Capitulo 14 Clases Primera Parte

Definiciones

Aunque te parezca mentira, hasta ahora no hemos visto casi nada de C++. La mayor parte de lo incluido hasta el momento forma parte de C, salvo muy pocas excepciones.

Ahora vamos a entrar a fondo en lo que constituye la mayor diferencia entre C y C++: las clases. Así que prepárate para cambiar la mentalidad, y el enfoque de la programación tal como lo hemos visto hasta ahora.

Recodemos que en nuestro entorno todo se puede ver como un objeto , con funciones y propiedades; como el caso de un automovil , tiene propiedades como el color, tipo de combutisble, capacidad, etc. y funciones como arrancarse, detenerse, acelerar. etc. en definitiva se puede ver casi todo como un objeto.

En las próximas paginas iremos introduciendo nuevos conceptos que normalmente se asocian a la programación orientada a objetos, como son: objeto, mensaje, método, clase, herencia, interfaz, etc.

POO

Siglas de "Programación Orientada a Objetos". En inglés se pone al revés "OOP". La idea básica de este tipo de programación es agrupar los datos y los procedimientos para manejarlos en una única entidad: el objeto. Un programa es un objeto, que a su vez está formado de objetos. La idea de la programación estructurada no ha desaparecido, de hecho se refuerza y resulta más evidente, como comprorarás cuando veamos conceptos como la herencia.

Clase

Una clase se puede considerar como un patrón de datos o colección de datos, recodemos el caso de estructuras, en principio existen muchas similitudes , hasta incluso en la manera que se definen. En C++ una clase tiene como objetivo crear ese nuevo tipo de datos que hasta puede contener funciones como mienbros, y el conjunto de miembros se pueden manejar como si fuese un elemento único.

Es importante notar que las clases no se deben Inicializa .. ya que no son variables sinó patrones de datos. Algo también importante de destacar que una diferencia fundamental con estructuras, es ellas cada miembro podía se accedido sin limitaciones , podríamos decir que los miembros eran publicos por defecto, en clases esto no es así, existen especificadores o modificadores de acceso que definen cuales de sus miembros se pueden acceder, en principio **TODOS** los miembros de una clase son **PRIVADOS**, salvo indicación contraria, pero ,mas adelante daremos mas detalles de esto. A los miembros de las clases se los puede llamar como **MIEMBROS** ó PROPIEDADES.

Objeto

Un objeto es una variable creada a partir de una clase, recordemos para el caso de la estructura , esto es similar creábamos una instancia a partir de una estructura, el proceso de crear un objeto (colección de variables y funciones) partir de una clase se llama instanciar. Es importante distinguir entre objetos y clases, la clase es simplemente una declaración, no tiene asociado ningún objeto (colección de variables y funciones). Para dar un ejemplo usemos una analogía, tomemos como que la clase sería como el tipo de variable (int, float, etc)

int x => int sería equivalente a la clase y x sería el objeto !! (ESTO ES SOLO UNA ANALOGÍA)

Un objeto es una unidad que engloba en sí mismo datos y procedimientos necesarios para el tratamiento de esos datos. Hasta ahora habíamos hecho programas en los que los datos y las funciones estaban perfectamente separadas, cuando se programa con objetos esto no es así, cada objeto contiene datos y funciones. Y un programa se construye como un conjunto de objetos, o incluso como un único objeto. Es importante notar que un objeto se puede inicializar a diferencia de una clase. Recordemos que los miembros de un objeto (o mas genéricamente de una una clase)se puede llamar o PROPIEDADES ó MIEMBROS.

Ejemplo de Sintaxis:

```
class <identificador de clase> {
   <lista de miembros>    // miembros o propiedades separados por ;
} [<lista de objetos>]; //vemos que existe una similitudo con estrucutras
```

Un ejemplo:

```
class Televisor {
      bool m_bEncedido;                   //propiedades de la clase
      unsigned int m_uiCanal;
      unsigned int m_uiVolumen;
                  }
 tv14pulgadas ; tv17pulgadas ; // existen 2 objetos para esta clase=>Instanciar


// estamos instanciado un objeto tv14pulgadas a partir de una clase teelvisor

// estamos instanciado un objeto tv17pulgadas a partir de una clase teelvisor
```

Método

Se trata de otro concepto de POO, en C++ un método no es otra cosa que una función procedimiento perteneciente a un objeto, hasta ahora solamente hemos incorporado como miembros de la clase variables...bueno también pueden ser funciones.

Veamos un ejemplo.. supongamos que se quiere modificar la propiedad Volumen deberíamos estar seguro que el televisor esté encendido, esto es lo normal, también sería algo parecido si queremos cambiar de canal, veamos como sería:

```
# include <iostream>
using namespace std;
class Televisor
{
      bool m_bEncedido;                   //miembros o propiedades de la clase
      public:   // Todos los metodos y propiedades
```

```
                //pueden ser accedidos desde afuera
      unsigned int m_uiCanal;       // la m_ indica que es un miembro
      unsigned int m_uiVolumen;
      void Encender()  //Definicion del metodo
      {
      m_bEncedido=true;
      }
      void Apagar (); //Protipo de Apagar
      bool ModificarVolumen( unsigned int  uiVolumen);//Prototipo
      bool ModificarCanal ( unsigned int  uiCanal)//Definicion del metodo
            {if (m_bEncedido)
                  {m_uiCanal=uiCanal;
                   return true;
                  }
            else return false;
            }
} ;
//Como solo tenía el prototipo defino el metodo
bool Televisor::ModificarVolumen ( unsigned int  uiVolumen)
            {if (m_bEncedido)
                  {m_uiVolumen=uiVolumen;
                   return true;
                  }
            else return false;
            }
// Como solo estaba el prototipo defino el metodo
void Televisor::Apagar()   //debo hacer ref. a la clase que pertenece
{m_bEncedido=false;
}

int main()
{Televisor tv14p; //Instancio el objeto tv14p con la clase Televisor
//Enciendo el televisor
tv14p.Encender();
//Modifico el volumen
tv14p.ModificarVolumen(10);
// cout << "El televisor está : "<< tv14p.m_bEncedido << endl; //ERROR !!
cout << "El volumen es :" << tv14p.m_uiVolumen << endl;
return 0;
}
```

Observaciones:

-Vemos que algunos métodos están dentro de la definición de la clase y otros afuera, conviene declarar los métodos afuera para simplificar la lectura en la clase ,salvo que sean muy simples como sería el caso de Apagar, que en este caso esta afuera.

-Vemos que cuando se definen afuera los métodos de la clases, se utiliza el operador de ambito "::", este es el que vincula el método a la clase, si esto no existiera ,parecería la definición de una función cualquiera .-

-Vemos que una línea del código esta comentada:

// cout << "El televisor está : "<< tv14p.m_bEncedido << endl; //ERROR !!

si no lo hiciéramos el compilador tiraría un error, ya que la propiedad o miembro m_bEncendido es de carácter privado y no se puede acceder, descomente esta linea y compile para ver el error, veremos en el ejemplo siguiente como alterar una propiedad, para ello definimos un método.

Veamos otro Ejemplo algo mas completo:

```
# include <iostream>
using namespace std;
class Televisor
{
      bool m_bEncedido; //miembros o propiedades de la clase, es privado
      unsigned int m_uiCanal;// la m_ indica que es un miembro
      unsigned int m_uiVolumen;

      public: //los metodos y propiedades pueden ser accedidos desde afuera
                  //son públicos
      void Encender() {m_bEncedido=true;};//Defino el metodo dentro de la clase
      void Apagar(){m_bEncedido=false;};//Defino el metodo dentro de la clase
      bool ModificarVolumen( unsigned int  uiVolumen);
      bool ModificarCanal ( unsigned int  uiCanal);
      unsigned int ConsultaVolumen ();
      unsigned int ConsultaCanal ();
} ; //Fin de definición de la clase Televisor
//Metodo de cambia de Canal,definido Fuera de Clase por eso el op. de ambito ::
bool Televisor::ModificarCanal ( unsigned int  uiCanal)
            {if (m_bEncedido)
                  {m_uiCanal=uiCanal;
                   return true;
                  }
            else return false;
            }
//Metodo Cambia el Volumen,definido Fuera de Clase por eso el op. de ambito ::
bool Televisor::ModificarVolumen ( unsigned int  uiVolumen)
            {if (m_bEncedido)
                  {m_uiVolumen=uiVolumen;
                   return true;
                  }
            else return false;
            }
//Metodo Consulta el Volumen,definido Fuera de Clase por eso el op. de ambito ::
unsigned int Televisor :: ConsultaVolumen ()
{return m_uiVolumen;}
```

```
//Metodo Consulta el Canal,,definido Fuera de Clase por eso el op. de ambito ::
unsigned int Televisor :: ConsultaCanal ()
{return m_uiCanal;}
//Comienza la Funcion main
int main()
{ char opcion; int volumen, canal;
Televisor tv14p; //Instancio el objeto tv14p con la clase Televisor
do{
cout << " Indique la acción :" << endl;
cout << " 1 para Encender" << endl;
cout << " 0 para Apagar" << endl;
cout << " 2 Para modificar el Volumen" << endl;
cout << " 3 Para modificar el Canal "<< endl;
cout << " Otra tecla para Salir "<< endl;

cin >> opcion;
switch (opcion){
      case '1': tv14p.Encender();break;//Enciendo el televisor
      case '0': tv14p.Apagar();break ;//Apago el televisor
      case '2': cout << "EL volumen esta en: "<< tv14p.ConsultaVolumen();
                  cout << "Ingrese el nuevo valor del Volumen: "<< endl;
                cin >> volumen;
                  tv14p.ModificarVolumen(volumen);break;
      case '3': cout << "EL Canal actual esta en: " << tv14p.ConsultaCanal();
                  cout << "Ingrese el nuevo valor del Canal: " << endl;
                cin >> canal;
                tv14p.ModificarCanal(canal);break;
           }} while ('0'==opcion||'1'==opcion||'2'==opcion||'3'==opcion);
           return 0;

}
```

Observaciones:

-Vemos que para alterar las propiedades privadas recurrimos a métodos , por ejemlpo Enceder , Apagar , estos alteran la propiedad privada m_bEncendido.

-Si corremos el código y consultamos el valor del canal y del volumen antes de asignar algún valor veremos que el mismo contiene BASURA , veremos mas adelante que exiten métodos o funciones para inicializar el objeto , estos son conocidos como "CONSTRUCTORES".

