

# **Programación C++**

## **Paso a paso**

**Edgar D'Andrea**

---

**El código para realizar las prácticas del libro se puede descargar de <http://www.edgardandrea.com/libros>**

---

# Tabla de contenidos

1. Introducción.....	17
2. Tipos de datos fundamentales.....	47
3. Tipos de datos derivados.....	75
4. Operadores, bucles y bifurcaciones.....	111
5. Funciones.....	139
6. Funciones avanzadas.....	171
7. Objetos y clases.....	203
8. Clases: operadores y conversiones.....	231
9. Clases y funciones implícitas.....	251
10. Herencia de clases.....	275
11. Contención y herencia privada de clases...	311
12. Templates para clases genéricas.....	343
13. Excepciones.....	367
14. Entrada y salida.....	385
15. Biblioteca STL.....	421
A. Respuestas a las prácticas.....	453

# Índice general

## Capítulo 1:

<b>Introducción.....</b>	<b>17</b>
Origen de C++.....	18
Creación de un programa.....	19
Creación de un programa desde la línea de comandos.....	20
Creación de un programa desde un IDE.....	22
Errores de sintaxis.....	25
Primeros pasos en C++.....	25
Comentarios.....	26
Comentarios estilo C.....	27
Sentencias para el preprocesador.....	27
Pasos del preprocesado.....	29
Namespace.....	31
Módulos.....	31
Definición de función main().....	31
Sentencia que produce salida de datos (cout).....	32
Otros ejemplos de cout.....	33
Sentencia que permite la entrada de datos (cin).....	34
Normas generales del formato del código fuente.....	34
Declaración de variables.....	36
Asignación de valor a una variable .....	37
Funciones .....	38
Funciones con o sin valor de retorno.....	38
Prototipo de función.....	39
Bibliotecas de funciones.....	40
Funciones definidas por el usuario.....	40
Palabras clave de C++.....	42
Resumen.....	45

## Capítulo 2:

<b>Tipos de datos fundamentales.....</b>	<b>47</b>
Tipos fundamentales.....	48
Tipos enteros.....	50
¿Qué sucede cuando se define un entero con signo?.....	51

Elección del tipo entero.....	54
Asignación de valores a variables enteras.....	55
Decimal, octal y hexadecimal.....	56
Tipo de las constantes enteras.....	57
Tipo char.....	57
Sentencias de escape y tipo char.....	59
Constantes de tipo entero (literales enteras).....	60
<b>Tipos reales (coma flotante).....</b>	<b>60</b>
Constantes de tipo real (literales reales).....	61
Representación de un número real con precisión simple.....	62
Precisión en tipos reales.....	62
<b>Operaciones aritméticas.....</b>	<b>64</b>
Precedencia de operadores.....	65
Paréntesis y precedencia.....	65
Divisiones y tipos de datos.....	66
<b>Conversiones de tipo.....</b>	<b>67</b>
Conversión en asignación.....	68
Conversiones en expresiones.....	70
Promoción entera.....	71
Promoción por combinación de tipos.....	71
Conversiones manuales.....	72
<b>Tipo bool.....</b>	<b>72</b>
Calificador const.....	73
<b>Resumen.....</b>	<b>73</b>

## **Capítulo 3:**

### **Tipos de datos derivados.....75**

Arrays: matrices de variables.....	76
Matriz de una dimensión.....	77
Matrices multidimensionales.....	77
Definición de matrices.....	78
Definición con inicialización de elementos.....	79
sizeof en matrices.....	81
Resumen de las características de las matrices o arrays .....	81
<b>Cadenas.....</b>	<b>81</b>
sizeof y strlen().....	83
Recortando una cadena.....	84
Entrada de cadenas de texto.....	85
Entrada numérica combinada con entrada de cadenas.....	87
Función gets(): Lectura de entrada por teclado.....	88

Función puts(): Impresión en consola .....	88
Estructuras.....	89
Inicialización de una estructura.....	89
Declaración con inicialización en un paso.....	90
Matrices de estructura.....	90
Uniones.....	91
Punteros .....	92
Operador new: reserva de memoria.....	94
Reserva dinámica de memoria para una matriz.....	96
Liberación de la memoria.....	99
Punteros a cadenas.....	100
Asignación de un puntero a char con un literal.....	102
Punteros y estructuras (Operador ->).....	103
Puntero NULL.....	106
Referencias (&).....	106
Enumeraciones.....	107
Resumen.....	108

## Capítulo 4:

### **Operadores, bucles y bifurcaciones.....111**

Bloques de sentencias.....	112
Expresiones relacionales.....	112
Operadores de bit.....	113
Precedencia de operadores.....	114
Lista de precedencia y asociatividad de los operadores C++.....	114
Expresiones lógicas.....	116
Operador OR lógico (  ).....	117
Operador AND lógico (&&).....	117
Operador NOT lógico (!).....	118
Bucles.....	119
Sentencia for.....	119
Operador coma.....	121
Operadores incremento (++) y decremento (--)......	122
Operadores de asignación compuestos.....	122
Sentencia while.....	123
while y la espera del tiempo transcurrido.....	124
Alias de tipos.....	126
Sentencia do-while.....	127
Sentencias de control de condiciones y de bifurcación.....	128

Sentencia if.....	128
Sentencia if-else.....	130
Sentencia if-else con if anidados.....	131
Operador condicional (?).....	132
Sentencia switch.....	133
Sentencia break.....	134
Sentencia continue.....	135
Resumen.....	136

## **Capítulo 5:**

### **Funciones.....139**

Definición de la función.....	140
Prototipo de una función.....	141
Variables utilizables en una función.....	142
Pase de argumentos a una función.....	144
Pase por valor.....	145
Pase por referencia.....	145
Matrices como argumentos de una función.....	147
Protección de parámetros con const.....	150
const utilizado con punteros.....	150
Cadenas como argumentos de una función.....	152
Cadenas como valor de retorno.....	155
Estructuras como argumentos de una función .....	155
Valor de retorno y referencia a tipo.....	157
Métodos que no tienen valor de retorno.....	158
Argumentos predeterminados.....	159
Argumentos desde la línea de comandos.....	160
Funciones sobrecargadas (polimorfismo).....	161
Resolución de ambigüedades en la sobrecarga de funciones .....	162
Recursividad.....	163
Funciones inline.....	164
Funciones inline o macros.....	166
Resumen.....	166

## **Capítulo 6:**

### **Funciones avanzadas.....171**

Funciones genéricas (templates).....	172
Funciones genéricas con más un de tipo parámetro.....	174

Funciones genéricas sobrecargadas.....	174
Funciones especializadas.....	175
Instancias de función genérica.....	176
Instancia explícita.....	176
Especialización de funciones genéricas.....	177
Elección de la función adecuada.....	179
Distribución del código.....	181
Clases de almacenamiento .....	184
Ámbito de las variables .....	184
Variables automáticas.....	185
Almacenamiento de las variables automáticas.....	186
Variables registro.....	187
Variables estáticas.....	188
Otros calificadores de clase de almacenamiento.....	192
Almacenamiento de las funciones.....	193
Secuencia de búsqueda de una función.....	193
Almacenamiento dinámico.....	193
Namespaces.....	194
Declaración using y directiva using namespace.....	198
Espacios de nombres anónimos.....	199
Resumen.....	199

## Capítulo 7:

### **Objetos y clases.....203**

Las estructuras, vistas como clases incompletas.....	204
El concepto de clase.....	205
Encapsulamiento.....	206
Miembros públicos y privados.....	207
Implementación de los métodos miembro.....	207
Métodos inline.....	208
Objetos: instancias de una clase.....	208
Uso de la clase.....	209
Constructores y destructores de objetos.....	210
Usos del constructor.....	212
Constructor predeterminado.....	212
Uso del destructor.....	213
Revisión del uso de clases.....	215
Código de declaración de la clase Empleado, archivo cempleado.h .....	216
Código de implementación de los métodos de la clase Empleado, archivo cempleado.cpp .....	217



Uso de la clase, archivo cap0701.cpp.....	219
Ejecución del programa.....	221
Puntero a objetos.....	222
Puntero this.....	222
Matrices de objetos.....	224
Ámbito de una clase.....	225
Clases y miembros constantes.....	226
Caso función miembro const.....	227
Caso del puntero constante.....	227
Resumen.....	229

## **Capítulo 8:**

### **Clases: operadores y conversiones.....231**

Sobrecarga de operadores.....	231
Sobrecarga en la práctica.....	232
Limitaciones de la sobrecarga de operadores.....	238
Ampliación de la sobrecarga de operadores.....	239
Funciones friend.....	240
Conversiones en las clases.....	242
Funciones de conversión.....	245
Conversión implícita con el objeto cout.....	247
Más conversiones funcionales.....	247
Resumen.....	249

## **Capítulo 9:**

### **Clases y funciones implícitas.....251**

Clase Empleado, versión inicial.....	252
Presentación de los 5 casos de estudio.....	254
Caso1 con la clase Empleado.....	257
Caso 2 con la clase Empleado.....	258
Caso 3 con la clase Empleado.....	259
Caso 4 con la clase Empleado.....	260
Caso 5 con la clase Empleado.....	260
Empleado1: revisión de la clase Empleado.....	261
Caso1 con la clase Empleado1.....	265
Caso 2 con la clase Empleado1.....	266
Caso 3 con la clase Empleado1.....	266
Caso 4 con la clase Empleado1.....	268

Caso 5 con la clase Empleado1.....	268
Empleado2: revisión de la clase Empleado1.....	269
Caso 3 con la clase Empleado2.....	271
Caso 4 con la clase Empleado2.....	272
Caso 5 con la clase Empleado2.....	273
Resumen.....	273

## **Capítulo 10:**

<b>Herencia de clases.....</b>	<b>275</b>
Creación de una clase base simple.....	277
Relación "es un" .....	281
Acceso a los miembros según el modo de herencia.....	282
Clase derivada.....	283
Lo que no se hereda de la clase base.....	284
Implementación de la clase derivada.....	286
Constructor.....	286
Constructor copia de objetos Gerente .....	287
Constructor conversión Empleado a Gerente.....	289
Otras funciones miembro de la clase derivada.....	290
Datos protected para el método modificarSalario() .....	291
Operador = en clases derivadas.....	293
Prueba de la clase derivada.....	294
Caso 1: creación de un objeto Empleado.....	296
Caso 2: creación de un objeto Gerente .....	297
Caso 3: conversión de un objeto Empleado en objeto Gerente.....	297
Caso 4: constructor copia de objetos Gerente.....	298
Caso 5: modificación del salario .....	299
Caso 6: asignación de un objeto Gerente a otro objeto Gerente (operator=) .....	299
Caso 7: conversión desde objeto Empleado.....	299
Herencia, referencias y punteros.....	299
Polimorfismo .....	300
Funciones miembro virtuales .....	301
Función no virtual o función virtual.....	303
Cambio a virtual.....	304
Métodos y el calificador virtual.....	306
Sobrecarga y métodos virtuales.....	306
Clases abstractas.....	307
Resumen sobre funciones virtuales.....	308
Razones por las que una función virtual deja de serlo.....	308
Resumen.....	308

**Capítulo 11:****Contención y herencia privada de clases.....311**

Clases que contienen clases.....	312
Clase Texto.....	313
Clase VectorSalario.....	316
Nueva clase Empleado como contenedora.....	321
Interfaz pública del objeto contenedor .....	322
Utilización de la clase contenedora Empleado.....	324
Resultado de la ejecución (proyecto cap11).....	326
Herencia privada.....	328
Nueva clase Empleado con herencia privada.....	329
Implementación de la clase Empleado2 (herencia privada).....	331
Prueba de la clase Empleado2.....	332
Herencia protected.....	334
Modificación de acceso privado con la declaración using.....	334
Clases friend.....	336
Clases anidadas.....	337
Ámbito de la clase anidada.....	337
Resumen.....	338

**Capítulo 12:****Templates para clases genéricas.....343**

Typedef comparado con templates.....	343
Clases genéricas o templates.....	346
Uso de la clase genérica.....	349
Templates con otros objetos.....	350
Templates como clases .....	352
Instancias implícitas.....	353
Instancias explícitas.....	353
Especializaciones .....	353
Especializaciones con más de un argumento.....	354
Especializaciones parciales.....	354
Templates y clases friend.....	354
RTTI.....	356
Operador dynamic_cast.....	357
dynamic_cast con referencias.....	361
Operador typeid.....	361

Otros operadores cast.....	362
const_cast.....	362
static_cast.....	363
reinterpret_cast.....	363
Clase auto_ptr y memoria dinámica.....	363
Resumen.....	365

## Capítulo 13:

### **Excepciones.....367**

Errores que pueden aparecer en un programa C++.....	368
Gestión básica de los errores.....	369
Gestión de excepciones.....	370
throw.....	371
catch.....	372
try.....	373
Continuación tras una excepción.....	374
Relanzado de una excepción.....	375
Excepciones previstas.....	375
Múltiples bloques try.....	376
Revisión de la secuencia de procesamiento y excepciones.....	376
Excepciones y clases.....	379
Excepciones y herencias.....	379
Excepciones no esperadas y excepciones no detectadas.....	380
Resumen.....	382

## Capítulo 14:

### **Entrada y salida.....385**

Flujos de datos (stream).....	386
Tipos de flujos de datos.....	387
Flujos de datos de C++.....	388
La clase ios y otras clases derivadas.....	389
Sobrecarga de operadores en la entrada/salida.....	390
Inserción de objetos en el flujo de datos.....	390
Otros métodos de inserción: put() y write().....	392
Descarga del búfer de salida.....	393
Sobrecarga de operadores de extracción.....	394
Formateo de entrada/salida.....	396
Asignación del ancho y la precisión de un campo numérico.....	398
Manipuladores de entrada/salida.....	399

