[1]Programación Orientada a Objetos con C++

Es importante entender que el estudio de C++ es necesario para el entendimiento de los conceptos de POO y de otros lenguajes que estudiaremos en otras unidades y que se basan en C++.

Para ampliar los conceptos aquí desarrollados puede consultar el libro de Delannoy[1] o los libros digitales disponibles en el Campus, particularmente:

- *Pozo Coronado – Curso de C++.pdf (capítulos 27 a 37)*

1) Estructura con declaración de funciones miembro o métodos

En C++ podemos asociar las funciones a los datos mediante una estructura que incluya dentro de su definición los prototipos de esas funciones:

```
// struct punto con funciones miembros
#include <iostream.h>

struct punto
{       int x;
        int y;
        void inicializa(int, int);
        void desplaza(int, int);
        void visualiza();
};

void punto::inicializa(int abs, int ord)
{       x=abs;
        y=ord;
}

void punto::desplaza(int dx, int dy)
{       x+=dx;
        y+=dy;
}

void punto::visualiza()
{       cout << "Estoy en : " << x << ", " << y << "\n";
}

void main ()
{ punto a, b;
a.inicializa(5, 2);
a.desplaza(-2,4);
a.visualiza();
b.inicializa(1,-1);
b.visualiza();
}
```

En este código lo único inusual es el empleo del **operador ::** para indicar la pertenencia de una función a una determinada estructura. También observamos que para invocar ahora a las funciones empleamos el **operador .** (punto) tal como lo hacíamos para acceder a los campos de la estructua.

2) Clase punto

[1] Delannoy – Aprender C++ - Disponible en Biblioteca FRTL.

La estructua en C++ es un caso particular de la **clase**. La declaración de una clase se parece a la de una esturctua, pero empleando la palabra reservada **class**:

```
// clase punto
#include <iostream.h>

class punto
{private:
        int x;
        int y;
  public:
        void inicializa(int, int);
        void desplaza(int, int);
        void visualiza();
};

void punto::inicializa(int abs, int ord)
{       x=abs;
        y=ord;
}

void punto::desplaza(int dx, int dy)
{       x+=dx;
        y+=dy;
}

void punto::visualiza()
{       cout << "ubicacion: " << x << ", " << y << "\n";
}

void main ()
{
punto a, b;
a.inicializa(5,2);
a.visualiza();
a.desplaza(-2,4);
a.visualiza();
b.inicializa(1,-1);
b.visualiza();
}
```

El código anterior se parece al del ejemplo 1, excepto en que ahora se han empleado las especificaciones **public** y **prívate** para los miembros (atributos o métodos) de la clase.
Por omisión los miembros son privados. Los miembros privados solo pueden ser accedidos por métodos de la propia clase, en tanto que los miembros públicos son accesibles por cualquier otra clase.

3) Clase punto con constructor

En en ejemplo 2 observamos que el programa que utilice esta clase debe prever el empleo del método *inicializa* ya que de lo contrario se produciría un error. Para evitarlo se emplea un método especial llamado constructor. El **constructor** debe tener el mismo nombre que la clase y no puede tener valor de retorno. Este método es invocado automáticamente al crear un objeto (instancia) de la clase.
Otro método especial es el **destructor**: este método se invoca automáticamente cuando se destruye un objeto y su utilidad la veremos más adelante. Lleva también el mismo nombre que la clase pero precedido del símbolo ~.

```
// clase punto con constructor y destructor
#include <iostream.h>
```

```
class punto
{       int x;
        int y;
 public:
        void desplaza(int, int);
        void visualiza();
        punto(int abs, int ord); // constructor
        ~punto(); // destructor
};

void punto::desplaza(int dx, int dy)
{       x+=dx;
        y+=dy;
}

void punto::visualiza()
{       cout << "ubicacion: " << x << ", " << y << "\n";
}

punto::punto(int abs, int ord) // constructor
{       cout << "constructor del punto: " << abs << ", " << ord << "\n";
                x=abs;
                y=ord;
}

punto::~punto()
{       cout << "destructor del punto: " << x << ", " << y << "\n";
}

void main ()
{ punto a(5,2);
a.visualiza();
a.desplaza(-2,4);
a.visualiza();
punto b(1,-1);
b.visualiza();
}
```

