

5.- La clase: Alcance de la definición.

5.a) Beneficios del Diseño en C++.

Uno de los principales beneficios de C++ es la separación de la **interface** de la **implementación**. La **interface** es la definición de clase y la implementación nos muestra como trabaja el método y consta de la definición de todas las funciones miembro. Mientras la interface esta disponible para todo el módulo, pudiendose usar la clase en cualquier lugar, la implementación solo existe en un sitio. Si en el futuro el programador desea mejorar la implementación, si el diseño fué bien planteado, no es necesario modificar la interface y recompilar el código relacionado con ella, solo se recompilara la sección de código modificada y se enlazará con el demás código.

Es posible también diseñar el código de la interface y tiempo después escribir el código de la implementación.

5.b) Declaración VS definición.

El ANSI establece una definición clara para "declaración" y "definición", con el C++ este concepto se refuerza, se puede decir que la descripción de una clase (excepto en el caso de miembros estáticos) no reserva espacio de memoria, y sí pertenece realmente a un modelo de un nuevo tipo de dato, no a una variable, se le deberá llamar una **declaración**.

Declaración de nombre de clase; es el nombre de una clase sin una descripción de la clase. Ejemplo:

```
class punto;
```

Declaración de clase; es la declaración de un nombre de clase, seguido del cuerpo de la clase.

```
class uno {  
    int i;  
    char x;  
public:  
    .....  
};
```

5.c) De nuevo Constructores y Destructores. (inicialización y limpiado)

Cuando se define una instancia de un tipo predefinido (semejante como **int**), el compilador reserva espacio de memoria para la variable, si también deseamos inicializarlo, también esta acción la realiza el compilador.

Cuando una variable de un tipo predefinido se sale de su ámbito, el compilador libera la memoria que se reservó para dicha variable, podemos decir que destruye la variable.

El C++ nos permite realizar el mismo proceso con los tipos definidos por el usuario (clases).

Para llevar a cabo este proceso el compilador necesita una función que sea llamada cuando la variable es creada (un constructor) y una función que sea llamada cuando la variable sale de su ámbito (un destructor).

El constructor puede ser más de uno; constructor sobrecargado.

El destructor solamente puede ser uno.

5.c.1) Constructor.

El constructor es una función miembro con el mismo nombre que la clase. El constructor asume que se dispuso del espacio suficiente de memoria para almacenar todas las variables dentro de la estructura del objeto cuando éste es llamado.

Ejemplo:

```
// Programa # 22, uso de un constructor sobrecargado.
#include <iostream.h>

class intmsg {
    int i, j, k;
    char msg[30];
public:
    intmsg() { i = j = k = 0; }
    intmsg(int q) { i = j = k = q; }
    intmsg(int u, int v, int w) {
        i = u;
        j = v;
        k = w;
    }
    void imprime(char *msg);
};

void main()
{
    intmsg A;
    intmsg B(50);
    intmsg C(10, 20, 30);
    A.imprime("Constructor sin argumentos");
    B.imprime("Constructor con un argumento");
    C.imprime("Constructor con 3 argumentos");
}
```

```
void intmsg::imprime(char *msg)
{
    cout << msg << "; " << endl;
    cout << "i = " << i << endl;
    cout << "j = " << j << endl;
    cout << "k = " << k << endl;
}
```

OBSERVACIONES.

La salida nos resulta:

```
Constructor sin argumentos;
i = 0
j = 0
k = 0
Constructor con un argumento;
i = 50
j = 50
k = 50
Constructor con 3 argumentos;
i = 10
j = 20
k = 30
```

La clase `intmsg` tiene tres constructores sobrecargados, uno sin argumentos (usado en la definición de **A**), el otro con un argumento (usado en la definición de **B**), y el último con tres argumentos (usado en la definición de **C**). El método **imprime()** muestra los valores privados del objeto luego que es inicializado.

5.c.2) Destructor.

El nombre del destructor es el mismo nombre que el de la clase con una tilde antes del nombre, en el ejemplo anterior el destructor debería llamarse **~intmsg()**. El destructor no debe tener argumentos, solo es llamado por el compilador y no puede ser llamado explícitamente por el programador, de manera indirecta puede ser invocado por el programador, al llamar éste al operador `delete`.

Mientras, en ocasiones es necesario realizar varios tipos de inicialización sobre un objeto, un destructor de default es suficiente (no hace nada), no siendo necesario definir un destructor. Sin embargo si un objeto inicializa algún hardware, por ejemplo pone una ventana sobre la pantalla, o cambia un valor global, deberá ser eliminado el efecto del objeto (cerrar la ventana), ésto se lleva a cabo cuando el objeto es destruido, en este caso sí es necesario un destructor.

