UNIDAD 9

Clases Derivadas.

1.- Introducción.

Cuando el programador inicia un nuevo proyecto, tiene la sensación de que el código que esta escribiendo ya lo había escrito antes. El código existente tal vez no sea totalmente adecuado para la aplicación en desarrollo, tal vez con algunos cambios pueda adecuarse, pero hay que tener presente que el hecho de tratar de modificar el código existente puede volverse complejo, a tal punto que en vez de tener un proyecto a desarrollar, éstos se conviertan en varios.

El hecho de no poder reutilizar el código existente es costoso, ya que invertimos tiempo escribiendo lineas para realizar acciones que se repiten una y otra vez cada vez que desarrollamos un proyecto.

En esta Unidad vamos a estudiar el soporte con que cuenta el C++ para la reutilización del código existente, las clases pueden reutilizarse de varias maneras.

* **COMPOSICION**. Es la creación de objetos miembro de otra clase dentro de una nueva clase, para ello nos basamos en que un objeto puede contener una colección de otros objetos. Entonces decimos; que un objeto puede estar compuesto de otros objetos.

* **HERENCIA O DERIVACION**. Un objeto puede ser también parte de otro objeto. De la misma forma que una criatura hereda características de sus padres, una clase derivada (subclase), hereda características de una clase base (superclase).

En esta unidad se analiza la construcción de una nueva clase a partir de una anterior; usando composición o herencia. La herencia, sin embargo, también crea otros tipos de relación. Así cuando se heredan en una nueva clase características de una clase anterior, la nueva clase no toma exactamente todas las características de la anterior, Así instancias de la nueva clase son también instancias de la clase anterior, ésto es, cualquier función u operaciones que trabajen sobre un objetos de la clase anterior, van a trabajar sobre objetos de la nueva clase. Todas las subclases de una clase base tienen el mismo conjunto de funciones, o interface, de la clase base. Este concepto es muy importante en el contexto de la herencia.

Iniciamos la unidad presentando la sintaxis para herencia y composición, avanzamos presentando varios ejemplos utilizando estas técnicas.

2.- Sintaxis para Composición y Herencia.

Para usar las clases en C++, se deberá contar con un archivo de cabecera él cual contendrá la definición de la clase y por lo menos, una archivo de código objeto o una librería con los archivos en código objeto. Coincidentemente, es también lo que se necesita para derivar una nueva clase de una anterior, no es necesario el código fuente de los metodos (aunque puede ser útil). En el pasado los vendedores de librerias de C, daban la opción de proporcionar también los códigos fuente. Con C++, cualquier librería puede ser facilmente adaptada, dispongase o no de los codigos fuente.

En la definición de una nueva clase debemos considerar dos puntos importantes.

1) La manera de declarar la nueva clase.
2) La manera de especificar la llamada a los constructores de la nueva clase.

2.a) **Composición**. Reutilización de código, mediante objetos miembro.

Para reutilizar código empleando **composición**, solo es necesario declarar un objeto de una clase como miembro de una nueva clase.

La creación de un objeto como un miembro de otra clase, es diferente a la creación de un objeto ordinario. Sin embargo, recuerde una de las principales características de C++, la garantía de inicialización adecuada de todos los objetos. Cuando se emplea composición o herencia, este hecho es válido para todos los objetos o subobjetos, cada uno deberá se inicializado de manera adecuada, el compilador nos garantiza ésto, llamando al constructor respectivo para cada uno de los objetos.

En un objeto ordinario, el constructor es llamado en el punto donde es definido el objeto, y podemos dar una lista de argumentos luego del identificador. Si no se dá una lista de argumentos para un objeto ordinario, se llama al constructor sin argumentos o usando los argumentos de default.

En el caso de manejar un objeto miembro, NUNCA se dá una lista de argumentos en el punto de la declaración del objeto dentro de la estructura de la clase. Cuando se declara un objeto miembro, se llama al compilador para que reserve memoria para éste dentro de cualquier instancia de la nueva clase. Sin embargo, el constructor no puede ser llamado hasta que se asigna espacio, lo que ocurre hasta que se crea una instancia de la nueva clase y el constructor de la nueva clase es llamado.

Para controlar la construcción de todos los subobjetos, se usa una sintaxis especial en el nuevo constructor, llamado **lista de inicialización** del constructor.

Cuando se entra al cuerpo del constructor de la nueva clase, ya se deberán haber inicializado todos los objeto miembro, por lo tanto, los constructores de todos los

objeto miembro deberán ser llamados **antes** de llamar al constructor de la nueva clase, entonces, la lista de inicialización de los objetos miembro, deberá estar antes de entrar al cuerpo del constructor de la nueva clase. Para declarar los constructores de los objetos miembro, lo hacemos después de la lista de argumentos del constructor de la nueva clase, poniendo (:), y a continuación, el nombre de cada objeto miembro, seguido de su lista de inicialización, como lo hacemos en la declaración e inicialización de un objeto simple.

Sintaxis.

```
class uno {
//...
        class_objeto obj_miem;
//...
public:
        uno(lista arg1);        // Constructor
        //...
};

uno::uno(lista arg1) : obj_miem(lista_arg2) {
        // cuerpo del constructor
}
```

Donde **lista_arg2** es la lista de argumentos para el constructor del objeto miembro, recuerde que puede o no haber una lista de argumentos.

Cuando la clase consta de varios objetos miembro, cada uno se separa por una coma en la **lista de inicialización** del constructor. Los constructores son llamados en el orden en que aparecen los objetos en la declaración de la clase.

Es posible también crear arreglos de objetos miembro dentro de una clase. (Recuerde que cuando se manejan arreglos de objetos, se usa un constructor sin argumentos para la inicialización, por lo tanto, no es necesario especificar ninguna inicialización en el constructor de la clase derivada).

2.b) Sintaxis de herencia.

Hay dos razones para usar herencia:

1) Si se tiene una clase, y se quiere usar, sin embargo no se quiere re-escribir. Se puede solo usar lo que se necesita e ignorar el resto, no es necesario tampoco saber como trabaja. Reusando código se puede programar mas rapidamente.

2) Nos permite expresar un problema como una jerarquía de clases. Donde la clase base es común a todas las clases derivadas. Las subclases pueden ser manipuladas por medio de esta interface común. Un programa bien diseñado puede ser ampliado simplemente derivando una nueva clase a partir de la clase base.

Cuando se hereda una nueva clase de una clase ya existente, de derecha a izquierda, se indica el nombre de la clase que se hereda, luego el nombre de la nueva clase, separando ambos por (:).

Sintaxis.

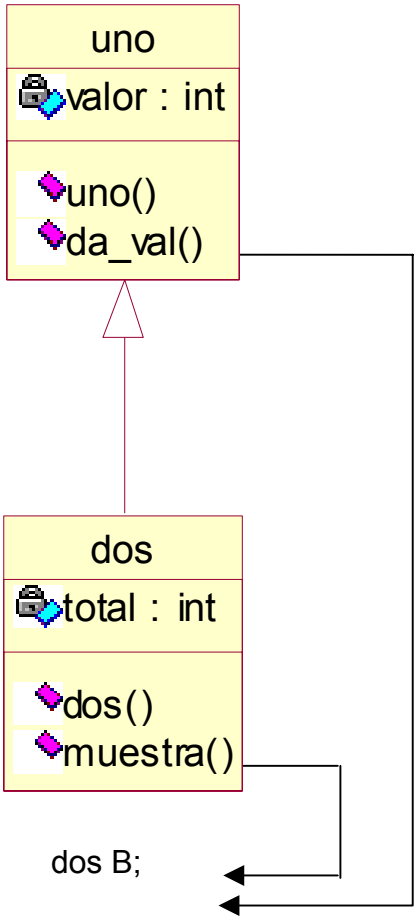
```
class nomb_class_deriv : nomb_class_base {
// cuerpo de la declaración de clase
};
```

Si recordamos, los miembros de una clase por default son **private** mientras no se indique explicitamente **public**. Esto también se cumple cuando se hereda; una clase base es **private** mientras no se declare de manera explicita **public**. En la declaración anterior, todos los miembros de la clase base son **private**, para objetos de la clase derivada, (**nomb_class_deriv**). Si se desea que los miembros **public** de la clase base puedan ser utilizados por objetos de la clase derivada, la clase base deberá declararse como **public**. Veamos la sintaxis.

```
class nomb_class_deriv : public nomb_class_base {
// cuerpo de la declaracióm de clase
};
```

Cuando se maneja herencia, tanto los datos miembro como las funciones miembro de la clase base son colocadas en la clase derivada, en esencia, la estructura de la clase base es copiada en la estructura de clase derivada. Se pueden agregar a la clase derivada tanto datos miembro como funciones miembro. Dentro de las funciones miembro de la clase derivada, se pueden llamar funciones miembro **public** de la clase base y manipular también datos miembro **public** o **protected**.