Creación de manipuladores personalizados.....	401
Entrada y salida con archivos .....	402
Pasos para la entrada/salida con archivos.....	402
Comprobación con la función <code>is_open()</code> .....	403
Lectura y escritura de archivos de texto.....	404
Entrada/salida con acceso binario.....	406
Proceso byte a byte: <code>get()-put()</code> .....	407
Proceso en bloques de bytes: <code>read()-write()</code> .....	409
Otras formas de <code>get()</code> .....	411
<code>getline()</code> .....	412
<code>flush()</code> .....	413
Resumen de funciones auxiliares.....	413
Acceso directo.....	414
Verificación del estado de entrada/salida.....	416
Método <code>exceptions()</code> .....	417
Resumen.....	418

## **Capítulo 15:**

### **Biblioteca STL.....421**

Contenedores de la biblioteca STL.....	422
Concepto de contenedor.....	423
Vector.....	423
Clase <code>string</code> como contenedor STL.....	424
Lista enlazada.....	424
Deque.....	425
Colas ( <code>Queue</code> ).....	425
Cola con prioridades.....	425
Pila ( <code>Stack</code> ).....	425
Conjuntos ( <code>set</code> y <code>multiset</code> ).....	426
Mapas.....	426
Clase <code>pair</code> .....	426
Algoritmos.....	426
Algoritmos no modificadores.....	427
Algoritmos aritméticos.....	428
Algoritmos buscadores.....	428
Algoritmos de comparación .....	429
Algoritmos operacionales .....	429
Algoritmos modificadores.....	429
Algoritmos clasificadores.....	430
Algoritmos de asignación.....	431
Algoritmos de utilidad.....	432

Iteradores.....	432
Tipos de iteradores.....	433
Operadores soportados por tipo de iterador.....	434
cout/cin con ostream_iterator e istream_iterator.....	435
Otros iteradores predefinidos.....	437
Uso práctico de los contenedores, algoritmos e iteradores.....	438
Ejemplo con vector.....	438
Ejemplo con list .....	440
Ejemplo con multiset.....	442
Ejemplo con multimap.....	444
Objetos función (functor).....	445
Ejemplo de ordenación con un objeto función.....	446
Ejemplo de eliminación de elementos con objeto función.....	447
Objetos función predefinidos.....	448
Extensibilidad de la biblioteca STL.....	449
Resumen.....	450

## Apéndice A:

<b>Respuestas a las prácticas.....</b>	<b>453</b>
--	------------

<b>Índice.....</b>	<b>471</b>
--------------------	------------

# Capítulo 1: Introducción

## Contenido:

- Creación y compilación de un programa
- Uso de un IDE
- Estructura de un programa
- Comentarios
- Directivas para el preprocesador
- Declaración y asignación de variables
- cout/cin
- Funciones

C++ es un lenguaje que suma la capacidad de la programación orientada a objetos a las ya conocidas virtudes del lenguaje C: rapidez, eficiencia y portabilidad.

En este capítulo de introducción haremos un poco de historia y veremos algunas de los fundamentos del lenguaje para crear programas en C++. Una de las razones que justifica el uso de C++ es la posibilidad de desarrollar programas orientados a objetos y si el lector ya conoce el lenguaje C ya tendrá más de medio camino ganado ya que C++ utiliza mucho del lenguaje C. Pero no todas son ventajas para quien conozca el lenguaje C, algunas cosas cambian en C++ y el programador de C tendrá que quitarse de encima algunos hábitos no aceptados por C++. Para el que no conozca C todo resultará nuevo: la codificación C, la programación orientada a objetos (salvo que la haya utilizado en otros lenguajes) y lo nuevo de la codificación C++. Pero para eso se escribe este libro, para que el lector domine C++ desde principio a fin.

Es cierto que C++ no es un lenguaje fácil. O, para ser más preciso, lo que no es fácil es saber sacarle todo el provecho que ofrece su enorme potencialidad. Ya veremos que hacer un programa sencillo puede ser tan fácil como hacerlo con Visual Basic. El aprendizaje completo del lenguaje C++ requiere tiempo, esfuerzo y práctica. La recompensa para el lector será que al dominar C++ sabrá programar en el lenguaje más potente de hoy en día: el lenguaje preferido para codificar las rutinas más complejas, el lenguaje de elección para desarrollar juegos para múltiples plataformas, el lenguaje con el que se desarrollan los sistemas operativos, en definitiva "el lenguaje de programación por excelencia".

En este libro no se dará por supuesto que el lector conozca C. En cada tema veremos las cosas que C++ comparte C para facilitar el aprendizaje para los que ya conozcan C y puedan relacionar rápidamente ambos lenguajes. Al añadir el componente propio de C++ el lector tendrá la visión completa de cada tema.

## Origen de C++

Tal como había ocurrido unos 10 años antes con el lenguaje C, C++ fue creado en Bell Laboratories a principios de la década del 80. En este caso el diseñador e implementador fue Bjarne Stroustrup y la principal razón de su creación fue desarrollar una alternativa al lenguaje assembler y al lenguaje C. Ya hemos dicho que C++ toma muchos elementos de C, a tal punto que al principio se denominaba "C con clases". Pero también Stroustrup declara que Simula67 fue otras de sus fuentes de inspiración, del que aprovecha el concepto de clases y herencia.

### ¿C++, Java o C#?

La respuesta a esta pregunta es: **se debe elegir el lenguaje más adecuado a nuestras necesidades. Ninguno es el mejor en todo.**

Java y C# se pueden considerar dos lenguajes hijos de C++, aunque difieran en las funciones disponibles la sintaxis es prácticamente la misma. El que conozca cualquiera de estos tres lenguajes podrá dominar rápidamente los otros ya que además de la sintaxis comparten el mismo enfoque para el modelo de objetos.

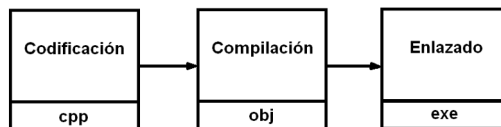
La principal diferencia entre Java, C# y C++ es el enfoque de ejecución. C++ está diseñado para ser ejecutado en un tipo de sistema y procesador específico; el código ejecutable que genera el compilador es específico para un determinado sistema. Si queremos migrar a otro sistema, habrá que recompilar la aplicación. En cambio, Java y C# se compilan generando un pseudo código denominado lenguaje intermedio; en el caso de Java, este pseudo código se ejecuta en una máquina virtual de Java y en el caso de C# el pseudo código se ejecuta en un tiempo de ejecución CLR. En Java y en C# simplemente se precisa que el equipo en donde se ejecuta la aplicación tenga instalado el entorno requerido (máquina virtual Java o el CLR). Esto hace que estos lenguajes sean más fáciles de transportar, en contrapartida, debido a esta compilación en tiempo de ejecución las ejecuciones de estos pseudo códigos Java y C# tienen un menor rendimiento que los ejecutables C++.



Java fue diseñado pensando en Internet para resolver la necesidad de un gran nivel de portabilidad entre diferentes sistemas; C++ fue diseñado para crear componentes de gran rendimiento y sistemas operativos; C# intenta ocupar un punto intermedio. Como siempre sucede, se trata de buscar el que mejor resuelva nuestras necesidades.

## Creación de un programa

Los pasos de creación de un programa se pueden esquematizar de la siguiente manera:



C++ es un lenguaje que se puede codificar en múltiples plataformas y además, en cada plataforma, se dispone de una amplia variedad de compiladores, por tal motivo resulta difícil precisar exactamente los pasos que se siguen para ejecutar un programa escrito en C++ ya que éstos dependerán del entorno del equipo y del compilador utilizado pero en términos generales son los siguientes:

- 1. Codificación del programa:** Se crea el código fuente y se guarda en un archivo. Para esto simplemente se puede utilizar el editor de textos más sencillo (por ejemplo, Bloc de notas) aunque normalmente lo haremos con algún producto más avanzado y elaborado, por ejemplo, con el IDE (*Integrated Development Environment*) de Visual C++, Borland C++, OpenWatcom C/C++, etc. El uso de un IDE tiene la ventaja que gestiona todos los pasos de creación de un programa. En todo caso, el resultado de la codificación es un archivo de texto, normalmente con la extensión `cpp`.
- 2. Compilación del código fuente:** El compilador es un programa que traduce el código fuente al lenguaje de máquina (depende del sistema operativo) y que genera un archivo de salida denominado código objeto del programa.
- 3. Enlazado del código objeto:** Todos los programas suelen utilizar código externo que está almacenado en bibliotecas. Lo habitual es aprovechar rutinas ya escritas por nosotros o por terceros (lo normal es que el compilador incluya un conjunto de bibliotecas de las funciones más comunes: funciones

aritméticas, de entrada salida, gráficas, etc.). Este paso de enlazado combina nuestro código objeto con el código objeto de las funciones utilizadas desde nuestro programa. El resultado de este paso es un archivo ejecutable.

---

*Aunque en este libro utilizaremos principalmente el compilador C++ que se incluye en Microsoft Visual C++ 2008 Express Edition (gratuito, disponible para todo el mundo), el código de los programas se puede considerar genérico y utilizable por todos los compiladores C++ modernos.*

---

## Creación de un programa desde la línea de comandos

El proceso de edición, compilación y ejecución se puede realizar desde la línea de comandos (Símbolo del sistema). En este ejemplo utilizamos el compilador C++ de Visual C++ 2008 pero el procedimiento es el mismo prácticamente en todos los compiladores C++, lo que cambia es el nombre del editor (notepad) y del compilador (cl).

Una compilación de un programa recién escrito es casi seguro que presentará algún tipo de error, el simple olvido de un punto y coma hará que el programa no pase a la fase de creación del archivo ejecutable. Por lo tanto, tendremos que habituarnos a la tarea de eliminar errores y volver a intentar la compilación hasta que quede libre de errores.

La ejecución de un programa también puede presentar errores, pero estos ya son de otro tipo: nuestro programa no hace lo que se espera, hay un error inesperado porque falta un archivo, porque un dato no tiene el formato esperado, porque nos hemos quedado sin memoria, porque el programa entra en un bucle infinito, etc. Las causas del mal funcionamiento de una ejecución son innumerables pero ya no son fallos de sintaxis (que detectaría el compilador) sino generalmente más graves (error de lógica o de tratamiento de los datos).

---

*Para forzar la terminación de la ejecución de un programa se pulsa Ctrl+C.*

---

Ahora veremos los pasos básicos para la creación de un programa C++ desde la consola Símbolo del sistema, en la sección siguiente veremos que estos mismos pasos se pueden realizar desde un IDE.

1. Se selecciona Inicio/Todos los programas/Microsoft Visual Studio 2008/Visual Studio Tools/Símbolo del sistema de Visual Studio 2008. El Símbolo del sistema de Visual Studio 2008 determina automáticamente la ruta de acceso al compilador de Visual C++ y del resto de las bibliotecas que utiliza.

2. En el símbolo del sistema, introduciremos `notepad micpp.cpp` y luego pulsamos la tecla **Entrar**. Como este archivo no existe deberemos responder **Sí** cuando se nos pida si se crea un archivo nuevo.
3. Este comando hará que se abra el **Bloc de notas** y ahí escribiremos estas líneas de código, que por ahora no analizaremos:

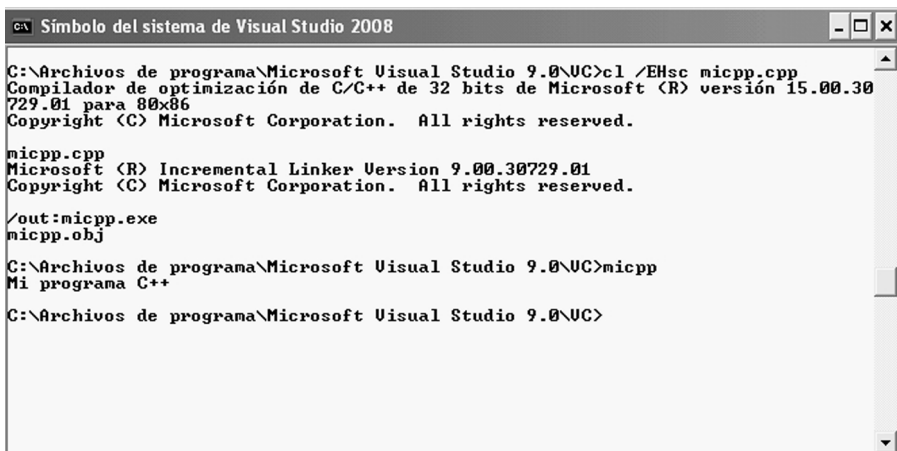
```
#include <iostream>

int main()
{
    std::cout << "Mi programa C++." << std::endl;
    return 0;
}
```

4. En el menú **Archivo**, se hace clic en **Guardar**. Ya hemos creado nuestro archivo de código fuente de Visual C++.
5. En el menú **Archivo**, se hace clic en **Salir** para cerrar el editor.
6. En el símbolo del sistema, se introduce este comando (`cl` es el compilador y `/EHsc` son opciones de compilación) y después se pulsa **Entrar**:

```
cl /EHsc micpp.cpp
```

7. El compilador de C++ `cl.exe` genera el programa ejecutable `micpp.exe`. Tal como se observa en la información de salida de la ejecución del compilador.
8. Para visualizar la lista de todos los archivos del directorio denominados `micpp` con cualquier extensión, se introduce `dir micpp.*` y se pulsa **Entrar**. Veremos que además de generar el archivo ejecutable (`.exe`) también se creó un archivo intermedio de la compilación de extensión `.obj`.
9. Para ejecutar el programa `micpp.exe`, se introduce `micpp` y se pulsa **Entrar**. El programa mostrará este resultado:



```
C:\Archivos de programa\Microsoft Visual Studio 9.0\VC>cl /EHsc micpp.cpp
Compilador de optimización de C/C++ de 32 bits de Microsoft (R) versión 15.00.30729.01 para 80x86
Copyright (C) Microsoft Corporation. All rights reserved.

micpp.cpp
Microsoft (R) Incremental Linker Version 9.00.30729.01
Copyright (C) Microsoft Corporation. All rights reserved.

/out:micpp.exe
micpp.obj

C:\Archivos de programa\Microsoft Visual Studio 9.0\VC>micpp
Mi programa C++

C:\Archivos de programa\Microsoft Visual Studio 9.0\VC>
```

**Vistazo parcial del libro**  
**Salto de páginas...**

## Capítulo 2: Tipos de datos fundamentales

### Contenido:

- **Tipos enteros**
- **Tipos reales**
- **Constantes**
- **Operaciones aritméticas**
- **Conversiones automáticas**
- **Cast**

En el capítulo anterior hemos visto cómo se definen las variables, las reglas de su nomenclatura y cómo se pueden asignar valores. El primer elemento que define a una variable es su tipo, es decir, el tipo de dato que almacena. Para simplificar y abreviar el ejemplo del capítulo de introducción trabajamos siempre con el tipo `int` (entero) y no se explicó demasiado acerca de las variantes posibles en C++. Pero el tipo dato es un concepto muy importante de la programación que merece un estudio completo y detallado.