Ejemplo:

```
// Programa # 23, uso de un destructor.
#include <iostream.h>

int cont = 0;

class obj {
public:
    obj() {
        cont++;
        cout << "C. Número de Objetos: " << cont << endl;
    }
    ~obj() {
        cont--;
        cout << "D. Número de Objetos: " << cont << endl;
    }
};

void main()
{
    obj A, B, C, D, E;
    {
        obj F;
    }
    {
        obj G;
    }
}
```

OBSERVACIONES.

La salida del programa nos resulta:

```
C. Número de Objetos: 1
C. Número de Objetos: 2
C. Número de Objetos: 3
C. Número de Objetos: 4
C. Número de Objetos: 5
C. Número de Objetos: 6
D. Número de Objetos: 5
C. Número de Objetos: 6
D. Número de Objetos: 5
D. Número de Objetos: 4
D. Número de Objetos: 3
```

D. Número de Objetos: 2
D. Número de Objetos: 1
D. Número de Objetos: 0

Mientras se crean los objetos el contador se incrementa y cuando los objetos salen de su ámbito, se llama al destructor, el cual decrementa el contador. Note que después de haber creado el primer conjunto de variables (A, B, C, D, E), F es creada y luego destruida, G también es creada y luego destruida, por último el resto de las variables son destruidas. Entonces, cuando se llega al final del ámbito, se llama el destructor para cada una de las variables en el ámbito.

5.d) Datos miembros de clase; static.

Cada vez que se define un objeto el cual pertenece a un tipo de clase dado, se genera espacio de memoria para todas las variables de la clase a la cual pertenece el objeto. Es posible definir una variable dentro de una clase de tal manera que solo una instancia de la variable es creada para todos los objetos declarados de ese tipo de clase, cualquier objeto de la clase tiene acceso a dicha variable, lo cual significa que el dato es compartido por todos los objetos de dicha clase, en vez de ser duplicado para cada objeto. Lo anterior se puede llevar a cabo anteponiendo a la declaración del dato miembro la palabra **static**.

Es posible declarar más de una variable **static**, para ello es necesario indicar el tipo en la declaración dentro de la clase. Ejemplo:

```
class clase_estatica {
    static int i;
    static char j;
    .
    .
public:
    .
    .
};
```

Veamos un ejemplo de uso de un miembro **static**.

```
// Programa # 24, Uso de miembros static.
#include <iostream.h>

class comun {
    static i;    // Declaración NO definición.
public:
    comun() { i++; }
```

```
    ~comun() { i--; }
    void observar() {
        if(i > 1)
            cout << "Hay otro objeto de esta clase" << endl;
        else
            cout << "No hay otro objeto de esta clase" << endl;
        }
};

// Se deberá proporcionar una definición para un miembro estatic
int comun::i = 0;

void main()
{
    comun A;
    A.observar();
    {
        comun B;
        B.observar();
    } // B se destruye aqui
    A.observar();
} // A se destruye aqui
```

OBSERVACIONES.

La salida nos resulta:

```
No hay otro objeto de esta clase
Hay otro objeto de esta clase
No hay otro objeto de esta clase
```

En la primer línea, solo hay declarado un objeto, en la segunda, hay dos objetos declarados, y en la tercera hay un objeto, el otro ya fué dado de baja, ya que nos encontramos fuera de su ámbito de declaración.

Note la actuación del destructor al llegar el objeto al término de su ámbito de declaración, lo que nos permite la verificación de si hay más de un objeto declarado.

El ejemplo también muestra otro motivo de usar un destructor; el mantener la información acerca del objeto

Se deberá reservar espacio de almacenamiento de manera explícita, e inicializar todos los datos miembro static. No se puede inicializar una variable de clase static de la misma forma que se hace con una variable static normal.

```
class mala {
    static int i = 50; //error
```

```
};
```

En vez de esto se deberá usar una sintaxis explícita para el miembro static; esto se hace usando el nombre de la clase y el operador de ámbito con el identificador, lo que lo hace diferente a la definición de objetos globales.

```
int mala::i = 50;
```

La definición e inicialización deberán hacerse fuera de todo cuerpo de clase y función.

Si se definen más de 2 datos miembro static es necesario indicar el tipo de cada uno de ellos, ejemplo.

```
class nocomun {  
    static char x;  
    static int y;  
public:  
    nocomun();  
    ~nocomun();  
};
```

La inicialización se lleva a cabo como en el ejemplo anterior.

5.e) Datos miembros de clase; const.