Veamos algunos ejemplos que nos aclaren el comportamiento de las palabras; **private**, **protected** y **public**.



Ejemplo # 1.

```
// Programa # 75A, análisis de los conceptos de:
// public private y protected.
#include <iostream.h>




class uno {
    int valor;
public:
    uno(int i) { valor = i;}
    int da_val() { return valor; }
```

```
};

class dos : public uno {
    int total;
public:
    dos(int c, int d) : uno(c) { total = d; }
    void muestra() {
        cout << "Clase dos:" << endl
             << "Valor = " << da_val() << endl
             << "Total = " << total << endl;
    }
};



void main()
{
    uno A(2);
    dos B(3, 4);

    cout << "Desde main() clase uno: valor = "
         << A.da_val() << endl;
    cout << "Desde main() clase dos: valor = "
         << B.da_val() << endl;
    B.muestra();
}
```

OBSERVACIONES.

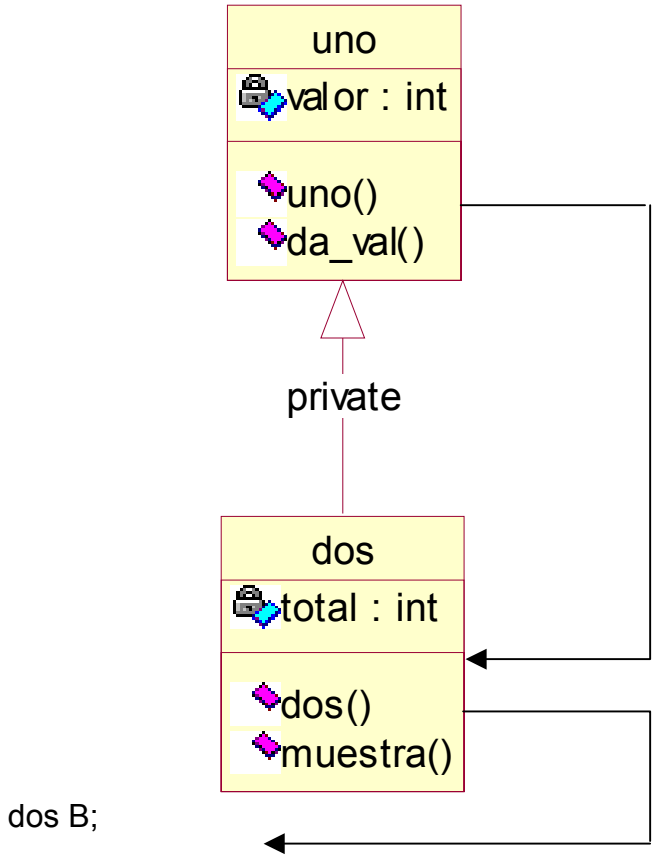
Es el caso mas frecuentemente utilizado, en donde se declara que los datos miembro de la clase base, serán privados en la clase derivada, ésto es, no se tendrá acceso a ellos. Y sus datos publicos serán públicos en la clase derivada.

Para tener acceso desde la clase derivada a datos miembro privados de la clase base, es necesario implementar una función que nos regrese el valor del dato miembro **valor**. En este caso, la función es, da_val().

La salida nos resulta:

```
Desde main() clase uno: valor = 2
Desde main() clase dos: valor = 3
Clase dos:
Valor = 3
```

```
Total = 4
```



Ekemplo # 2.

```
// Programa # 75B, análisis de los conceptos de:
// public private y protected.
// ESTE PROGRAMA NO COMPILA.
#include <iostream.h>

class uno {
    int valor;
public:
    uno(int i) { valor = i;}
    int da_val() { return valor; }
};
```

```
class dos : uno {
    int total;
public:
    dos(int c, int d) : uno(c) { total = d; }
    void muestra() {
        cout << "Clase dos:" << endl
             << "Valor = " << da_val() << endl
             << "Total = " << total << endl;
    }
};

void main()
{
    uno A(2);
    dos B(3, 4);

    cout << "Desde main() clase uno: valor = "
         << A.da_val() << endl;
    // No hay acceso a la función da_val(), desde B
    cout << "Desde main() clase dos: valor = "
         << B.da_val() << endl;
    B.muestra();
}
```

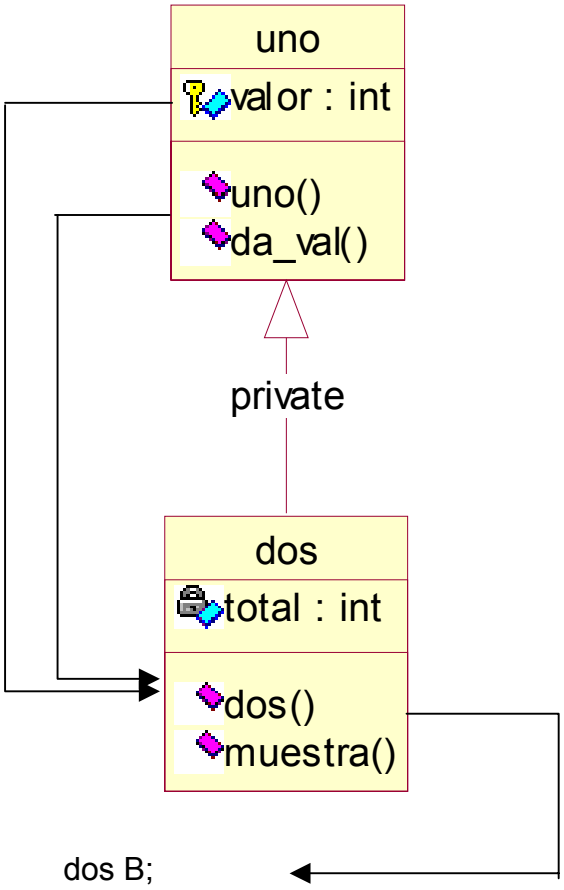
OBSERVACIONES.

Para empezar, el programa genera un error en la línea:

```
cout << "Desde main() clase dos: valor = "
        << B.da_val() << endl;
```

nos dice que no hay acceso a la función da_val(), lo cual se debe a que la clase base está siendo declarada como **private** en la clase derivada, por lo cual todos los miembros de la clase base, serán privados para objetos de la clase derivada, no así para objetos de la clase base.

Podemos observar también que los miembros públicos si pueden ser utilizados por funciones de la clase derivada, pero no por objetos de la clase derivada.