4) Clase azar

Veremos ahora la clase que llamamos *azar* que tiene como atributo un array de 10 enteros. El constructor de esta clase se programa para que cargue el array con números aleatorios en un rango indicado por el parámetro *max*.

```
// clase azar con atributos estáticos (arreglo de tamaño fijo)
#include <iostream.h>
#include <stdlib.h>          // para la función rand
class azar
{  int val[10] ;
 public:
        azar (int) ;
        void visualiza () ;
} ;

azar::azar (int max) // constructor: crea 10 valores al azar
{  int i ;
srand(time(NULL));
   for (i=0 ; i<10 ; i++)
        val[i] = rand()% (max+1) ;
```

```
}

void azar::visualiza ()          // para visualizar los 10 valores
{  int i ;
   for (i=0 ; i<10 ; i++)
            cout << val[i] << " " ;
   cout << "\n" ;
}

int main()
{           azar juego1 (5) ;
            juego1.visualiza () ;
            azar juego2 (12) ;
            juego2.visualiza () ;
}
```

No hay en este ejemplo elementos nuevos de sintaxis. Pero nos sirve para pensar el siguiente ejemplo:

5) Clase azar con datos miembro dinámicos. Destructor

Ahora queremos tener la misma clase *azar* pero con el array de tamaño configurable, para lo cual ha de declararse como un puntero y luego en el constructor se reservará memoria dinámica (mediante el operador new) para alojar la cantidad de enteros indicada por el parámetro *nb.* También se ha añadido un atributo llamado *nbval* el cual guardará el tamaño del array.

```
// clase azar con atributos dinámicos (arreglo de tamaño configurable)
#include <iostream.h>
#include <stdlib.h>      // para la función rand
class azar
{
            int nbval ;           // cantidad de valores
            int * val ;           // puntero a enteros (array de enteros)
 public :
            azar (int, int) ;   // constructor
            ~azar () ;          // destructor
            void visualiza () ;
} ;

azar::azar (int nb, int max)
{  int i ;
   srand(time(NULL));
   nbval = nb;
   val = new int [nb] ;
            for (i=0 ; i<nb ; i++)
            val[i] = rand()% (max+1) ;
}

azar::~azar ()
{  delete val ; // uso del destructor
}

void azar::visualiza ()          // para visualizar los valores
{  int i ;
            for (i=0 ; i<nbval ; i++)
                        cout << val[i] << " " ;
            cout << "\n" ;
}
```

```
void main()
{
        azar juego1 (10, 5) ;       // 10 valores entre 0 y 5
        juego1.visualiza () ;
        azar juego2 (6, 12) ;       // 6 valores entre 0 y 12
        juego2.visualiza () ;
}
```

Vemos aquí la utilidad del destructor para liberar la memoria dinámica reservada para el array.

6) Sobrecarga de funciones

En C++ podemos tener varios métodos con el mismo nombre, siempre que cada uno de ellos tengan distinta cantidad de argumentos o argumentos de distinto tipo.

```
// sobrecarga de funciones
#include <iostream.h>
class punto
{  int x, y ;
 public :
        punto () ;              // constructor 1 sin argumentos
        punto (int) ;           // constructor 2 con un argumento
        punto (int, int) ;      // constructor 3 con dos argumentos
        void visualiza () ;     // función visualiza sin argumentos
        void visualiza (char *) ;   // función visualiza con argumento (char * =  string)
} ;

punto::punto ()                         // constructor 1
{  x = 0 ; y = 0 ;
}

punto::punto (int abs)                  // constructor 2
{  x = y = abs ;
}

punto::punto (int abs, int ord)         // constructor 3
{  x = abs ; y = ord ;
}

void punto::visualiza ()                // función visualiza 1
{  cout << "Estoy en : " << x << " " << y << "\n" ;
}
void punto::visualiza (char * message)      // función visualiza 2
{  cout << message ; visualiza () ;
}

void main()
{
        punto a ;                   // llama al constructor 1
        a.visualiza () ;            // llama a la función visualiza 1
        punto b (5) ;               // llama constructor 2
        b.visualiza ("Punto b - ") ;    // llama función visualiza 2
        punto c (3, 12) ;           // llama al constructor 3
        c.visualiza ("Hola ---- ") ;    // llama a la función visualiza 2
}
```

