# Programación 4

## CONCEPTOS BÁSICOS DE IMPLEMENTACIÓN EN C++

## OBJETIVO

En este documento se presentan las construcciones básicas de orientación a objetos del lenguaje de programación C++, y recomendaciones de estilo particulares. No se incluye una descripción de la implementación de relaciones en C++.

## ELEMENTOS BÁSICOS

### CLASES, ATRIBUTOS, OPERACIONES Y MÉTODOS

A continuación se presentan los elementos básicos en la definición de una clase: la propia clase, sus atributos, sus operaciones y sus métodos. Las propiedades que sigan a un calificador `private` son privadas hasta encontrar un calificador `public`.

Por convención, el nombre de la clase comienza en mayúsculas, y las operaciones de acceso a los atributos se nombran igual que el atributo con el prefijo `get` o `set` según si la operación lee o escribe el valor del atributo respectivamente.

```
class Coordenada {
      private:
            int x;
            int y;
      public:
            void setX (int);
            int getX ();
            void setY (int);
            int getY ();
};
```

Al implementar el método de una operación, es necesario especificar el nombre de la clase a la que el método pertenece. Esto se hace previo al nombre de la operación. Toda operación de instancia se aplica sobre un objeto, al cual puede ser necesario acceder desde el código que implementa su método. El objeto sobre el cual se aplica una operación se denomina ***objeto implícito*** y es referenciado desde el método por el apuntador ***this***.

```
void Coordenada::setX (int val) {
      (*this).x = val;                  // Setea el atributo x del
}                                       // objeto implícito

int Coordenada::getX () {
      return this->x;                   // Es equivalente a (*this).x
}

void Coordenada::setY (int val) {
      y = val;                          // Se puede omitir el objeto
}                                       // implícito

int Coordenada::getY () {
      return y;
}
```

## CONSTRUCTORES

Los constructores son operaciones (sobrecargadas) que son invocadas al momento de construir un objeto. El código de un constructor se ejecuta inmediatamente después de que el objeto fue creado en memoria. Este código se utiliza en general para inicializar los atributos y para realizar alguna reserva de memoria dinámica (si la instancia creada la necesita). Todos los constructores tienen el mismo nombre (el de la clase) y se diferencian por los argumentos que reciben. Para los constructores no se especifica un tipo de retorno.

### Constructor por Defecto

Existe solamente uno por cada clase, y no recibe argumentos. Inicializa los campos con valores por defecto.

```
Coordenada::Coordenada() {
	x = 0;
	y = 0;
}
```

Invocaciones a este constructor pueden ser:

```
Coordenada c;                          // Un objeto estático

Coordenada *pc = new Coordenada();     // Un objeto dinámico
```

En el primer caso, el objeto se utiliza de manera local al procedimiento donde ejecuta este código. El objeto es ubicado automáticamente en el stack al entrar en el bloque del procedimiento, por lo tanto, al finalizar dicho procedimiento el objeto se destruye. Se lo denomina estático pues la memoria es reservada en tiempo de compilación.
En el segundo caso, la memoria es reservada en tiempo de ejecución. El objeto es ubicado en el heap, al finalizar el procedimiento que incluye a este código, la variable pc se destruye pero la memoria del objeto no es liberada.

### Constructor Común

Recibe argumentos y por lo general uno por cada atributo diferente que la clase tenga. Generalmente inicializa los atributos con esos valores.

```
Coordenada::Coordenada(int a,int b) {
	x = a;
	y = b;
}
```

Invocaciones a este constructor pueden ser:

```
Coordenada c(1,3);                     // Un objeto estático

Coordenada *pc = new Coordenada(7,4);  //Un objeto dinámico
```

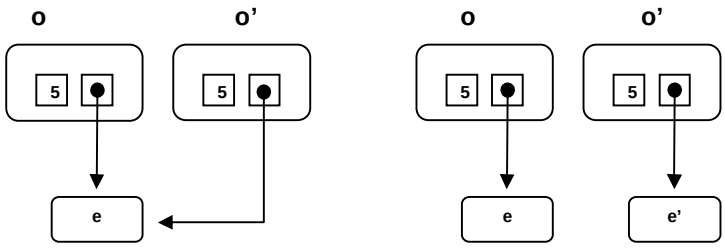
Puede haber varios constructores comunes, cada uno con diferente cantidad de argumentos.