Los tipos de datos que incluye C++ se pueden clasificar en dos categorías: tipos de datos fundamentales y tipos de datos derivados. C++ nos permite crear tipos de datos por lo que es importante hacer esa diferenciación. Los tipos fundamentales representan básicamente a los números enteros y de coma flotante, mientras que los tipos derivados se denominan así ya que se derivan de los tipos fundamentales, entre estos veremos las cadenas, las matrices, las estructuras y los punteros. En este capítulo analizaremos los tipos fundamentales y el capítulo siguiente estará dedicado a los tipos derivados.

## Tipos fundamentales

Básicamente, los tipos fundamentales representan a los números enteros y a los números reales (coma flotante). Los tipos enteros representan números sin parte fraccionaria y los tipos reales permiten almacenar números con decimales. En C++ hay diversos tipos de enteros que se diferencian principalmente por su capacidad de almacenamiento y, por consecuencia, por el espacio de memoria ocupado. Otro factor que determina el tipo es su posibilidad de almacenar el signo del número. El almacenamiento del signo hace que el tipo distribuya su rango de valores entre negativos y positivos, por lo que tiene un valor máximo almacenable menor que si no utilizase signo. La cantidad de valores posibles de un tipo es la misma tenga signo o no lo tenga, lo que varía es el rango de valores.

En la siguiente tabla se detalla un resumen de los tipos fundamentales.

Tipo	Tamaño	Característica
<b>char</b>	1 byte	<p>Normalmente un ASCII. Se describe entre apóstrofes. Se considera como tipo entero, lo que implica que se pueden hacer operaciones lógicas o aritméticas. Puede ser con signo (-128 a 127) o sin signo (0 a 255), signed o unsigned, respectivamente. Ejemplos:</p> <pre>char a; // variable a sin asignación char b = "b";</pre>
<b>wchar_t</b>	2 bytes	<p>Similar a char con mayor capacidad. Siempre con signo: capacidad de 0 a 65.535. Ejemplos:</p> <pre>wchar_t = L'U';</pre>
<b>short</b>	2 bytes	<p>Es un tipo entero. Puede ser con signo o sin signo (signed o unsigned). Por defecto se asume con signo, rango de valores -32.768 y 32.767. Si es sin signo el rango de valores de la variable es 0 a 65.535. También se puede especificar short int (es un sinónimo de short). Ejemplos:</p> <pre>short n; // n ocupa 2 bytes y tiene signo short m = 3000; // definición y asignación // en una sentencia</pre>
<b>int</b>	4 bytes	<p>Es un tipo entero. Puede ser con signo o sin signo (signed o unsigned). Por defecto se asume con signo. Es = o &gt; a short e igual o menor a long. Rango de valores sin signo desde 0 y 4.294.967.295. Rango de valores con signo desde 2.147.483.648 hasta</p>

Tipo	Tamaño	Característica (continuación)
		<p>2.147.483.647.</p> <p>Ejemplos:</p> <pre>int      n; // n ocupa 4 bytes y tiene signo int      m = 3000; // definición y asignación                 // en una sentencia unsigned int p; // variable p entero sin signo</pre>
<b>_intn</b>	var.	<p>Es un tipo entero. Puede ser con signo o sin signo (signed o unsigned). Por defecto se asume con signo. n indica la longitud en bits de la variable. Puede ser 8, 16, 32 y 64 bits.</p> <p>Ejemplos:</p> <pre>_int16 n; // n ocupa 2 bytes y tiene signo _int64 m; // variable de 64 bits y tiene                 // signo</pre>
<b>long</b>	4 bytes	<p>Es un tipo entero. Puede ser con signo o sin signo (signed o unsigned). Por defecto se asume con signo. También se puede especificar long int (es un sinónimo de long).</p> <p>Rango de valores sin signo desde 0 y 4.294.967.295.</p> <p>Rango de valores con signo desde 2.147.483.648 hasta 2.147.483.647.</p> <p>Ejemplos:</p> <pre>unsigned long int n; // n ocupa 4 bytes y no                 // tiene signo signed long m; // long m sería una definición                 // idéntica long      n = 3000L; // el sufijo L define con                 // seguridad el tipo unsigned long p; long      w = 30UL; // el sufijo U indica                 // sin signo</pre>
<b>float</b>	4 bytes	<p>Es de tipo flotante. Es el de menor precisión dentro de los tipos flotantes. Rango de valores desde <math>1,38E-38</math> a <math>3,4E+38</math>.</p> <p>Ejemplo:</p> <pre>float      q; float      x = 1E+2f; // sufijo f o F indica                 // el tipo float</pre>
<b>double</b>	8 bytes	<p>Es de tipo flotante. Es de precisión mayor que el tipo float. Rango de valores desde <math>2,2E-308</math> a <math>1,8E+308</math>.</p>

Tipo	Tamaño	Característica (continuación)
<b>long double</b>	8 bytes	<p><i>Ejemplo:</i></p> <p><b>double r;</b></p> <p><i>Es de tipo flotante. Es de precisión mayor que el tipo double. Rango de valores desde 2,2E-308 a 1,8E+308.</i></p> <p><i>Ejemplo:</i></p> <p><b>long double r;</b></p>
<b>bool</b>	N/A	<p><i>True o False. False es 0, True diferente a 0.</i></p> <p><i>Ejemplo:</i></p> <p><b>bool r;</b></p>
<b>void</b>	-	<p><i>Tipo nulo. No se pueden definir variables de tipo void, pero es un tipo de dato que tiene diversos usos especiales dentro un programa C++. Permite especificar que una función no devuelve valores. Permite definir el tipo base de punteros a objetos de tipo desconocido.</i></p> <p><i>Ejemplo:</i></p> <pre>void suma();    // la función suma no                 // devuelve ningún valor void* xx;       // puntero a un objeto de                 // tipo desconocido void var2;      // error: no hay variables de                 // tipo void</pre>

## Tipos enteros

Los tipos enteros permiten almacenar números sin parte decimal, es decir, 2, 5, -8, etc. (pero no 5,5 ni 3,14). C++ ofrece varios tipos enteros que difieren entre sí en la capacidad de almacenamiento (el rango de valores que puede albergar). Tal como se puede observar en la tabla de tipos, existen cuatro tipos de datos enteros: **char**, **short**, **int** y **long**. Cada uno de ellos puede ser con signo o sin signo.

Debido a que C++ se utiliza en distintas máquinas y sistemas no siempre un tipo de dato tiene una longitud determinada; por ese motivo de lo que podemos estar seguros es que un tipo de dato tiene **al menos** una determinada longitud. Lo siguiente es más exacto que lo expresado en la tabla de tipos:



- Un tipo short tiene al menos 2 bytes
- Un tipo int tiene al menos el tamaño de un tipo short
- Un tipo long tiene al menos 4 bytes y al menos es igual al tipo int

Por lo tanto, **short** y **long** tienen al menos 2 y 4 bytes respectivamente, tal como se indica en la tabla. El problema está en el tipo **int** que aunque lo más habitual es el tamaño de 4 bytes (tal como se lo encuentra en el entorno Windows y Mac), también tenemos versiones de 2 o 3 bytes en otras implementaciones de C++.

C++ nos da una herramienta para gestionar esta diversidad, teniendo en cuenta que C++ es un lenguaje pensado para ser portable entre distintos sistemas. El operador **sizeof()** nos informa la cantidad de bytes de una variable, por lo que podemos averiguar su capacidad.

Una variable **int** en un entorno que considere al tipo **int** como una variable de 2 bytes podrá almacenar un valor sin signo entre 0 y 65.535 ( $2^{16} - 1$ ). Mientras que la misma variable **int** definida en un entorno que la trata como una variable de 4 bytes podrá almacenar un valor entre 0 y 4.294.967.295 ( $2^{32} - 1$ ), que es la capacidad mínima que también nos asegura el tipo **long**.

---

*Los datos de tipo entero se utilizan para representar números enteros. La cardinalidad del tipo es el número de datos exactamente representables dentro de cada tipo. El desbordamiento u overflow es el resultado de una operación aritmética en la que se excede la cardinalidad del tipo*

---

## ¿Qué sucede cuando se define un entero con signo?

El entero de 4 bytes, tal es el caso del tipo **int** en Windows de 32 bits, puede almacenar  $2^{32}$  valores diferentes. Si la variable tiene signo entonces el rango de valores va desde -2.147.483.648 hasta -2.147.483.647 y si la variable no tiene signo el rango de valores va desde 0 hasta 4.294.967.295. La cantidad de valores diferentes es la misma en los dos casos.

Lo mismo sucede con los otros tipos enteros (**char**, **short** y **long**), adaptando el rango de valores a la capacidad de cada tipo (normalmente, 1, 2 y 4 bytes respectivamente).

Veamos un programa que nos ayuda a averiguar la capacidad de un tipo de dato:

```
// cap0201.cpp
#include <iostream>
void ej0201();

int main()
{
    // llamada a las funciones de los ejercicios del capítulo 2
    ej0201();
}
```

```
}

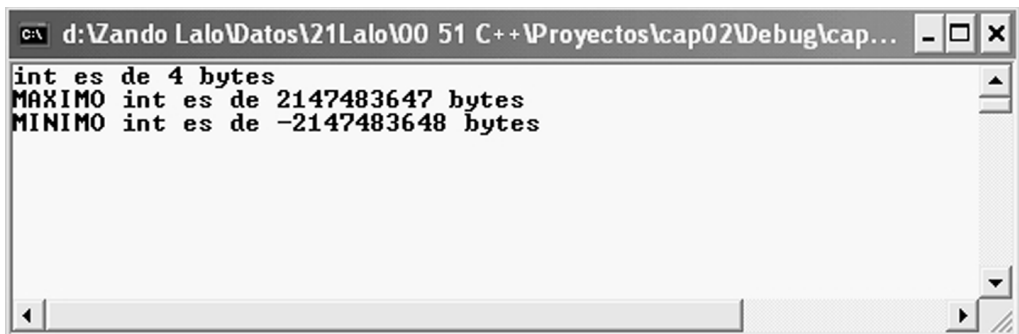
void ej0201()
{
    // Averiguo el tamaño del tipo int en mi equipo
    // para no utilizar std:: habría que haber
    // utilizado using namespace std;

    std::cout << "int es de " << sizeof(int) << " bytes\n";

    // Valor máximo que puede almacenar
    std::cout << "MÁXIMO int es de " << INT_MAX << " bytes\n";

    // valor mínimo que puede almacenar
    std::cout << "MÍNIMO int es de " << INT_MIN << " bytes\n";
}
```

Este programa produce esta salida:



The screenshot shows a Windows command prompt window with the title bar "d:\Zando Lalo\Datos\21Lalo\00 51 C++\Proyectos\cap02\Debug\cap...". The window contains the following output:

```
int es de 4 bytes
MAXIMO int es de 2147483647 bytes
MINIMO int es de -2147483648 bytes
```

El operador `sizeof()` nos permite averiguar la cantidad de bytes que ocupa una variable. Podemos utilizar dos formatos diferentes:

```
sizeof(tipo-dato);
sizeof nombre-variable;
```

En el ejemplo utilizamos el nombre de un tipo, por eso se codificó entre paréntesis. En todo caso, el operador `sizeof` devuelve un valor de tipo entero (`size_t`, definido en el archivo `STDDEF.H`).

Las constantes `INT_MAX` e `INT_MIN` forman parte de un grupo de constantes que nos permiten conocer los rangos de valores de los tipos. Los valores que muestran estas constantes dependen de la implementación y su definición la encontraremos en el archivo `LIMITS.H`.

Éste es un fragmento del código de definición de las constantes en el archivo `limits.h`:

```

/****
*limits.h - implementation dependent values
*
*      Copyright (c) Microsoft Corporation.  All rights reserved.
*
...
#define SHRT_MIN      (-32768)      /* minimum (signed) short value */
#define SHRT_MAX      32767        /* maximum (signed) short value */
#define USHRT_MAX     0xffff       /* maximum unsigned short value */
#define INT_MIN       (-2147483647 - 1) /* minimum (signed) int value */
#define INT_MAX       2147483647 /* maximum (signed) int value */
...

#endif /* _INC_LIMITS */
...