7) Argumentos por defecto

Cuando en la llamada a un método (mensaje) no se especificara el valor de los argumentos, el método tomará el valor indicado en su prototipo a continuación del signo =.

```
// argumentos predeterminados o por omisión
#include <iostream.h>
class punto
{  int x, y ;
 public :
          punto (int=0, int=0) ; // constructor con dos argumentos (por emisión se inicializa en 0,0)
          void visualiza (char * = "Hola ---- ") ;   // función visualiza cuyo argumento por omisión
} ;

punto::punto (int abs, int ord)              // constructor
{  x = abs ; y = ord ;
}

void punto::visualiza (char * mensaje)         // función visualiza
{  cout << mensaje << " Estoy en : " << x << " " << y << "\n" ;
}

void main()
{
          punto a ;          // llama al constructor emplea parámetros por omisión
          a.visualiza () ;  // llama a la función visualiza y emplea el parámetro por omisión
          punto c (3, 12) ;             // llama al constructor
          c.visualiza ("Adios --- ") ;    // llama a la función visualiza
}
```

8) Métodos inline

Cuando un método se define dentro de la estructura de definición de la clase, es de tipo inline. Esto determina el modo en que el compilador linkea el código del método.

```
// métodos inline
#include <iostream.h>

class punto
{          int x;
          int y;
 public:
          void desplaza(int, int);
          void punto::visualiza()     // funcion inline
                    {          cout << "ubicacion: " << x << ", " << y << "\n";
                    }
          punto(int abs=0, int ord=0) // constructor inline
          {          cout << "constrctor del punto: " << abs << ", " << ord << "\n";
                              x=abs;
                              y=ord;
                    }
          ~punto();
};

void punto::desplaza(int dx, int dy)
{          x+=dx;
          y+=dy;
}

punto::~punto()
{          cout << "destructor del punto: " << x << ", " << y << "\n";
}
```

```
void main ()
{ punto a(5,2);
a.visualiza();
a.desplaza(-2,4);
a.visualiza();
punto b(1,-1);
b.visualiza();
}
```

9) Compilación separada de la clase

Con el objeto de reutilización de la clase, ésta debe compilarse separando la definición de la misma en un archivo (llamado Archivo de encabezamiento: con extensión .h) y la definición de los métodos en otro archivo aparte (llamado Archivo de implementación). El archivo de implementación se distribuirá **compilado** (archivo binario con extensión .obj) de modo que su código no será accesible para quien utilice la clase.

Archivo de encabezamiento: punto.h

```
#include <iostream.h>
// archivo de encabezamiento o definición de la clase punto
class punto
{       int x;
        int y;
 public:
        void inicializa(int, int);
        void desplaza(int, int);
        void visualiza();
        punto(int abs=0, int ord=0); // constructor
        ~punto();
};
```

Archivo de implementación: punto.cpp

```
#include <iostream.h>
#include "punto.h"
// archivo de implementacion de la clase punto
void punto::inicializa(int abs, int ord)
{       x=abs;
        y=ord;
}

void punto::desplaza(int dx, int dy)
{       x+=dx;
        y+=dy;
}

void punto::visualiza()
{       cout << "ubicacion: " << x << ", " << y << "\n";
}

punto::punto(int abs, int ord) // constructor inline
{       cout << "constructor del punto: " << abs << ", " << ord << "\n";
                x=abs;
                y=ord;
}
punto::~punto()
{       cout << "destructor del punto: " << x << ", " << y << "\n";
}
```

Utilización la clase en un programa: ptomain.cpp

```
#include <iostream.h>
#include "Punto.h"
void main ()
{ punto a, b(2,4);
a.visualiza();
a.desplaza(9,9);
a.visualiza();
```

```
b.visualiza();
}
```

10) Paso de objetos como parámetro. Parámetros implícitos y explícitos.

Los objetos de una clase pueden ser argumentos de métodos. Las llamadas a esos métodos (mensajes) reciben entonces los objetos pasados como argumentos de dos modos:

- **explícito**: es el modo habitual de paso de parámetros indicados entre paréntesis (tipo e identificador) a continuación del nombre del método;