## Constructor por Copia

Recibe como argumento a un objeto de la misma clase. Si no se implementa y el compilador debe utilizarlo, genera uno en forma automática que construye una copia plana del objeto pasado. La copia por defecto es entonces plana y en casos donde el objeto a crear no utiliza memoria dinámica, esto suele ser suficiente.

```
Coordenada::Coordenada(Coordenada &c) {
      x = c.x;
      y = c.y;
}
```

La copia plana crea una réplica exacta del objeto incluyendo los valores de los atributos. En caso de que uno de esos atributos sea un puntero, el valor de dicho atributo en el objeto copiado será exactamente igual al del original, produciéndose un alias. Si esto no es deseado, es necesario implementar al constructor por copia de forma tal que realice una copia en profundidad. De esa manera, los valores de los atributos que no sean punteros se replican, y la memoria referenciada por los atributos también.




Invocaciones a este constructor pueden ser:

```
Coordenada c2 = c1;                    // c1 ya existía

Coordenada * pc = new Coordenada(c);   // c ya existía
```

Otros casos donde se invoca al constructor por copia son:

- Al recibir un parámetro por valor. El parámetro efectivo es una copia del que se paso. Por ejemplo:

  ```
  void MiClase::oper(Coordenada c) {
     ...
  ```

  Por esta razón el parámetro que recibe el constructor por copia es pasado por referencia.

- Al retornar un valor. El objeto temporal que se crea es una copia del objeto retornado. Por ejemplo:

  ```
     ...
     return coord;
  }
  ```

## Destructor

El destructor es una operación que es invocada al momento de destruir un objeto. El código de un destructor se ejecuta inmediatamente antes de que el objeto sea eliminado de la memoria. Este código se utiliza en general para realizar alguna limpieza de memoria dinámica (si el objeto a destruir la utiliza). El destructor recibe el nombre de la clase antecedido por un '~'.

```
Coordenada::~Coordenada() {
      // no hay nada que hacer
}
```

Invocaciones (implícitas) a un destructor ocurren cuando:

- Si es un objeto estático, al llegar al final del bloque donde fue creado:

```
{
   Coordenada c;
   …
   …
} // Se destruye automáticamente porque es estático
```

- Si es un objeto dinámico, al aplicar delete sobre un puntero que lo referencie:

```
{
   Coordenada *pc;
   pc = new Coordenada;
   …
   …
   delete pc;    // De no invocar a delete, se elimina el puntero,
                 // pero la coordenada sigue en memoria
}
```

## TIPOS DE LOS ATRIBUTOS

Los tipos de los atributos de las clases pueden ser predefinidos (int, char, etc.), definidos por el usuario (typedef), o clases.
Las dos primeras categorías de atributos corresponden con la definición de atributo en UML, son datatypes.
Es posible también que el tipo de un atributo sea una clase X. En este caso es necesario distinguir dos situaciones:

- X en el diagrama de clases es un «datatype». Esto quiere decir que las instancias de X son data values y fue implementado como una clase porque el lenguaje no provee un tipo predefinido para X.

- X en el diagrama de clases es una clase. La necesidad de tener un atributo de este tipo viene dada por una visibilidad «association» en el diagrama. Estos atributos se representan mediante ***referencias*** y se discutirán en la implementación de asociaciones.

Suponiendo que Coordenada es implementada como una clase pero corresponde a un «datatype», se puede definir la clase Segmento de esta forma.

```
class Segmento {
	private:
		Coordenada ext1;
		Coordenada ext2;
	public:
		Segmento();
		Segmento(Coordenada,Coordenada);
		float getNorma();
		//operaciones de acceso
};
```

Notar que los atributos ext1 y ext2 corresponden al primer caso antes descrito. Veamos cómo se implementan los constructores de Segmento:

```
Segmento::Segmento() {
	// invoca automáticamente al constructor por defecto de
	// Coordenada para crear ext1 y ext2
}

Segmento::Segmento(Coordenada c1, Coordenada c2) : ext1(c1),ext2(c2) {
	// invoca al constructor por copia de Coordenada
	// para crear ext1 y ext2
}
```

También se podría haber definido otro constructor común más:

```
Segmento::Segmento(int a, int b, int c, int d) : ext1(a,b), ext2(c,d) {
	// invoca al constructor común de Coordenada para
	// crear ext1 y ext2
}
```

## EJEMPLO

A continuación se presenta un ejemplo que ayuda a comprender las invocaciones a los distintos tipos de constructores de una clase.

