Constructores y Destructores

Hasta el momento, solíamos decir que el contenido de las variables declaradas sin asignarles nada es “basura”, es decir, un valor desconocido que tiene que ver con lo que contengan las posiciones de memoria que el programa utilizó para localizar a la variable. Sabemos que la única posibilidad de garantizar el contenido de una variable es asignar un valor cierto, en su declaración, o con posterioridad a ella.

A diferencia de lo anterior, las clases proveen un mecanismo que nos permite que el contenido inicial de las propiedades de los objetos sea un valor conocido, definido de acuerdo a las necesidades del programa en desarrollo por medio de los **constructores**. Los constructores también nos posibilitan asignarle valores explícitos a las propiedades de los objetos al momento de la declaración de estos.

Por ejemplo, si tuviéramos una clase para representar círculos (la clase Circulo) en la pantalla:

```
int main(){
    Circulo c1(512, 384, 300, "rojo", "rojo");

    return 0;
}
```

donde los valores podrían representar lo siguiente:
512: posición inicial en el eje x
384: posición inicial en el eje y
300: diámetro del círculo
rojo: color del borde
rojo: color de fondo

al momento de la declaración del objeto c1, además le asignamos valores iniciales a sus propiedades.

O para el caso de un objeto de una clase para representar puntos (clase Punto)

```
int main(){
    Punto obj(10,5, "AZUL");

    return 0;
}
```

donde 10 y 5 podrían representar la posición en la pantalla y AZUL el color.

En ambos casos hemos utilizado constructores, que están definidos dentro de las clases.

Los constructores son métodos especiales que se caracterizan por lo siguiente:

- **tienen el mismo nombre de la clase;**
- **no devuelven ningún valor, ni siquiera void**
- **se ejecutan sin necesidades de ser invocados, en el momento de la declaración de cada objeto.**

Para una clase para representar artículos (clase Articulo), un constructor podría ser el siguiente:

```
class Articulo{
    private:
        char codArt[5];
        char descripcion[30];
        int stock;
        float pu;

    public:
         Articulo(){
            pu=0;
            stock=0;
            strcpy(codArt,"0000");
            strcpy(descripcion, "0000");
        }
```

como el constructor no tiene parámetros, no se podrá asignarle valores al declarar los objetos; todos los objetos tendrán como valores iniciales los establecidos en el constructor.

Si necesitáramos hacer una asignación de valores al momento de la declaración de un objeto Articulo, tendríamos que modificar el constructor anterior, o agregar otro constructor con parámetros, como el siguiente:

```
         Articulo::Articulo(const char *c, const char *d, float p, int
s){
            strcpy(codArt, c);
            strcpy(descripcion, d);
            pu=p;
            stock=s;
        }
```

Pero, si un constructor es un método, y un método es una función, ¿pueden existir dos funciones de idéntico alcance y nombre? La respuesta es sí: C++ admite funciones con el mismo nombre, siempre y cuando éstas puedan ser diferenciadas por el compilador por los parámetros que reciben. Esto se llama **sobrecarga de funciones**, y es aplicable a cualquier función, no sólo a los constructores.

C++ tiene la capacidad de decidir cuál de las funciones de mismo nombre utilizar. En nuestro caso:

```
int main(){
        Articulo obj; //se ejecutará el constructor sin parámetros
        Articulo obj2 ("hola", "chau",3.5,0); // se ejecutará el
constructor con parámetros.

    return 0;
}
```

Otra de las características de las funciones en C++ es que admiten **parámetros por omisión**, lo que significa que una función definida para recibir un número determinado de parámetos, puede recibir menos o ninguno, si a esos parámetros se le asignan valores (serán los valores que se utilizarán en caso de no enviarlos). Por ejemplo:

```
    void funcion(int n1=3, int n2= 5){

    }
```

es una función definida con 2 parámetros enteros. Podría ser llamada correctamente de las siguientes formas:

```
int main(){
      funcion(10,50); // se envían los 2 parámetros; n1 obtendría el
valor 10, y n2 el valor 50
     funcion(10);    // se envía sólo 1 parámetro; n1 obtendría el valor
10, y n2 el valor 5, que es el definido por defecto
     funcion();       // no se envían parámetros; n1 obtendría el valor
3, y n2 el valor 5

    return 0;
}
```

Desde esta perspectiva, podríamos hacer un solo constructor, que sirva de reemplazo de los dos vistos:

```
          Articulo(int _codArt=-1, const char _descr[30]="nada", float
_pu=0, int _stock=0){
            //codArt=_codArt;
            strcpy(descripcion, _descr);
            pu=_pu;
            stock=_stock;
        }
```

En resumen, podemos, si es de utilidad para nuestro programa, agregar un constructor o varios en una clase. En caso de que utilicemos más de 1 constructor, debemos garantizar que los constructores se diferencien entre sí, para garantizar que no exista ambigüedad, es decir que el programa pueda elegir sin problemas cuál de todos los constructores utilizar. Un caso de ambigüedad se presentaría si en la clase Articulo agregáramos el constructor sin parámetros (el primero que vimos), y el que tiene parámetros por omisión (el último), ya que al declarar un objeto sin parámetros, no podría distinguirse cuál de los dos utilizar.