- **implícito**: es alguno de los atributos del mismo objeto que recibe el mensaje y por lo tanto, no hace falta detallarlo en el prototipo del método. Ya que el método debe ser programado en forma genérica para que funcione con cualquier objeto de la clase, no es posible, en su código, referirse al objeto implícito por su nombre.

```
// Paso de objetos como parámetros: implícitos y explícitos y como valor de retorno
#include <iostream.h>
class punto
{
   int x, y ;
 public :
          punto (int abs=0, int ord=0)      // un constructor inline
                    { x=abs; y=ord ; }
          int coincide (punto) ;            // función miembro
          punto simetrico ();
          void visualiza();
} ;

int punto::coincide (punto pt)              // pt es un  parámetro explicito
{  if ( (pt.x == x) && (pt.y == y) )        // x  es un parámetro implícito ya que no se indica cual
                                            // es el objeto al que pertenece
          return 1 ;                        // observe la asimetría de la notación : pt.x y x
    else
          return 0 ;
}

punto punto::simetrico ( )
{    punto res ;
     res.x = -x ;
     res.y = -y ;
     return res ; // se retorna un objeto de la clase punto
}

void punto::visualiza()
{          cout << "ubicacion: " << x << ", " << y << "\n";
}

void main()
 {         punto a, b(1), c(1,0), d ;
           cout << "Verifico coincidencia de a con b: " << "\n";
           cout << "a y b : " << a.coincide(b) << " o " << b.coincide(a) << "\n" ;
            cout << "Verifico coincidencia de b con c: "<< "\n";
           cout << "b y c : " << b.coincide(c) << " o " << c.coincide(b) << "\n" ;
           d=c.simetrico();
           cout << "El simétrico de c está en  ";
           d.visualiza();
}
```

12) Paso de un objeto como parámetro por referencia

```
#include <iostream.h>
// Paso de un objeto como parámetro por referencia
class punto
{  int x, y ;
 public :
        punto (int abs=0, int ord=0)
                { x=abs; y=ord ; }
        int coincide (punto &) ;
} ;

int punto::coincide (punto & pt)
{  if ( (pt.x == x) && (pt.y == y) )
        return 1 ;
    else
        return 0 ;
}

void main()
{
        punto a, b(1), c(1,0) ;
        cout << "a y b : " << a.coincide(b) << " o " << b.coincide(a) << "\n" ;
        cout << "b y c : " << b.coincide(c) << " o " << c.coincide(b) << "\n" ;
}
```

12) Paso de la dirección de un objeto (puntero) como parámetro en una función miembro

Pasar la dirección de un objeto como parámetro (o sea un puntero al objeto) produce en mismo efecto que el paso por referencia, aunque el manejo de los atributos dentro del método es algo distinto:

```
#include <iostream.h>
// Paso de la dirección de un objeto como argumento
class punto
{  int x, y ;
 public :
        punto (int abs=0, int ord=0)       // constructor inline
                { x=abs; y=ord ; }
        int coincide (punto *) ;           // función miembro o método
} ;

int punto::coincide (punto * adpt)        // punto * denota que adpt es un puntero
{  if ( (adpt->x == x) && (adpt->y == y) )  // para acceder a una atributo se emplea el operador ->
                return 1 ;
    else
                return 0 ;
}

void main()
{  punto a, b(1), c(1,0) ;
        cout << "a y b : " << a.coincide(&b) << " o " << b.coincide(&a) << "\n" ;
        cout << "b y c : " << b.coincide(&c) << " o " << c.coincide(&b) << "\n" ;
}
```

13) Autoreferencia this

El operador **this** permite obtener, dentro de la definición de un método, la dirección del objeto implícito, o sea el que recibió el mensaje. Ya que el método debe ser programado en forma genérica para que funcione con cualquier objeto de la clase, no es posible, en su código, referirse al objeto implícito por su nombre para obtener un puntero mediante el operador * por eso se emplea this:

```
// Autoreferencia this
#include <iostream.h>
class punto
{
   int x, y ;
 public :
          punto (int abs=0, int ord=0)
                    { x=abs; y=ord ; }
          void visualiza () ;
} ;

void punto::visualiza ()              // this devuelve la dirección del objeto parámetro implícito
{  cout << "Direccion : " << this << " - Coordenadas " << x << " , " << y << "\n" ;
}

void main()
{
          punto a(5), b(3,15) ;
          a.visualiza ();
          b.visualiza ();
}
```

