C++ paso a paso

Sergio Luján Mora

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

1. Introducción **1**
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

2. Clases y objetos **7**
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

3. Constructor y destructor **31**
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

4. Funciones y clases amigas y reserva de memoria **55**
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

5. Sobrecarga de operadores **81**
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

6. Composición y herencia **119**
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

7. Otros temas **135**
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

8. Errores más comunes **141**
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

9. Ejercicios **185**
9.1. Mentiras arriesgadas . . . 185
9.2. La historia interminable . . . 186
9.3. Pegado a ti . . . 187
9.4. Clase TComplejo . . . 188

A. Palabras clave **191**
A.1. Lista de palabras clave . . . 191

B. Operadores **197**
B.1. Lista de operadores . . . 197

C. Sentencias **201**
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

D. Herramientas **207**
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

E. Código de las clases **229**
E.1. La clase TCoordenada 229
E.2. La clase TLinea 235
E.3. La clase TVector 237
E.4. La clase TCalendario 242

Bibliografía recomendada **249**

Índice alfabético **253**

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 cometen 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 presentado 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[1] de flujo de estrada/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[2] `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[3]:

```
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`.

[1] Para más información sobre archivos de encabezado, consultar la Sección 2.9.
[2] Para más información sobre espacios de nombres, consultar la Sección 2.10.
[3] 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`[4] para crear una clase. La construcción

```
class NombreClase {
// Contenido de la clase
};
```

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:

```
class TCoordenada {
  int x, y, z;
};
```

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

```
#include <iostream>

using namespace std;

class TCoordenada {
  int x, y, z;
};

int
main(void)
{
  int i;
  TCoordenada p1;

  return 0;
}
```

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?

[4] 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:

```
TCoordenada p1;

cout << "Componente x: " << p1.x << endl;
cout << "Componente y: " << p1.y << endl;
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`:

```
TCoordenada p1;
TCoordenada *ptr1 = &p1;

cout << "Componente x: " << ptr1->x << endl;
cout << "Componente y: " << ptr1->y << endl;
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

```
#include <iostream>

using namespace std;

class TCoordenada {
  int x, y, z;
};

int
main(void)
{
  int i;
  TCoordenada p1;

  p1.x = 1;
  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

```
ejem3.cc: In function ‘int main()’:
ejem3.cc:6: ‘int TCoordenada::x’ is private
ejem3.cc:15: within this context
ejem3.cc:6: ‘int TCoordenada::y’ is private
ejem3.cc:16: within this context
ejem3.cc:6: ‘int TCoordenada::z’ is private
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[5] 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:

```
// nSegundos = 920
class THora {
  int nSegundos;
};

// hora = "00:15:20"
```

[5] 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[6] 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
```

[6] Las funciones y clases amigas se explican en el Capítulo 4.

	Descripción
`public:`	Accesible tanto desde la propia clase como desde funciones ajenas a la clase
`private:`	Accesible exclusivamente desde las funciones miembros y funciones y clases amigas
`protected:`	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`[7]. Las funciones `getX()`, `getY()` y `getZ()` se denominan de cero parámetros[8] 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();
```

[7] Para más información sobre las funciones `inline`, consultar la Sección 7.4.

[8] 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[9] 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:

[9] 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[10]:

- `.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 };
```

[10] 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**[11] 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:

```
g++ -c tcoordenada.cc
g++ -c main.cc
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.

[11] 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`[12]. 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.

[12] 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 prerrequisito (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 prerrequisito.
- 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**[13], 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;
```

[13] 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 };
```