Mensaje

El mensaje es el modo en que se comunican los objetos entre si. En C++, un mensaje no es más que una llamada a una función de un determinado objeto. Cuando llamemos a una función de un objeto, muy a menudo diremos que estamos enviando un mensaje a ese objeto.

En este sentido, mensaje es el término adecuado cuando hablamos de programación orientada a objetos en general.

Constructores: Inicializaciones de Objetos.

Los constructores son funciones miembro especiales que sirven para inicializar un objeto de una determinada clase al mismo tiempo que se declara, vimos para el ejemplo del objeto tv14p que el volumen y el canal tenían basura al comenzar a utilizar el objeto.

Sería algo parecido a poner:

int a=20; /* defino el tipo e inicializo */

Los constructores tienen el mismo nombre que la clase, no retornan ningún valor y no pueden ser heredados. Además deben ser públicos, no tendría ningún sentido declarar un constructor como privado, ya que siempre se usan desde el exterior de la clase, ni tampoco como protegido, ya que no puede ser heredado.

Añadamos un constructor a nuestra clase pareja:

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

class pareja {
   public:
      // Constructor
      pareja(int a2, int b2);
      // Funciones miembro de la clase "pareja"
      void Lee(int &a2, int &b2);
      void Guarda(int a2, int b2);
   private:
      // Datos miembro de la clase "pareja"
      int a, b;
   public:
};

pareja::pareja(int a2, int b2) {
   a = a2;
   b = b2;
}

void pareja::Lee(int &a2, int &b2) {
   a2 = a;
   b2 = b;
}

void pareja::Guarda(int a2, int b2) {
   a = a2;
   b = b2;
}

int main() {
```

```
   pareja par1(12, 32);
   int x, y;

   par1.Lee(x, y);
   cout << "Valor de par1.a: " << x << endl;
   cout << "Valor de par1.b: " << y << endl;

   cin.get();
   return 0;
}
```

Si una clase posee constructor, será llamado siempre que se declare un objeto de esa clase, y si requiere argumentos, es obligatorio suministrarlos.

Por ejemplo, las siguientes declaraciones son ilegales:

```
pareja par1;
pareja par1();
```

La primera porque el constructor de "pareja" requiere dos parámetros, y no se suministran.

La segunda es ilegal por otro motivo más complejo. Aunque existiese un constructor sin parámetros, no se debe usar esta forma para declarar el objeto, ya que el compilador lo considera como la declaración de un prototipo de una función que devuelve un objeto de tipo "pareja" y no admite parámetros. Cuando se use un constructor sin parámetros para declarar un objeto no se deben escribir los paréntesis.

Y las siguientes declaraciones son válidas:

```
pareja par1(12,43);
pareja par2(45,34);
```

Cuando no especifiquemos un constructor para una clase, el compilador crea uno por defecto sin argumentos. Por eso en los ejemplos anteriores funcionaba correctamente. Cuando se crean objetos locales, los datos miembros no se inicializarían, contendrían la "basura" que hubiese en la memoria asignada al objeto. Si se trata de objetos globales, los datos miembros se inicializan a cero.

Para declarar objetos usando el constructor por defecto o un constructor que hayamos declarado sin parámetros no se debe usar el paréntesis:

```
pareja par2();
```

Se trata de un error frecuente cuando se empiezan a usar clases, lo correcto es declarar el objeto sin usar los paréntesis:

```
pareja par2;
```

Observaciones:

A) Se pueden definir los Constructores de 2 maneras distintas, para el caso de la clase "pareja":

Forma 1:

```
pareja::pareja(int a2, int b2) {
   a = a2;
```

```
   b = b2;
}
```

Forma 2:

Podemos sustituir el constructor por:

```
pareja::pareja(int a2, int b2) : a(a2), b(b2) {}
```

B) El constructor es Public !! (no tendría sentido otro tipo de acceso)

C) El constructor NO TIENE TIPO

D) El nombre del Constructor es IGUAL al de la CLASE

Interfaz

Las clases y por lo tanto también los objetos, tienen partes públicas y partes privadas. Algunas veces llamaremos a la parte pública de un objeto su interfaz. Se trata de la única parte del objeto que es visible para el resto de los objetos, de modo que es lo único de lo que se dispone para comunicarse con ellos.

Herencia

Veremos que es posible diseñar nuevas clases basándose en clases ya existentes. En C++ esto se llama derivación de clases, y en POO herencia. Cuando se deriva una clase de otra, normalmente se añadirán nuevos métodos y datos. Es posible que algunos de estos métodos o datos de la clase original no sean válidos, en ese caso pueden ser enmascarados en la nueva clase o simplemente eliminados. El conjunto de datos y métodos que sobreviven, es lo que se conoce como herencia. Veremos en otro momento mas en detalle este tema.

Mas ejemplo de definición de clase.

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

class pareja {
   private:
      // Datos miembro de la clase "pareja"
      int a, b;
   public:
      // Funciones miembro de la clase "pareja"
      void Lee(int &a2, int &b2);
      void Guarda(int a2, int b2) {
         a = a2;
         b = b2;
      }
};

void pareja::Lee(int &a2, int &b2) {
   a2 = a;
```

```
   b2 = b;
}

int main() {
   pareja par1;
   int x, y;

   par1.Guarda(12, 32);
   par1.Lee(x, y);
   cout << "Valor de par1.a: " << x << endl;
   cout << "Valor de par1.b: " << y << endl;

   cin.get();
   return 0;
}
```

Nuestra clase "pareja" tiene dos miembros de tipo de datos: a y b.

Y dos funciones, una para leer esos valores y otra para modificarlos.

En el caso de la función "Lee" la hemos declarado en el interior de la clase y definido fuera, observa que en el exterior de la declaración de la clase tenemos que usar la expresión:

```
void pareja::Lee(int &a2, int &b2)
```

Para que quede claro que nos referimos a la función "Lee" de la clase "pareja". Ten en cuenta que pueden existir otras clases que tengan funciones con el mismo nombre, y también que si no especificamos que estamos definiendo una función de la clase "pareja", en realidad estaremos definiendo una función corriente.

En el caso de la función "Guarda" la hemos definido en el interior de la propia clase. Esto lo haremos sólo cuando la definición sea muy simple , como ya lo aclaramos , ya que dificulta la lectura y comprensión del programa.

Además, las funciones definidas de este modo serán tratadas como "inline", y esto sólo es recomendable para funciones cortas, ya que, (como recordarás), en estas funciones se inserta el código cada vez que son llamadas.

Especificadores de acceso

Dentro de la lista de miembros, cada miembro puede tener diferentes niveles de acceso, hasa ahora hemos visto los privados y los públicos.

En nuestros ejemplo hemos usado dos de esos niveles, el privado y el público, aunque hay más.

```
class <identificador de clase> {
  public:
   <lista de miembros>
  private:
   <lista de miembros>
  protected:
   <lista de miembros>
};
```

Acceso privado, private

Los miembros privados de una clase sólo son accesibles por los propios miembros de la clase y en general por objetos de la misma clase, pero no desde funciones externas o desde funciones de clases derivadas.

Acceso público, public

Cualquier miembro público de una clase es accesible desde cualquier parte donde sea accesible el propio objeto. Veamos un Ejemplo:

```
#include <iostream>
using namespace std;
class  X {int a;};//Error deberia poner public:int a
class  Z {int b;}; //Error deberia poner public:int b
int main()
{X a1,a4;
Z a2,a5;
int a3;
a1.a=3;
a2=a1;  //Error no se pueden igualar Clases Distintas !!
a3=a1 ; //Error: No se puede convertir X a int !!
a5.b=8;
a4=a1 ; //esto es correcto
a2=a5; // esto es correcto
}
```

Observación: este mismo ejemplo es válido si en lugar de la palabra class se usa struct !

Acceso protegido, protected

Con respecto a las funciones externas, es equivalente al acceso privado, pero con respecto a las clases derivadas se comporta como público.

Cada una de éstas palabras, seguidas de ":", da comienzo a una sección, que terminará cuando se inicie la sección siguiente o cuando termine la declaración de la clase. Es posible tener varias secciones de cada tipo dentro de una clase.

Si no se especifica nada, por defecto, los miembros de una clase son privados.

Modificadores para miembros

Existen varias alternativas a la hora de definir algunos de los miembros de las clases. Esto es lo que veremos en este capítulo. Estos modificadores afectan al modo en que se genera el código de ciertas funciones y datos, o al modo en que se tratan los valores de retorno.

Métodos ó Funciones en línea (inline):

A menudo nos encontraremos con funciones miembro cuyas definiciones son muy pequeñas. En estos casos suele ser interesante declararlas como inline. Cuando hacemos eso, el código generado para la función cuando el programa se compila, se inserta en el punto donde se invoca a la función, en lugar de hacerlo en otro lugar y hacer una llamada.

Esto nos proporciona una ventaja, el código de estas funciones se ejecuta más rápidamente, ya que se evita usar la pila para pasar parámetros y se evitan las instrucciones de salto y retorno. También tiene un inconveniente: se generará el código de la función tantas veces como ésta se use, con lo que el programa ejecutable final puede ser mucho más grande.

Es por esos dos motivos por los que sólo se usan funciones inline cuando las funciones son pequeñas. Hay que elegir con cuidado qué funciones declararemos inline y cuales no, ya que el resultado puede ser muy diferente dependiendo de nuestras decisiones.

Hay dos maneras de declarar una función como inline.

La primera ya la hemos visto. Las funciones que se definen dentro de la declaración de la clase son inline implícitamente. Por ejemplo:

```
class Ejemplo {
  public:
   Ejemplo(int a = 0) : A(a) {}

  private:
   int A;
};
```

En este ejemplo hemos definido el constructor de la clase Ejemplo dentro de la propia declaración, esto hace que se considere como inline. Cada vez que declaremos un objeto de la clase Ejemplo se insertará el código correspondiente a su constructor.

Si queremos que la clase Ejemplo no tenga un constructor inline deberemos declararla y definirla así:

```
class Ejemplo {
  public:
   Ejemplo(int a = 0);

  private:
   int A;
};

Ejemplo::Ejemplo(int a) : A(a) {}
```

En este caso, cada vez que declaremos un objeto de la clase Ejemplo se hará una llamada al constructor y sólo existirá una copia del código del constructor en nuestro programa.