14) Atributos y métodos estáticos

Los **atributos estáticos** son atributos que tendrán el mismo valor para todos los objetos de la clase. Dicho de otro modo: son atributos de la clase y no de algún objeto en particular.
Los **métodos estáticos** son aquellos que cumplen alguna función totalmente independiente de un objeto determinado, es decir: su resultado será el mismo independientemente del objeto que lo invoque, más aun, pueden emplearse los métodos estáticos sin haber creado ningún objeto de la clase.
Analice el código y las salidas de este programa:

```
// atributos y métodos estáticos
#include <iostream.h>
class objeto
{  static int ctr ;               // atributo estático de la cantidad de objetos creados
 public :
   objeto () ;
   ~objeto() ;
   static void cuenta () ;   // método estático para mostrar la cantidad de objetos creados
} ;

int objeto::ctr = 0 ;          // inicialización del atributo estático

objeto::objeto ()              // constructor
{
          cout << "++ construccion : actualmente hay  " << ++ctr << " objetos\n" ;
}

objeto::~objeto ()          // destructor
{
          cout << "-- destruccion  : quedan ahora     " << --ctr << " objetos\n" ;
}
```

```
void objeto::cuenta ()
{  cout << "   llamada a la funcion : hay      " << ctr   << " objetos\n" ;
}

 void fct()
{
        cuenta u, v;            // esta función independiente crea dos objetos de la clase
}

void main()
{
   void fct () ;                  // declaración del prototipo de la función fct()
  objeto::cuenta () ;             // llamada a la función miembro (método) estática cuenta
                                  // aun cuando no existe ningún objeto de la clase
   objeto a ;                     // aquí se crea un objeto
   objeto::cuenta () ;            // y ese objeto también puede emplear el método estático
   fct () ;                       // invocación de la función fct()
   objeto::cuenta () ;
   objeto b ;
}
```

15) Objetos dinámicos

Pueden crearse punteros a objetos de una clase mediante el operador * como se hacía para crear punteros a variables en C. Una vez creado el **puntero a un objeto** debemos reservar memoria dinámica para el mismo mediante el operador **new**. Al hacer esto, se invocará automáticamente al constructor de la clase. De igual modo, el operador **delete** invoca automáticamente al destructor.

Analice el código y las salidas de este programa:

```
// objetos dinamicos
#include <iostream.h>
class punto
{
   int x, y ;
 public :
          punto (int abs, int ord)         // constructor inline
           { x=abs ; y=ord ;
                    cout << "++ Llamada al Constructor \n" ;
          }

          ~punto ()                        // destructor inline
          { cout << "-- Lllamada al Destructor \n" ;
          }
} ;

void fct (punto * adp)
{          cout << "** Comienza fct \n" ;
          delete adp ;                  // delete libera la memoria dinámica y llama al destructor
          cout << "** Termina fct \n" ;
}

void main()
{          void fct (punto *) ;            // prototipo de la función fct de prueba
          punto * adr ;                   // puntero a un objeto de la clase punto
          cout << "** Comienza main \n" ;
          adr = new punto (3,7) ;       // new reserva memoria dinámica y llama al constructor
```

```
        fct (adr)                       // se invoca a la función
        cout << "** Termina main \n" ;
        delete adr;
}
```

Note entonces, que el tratamiento de la memoria dinámica cuando se tienen **objetos dinámicos** es distinto al caso de que se tengan **atributos dinámicos** (ejemplo 5).

16) Problema de la copia de objetos que incluyen elementos dinámicos

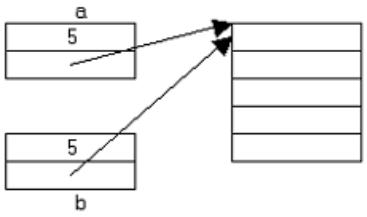
Como en el ejemplo anterior de la clase *azar,* las clases pueden contener atributos dinámicos (punteros) para obtener arrays de tamaño configurable (mediante new).
Este tipo de clase tiene problemas en los casos en que se necesite hacer una copia de objetos (clonación), debido a que la copia se realiza solo sobre la estructura estática del objeto y no sobre los elementos dinámicos.
Supongamos una clase *vect* que tiene dos atributos: uno es un puntero a double para generar arrays de tamaño configurable y el otro es un entero que guarda el tamaño del array.
Ahora creamos dos objetos de la clase *vect:*
- un objeto *a* cuyo puntero apuntará a un array de 5 elementos, mediante:

```
vect a(5);
```

- un objeto *b*, clonado del objeto *a,* mediante inicialización en la declaración con los valores del objeto *a.*

```
vect b=a;
```

Pero esto es lo que obtenemos:



Evidentemente no es lo deseado ya que ahora los dos objetos apuntan al mismo array en vez de obtenerse dos arrays separados.
El mismo problema se origina cuando se pasa un objeto de esta clase como parámetro, ya que en la llamada a la función o método, se produce una copia del parámetro real al parámetro formal.