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

class MiClase {
	private:
		int dato;
	public:
		MiClase();                          // constructor por defecto
		MiClase(int);                       // constructor común
		MiClase(const MiClase &);           // constructor por copia
		~MiClase();                         // destructor
		MiClase operator=(MiClase);         // operador de asignación
		int getDato();                      // selectora de “dato”
		void funcion1(MiClase);
		void funcion2(MiClase &);
		MiClase funcion3(MiClase);
		MiClase funcion4(MiClase);
};
```

La implementación de las operaciones de la clase `MiClase` es la siguiente:

```
MiClase::MiClase(){
	dato = 0;
	cout << "Soy el constructor por defecto  ";
	cout << " Mi Dato es " << dato << '\n';
}

MiClase::MiClase(int i){
	dato = i;
	cout << "Soy el constructor comun ";
	cout << " Mi dato es " << dato << '\n';
}

MiClase::MiClase(const MiClase &m){
	dato = m.dato;
	cout << "Soy el constructor de copia ";
	cout << " Mi Dato es " << dato << '\n';
}

MiClase::~MiClase(){
	cout << "Soy el destructor ";
	cout << " Mi dato era " << dato << '\n';
}

MiClase MiClase::operator=(MiClase m){
	dato = m.dato;
	cout << "Soy el operador de asignacion" << '\n';
	return *this;
}
int MiClase::getDato(){
	return dato;
}

void MiClase::funcion1(MiClase m){
	MiClase local(5);

	cout << "Esta es la funcion 1 " << '\n';
	cout << '\t' <<" El dato del objeto local es " << local.dato << '\n';
	cout << '\t' << " El dato del objeto implicito es " << dato << '\n';
	cout << '\t' << " El dato del parametro es " << m.dato << '\n';
}

void MiClase::funcion2(MiClase &m){
	cout << " Esta es la funcion 2" << '\n';
}

MiClase MiClase::funcion3(MiClase m){
	MiClase local;

	local.dato = m.dato * dato;
	cout << "Esta es la funcion 3 " << '\n';
	cout << '\t' << " El dato del objeto local es " << local.dato << '\n';
	return local;
}

MiClase MiClase::funcion4(MiClase m){
	 MiClase local(MiClase(10));
	cout << "Esta es la funcion 4 " << '\n';
	return local;
}
```

El siguiente programa principal crea diferentes instancias de clase `MiClase` e invoca operaciones sobre ellas.

```
1   int main(){
2
3       MiClase m1(10);
4       MiClase m2=m1;
5       MiClase m3=30;
6       MiClase M[5]={m1,m2,m3,m1};
7       MiClase * pm;
8
9       pm = new MiClase[2];
10
11      for (int i=0; i<5;i++)
12        cout << M[i].getDato() << " ";
13      cout << '\n';
14
15      delete []pm;
16
17      cout << '\n';
18      cout << "antes de la asignacion" << '\n';
19      m2=m1;
20      cout << "despues de la asignacion" << '\n';
21      cout << '\n';
22
23      cout << "Antes de invocar a funcion1" << '\n';
24      m1.funcion1(m3);
25      cout << "Despues de invocar a funcion1" << '\n';
26      cout << '\n';
27
28      cout << "Antes de invocar a funcion2" << '\n';
29      m1.funcion2(m3);
30      cout << "Despues de invocar a funcion2" << '\n';
31      cout << '\n';
32
33      cout << "Antes de invocar a funcion3" << '\n';
34      m2=m1.funcion3(m3);
35      cout << "Despues de invocar a funcion3" << '\n';
36      cout << '\n';
37
38      cout << "Antes de invocar a funcion4" << '\n';
39      m2=m1.funcion4(m3);
40      cout << "Despues de invocar a funcion4" << '\n';
41      cout << '\n';
42
43      return 0;
44  }
```

A continuación se reproduce la salida producida por el programa y se comentan algunas de las invocaciones.