```

**#define** es otra instrucción para el preprocesador que define una constante. El preprocesador reemplazará en el programa los símbolos por el valor relacionado. Por ejemplo, si en nuestro archivo `cpp` utilizamos la constante `SHRT_MAX` cada aparición de ese nombre a lo largo del código ejecutable de nuestro programa será reemplazado por el valor `32767`. No reemplazará ese texto dentro de comentarios o cuando ese texto aparece como parte de otro nombre más largo, por ejemplo, `MISHRT_MAX`. La compilación del código se ejecuta después de la tarea de reemplazo que realiza el preprocesador: el compilador trabaja sobre el código fuente resultante del preprocesamiento.

---

*Más adelante en este capítulo veremos otro modo de definir constantes sin necesidad de recurrir a sentencias del preprocesador. La macro del procesador no ocupa memoria y la variable siempre.*

---

En la siguiente tabla se detallan las constantes disponibles para los tipos enteros:

Constante	Valor	Descripción
<i>SCHAR_MAX</i>	127	Máximo valor para char con signo
<i>SCHAR_MIN</i>	-128	Mínimo valor para char con signo
<i>CHAR_MAX</i>	255 (*1)	Máximo valor para char sin signo
<i>CHAR_MIN</i>	0 (*2)	Mínimo valor para char sin signo
<i>UCHAR_MAX</i>	255	Máximo valor para char sin signo
<i>CHAR_BIT</i>	8	Cantidad de bits en un char
<i>MB_LEN_MAX</i>	5	Cantidad máximo de bytes en un char de bytes múltiples
<i>USHRT_MAX</i>	65535	Máximo valor en short sin signo
<i>SHRT_MAX</i>	32767	Máximo valor en short con signo
<i>SHRT_MIN</i>	-32768	Mínimo valor en short con signo
<i>UINT_MAX</i>	4294967295	Máximo valor en int sin signo
<i>INT_MAX</i>	2147483647	Máximo valor en int con signo
<i>INT_MIN</i>	-2147483648	Mínimo valor en int con signo
<i>ULONG_MAX</i>	4294967295	Máximo valor en long sin signo
<i>LONG_MAX</i>	2147483647	Máximo valor en long con signo
<i>LONG_MIN</i>	-2147483648	Mínimo valor en long con signo

Notas: Los valores son los obtenidos en la implementación de C++ de Microsoft Visual C++ 9.0 (2008).

(\*1) Con la opción /J del compilador cambia el rango de valores del tipo char sin signo. En caso contrario, el valor es 127.

(\*2) Con la opción /J del compilador cambia el rango de valores del tipo char sin signo. En caso contrario, el valor es -128.

## Elección del tipo entero

Ante tal cantidad de posibilidades de tipos de datos nos surge la duda de qué tipo utilizar cuando llega el momento de definir una variable. Aquí se incluyen algunas recomendaciones generales:

- Si sabemos que la variable nunca almacenará un valor negativo, utilicemos un tipo sin signo ya que tendremos un rango de valores que duplica al mismo tipo con signo.
- Si conocemos exactamente el rango de valores que puede tomar la variable podemos elegir el tipo que mejor se ajusta a dicho rango.
- El tipo `int` es adecuado para la mayor parte de los casos y el que se procesa más velozmente; pero el tipo `long` nos asegura que el rango de valores nunca será menos de -2.147.483.648 a 2.147.483.647, ya que en todas las implementaciones de C++ siempre tendrá al menos 4 bytes.
- El tipo para una matriz muy larga debe ser pensado con más cuidado ya que la memoria ocupada es igual al tamaño de cada elemento de la matriz por el tamaño del tipo elegido. Cuando sea posible en estos casos conviene un `short` en lugar de un `int` o un `long`.

---

*Una matriz es una secuencia de variables del mismo tipo que comparten el mismo nombre y que para poder referirnos a un elemento determinado de la matriz utilizaremos un índice.*

---

### Dígitos decimales codificados en binario

Dígito decimal	Valor binario
0	0000 0000
1	0000 0001
2	0000 0010
3	0000 0011
4	0000 0100
8	0000 1000
15	0000 1111
255	1111 1111

## Asignación de valores a variables enteras

A una variable entera se le asignan valores numéricos sin decimales, podríamos asignar un valor con decimales pero en ese caso se truncaría directo la parte decimal

**Vistazo parcial del libro**  
**Salto de páginas...**

# Capítulo 8: Clases: Operadores y conversiones

## Contenido:

- **Sobrecarga de operadores**
- **Funciones friend**
- **Conversiones funcionales**
- **Conversiones implícitas y explícitas**

## Sobrecarga de operadores

Ya hemos visto que C++ utiliza una serie de operadores para trabajar con sus tipos básicos. Los operadores indican lo que podemos hacer con un tipo de dato. Un tipo `double` se puede sumar, restar, dividir, etc. pero C++ no nos permite multiplicar un puntero dado que es una operación que no tiene sentido.

Muchos de los operadores de C++ están sobrecargados; por ejemplo, el operador `*` puede significar el contenido de una dirección (si el operando es un puntero) o puede significar una multiplicación si sus operandos son de tipo numérico. C++ decide qué debe hacer en base al tipo de dato y la cantidad de operandos, es decir, dependiendo de la firma de una función que podríamos denominar `operator*()`.

Los tipos definidos por el usuario también tienen sus operadores y si definimos una clase también podemos definir qué operadores se pueden aplicar al objeto y cuál es la acción que se realiza en cada caso. Esto nos da una gran libertad de codificación.

Ya conocemos la clase **Empleado** que utilizamos en el capítulo anterior. ¿Qué sentido tendría sumar dos objetos de tipo **Empleado**? La respuesta es: ninguno o el sentido que necesitemos darle. Por ejemplo, la suma de objetos **Empleado** podría ser una operación no permitida (tal como lo es si no hacemos nada en particular) o

podría significar: generar un objeto **Empleado** con el sumatorio de miembro de datos **total\_debido** y el promedio de **valor\_hora**. Es cuestión que la clase **Empleado** implemente o no una función **operator+()**.

El uso de sobrecarga de operadores hace que nuestro código resulte más fácil de leer, por ejemplo, suponiendo que **a** y **b** son dos objetos **Empleado** podríamos escribir:

```
Empleado c = a + b;
```

Cuando C++ se encuentra con una operación como esta la interpreta de esta manera:

```
Empleado c = a.operator+(b);
```

Es decir, busca una función **operator+** en la definición de la clase y que además, en este caso, acepte como argumento un objeto de tipo **Empleado** (objeto **b**). El objeto **a** se trata dentro de la función como objeto **this**.

Los operadores que se sobrecargan deben ser operadores C++ válidos (no podemos inventar un operador). Pero no sólo eso, no todos los operadores C++ se pueden sobrecargar (aunque sí la mayoría ), después veremos la lista completa.

---

*La sobrecarga de un operador básicamente es crear una función **operator** son un código determinado. Esto nos da gran libertad de codificación pero debemos utilizar esta herramienta con criterio lógico. Aunque C++ no lo prohíbe, dentro de una función **operator+** lo que se espera encontrar es algo relacionado a una operación suma.*

---

## Sobrecarga en la práctica

Vamos a desarrollar una clase que representa un objeto **Hora** con métodos que nos permiten sumar tiempos en forma de horas, minutos y segundos o en forma de dos objetos **Hora**. En el primer enfoque haremos una implementación básica y posteriormente veremos cómo aprovechar la posibilidad de sobrecargar el operador **+** para simplificar la codificación en la suma de dos objetos **Hora**.

El objeto **Hora** tiene tres datos miembros que permiten almacenar la hora en forma de tres variables de tipo **int** para horas, minutos y segundos. El proyecto se denomina **cap08**.

El método **sumar()** está sobrecargado para permitir sumar horas, minutos y segundos o un objeto **Hora** al objeto **Hora** vigente.

El archivo de cabecera **cHora.h** es el siguiente:

```
#ifndef _CLASE_HORA_DEF_
```



```
#define _CLASE_HORA_DEF_

class Hora{
    int horas;
    int minutos;
    int segundos;
    void ajustar60();
public:
    Hora();
    Hora(int h, int m = 0, int s = 0);
    void sumar(int h= 0, int m = 0, int s =0);
    void sumar(const Hora & h);
    void Listar() ;
};
#endif
```

La clase prevé realizar un ajuste de los datos miembros para que los segundos y los minutos no excedan el valor 60, obviamente, el exceso se suma en el nivel siguiente y finalmente en el dato horas, que no tiene límite.

A continuación se incluye implementación de los métodos desarrollados en el archivo `cHora.cpp`.

```
#include "cHora.h"
#include <iostream>
using namespace std;

Hora::Hora()
{
    // Inicializa sin valores

    horas = 0;
    minutos = 0;
    segundos = 0;
}

Hora::Hora(int h, int m, int s)
{
    // Inicializa con valores

    horas = h;
    minutos = m;
    segundos = s;
}

void Hora::sumar(int h, int m, int s)
{
    // Suma horas, minutos y segundos
    // en el objeto Hora

    horas += h;
    minutos += m;
    segundos += s;
}
```

```
void Hora::sumar(const Hora & h)
{
    // Suma de dos objetos Hora
    // con la sintaxis
    // x1.sumar(x2);
    //Hora nHora;

    horas += h.horas;
    minutos += h.minutos;
    segundos += h.segundos;
}

void Hora::Listar()
{
    // Cada vez que lista el objeto
    // corrige los valores de
    // horas, minutos y segundos
    // para que minutos y segundos no superen
    // el valor de 60

    ajustar60();

    cout << "HH::MM:SS = " << horas << ":" << minutos << ":"
         << segundos << "\n\n";
}

void Hora::ajustar60()
{
    // Ajusta que los datos miembros
    // minutos y segundos no excedan
    // de 60

    minutos += segundos / 60;
    segundos %= 60;
    horas += minutos / 60;
    minutos %= 60;
}
```

En este código se destacan los dos métodos `sumar()` que, en un caso, recibe como argumento un objeto `Hora` en forma de referencia y que, en el otro caso, recibe valores enteros que representan horas, minutos y segundos.

En la ejecución del código podemos comprobar que los dos métodos `sumar()` funcionan correctamente. El método privado `ajuste60()` hace que los valores de minutos y segundos no excedan ese límite, pasando el excedente al nivel siguiente mediante la correspondiente conversión de unidades. Si quisiéramos añadir el dato miembro `día` sería muy sencillo controlar que las horas no superen el valor 23 y en tal caso incrementar en 1 el valor del dato miembro `día`. Pero, por ahora seguiremos con el enfoque original de la clase `Hora`.

El código fuente del archivo `cap08.cpp` es el siguiente:

```
#include <iostream>
#include "cHora.h"
```

```
using namespace std;

int main()
{
    // Modos de inicialización
    Hora H1;
    Hora H2(20, 50, 55);
    Hora H3(4, 77, 99);

    cout << "Caso Inicializaciones \n";
    H1.Listar();
    H2.Listar();
    H3.Listar();

    // Suma con el método sumar
    H2.sumar(H3);
    cout << "Caso Hora::sumar(Hora) \n";
    H2.Listar();

    Hora H4;
    // reasignar la hora
    H4 = Hora(100, 400, 500);
    cout << "Caso reasigna objeto \n";
    H4.Listar();

    // Suma un objeto Hora
    // con valores independientes de horas
    // minutos y segundos
    H4.sumar(5, 1, 1);

    cout << "Caso Hora::sumar(h,m,s) \n";
    H4.Listar();
    return 0;
}
```

La ejecución da este resultado:



```
C:\ d:\Vando Lalo\Datos\21Lalo\00 51 C++\Proyectos\cap08\Debug\cap08.exe
Caso Inicializaciones
HH:MM:SS = 0:0:0
HH:MM:SS = 20:50:55
HH:MM:SS = 5:18:39
Caso Hora::sumar(Hora)
HH:MM:SS = 26:9:34
Caso reasigna objeto
HH:MM:SS = 106:48:20
Caso Hora::sumar(h,m,s)
HH:MM:SS = 111:49:21
```

Hasta ahora todo correcto. Pero, no sería interesante que nuestra suma de horas pudiese hacerse de esta manera:

```
H4 = Hora(10,40,50);
Hora H5 = Hora(20, 40,20);
Hora H7 = H4 + H5; // Resultado: 31:30:10
```

Queda evidente que se trata de una suma de dos objetos `Hora` y el código es más legible. De modo predeterminado, el operador suma (+) no está capacitado para trabajar con dos objetos `Hora` como operandos.

Crearemos una función `operator+` en nuestra clase con exactamente el código del método `Hora::sumar(const Hora & h)` (después de crear el prototipo en la clase). Ahora la definición de la clase quedaría de esta manera:

```
// Archivo de cabecera clase Hora (cHora.h)
//

#ifndef _CLASE_HORA_DEF_
#define _CLASE_HORA_DEF_
class Hora{
    int horas;
    int minutos;
    int segundos;
    void ajustar60();
public:
    Hora();
    Hora(int h, int m = 0, int s = 0);
    void sumar(int h= 0, int m = 0, int s =0);
    void sumar(const Hora & h);
    void Listar() ;
    Hora operator+(const Hora & h) const;
};
#endif
```

Y el nuevo método es una simple copia del anterior `Hora::sumar()`, con su nuevo nombre `Hora:operator+()`:

```
Hora Hora::operator+(const Hora & h) const
{
    // Suma de dos objetos Hora
    // con la sintaxis
    // x1 = x2 + x3;

    Hora nHora;
    nHora.horas = horas + h.horas;
    nHora.minutos = minutos + h.minutos;
    nHora.segundos = segundos + h.segundos;
    return nHora;
}
```

**Vistazo parcial del libro**  
**Salto de páginas...**

# Capítulo 10: Herencia de clases

## Contenido:

- Reutilización de código
- Herencia de clases
- Relación "es un"
- Clases derivadas
- Acceso protected
- Polimorfismo
- Funciones virtuales
- Clases abstractas

Uno de los principales objetivos de la programación orientada a objetos es posibilitar la reutilización del código para agilizar y simplificar los desarrollos. Cuando desarrollamos un proyecto siempre intentamos utilizar componentes probados de bibliotecas de código como un medio de obtener con mayor rapidez un programa más seguro y fiable; por supuesto existe la alternativa de generar código nuevo pero esa posibilidad comparada con la reutilización de código ya probado siempre requiere más tiempo de desarrollo y mayor carga de trabajo en las pruebas. Es evidente que la reutilización del código ahorra tiempo y genera un código más fiable y nos permite concentrarnos más en los problemas nuevos que debe resolver nuestro programa.