Ejemplo # 3.

```
// Programa # 75C, análisis de los conceptos de:
// public private y protected.
#include <iostream.h>

class uno {
protected:
    int valor;
public:
    uno(int i) { valor = i;}
    int da_val() { return valor; }
```

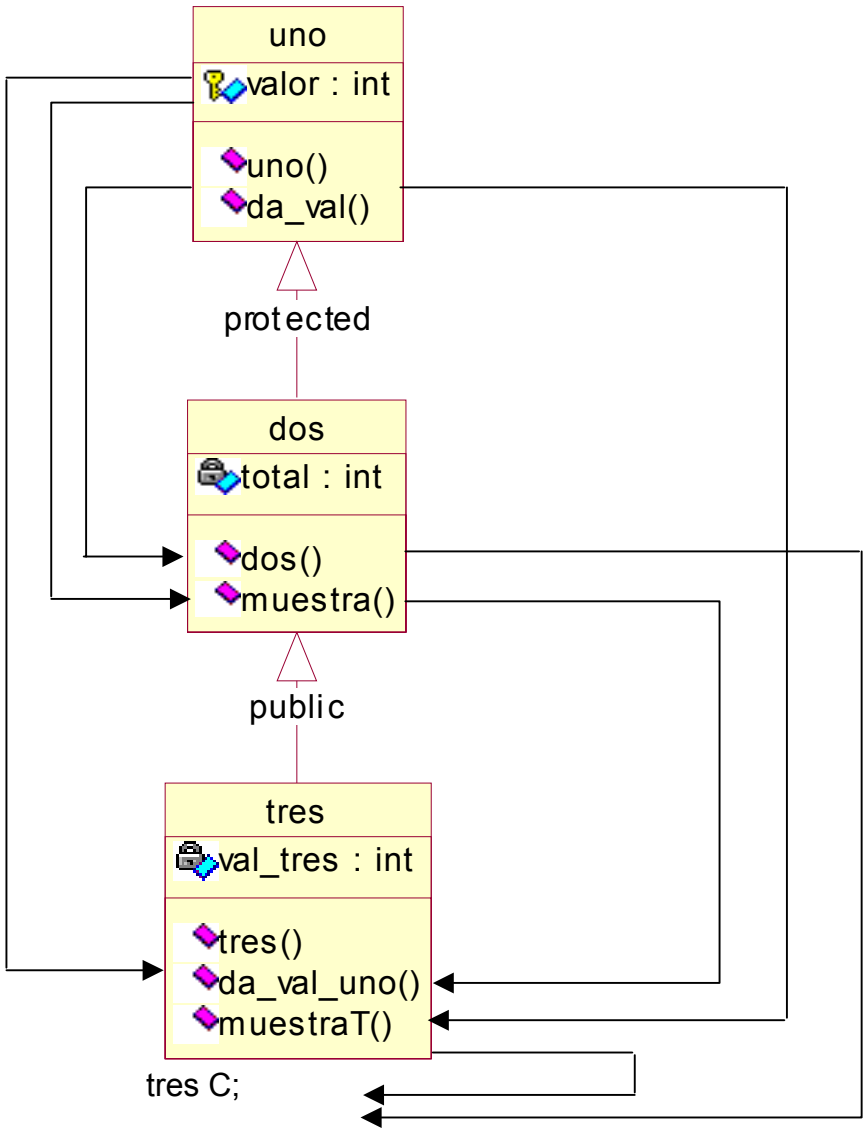
```
};

class dos : uno {
    int total;
public:
    dos(int c, int d) : uno(c) { total = d; }
    void muestra() {
        cout << "Clase dos:" << endl
             << "Valor = " << valor << endl
             << "Total = " << total << endl;
    }
    int da_valor() { return valor; }
};

void main()
{
    uno A(2);
    dos B(3, 4);

    cout << "Desde main() clase uno: valor = "
         << A.da_val() << endl;
    // para tener acceso a valor desde B es necesario definir
    // una función en uno, en este caso; da_valor()
    cout << "Desde main() clase dos: valor = "
         << B.da_valor() << endl;
    B.muestra();
}
```

OBSERVACIONES.

En este ejemplo, seguimos manteniendo a la clase base como **private** en la clase derivada, pero ahora declaramos al dato miembro de la clase base como **protected**, lo que nos va a permitir tener acceso a él, pero solo desde funciones de la clase derivada. Todo lo anterior se sigue manteniendo válido.

La salida nos resulta:

```
Desde main() clase uno: valor = 2
Desde main() clase dos: valor = 3
Clase dos:
Valor = 3
Total = 4
```



Ejemplo # 4.

```
// Programa # 75D, análisis de los conceptos de:
// public private y protected.
#include <iostream.h>

class uno {
protected:
    int valor;
public:
```

```
    uno(int i) { valor = i;}
    int da_val() { return valor; }
};

// protected permite la propagación de protected desde la
// clase uno hacia las clases derivadas.
class dos : protected uno {
    int total;
public:
    dos(int c, int d) : uno(c) { total = d; }
    void muestraD() {
        cout << "Clase dos:" << endl
             << "Valor = " << valor << endl
             << "Total = " << total << endl;
    }
    int da_valor() { return valor; }
};

class tres : public dos {
public:
    tres(int a, int b) : dos(a, b) {}
    int da_val_uno() { return valor; }
    void muestraT() {
        cout << "Clase Tres:" << endl
             << "Valor = " << valor << endl;
        }
    // Se elimina el dato miembro total ya no es accesible
    // desde esta clase. Es privado de la clase dos.
};


void main()
{
    uno A(2);
    dos B(3, 4);
    tres C(5, 6);

    cout << "Desde main() clase uno:  valor = "
         << A.da_val() << endl;
    cout << "Desde main() clase dos:  valor = "
         << B.da_valor() << endl;
    cout << "Desde main() clase tres: valor = "
         << C.da_val_uno() << endl;
    B.muestraD();
    C.muestraD();
    C.muestraT();
}
```

OBSERVACIONES.

Para ejemplificar el uso de la palabra **protected** en el ejemplo anterior declaramos una nueva clase derivada (clase tres), desde la cual queremos accesar directamente el dato miembro de la clase uno (valor), valor es declarado **protected** en la clase uno, el mismo atributo es pasado a la clase dos, y al declarar, **public** la clase dos, en la clase tres, los atributos dados a la clase uno y transferidos a la clase dos, son utilizados en la clase tres, por lo tanto, hay acceso desde esta clase al dato miembro valor, de la clase uno.

La salida nos resulta:

```
Desde main() clase uno:  valor = 2
Desde main() clase dos:  valor = 3
Desde main() clase tres: valor = 5
Clase dos:
Valor = 3
Total = 4
Clase dos:
Valor = 5
Total = 6
Clase Tres:
Valor = 5
```

Hay algunas funciones miembro que no son heredadas de manera automática en la clase derivada, estos son: los constructores, los destructores, y el operador sobrecargado **operator=()**.

Si se necesita un destructor, éste deberá crearse para cada nueva clase derivada, llamandose de manera automática el destructor de la clase base cuando el objeto sale de ámbito. Como solo hay un destructor para cada clase, no existe ambigüedad (ni argumentos), y el compilador puede de manera automática llamar a todos los destructores de todas las clases base. Note que todos los destructores de todas las clases base son llamados automaticamente por el compilador para garantizar el limpiado adecuado de cada parte de un objeto.

El orden de llamada de los destructores es desde el de la clase derivada hacia el de la clase base.

Los destructores de los objeto miembro se llaman en el orden inverso a como fueron declarados los objetos en la nueva clase.

En el caso de los constructores hay diferencia, ya que éstos pueden ser sobrecargados y tienen argumentos. De la misma forma que en el caso de composición, se usa la **lista de inicialización** del constructor para llevar a cabo la llamada al constructor de la clase base. Esto es, la sintaxis para herencia se especifica iniciando con (:) y terminando con la (**{**) que abre el cuerpo del constructor.