Es posible definir un miembro de una clase como **const**, aunque el significado es un poco diferente al de "C". Esto es, siempre se asigna espacio de memoria para un dato miembro **const**, entonces, un dato miembro definido como **const** ocupa espacio dentro de la clase, por otra parte, un **const** deberá ser inicializado en el punto donde es definido. ¿Qué significa esto, en el caso de la clase?. No se asigna memoria para una variable **const** de una clase hasta el momento de crear un objeto, éste es el punto donde el **const** deberá ser inicializado. De esta forma, el significado de **const** para miembros de una clase es "constante durante el tiempo de vida del objeto".

La inicialización de un **const** deberá hacerse en el constructor, esto se realiza de una forma especial, después de la lista de argumentos del constructor, pero antes de su cuerpo, lo que indica que este código es ejecutado antes de entrar en el cuerpo del constructor, esto agrega un propósito más al uso del constructor.

```
// Programa # 25, uso de const en datos miembro.  
  
#include <iostream.h>  
class contador {
```

```
int cont;
const int max; // int es opcional
public:
    contador(int Max = 10, int ini = 0) : max(Max)
    {
        cont = ini;
    }
    int inc() {
        if( ++cont == max) {
            cout << "Entró en if  " << cont << endl;
            return 1;
        }
        cout << "No entró en if  " << cont << endl;
        return 0;
    }
};

void main()
{
    contador A, B(12), C(3,2);

    while(!B.inc()) {
        cout << "Dentro del while" << endl;
        if(A.inc()) {
            cout << "Dentro del if" << endl;
            C.inc();
        }
    }
}
```

OBSERVACIONES.

Al inicializar los objetos A, B y C, éstos se inicializan de la siguiente forma:

A(10, 0)
B(12, 0)
C(3, 2)

Donde el segundo número pertenece a **cont**
y el primero pertenece a **max** (const), él
cual va a permanecer constante, solo va
a variar el primer término (cont), por
efecto de la función miembro **inc()**.

La corrida nos resulta:

```
No entró en if  1      // B.inc()
Dentro del while
No entró en if  1      // A.inc()
No entró en if  2      // B.inc()
Dentro del while
```

```
No entró en if 2      // A.inc()
No entró en if 3      // B.inc()
Dentro del while
No entró en if 3      // A.inc()
No entró en if 4      // B.inc()
Dentro del while
No entró en if 4      // A.inc()
No entró en if 5      // B.inc()
Dentro del while
No entró en if 5      // A.inc()
No entró en if 6      // B.inc()
Dentro del while
No entró en if 6      // A.inc()
No entró en if 7      // B.inc()
Dentro del while
No entró en if 7      // A.inc()
No entró en if 8      // B.inc()
Dentro del while
No entró en if 8      // A.inc()
No entró en if 9      // B.inc()
Dentro del while
No entró en if 9      // A,inc()
No entró en if 10     // B.inc()
Dentro del while
Entró en if 10        // A.inc()
Dentro del if
Entró en if 3         // C.inc()
No entró en if 11     // B.inc()
Dentro del while
No entró en if 11     // A.inc()
Entró en if 12        // B.inc()
```

La sentencia **max(Max)** realiza la inicialización, también podremos pasar la inicialización de `cont` a la lista de inicialización del constructor, entonces, esta sentencia nos resulta:

```
contador(int max = 10, int ini = 0) : max(Max), cont(ini) {}
```

es otra manera de inicialización.

5.f) Una constante real dentro de la clase.

El tratamiento de **const** dentro de la clase crea una situación inconveniente cuando se crea un arreglo dentro de una clase. Cuando se manejan arreglos comunes (no dentro de una clase), es una buena práctica de programación usar una constante con nombre para definir el tamaño del arreglo.

```
const tam = 10;  
char cad[tam];
```

De esta manera, cualquier código que se refiera al tamaño del arreglo usará **tam**, y si es necesario cambiar el tamaño del arreglo, éste solo es cambiado en un lugar, en la definición **const**.

No se puede usar una definición **const** de datos en la definición de un arreglo dentro de una clase porque el compilador deberá conocer el tamaño de un arreglo cuando es definido y porque **const** dentro de la clase siempre reserva memoria. (el compilador no puede conocer el contenido de una localidad de almacenamiento). Afortunadamente existe una manera de tratar este problema, la solución es el uso de **enum**, como sigue.

```
enum { tam = 10 };
```

La clase es un tipo, no se reserva memoria para ella, **const** reserva memoria, hay conflicto.

Esta declaración introduce un nombre llamado **tam**, él cual representa un valor entero de 10.