Las bibliotecas de funciones suministran un modo práctico de reutilización de código, a nadie se le ocurriría escribir una función para resolver una tangente o una raíz cuadrada.

El uso de clases en C++ nos conduce a un nuevo modo de reutilización del código que nos permite dos cosas: reutilizar código probado y además, extender o

ampliarlo para adaptarlo perfectamente a nuestras necesidades. Este mecanismo se denomina herencia de clases, que nos permite crear clases a partir de una clase ya existente. El estudio de este concepto es el tema principal de este capítulo.

## Herencia de clases

La herencia es un mecanismo de la programación orientada a objetos que sirve para definir una clase nueva tomando como base otra clase ya existente, que se denomina clase base o superclase. Por ejemplo, si tenemos una clase que resuelve el problema de la gestión de una cuenta bancaria normal y queremos crear una cuenta bancaria bonificada, no tenemos que redefinir todo el código, simplemente podemos heredar todo el funcionamiento de la clase de la cuenta bancaria normal para crear una clase hija de esa clase (la denominación correcta es clase derivada o subclase), la cuenta bancaria bonificada. A partir de ahí, en la clase derivada se incluyen las particularidades de esa clase, sólo lo que la diferencia de una cuenta bancaria normal. Esto implica que tendremos dos clases, la clase base (que podremos seguir utilizando como antes) y la clase derivada.

Una clase derivada puede, por su parte, servir como clase base de otra nueva clase. De esta manera se puede generar una cadena jerárquica de herencias en donde todas las clases derivan de otra, salvo la primera de todas.

---

*La teoría de la programación orientada a objetos contempla la posibilidad de que una clase pueda derivarse de una o más clases base, más adelante veremos la herencia múltiple, por ahora analizaremos el caso de una única clase base.*

---

Suponiendo que tenemos una clase **Empleado**, la sintaxis para crear una clase derivada denominada **Gerente** es la siguiente:

```
class Empleado {
private:
...
public:
    void met1();
}

class Gerente : public Empleado {
private:
...
public:
    void met2();
}
```

La sintaxis que se utiliza para derivar de otra clase son dos puntos seguidos por el nombre de la clase base (opcionalmente se puede indicar un calificador de acceso, tal como se explicará más adelante).

La creación de una instancia o ejemplar de cada una de estas clases utiliza una sintaxis ya conocida:

```
Empleado empl;  
Gerente ger1;
```

Y los métodos se utilizan de la siguiente manera:

```
empl.met1();  
ger1.met2();
```

### ¿Herencia o contención?

A veces suelen confundirse los conceptos de herencia y contención. Sabemos que una clase está compuesta por datos miembro de diversos tipo (**int**, **double** o cualquier otra clase), por ejemplo, la clase **Empleado** podría tener un dato miembro de la clase **Salario** (ya que en lugar de ser un tipo **double** podría ser una clase específica con sus datos y sus métodos). Cuando esto sucede se dice que la clase **Empleado** "contiene" a la clase **Salario**. Aquí no se aplica el mecanismo de herencia ni ninguna de sus reglas (que veremos en este capítulo). La clase contenida forma parte de la clase contenedora como un miembro más.

La herencia nos permite tomar como base una clase para crear otra clase más especializada y el mecanismo de herencia determina de qué manera se reciben por herencia los datos y funciones miembro de la clase base. Es un concepto más complejo que la contención al que prácticamente le dedicaremos todo este capítulo.

## Creación de una clase base simple

Para explicar el mecanismo de herencia utilizaremos una clase ya conocida, la clase **Empleado**. La clase **Empleado** es una clase candidata para ser una clase base ya que cuando nos encontramos ante la necesidad de crear un objeto **Gerente** nos damos cuenta que los gerentes son empleados más jerarquizados, con particularidades que los diferencian del resto de los empleados; pero, en definitiva, un gerente es un empleado. Un empleado tiene nombre, dirección, teléfono, número de seguridad social, salario, estudios, antecedentes laborales, fecha de ingreso a la empresa, etc. Todos esos datos miembro también sirven para una clase **Gerente**, pero el gerente tiene particularidades que lo diferencian de un empleado normal: participación en los beneficios, asignación de vehículo, plus por objetivos, etc. Si tenemos que crear una clase **Gerente** no sería lógico que tengamos que codificar todo lo que tiene en común con un empleado normal (código de validación de dirección, teléfono, seguridad social, etc.). Lo lógico es crear una clase **Gerente**



**Vistazo parcial del libro**  
**Salto de páginas...**

# Capítulo 13: Excepciones

## Contenido:

- Errores comunes en C++
- Gestión básica de errores
- Gestión de excepciones
- throw-try-catch
- Relanzado de excepciones
- try anidados
- Clase Exception

Por más experiencia que se tenga como programador nuestro código no estará exento de generar errores. Incluso un programa perfectamente codificado y utilizado miles de veces puede encontrarse en una situación de excepción por problemas ajenos al propio programa, por ejemplo, un archivo no encontrado, un archivo corrupto por problemas físicos en la unidad, falta de memoria para la creación de nuevos objetos, la no disponibilidad de un recurso de red, falta de permiso o permiso insuficiente para acceder a un recurso, y podríamos seguir con una lista muy larga. Además están los errores propios del programa en sus primeras pruebas y los errores que quedan ocultos después de esas primeras pruebas y que saldrán a la luz cuando se da una determinada condición.

---

*Siempre se dice que es más sencillo prevenir los errores que detectarlos después que provocan una cancelación del programa o cuando generan un resultado incorrecto.*

---

La programación orientada a objetos y la reutilización del código nos ayudan a que nuestros programas tiendan a generar menos errores que con las técnicas anteriores de programación. Pese a todas las ayudas, es inevitable que nos tengamos que enfrentar a los errores de un programa. Algunos de los fallos son evidentes y se detectan de inmediato con sólo observar los resultados producidos o al ver que el programa queda bloqueado; en cambio hay otros tipos de errores que aparecen muy de vez en cuando y por lo tanto son más difíciles de detectar. La tarea de detección de un error se comienza con el intento de repetir la situación que lo produjo y analizando paso a paso qué es lo que va haciendo el código para llegar al punto en el que está mal resuelto y corregirlo.

Una herramienta muy útil para la detección de los errores es el uso de la macro **assert** (en el archivo de encabezado **assert.h**):

```
...
assert(ptr == NULL); // mensaje + cancelación si ptr no es NULL
...
```

Esta macro generaría un error si **ptr** no es **NULL**, es decir, si no se cumple la condición de la macro se genera un error. Las macros **assert** se pueden activar o desactivar con facilidad: definiendo o no la macro **NDEBUG**, que se debe codificar antes de la inclusión de **assert.h**.

```
// NDEBUG desactiva assert sin necesidad de quitar
// las macro del programa
#define NDEBUG
#include "assert.h"
```

La macro **assert** se utiliza principalmente durante el desarrollo de un programa mediante la inclusión de una expresión que genera un mensaje de diagnóstico cuando se evalúa como false y posteriormente cancela el programa mediante un llamado a la función **abort()**. Cuando la expresión se evalúa como true, **assert()** no hace nada y el programa continúa su ejecución de modo normal.

## Errores que pueden aparecer en un programa C++

Anteriormente mencionamos varios de los errores más comunes que suelen aparecer en los programas C++ y que no son necesariamente errores del código. Veamos una lista de los errores más comunes que encontraremos al probar nuestros programas:

- **Errores por falta de memoria:** puede ser un error del programa o del propio sistema que se ha quedado sin memoria. Un típico error de programa es la adquisición repetida e infinita de memoria en un bucle fuera de control. Deberíamos verificar que las funciones de reserva de memoria hayan devuelto

un valor correcto. También deberíamos verificar si se está liberando la memoria reservada al dejarla de utilizar.

- **Errores en un bucle de proceso:** si nos olvidamos de incrementar la variable que controla la ejecución de un bucle nos encontraremos que la ejecución queda encerrada dentro del bucle indefinidamente absorbiendo el uso del procesador hasta llegar a un 100%. Esto también puede ocurrir cuando la condición de fin del bucle no se cumple nunca.
- **Errores de límites de matrices:** si tratamos de acceder a una matriz con un índice fuera de los límites tendremos un error seguro pero difícil de determinar anticipadamente. El uso de matrices seguras es un mecanismo útil contra este tipo de error.
- **Errores por división por cero:** el error más típico de todos y más fácil de erradicar: comprobar siempre el valor del divisor antes de realizar la división y ejecutar un código alternativo cuando se cumpla esa condición (por ejemplo, emitir un mensaje). De todas maneras, hay casos en los que por la lógica del programa no es posible que el divisor sea cero y quizá el origen del error se encuentre en otro algoritmo del programa.
- **Errores por desborde de la pila:** cuando se utilizan funciones recursivas existen un riesgo de desborde de la pila por la cantidad de variables que pueden estar activas.
- **Errores con punteros:** el uso incorrecto de los punteros es el motivo principal de los errores en C++. No sólo eso, es un tipo de error muy difícil de detectar ya que el error se puede manifestar varias líneas después de la asignación del puntero. El uso de la memoria dinámica debe hacerse de modo controlado y se deben evitar los punteros innecesarios.
- **Errores en conversiones:** los truncamientos en las conversiones aritméticas pueden ser inofensivos cuando los valores no superan los límites pero pueden ser fatales cuando hay pérdidas reales de datos.
- **Errores por portabilidad:** C++ es un lenguaje muy transportable, no obstante si se utilizan opciones que dependen del sistema tendremos problemas seguros al llevar los programas a otro entorno.

## Gestión básica de los errores

Los errores se deben gestionar lo más cerca posible de donde se pueden producir. Cuando diseñamos una clase debemos tener en cuenta que cada método tiene que saber detectar sus errores y debe retornar alguna indicación de lo ocurrido para que el usuario de la clase pueda gestionar el error producido. Si el error generado se puede gestionar con un tratamiento específico, el proceso puede continuar, en caso contrario se eleva el problema un nivel a la función llamante y así sucesivamente hasta que algún nivel del programa sepa qué hacer con esa condición de error, y eventualmente, si no hay un tratamiento del error al llegar hasta la función `main()`,

el programa termina cancelando su ejecución porque pasará el error al sistema operativo.

Esta cadena de análisis del error es un mecanismo lógico pero hay ciertos errores que son tan graves que lo mejor es interrumpir el programa mediante la función `abort()` o la función `exit()`. La función `exit()` nos permite devolver un código al sistema operativo. Este modo de terminación de un programa no debe ser la norma sino la excepción ya que el programa termina sin liberar recursos.

---

*La función `exit()` permite devolver un valor de retorno al sistema operativo; normalmente, si se devuelve el valor cero se considera una terminación correcta.*

---

La función `abort()` tiene su prototipo en el archivo de encabezado `stdlib.h` (`csdtdlib.h`). La gestión típica de un error, antes de uso de las excepciones que veremos a continuación, era enviar un mensaje del estilo "el programa cancela por..." en el flujo de salida de errores y finalizar el programa finaliza. La función `abort()` hace que el programa finalice de inmediato sin regresar a la función `main()`.

---

*La función `abort()` no devuelve el control a la función llamante, cancela el proceso vigente y devuelve un código de error.*

---

Un enfoque más práctico que la cancelación abrupta con `abort()` es utilizar un valor de retorno en la función para señalar la existencia de un problema. Esto a veces no es posible porque la función puede utilizar el valor de retorno para otros fines. Una alternativa es utilizar un argumento adicional de tipo referencia en la llamada a la función para que actúe como valor de error (0 podría ser, no hubo error).

## Gestión de excepciones

C++ incluye un esquema de tratamiento de excepciones basado en la estructura `try`, `catch` y `throw`. Veamos cómo gestionar los problemas de ejecución con el mecanismo de las excepciones. Una excepción C++ es una situación especial que ocurre durante la ejecución del programa, por ejemplo, no encontrar un archivo en el disco duro o una división por cero.

La gestión de excepciones está compuesta por tres elementos:

- Lanzamiento o generación de la excepción (**throw**)
- Detección de la excepción mediante un gestor (**catch**)
- Definición de un bloque de control (**try**)

---

*Lo que quedará como enseñanza de esta sección es que la gestión de excepciones se debe pensar en conjunto con el programa o la clase y no como un añadido posterior.*

---

La gestión de excepciones es un mal necesario que debemos asimilar: sabemos que la gestión de excepciones hará que el programa sea más largo y algo más lento. También se verá que la gestión de excepciones no se lleva muy bien con los templates, dado que las funciones genéricas lanzan diferentes tipos de excepciones según la versión de la función que se esté utilizando en cada caso. Pero es evidente que la gestión de excepciones es imprescindible en todo programa profesional y no se puede prescindir de ella en ningún caso.

Para analizar este mecanismo comenzaremos por la palabra clave **throw**.