La otra forma de declarar funciones inline es hacerlo explícitamente, usando la palabra reservada **inline**. En el ejemplo anterior sería así:

```
class Ejemplo {
  public:
   Ejemplo(int a = 0);

  private:
```

```
   int A;
};

inline Ejemplo::Ejemplo(int a) : A(a) {}
```

Funciones miembro constantes

Esta es una propiedad que nos será muy útil en la depuración de nuestras clases. Además proporciona ciertos mecanismos necesarios para mantener la protección de los datos.

Cuando una función miembro no modifique el valor de ningún dato de la clase, podemos y debemos declararla como constante. Esto no evitará que la función intente modificar los datos del objeto; a fin de cuentas, el código de la función lo escribimos nosotros; pero generará un error durante la compilación si la función intenta modificar alguno de los datos miembro del objeto.

Por ejemplo:

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

class Ejemplo2 {
  public:
   Ejemplo2(int a = 0) : A(a) {}
   void Modifica(int a) { A = a; }
   int Lee() const { return A; }

  private:
   int A;
};

int main() {
   Ejemplo2 X(6);

   cout << X.Lee() << endl;
   X.Modifica(2);
   cout << X.Lee() << endl;

   cin.get();
   return 0;
}
```

Para experimentar, comprueba lo que pasa si cambias la definición de la función "Lee()" por estas otras:

```
int Lee() const { A++; return A; }
int Lee() const { Modifica(A+1); return A; }
int Lee() const { Modifica(3); return A; }
```

Verás que el compilador no lo permite.

Evidentemente, si somos nosotros los que escribimos el código de la función, sabemos si la función modifica o no los datos, de modo que en rigor no necesitamos saber si es o no constante, pero frecuentemente otros programadores pueden usar clases definidas por nosotros, o nosotros las definidas por otros. En ese caso es frecuente que sólo se disponga de la declaración de la clase, y el modificador "const" nos dice si cierto modifica o no los datos del objeto.

Valores de retorno constantes

Otra técnica muy útil y aconsejable en muchos casos es usar valores de retorno de las funciones constantes, en particular cuando se usen para devolver punteros miembro de la clase.

Por ejemplo, supongamos que tenemos una clase para cadenas de caracteres:

```
class cadena {
   public:
      cadena();         // Constructor por defecto
      cadena(char *c); // Constructor desde cadena c
      cadena(int n); // Constructor para cadena de n caracteres
      cadena(const cadena &);   // Constructor copia
      ~cadena();        // Destructor

      void Asignar(char *dest);
      char *Leer(char *c) {
         strcpy(c, cad);
         return c;
      }
   private:
      char *cad;        // Puntero a char: cadena de caracteres
};
```

Si te fijas en la función "Leer", verás que devuelve un puntero a la cadena que pasamos como parámetro, después de copiar el valor de cad en esa cadena. Esto es necesario para mantener la protección de cad, si nos limitáramos a devolver ese parámetro, el programa podría modificar la cadena almacenada a pesar de se cad un miembro privado:

```
      char *Leer() { return cad; }
```

Para evitar eso podemos declarar el valor de retorno de la función "Leer" como constante:

```
      const char *Leer() { return cad; }
```

De este modo, el programa que lea la cadena mediante esta función no podrá modificar ni el valor del puntero ni su contenido. Por ejemplo:

```
class cadena {
...
};
...
int main() {
   cadena Cadena1("hola");

   cout << Cadena1.Leer() << endl; // Legal
   Cadena1.Leer() = cadena2;       // Ilegal
   Cadena1.Leer()[1] = 'O';        // Ilegal
}
```

Puntero “this”

Para cada objeto declarado de una clase se mantiene una copia de sus datos, pero todos comparten la misma copia de las funciones de esa clase.

Esto ahorra memoria y hace que los programas ejecutables sean más compactos, pero plantea un problema.

Cada función de una clase puede hacer referencia a los datos de un objeto, modificarlos o leerlos, pero si sólo hay una copia de la función y varios objetos de esa clase, ¿cómo hace la función para referirse a un dato de un objeto en concreto?

La respuesta es: usando el puntero especial llamado this. Se trata de un puntero que tiene asociado cada objeto y que apunta a si mismo. Ese puntero se puede usar, y de hecho se usa, para acceder a sus miembros.

Volvamos al ejemplo de la clase pareja:

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

class pareja {
   public:
      // Constructor
      pareja(int a2, int b2);
      // Funciones miembro de la clase "pareja"
      void Lee(int &a2, int &b2);
      void Guarda(int a2, int b2);
   private:
      // Datos miembro de la clase "pareja"
      int a, b;
};
```

Para cada dato podemos referirnos de dos modos distintos, lo veremos con la función Guarda. Esta es la implementación que usamos en el capítulo 29, que es como normalmente nos referiremos a los miembros de las clases:

void pareja::Guarda(int a2, int b2) {

```
   a = a2;
   b = b2;
}
```

Veamos ahora la manera equivalente usando el puntero this:

```
void pareja::Guarda(int a2, int b2) {
   this->a = a2;
   this->b = b2;
}
```

Veamos otro ejemplo donde podemos aplicar el operador this. Se trata de la aplicación más frecuente, como veremos al implementar el constructor copia, o al sobrecargar ciertos operadores.

A veces necesitamos invocar a una función de una clase con una referencia a un objeto de la misma clase, pero las acciones a tomar serán diferentes dependiendo de si la referencia que pasamos se refiere al mismo objeto o a otro diferente, veamos cómo podemos usar el puntero this para determinar esto:

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

class clase {
  public:
   clase() {}
   void EresTu(clase& c) {
      if(&c == this) cout << "Sí, soy yo." << endl;
      else cout << "No, no soy yo." << endl;
   }
};

int main() {
   clase c1, c2;

   c1.EresTu(c2);
   c1.EresTu(c1);

   return 0;
}
```

La función "EresTu" recibe una referencia a un objeto de la clase "clase". Para saber si se trata del mismo objeto, comparamos la dirección del objeto recibido con el valor de this, si son la misma, es que se trata del mismo objeto.

```
No, no soy yo.
Sí, soy yo.
```

Este puntero nos resultará muy útil en próximos capítulos, en los que nos encontraremos situaciones en las que es imprescindible su uso.

Miembros estáticos de una clase (Static)

Ciertos miembros de una clase pueden ser declarados como static. Los miembros static tienen algunas propiedades especiales.

En el caso de los datos miembro static sólo existirá una copia que compartirán todos los objetos de la misma clase. Si consultamos el valor de ese dato desde cualquier objeto de esa clase obtendremos siempre el mismo resultado, y si lo modificamos, lo modificaremos para todos los objetos.

Por ejemplo:

```
#include <iostream>
using namespace std;
class showstatic{
public:
static int nStatic;    // Valor de nStatic es el mismo entre cada llamado de != objetos
void mostrar(void){cout << "nStatic = " << nStatic << endl;}
```

```
void cargar(int y){nStatic=y;}
};

int showstatic::nStatic=1;  //IMPORTANTE!!

int main() {
int x;
showstatic Objeto1, Objeto2;
cout<<"Ingrese un valor para la variable static de Objeto1:" <<endl;
cin>>x;
Objeto1.cargar( x );
cout<<"Mostramos la varaible en Objeto2 que no fue inicializado :" <<endl;
Objeto2.mostrar();
cout<<" Por tal motivo vemos que la variable Static <<endl;
cout<<" es comun para los dos Objetos"<<endl;
}
```

Otro Ejemplo:

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

class Numero {
  public:
   Numero(int v = 0);
   ~Numero();

   void Modifica(int v);
   int LeeValor() const { return Valor; }
   int LeeCuenta() const { return Cuenta; }
   int LeeMedia() const { return Media; }

  private:
   int Valor;
   static int Cuenta;
   static int Suma;
   static int Media;

   void CalculaMedia();
};

Numero::Numero(int v) : Valor(v) {
   Cuenta++;
   Suma += Valor;
   CalculaMedia();
}

Numero::~Numero() {
   Cuenta--;
   Suma -= Valor;
   CalculaMedia();
}

void Numero::Modifica(int v) {
   Suma -= Valor;
   Valor = v;
   Suma += Valor;
   CalculaMedia();
}

// Definición e inicialización de miembros estáticos
```

```
int Numero::Cuenta = 0;     //IMPORTANTE!!
int Numero::Suma = 0;       //IMPORTANTE!!
int Numero::Media = 0;      //IMPORTANTE!!

void Numero::CalculaMedia() {
   if(Cuenta > 0) Media = Suma/Cuenta;
   else Media = 0;
}

int main() {
   Numero A(6), B(3), C(9), D(18), E(3);
   Numero *X;

   cout << "INICIAL" << endl;
   cout << "Cuenta: " << A.LeeCuenta() << endl;
   cout << "Media:  " << A.LeeMedia() << endl;

   B.Modifica(11);
   cout << "Modificamos B=11" << endl;
   cout << "Cuenta: " << B.LeeCuenta() << endl;
   cout << "Media:  " << B.LeeMedia() << endl;

   X = new Numero(548);
   cout << "Nuevo elemento dinámico de valor 548" << endl;
   cout << "Cuenta: " << X->LeeCuenta() << endl;
   cout << "Media:  " << X->LeeMedia() << endl;

   delete X;
   cout << "Borramos el elemento dinámico" << endl;
   cout << "Cuenta: " << D.LeeCuenta() << endl;
   cout << "Media:  " << D.LeeMedia() << endl;

   cin.get();
   return 0;
}
```

Observar que es necesario declarar e inicializar los miembros static de la clase, esto es por dos motivos. El primero es que los miembros static deben existir aunque no exista ningún objeto de la clase, declarar la clase no crea los datos miembro estáticos, **es necesario hacerlo explícitamente**. El segundo es porque no lo hiciéramos, al declarar objetos de esa clase los valores de los miembros estáticos estarían indefinidos, y los resultados no serían los esperados.

En el caso de la funciones miembro static la utilidad es menos evidente. Estas funciones no pueden acceder a los miembros de los objetos, sólo pueden acceder a los datos miembro de la clase que sean static. Esto significa que no tienen puntero this, y además suelen ser usadas con su nombre completo, incluyendo el nombre de la clase y el operador de ámbito (::).