En la mayoría de los casos, no es obligatorio agregar constructores. Cada clase tiene un constructor por defecto; los utilizaremos cuando –como ya se dijo- sean de utilidad para el desarrollo de los programas.

Además de constructores, las clases nos posibilitan crear **destructores**.

Los destructores son métodos especiales que tiene las siguientes características:
- **tienen el mismo nombre que la clase que los contiene, antecedido por el carácter ~**
- **no devuelven valor, ni aceptan parámetros.**
- **se ejecutan automáticamente cuando un objeto termina su ciclo de vida**
- **sólo se admite un destructor en una clase**

Un ejemplo de un destructor para la clase Articulo sería:

```
Articulo::~Articulo(){
cout<<endl<<"HA MUERTO EL OBJETO. VIVA EL OBJETO!!";
}
```

El ejemplo es un tanto ridículo, ya que no hace nada que sirva; sólo imprime una leyenda. Se lo presenta a modo de prueba, y porque no es de utilidad un destructor para esta clase.

Generalmente el destructor es utilizado para realizar tareas que eviten tener que prestar atención a detalles importantes. Por ejemplo, una clase Vector, o una clase Cadena, podrían pedir memoria en su constructor para construir un objeto, y el destructor liberar la memoria pedida. De esta manera el programador no tiene que preocuparse por estos detalles que podrían ocasionar problemas, ya que la propia clase se encarga tanto de pedir como de liberar los recursos.

Como hemos visto en la clase sobre el tema, la asignación dinámica se utiliza para crear variables de memoria, por lo general vectores o matrices. Usa en C++ los siguientes operadores:
- new: para pedir memoria
- delete: para devolver la memoria asignada

Veamos el caso siguiente:

Diseñar una clase de nombre Cadena que almacene cadenas de texto que ocupen en memoria el tamaño estrictamente necesario, es decir el equivalente a la cantidad de caracteres que contiene más el carácter terminador.

Por el momento la cadena que almacenará el objeto será la que se indique en el constructor.

La idea de esta clase es superar las limitaciones que tenemos al usar vectores de caracteres para representar cadenas, ya que por lo general debemos sobredimensionar los vectores para garantizar que se puedan almacenar todos los caracteres necesarios.

Por ejemplo si se declarara

```
int main(){
    Cadena cad1("Hola");

    return 0;
}
```

el objeto cad1 debería ocupar 5 caracteres.

```
class Cadena{
  private:
        char *p;
  public:
    Cadena(const char *cad="Hola"){
        int tam=strlen(cad);     // se mide la cantidad de caracteres
            p=new char[tam+1];   // se pide memoria para el vector
        if(p==NULL)exit(1);      // se chequea
        strcpy(p,cad);           // se copia la cadena recibida
      }
    void Mostrar(){cout<<p;};
    char *getP(){return p;}
    void setP(char *cad){
        delete p;
        int tam=strlen(cad);
        p=new char[tam+1];
        if(p==NULL)exit(1);
        strcpy(p,cad);
      }
     ~Cadena(){delete p;}        // se devuelve la memoria pedida
  };
```

En este caso el constructor es utilizado para construir el vector de caracteres para almacenar la cadena. Veamos en detalle el código del constructor

```
Cadena::Cadena(const char *cad="Hola"){
    int tam=strlen(cad);    // se almacena en la variable tam el tamaño
de la cadena que se recibe como parámetro
      p=new  char[tam+1];         //  se pide memoria para la cantidad
necesaria de caracteres
    if(p==NULL)exit(1);     // se chequea que se haya podido asignar la
memoria
     strcpy(p,cad);             // se copia en el vector dinámico p la
cadena recibida
    }
```

Por su parte el destructor libera la memoria pedida mediante la instrucción delete p;

De esta manera, al programar usando esta clase se ocultan los detalles de como se pide memoria, cuando se la libera, etc. No hay que preocuparse por esto ya que la clase se encarga de resolver el problema.

El siguiente programa utiliza un objeto Cadena

```
int main(){
    Cadena cad1("Prueba de clase Cadena");
    cad1.Mostrar();

    return 0;
}
```

A la clase Cadena le iremos agregando funcionalidad a medida que avancemos en el curso.