Sintaxis.

```
momb_class_deriv::momb_class_deriv(lista_arg1)
                                : nomb_class_base(lista_arg2) {

        //cuerpo del constructor de la clase
}
```

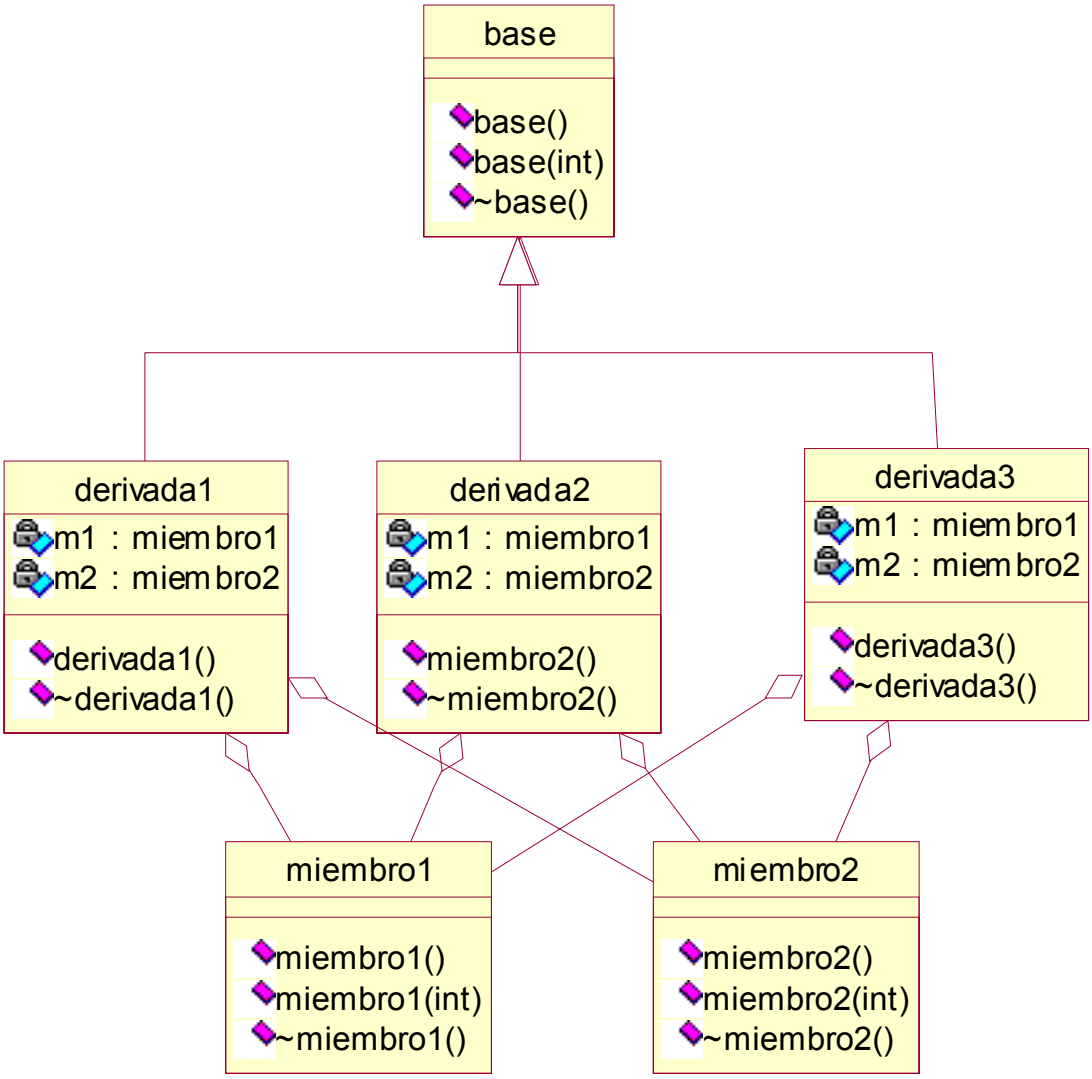
Donde **lista_arg1** es la lista de argumentos para el constructor de la clase derivada y **lista_arg2** es la lista de argumentos para el constructor de la clase base.

Si el constructor de la clase base es sobrecargado, el constructor a ser llamado va a depender de la lista de argumentos, de igual forma que la llamada a cualquier constructor. Si no se realiza una llamada explicita al constructor de la clase base en el constructor de la clase derivada, entonces, se llama al constructor de la clase base que no tiene argumentos.

Si la clase derivada tiene objetos miembro, se deberán llamar sus constructores también dentro de la **lista de inicialización** de constructores. La lista de llamadas puede ser separada por comas y puede escribirse en cualquier orden. La llamada a los constructores de los objetos miembro ocurre siempre después de la llamada a los constructores de las clases base, siguiendo el orden en el que los objeto miembro son declarados en la clase.

En el siguiente ejemplo se muestra como constructores y destructores son llamados en clases derivadas. Todas las clases en el ejemplo imprimen un mensaje cuando su constructor y destructor es llamado, de tal manera de poder apreciar cuales y en que tiempo son llamados. El archivo inicia con una macro del preprocesador que, da un nombre, crea una clase con "Mensaje Constructor" y "Mensaje Destructor". Existen dos constructores, de tal manera de poder apreciar la diferencia entre una llamada explicita a un constructor (constructor con un argumento) y una llamada implicita a un constructor (constructor sin argumentos).

Se definen tres clases; **base**, **miembro1** y **miembro2**. Se construyen entonces tres clases derivadas, con **base** usada como una clase base y **miembro1** y **miembro2** usadas como objetos miembro. En la primera, **derivada1** se llama de manera explicita los constructores para la clase base y los objetos miembro, usando argumentos. En la segunda clase **derivada2** no se realizan llamadas a los constructores, ni al de la clase base ni al de los objetos miembro. En la tercera clase **derivada3**, se realizan las llamadas a los constructores de manera intencional en cualquier orden, de tal forma de ver su comportamiento.




```
// Programa 75E, Llamada a constructores y destructores
// de clase base, clase miembro y clase derivada.
#include <iostream.h>

// macro para crear una clase

#define cldecl(NCLASE)  class NCLASE { \
    int i; \
public: \
    NCLASE() {  \
        cout << #NCLASE "() Llamada al constructor" << endl;    \
    }   \
    NCLASE(int)  {  \
        cout << #NCLASE "(int) Llamada al constructor" << endl; \
```

```
    }   \
    ~NCLASE()    {  \
        cout << #NCLASE " Llamada al destructor" << endl;  \
    }   \
}

cldecl (base);
cldecl (miembro1);
cldecl (miembro2);

// La forma estandar
class derivada1 : public base {
    miembro1 m1;
    miembro2 m2;
public:
    derivada1() : base(), m1(1), m2 (2) {
        cout << "Llamada al constructor de derivada1" << endl;
    }
    ~derivada1() {
        cout << "Llamada al destructor de derivada1" << endl;
    }
};



// Llamada implicita al constructor
class derivada2 : public base {
    miembro1 m1;
    miembro2 m2;
public:
    derivada2() {
        cout << "Llamada al constructor de derivada2" << endl;
    }
    ~derivada2() {
        cout << "Llamada al destructor de derivada2" << endl;
    }
};



// Llamada en desorden a los constructores.
class derivada3 : public base {
    miembro1 m1;
    miembro2 m2;
public:
    derivada3() : m2(2), m1(1), base(3) {
        cout << "Llamada al constructor de derivada3" << endl;
    }
    ~derivada3() {
```

```
        cout << "Llamada al destructor de derivada3" << endl;
    }
};


void main()
{
    {
        cout << "\n\t derivada1 X =:" << endl;
        derivada1 X;
    }
    {
        cout << "\n\t derivada2 X =:" << endl;
        derivada2 X;
    }
    {
        cout << "\n\t derivada3 X =:" << endl;
        derivada3 X;
    }
}
```

OBSERVACIONES.

