C++: Clases y objetos, Constructores, Sobrecarga de constructores, Destructores

Clases y objetos de C++

Los objetos y las clases se utilizan para agrupar *funciones* y *datos* relacionados en un solo lugar en C++.

Supongamos que necesitamos almacenar la `longitud`, el `ancho` y la `altura` de una habitación rectangular y calcular su `área` y `volumen`.

Para realizar esta tarea, podemos crear tres variables, por ejemplo, `longitud`, `ancho` y `altura`, junto con las funciones `calcula_area()` y `calcula_volume()`.

Sin embargo, en C++, en lugar de crear variables y funciones independientes, también podemos agrupar los datos y las funciones relacionados en un solo lugar (mediante la creación de **objetos**).

Este paradigma de programación se conoce como *Programación Orientada a Objetos*.

Pero antes de poder crear objetos y usarlos en C++, primero debemos aprender sobre las **clases**.

Clase de C++

Una clase es un plano del objeto.

Podemos pensar en una clase como un boceto (prototipo) de una casa.

Contiene todos los detalles sobre los pisos, las puertas, las ventanas, etc. Construimos la casa en base a estas descripciones.

La casa es el objeto.

Crear una clase

En C++, una clase se define con la palabra clave `class` seguida del nombre de la clase.

El cuerpo de la clase se define entre llaves y termina con un punto y coma al final.

```
class ClassName {
   // some data
   // some functions
};
```

Por ejemplo:

```
class Habitacion {
    public:
        double longitud;
        double ancho;
        double altura;

        double calcular_area(){
            return longitud * ancho;
        }

        double calcular_volumen(){
            return longitud * ancho * altura;
        }

};
```

Aquí, definimos una clase llamada `Habitacion`.

Las variables `longitud`, `ancho` y `altura` declaradas dentro de la clase se conocen como **miembros de datos**.

Y las funciones `calcula_area()` y `calcula_volumen()` se conocen como **funciones miembro** de una clase.

Objetos de C++

Cuando se define una clase, solo se define la *especificación del objeto*; no se asigna memoria ni almacenamiento.

Para utilizar los datos y las funciones de acceso definidas en la clase, necesitamos **crear objetos**.

Sintaxis para definir un objeto en C++

```
ClassName object_name;
```

Podemos crear objetos de la clase `Habitacion` (definida en el ejemplo anterior) de la siguiente manera:

```
// función ejemplo
void funcion_ejemplo() {
    // create objects
    Habitacion habitacion1, habitacion2;
}

int main(){
    // crear objetos
    Habitacion habitacion3, habitacion4;
}
```

Se muestra que se crean dos objetos: `habitacion1` y `habitacion2` de la clase `Habitacion` en `funcion_ejemplo()`. De manera similar, se crean los objetos `habitacion3` y `habitacion4`, de la clase `Habitacion` en `main()`.

Como podemos ver, podemos crear objetos de una clase en cualquier función del programa.

También podemos crear objetos de una clase dentro de la propia clase o en otras clases.

Además, podemos crear tantos objetos como queramos de una sola clase.

Acceso a miembros de datos y funciones miembro en C++

Podemos acceder a los *miembros de datos* y *funciones miembro* de una clase mediante el operador **.** (punto).

Por ejemplo,

```
habitacion2.calcula_area();
```

Esto llamará a la función `calcula_area()` en la clase `Habitación` para el objeto `habitacion2`.

De manera similar, se puede acceder a los *miembros de datos* como:

```
habitacion1.longitud = 5.5;
```

En este caso, inicializa la variable de `longitud` de `habitacion1` a 5.5.

Ejemplo: Objeto y clase en programación en C++

```
// Programa para ilustrar el funcionamiento de objetos
// y clases en programación C++

#include <iostream>
using namespace std;

// create a class
class Room {

   public:
    double longitud;
    double ancho;
    double altura;

    double calcula_area() {
        return longitud * ancho;
```

```
    }

    double calcula_volumen() {
        return longitud * ancho * altura;
    }
};

int main() {

    // crea objeto de la clase Habitacion
    Habitacion habitacion1;

    // asignar valores a los miembros de datos
    habitacion1.longitud = 42.5;
    habitacion1.ancho = 30.8;
    habitacion1.altura = 19.2;

    // Calcular y mostrar el área y el volumen de la habitación.
    cout << "Area de la Habitación =  " << habitacion1.calculate_area() << endl;
    cout << "Volumen de la Habitación =  " << habitacion1.calculate_volumen() << endl;

    return 0;
}
```

Salida

```
Area de la Habotación =  1309
Volumen de la Habitación =  25132.8
```

En este programa, hemos utilizado la clase `Habitacion` y su objeto `habitacion1` para calcular el `área` y el `volumen` de una habitación.