Analice el código y las salidas del siguiente ejemplo:

```
// muestra el problema de la copia cuando la clase tiene miembros dinámicos
#include <iostream.h>
class vect
{
  public :
        int nelem ;                 // cantidad de elementos
        double * adr ;              // puntero a doubles = array de doubles

        vect (int n)                // constructor usual inline
        { adr = new double [nelem = n] ;
          cout << "+ const. usual   - direccion del objeto : " << this  << "\n" ;
          cout << "- direccion del vector : " << adr << "\n" ;
        }

        ~vect ()                    // destructor inline
        {  cout << " - Destr. objeto - direccion del objeto : "  << this << "\n";
                cout << "- direccion del vector : " << adr << "\n" ;
                delete adr ;
```

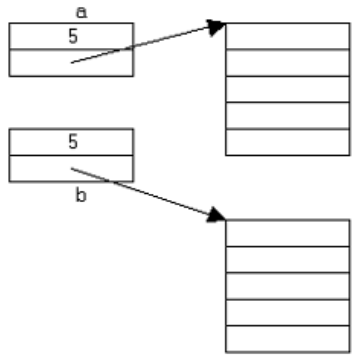
```
        }
         void fct(vect);          // una función de prueba
} ;

void vect::fct (vect c)
{
        cout << "+ const. usual   - dirección del objeto : " << &c  << "\n" ;
        cout << "- dirección del vector : " << c.adr << "\n" ;
        cout << "*** termina la función fct***\n" ; }

void main()
{ vect a(5) ;
 cout << "*** copia en la inicialización de un objeto ***\n" ;
 vect b=a;
 cout << "+ const. usual   - dirección del objeto : " << &b  << "\n" ;
 cout << "- dirección del vector : " << b.adr << "\n" ;
 cout << "*** copia en el paso de parámetro a una función ***\n" ;
 a.fct(b);
 cout << "*** termina la función main***\n" ;
```

17) Definición de un constructor de copia

He aquí el modo de solucionar la dificultad planteada en el ejemplo anterior: debemos definir un constructor especial, llamado constructor de copia, cuyo prototipo debe ser:

```
        vect (vect &)
```

este constructor tendrá la tarea de hacer la copia en la forma adecuada para obtener:




```
// definicion de un constructor de copia
#include <iostream.h>
class vect
{
        int nelem ;                 // cantidad de elementos
        double * adr ;              // puntero a doubles = array de doubles
  public :
        vect (int n)                // constructor usual inline
        { adr = new double [nelem = n] ;
          cout << "+ const. usual   - direccion del objeto : " << this  << "\n" ;
          cout << "- direccion del vector : " << adr << "\n" ;
        }
        vect (vect & );             // constructor de copia
        ~vect ()                    // detructor inline
        {  cout << " - Destr. objeto - direccion del objeto : "  << this << "\n";
                cout << "- direccion del vector : " << adr << "\n" ;
                delete adr ;
        }
} ;
```

```
vect::vect (const vect & v)                  // constructor de copia
{          nelem = v.nelem;
           adr = new double [nelem] ;     // crea un nuevo objeto
            for (int i=0 ; i<nelem ; i++)
                              adr[i]=v.adr[i]; // copia elementos del viejo
             cout << "+ constructor de copia - dirección del objeto : " << this << "\n";
             cout << "-  dirección del vector : " << adr << "\n" ;
}

void fct (vect b)                            // función de prueba
{ cout << "*** termina la función fct***\n" ; }

void main()
{ vect a(5) ;
 cout << "*** copia en la inicialización de un objeto ***\n" ;
 vect b=a;
 cout << "*** copia en el paso de parámetro por valor ***\n" ;
 fct(a);
 cout << "*** termina la función main***\n" ;
}
```

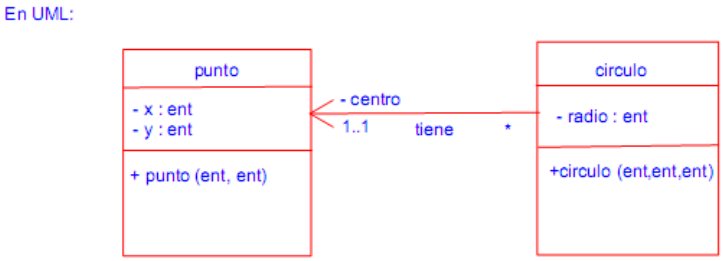
18) Objetos miembros – Clase contenedora

