if(dia == 0)
114     {
115         mes--;
116         if(mes == 0)
117         {
118             mes = 12;
119             anyo--;
120         }
121         if(mes == 2 && !Bisiesto(*this))
122             dia = 28;
123         else if(mes == 2 && Bisiesto(*this))
124             dia = 29;
125         else if(mes == 1 || mes == 3 || mes == 5 || mes == 7 ||
126                 mes == 8 || mes == 10 || mes == 12)
127             dia = 31;
128         else
129             dia = 30;
130     }
131
132     return *this;
133 }
134
135 TCalendario
136 TCalendario::operator--(int a) {
137     TCalendario temp(*this);
138
139     dia--;
140     if(dia == 0)
141     {
```

```
142     mes--;
143     if(mes == 0)
144     {
145         mes = 12;
146         anyo--;
147     }
148     if(mes == 2 && !Bisiesto(*this))
149         dia = 28;
150     else if(mes == 2 && Bisiesto(*this))
151         dia = 29;
152     else if(mes == 1 || mes == 3 || mes == 5 || mes == 7 ||
153             mes == 8 || mes == 10 || mes == 12)
154         dia = 31;
155     else
156         dia = 30;
157 }
158
159     return temp;
160 }
161
162 TCalendario&
163 TCalendario::operator=(const TCalendario &cal) {
164     dia = cal.dia;
165     mes = cal.mes;
166     anyo = cal.anyo;
167     if(mensaje != NULL)
168         delete mensaje;
169     if(cal.mensaje == NULL)
170         mensaje == NULL;
171     else
172         mensaje = new char[strlen(cal.mensaje) + 1];
173     if(mensaje == NULL)
174         return *this;
175     strcpy(mensaje, cal.mensaje);
176     // También funciona
177     // strdup(mensaje, cal.mensaje);
178
179     return *this;
180 }
181
182 bool
183 TCalendario::operator==(const TCalendario &cal) {
184     if(dia == cal.dia && mes == cal.mes && anyo == cal.anyo &&
185         !strcmp(mensaje, cal.mensaje))
186         return true;
187 }
```

```
188     return false;
189 }
190
191 bool
192 TCalendario::operator!=(const TCalendario &cal) {
193     return !(*this == cal);
194 }
195
196 ostream&
197 operator<<(ostream &os, const TCalendario &cal) {
198     os << cal.dia << "/" << cal.mes << "/" << cal.anyo << ":" ;
199     if(cal.mensaje != NULL)
200         os << cal.mensaje;
201
202     return os;
203 }
204
205 bool
206 Bisiesto(const TCalendario &cal) {
207     if(cal.anyo % 4 == 0)
208     {
209         if(cal.anyo % 100 == 0)
210             if(cal.anyo % 400 != 0)
211                 return false;
212
213         return true;
214     }
215
216     return false;
217 }
```

---

# Bibliografía recomendada

A continuación se incluye una serie de libros recomendados con una pequeña descripción. Además, también se incluyen algunos enlaces a páginas de Internet donde se puede encontrar más información sobre el lenguaje C++ o sobre herramientas de programación.

## Lenguaje C++

- Miguel de Pereda, Javier Mateo. *Programación Orientada a Objetos con C++*. Anaya Multimedia, Madrid, 1994. ISBN: 84-7614-681-7  
En menos de 300 páginas, este libro explica los conceptos básicos de la orientación a objetos y cómo se aplican en el lenguaje C++. Su lectura es fácil y amena y posee una gran cantidad de ejemplos. Sin embargo, al ser del año 1994, no refleja los cambios introducidos en las últimas versiones del lenguaje C++: emplea constantes para los valores booleanos verdadero/falso, no emplea los espacios de nombres, etc.
- Bruce Eckel. *Thinking in C++, Volume 1: Introduction to Standard C++*. Prentice Hall, Upper Saddle River, 2 edición, 2000. ISBN: 0139798099 y Bruce Eckel, Chuck Allison. *Thinking in C++, Volume 2: Practical Programming*. Prentice Hall, Upper Saddle River, 2 edición, 2003. ISBN: 0130353132  
Bruce Eckel es un notable autor de libros de programación. En estos dos libros, trata en profundidad todas las características del lenguaje C++. Ambos libros contienen numerosos ejemplos, desde muy sencillos a muy avanzados. Y una ventaja: se pueden descargar de forma gratuita de Internet.
- Walter Savitch. *Resolución de problemas con C++*. Pearson Educación, México, 2 edición, 2000. ISBN: 968-444-416-8  
Este libro está diseñado para utilizarse en un primer curso de programación y ciencias de la computación usando el lenguaje C++. Como no requiere experiencia previa en programación, comienza desde lo más básico (variables, tipos de datos y demás conceptos). El libro posee numerosos ejemplos y ejercicios.
- Bjarne Stroustrup. *El lenguaje de programación C++*. Pearson Educación, Madrid, 2002. ISBN: 84-7829-046-X

Escrito por el creador y diseñador original de C++, presenta el lenguaje al completo. Es un libro de referencia esencial imprescindible cuando se quiere resolver alguna duda.