Por ejemplo:

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

class Numero {
  public:
   Numero(int v = 0);

   void Modifica(int v) { Valor = v; }
   int LeeValor() const { return Valor; }
   int LeeDeclaraciones() const { return ObjetosDeclarados; }
```

```
   static void Reset() { ObjetosDeclarados = 0; }

  private:
   int Valor;
   static int ObjetosDeclarados;
};

Numero::Numero(int v) : Valor(v) {
   ObjetosDeclarados++;
}

int Numero::ObjetosDeclarados = 0;

int main() {
   Numero A(6), B(3), C(9), D(18), E(3);
   Numero *X;

   cout << "INICIAL" << endl;
   cout << "Objetos de la clase Numeros: "
        << A.LeeDeclaraciones() << endl;

   Numero::Reset();
   cout << "RESET" << endl;
   cout << "Objetos de la clase Numeros: "
        << A.LeeDeclaraciones() << endl;

   X = new Numero(548);
   cout << "Cuenta de objetos dinámicos declarados" << endl;
   cout << "Objetos de la clase Numeros: "
        << A.LeeDeclaraciones() << endl;

   delete X;
   X = new Numero(8);
   cout << "Cuenta de objetos dinámicos declarados" << endl;
   cout << "Objetos de la clase Numeros: "
        << A.LeeDeclaraciones() << endl;

   delete X;
   cin.get();
   return 0;
}
```

Observa cómo hemos llamado a la función Reset con su nombre completo. Aunque podríamos haber usado "A.Reset()", es más lógico usar el nombre completo, ya que la función puede ser invocada aunque no exista ningún objeto de la clase.

Sobrecarga de constructores:

Además, también pueden definirse varios constructores para cada clase, es decir, la función constructor puede sobrecargarse. La única limitación es que no pueden declararse varios constructores con el mismo número y el mismo tipo de argumentos.

Por ejemplo, añadiremos un constructor adicional a la clase "pareja" que simule el constructor por defecto:

```
class pareja {
   public:
      // Constructor
```

```
      pareja(int a2, int b2) : a(a2), b(b2) {}
      pareja() : a(0), b(0) {}
      // Funciones miembro de la clase "pareja"
      void Lee(int &a2, int &b2);
      void Guarda(int a2, int b2);
   private:
      // Datos miembro de la clase "pareja"
      int a, b;
   public:
};
```

De este modo podemos declarar objetos de la clase pareja especificando los dos argumentos o ninguno de ellos, en este último caso se inicializarán los datos miembros con ceros.

Constructores con argumentos por defecto

También pueden asignarse valores por defecto a los argumentos del constructor, de este modo reduciremos el número de constructores necesarios.

Para resolver el ejemplo anterior sin sobrecargar el constructor suministraremos valores por defecto nulos a ambos parámetros:

```
class pareja {
   public:
      // Constructor
      pareja(int a2=0, int b2=0) : a(a2), b(b2) {}
      // Funciones miembro de la clase "pareja"
      void Lee(int &a2, int &b2);
      void Guarda(int a2, int b2);
   private:
      // Datos miembro de la clase "pareja"
      int a, b;
   public:
};
```

Asignación de objetos

Probablemente ya lo imaginas, pero la asignación de objetos también está permitida. Y además funciona como se supone que debe hacerlo, asignando los valores de los datos miembros.

Con la definición de la clase del último ejemplo podemos hacer lo que se ilustra en el siguiente:

```
int main() {
   pareja par1(12, 32), par2;
   int x, y;

   par2 = par1;
   par2.Lee(x, y);
   cout << "Valor de par2.a: " << x << endl;
   cout << "Valor de par2.b: " << y << endl;

   cin.get();
   return 0;
}
```

La línea "par2 = par1;" copia los valores de los datos miembros de par1 en par2.

Constructor copia

Un constructor de este tipo crea un objeto a partir de otro objeto existente. Estos constructores sólo tienen un argumento, que es una referencia a un objeto de su misma clase.

En general, los constructores copia tienen la siguiente forma para sus prototipos:

```
tipo_clase::tipo_clase(const tipo_clase &obj);
```

De nuevo ilustraremos esto con un ejemplo y usaremos también "pareja":

```
class pareja {
   public:
      // Constructor
      pareja(int a2=0, int b2=0) : a(a2), b(b2) {}
      // Constructor copia:
      pareja(const pareja &p);

      // Funciones miembro de la clase "pareja"
      void Lee(int &a2, int &b2);
      void Guarda(int a2, int b2);
   private:
      // Datos miembro de la clase "pareja"
      int a, b;
   public:
};

pareja::pareja(const pareja &p) : a(p.a), b(p.b) {}
```

Para crear objetos usando el constructor copia se procede como sigue:

```
int main() {
   pareja par1(12, 32)
   pareja par2(par1); // Uso del constructor copia: par2 = par1
   int x, y;

   par2.Lee(x, y);
   cout << "Valor de par2.a: " << x << endl;
   cout << "Valor de par2.b: " << y << endl;

   cin.get();
   return 0;
}
```

También en este caso, si no se especifica ningún constructor copia, el compilador crea uno por defecto, y su comportamiento es exactamente el mismo que el del definido en el ejemplo anterior. Para la mayoría de los casos esto será suficiente, pero en muchas ocasiones necesitaremos redefinir el constructor copia.

Destructores

Los destructores son funciones miembro especiales que sirven para eliminar un objeto de una determinada clase, liberando la memoria utilizada por dicho objeto.

Los destructores tienen el mismo nombre que la clase, pero con el símbolo ~ delante, no retornan ningún valor y no pueden ser heredados.

Cuando se define un destructor para una clase, éste es llamado automáticamente cuando se abandona el ámbito en el que fue definido. Esto es así salvo cuando el objeto fue creado dinámicamente con el operador new, ya que en ese caso, si es necesario eliminarlo, hay que usar el operador delete.

En general, será necesario definir un destructor cuando nuestra clase tenga datos miembro de tipo puntero, aunque esto no es una regla estricta. El destructor no puede sobrecargarse, por la sencilla razón de que no admite argumentos.

Ejemplo:

```
#include <iostream>
#include <cstring>
using namespace std;

class cadena {
  public:
   cadena();        // Constructor por defecto
   cadena(char *c); // Constructor desde cadena c
   cadena(int n);   // Constructor de cadena de n caracteres
   cadena(const cadena &);   // Constructor copia
   ~cadena();       // Destructor

   void Asignar(char *dest);
   char *Leer(char *c);
  private:
   char *cad;       // Puntero a char: cadena de caracteres
};

cadena::cadena() : cad(NULL) {}

cadena::cadena(char *c) {
   cad = new char[strlen(c)+1];// Reserva memoria para cadena
   strcpy(cad, c);             // Almacena la cadena
}

cadena::cadena(int n) {
   cad = new char[n+1]; // Reserva memoria para n caracteres
   cad[0] = 0;          // Cadena vacía
}

cadena::cadena(const cadena &Cad) {
   // Reservamos memoria para la nueva y la almacenamos
   cad = new char[strlen(Cad.cad)+1];
   // Reserva memoria para cadena
   strcpy(cad, Cad.cad);             // Almacena la cadena
}

cadena::~cadena() {
   delete[] cad;        // Libera la memoria reservada a cad
}

void cadena::Asignar(char *dest) {
   // Eliminamos la cadena actual:
   delete[] cad;
   // Reservamos memoria para la nueva y la almacenamos
   cad = new char[strlen(dest)+1];
   // Reserva memoria para la cadena
   strcpy(cad, dest);              // Almacena la cadena
}
```

```
char *cadena::Leer(char *c) {
   strcpy(c, cad);
   return c;
}

int main() {
   cadena Cadena1("Cadena de prueba");
   cadena Cadena2(Cadena1);   // Cadena2 es copia de Cadena1
   cadena *Cadena3;           // Cadena3 es un puntero
   char c[256];

   // Modificamos Cadena1:
   Cadena1.Asignar("Otra cadena diferente");
   // Creamos Cadena3:
   Cadena3 = new cadena("Cadena de prueba nº 3");

   // Ver resultados
   cout << "Cadena 1: " << Cadena1.Leer(c) << endl;
   cout << "Cadena 2: " << Cadena2.Leer(c) << endl;
   cout << "Cadena 3: " << Cadena3->Leer(c) << endl;

   delete Cadena3;  // Destruir Cadena3.
   // Cadena1 y Cadena2 se destruyen automáticamente

   cin.get();
   return 0;
}
```

Voy a hacer varias observaciones sobre este programa:

1. Hemos implementado un constructor copia. Esto es necesario porque una simple asignación entre los datos miembro "cad" no copiaría la cadena de un objeto a otro, sino únicamente los punteros.

   Por ejemplo, si definimos el constructor copia como:

   ```
   cadena::cadena(const cadena &Cad) {
      cad = Cad.cad;
   }
   ```

   En lugar de cómo lo hacemos, lo que estaríamos copiando sería el valor del puntero cad, con lo cual, ambos punteros estarían apuntando a la misma posición de memoria. Esto es desastroso, y no simplemente porque los cambios en una cadena afectan a las dos, sino porque al abandonar el programa se intenta liberar automáticamente la misma memoria dos veces. Lo que realmente pretendemos al asignar cadenas es crear una nueva cadena que sea copia de la cadena antigua. Esto es lo que hacemos con el constructor copia, y es lo que haremos más adelante, y con más elegancia, sobrecargando el operador de asignación.

   La definición del constructor copia que hemos creado en este último ejemplo es la equivalente a la del constructor copia por defecto.

2. La función Leer, que usamos para obtener el valor de la cadena almacenada, no devuelve un puntero a la cadena, sino una copia de la cadena. Esto está de acuerdo con las

recomendaciones sobre la programación orientada a objetos, que aconsejan que los datos almacenados en una clase no sean accesibles directamente desde fuera de ella, sino únicamente a través de las funciones creadas al efecto. Además, el miembro cad es privado, y por lo tanto debe ser inaccesible desde fuera de la clase. Más adelante veremos cómo se puede conseguir mantener la seguridad sin crear más datos miembro.

3. La Cadena3 debe ser destruida implícitamente usando el operador delete, que a su vez invoca al destructor de la clase. Esto es así porque se trata de un puntero, y la memoria que se usa en el objeto al que apunta no se libera automáticamente al destruirse el puntero Cadena3.

El puntero this

Para cada objeto declarado de una clase se mantiene una copia de sus datos, pero todos comparten la misma copia de las funciones de esa clase.

Esto ahorra memoria y hace que los programas ejecutables sean más compactos, pero plantea un problema.

Cada función de una clase puede hacer referencia a los datos de un objeto, modificarlos o leerlos, pero si sólo hay una copia de la función y varios objetos de esa clase, ¿cómo hace la función para referirse a un dato de un objeto en concreto?

La respuesta es: usando el puntero this. Cada objeto tiene asociado un puntero a si mismo que se puede usar para manejar sus miembros.

Volvamos al ejemplo de la clase pareja:

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

class pareja {
   public:
      // Constructor
      pareja(int a2, int b2);
      // Funciones miembro de la clase "pareja"
      void Lee(int &a2, int &b2);
      void Guarda(int a2, int b2);
   private:
      // Datos miembro de la clase "pareja"
      int a, b;
   public:
};
```

Para cada dato podemos referirnos de dos modos distintos, lo veremos con la función Guarda. Esta es la implementación que usamos en capítulos anteriores , que es como normalmente nos referiremos a los miembros de las clases:

```
void pareja::Guarda(int a2, int b2) {
   a = a2;
   b = b2;
}
```

Veamos ahora la manera equivalente usando el puntero this clase1.ccs:

```
void pareja::Guarda(int a2, int b2) {
   this->a = a2;
   this->b = b2;
}
```

Veamos otro ejemplo donde podemos aplicar el operador this. Se trata de la aplicación más frecuente, como veremos al implementar el constructor copia, o al sobrecargar ciertos operadores.