La salida nos resulta:

```
      derivada1 X =:
base() Llamada al constructor
miembro1(int) Llamada al constructor
miembro2(int) Llamada al constructor
Llamada al constructor de derivada1
Llamada al destructor de derivada1
miembro2 Llamada al destructor
miembro1 Llamada al destructor
base Llamada al destructor

      derivada2 X =:
base() Llamada al constructor
miembro1() Llamada al constructor
miembro2() Llamada al constructor
Llamada al constructor de derivada2
Llamada al destructor de derivada2
miembro2 Llamada al destructor
miembro1 Llamada al destructor
base Llamada al destructor

      derivada3 X =:
base(int) Llamada al constructor
```

```
miembro1(int) Llamada al constructor
miembro2(int) Llamada al constructor
Llamada al constructor de derivada3
Llamada al destructor de derivada3
miembro2 Llamada al destructor
miembro1 Llamada al destructor
base Llamada al destructor
```

En la prueba una instancia de cada clase es puesta dentro de su propio ámbito de tal manera de poder observar de forma aislada la llamada a los constructores y destructores de cada objeto de la clase derivada respectiva.

Note que en la macro usada para la definición de la clase, en un contructor se usa, como argumento solo el tipo (int), esto le permite al compialdor distinguir a un constructor sin argumentos de uno con argumentos.

Lo que podemos observar en los tres casos es que siempre se sigue el mismo orden, sean inicializados los miembros y llamados los constructores, no sean inicializados ni llamados explicitamente los constructores, o se escriba la lista de inicialización de constructores en un orden arbitrario.

También se puede apreciar la llamada de los destructores en orden inverso a la llamada de los constructores.

2.c) Composición con punteros a objetos.

Al crear una instancia de un objeto como un miembro de otro objeto, el programador puede decidir la composición de la clase, ya sea en tiempo de compilación o en tiempo de ejecución, el último caso es más interesante, se puede llevar a cabo mediante un puntero a un objeto en vez de un objeto. A esta forma de trabajo se le llama **pluggable pointer composition**, ya que el objeto es definido en tiempo de ejecución, lo que le da flexibilidad al sistema.

Veamos un ejemplo, para ello utilicemos parte del programa anterior.

```
// Programa 76, Llamada a constructores y destructores
// de clase base, clase miembro y clase derivada.
// Y Composición con punteros a objetos.

#include <iostream.h>

// macro para crear una clase

#define cldecl(NCLASE)  class NCLASE { \
    int i; \
```

```
public: \
    NCLASE() {  \
        cout << #NCLASE "() Llamada al constructor" << endl;   \
    }   \
    NCLASE(int)  {  \
        cout << #NCLASE "(int) Llamada al constructor" << endl; \
    }   \
    ~NCLASE()    {  \
        cout << #NCLASE " Llamada al destructor" << endl;  \
    }   \
}

cldecl (base);
cldecl (miembro1);
cldecl (miembro2);


class derivada1 : public base {
    miembro1 m1;
    miembro2 m2;
public:
    derivada1() : base(), m1(1), m2 (2) {
        cout << "Llamada al constructor de derivada1" << endl;
    }
    ~derivada1() {
        cout << "Llamada al destructor de derivada1" << endl;
    }
};

// Llamada implicita al constructor
class derivada2 : public base {
    miembro1 m1;
    miembro2 m2;
public:
    derivada2() {
        cout << "Llamada al constructor de derivada2" << endl;
    }
    ~derivada2() {
        cout << "Llamada al destructor de derivada2" << endl;
    }
};

// Llamada en desorden a los constructores.
class derivada3 : public base {
    miembro1 m1;
    miembro2 m2;
public:
    derivada3() : m2(2), m1(1), base(3) {
        cout << "Llamada al constructor de derivada3" << endl;
```

```
    }
    ~derivada3() {
        cout << "Llamada al destructor de derivada3" << endl;
    }
};

class compuesta {
    base *ptr;
public:
    compuesta(base *puntero) {
        ptr = puntero;
        cout << "Llamada a constructor de clase compuesta"
             << endl;
    }
    ~compuesta() {
        cout << "Llamada al destructor de clase compuesta"
             << endl;
    }
};

void main()
{
    cout << "\nDefiniendo punteros a objetos" << endl;
    derivada1 X;
    derivada2 Y;
    derivada3 Z;
    compuesta A1(&X), A2(&Y), A3(&Z);
}
```

OBSERVACIONES.

En la función main(), el objeto A1 es creado con un tipo de clase **derivada1**, A2 es creado con un tipo de clase **derivada2**, etc. Esto funciona ya que un objeto de una clase derivada es también un miembro de la clase base, entonces un puntero a un objeto de una clase derivada puede ser substituido en cualquier lugar por un puntero a un objeto de una clase base, si ésto es necesario

La corrida del programa nos resulta:

```
Definiendo punteros a objetos

base() Llamada al constructor
miembro1(int) Llamada al constructor
miembro2(int) Llamada al constructor
Llamada al constructor de derivada1
base() Llamada al constructor
miembro1() Llamada al constructor
miembro2() Llamada al constructor
```

```
Llamada al constructor de derivada2
base(int) Llamada al constructor
miembro1(int) Llamada al constructor
miembro2(int) Llamada al constructor
Llamada al constructor de derivada3
Llamada a constructor de clase compuesta
Llamada a constructor de clase compuesta
Llamada a constructor de clase compuesta
Llamada al destructor de clase compuesta
Llamada al destructor de clase compuesta
Llamada al destructor de clase compuesta
Llamada al destructor de derivada3
miembro2 Llamada al destructor
miembro1 Llamada al destructor
base Llamada al destructor
Llamada al destructor de derivada2
miembro2 Llamada al destructor
miembro1 Llamada al destructor
base Llamada al destructor
Llamada al destructor de derivada1
miembro2 Llamada al destructor
miembro1 Llamada al destructor
base Llamada al destructor
```

De nuevo podemos observar el orden inverso en que se van llamando los destructores, cuando se sale el objeto de ámbito.

3.- **Ejemplos de Composición y Herencia.**

3.1) Una clase para manejo de errores.

La necesidad de manejar errores cuando se construye una clase es muy común. Cuando ocurre un error sobre un objeto, es conveniente llamar a una función que imprima en la pantalla un mensaje, y luego a otra que nos recupere del error o aborte el programa.

La siguiente clase, a la cual se le llamó **error_handler**, se usa como un objeto miembro de otra clase. El argumento en el constructor es una cadena, la cual se va a utilizar para todos los mensajes resultantes de la nueva clase, en la cual se define una variable static **error**. Los metodos, **mensaje**() y **termina**(), tienen sus argumentos de la misma forma que la función printf(), de tal manera de poder manejar cualquier número de mensajes cuando ocurre un error.

Veamos la interface.

```
// Programa 77.H. Interface de una clase para manejo de ERRORES
// Clases. Esta clase nos muestra el reuso de código
// via un objeto miembro
// clase_msg, describe la clase error_handler
// termina(), tiene un número variable de argumentos.
#ifndef ERROR_H_
#define ERROR_H_

class error_handler {
    char *msg;
public:
    error_handler(char *clase_msg) : msg(clase_msg) {}
    void mensaje(char *cadena, ...);
    void termina(char *cadena, ...);
};


#endif // ERROR_H_
```

El constructor simplemente hace la asignación de la dirección de la cadena pasada en **clase_msg** a la variable puntero **msg**, lo cual tiene lugar en la lista de inicialización del constructor, donde los constructores para todas las clases base y los objeto miembro, son normalmente llamados, sin embargo, note que **char *** es un tipo predefinido, por lo que no llama a un constructor. Esto es cierto, sin embargo, solo para consistencia, se puede suponer que si tiene un constructor, y la llamada a él es simplemente una asignación.

Es una buena práctica inicializar todos los objeto miembro, ya sean estos del tipo predefinido, o definidos por el usuario, en la lista de inicialización. Esto nos asegura que en el cuerpo de la función, ya todos los objetos han sido inicializados.

Las funciones **mensaje()** y **termina()** imprimen un mensaje a la salida estandar de error usando **stderr** de C, una vez hecho ésto, se continúa con el proceso o aborta al programa usando la función de C, **exit()**.