Los objetos de una clase pueden ser atributos de otra clase. Aquí la clase *circulo* tiene como atributo un objeto de la clase *punto*.

```
#include <iostream.h>
class punto
{ int x, y ;
 public :
  punto (int abs=0, int ord=0) // constructor inline de punto
    { x=abs ; y=ord ;
      cout << "Constructor de punto " << x << " " << y << "\n" ;
    }
} ;

class circulo                    // clase contenedora
{ punto centro ;                 // centro es un objeto de la clase punto
  int radio ;
 public :
  circulo (int , int , int) ;    // constructor de circulo
} ;

circulo:: circulo (int abs, int ord, int ray) : centro(abs, ord) // constructor de circulo
/* se establecen los valores de los parámetros para el constructor de punto que sera invoca-
do implicitamente*/
   { radio = ray ;
     cout << "Constructor de circulo " << radio << "\n" ;
   }

main()
{  circulo a (1,3,9) ;
}
```

UML (Lenguaje Unificado de Modelado) es un lenguaje para la escritura de proyectos de software. UML puede ser usado para visualizar, especificar, construir y documentar los componentes de un sistema de software, como se verá en otras materias,

En UML estas estructuras se originan como resultado de una relación de asociación entre dos clases: observe que *centro* es el rol de la relación y en el código da origen a un atributo.



19) Función independiente amiga de una clase

Cuando una clase declara a una función como amiga con la palabra reservada **friend** está permitiendo que esa función acceda a sus miembros privados, infringiendo el principio de encapsulación:

```
#include <iostream.h>
class punto
 { int x, y ;
  public :
            punto (int abs=0, int ord=0)          // un constructor ("inline")
                    { x=abs ; y=ord ; }
            friend int coincide (punto, punto) ;  // declaración de la función amiga independiente
 } ;

int coincide (punto p, punto q)                   // definición de la función amiga
{   if ((p.x == q.x) && (p.y == q.y))
            return 1 ;
    else
            return 0 ;}

void main()
{  punto a(1,0), b(1), c ;
   if (coincide (a,b))
            cout << "a coincide con b \n" ;
  else
            cout << "a y b son diferentes \n" ;
   if (coincide (a,c))
            cout << "a coincide con c \n" ;
   else
            cout << "a y c son diferentes \n" ;
}
```

20) Función miembro de una clase amiga de otra clase

```
class A ; // declara la clase A para que el compilador la reconozca

class B
 {
       .....
       int f (char, A) ; // declaración del método de la clase B
                         // que usa objetos de la clase A como parámetros
       .....
 };
```

```
class A
{
    // parte privada
       .....
    // parte publica
       friend int B::f (char, A) ;
       .....
} ;

  int B::f (char ..., A ...)  // definición del método
  {  // tiene acceso a los miembos privados
     // de las clases A y B
  }
```

21) Función amiga de varias clases

```
class B; // declara B como una clase
class A
{
     // parte privada
        .....
     // parte publica
        friend void f(A, B) ;
;
        .....
} ;

class B
{
      // parte privada
        .....
      // parte publica
         friend void f(A, B)
        .....
} ;
```

Todas las funciones de una clase son amigas de otra clase
En la clase A:

```
friend class B ;
```

22) Sobrecarga del operador + con una función amiga

Así como en el caso de los métodos y funciones, los operadores usualmente empleados con los tipos de datos básicos, pueden sobrecargarse para extender su aplicabilidad a las clases. Esto se logra definiendo una función o método con la palabra reservada **operator** precediendo al símbolo del operador que se desea sobrecargar:

```
// sobrecarga de operador + con función amiga para suma dos objetos de la clase punto
#include <iostream.h>
class punto
{ int x, y;
          public:
          punto (int abs=0, int ord=0) {x=abs; y=ord;}  // constructor
          friend punto operator+ (punto, punto);
          void visualiza() {cout << "coordenadas " << x << " " << y << "\n";}
};

punto operator + (punto a, punto b)
{         punto p;
```

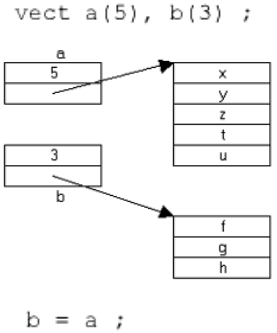
```
        p.x=a.x + b.x;
        p.y=a.y + b.y;
        return p;
}
void main()
{ punto a(1,2); a.visualiza();
  punto b(2,5); b.visualiza();
  punto c;
  c=a+b; c.visualiza();
  c=a+b+c; c.visualiza();
}
```