## throw

Cuando surge un problema en la ejecución normal del código se genera una excepción (o se lanza). La palabra clave **throw** nos permite generar esa excepción cuando nuestro código detecta una situación anormal (una excepción). La sentencia **throw** es el reemplazo a la función **abort()**. Desde el punto de vista de ejecución, la sentencia **throw** es una bifurcación del programa hacia otra dirección del programa, saltándose todo lo que haya en el medio.

La sentencia **throw** está seguida de un valor que puede ser una cadena de texto, un número o un objeto de cualquier clase. Este valor que se utiliza en la sentencia **throw** es fundamental para el proceso posterior de la excepción, tal como veremos al analizar la sentencia **catch**.

```
if (ptr == NULL)
    throw "Puntero nulo";
...
```

Si se cumple esa condición de error la sentencia **throw** genera la excepción y el proceso de ejecución vuelve a la función llamante (se abandona la ejecución de la función vigente) como si fuese un **return**.

```
void a()
{
    ...
    b();
    // ¿qué hacer aquí?
    ...
}
void b()
{
    ...
    if (ptr == NULL)
        throw "Puntero nulo"; // retorna a la función a()
    ...
}
```

**La pregunta es: ¿hasta qué nivel sube la ejecución?**

## catch

El bloque **catch** es la respuesta a la pregunta que quedó pendiente en la sección anterior. La excepción se eleva al nivel previo hasta que aparece una palabra clave **catch** con la clase coincidente al objeto lanzado por la sentencia **throw**. Si la sentencia **throw** lanzó una excepción con un mensaje de texto, la clase de la sentencia **catch** debe ser una cadena. Si la sentencia **throw** hubiera lanzado una excepción con un valor numérico, la clase de la sentencia **catch** tendría que haber sido de una clase numérica.

La sentencia **catch** es un gestor de excepciones que sabe qué hacer cuando ocurre un determinado tipo de excepción. Por ejemplo, si la excepción es porque no se encuentra un archivo, el bloque **catch** correspondiente a esa excepción puede saber dónde abrir un archivo auxiliar para seguir con el proceso.

Un gestor o bloque **catch** comienza con la palabra clave **catch** seguida por una declaración de un tipo entre paréntesis. Este tipo es el que permite indicar al gestor **catch** si debe intervenir o no ante una excepción. A continuación se define un bloque de sentencias que son las que se ejecutan si hay coincidencia entre la clase de la excepción lanzada y la clase de la sentencia **catch**.

```
void a()
{
    ...
    b();
    // ¿qué hacer aquí?
    catch (char * txt){
        // Tratamiento de la excepción
        ...
    }
    ...
}
```

**Vistazo parcial del libro**  
**Salto de páginas...**



# Apéndice A: Respuestas a las prácticas

## Respuestas del capítulo 1

1. Función `main()`
2. Al finalizar la función `main()`.
- 3.

```
cout << "Programa C++";
```

- 4.

```
V1 = 8; // cualquier valor entero estará bien
```

- 5.

```
cin >> J;
```

6. Indicar Verdadero o Falso:

6.1V

6.2F

6.3V

6.4F

6.5V

6.6V

6.7F

6.8V

6.9V

6.10 F

## Respuestas del capítulo 2

1. Se pueden clasificar en tipos enteros y reales.
2. `char`, `short`, `int`, `long`.

**3. Definiciones:**

1. `long x1 = 700L;`
2. `char x2 = 'X';`
3. `float x3 = 5.2f;`
4. `double x4 = 500045e-2; // hay varios resultados correctos`
5. `bool x5 = true;`

4. Hay un desborde por falta de capacidad, el resultado es incorrecto.

5. Para saber en qué orden se realizan las operaciones en una expresión aritmética.

6. Para forzar un orden diferente a la establecida por la precedencia de operadores.

7. Mediante el uso de `cast short`.

8. ¿Qué resultados se obtienen en estos cálculos?

1. -144
2. -28
3. 30
4. -2
5. -2,666..

9. ¿Cuál es el resultado de estas operaciones?

`u3 = 7`

`u4 = 7,09`

10. C++ permite diversos tipos de datos numéricos, con diferentes tamaños y capacidades, para que podamos elegir el tipo adecuado para nuestras necesidades. Si bien el tipo `long double` es el de mayor capacidad, su tamaño es totalmente inadecuado si simplemente queremos definir una tabla de códigos de 1 byte. Por otra parte, los procesadores están optimizados para realizar operaciones con `int`, por lo que para una operación no se precisan decimales, la mejor opción es `int`. Todos los tipos tienen su razón de ser.

## Respuestas del capítulo 3

1. Declara las siguientes variables:

1. `int mat1[200];`
2. `float mat2[50];`
3. `short mat3[5][30][4];`
4. `int mat4[2]={47, 3};`
5. `char mat5[] = "AZUL";`
6. `double mat6[] = { 56.89, 45.567, y -1};`
7. `enum color {AZUL, ROJO, VERDE=7};`

```
8.
struct EST1{
    char* nombre;
    int edad;
    double salario;
};
```

```
9.
EST1 var1 = {"Pedro", 37, 2500.45};
```

```
10.
double var2;
double * ptr = &var2;
```

2. Indicar si estas declaraciones son correctas o incorrectas:

```
1. float campo4[2] ;           // Correcta
2. campo4[2] ={5.1, 1.3};      // INCORRECTO
```

3. Escribir una sentencia para imprimir lo siguiente:

```
1. cout << mat1[1];
2. cout << "1." << var1.nombre << " 2." << var1.edad << "
3."
   << var1.salario;
3. cout << int(&X);
```

4.

```
EST1 * var2 = new EST1;
```

5. Referencias: Una manera de llamar a un objeto con un nombre alternativo.

6.

```
0x1010
```

## Respuestas del capítulo 4

1. Un bloque está definido por 0, 1 o más de una sentencia.
2. Falso.
3. !, &&, ||
4. Falso. Postincremento tiene mayor prioridad que preincremento.
5. for, while y do-while.
6. do-while.

7. 3 5 7

8. 1 30

3 29

9 28

27 27

9.

```
if (edad < 11)
    fun10();
else if (edad < 21)
    fun20();
else fun21();
funtodos();
```

10. Hay dos maneras para definir un alias para un tipo de datos:

**#define:** Instrucción al preprocesador que se encarga del reemplazo de un alias por un tipo real.

**typedef:** Definición de un alias para un tipo real que procesa el compilador.

## Respuestas del capítulo 5

1. La respuesta normal sería no: no se puede utilizar una función si no se declara previamente un prototipo (dentro del propio archivo cpp o en un archivo de cabecera .h). Pero lo cierto es que hay una excepción en la que no se precisa el prototipo: cuando se incluye la definición de la función antes del primer uso (antes de la primera llamada).

2. Prototipos:

```
1. float áreaCuadrado(float);
2. double áreaTriángulo(double, double);
3. double & pos(struct4);
4. long cálculo(const long [], int);
5. int cadena(const char *);
6. void comprobar(struct4);
```

3. Una solución posible:

```
double ej3 (float x[], int q)
{
    double suma = 0;
    for (int i = 0; i < q; i++)
        suma += x[i];
    suma /= q;
    return suma;
```

}

4. El primer método de protección de argumentos es pasarlos por valor. Cuando esto no es posible porque se pasa un puntero a los datos (caso de las matrices o cadenas), se utiliza el calificador **const**.
5. Las matrices y las cadenas se pasan mediante un puntero, por lo que para impedir que sean modificados en la función llamada hay que hacer que los datos apuntados se mantengan constantes mediante el calificador **const**.
6. Nada en particular. En C++ se presupone el pase por valor. Una estructura, si no utilizamos **&**, se pasa por valor.
7. Una solución posible:

```
int cuentaChar (const char * texto, char c)
{
    int suma = 0;
    for (int i = 0; *texto ; i++)
    {
        if (*texto == c)
            suma++;
        texto++;
    }
    return suma;
}
```

8. Falso. Pero sí se puede devolver una matriz si está dentro de una estructura o de otro tipo de objeto. También es posible actualizar una matriz pasada como argumento y después el programa principal (llamante) puede utilizar la matriz modificada en la función. Esto último no es estrictamente un valor de retorno pero a los fines prácticos funciona como tal.
9. Porque una matriz se pasa como puntero (aunque codifiquemos su nombre con corchetes) al tipo de datos de la matriz (puntero a **int**, **double**, etc.); la función desconoce la cantidad de elementos. En el caso de una cadena (que básicamente es una matriz de **char**) no es necesario indicar la longitud de la cadena porque el elemento final es un elemento con 0 binario (carácter nulo).
10. La respuesta correcta es a.

Para lograr lo establecido en el punto b, la codificación debería ser:

```
int * const ptrx = &variable;
```

Para lograr lo establecido en el punto c, la codificación debería ser:

```
const int * const ptrx = &variable;
```

11.

```
void fnx(const char *);
```

12. La respuesta correcta es 12.b.
13. La compilación es correcta, pero la llamada a la función exige una conversión aritmética de long a short que puede producir una pérdida de datos. De hecho, con la constante 60000L se produce una conversión con pérdida de datos.
14. No compilará porque el compilador asumirá que `fnx3()` es una variable de tipo `double` y no el nombre de una función.
15. Indicar si estos prototipos son correctos o no:

- a. `void fnx1(int, int, int i1=20, int i2= 30);`  
// Correcto
- b. `void fnx1(int i0 = 300, int, int i1=20, int i2= 30);`  
// Incorrecto
- c. `void fnx1(int, int, int i1=20, int i2= 30, int);`  
// Incorrecto

16. Dado este prototipo:

```
void fnx1(int, int, int i1=20, int i2= 30, int i3=0);  
int x1, x2, x3, x4, x5;
```

- Indicar si estas llamadas son correctas:

- a. `fnx(x1, x2, x3, x4, x5);` //Correcta
- b. `fnx(x1, , x3, x4, x5);` // Incorrecta
- c. `fnx();` // Incorrecta
- d. `fnx(x1, x2, x3, , x5);` // Incorrecta
- e. `fnx(x1, x2);` // Correcta

17. Es aquella función que dentro de su bloque de código incluye una llamada a sí misma.
18. La función `inline` no puede ser recursiva porque el compilador debe incluir el código de la función en el sitio en donde aparece la llamada a la función. Si la función `inline` fuese recursiva el compilador debería saber cuántas veces se produce la recursión y eso sólo se sabe en tiempo de ejecución.
19. Código para recuperar los argumentos de la línea de comandos:

```
for ( int i = 0 ; i < argc ; i++ )  
    cout << "Lista de argumentos de la línea de comandos : "  
        << argv[i] << "\n";
```

**Vistazo parcial del libro**  
**Salto de páginas...**

# Índice

## Símbolos

! 117  
!= 113  
#define 53, 126  
#define NDEBUG 368  
#else 30  
#endif 30  
#if 30  
#if !defined 30  
#ifdef 30  
#ifndef 30  
#ifndef (instrucción al preprocesador)  
217, 230  
#include 28, 29  
#include <cmath> 40  
#include <iostream> 40  
%= (módulo y asigna) 123  
&, operador de bit 113  
&& 117  
&&, AND lógico 117  
\*= (multiplica y asigna) 123  
+= (suma y asigna) 123  
- (unario); complemento a 2 113  
-= (resta y asigna) 123  
/= (divide y asigna) 123  
:: operador de ámbito (o campo) 194  
< 112  
<< 33  
<<, operador de inserción 34  
<<; desplazamiento de bits 113  
<= 112  
<functional> 445  
<iterator> 437  
<memory> 364  
== 112  
> 113  
>= 113  
>>, operador de extracción 34  
>>; desplazamiento de bits 113  
?, operador condicional 132  
\include 30  
\_intn 49  
|, operador de bit 113  
|| 117

~ (unario): complemento 1 113  
¿C++, Java o C#? 18  
¿Para qué sirve el prototipo? 142  
¿using es igual a using namespace? 198

## A

abort(), función 370  
abrir un archivo 403  
abstracta, clase 307  
acceso a los miembros según el modo de  
herencia 282  
acceso binario 406  
acceso directo 414  
acceso random, iterador 435  
accumulate(), algoritmo 428  
adaptadores de contenedores 423  
adjacent\_difference(), algoritmo 428  
adjacent\_find(), algoritmo 428  
adjustfield 397  
algoritmos 421, 426  
algoritmos aritméticos 428  
algoritmos buscadores 428  
algoritmos clasificadores 430  
algoritmos de asignación 431  
algoritmos de comparación 429  
algoritmos modificadores 429  
algoritmos no modificadores 427  
alias de espacios de nombres 196  
alias de tipos 126  
almacenamiento 184  
almacenamiento de las funciones 193  
almacenamiento de las variables automáticas  
186  
almacenamiento dinámico 193  
alternativa al uso de variables estáticas  
globales 199  
ambigüedades en la sobrecarga de funciones  
162  
ámbito de archivo 185  
ámbito de bloque 185  
ámbito de clase 185  
ámbito de espacio de nombres 185  
ámbito de la clase 208  
ámbito de las variables 184



- ámbito de una clase 225
- ámbito global 185
- ámbito local 185
- AND lógico 117
- AND lógico (&&) 117
- anidado de clases o contención 337
- anidado de if-else 131
- apuntadores (punteros) 92
- árbol binario 426
- Archivo C++ (cpp) 24
- archivos 402
- archivos de cabecera 181
- archivos de código (implementación) 31
- archivos de código fuente 24
- archivos de salida del compilador 31
- archivos de texto 404
- argumento argc 160
- argumento argv 160
- argumento predeterminado 159
- argumentos de función (pasados por referencia) 143
- argumentos de función (pasados por valor) 143
- argumentos desde la línea de comandos 160
- argumentos por valor o por referencia 157
- argumentos predeterminados 179
- array 43
- Arrays: matrices de variables 76
- ASCII 42
- asignación de un puntero a char con un literal 102
- asignación de valor a una variable 37
- asignación de valores 55
- asignación del ancho y la precisión 398
- asm 43
- asociatividad 114
- asociatividad de los operadores C++ 114
- assert 368
- auto 43, 186
- auto\_ptr 363
- Automático 95, 184