- Harvey M. Deitel, Paul J. Deitel. *Cómo programar en C++*. Pearson Educación, México, 4 edición, 2003. ISBN: 970-26-0254-8

Presenta los fundamentos de la programación orientada a objetos y su empleo con C++ en 1300 páginas. Su estructura es muy clara, lo que facilita su lectura. Sin embargo, le sobran los capítulos sobre UML y programación web con CGI.

- Harvey M. Deitel, Paul J. Deitel. *Cómo programar en C/C++ y Java*. Pearson Educación, México, 4 edición, 2004. ISBN: 970-26-0531-8

La mitad del libro está dedicado al lenguaje de programación C, mientras que la otra mitad se reparte a partes iguales entre C++ y Java. No profundiza en muchos aspectos de C++, pero si se necesita aprender o emplear los tres lenguajes que incluye este libro, es una buena opción.

**Enlaces en Internet** En las siguientes páginas web se pueden encontrar cursos, ejemplos y documentación sobre C++:

- The C++ Resources Network:  
<http://www.cplusplus.com/>
- Bjarne Stroustrup's homepage:  
<http://www.research.att.com/~bs/>
- Programming Tutorials:  
<http://www.cprogramming.com/tutorial.html>
- C++ Programming Tutorial:  
<http://cplusplus.about.com/od/beginnerctutorial/l/blcplustut.htm>
- C++ Annotations Version 6.2.4:  
<http://www.icce.rug.nl/documents/cplusplus/>
- C/C++ Reference:  
<http://www.cppreference.com>

**Herramientas** A continuación se incluyen las direcciones de Internet de algunos de los compiladores de C++ más empleados. Algunos de ellos están disponibles para diferentes sistemas operativos:

- djgpp:  
<http://www.delorie.com/djgpp/>

- Microsoft Visual C++ Toolkit 2003:  
<http://msdn.microsoft.com/visualc/vctoolkit2003/>
- Borland C++ Builder:  
[http://www.borland.com/downloads/download\\_cbuilder.html](http://www.borland.com/downloads/download_cbuilder.html)
- gcc:  
<http://gcc.gnu.org/>
- Dev-C++:  
<http://www.bloodshed.net/>
- lcc-win32:  
<http://www.cs.virginia.edu/~lcc-win32/>
- Digital Mars C/C++ compiler for Win32:  
<http://www.digitalmars.com/>

# Índice alfabético

- #define, 63
- #elif, 63
- #else, 63
- #endif, 63
- #error, 73
- #if, 63
- #ifdef, 63
- #ifndef, 63
- #include, 20, 26, 57
- #undef, 63
- #warning, 72
- Addrcheck, 223
- administración de memoria dinámica, 64
- amiga, 56
- argumentos predeterminados, 137
- autoasignación, 92
- bcc32, 142
- bzip2, 227
- código fuente, 9
- código máquina, 9
- código objeto, 9
- Cachegrind, 223
- casting, 128
- clase base, 126
- clase derivada, 126
- clases, 8
- compilación condicional, 70
- compilar, 9
- composición, 60, 120, 136
- const, 4, 83, 84
- constantes, 83
- constructor, 34
- constructor de copia, 37, 135
- constructor por defecto, 34, 135
- conversión de tipo, 128
- declaración de clase, 10
- definición de clase, 10
- delete, 64, 65, 68, 70, 137
- delete [], 69
- depuración, 71
- destructor, 43, 136
- DOS, 228
- emacs, 208
- espacios de nombres, 25
- especificadores de acceso a miembros, 13
- forma canónica, 44, 135
- forma ortodoxa, 44, 135
- friend, 56, 59, 89
- función de cero parámetros, 16, 34, 137
- función de lista de parámetros vacía, 137
- función inline, 16, 139
- función miembro, 88
- función no miembro, 88
- g++, 5, 6, 9, 20, 142
- gdb, 4, 6, 215, 218
- get, 15
- guardas de inclusión, 62
- gzip, 227



9 788479 088880

<http://publicaciones.ua.es>



TEXTOSDOCENTES

## C++ PASO A PASO

El libro está estructurado como soporte de un curso de introducción al lenguaje C++. Todas las explicaciones van acompañadas de ejemplos para afianzar los conceptos. Es aconsejable que el lector lea el libro delante del ordenador, para que al realizar y modificar los ejemplos comprenda mejor su funcionamiento. Al final de cada capítulo se proponen ejercicios de autoevaluación y de programación, acompañados todos ellos de sus correspondientes soluciones.

La principal aportación de este libro, frente a otros libros similares, es que en él hemos querido reflejar los problemas a los que se enfrenta un lector cuando aprende un lenguaje de programación nuevo. La mayoría de los textos suponen que el lector no va a cometer errores, por lo que no hacen ninguna referencia a los posibles problemas de compilación del código o de comprensión de los conceptos explicados. Nosotros hemos optado por incluir algunos ejemplos con errores para mostrar los mensajes que genera el compilador.

Sergio Luján Mora es doctor ingeniero en Informática por la Universidad de Alicante. Desde 1999, forma parte del Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Alicante. Ha escrito los libros *Programación en Internet: Clientes Web*, *Programación de servidores web con CGI, SSI, e IDC*, *Programación de aplicaciones web: historia, principios básicos y cliente web* y *Cuestionario básico sobre Programación en Internet*.



PUBLICACIONES  
UNIVERSIDAD DE ALICANTE