A veces necesitamos invocar a una función de una clase con una referencia a un objeto de la misma clase, pero las acciones a tomar serán diferentes dependiendo de si la referencia que pasamos se refiere al mismo objeto o a otro diferente, veamos cómo podemos usar el puntero this para determinar esto:

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

class clase {
  public:
   clase() {}
   void EresTu(clase& c) {
      if(&c == this) cout << "Sí, soy yo." << endl;
      else cout << "No, no soy yo." << endl;
   }
};

int main() {
   clase c1, c2;

   c1.EresTu(c2);
   c1.EresTu(c1);

   cin.get();
   return 0;
}
```

La función "EresTu" recibe una referencia a un objeto de la clase "clase". Para saber si se trata del mismo objeto, comparamos la dirección del objeto recibido con el valor de this, si son la misma, es que se trata del mismo objeto. La salida de este programa sería:

```
No, no soy yo.
Sí, soy yo.
```

Normalmente no será necesario usar el puntero this en nuestros programas, pero nos resultará muy útil en el futuro, ya que existen situaciones en las que es necesario recurrir a este puntero.

Capitulo 15 Trabajar con ficheros

Usar streams facilita mucho el acceso a ficheros en disco, veremos que una vez que creemos un stream para un fichero, podremos trabajar con él igual que lo hacemos con cin o cout.

Mediante las clases ofstream, ifstream y fstream tendremos acceso a todas las funciones de las clases base de las que se derivan estas: ios, istream, ostream, fstreambase, y como también contienen un objeto filebuf, podremos acceder a las funciones de filebuf y streambuf.

En apendice D hay una referencia bastante completa de las clases estándar de entrada y salida.

Evidentemente, muchas de estas funciones puede que nunca nos sean de utilidad, pero algunas de ellas se usan con frecuencia, y facilitan mucho el trabajo con ficheros.

Crear un fichero de salida, abrir un fichero de entrada

Empezaremos con algo sencillo. Vamos a crear un fichero mediante un objeto de la clase ofstream, y posteriormente lo leeremos mediante un objeto de la clase ifstream:

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

int main() {
   char cadena[128];
   // Crea un fichero de salida
   ofstream fs("nombre.txt");

   // Enviamos una cadena al fichero de salida:
   fs << "Hola, mundo" << endl;
   // Cerrar el fichero,
   // para luego poder abrirlo para lectura:
   fs.close();

   // Abre un fichero de entrada
   ifstream fe("nombre.txt");

   // Leeremos mediante getline, si lo hiciéramos
   // mediante el operador >> sólo leeríamos
   // parte de la cadena:
   fe.getline(cadena, 128);

   cout << cadena << endl;

   cin.get();
   return 0;
}
```

Este sencillo ejemplo crea un fichero de texto y después visualiza su contenido en pantalla.

Veamos otro ejemplo sencillo, para ilustrar algunas *limitaciones* del operador >> para hacer lecturas, cuando no queremos perder caracteres.

Supongamos que llamamos a este programa "streams.cpp", y que pretendemos que se autoimprima en pantalla:

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

int main() {
   char cadena[128];
   ifstream fe("streams.cpp");

   while(!fe.eof()) {
      fe >> cadena;
      cout << cadena << endl;
   }
   fe.close();

   cin.get();
   return 0;
}
```

El resultado quizá no sea el esperado. El motivo es que el operador >> interpreta los espacios, tabuladores y retornos de línea como separadores, y los elimina de la cadena de entrada.

Ficheros binarios:

Muchos sistemas operativos distinguen entre ficheros de texto y ficheros binarios. Por ejemplo, en MS-DOS, los ficheros de texto sólo permiten almacenar caracteres.

En otros sistemas no existe tal distinción, todos los ficheros son binarios. En esencia esto es más correcto, puesto que un fichero de texto es un fichero binario con un rango limitado para los valores que puede almacenar.

En general, usaremos ficheros de texto para almacenar información que pueda o deba ser manipulada con un editor de texto. Un ejemplo es un fichero fuente C++. Los ficheros binarios son más útiles para guardar información cuyos valores no estén limitados. Por ejemplo, para almacenar imágenes, o bases de datos. Un fichero binario permite almacenar estructuras completas, en las que se mezclen datos de cadenas con datos numéricos.

En realidad no hay nada que nos impida almacenar cualquier valor en un fichero de texto, el problema surge cuando se almacena el valor que el sistema operativo usa para marcar el fin de fichero en un archivo de texto. En MS-DOS ese valor es 0x1A. Si abrimos un fichero en modo de texto que contenga un dato con ese valor, no nos será posible leer ningún dato a partir de esa posición. Si lo abrimos en modo binario, ese problema no existirá.

Los ficheros que hemos usado en los ejemplos anteriores son en modo texto, veremos ahora un ejemplo en modo binario:

```
#include <fstream>
#include <cstring>

struct tipoRegistro {
   char nombre[32];
   int edad;
   float altura;
```

```
};

int main() {
   tipoRegistro pepe;
   tipoRegistro pepe2;
   ofstream fsalida("prueba.dat",
      ios::out | ios::binary);

   strcpy(pepe.nombre, "Jose Luis");
   pepe.edad = 32;
   pepe.altura = 1.78;

   fsalida.write(reinterpret_cast<char *>(&pepe),
      sizeof(tipoRegistro));
   fsalida.close();

   ifstream fentrada("prueba.dat",
      ios::in | ios::binary);

   fentrada.read(reinterpret_cast<char *>(&pepe2),
      sizeof(tipoRegistro));
   cout << pepe2.nombre << endl;
   cout << pepe2.edad << endl;
   cout << pepe2.altura <<endl;
   fentrada.close();

   cin.get();
   return 0;
}
```

Al declarar streams de las clases ofstream o ifstream y abrirlos en modo binario, tenemos que añadir el valor ios::out e ios::in, respectivamente, al valor ios::binary. Esto es necesario porque los valores por defecto para el modo son ios::out e ios:in, también respectivamente, pero al añadir el flag ios::binary, el valor por defecto no se tiene en cuenta.

Cuando trabajemos con streams binarios usaremos las funciones write y read. En este caso nos permiten escribir y leer estructuras completas.

En general, cuando usemos estas funciones necesitaremos hacer un casting, es recomendable usar el operador "reinterpret_cast".

Ficheros de acceso aleatorio.

Hasta ahora sólo hemos trabajado con los ficheros secuencialmente, es decir, empezamos a leer o a escribir desde el principio, y avanzamos a medida que leemos o escribimos en ellos.

Otra característica importante de los ficheros es la posibilidad de trabajar con ellos haciendo acceso aleatorio, es decir, poder hacer lecturas o escrituras en cualquier punto del fichero. Para eso disponemos de las funciones seekp y seekg, que permiten cambiar la posición del fichero en la que se hará la siguiente escritura o lectura. La 'p' es de *put* y la 'g' de *get*, es decir escritura y lectura, respectivamente.

Otro par de funciones relacionadas con el acceso aleatorio son tellp y tellg, que sirven para saber en qué posición del fichero nos encontramos.

```
#include <fstream>
```

```
using namespace std;

int main() {
   int i;
   char mes[][20] = {"Enero", "Febrero", "Marzo",
      "Abril", "Mayo", "Junio", "Julio", "Agosto",
      "Septiembre", "Octubre", "Noviembre",
      "Diciembre"};
   char cad[20];

   ofstream fsalida("meses.dat",
      ios::out | ios::binary);

   // Crear fichero con los nombres de los meses:
   cout << "Crear archivo de nombres de meses:" << endl;
   for(i = 0; i < 12; i++)
      fsalida.write(mes[i], 20);
   fsalida.close();

   ifstream fentrada("meses.dat", ios::in | ios::binary);

   // Acceso secuencial:
   cout << "\nAcceso secuencial:" << endl;
   fentrada.read(cad, 20);
   do {
      cout << cad << endl;
      fentrada.read(cad, 20);
   } while(!fentrada.eof());

   fentrada.clear();
   // Acceso aleatorio:
   cout << "\nAcceso aleatorio:" << endl;
   for(i = 11; i >= 0; i--) {
      fentrada.seekg(20*i, ios::beg);
      fentrada.read(cad, 20);
      cout << cad << endl;
   }

   // Calcular el número de elementos
   // almacenados en un fichero:
   // ir al final del fichero
   fentrada.seekg(0, ios::end);
   // leer la posición actual
   pos = fentrada.tellg();
   // El número de registros es el tamaño en
   // bytes dividido entre el tamaño del registro:
   cout << "\nNúmero de registros: " << pos/20 << endl;
   fentrada.close();

   cin.get();
   return 0;
}
```

La función seekg nos permite acceder a cualquier punto del fichero, no tiene por qué ser exactamente al principio de un registro, la resolución de la funciones seek es de un byte.

Cuando trabajemos con nuestros propios streams para nuestras clases, derivándolas de ifstream, ofstream o fstream, es posible que nos convenga sobrecargar las funciones seek y tell para que trabajen a nivel de registro, en lugar de hacerlo a nivel de byte.

La función seekp nos permite sobrescribir o modificar registros en un fichero de acceso aleatorio de salida. La función tellp es análoga a tellg, pero para ficheros de salida.

Ficheros de entrada y salida:

Ahora veremos cómo podemos trabajar con un stream simultáneamente en entrada y salida.

Para eso usaremos la clase fstream, que al ser derivada de ifstream y ofstream, dispone de todas las funciones necesarias para realizar cualquier operación de entrada o salida.

Hay que tener la precaución de usar la opción ios::trunc de modo que el fichero sea creado si no existe previamente.

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

int main() {
   char l;
   long i, lon;
   fstream fich("prueba.dat", ios::in |
      ios::out | ios::trunc | ios::binary);

   fich << "abracadabra" << flush;

   fich.seekg(0L, ios::end);
   lon = fich.tellg();
   for(i = 0L; i < lon; i++) {
      fich.seekg(i, ios::beg);
      fich.get(l);
      if(l == 'a') {
         fich.seekp(i, ios::beg);
         fich << 'e';
      }
   }
   cout << "Salida:" << endl;
   fich.seekg(0L, ios::beg);
   for(i = 0L; i < lon; i++) {
      fich.get(l);
      cout << l;
   }
   cout << endl;
   fich.close();

   cin.get();;
   return 0;
}
```

Este programa crea un fichero con una palabra, a continuación lee todo el fichero e cambia todos los caracteres 'a' por 'e'. Finalmente muestra el resultado.