## **B**

- back\_insert\_iterator, iterador 437
- bad\_cast 361
- bad\_typeid 362
- basefield 397
- basic\_ios 389
- basic\_streambuf 389

- biblioteca STL 421, 426, 468
- bibliotecas de funciones 40
- binarios, flujos 387
- binary\_search(), algoritmo 428
- bits, operadores de desplazamiento 113
- Bjarne Stroustrup 18
- bloque anidado 185
- bloque de una función 26
- bloques de sentencias 112
- bool 43, 50
- boolalpha 397
- Borland C++ 19
- break 43, 134
- bucles 119
- búfer 386, 387
- búferes de entrada/salida 387
- búsqueda de una función 193
- búsqueda en la tabla direcciones 305

## **C**

- C 18
- C con clases 18
- C# 18
- C++
  - bloques de sentencias 112
  - bucles 119
  - cuadro resumen de operadores 114
  - definición de función main() 31
  - normas generales del formato 34
  - sentencia que produce salida de datos 32, 34
  - sentencias condicionales y de bifurcación 128
  - sentencias declarativas 111
  - tipos fundamentales 48
- C++ siempre exige una declaración 141
- cadena 81
- cadena como valor de retorno 155
- calificador const 73
- capítulo 11: Contención 311
- carácter nulo ('\0', ASCII 0) 81
- caracteres ASCII 42
- características de las matrices o arrays 81
- case 43
- casos de estudio 254
- cast 72
- cast implícito (upcasting) 300
- cast, operadores 362
- catch 43, 372

- cerr 388
- cerr no implementa un búfer 388
- cerrar un archivo 404
- ciclo de desarrollo de un programa 19
- ciclo de vida de las variables 95
- cin 34, 388
- clase abstracta 307
- clase auto\_ptr 364
- clase base 276
- clase con varios constructores 213
- clase de excepción utilizando puntos suspensivos 374
- clase derivada 276
- clase derivada, declaración 283
- clase derivada, secuencia de construcción 284
- clase derivada: cómo llama al constructor de la cl 286
- clase derivadas: constructor copia 287
- clase e instancia 209
- clase Empleado como contenedora 321
- clase Empleado con herencia privada 329
- clase hija 276
- clase pair 426
- clase string como contenedor STL 424
- clase VectorSalario 316
- clases anidadas 337
- clases de almacenamiento 184
- clases Friend 336
- clases friend y templates 354
- clases genéricas o templates 346
- clases que contienen clases 312
- clases: calificador de acceso 282
- class 43
- clear() 417
- clock() 125
- clock\_t 125
- CLOCKS\_PER\_SEC 125
- clog 388
- CLR 18
- cmath 39
- codificación del programa 19
- códigos de escape C++ 59
- coincidencia exacta 162
- coincidencia trivial 162
- coincidencias exactas y conversiones triviales 181
- cola con prioridades, contenedor STL 425
- cola de doble final 425
- colas (Queue), contenedor STL 425
- coma flotante 60
- coma, operador 121
- combinación de dos listas en una 440
- comentarios 25, 26
- comentarios estilo C 27
- cómo se convierte un número de hexadecimal a decim 56
- cómo se convierte un número de octal a decimal 56
- cómo se reciben los parámetros en un programa 160
- compilación condicional 30
- compilación del código fuente 19
- concepto de clase 205
- concepto de contenedor 423
- conexión del flujo de datos 386
- conflictos de nombres 195
- conjunto de caracteres ASCII 42
- const 43, 73, 150, 192
- const utilizado con punteros 150
- const\_cast 43, 362
- constante 53
- constantes de cadena 63
- constantes de tipo real 60, 61
- constantes enteras 57
- constantes o literales de carácter 63
- constructor conversión 289
- constructor copia 215, 434
- constructor copia en clases derivadas 287
- constructor copia personalizado 259
- constructor copia predeterminado 259
- constructor predeterminado 212, 214, 434
- constructores inline 322
- constructores múltiples y operador new 272
- constructores, varios por clase 213
- constructores y destructores de objetos 210
- contador de instancias 253
- contención 311
- contención o anidado de clases 337
- contención o herencia privada 333
- contenedor 421
- contenedores asociativos 422, 426
- contenedores de la biblioteca STL 422
- contenedores de secuencia 422
- continuación tras una excepción 374
- continue 43, 135
- continue, en excepciones 374
- conversión automática de tipos 243
- conversión de punteros 363
- conversión en asignación 68

conversión explícita (downcasting) 300  
 conversión implícita con el objeto cout 247  
 conversiones de tipo 67  
 conversiones en expresiones 70  
 conversiones en las clases 242  
 conversiones estándar 163  
 conversiones implícitas 244  
 conversiones implícitas y tipos de herencia 335  
 conversiones manuales 72  
 conversiones triviales 181  
 copia dato miembro a dato miembro 293  
 copia implícita 293  
 copy(), algoritmo 429  
 copy\_backward(), algoritmo 429  
 count(), count\_if(), algoritmos 428  
 cout 32, 388  
 cout/cin con ostream\_iterator e istream\_iterator 435  
 creación de instancias de un objeto 220  
 creación de un ejemplar de la clase 209  
 creación de un programa 19  
 creación de un programa desde la línea de comandos 20, 22  
 creación de una clase base simple 277  
 creación un contenedor multimap 444  
 creación y destrucción de un objeto de una clase d 284  
 crear una variable de un objeto 209

## CH

Char 57  
 char 43, 48  
 checked\_array\_iterator, iterador 437

## D

datos protected para el método modificarSalario() 291  
 dec 397  
 decimal 56  
 declaración con inicialización en un paso 90  
 declaración de la clase Empleado 216  
 declaración de variables 36  
 declaración friend 336  
 declaración using, modificación acceso heredado 334  
 deep copy 293  
 default 43  
 default en switch 134

defined 30  
 definición con inicialización de elementos 79  
 definición de función main() 31  
 definición de la función 140  
 definición de matrices 78  
 delete 43, 99  
 dependencia entre función y variable global 142  
 deque, contenedor STL 425  
 desborde de la pila 369  
 descarga del búfer 387  
 descarga del búfer de salida 393  
 desplazamiento de bits, operadores 113  
 destructor predeterminado 215  
 destructores de objetos 210  
 diferencia simétrica 442  
 dinámico 184  
 dinámico, enlace 302  
 directiva al preprocesador 27  
 distribución del código 181  
 división en potencia de dos 113  
 división por cero 369  
 Divisiones y tipos de datos 66  
 do 43  
 do-while 127  
 doble barra (//) 26  
 dos variables con el mismo nombre 143  
 double 43, 50, 61  
 downcasting 300  
 dynamic\_cast 43, 357  
 dynamic\_cast, operador 357

## E

ejemplo con funciones genéricas 173  
 ejemplo con multiset 442  
 ejemplo de eliminación de elementos con objeto fun 447  
 ejemplo de ordenación con un objeto función 446  
 ejercicios prácticos (respuestas) 453  
 elección de la función adecuada 179  
 elección del tipo entero 54  
 else 43  
 encabezado typeinfo 361  
 encapsulamiento 206  
 enlace de archivos compilados 31  
 enlace dinámico 302  
 enlace dinámico o tardío 302

enlace estático 302  
enlazado del código objeto 19  
enlazado externo 185  
entero a entero 68  
entero a real 68  
entrada de cadenas de texto 85  
entrada numérica combinada con entrada de cadenas 87  
entrada y salida 385  
entrada y salida con archivos 402  
entrada/salida basado en búfer 387  
entrada/salida con acceso binario 406  
enum 43  
enumeración 107  
equal(), algoritmo 429  
equal\_range(), algoritmo 428  
error pasa al sistema operativo 370  
error por desborde de la pila 369  
errores con punteros 369  
errores de límites de matrices 369  
errores de sintaxis 25  
errores de un programa 368  
errores en conversiones 369  
errores en un bucle de proceso 369  
errores por división por cero 369  
errores por portabilidad 369  
escritura de archivos de texto 404  
espacios de nombres 194  
espacios de nombres anidados 195  
espacios de nombres anónimos 199  
especialización de funciones genéricas 177  
especializaciones 353  
especializaciones con más de un argumento 354  
especializaciones parciales 354  
estado de entrada/salida 416  
estática 188  
estática externa 188  
estático 95, 184  
estructura try, catch y throw 370  
estructuras 89  
estructuras, vistas como clases incompletas 204  
evolución de la pila 186  
excepción C++ 370  
excepciones 370  
excepciones no detectadas 380  
excepciones no esperadas 380  
excepciones previstas 375  
excepciones y clases 379

excepciones y herencias 379  
exception, clase 361  
exceptions() 417  
exit(), función 370  
expansión de cadenas dentro de macros 29  
explicit 43, 244  
export, palabra clave 348  
expresiones 120  
expresiones lógicas 116  
expresiones relacionales 112  
extensibilidad de la biblioteca STL 449  
extern 43  
extern const 192  
externa 188

## F

false 43  
falta de memoria 368  
fill() 398  
fill(), fill(n), algoritmos 429  
find(), find\_if(), algoritmos 428  
find\_end(), algoritmo 428  
find\_first\_of(), algoritmo 428  
fixed 397  
flags() 398  
float 43, 50  
floatfield 397  
flujos de datos 386  
flujos de datos binarios 387  
flujos de datos de C++ 388  
flujos de texto 387  
flush() 413  
flushing 387  
fmtflags 397  
for 43, 119  
for\_each(), algoritmo 429  
formateo de entrada/salida 396  
formato decimal 56  
formato hexadecimal 56  
formato octal 56  
formato texto predeterminado 396  
forzar la terminación de la ejecución 20  
friend 43, 240  
friend, clases 336  
friend y templates 354  
front\_insert\_iterator, iterador 437  
fstream 402  
función 139  
función de terminación 380

- función en base a un tipo genérico 172, 343, 459
- función main() 26
- función miembro const 227
- función puts(): Impresión en consola 88
- función virtual 303
- función virtual que deja de serlo 308
- funciones 38
- funciones de conversión 245
- funciones definidas por el usuario 40
- funciones especializadas 175
- funciones friend 240
- funciones genéricas (templates) 172
- funciones genéricas sobrecargadas 174
- funciones implícitas 251
- funciones inline 164
- funciones inline o macros 166
- funciones miembro constantes 226
- funciones miembro de la clase derivada 290
- funciones miembro implícitas 214
- funciones miembro virtuales 301
- funciones miembros estáticas 255
- funciones no virtuales 302
- funciones sobrecargadas (polimorfismo) 161
- funciones virtuales 301
- functors, objetos función 421

## G

- gcount() 411
- generate(), generate(n), algoritmos 429
- genericidad de clases 343
- gestión básica de los errores 369
- gestión de excepciones 370
- get() 407
- getline() 86, 412
- gets() 88
- goto 43
- greater<int> 446
- grupo de palabras clave C++ 42

## H

- heap 95, 184
- herencia, referencias y punteros 299
- herencia de clases 276
- herencia, lo que no se hereda de la clase base 284
- herencia múltiple 294
- herencia o contención 277
- herencia privada 328

- herencia privada múltiple 329
- herencia privada o contención 333
- herencia protected 334
- herencia y conversiones implícitas 335
- herencia y funciones virtuales 301
- hex 397
- hexadecimal 56
- hora 232

## I

- if 43, 128
- if-else 130
- ifstream 402
- igual a 112
- igualdad de nombres de variables 143
- implementación de la clase derivada 286
- implementación de la función 38, 140
- implementación de los métodos de la clase Empleado 217
- implementación de los métodos miembro 207
- includes(), algoritmo 431
- inclusión de archivos 29
- indirección 92, 94
- inicialización de una estructura 89
- inicializar correctamente los objetos 211
- inline 43, 164
- inline, constructor 322
- inner\_product(), algoritmo 428
- inserción de objetos en el flujo de datos 390
- insert\_iterator, iterador 437
- instancia explícita de la función genérica 176
- instancia implícita de la función genérica 176
- instancia y clase, diferencia entre 209
- instanciar un objeto de la clase 209
- instancias, contador 253
- instancias de función genérica 176
- instancias de un objeto 220
- instancias de una clase 208
- instancias explícitas 353
- instancias implícitas 353
- instrucciones al preprocesador 26
- int 43, 49
- interfaz pública del objeto contenedor 322
- internal 397
- intersección de dos contenedores 442
- invocación a la función 140

invocar al constructor 212  
ios 387  
ios::app 403  
ios::ate 403  
ios::badbit 416  
ios::beg 414  
ios::binary 403  
ios::cur 414  
ios::end 414  
ios::eofbit 416  
ios::failbit 416  
ios::goodbit 416  
ios::in 403  
ios::out 403  
ios::trunc 403  
iostream 34, 387, 389  
is\_open() 403  
istream 387, 395  
istream\_iterator 435  
istreambuf\_iterator, iterador 437  
iter\_swap(), algoritmo 429  
iterador 427  
iterador bidireccional 434  
iterador de acceso aleatorio 434  
iterador de acceso random 435  
iterador de entrada 433  
iterador de navegación hacia adelante 434  
iterador de salida 434  
iteradores 421, 432, 469  
iteradores predefinidos 437  
iteradores, tipos 433

## J

Java 18

## L

lectura de entrada por teclado 88  
lectura y escritura de archivos de texto 404  
left 397  
lexicographical\_compare(), algoritmo 429  
liberación adecuada de la memoria 253  
liberación de la memoria 99  
liberación de la memoria 364  
Libre 95, 184  
LIFO (last input first output) 377  
LIFO, en pilas 425  
limitaciones de la sobrecarga de operadores 238  
límites de matrices 369