Recuerde que para usar una lista de número variable de argumentos, se deberán usar macros definidos en el archivo de cabecera **stdarg.h**.

Veamos la implementación.

```
// Programa # PLCP77.CPP
// La implementación de la lista de número variable de
// argumentos, se implementa usando la librería de ANSI C
// stdarg.h.
```

```
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include "PLCP77.H"

void error_handler::mensaje(char *cadena, ...) {
    va_list arg_ptr;
    va_start(arg_ptr, cadena);
    fprintf(stderr, "%s error: ", msg); // mensaje
    vfprintf(stderr, cadena, arg_ptr);
    va_end(arg_ptr);
}

void error_handler::termina(char *cadena, ...) {
    va_list arg_ptr;
    va_start(arg_ptr, cadena);
    fprintf(stderr, "%s error: ", msg); // mensaje
    vfprintf(stderr, cadena, arg_ptr);
    va_end(arg_ptr);
    exit(1);
}
```

El **arg_ptr** es un tipo especial de puntero mediante el cual podemos hacer indexación a través de la lista de argumentos. Cuando se llama a la macro **va_start()**, inicializa el puntero, para posteriormente ser manejado por la función **vfprintf()**, mediante la cual, podemos seleccionar todos los argumentos de la lista. Antes de salir de la función llamamos al macro **va_end()**, él cual resetea el puntero **arg_ptr**, esta acción es importante, ya que restaura el stack a la posición en que se encontraba antes de llamar a la función, en este caso **mensaje()** y **termina()**.

A continuación se presenta una aplicación simple de la clase **error_handler**.

```
// Programa PLCP77A.CPP
// Aplicación de la clase error_handler
#include "PLCP77.H"

class prueba {
    static error_handler error;
public:
    void demostracion() {
        error.termina("En función %s, valor %d\n",
                      "demostracion", 1);
    }
};

// Inicializa el objeto miembro static;
error_handler prueba::error("prueba");
```

```
void main() {
    prueba E;
    E.demostracion();
}
```

OBSERVACIONES.

Note que el objeto miembro del tipo **error_handler**, dentro de la clase **prueba** es static, con lo cual solo se va a crear un objeto de manejo de error, él cual va a ser compartido por todos los objetos de la clase, en este caso, **prueba**, con ésto también se ahorra memoria ya que no se crea memoria para el objeto **error**, cada vez que se declara un objeto de la clase **prueba**.

La definición del objeto **static** error, reserva memoria para el objeto solo una vez, llamando al constructor del objeto antes que **main()** inicie. Su destructor es llamado después de salir de **main()**.

Sintaxis para la función **vfprintf()**.

Descripción: La función **vfprintf()**, es una alternativa de la función **printf()**, la diferencia estriba en que **printf()** acepta una lista de argumentos y **vfprintf()**, acepta un puntero a una lista de argumentos.
vfprintf() acepta un puntero a una serie de argumentos, aplicando a cada argumento un formato especificado en la cadena **format**.

Sintaxis:
```
#include <stdio.h>
int vfprintf(FILE *stream,
        const char *format, va_list arglist);
```

Prototipo: **stdio.h**

V. regresado: Regresa el número de bytes enviados, si hay un error, regresa EOF.

3.1.a) Una aproximación a la clase iostream.

La aproximación a printf() para el manejo de errores, trabaja bien, es especialmente útil cuando los usuarios están más familiarizados con C que con C++.

El uso de funciones que pueden recibir un número variable de argumentos, los cuales también pueden ser de diferentes tipos, trae problemas ya que el compilador suspende la verificación de tipos para estos argumentos, adicional a ésto, si no se escribió correctamente la lista de argumentos, nos puede imprimir basura.

La verificación de tipos en E/S es una de las razones de más peso que llevaron al desarrollo de las librerias de E/S de C++.

Entonces usando estas librerias podemos escribir:

cout << "hola muchachas, son" << 5 << ctime(&x) << endl;

El objeto **cout** es análogo a **stdout** en C y el operador sobrecargado **<<** significa "pone en", cuando se usa con objetos del tipo iostream. Cada expresión entre **<<** es evaluada de manera separada.

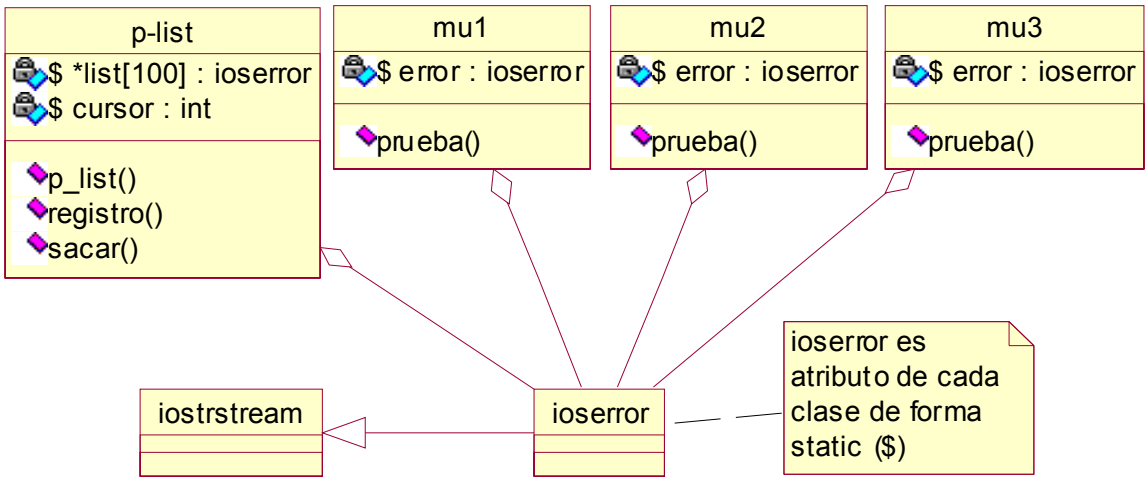
En la sentencia anterior hay varios item, y son diferentes; hay una cadena, una función **ctime()** (de ANSI C), la cual regresa una cadena, un entero (5) y un manipulador **endl**. Lo anterior ocasiona que el operador **<<** sobrecargado haga diferentes llamadas a función dependiendo del tipo de argumento que recibe.

Es posible crear nuestras funciones propias cuando trabajamos con una clase, de tal manera de poder enviar los objetos creados a la clase iostream. Uno de los atractivos principales de iostream es que el compilador determina el uso adecuado de los argumentos y cual función es llamada. Esto significa que no solamente se obtiene verificación de errores en tiempo de compilación, sino también una mayor eficiencia que la lograda con printf(). Cuando se llama a la función printf(), la cadena formateada y los argumentos, deberán ser pasados por la función en tiempo de ejecuación, ésto es lento. Debido a que los argumentos en el caso de iostream son pasados en tiempo de compilación, el trabajo es realizado por el compilador en vez de por la función de impresión, por lo tanto, la función es más pequeña y se ejecuta más rápido.

Además de lo anterior, la sintaxis de una expresión iostream es más fácil de leer, se puede ver en el stream el orden en que van a ser impresos.

3.1.b) Manejando iostream con herencia.

En el siguiente programa, se usa el estilo de iostream para reportar errores. Veamos el árbol de herencia.



NOTA: Como es contumbre, los datos miembro static de las clases son inicializados fuera de cualquier clase.

En la clase p_list, sus datos y sus funciones miembro registro() y saca() son static, también lo son los datos miembro error de mu1 y m2 por lo que se deben inicializar fuera de toda clase y función. La razón de su definición del tipo static, permite que solo exista un manejador de error por clase para todos los objetos de una misma clase dada.