```
Soy el constructor comun  Mi dato es 10              linea 3
Soy el constructor de copia  Mi Dato es 10           linea 4
Soy el constructor comun  Mi dato es 30              linea 5
Soy el constructor de copia  Mi Dato es 10
Soy el constructor de copia  Mi Dato es 10           array de linea 6
Soy el constructor de copia  Mi Dato es 30
Soy el constructor de copia  Mi Dato es 10
Soy el constructor por defecto   Mi Dato es 0        quinto elemento de linea 6
Soy el constructor por defecto   Mi Dato es 0
Soy el constructor por defecto   Mi Dato es 0        array de linea 9
10 10 30 10 0                                        for de lineas 11-12
Soy el destructor  Mi dato era 0
Soy el destructor  Mi dato era 0                     elim. de array en linea 15

antes de la asignacion
Soy el constructor de copia  Mi Dato es 10           m es param. formal de operator=
Soy el operador de asignacion
Soy el constructor de copia  Mi Dato es 10           tmp es copia de *this en return
Soy el destructor   Mi dato era 10                   elim. de m
Soy el destructor   Mi dato era 10                   elim. de tmp
despues de la asignacion

Antes de invocar a funcion1
Soy el constructor de copia  Mi Dato es 30           m es param. formal de funcion1
Soy el constructor comun  Mi dato es 5               constr. de local
Esta es la funcion 1
       El dato del objeto local es 5
       El dato del objeto implicito es 10
       El dato del parametro es 30
Soy el destructor  Mi dato era 5                     elim. de local
Soy el destructor  Mi dato era 30                    elim. de m
Despues de invocar a Function1

Antes de invocar a Function2                         no hay copia del parámetro m
 Esta es la funcion 2                                porque fue pasado por referencia
Despues de invocar a Function2

Antes de invocar a Function3
Soy el constructor de copia  Mi Dato es 30           m es param. formal de funcion3
Soy el constructor por defecto   Mi Dato es 0       constr. de local
Esta es la funcion 3
       El dato del objeto local es 300
Soy el operador de asignacion
Soy el constructor de copia  Mi Dato es 300          m (de operator=) es copia de local#
Soy el destructor  Mi dato era 300                   elim. de local
Soy el destructor  Mi dato era 300                   elim. de m (de operator=)
Soy el destructor  Mi dato era 30                    elim. de m (de funcion3)
Despues de invocar a Function3

Antes de invocar a Function4
Soy el constructor de copia  Mi Dato es 30           m es param. formal de funcion4
Soy el constructor comun  Mi dato es 10              local es el objeto creado con 10*
Esta es la funcion 4
Soy el operador de asignacion
Soy el constructor de copia  Mi Dato es 10           m (de operator=) es copia de local
Soy el destructor  Mi dato era 10                    elim. de local
Soy el destructor  Mi dato era 10                    elim. de m (de operator=)
Soy el destructor  Mi dato era 30                    elim. de m (de funcion4)
Despues de invocar a Function4
```

# El compilador realizó una optimización. En lugar de crear un temporal como copia de `local`, para luego crear el parámetro formal `m` como copia del temporal, crea directamente `m` como copia de `local`.

* El compilador realizó otra optimización. En lugar de crear un objeto temporal con el constructor común para luego construir `local` como una copia de éste, construyó un solo objeto: `local`.

```
Soy el destructor  Mi dato era 0                    elim. de M[4]
Soy el destructor  Mi dato era 10                   elim. de M[3]
Soy el destructor  Mi dato era 30                   elim. de M[2]
Soy el destructor  Mi dato era 10                   elim. de M[1]
Soy el destructor  Mi dato era 10                   elim. de M[0]
Soy el destructor  Mi dato era 30                   elim. de m3
Soy el destructor  Mi dato era 10                   elim. de m2
Soy el destructor  Mi dato era 10                   elim. de m1
```

Notar que los objetos estáticos se destruyen automáticamente al llegar al final del bloque donde fueron definidos, y en el orden inverso al que fueron creados.