Básicamente muestra cómo trabajar con ficheros simultáneamente en entrada y salida.

Sobrecarga de operadores << y >>

Veremos en otro momento mas sobre el tema sobrecarga de Operadores, pero en este momento haremos referencia al pasar de una de las principales ventajas de trabajar con streams es que nos

permiten sobrecargar los operadores << y >> para realizar salidas y entradas de nuestros propios tipos de datos.

Por ejemplo, tenemos una clase:

```
#include <iostream>
#include <cstring>
using namespace std;

class Registro {
  public:
   Registro(char *, int, char *);
   const char* LeeNombre() const {return nombre;}
   int LeeEdad() const {return edad;}
   const char* LeeTelefono() const {return telefono;}

  private:
   char nombre[64];
   int edad;
   char telefono[10];
};

Registro::Registro(char *n, int e, char *t) : edad(e) {
   strcpy(nombre, n);
   strcpy(telefono, t);
}

ostream& operator<<(ostream &os, Registro& reg) {
   os << "Nombre: " << reg.LeeNombre() <<"\nEdad: " <<
      reg.LeeEdad() << "\nTelefono: " << reg.LeeTelefono();

   return os;
}

int main() {
   Registro Pepe("José", 32, "61545552");

   cout << Pepe << endl;

   cin.get();
   return 0;
}
```

Comprobar estado de un stream:

Hay varios flags de estado que podemos usar para comprobar el estado en que se encuentra un stream.

Concretamente nos puede interesar si hemos alcanzado el fin de fichero, o si el stream con el que estamos trabajando está en un estado de error.

La función principal para esto es *good()*, de la clase ios.

Después de ciertas operaciones con streams, a menudo no es mala idea comprobar el estado en que ha quedado el stream. Hay que tener en cuenta que ciertos estados de error impiden que se puedan seguir realizando operaciones de entrada y salida.

Otras funciones útiles son *fail()*, *eof()*, *bad()*, *rdstate()* o *clear()*.

En el ejemplo de archivos de acceso aleatorio hemos usado clear() para eliminar el bit de estado eofbit del fichero de entrada, si no hacemos eso, las siguientes operaciones de lectura fallarían.

Otra condición que conviene verificar es la existencia de un fichero. En los ejemplos anteriores no ha sido necesario, aunque hubiera sido conveniente, verificar la existencia, ya que el propio ejemplo crea el fichero que después lee.

Cuando vayamos a leer un fichero que no podamos estar seguros de que existe, o que aunque exista pueda estar abierto por otro programa, debemos asegurarnos de que nuestro programa tiene acceso al stream. Por ejemplo:

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

int main() {
   char mes[20];
   ifstream fich("meses1.dat", ios::in | ios::binary);

   // El fichero meses1.dat no existe, este programa es
   // una prueba de los bits de estado.

   if(fich.good()) {
      fich.read(mes, 20);
      cout << mes << endl;
   }
   else {
      cout << "Fichero no disponible" << endl;
      if(fich.fail()) cout << "Bit fail activo" << endl;
      if(fich.eof())  cout << "Bit eof activo" << endl;
      if(fich.bad())  cout << "Bit bad activo" << endl;
   }
   fich.close();

   cin.get();
   return 0;
}
```

Ejemplo de fichero previamente abierto:

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

int main() {
   char mes[20];
   ofstream fich1("meses.dat", ios::out | ios::binary);
   ifstream fich("meses.dat", ios::in | ios::binary);

   // El fichero meses.dat existe, pero este programa
   // intenta abrir dos streams al mismo fichero, uno en
   // escritura y otro en lectura. Eso no es posible, se
   // trata de una prueba de los bits de estado.

   fich.read(mes, 20);
   if(fich.good())
      cout << mes << endl;
   else {
      cout << "Error al leer de Fichero" << endl;
      if(fich.fail()) cout << "Bit fail activo" << endl;
      if(fich.eof())  cout << "Bit eof activo" << endl;
```

```
      if(fich.bad())  cout << "Bit bad activo" << endl;
   }
   fich.close();
   fich1.close();

   cin.get();
   return 0;
}
```

Punteros a miembros de clases o estructuras

C++ permite declarar punteros a miembros de clases, estructuras y uniones. Aunque en el caso de las clases, los miembros deben ser públicos para que pueda accederse a ellos.

La sintaxis para la declaración de un puntero a un miembro es la siguiente:

```
<tipo> <clase|estructura|unión>::*<identificador>;
```

De este modo se declara un puntero "identificador" a un miembro de tipo "tipo" de la clase, estructura o unión especificada.

Ejemplos:

```
struct punto3D {
   int x;
   int y;
   int z;
};

class registro {
  public:
   registro();

   float v;
   float w;
};

int punto3D::*coordenada; // (1)
float registro::*valor;   // (2)
```

El primer ejemplo declara un puntero "coordenada" a un miembro de tipo "int" de la estructura "punto3D". El segundo declara un puntero "valor" a un miembro público de la clase "registro".

Asignación de valores a punteros a miembro:

Una vez declarado un puntero, debemos asignarle la dirección de un miembro del tipo adecuado de la clase, estructura o unión. Podremos asignarle la dirección de cualquiera de los miembros del tipo adecuado. La sintaxis es:

```
<identificador> = &<clase|estructura|unión>::<campo>;
```

En el ejemplo anterior, serían válidas las siguientes asignaciones:

```
coordenada = &punto3D::x;
coordenada = &punto3D::y;
```

```
coordenada = &punto3D::z;
valor = &registro::v;
valor = &registro::w;
```

Operadores .* y ->*:

Ahora bien, ya sabemos cómo declarar punteros a miembros, pero no cómo trabajar con ellos.

C++ dispone de dos operadores para trabajar con punteros a miembros: **.*** y **->***. A lo mejor los echabas de menos :-).

Se trata de dos variantes del mismo operador, uno para objetos y otro para punteros:

```
<objeto>.*<puntero>
<puntero_a_objeto>->*<puntero>
```

La primera forma se usa cuando tratamos de acceder a un miembro de un objeto.

La segunda cuando lo hacemos a través de un puntero a un objeto.

Veamos un ejemplo completo:

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

class clase {
  public:
   clase(int a, int b) : x(a), y(b) {}

  public:
   int x;
   int y;
};

int main() {
   clase uno(6,10);
   clase *dos = new clase(88,99);
   int clase::*puntero;

   puntero = &clase::x;
   cout << uno.*puntero << endl;
   cout << dos->*puntero << endl;

   puntero = &clase::y;
   cout << uno.*puntero << endl;
   cout << dos->*puntero << endl;

   delete dos;
   cin.get();
   return 0;
}
```

La utilidad práctica no es probable que se presente frecuentemente, y casi nunca con clases, ya que no es corriente declarar miembros públicos. Sin embargo nos ofrece algunas posibilidades interesantes a la hora de recorrer miembros concretos de arrays de estructuras, aplicando la misma función o expresión a cada uno.

También debemos recordar que es posible declarar punteros a funciones, y las funciones miembros de clases no son una excepción. En ese caso sí es corriente que existan funciones públicas.

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

class clase {
  public:
   clase(int a, int b) : x(a), y(b) {}
   int funcion(int a) {
      if(0 == a) return x; else return y;
   }

  private:
   int x;
   int y;
};

int main() {
   clase uno(6,10);
   clase *dos = new clase(88,99);
   int (clase::*pfun)(int);

   pfun = &clase::funcion;

   cout << (uno.*pfun)(0) << endl;
   cout << (uno.*pfun)(1) << endl;
   cout << (dos->*pfun)(0) << endl;
   cout << (dos->*pfun)(1) << endl;

   delete dos;
   cin.get();
   return 0;
}
```

Para ejecutar una función desde un puntero a miembro hay que usar los paréntesis, ya que el operador de llamada a función "()" tiene mayor prioridad que los operadores ".*" y "->*".

Sistema de protección

Ya sabemos que los miembros privados de una clase no son accesibles para funciones y clases exteriores a dicha clase.

Esto es un concepto de POO, el encapsulamiento hace que cada objeto se comporte de un modo autónomo y que lo que pase en su interior sea invisible para el resto de objetos. Cada objeto sólo responde a ciertos mensajes y proporciona determinadas salidas.

Pero, en ciertas ocasiones, querremos tener acceso a determinados miembros privados de un objeto de una clase desde otros objetos de clases diferentes. C++ proporciona un mecanismo para sortear el sistema de protección. En otros capítulos veremos la utilidad de esta técnica, de momento sólo explicaremos en qué consiste.

Operadores sobrecargados:

Ya habíamos visto el funcionamiento de los operadores sobrecargados en el capítulo 22, aplicándolos a operaciones con estructuras. Ahora veremos todo su potencial, aplicándolos a clases.

Sobrecarga de operadores binarios:

Empezaremos por los operadores binarios, que como recordarás son aquellos que requieren dos operandos, como la suma o la resta.

Existe una diferencia entre la sobrecarga de operadores que vimos en el capítulo 24, que se definía fuera de las clases. Cuando se sobrecargan operadores en el interior se asume que el primer operando es el propio objeto de la clase donde se define el operador. Debido a esto, sólo se necesita especificar un operando.

Sintaxis:

```
<tipo> operator<operador binario>(<tipo> <identificador>);
```

Normalmente el **<tipo>** es la clase para la que estamos sobrecargando el operador, tanto en el valor de retorno como en el parámetro.