La clase ostrstream se encuentra definida en el archivo de cabecera strstream.h, tiene 2 constructores de los cuales estamos usando el siguiente:

ostrstream(char *, int, int = ios::out);

Este constructor escribe la cadena dada en el argumento (char *) al heap, usa por default **out**, la cual es una constante definida en ios, para la apertura de un archivo en modo de escritura.

```
// Manejo de errores usando iostream.
// Instalación de objetos static de esta clase
// en otras clases, y colocando un framework
// dentro de la clase, el cual podrá ser modificado
// Manejo de errores

#ifndef IOSERROR_H_
#define IOSERROR_H_
#include <iostream.h>
#include <strstream.h>


// Este manipulador inserta una nueva línea en el
// stream de salida.


inline ostream& nl(ostream& os) {return os << "\n"; }

class ioserror : public ostrstream {
    enum { tam = 300 };
    char msg[tam];       // Buffer para almacenar mensajes
    char *nclase;
public:
    ioserror(char *nom_clase);

    void saca() {
        if (*msg) { // saca solo si hay mensajes.
```

```
            clog << nl << nclase << " error: " << msg << endl;
        }
    }
};


// Manipulador para imprimir todos los mensajes de error

ostream &termina(ostream& os);

#endif // IOSERROR_H_
```

La clase **ioserror** es una clase derivada de **ostrstream**, la cual representa un tipo especial de iostream, parecido a la función de ANSI C, **sprintf()**, la cual realiza un **printf()** a una área de memoria, en vez de hacerlo a la salida estandar.

La clase **ioserror** guarda todos los mensajes en el buffer **msg**, y como es derivada de ostrstream, cuenta con todas sus propiedades, incluyendo las funciones para manejo de enteros, reales, cadenas, etc.

En la clase **ioserror** se puede ver el uso de **enum** para la simulación de una **const** local. En el caso de un **const** dentro de una clase, **const** crea una instancia local, la cual deberá ser inicializada en la lista de inicialización del constructor y no puede ser cambiada durante la vida del objeto. Sin embargo, cada objeto de la clase puede tener un valor diferente para este tipo de **const**, por lo que **const** requiere espacio de almacenamiento.

En el **const** que usamos, definido mediante **enum,** el compilador no lo puede emplear en una expresión, solo lo puede usar en la definición de un arreglo.

Para crear un valor constante él cual es local de una clase, no se requiere ninguna clase de almacenamiento y puede ser usado en la definición de arreglos, por eso se prefiere esta técnica de definición mediante un enumerado.

3.1.c) Creando manipuladores.

En el caso de los manipuladores, no se hace una llamada a una función, con paréntesis, para llamarlo, simplemente se da el nombre del manipulador. Este proceso trabaja ya que el compilador ve un puntero a una función en la lista de iostream y recuerda que existe un **operator<<()** sobrecargado él cual reconoce como argumento un apuntador a función. Este operador sobrecargado se le llama **applicator**, solo acepta un puntero a función él cual tiene como argumento **ostream&** y regresa un **ostream&** (en el caso de manipuladores para istream, éstos aceptan un argumento **istream** y regresan valores). Lo que hace el **applicator** es llamar a la función y regresar el **ostream**.

En el programa se implementan 2 manipuladores:

* El primero, es **nl,** manda a la pantalla el caracter nueva línea.

```
inline ostream& nl(ostream& os) {return os << "\n"; }
```

Note que los manipuladores no pueden ser funciones miembro, por lo tanto, se pueden aplicar a cualquier iostream.

* El segundo, es **termina**, el cual al ser llamado imprime todos los mensajes de error y el programa es terminado.

La clase ioserror puede ser usada de varias formas, lo que afecta al diseño del manipulador **termina**.

Primero, si va a ser usada dentro de otras clases, para un mensaje de error localizado. Si un objeto de una clase en particular tiene un problema, entonces, envia un mensaje a su manejador local de error. Usando este método, se ve claramente que clase tiene el problema. Sin embargo, sería conveniente que las clases puedan reportar problemas antes de terminar el programa, de tal forma que se pueda obtener una mayor información del problema antes que el programa termine. Lo cual significa que no solo cada objeto va a contar con su propio buffer, cuando se almacenan mensajes de error, (el buffer **msg**) deberá existir una lista de apuntadores a objetos ioserror en algún lugar, de tal manera de poder mostrar todos los mensajes de error antes de salir el programa.

Segundo, es deseable a veces usar objetos globales, sobre todo en el archivo de **main()**, de tal forma de poder reportar, tanto, errores del programa, y errores de las clases.

Como no hay garantía de que el manipulador **termina** va a ser llamado. Los objetos a ioserror pueden simplemente reportar errores y esperar a que sea llamado **termina**. La función de ANSI C **atexit()**, considera este problema y crea una lista de funciones a ser llamadas cuando la función **exit()** es llamada. La función **exit()** puede ser llamada como una terminación normal de **main()**. Hay solo una ocasión en que no se llama esta función, ésto es cuando se utiliza la función **abort()**. En C++ no se debe usar esta función ya que no son llamados los destructores de los objetos static.

Sintaxis de **atexit()**.

Descripción: Registra los apuntadores a funciones en **func** como una función de salida. Entonces en una terminación normal del

programa, **exit()**, llama a (*func)() antes de regresar al sistema operativo.
Cada llamada a la función **atexit()**, registra otra función de salida. Se pueden registrar hasta 32 funciones. Antes de salir del programa se ejecutan estas funciones, en el orden primera en entrar, última en salir.

Sintaxis: #include <stdlib.h>
int atexit(atexit_t func);

Prototipo: stdlib.h

V. regresado: Si se ejecuta correctamente, regresa CERO, de otra forma regresa un valor diferente de CERO.

3.1.d) Implementación de los metodos de ioserror.

```
// Programa PLCP78.CPP
// Implementación de la clase ioserror
// Incluye el manejo de new_handler
#include <stdlib.h>
#include <new.h>
#include "plcp78.h"

class p_list {
    static ioserror *list[];
    static int cursor;
public:
    enum { tam = 100 }; // Número de errores a manejar.
    p_list() {
        atexit(p_list::saca);
    }
    // Agrega manejo de errores
    static void registro(ioserror *p) {
        if(cursor < tam)
            list[cursor++] = p;
        else
            clog << "incrementa tamaño de p_list"  << nl;
   }

   // Saca el contenido de todos los manejadores de error
static void saca();
};

// definición de datos miembro static
// Inicialización de cada uno a 0
```

```
ioserror *p_list::list[p_list::tam] = { 0 };
int p_list::cursor = 0;
static p_list phandler_list;    // Conocido solo en este archivo

// Inicializa el ostrstream para escribir el msg en
// el buffer y registra este objeto en p_list:
ioserror::ioserror(char *n_clase)
    : ostrstream(msg, tam), nclase(n_clase) {
    p_list::registro(this); // llamada a función miembro static
}

// El manejador de error termina después de sacar
// el dato, No se pierde la información de error
ostream& termina(ostream& os) {     // manipulador
    exit(1);
    return os;  // Este return no se ejecuta
}               // Permite escribir correctamente
                // la sintaxis.


void p_list::saca() {
    while(cursor -- > 0)
    list[cursor] -> saca();
}

// Obtiene un mensaje, si corre fuera del heap.

struct _INIT_NEW_HANDLER {
    void (*old_handler)();
    static void new_handler() {
        cerr << "Error en el heap, Memoria agotada" << endl;
        exit(1);
    }
    _INIT_NEW_HANDLER() {
        old_handler = set_new_handler(new_handler);
    }
    ~_INIT_NEW_HANDLER() {
    set_new_handler(old_handler);
    }
};

// Válido solamente en este archivo

static _INIT_NEW_HANDLER _init_new_handler;
```

OBSERVACIONES.