No se asigna memoria para nombres de enumeraciones, así que el compilador siempre tiene el valor disponible, a esta técnica se le llama "enum hack", nos salva también de usar el **#define** del preprocesador.

```
class arreglo {  
    enum { tam = 10 };  
    int a[tam];  
};
```

6.- El archivo de cabecera.

Cuando se crea una clase se ha creado un nuevo tipo de dato, se desea que este tipo de dato sea accesible para cualquier programador, para conseguir este propósito, es conveniente separar la interface (la declaración de clases) de la implementación (la definición de las funciones miembro de la clase), de esta forma la implementación puede ser cambiada sin forzar la recompilación de todo el sistema, esto se puede llevar a cabo colocando la declaración de la clase en un archivo de cabecera.

6.a) Funciones de librería y la compilación por separado.

En vez de colocar la declaración de las clases, la definición de las funciones miembro y la función `main()` en el mismo archivo, es mejor aislar la declaración de clases en un archivo de cabecera el cual se incluye en cada archivo donde la clase es usada. Las definiciones de las funciones miembro son también separadas en su propio archivo. El usuario de la clase simplemente incluye el archivo de cabecera, crea objetos (instancias de la clase) y enlaza los módulos objeto con las librerías.

El concepto de una librería conteniendo funciones dentro del mismo módulo objeto y un archivo de cabecera conteniendo las declaraciones de las funciones, es la forma estándar de manejar grandes proyectos en C, lo cual es un deber en C++; en C cualquier definición de función deberá ser puesta en una librería, en C++ las clases determinan cuáles funciones son asociadas con ellas para tener acceso a sus datos privados.

Cualquier función miembro de una clase, podrá ser declarada en la misma declaración de clases, es válido para funciones pequeñas, al definir las dentro de la clase se convierten en funciones **inline**. Esto es más estricto cuando se manejan funciones **inline**, aunque no sean definidas dentro de la clase éstas deberán definirse en la interface, anteponiéndoles la palabra **inline**, no se acepta su definición en otro lugar o archivo del programa.

6.b) Importancia de usar un archivo de cabecera común.

Cuando se usa una función de una librería, C nos da la opción de ignorar el archivo de cabecera y simplemente declarar la función a manipular. Si no se incluye el archivo de cabecera se omite cualquier verificación de errores sobre los argumentos. Esta práctica puede causar serios problemas.

Si se colocan todas las declaraciones de las funciones en un archivo de cabecera y se incluye dicho archivo cada vez que se usen las funciones, se asegura la coincidencia de declaraciones y definiciones de las funciones utilizadas.

Si una clase es declarada en un archivo de cabecera en C++, se deberá incluir dicho archivo cada vez que la clase sea usada y donde las funciones miembro son definidas. El compilador generará un mensaje de error si olvidamos llamar una función sin declararla primero. Forzando el uso de archivos de cabecera el lenguaje asegura consistencia en manejo de librerías y reduce **bugs** al forzar siempre el uso de la interface.

6.c) Previniendo Redeclaración de clases.

Cuando se coloca una declaración de clase en un archivo de cabecera, existe la posibilidad de incluir el archivo de cabecera más de una vez en un programa. La clase

iostream.h es un buen ejemplo, si se incluye más de una vez se corre el riesgo de redeclarar clases.

El compilador considera la redeclaración de clases como un error, ya que no se puede usar el mismo nombre para declarar diferentes clases, para prevenir este error cuando se incluyen varios archivos de cabecera, deberá incluirse alguna inteligencia al archivo de cabecera para prevenir esta situación.

6.d) Un estandar para cada clase en el archivo de cabecera.

En cada archivo de cabecera que contenga declaraciones de clase, se deberá checar para constatar que esta porción de código no ha sido incluida con anterioridad. Esto se puede llevar a cabo mediante el uso de una bandera del preprocesador, si la bandera no esta activa, el archivo no ha sido incluido, el archivo se deberá incluir y activar la bandera, de tal forma que las clases no puedan ser redeclaradas posteriormente. Si la bandera ya fué definida, se deberá ignorar el código que sigue.

A continuación se muestra la porción de código en el archivo de cabecera, el cual deberá ser incluido al inicio del archivo fuente.

```
#ifndef CLASE_FLAG_  
#define CLASE_FLAG_  
  
// Aqui va la declaración de clases.  
  
#endif // CLASE_FLAG_
```

Como se puede ver, la primera vez que se incluya el archivo de cabecera, la declaración de las clases será incluida por el preprocesador, y las siguientes veces que se trate de incluir el archivo, todas sus declaraciones de las clases serán ignoradas.