Veamos un ejemplo para una clase para el tratamiento de tiempos:

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

class Tiempo {
  public:
   Tiempo(int h=0, int m=0) : hora(h), minuto(m) {}

   void Mostrar();
   Tiempo operator+(Tiempo h);

  private:
   int hora;
   int minuto;
};

Tiempo Tiempo::operator+(Tiempo h) {
   Tiempo temp;

   temp.minuto = minuto + h.minuto;
   temp.hora   = hora   + h.hora;

   if(temp.minuto >= 60) {
      temp.minuto -= 60;
      temp.hora++;
   }
   return temp;
}

void Tiempo::Mostrar() {
   cout << hora << ":" << minuto << endl;
}
```

```
int main() {
   Tiempo Ahora(12,24), T1(4,45);

   T1 = Ahora + T1;   // (1)
   T1.Mostrar();

   (Ahora + Tiempo(4,45)).Mostrar(); // (2)

   cin.get();
   return 0;
}
```

Me gustaría hacer algunos comentarios sobre el ejemplo:

Observa que cuando sumamos dos tiempos obtenemos un tiempo, se trata de una propiedad de la suma, todos sabemos que no se pueden sumar peras y manzanas.

Pero en C++ sí se puede. Por ejemplo, podríamos haber sobrecargado el operador suma de este modo:

```
int operator+(Tiempo h);
```

Pero no estaría muy clara la naturaleza del resultado, ¿verdad?. Lo lógico es que la suma de dos objetos produzca un objeto del mismo tipo o la misma clase.

Hemos usado un objeto temporal para calcular el resultado de la suma, esto es necesario porque necesitamos operar con los minutos para prevenir el caso en que excedan de 60, en cuyo caso incrementaremos el tiempo en una hora.

Ahora observa cómo utilizamos el operador en el programa.

La forma (1) es la forma más lógica, para eso hemos creado un operador, para usarlo igual que en las situaciones anteriores.

Pero verás que también hemos usado el operador =, a pesar de que nosotros no lo hemos definido. Esto es porque el compilador crea un operador de asignación por defecto si nosotros no lo hacemos, pero veremos más sobre eso en el siguiente punto.

La forma (2) es una pequeña aberración, pero ilustra cómo es posible crear objetos temporales sin nombre.

En esta línea hay dos, el primero Tiempo(4,45), que se suma a Ahora para producir otro objeto temporal sin nombre, que es el que mostramos en pantalla.

Sobrecargar el operador de asignación: ¿por qué?

Ya sabemos que el compilador crea un operador de asignación por defecto si nosotros no lo hacemos, así que ¿por qué sobrecargarlo?.

Bueno, veamos lo que pasa si nuestra clase tiene miembros que son punteros, por ejemplo:

```
class Cadena {
  public:
   Cadena(char *cad);
   Cadena() : cadena(NULL) {};
   ~Cadena() { delete[] cadena; };
```

```
   void Mostrar() const;
  private:
   char *cadena;
};

Cadena::Cadena(char *cad) {
   cadena = new char[strlen(cad)+1];
   strcpy(cadena, cad);
}

void Cadena::Mostrar() const {
   cout << cadena << endl;
}
```

Si en nuestro programa declaramos dos objetos de tipo Cadena:

```
Cadena C1("Cadena de prueba"), C2;
```

Y hacemos una asignación:

```
C2 = C1;
```

Lo que realmente copiamos no es la cadena, sino el puntero. Ahora los dos punteros de las cadenas C1 y C2 están apuntando a la misma dirección. ¿Qué pasará cuando destruyamos los objetos?. Al destruir C1 se intentará liberar la memoria de su puntero cadena, y al destruir C2 también, pero ambos punteros apuntan a la misma dirección y el valor original del puntero de C2 se ha perdido, por lo que su memoria no puede ser liberada.

En estos casos, análogamente a lo que sucedía con el constructor copia, deberemos sobrecargar el operador de asignación. En nuestro ejemplo podría ser así:

```
Cadena &Cadena::operator=(const Cadena &c) {
   if(this != &c) {
      delete[] cadena;
      if(c.cadena) {
         cadena = new char[strlen(c.cadena)+1];
         strcpy(cadena, c.cadena);
      }
      else cadena = NULL;
   }
   return *this;
}
```

Hay que tener en cuenta la posibilidad de que se asigne un objeto a si mismo. Por eso comparamos el puntero this con la dirección del parámetro, si son iguales es que se trata del mismo objeto, y no debemos hacer nada. Esta es una de las situaciones en las que el puntero this es imprescindible.

También hay que tener cuidado de que la cadena a copiar no sea NULL, en ese caso no debemos copiar la cadena, sino sólo asignar NULL a cadena.

Y por último, también es necesario retornar una referencia al objeto, esto nos permitirá escribir expresiones como estas:

```
C1 = C2 = C3;
if((C1 = C2) == C3)...
```

Por supuesto, para el segundo caso deberemos sobrecargar también el operador ==.

Operadores binarios que pueden sobrecargarse:

Además del operador + pueden sobrecargarse prácticamente todos los operadores:

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

Los operadores =, [], () y -> sólo pueden sobrecargarse en el interior de clases.

Por ejemplo, el operador > podría declararse y definirse así:

```
class Tiempo {
...
bool operator>(Tiempo h);
...
};

bool Tiempo::operator>(Tiempo h) {
   return (hora > h.hora ||
           (hora == h.hora && minuto > h.minuto));
}
...
if(Tiempo(1,32) > Tiempo(1,12))
   cout << "1:32 es mayor que 1:12" << endl;
else
   cout << "1:32 es menor o igual que 1:12" << endl;
...
```

Para los operadores de igualdad el valor de retorno es bool, lógicamente, ya que estamos haciendo una comparación.

Y el operador +=, de esta otra:

```
class Tiempo {
...
void operator+=(Tiempo h);
...
};

void Tiempo::operator+=(Tiempo h) {
   minuto += h.minuto;
   hora   += h.hora;

   while(minuto >= 60) {
      minuto -= 60;
      hora++;
   }
}
...
Ahora += Tiempo(1,32);
Ahora.Mostrar();
...
```

Los operadores de asignación mixtos no necesitan valor de retorno, ya que es el propio objeto al que se aplican el que recibe el resultado de la operación y además, no pueden asociarse.

Con el resto de lo operadores binarios se trabaja del mismo modo.

No es imprescindible mantener el significado de los operadores. Por ejemplo, para la clase Tiempo

no tiene sentido sobrecargar el operadores >>, <<, * ó /, pero podemos hacerlo de todos modos, y olvidar el significado que tengan habitualmente. De igual modo podríamos haber sobrecargado el operador + y hacer que no sumara los tiempos sino que, por ejemplo, los restara. En última instancia, es el programador el que decide el significado de los operadores.

Por ejemplo, sobrecargaremos el operador >> para que devuelva el mayor de los operandos.

```
class Tiempo {
...
Tiempo operator>>(Tiempo h);
...
};

Tiempo Tiempo::operator>>(Tiempo h) {
   if(*this > h) return *this; else return h;
}
...

T1 = Ahora >> Tiempo(13,43) >> T1 >> Tiempo(12,32);
T1.Mostrar();
...
```

En este ejemplo hemos recurrido al puntero this, para usar el objeto actual en una comparación y para devolverlo como resultado en el caso adecuado.

Esta es otra de las aplicaciones del puntero this, si no dispusiéramos de él, sería imposible hacer referencia al propio objeto al que se aplica el operador.

También vemos que los operadores binarios deben seguir admitiendo la asociación aún estando sobrecargados.

Forma funcional de los operadores:

Por supuesto también es posible usar la forma funcional de los operadores sobrecargados, aunque no es muy habitual ni aconsejable.

En el caso del operador + las siguientes expresiones son equivalentes:

```
T1 = T1.operator+(Ahora);

T1 = Ahora + T1;
```

Sobrecarga de operadores para clases con punteros:

Si intentamos sobrecargar el operador suma con la clase Cadena usando el mismo sistema que con Tiempo, veremos que no funciona.

Cuando nuestras clases tienen punteros con memoria dinámica asociada, la sobrecarga de funciones y operadores puede complicarse un poco.

Por ejemplo, sobrecarguemos el operador + para la clase Cadena:

```
class Cadena {
...
   Cadena operator+(const Cadena &);
```

```
...
};

Cadena Cadena::operator+(const Cadena &c) {
   Cadena temp;

   temp.cadena = new char[strlen(c.cadena)+strlen(cadena)+1];
   strcpy(temp.cadena, cadena);
   strcat(temp.cadena, c.cadena);
   return temp;
}
...
Cadena C1, C2("Primera parte");

C1 = C2 + " Segunda parte";
```

Ahora analicemos cómo funciona el código de este operador.

El equivalente de ésta última línea es:

```
C1.operator=(Cadena(C2.operator+(Cadena(" Segunda parte"))));
```

1) Se crea automáticamente un objeto temporal sin nombre para la cadena " Segunda parte". Y se llama al operador + del objeto C2.

2) Dentro del operador + se crea un objeto temporal: temp, reservamos memoria para la cadena que almacenará la concatenación de this->cadena y c.cadena, y le asignamos el valor de ambas cadenas, temp contiene la cadena: "Primera parte Segunda parte".

3) Retornamos el objeto temporal.

4) Ahora el objeto temporal temp se copia a otro objeto temporal sin nombre, y temp es destruido. Y el objeto temporal sin nombre se pasa como parámetro al operador de asignación. Si esto es difícil de entender, piensa lo que pasa cuando usamos el operador de asignación con una cadena, por ejemplo:

C1 = "hola";

En este caso se crea un objeto temporal sin nombre para "hola", igual que pasó con la cadena " Segunda parte".

5) Se asigna el objeto temporal sin nombre a C1, y se destruye.

Parece que todo ha ido bien, pero en el paso 4 hay un problema. Para copiar temp en el objeto temporal sin nombre se usa el constructor copia de Cadena. Pero como nosotros no hemos creado un constructor copia, se usará el constructor copia por defecto. Recuerda que ese constructor copia los punteros, no los contenidos de estos.

Recapitulemos: el objeto temp se copia en un temporal sin nombre, y después se destruye, ¿qué pasa con el dato temp.cadena?, evidentemente también se destruye, pero el constructor copia por defecto ha copiado ese puntero, por lo tanto, también su cadena es destruida. El resultado es que C1 no recibe la suma de las cadenas.

Para evitar eso tenemos que sobrecargar el constructor copia, afortunadamente es sencillo ya que disponemos del operador de asignación, sin olvidar que tenemos que inicializar los datos miembros, el constructor copia no deja de ser un constructor:

```
class Cadena {
...
   Cadena(const Cadena &c) : cadena(NULL) { *this = c; }
...
};
```

Si no tenemos cuidado de iniciar el valor de cadena, cuando se invoque al operador "=" el puntero cadena tendrá algún valor inválido, y al ejecutar el código del operador de asignación se producirá un error al intentar liberarlo.