LIMITS.H 53  
línea de comandos 20  
lista de códigos de escape C++ 59  
lista de especificaciones de excepciones esperadas 380  
lista de inicialización 286  
lista de precedencia y asociatividad de los operad 114  
lista doblemente enlazada 424  
lista enlazada, contenedor STL 424  
listado utilizando un iterador ostream\_iterator 440  
literales de carácter 63  
literales enteras 60  
literales reales 61  
long 43, 49  
long double 50  
lower\_bound(), algoritmo 429  
Lvalue 108

## LL

llamada a la función 140  
llamada a una función virtual 305  
llamar mediante un puntero 301

## M

macro 53  
macro assert 368  
macro del procesador 53  
macros 166  
main(), función 26  
make\_heap(), algoritmo 430  
manipulador flush 394  
manipuladores de entrada/salida 399  
manipuladores personalizados 401  
manipulando la matriz 150  
mapas, contenedores STL 426  
math 39  
matrices de estructura 90  
matrices de objetos 224  
matrices de variables 76  
matrices multidimensionales 77  
matriz de una dimensión 77  
max(), algoritmo 432  
max\_element(), algoritmo 428  
mayor o igual a 113  
mayor que 113  
mecanismo de herencia 277  
mecanismo LIFO 377

memoria dinámica y auto\_ptr 363  
 memoria montón 95, 184, 364  
 menor o igual a 112  
 menor que 112  
 merge(), inplace\_merge(), algoritmos 430  
 método virtual puro 307  
 métodos inline 208  
 métodos que no tienen valor de retorno 158  
 métodos y el calificador virtual 306  
 Microsoft Visual C++ 2008 Express Edition 20  
 miembros constantes 226  
 miembros de clase static 255  
 miembros públicos del objeto contenido 322  
 miembros públicos y privados 207  
 min(), algoritmo 432  
 min\_element(), algoritmo 429  
 minúsculas y mayúsculas 37  
 mismatch(), algoritmo 429  
 modificación de los formatos en un flujo de datos 399  
 modo de usar un método de clase 209  
 modo decimal 61  
 modo exponencial 61  
 modos de creación de instancias de un objeto 220  
 modos de inicialización 235  
 módulos 31  
 multimap, ejemplo 444  
 múltiple, herencia 329  
 múltiples bloques try 376  
 multiplicación en potencia de dos 113  
 multiset 442  
 multiset, contenedor STL 426  
 multiset, ejemplo 442  
 mutable 43, 193

## N

Namespace 31  
 namespace 44  
 namespaces 194  
 navegación inversa en vector 424  
 NDEBUG 368  
 new 44, 94  
 next\_permutation(), algoritmo 430  
 no igual a 113  
 nombres de variables 143  
 NOT lógico 117  
 NOT lógico (!) 118

notación E 396  
 notepad 21  
 nth\_element(), algoritmo 431  
 nueva línea 59  
 NULL, puntero 106

## O

objeto contenedor, interfaz pública 322  
 objeto función (functor) 445  
 objeto String 81  
 objetos dependen de las funciones 139  
 objetos en el flujo de datos 390  
 objetos función predefinidos 448  
 objetos funciones (functors) 421  
 objetos: instancias de una clase 208  
 oct 397  
 octal 56  
 ocultamiento del mecanismo de sobrecarga de funcio 306  
 ofstream 402  
 open() 402  
 OpenWatcom 19  
 operaciones aritméticas 64  
 operador -> 103  
 operador "." 89  
 operador = 37  
 operador = en clases derivadas 293  
 operador AND lógico (&&) 117  
 operador coma 121  
 operador condicional (?) 132  
 operador const\_cast 362  
 operador de ámbito (o campo) 194  
 operador de asignación 215  
 operador de bit & 113  
 operador de bit | 113  
 operador de dirección 215  
 operador de extracción >> 34  
 operador de indirección \* 105  
 operador de inserción (<<) 34  
 operador de preprocesamiento # 29  
 operador de resolución de nombres (::) 194  
 operador decremento (-- ) 122  
 operador delete 99  
 operador dynamic\_cast 357  
 operador incremento (++) 122  
 operador new 184  
 operador new: reserva de memoria 94  
 operador NOT lógico (!) 118  
 operador OR lógico (||) 117

operador punto 105  
operador static\_cast 363  
operador typeid 357  
operador typeinfo 362  
operadores cast 362  
operadores de asignación compuestos 122  
operadores de bit 113  
operadores de extracción 394  
operadores en la entrada/salida 390  
operadores no sobrecargables 238  
operadores que se pueden sobrecargar 238  
operadores relacionales 112  
operadores sobrecargables sólo en funciones  
miembr 238  
operadores soportados por tipo de iterador  
434  
operator 44  
operator != 434  
operator double() 248  
operator int() 245  
operator\* 434  
operator+ 435  
operator++ 434  
operator+= 435  
operator- 435  
operator-- 434  
operator-= 435  
operator-> 434  
operator< 435  
operator<= 435  
operator= 261, 434  
operator== 350, 434  
operator> 435  
operator>= 435  
operator[] 435  
OR lógico 117  
OR lógico (||) 117  
ordenación de una lista 440  
origen de C++ 18  
ostream 387  
ostream\_iterator e istream\_iterator 435  
otras formas de get() 411  
otros calificadores de clase de  
almacenamiento 192

## P

pair, clase 426  
palabra clave export 348  
palabra clave protected 293

palabra clave static 252  
palabra clave template 173, 346  
palabras clave de C++ 42  
paquete de clases de entrada/salida 402  
partial\_sort(), partial\_sort\_copy(),  
algoritmos 431  
paréntesis y precedencia 65  
pares de valores en mapas 426  
partial\_sum(), algoritmo 428  
particularidades de la clase derivada 285  
partition(), stable\_partition(), algoritmos 431  
pase de argumentos a una función 144  
pase por referencia 145  
pase por valor 145  
pasos del preprocesado 29  
pausa para ver ventana de consola 45  
peek() 412  
pérdida de precisión 63  
pila 95, 184  
pila, contenedor STL 425  
pila LIFO 186  
plantillas de funciones (templates) 172  
polimorfismo 161, 300  
pop\_heap(), algoritmo 431  
portabilidad 369  
postfijo (i--) 122  
potencia de dos 113  
prácticas del capítulo 1 46  
precedencia de operadores 65, 114  
precisión en tipos reales 62  
precision() 399  
predicado binario 445  
predicado unario 445  
prefijo (++) 122  
preprocesado 29  
preprocesador 26  
prev\_permutation(), algoritmo 430  
primeros pasos en C++ 25  
private 44  
problemas de las funciones implícitas 251  
procesamiento y excepciones 376  
proceso byte a byte 407  
proceso en bloques de bytes 409  
producto binario de bits (AND) 113  
programación genérica 172, 343  
promoción entera 71  
promoción por combinación de tipos 71  
promociones 163  
protección de parámetros con const 150  
protected 44



protected, en clases base 293  
protected, en contención de clases 333  
prototipo de la función 38, 140  
prototipo de template 173  
prototipo de una función 141  
proyecto vacío 23  
prueba de la clase derivada 294  
pseudo puntero 364  
public 44  
puntero a char con un literal 102  
puntero a objetos 222  
puntero constante 227  
puntero nulo 106  
puntero this 222, 462  
punteros a cadenas 100  
punteros a memoria montón 293  
punteros y estructuras (Operador ->) 103  
punto y coma 34  
puntos suspensivos 374  
push\_heap(), algoritmo 431  
put() 407  
put(), método en ostream 392  
puts() 88

## Q

queue, contenedor STL 425

## R

random, iterador 435  
random\_shuffle(), algoritmo 431  
razones por las que una función virtual se trata c 308  
rbegin() 437  
read() 409  
real a entero 68  
real a real: 68  
recortando una cadena 84  
recursividad 163  
referencias 106, 145  
referencias a estructuras 157  
región declarativa 198  
register 44, 187  
registro 187  
reinterpret\_cast 44, 363  
relación "es un" 281  
relación reversible 300  
relación "tiene un" 282, 328  
relacionales, operadores 112  
relanzado de una excepción 375

remove(), algoritmo 430  
remove\_copy(), algoritmo 430  
remove\_copy\_if(), algoritmo 430  
remove\_if() 446  
remove\_if(), algoritmo 430  
rend() 437  
replace(), algoritmo 430  
representación predeterminada de los valores 397  
requerimientos de STL 450  
reserva de memoria 94  
resolución de nombres (::) 194  
resolución de sobrecarga 179  
respuesta a los ejercicios prácticos 453  
resumen de las características de las matrices o a 81  
resumen sobre funciones virtuales 308  
retorno de una función 26  
retorno de valores de la función 141  
return 44  
reutilización del código 343  
reverse(), reverse\_copy(), algoritmos 430  
reverse\_iterator, iterador 437  
revisión de la clase Empleado 261  
revisión del uso de clases 215  
right 397  
rotate(), rotate\_copy(), algoritmos 430  
RTTI, runtime type information 357  
runtime type information 357

## S

salida de datos 385  
scientific 397  
search(), algoritmo 429  
search(n), algoritmo 429  
secuencia de búsqueda de una función 193  
secuencia de procesamiento y excepciones 376  
seekg() 414  
sentencia break 134  
sentencia continue 135  
sentencia de salida de datos 26  
sentencia Do-While 124  
sentencia for 119  
sentencia If 128  
sentencia If-else 130, 131  
sentencia que produce salida de datos 32, 34  
sentencia switch 133  
sentencia While 123

sentencias 26  
sentencias condicionales y de bifurcación 128  
sentencias de escape 59  
sentencias declarativas 111  
sentencias para el preprocesador 27  
set, contenedor STL 426  
set\_intersection(), algoritmo 431  
set\_symmetric\_difference(), algoritmo 431  
set\_terminate() 380  
set\_union(), algoritmo 431  
setf() 398  
setstate() 417  
short 44, 48  
showbase 397  
showpoint 397  
showpos 397  
signed 44  
significado de las palabras clave en una línea 42  
símbolo del sistema 20  
Simula67 18  
sintaxis que se utiliza para clase derivada 277  
sizeof 44, 83  
sizeof en matrices 81  
sizeof() 51  
skipws 397  
sobrecarga de operadores 231  
sobrecarga de operadores de extracción 394  
sobrecarga de operadores en la entrada/salida 390  
sobrecarga del operador () 445  
sobrecarga del operador [] 318  
sobrecarga en la práctica 232  
sobrecarga y métodos virtuales 306  
sobrecargada de funciones 161  
sort(), stable\_sort(), algoritmos 431  
sort\_heap(), algoritmo 431  
stack 95, 184  
stack, contenedor STL 425  
Standard Template Library 421  
static 44, 188  
static, dato miembro 252  
static, funciones miembro 255  
static\_cast 44, 363  
std::cout << 21  
STDDEF.H 52  
STL, biblioteca 421, 468  
stream 386

streambuf 387, 389  
string como contenedor STL 424  
strlen() 83  
strnlen() 85  
struct 44, 89  
subclase 276  
suma binaria de bits (OR) 113  
superclase 276  
swap(), algoritmo 432  
switch 44, 133  
switch con default 134

## T

tabla direcciones de funciones 305  
tablas de funciones virtuales 305  
tamaño tiene un objeto de una clase derivada 286  
tardío, enlace dinámico o 302  
tarea de detección de un error 368  
template 44, 172, 346  
templates como clases 352  
templates comparados con typedef 343  
templates con otros objetos 350  
templates y clases friend 354  
terminate(), función 380  
texto, flujos de 387  
this 44, 222, 462  
throw 44, 371  
tiempos ociosos en el procesador 387  
"tiene un", relación 282  
time.h 125  
tipo Bool 72  
tipo de las constantes enteras 57  
tipos de flujos de datos 387  
tipos de iteradores 433  
tipos derivados 76  
tipos enteros 50  
tipos fundamentales 48  
tipos reales 60  
transform(), algoritmo 430  
tratamiento de excepciones 370  
true 44  
try 44, 373  
typedef 44, 126  
typedef comparado con templates 343  
typeid 44  
typeid, operador 357  
typeid, operador 362  
typename 44

## U

- último en entrar, primero en salir, LIFO 377
- unexpected() 382
- union 44, 91
- unión de dos contenedores 442
- uniones 91
- unique(), unique\_copy(), algoritmo 430
- unitbuf 397
- unsetf() 398
- unsigned 44
- upcasting 300
- upper\_bound(), algoritmo 429
- uppercase 397
- using 44
- using, declaración 198
- using, modificación acceso heredado 334
- using namespace, directiva 198
- using namespace std; 40
- uso de la clase 209
- uso de la clase, archivo cap0701.cpp 219
- uso de la clase genérica 349
- uso de la clase pair 444
- uso de la recursividad 164
- uso del destructor 213
- usos del constructor 212
- utilidad de definir una clase especializada 312

## V

- valor de retorno, métodos sin 158
- valor de retorno y referencia a tipo 157
- variables 36
- variables automáticas 185
- variables estáticas 188
- variables globales 142
- variables locales (estáticas) 143
- variables locales (no estáticas) 143
- variables registro 187
- variables utilizables en una función 142
- vector 77
- vector, contenedor STL 423
- ventana de la consola 45
- virtual 44
- virtual puro, método 307
- virtual: que no puede ser 306
- virtuales, funciones 301
- Visual C++ 19
- void 44, 50
- volatile 44, 192

## W

- wcerr 388
- wcin 388
- wclog 388
- wcout 388
- wchar\_t 42, 45, 60, 388
- while 45
- write() 392
- write() , método en ostream 392
- www.microsoft.com 22