Veamos un ejemplo:

```
#ifndef SIMPLE_H_  
#define SIMPLE_H_  
  
class simple {  
    int i, j, k;  
public:  
    simple() { i = j = k = 0; }  
};  
#endif // SIMPLE_H_
```

7.- Definiendo funciones miembros de clase.

7.a) El ámbito del operador de resolución. (::)

Para definir una función miembro, primero se le deberá decir al compilador que la función definida esta asociada a una clase particular, ésto se hace mediante el uso del operador de ámbito (::). Por ejemplo:

```
// Programa # 26, definiendo una función no en línea.
#include <iostream.h>
class entero {
    int a, b, c;
public:
    entero();    // Declaración del constructor.
    void imprime();
};

entero::entero()
{
    a = 50;
    b = 60;
    c = 70;
}

void entero::imprime()
{
    cout << "a = " << a;
    cout << ", b = " << b;
    cout << ", c = " << c << endl;
}

void main()
{
    entero prueba;
    prueba.imprime();
}
```

OBSERVACIONES.

Como ya se había mencionado en líneas anteriores, la función miembro es asociada a la clase, anteponiendo al nombre de la función el nombre de la clase, seguido por el operador de ámbito de resolución. La función es compilada como una función normal, en vez de una función en línea.

El operador de resolución de ámbito puede usarse cuando no se tenga la seguridad de la definición que va a usar el compilador. Este operador también se puede usar para seleccionar otra definición en vez de la de default. Por ejemplo, si se crea una clase en la cual se define una función cuyo nombre ya ha sido definido en alguna parte,

por ejemplo en una librería, es posible seleccionar una u otra función dependiendo a la que se quiera llamar. Ejemplo:

```
// Programa #27, uso del operador de resolución de ámbito.
#include <stdio.h>

class muestra {
public:
    void puts(char *);
};

void muestra::puts(char *msg)
{
    for(int i = 0; *(msg+i); i++)
        printf("%d ", *(msg+i));
}

void main()
{
    char x[] = "Saludos de main";
    muestra D;

    ::puts(x);
    D.puts(x);
}
```

OBSERVACIONES.

Primero llamamos a la función **puts()** de la librería de C y luego llamamos a la función **puts()** de la clase muestra.

No es necesario el operador de ámbito (::) en la función global **puts**, sin embargo da claridad al código.

La salida nos resulta:

```
Saludos de main
83 97 108 117 100 111 115 32 100 101 32 109 97 105 110
```

7.b) Llamando otras funciones miembro.

Es posible llamar funciones miembro desde otras funciones miembro. Una función miembro puede ser llamada por su nombre desde otra función miembro, no es necesario el nombre de un objeto ni el punto. Ejemplo:

```
// Programa # 28, llamada de una función miembro
// desde una función miembro.
#include <iostream.h>
#include <stdlib.h>
class arreglo {
    enum { tam = 10 };
    int a[tam];
    void v_indice(const int indice);
public:
    arreglo(const int iniv = 0);
    void ponev(const int indice, const int valor);
    int leeval(const int s);
};

arreglo::arreglo(const int iniv)
{
    for(int i = 0; i < tam; i++)
        ponev(i, iniv+i);
}

void arreglo::v_indice(const int indice)
{
    if(indice < 0 || indice >= tam) {
        cerr << "ERROR, arreglo fuera de rango" << endl;
        exit(1);
    }
}

void arreglo::ponev(const int indice, const int valor)
{
    v_indice(indice);
    a[indice] = valor;
}

int arreglo::leeval(const int s)
{
    v_indice(s);
    return a[s];
}

void main()
{
    arreglo B(4);
    int x;
    for(int i = 0; i < 10; i++) {
        x = B.leeval(i);
        cout << "Valor de x = " << x << endl;
    }
    x = B.leeval(12);
}
```

```
    cout << "Valor de x = " << x << endl;  
}
```

OBSERVACIONES.

La corrida nos resulta:

```
Valor de x = 4  
Valor de x = 5  
Valor de x = 6  
Valor de x = 7  
Valor de x = 8  
Valor de x = 9  
Valor de x = 10  
Valor de x = 11  
Valor de x = 12  
Valor de x = 13
```

ERROR, arreglo fuera de rango

En el programa podemos notar que la función `v_indice()`, es un miembro privado, el cual solo puede ser llamado por otras funciones miembro. Mientras el usuario quiera poner o leer un valor, la función `v_indice()` es llamada para asegurar que el arreglo no ha sobrepasado su límite.

Cuando definimos el objeto el arreglo es llenado a partir del valor de inicialización, al leer el arreglo si el elemento que queremos leer sobrepasa de 9, mandará el mensaje de ERROR a la pantalla.