Lo primero que se puede observar en el archivo de la implementación es un contenedor para la clase ioserror, para lo cual se usan apuntadores llamados **p_list**. Solo hay una instancia **static** de la clase llamada **phandler_list**. Aquí es donde todos los objetos de la clase ioserror se registran cuando son creados; ésto se puede observar en el constructor de la clase **ioserror**. La función **p_list::registro()** coloca cada puntero en el arreglo interno (después de verificar el rango).

Es necesario aclarar, que esta es una implementación simple, suceptible a mejorar.

El constructor para **p_list** llama a la función **atexit()** con la dirección de la función **p_list::saca()**, de esta forma todos los objetos son mostrados cuando el programa termina. En este caso es necesario definir a **p_list::registro()** y a **p_list::saca()** como funciones miembro **static**, de tal manera que no pertenescan a un objeto en particular, sino a la clase en general.

Cuando se manejan funciones miembro **static**, trabajan para la clase, manipulando datos miembro **static**, este tipo de funciones son tratadas como globales dentro de la clase, permaneciendo ocultas para otros objetos de otras clases.

Recuerde que es posible llamar una función miembro **static** de la manera ordinaria, o bien especificando el nombre de la clase, seguido del operador de alcance y a continuación el nombre de la función, ejemplo: **p_list::saca()**. Puesto que cuenta con las características de una función miembro ordinaria, entonces, su dirección puede ser pasada a la función **atexit().**

El manipulador **termina**, nos saca el contenido de todos los objetos del tipo **ioserror** de la lista, ésto lo hace simplemente llamando a la función **exit()**, la cual manda a ejecutar la función **atexit()**, y ésta usa la dirección de **p_list::saca()**, que almacenó con anterioridad, cuando se fueron produciendo los errores.

Note que el manipulador **termina** regresa un valor a un tipo **ostream**, ésto es necesario solo para satisfacer la sintaxis del compilador.

La función **p_list::saca()**, lo que hace, es recorrer una lista, llamando a **saca()** para cada uno de los objetos del tipo **ioserror**, que fueron almacenados con anterioridad. (note que no se esta haciendo una llamada recursiva; ya que tanto **ioserror** como **p_list**, cada uno tiene su propia versión de la función **saca()**. El compilador sabe cual versión esta llamando, ya que comoce el tipo de objeto al que se esta refiriendo. La función **ioserror::saca()** (de la clase ioserror) solamente coloca información en el objeto **clog**, si hay mensajes en el buffer **msg**, envia, el nombre de la clase, la cadena "error", la dirección del buffer **msg**, aquí es donde se encuentra el mensaje de error y por último el manipulador **endl**.

Veamos ahora como los mensajes de error son colocados en el buffer **msg**. El constructor de la clase **ioserror** lo primero que hace es inicializar la lista, lo cual realiza haciendo una llamada al constructor de la clase base **ostrstream(msg, tam)**. La clase ostrstream nos brinda una forma de manejar memoria como si se tratara de un

iostream. Cuando se usa este tipo de constructor, se le esta indicando al compilador la cantidad de memoria a utilizar, y donde almacenar su dirección inicial.

Si se usa el constructor sin argumentos, el objeto crea y maneja su propia memoria.

El código usado en este caso le indica al compilador que use ostrstream, separe **tam** bytes, y almacene la dirección inicial en el dato miembro **msg**. Al manejar un objeto del tipo **ioserror** si se escribe:

error << "hola";

La cadena "hola" va a ser puesta en el buffer **msg**, de este objeto.

A continuación se presenta una aplicación de la clase **ioserror**.

```
// Programa PLCP78A.CPP
// Demuestra el reporte de errores.

#include <stdlib.h>
#include <conio.h>      // getch()
#include <new.h>
#include "PLCP78.h"


class mu1 {
    static ioserror error;
public:
    void prueba() {
        error << "Número: " << 47 << nl;
    }
};


class mu2 {
    static ioserror error;
public:
    void prueba() {
        error << "Valor: " << 3.14159 << nl;
    }
};


class mu3 {
public:
    friend ostream& operator<<(ostream& os, mu2&);
};



ostream& operator<<(ostream& os, mu2&)
```

```
{
        return os << "mu2" << nl;
}



// Definición de datos miembro static

ioserror mu1::error("Objeto de: mu1");
ioserror mu2::error("Objeto de: mu2");


// Función de error global

ioserror error("Objeto Global: ");

void prueba_sale(int indice) {
    cout << "Para salir: presiona ESC" << endl;
    if(getch() == 27)
        error << "Terminando sobre indice = " << indice
              << nl << termina;
    cout << "Indice = " << indice << endl;
}


void main()
{
    mu1 G;
    mu2 B;
    mu3 U;
    prueba_sale(1);
    G.prueba();
    prueba_sale(2);
    B.prueba();
    prueba_sale(3);
    error << B;     // Operador sobrecargado <<
    cout << "Preparando la salida" << endl;
    // Llenando el heap
    for(;;) new int[1000];
}
```

OBSERVACIONES.

Note que la clase **mu1** y la clase **mu2** contienen un objeto del tipo **static ioserror**, los objetos **ioserror** siempre deberán ser **static**, recuerde que un dato **static** es compartido por todos los objetos de dicha clase.

En la función **prueba()** se puede observar que estos objetos van a trabajar de la misma forma como si fueran de la clase iostream. Cada vez que la información es enviada al objeto miembro **error**, ésta es almacenada. Solamente cuando la función **termina** es llamada (o el programa sale por medio de la función **exit()**, la cual también se puede usar para una terminación normal), muestra todos los mensajes de error en la pantalla, en orden inverso a como fueron almacenados.

La clase **mu3** nos muestra la manera de usar una función friend global para reportar errores de dicha clase.

En la función **main()** se puede ver la llamada al operador (**<<)** en la sentencia:

error << B;

Note que cualquier acción que se puede realizar sobre la clase iostream se puede realizar con un objeto de la clase **ioserror**. Esto es debido a que la clase **ioserror** es una clase derivada del tipo iostream, de tal manera que nuestra clase pasa a ser un tipo de iostream. Por lo tanto, si se crea una salida sobrecargada del **operator<<()**, para una clase hecha por nosotros, este operador también va a trabajar con objetos de la clase **ioserror**. Este hecho nos ahorra código y además nos pone a nuestra disposición toda la potencia de la clase **iostream**.

Usando objetos de la clase **ioserror** en todas nuestras clases se crea un framwork consistente y portable para el reporte de errores. Y como los objeto miembro son **static**, no tienen efecto en el tamaño y comportamiento de ningún objeto que contenga la clase **ioserror**. Si se desea modificar la manera de reportar y manejar errores el framwork puede ser modificado de cierta manera, de tal forma de hacerlo más versátil, y lo interesante es que los cambios van a ser propagados solo con recompilar el programa.

Como se puede observar, el agregar mensajes de error a una clase es muy fácil, y nos vamos a dar cuenta que el manejo de mensajes de error es excepcionalmente útil cuando se trabaja con grandes proyectos los cuales contienen muchas clases. Con ésto, el mismo objeto dará la alarma cuando se encuentra un problema, con lo cual se consigue reducir enormemente el tiempo de rastreo (debugging) de un programa.

La salida nos resulta:

a) Para el caso en que no tecleamos ESC

```
Para salir presione ESC
indice = 1
Para salir presione ESC
indice = 2
Para salir presione ESC
indice = 3
Preparando la salida
Error en el heap, Memoria agotada
```

```
Objeto Global: error: mu2

Objeto de mu2 error: valor: 3.14159

Objeto de mu1 error: Número: 47
```

b) Para el caso en que tecleamos ESC

```
Para salir presione ESC
indice = 1
Para salir presione ESC

Objeto Global: error: error: Terminando sobre indice = 2

Objeto de mu1 error: Número: 47
```

En el caso (a), salimos del programa cuando se llena el heap y se llama a la función new_handler(), de la estructura _INIT_NEW_HANDLER.

En el caso (b), solo se llamó a la función prueba() del objeto de la clase mu1.