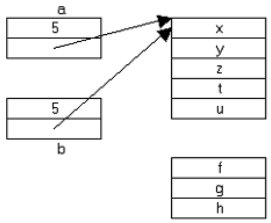
23) Sobrecarga del operador + con una función miembro

```
// sobrecarga de operador + con función miembro para sumar dos objetos de la clase punto
#include <iostream.h>
class punto
{ int x, y;
        public:
        punto (int abs=0, int ord=0) {x=abs; y=ord;}  // constructor
        punto operator+ (punto);
        void visualiza() {cout << "coordenadas " << x << " " << y << "\n";}
};

punto punto::operator + (punto a)
{       punto p;
        p.x=a.x + x;
        p.y=a.y + y;
        return p;
}

void main()
{ punto a(1,2); a.visualiza();
  punto b(2,5); b.visualiza();
  punto c;
  c=a+b; c.visualiza();
  c=a+b+c; c.visualiza();
}
```

24) Sobrecarga del operador = para evitar el problema de la copia de objetos que incluyen elementos dinámicos.

Con el constructor de copia no solucionamos el problema de la copia en los casos de **asignación de un objeto a otro,** no en la inicialización del objeto como en los ejemplos 16 y 17, sino en el cuerpo de un método, mediante una instrucción:



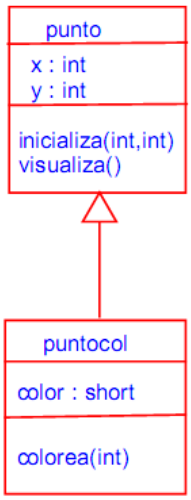


En este caso, el problema de la copia de elementos dinámicos se resuelve sobrecargando en operador de asignación =, de modo que el método operator= realice la copia en forma correcta:

```
# include <iostream.h>
class vect
{       int nelem;      // núemro de elementos
        double *dir;    // puntero a elementos
public:
        vect(int n)     // constructor
        {       dir = new double[nelem = n];
        cout << "contruye objeto: " << this << " dir vector: " << dir << "\n";
        }
        ~vect()         // destructor
        {    cout << "destruye objeto: " << this << " dir vector: " << dir<< "\n";
             delete dir;
        }
      vect & operator = (vect & );

};

vect & vect::operator = (vect & v)
{       cout << "llamada al operador = con direcciones: " << this << " y  " << &v << "\n";
        if (this != &v)
        {       cout << "borrado del vector dinámico en: " << dir << "\n";
                delete dir;
                nelem = v.nelem;
                dir = new double [nelem];
                cout << "nuevo vector dinámico en: " << dir << "\n";
                for (int i=0; i<nelem ; i++)
                        dir[i] = v.dir[i];
      }
        else
                cout << "no se hace nada";
         return *this;
}

void main()
{       vect a(5);
        vect b(5);
        b = a;
}
```

25) Herencia

La clase puntocol hereda de la clase punto. Esto se representa en UML como:



En C++ veamos la definición de la clase puntocol derivada de la clase punto:

```
#include<iostream.h>
#include<punto.h>

class puntocol: public punto    // los miembros publicos de la clase punto
                                // son tambien miembro publicos de la clase puntocol
{     short color;
 public:
      void colorea(short cl)
      {          color = cl;
      }
};

//  programa principal de prueba

void main()
{     puntocol p;
      p.inicializa(10, 20);       // el objeto p accede a las funciones miembro
                                  // de la clase base
      p.colorea(5);
      p.visualiza();
      p.desplaza(2,4);
      p.visualiza()
}
```

26) Definición de la clase puntocol derivada de la clase punto con redefinición de funciones

```
#include<iostream.h>
#include<punto.h>


class puntocol: public punto    // los miembros públicos de la clase punto
                                // son también miembro públicos de la clase puntocol
{     short color;
 public:
      void colorea(short cl)
                {color = cl; }
```

```
      void visualiza();                    // redefinición de la función visualiza
       void inicializa(int, int, short);   // redefinición de la función inicializa
};


void puntocol::visualiza()
{    punto::visualiza();
      cout << "color: " << color << "\n";
}

void puntocol::inicializa(int abs, int ord, short cl)
{     punto::inicializa(abs, ord);
      color=cl;
}

//  programa principal de prueba

void main()
{
      puntocol p;
      p.inicializa(10, 20,5);  p.visualiza();
      p.desplaza(2,4);         p.visualiza();
      p.colorea(2);            p.visualiza();
}
```

Los temas de POO no se agotan aquí. Debemos profundizar el empleo de la herencia y otros temas importantes como clases abstractas y métodos virtuales. Estos temas los desarrollaremos en las próximas unidades cuando estudiemos PHP orientado a objeto, y otros lenguajes que se basan en C++.