En el programa leemos todo el arreglo, es lo que se muestra en la pantalla como valor de x, al final intentamos leer el elemento 12, el cual no es válido y el programa aborta, enviando un mensaje de error.

7.c) **friend**: Acceso a elementos privados de otra clase.

7.c.1) Función común.

Cuando es necesario que funciones que no son miembros de una clase **X** tengan acceso a los datos de dicha clase; una manera es hacer los datos publicos, pero esto no es buena idea, ya que quedarán expuestos y podrán ser alterados inadvertidamente.

La solución en C++ es la creación de una función **friend**, estas funciones no son miembros de la clase **X**. (pueden ser miembros de alguna otra clase, de hecho toda una clase puede declararse **friend**). Un **friend** cuenta con privilegios de acceso como si

fuera una función miembro, pero no se asocia con un objeto de la clase de la cual es **friend** (no puede ser llamada una función de la clase sin asociarla con un objeto). La clase **X** conserva el control de conceder privilegios a funciones **friend**, de tal manera que se sabe claramente quien tiene el permiso de cambiar los datos privados.

Para ver un ejemplo, consideremos un programa que define 2 clases llamadas **linea** y **rectangulo**. La clase **linea** contiene todos los datos y código necesarios para dibujar una línea horizontal discontinua de cualquier longitud, empezando en la coordenada X,Y que se indique y utilizando un color especificado. La clase **rectangulo** contiene todo el código y los datos necesarios para dibujar un rectangulo en las coordenadas especificadas para la esquina superior izquierda y para la esquina inferior derecha, y con el color que se indique. Las 2 clases utilizan la función **mismo_color()** para determinar si una línea y un rectangulo estan pintados del mismo color.

```
// Programa # 29, uso de funciones friend

#include <iostream.h>

#include <conio.h>

class linea;          // Es necesario declarar la clase linea

class rectangulo {
    int color;          // Color del rectangulo
    int xsup, ysup;     // Esquina superior izquierda
    int xinf, yinf;     // Esquina inferior derecha
public:
    friend int mismo_color(linea l, rectangulo b);
    void pon_color(int c);
    void definir_rectangulo(int x1, int y1, int x2, int y2);
    void mostrar_rectangulo();
};

class linea {
    int color;
    int xinicial, yinicial;
    int lon;
public:
    friend int mismo_color(linea l, rectangulo b);
    void pon_color(int c);
    void definir_linea(int x, int y, int l);
    void mostrar_linea();
};

// Regresa un valor verdadero si linea y rectangulo
// tienen el mismo color
```

```
int mismo_color(linea l, recuadro b)
{
    if(l.color == b.color)
        return 1;
    else
        return 0;
}

void recuadro::pon_color(int c)
{
    color = c;
}

void linea::pon_color(int c)
{
    color = c;
}

void recuadro::definir_recuadro(int x1, int y1, int x2, int y2)
{
    xsup = x1;
    ysup = y1;
    xinf = x2;
    yinf = y2;
}

void recuadro::mostrar_recuadro()
{
    textcolor(color);
    gotoxy(xsup, ysup);
    for(int i = xsup; i <= xinf; i++)
        cprintf("-");

    gotoxy(xsup, yinf-1);
    for(i = xsup; i <= xinf; i++)
        cprintf("-");

    gotoxy(xsup, ysup);
    for(i = ysup; i <= yinf; i++) {
        cprintf("|");
        gotoxy(xsup, i);
    }

    gotoxy(xinf, ysup);
    for(i = ysup; i <= yinf; i++) {
        cprintf("|");
        gotoxy(xinf, i);
    }
}
```

```
    }

}

void linea::definir_linea(int x, int y, int l)
{
    xinicial = x;
    yinicial = y;
    lon = l;
}

void linea::mostrar_linea()
{
    textcolor(color);
    gotoxy(xinicial, yinicial);
    for(int i = 0; i < lon; i++)
        cprintf("-");
}

void main()
{
    recuadro b;
    linea l;

    clrscr();
    b.definir_recuadro(10, 10, 20, 20);
    b.pon_color(3);
    b.mostrar_recuadro();

    l.definir_linea(4, 4, 30);
    l.pon_color(2);
    l.mostrar_linea();

    if(!mismo_color(l, b))
        cout << "Colores NO iguales" << endl;
    cout << "Pulse una Tecla";
    getch();

    // La línea y recuadro tendrán el mismo color

    l.definir_linea(4, 4, 30);
    l.pon_color(3);
    l.mostrar_linea();

    if(mismo_color(l, b))
        cout << "Tienen el mismo color" << endl;
}
```

OBSERVACIONES.