```
Cadena &Cadena::operator=(const Cadena &c) {
   if(this != &c) {
      delete[] cadena; // (1)
      if(c.cadena) {
         cadena = new char[strlen(c.cadena)+1];
         strcpy(cadena, c.cadena);
      }
      else cadena = NULL;
   }
   return *this;
}
```

En (1), si cadena no es NULL, pero tampoco es un puntero válido, se producirá un error de ejecución. En general, si se usa el operador de asignación con objetos que existan no habrá problema, pero si se usa desde el constructor copia debemos asegurarnos de que el puntero es NULL.

La moraleja es que cuando nuestras clases tengan datos miembro que sean punteros a memoria dinámica debemos sobrecargar siempre el constructor copia, ya que nunca sabemos cuándo puede ser invocado sin que nos demos cuenta.

(Gracias a Steven por la idea de crear una clase Tiempo como ejemplo para la sobrecarga de operadores)

Sobrecarga de operadores unitarios:

Ahora le toca el turno a los operadores unitarios, que son aquellos que sólo requieren un operando, como la asignación o el incremento.

Cuando se sobrecargan operadores unitarios en una clase el operando es el propio objeto de la clase donde se define el operador. Por lo tanto los operadores unitarios dentro de las clases no requieren operandos.

Sintaxis:

```
<tipo> operator<operador unitario>();
```

Normalmente el <tipo> es la clase para la que estamos sobrecargando el operador. Sigamos con el ejemplo de la clase para el tratamiento de tiempos, sobrecargaremos ahora el operador de incremento ++:

```
class Tiempo {
...
```

```
Tiempo operator++();
...
};

Tiempo Tiempo::operator++() {
   minuto++;
   while(minuto >= 60) {
      minuto -= 60;
      hora++;
   }
   return *this;
}
...

T1.Mostrar();
++T1;
T1.Mostrar();
...
```

Operadores unitarios sufijos:

Lo que hemos visto vale para el preincremento, pero, ¿cómo se sobrecarga el operador de postincremento?

En realidad no hay forma de decirle al compilador cuál de las dos modalidades del operador estamos sobrecargando, así que los compiladores usan una regla: si se declara un parámetro para un operador ++ ó -- se sobrecargará la forma sufija del operador. El parámetro se ignorará, así que bastará con indicar el tipo.

También tenemos que tener en cuenta el peculiar funcionamiento de los operadores sufijos, cuando los sobrecarguemos, al menos si queremos mantener el comportamiento que tienen normalmente.

Cuando se usa un operador en la forma sufijo dentro de una expresión, primero se usa el valor actual del objeto, y una vez evaluada la expresión, se aplica el operador. Si nosotros queremos que nuestro operador actúe igual deberemos usar un objeto temporal, y asignarle el valor actual del objeto. Seguidamente aplicamos el operador al objeto actual y finalmente retornamos el objeto temporal.

Veamos un ejemplo:

```
class Tiempo {
...
Tiempo operator++();    // Forma prefija
Tiempo operator++(int); // Forma sufija
...
};

Tiempo Tiempo::operator++() {
   minuto++;
   while(minuto >= 60) {
      minuto -= 60;
      hora++;
   }
   return *this;
}

Tiempo Tiempo::operator++(int) {
```

```
   Tiempo temp(*this); // Constructor copia

   minuto++;
   while(minuto >= 60) {
      minuto -= 60;
      hora++;
   }
   return temp;
}
...

// Prueba:
T1.Mostrar();
(T1++).Mostrar();
T1.Mostrar();
(++T1).Mostrar();
T1.Mostrar();
...
```

Salida:

```
17:9 (Valor inicial)
17:9 (Operador sufijo, el valor no cambia
      hasta después de mostrar el valor)
17:10 (Resultado de aplicar el operador)
17:11 (Operador prefijo, el valor cambia
       antes de mostrar el valor)
17:11 (Resultado de aplicar el operador)
```

Operadores unitarios que pueden sobrecargarse:

Además del operador ++ y -- pueden sobrecargarse prácticamente todos los operadores unitarios:

+, -, ++, --, *, & y !.

Operadores de conversión de tipo:

Volvamos a nuestra clase Tiempo. Imaginemos que queremos hacer una operación como la siguiente:

```
Tiempo T1(12,23);
unsigned int minutos = 432;

T1 += minutos;
```

Con toda probabilidad no obtendremos el valor deseado.

Como ya hemos visto, en C++ se realizan conversiones implícitas entre los tipos básicos antes de operar con ellos, por ejemplo para sumar un int y un float, se convierte el entero a float. Esto se hace también en nuestro caso, pero no como esperamos.

El valor "minutos" se convierte a un objeto Tiempo, usando el constructor que hemos diseñado. Como sólo hay un parámetro, el parámetro m toma el valor 0, y para el parámetro h se convierte el valor "minutos" de unsigned int a int.

El resultado es que se suman 432 horas, y nosotros queremos sumar 432 minutos.

Esto se soluciona creando un nuevo constructor que tome como parámetro un unsigned int.

```
Tiempo(unsigned int m) : hora(0), minuto(m) {
   while(minuto >= 60) {
      minuto -= 60;
      hora++;
   }
}
```

Ahora el resultado será el adecuado.

En general podremos hacer conversiones de tipo desde cualquier objeto a un objeto de nuestra clase sobrecargando el constructor.

Pero también se puede presentar el caso contrario. Ahora queremos asignar a un entero un objeto Tiempo:

```
Tiempo T1(12,23);
int minutos;

minutos = T1;
```

En este caso obtendremos un error de compilación, ya que el compilador no sabe convertir un objeto Tiempo a entero.

Para eso tenemos que diseñar nuestro operador de conversión de tipo, que se aplicará automáticamente.

Los operadores de conversión de tipos tienen el siguiente formato:

```
operator <tipo>();
```

No necesitan que se especifique el tipo del valor de retorno, ya que este es precisamente <tipo>. Además, al ser operadores unitarios, tampoco requieren argumentos, se aplican al propio objeto.

```
class Tiempo {
...
operator int();
...

operator int() {
   return hora*60+minuto;
}
```

Por supuesto, el tipo no tiene por qué ser un tipo básico, puede tratarse de una estructura o una clase.

Sobrecarga del operador de indexación []:

El operador [] se usa para acceder a valores de objetos de una determinada clase como si se tratase de arrays. Los índices no tienen por qué ser de un tipo entero o enumerado, ahora no existe esa limitación.

Donde más útil resulta este operador es cuando se usa con estructuras dinámicas de datos: listas y

árboles. Pero también puede servirnos para crear arrays asociativos, donde los índices sean por ejemplo, palabras.

De nuevo explicaremos el uso de este operador usando un ejemplo.

Supongamos que hacemos una clase para hacer un histograma de los valores de rand()/RAND_MAX, entre los márgenes de 0 a 0.0009, de 0.001 a 0.009, de 0.01 a 0.09 y de 0.1 a 1.

Nota: Un histograma es un gráfico o una tabla utilizado en la representación de distribuciones de frecuencias de cualquier tipo de información o función. La clase de nuestro ejemplo podría usar los valores de la tabla para generar ese gráfico.

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

class Cuenta {
  public:
   Cuenta() { for(int i = 0; i < 4; contador[i++] = 0); }
   int &operator[](double n); // (1)

   void Mostrar() const;

  private:
   int contador[4];
};

int &Cuenta::operator[](double n) { // (2)
   if(n < 0.001) return contador[0];
   else if(n < 0.01) return contador[1];
   else if(n < 0.1) return contador[2];
   else return contador[3];
}

void Cuenta::Mostrar() const {
   cout << "Entre      0 y 0.0009: " << contador[0] << endl;
   cout << "Entre 0.0010 y 0.0099: " << contador[1] << endl;
   cout << "Entre 0.0100 y 0.0999: " << contador[2] << endl;
   cout << "Entre 0.1000 y 1.0000: " << contador[3] << endl;
}

int main() {
   Cuenta C;

   for(int i = 0; i < 50000; i++)
      C[(double)rand()/RAND_MAX]++; // (3)
   C.Mostrar();

   cin.get();
   return 0;
}
```

En este ejemplo hemos usado un valor double como índice, pero igualmente podríamos haber usado una cadena o cualquier objeto que hubiésemos querido.

El tipo del valor de retorno de operador debe ser el del objeto que devuelve (1). En nuestro caso, al tratarse de un contador, devolvemos un entero. Bueno, en realidad devolvemos una referencia a un entero, de este modo podemos aplicarle el operador de incremento al valor de retorno (3).

En la definición del operador (2), hacemos un tratamiento del parámetro que usamos como índice para adaptarlo al tipo de almacenamiento que usamos en nuestra clase.

Cuando se combina el operador de indexación con estructuras dinámicas de datos como las listas, se puede trabajar con ellas como si se tratada de arrays de objetos, esto nos dará una gran potencia y claridad en el código de nuestros programas.

Sobrecarga del operador de llamada ():

El operador () funciona exactamente igual que el operador [], aunque admite más parámetros.

Este operador permite usar un objeto de la clase para el que está definido como si fuera una función.

Como ejemplo añadiremos un operador de llamada a función que admita dos parámetros de tipo double y que devuelva el mayor contador de los asociados a cada uno de los parámetros.

```
class Cuenta {
...
   int operator()(double n, double m);
...
};

int Cuenta::operator()(double n, double m) {
   int i, j;

   if(n < 0.001) i = 0;
   else if(n < 0.01) i = 1;
   else if(n < 0.1) i = 2;
   else i = 3;

   if(m < 0.001) j = 0;
   else if(m < 0.01) j = 1;
   else if(m < 0.1) j = 2;
   else j = 3;

   if(contador[i] > contador[j]) return contador[i];
   else return contador[j];
}
...

cout << C(0.0034, 0.23) << endl;
...
```

Por supuesto, el número de parámetros, al igual que el tipo de retorno de la función depende de la decisión del programador.

xx