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

Índice general

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 Strouptup 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 Strouptup 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.**

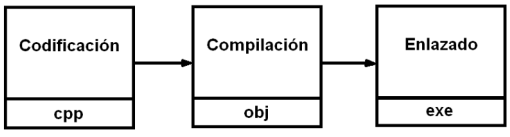
Java y C# se pueden considerar dos lenguajes hijos de C++, aunque difieran en las funciones disponibles la sintaxis es prácticamente la misma. El que conozca cualquiera de estos tres lenguajes podrá dominar rápidamente los otros ya que además de la sintaxis comparten el mismo enfoque para el modelo de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

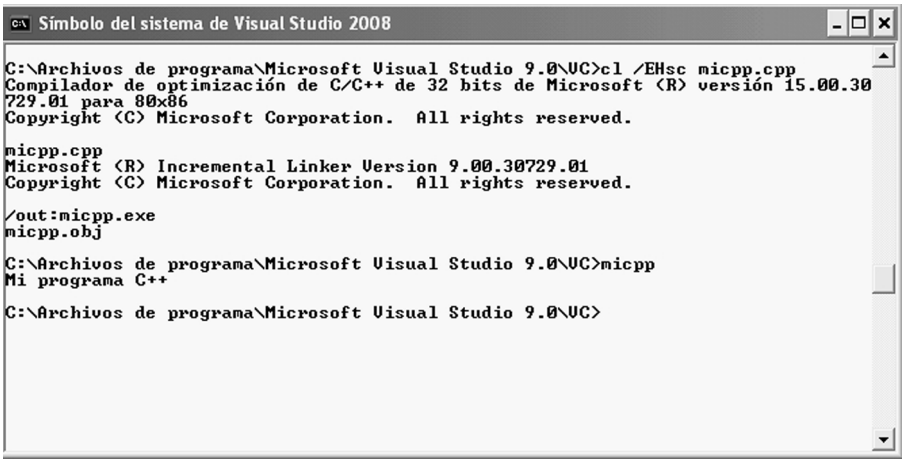
2. En el símbolo del sistema, introduciremos notepad micpp.cpp y luego pulsamos la tecla Entrar. Como este archivo no existe deberemos responder Sí cuando se nos pida si se crea un archivo nuevo.
3. Este comando hará que se abra el Bloc de notas y ahí escribiremos estas líneas de código, que por ahora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



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
char	*1 byte*	*Normalmente un ASCII. Se describe entre apóstrofos. 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:* `char a; // variable a sin asignación` `char b = "b";`
wchar_t	*2 bytes*	*Similar a char con mayor capacidad. Siempre con signo: capacidad de 0 a 65.535. Ejemplos:* `wchar_t = L'U';`
short	*2 bytes*	*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:* `short n; // n ocupa 2 bytes y tiene signo` `short m = 3000; // definición y asignación` `// en una sentencia`
int	*4 bytes*	*Es un tipo entero. Puede ser con signo o sin signo (signed o unsigned). Por defecto se asume con signo. Es = o > 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*

Tipo	Tamaño	Característica (continuación)
		2.147.483.647.
		Ejemplos:
		``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```
_intn	*var.*	*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.*
		Ejemplos:
		``_int16 n; // n ocupa 2 bytes y tiene signo```
		``_int64 m; // variable de 64 bits y tiene // signo```
long	*4 bytes*	*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).*
		Rango de valores sin signo desde 0 y 4.294.967.295.
		Rango de valores con signo desde 2.147.483.648 hasta 2.147.483.647.
		Ejemplos:
		``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```
float	*4 bytes*	*Es de tipo flotante. Es el de menor precisión dentro de los tipos flotantes. Rango de valores desde 1,38E-38 a 3,4E+38.*
		Ejemplo:
		``float q;```
		``float x = 1E+2f; // sufijo f o F indica // el tipo float```
double	*8 bytes*	*Es de tipo flotante. Es de precisión mayor que el tipo float. Rango de valores desde 2,2E-308 a 1,8E+308.*

Tipo	Tamaño	Característica (continuación)
		Ejemplo: `double r;`
long double	*8 bytes*	*Es de tipo flotante. Es de precisión mayor que el tipo double. Rango de valores desde 2,2E-308 a 1,8E+308. Ejemplo:* `long double r;`
bool	*N/A*	*True o False. False es 0, True diferente a 0. Ejemplo:* `bool r;`
void	-	*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.* *Ejemplo:* `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`

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.

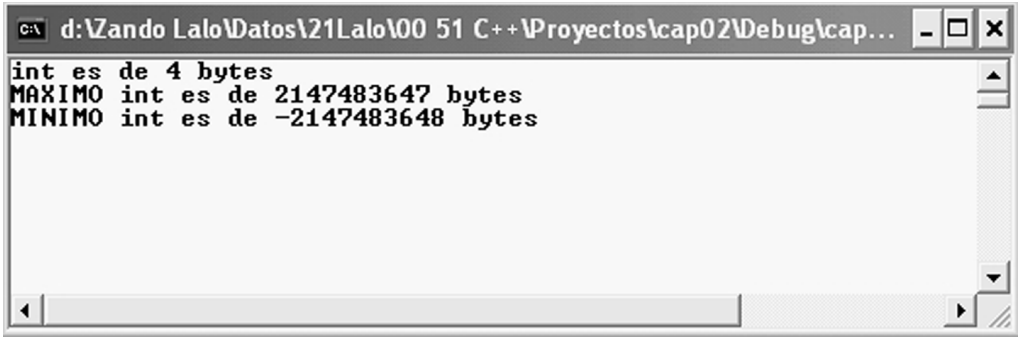
Lo mismo sucede con los otros tipos enteros (char, short y long), adaptando el rango de valores a la capacidad de cada tipo (normalmente, 1, 2 y 4 bytes respectivamente).

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:



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

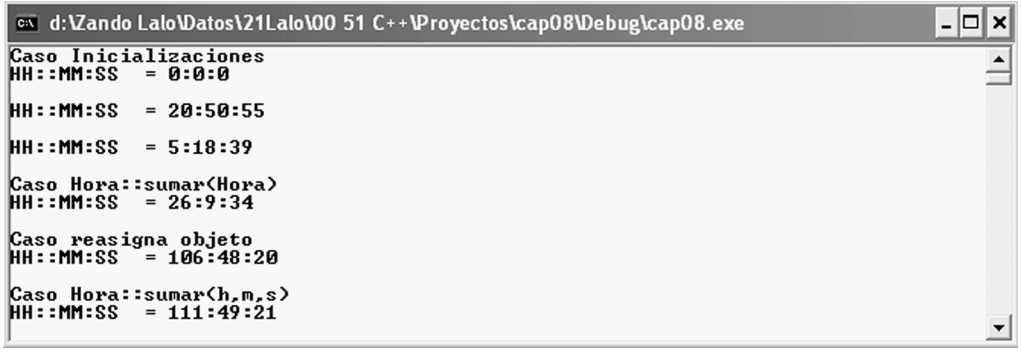
En la ejecución del código podemos comprobar que los dos métodos sumar() funcionan correctamente. El método privado ajuste60() hace que los valores de minutos y segundos no excedan ese límite, pasando el excedente al nivel siguiente mediante la correspondiente conversión de unidades. Si quisiéramos añadir 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:



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 partir, 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 emp1;
Gerente ger1;
```

Y los métodos se utilizan de la siguiente manera:

```
emp1.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 (csdtlib.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 x 1= 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

A

B

C

CH

D

E

F

G

H

I

J

L

LL

M

N

O

P

Q

R

S

T

U

V

W