La función `mismo_color()`, la cual no es miembro de ninguna de las clases pero es **friend** de ambas, nos regresa un valor verdadero si tanto el objeto del tipo `linea` y el objeto del tipo `recuadro` se dibujan con el mismo color, en caso contrario regresa un cero. Esta función necesita acceder las partes privadas de las clases **linea** y **recuadro**.

NOTA: observe la declaración vacía de **linea** que aparece al principio de la declaración de clases, esto se debe a que la función `mismo_color()` que está declarada dentro de la clase **recuadro** hace referencia a la clase **linea**, antes de que esta clase haya sido declarada, por ello, es necesario hacer una referencia anticipada a la clase **linea**. Si no se hace esto, el compilador no sabrá lo que es **linea** cuando la encuentre en la declaración de **recuadro**. En C++ una referencia anticipada a clase es simplemente la palabra **class** seguida del nombre del tipo de dicha clase. Normalmente las únicas ocasiones en que es necesario hacer referencias anticipadas son aquellas en las que están implicadas funciones **friend**.

7.c.2) Clase friend.

En el ejemplo anterior se declaró una función **friend** en ambas clases, siendo sus argumentos objetos a cada una de las clases. Otra forma de realizar este mismo trabajo es declarar una clase completa como **friend** (`linea`), entonces, en la clase (`linea`) podemos declarar la función **`mismo_color()`** como un miembro, cuyo argumento es un objeto a la clase (`recuadro`). Esta función se manejará como una función miembro de la clase (`linea`). Veamos cómo es modificado el programa.

```
// Programa # 29A, uso de clase friend
#include <iostream.h>
#include <conio.h>

class recuadro {
    int color;           // Color del recuadro
    int xsup, ysup;      // Esquina superior izquierda
    int xinf, yinf;      // Esquina inferior derecha
public:
    void pon_color(int c);
    void definir_recuadro(int x1, int y1, int x2, int y2);
    void mostrar_recuadro();
    friend class linea;
};

class linea {
    int color;
    int xinicial, yinicial;
```

```
        int lon;
public:
    int mismo_color(recuadro b); // Miembro de class linea
    void pon_color(int c);
    void definir_linea(int x, int y, int l);
    void mostrar_linea();
};

// Regresa un valor verdadero si linea y recuadro
// tienen el mismo color

int linea::mismo_color(recuadro b)
{
    if(color == b.color)
        return 1;
    else
        return 0;
}

void recuadro::pon_color(int c)
{
    color = c;
}

void linea::pon_color(int c)
{
    color = c;
}

void recuadro::definir_recuadro(int x1, int y1, int x2, int y2)
{
    xsup = x1;
    ysup = y1;
    xinf = x2;
    yinf = y2;
}

void recuadro::mostrar_recuadro()
{
    textcolor(color);
    gotoxy(xsup, ysup);
    for(int i = xsup; i <= xinf; i++)
        cprintf("-");

    gotoxy(xsup, yinf-1);
    for(i = xsup; i <= xinf; i++)
        cprintf("-");
}
```

```
        gotoxy(xsup, ysup);
        for(i = ysup; i <= yinf; i++) {
            cprintf("|");
            gotoxy(xsup, i);
        }

        gotoxy(xinf, ysup);
        for(i = ysup; i <= yinf; i++) {
            cprintf("|");
            gotoxy(xinf, i);
        }
    }

void linea::definir_linea(int x, int y, int l)
{
    xinicial = x;
    yinicial = y;
    lon = l;
}

void linea::mostrar_linea()
{
    textcolor(color);
    gotoxy(xinicial, yinicial);
    for(int i = 0; i < lon; i++)
        cprintf("-");
}

void main()
{
    recuadro b;
    linea l;

    clrscr();
    b.definir_recuadro(10, 10, 20, 20);
    b.pon_color(3);
    b.mostrar_recuadro();

    l.definir_linea(4, 4, 30);
    l.pon_color(2);
    l.mostrar_linea();

    if(!l.mismo_color(b))
        cout << "Colores NO iguales" << endl;
    cout << "Pulse una Tecla";
}
```

```
        getch();

// La línea y recuadro tendrán el mismo color

    l.definir_linea(4, 4, 30);
    l.pon_color(3);
    l.mostrar_linea();

    if(l.mismo_color(b))
        cout << "Tienen el mismo color" << endl;
}
```

OBSERVACIONES.

Los cambios realizados en el archivo anterior fueron los siguientes:

- 1) Se eliminó la declaración de; **class linea**;
- 2) Se insertó en la clase **recuadro**; friend class linea; (se puede omitir la palabra **class**)
- 3) Se eliminó de la clase **recuadro** la sentencia:
friend int mismo_color(linea l, recuadro b);
- 4) Ahora mismo_color() es función miembro de class linea. Se declaró como:
int mismo_color(recuadro b);
Donde su argumento es un objeto a class recuadro.
- 5) Para llamar a la función mismo_color(), se escribe: l.mismo_color(b);
Donde l es un objeto a linea.
- 6) Cambió un poco la definición de la función mismo_color().
Cuando se llama al dato **color** de la clase linea, el proceso es idéntico al manejo de los datos de una clase en una función miembro.
Cuando se llama a **color** de la clase recuadro se hace de la forma como se referencia un miembro de una estructura.
b.color
- 7) La definición de la función mismo_color(), se realiza de la misma forma que cualquier función miembro.

7.c.3) Declarando una función miembro friend.

Es posible también seleccionar una sola función miembro de una clase como **friend**, para lo cual es necesario declararla como tal, (friend) en la clase de la cual no es miembro. En este caso como sucedió también en el primer método, hay que tener muy en cuenta el orden de las declaraciones. Veamos el mismo ejemplo, pero ahora usando este método.

```
// Programa # 29B, uso de una función friend

#include <iostream.h>
#include <conio.h>

class linea;           // Declaración de clase linea

class recuadro {
    int color;          // Color del recuadro
    int xsup, ysup;     // Esquina superior izquierda
    int xinf, yinf;     // Esquina inferior derecha
public:
    int mismo_color(linea l);
    void pon_color(int c);
    void definir_recuadro(int x1, int y1, int x2, int y2);
    void mostrar_recuadro();
};

class linea {
    int color;
    int xinicial, yinicial;
    int lon;
public:
    friend int recuadro::mismo_color(linea l);
    void pon_color(int c);
    void definir_linea(int x, int y, int l);
    void mostrar_linea();
};

// Regresa un valor verdadero si linea y recuadro
// tienen el mismo color

int recuadro::mismo_color(linea l)
{
    if(l.color == color)
        return 1;
    else
        return 0;
}

void recuadro::pon_color(int c)
{
    color = c;
}
```

```
void linea::pon_color(int c)
{
    color = c;
}

void recuadro::definir_recuadro(int x1, int y1, int x2, int y2)
{
    xsup = x1;
    ysup = y1;
    xinf = x2;
    yinf = y2;
}

void recuadro::mostrar_recuadro()
{
    textcolor(color);
    gotoxy(xsup, ysup);
    for(int i = xsup; i <= xinf; i++)
        cprintf("-");

    gotoxy(xsup, yinf-1);
    for(i = xsup; i <= xinf; i++)
        cprintf("-");

    gotoxy(xsup, ysup);
    for(i = ysup; i <= yinf; i++) {
        cprintf("|");
        gotoxy(xsup, i);
    }

    gotoxy(xinf, ysup);
    for(i = ysup; i <= yinf; i++) {
        cprintf("|");
        gotoxy(xinf, i);
    }
}

void linea::definir_linea(int x, int y, int l)
{
    xinicial = x;
    yinicial = y;
    lon = l;
}

void linea::mostrar_linea()
```

```
{
    textcolor(color);
    gotoxy(xinicial, yinicial);
    for(int i = 0; i < lon; i++)
        cprintf("-");
}

void main()
{
    recuadro b;
    linea l;

    clrscr();
    b.definir_recuadro(10, 10, 20, 20);
    b.pon_color(3);
    b.mostrar_recuadro();

    l.definir_linea(4, 4, 30);
    l.pon_color(2);
    l.mostrar_linea();

    if(!b.mismo_color(l))
        cout << "Colores NO iguales" << endl;
    cout << "Pulse una Tecla";
    getch();

    // La línea y recuadro tendrán el mismo color

    l.definir_linea(4, 4, 30);
    l.pon_color(3);
    l.mostrar_linea();

    if(b.mismo_color(l))
        cout << "Tienen el mismo color" << endl;
}
```

OBSERVACIONES.

Declaramos la función `mismo_color()` como miembro de la clase **recuadro**, y ahora recibe como argumento un objeto de la clase **linea**, como la clase **linea** en este punto todavía no ha sido declarada, es necesario declararla antes de la clase **recuadro**.

La función `mismo_color()`, perteneciente a la clase **recuadro** es declarada en la clase **linea** como función **friend**.

Ahora **color** es dato de la clase **recuadro**, y para referenciar **color** de la clase **linea**, es necesario hacer referencia al objeto, como `l.color`, en este caso en particular.