En `main()`, asignamos los valores de `longitud`, `ancho` y `altura` con el código:

```
habitacion1.longitud = 42.5;
habitacion1.ancho = 30.8;
habitacion1.altura = 19.2;
```

Luego, llamamos a las funciones `calcula_area()` y `calcula_volumen()` para realizar los cálculos necesarios.

Observe el uso de la palabra clave `public` en el programa. Esto significa que los miembros son públicos y se puede acceder a ellos desde cualquier lugar del programa.

Constructores de C++

Un constructor es una **función** miembro especial que se llama automáticamente cuando se crea un **objeto**.

En C++, un constructor tiene el mismo nombre que el de la clase y no tiene un tipo de retorno. Por ejemplo,

```
class  Wall {
  public:
    // Crear un constructor
    Wall() {
      // code
    }
};
```

Aquí, la función `Wall()` es un constructor de la clase `Wall`. Observe que el constructor

- Tiene el mismo nombre que la clase,
- No tiene un tipo de retorno
- Es `public`

Constructor predeterminado de C++

Un constructor sin parámetros se conoce como **constructor predeterminado**. Por ejemplo,

```
// Programa C++ para demostrar el uso del constructor predeterminado

#include <iostream>
using namespace std;

// declare a class
class  Wall {
  private:
    double longitud;

  public:
    // constructor predeterminado para inicializar la variable
    Wall()
      : longitud{5.5} {
      cout << "Creando un muro." << endl;
      cout << "Longitud = " << longitud << endl;
    }
};

int main() {
  Wall wall1;
  return 0;
}
```

Salida

```
Creando un muro
Longitud = 5.5
```

Aquí, cuando se crea el objeto `wall1`, se llama al constructor `Wall().longitud5.5` se invoca cuando se llama al constructor y establece la variable `longitud` del objeto en 5.5.

Nota: Si no hemos definido ningún constructor, constructor de copia o constructor de movimiento en nuestra clase, entonces el compilador de C++ **creará automáticamente un constructor predeterminado sin parámetros y con el cuerpo vacío**.

Constructor predeterminado

Cuando tenemos que confiar en el constructor predeterminado/*default constructor* para inicializar las variables miembro de una clase, debemos marcar explícitamente el constructor como predeterminado de la siguiente manera:

```
Wall() = default;
```

Si queremos establecer un valor predeterminado, entonces debemos utilizar la inicialización de valores. Es decir, incluimos el valor predeterminado entre llaves en la declaración de las variables miembro de la siguiente manera.

```
double longitud {5.5};
```

Veamos un ejemplo:

```
// C++ program to demonstrate the use of defaulted constructor

#include <iostream>
using namespace std;

// declarar una clase
class  Wall {
  private:
    double longitud {5.5};

  public:
    // constructor predeterminado para inicializar la variable
```

```
    Wall() = default;

    void print_longitud() {
     cout << "longitud = " << longitud << endl;
    }
};

int main() {
  Wall wall1;
  wall1.print_longitud();
  return 0;
}
```

Salida

```
longitud = 5.5
```

Constructor parametrizado de C++

En C++, un constructor con parámetros se conoce como **constructor parametrizado**. Este es el método preferido para inicializar datos de miembros. Por ejemplo,

```
// Programa en C++ para calcular el área de una pared

#include <iostream>
using namespace std;

// declarar una clase
class Wall {
  private:
    double longitud;
    double altura;

  public:
    // constructor parametrizado para inicializar variables
    Wall(double lon, double alt)
      : longitud{lon}
      , altura{alt} {
    }

    double calculaArea() {
      return longitud * altura;
    }
};

int main() {
  // crear objeto e inicializar miembros de datos
  Wall wall1(10.5, 8.6);
  Wall wall2(8.5, 6.3);

  cout << "Área de Wall 1: " << wall1.calculaArea() << endl;
  cout << "Área de Wall 2: " << wall2.calculaArea();

  return 0;
}
```

Salida

```
Área de Wall 1: 90.3
Área de Wall 2: 53.55
```

Aquí, hemos definido un constructor parametrizado `Wall()` que tiene dos parámetros: `double lon` y `double alt`. Los valores contenidos en estos parámetros se utilizan para inicializar las variables miembro longitud y altura.

`: longitud{lon}, altura{alt}` es la lista de inicializadores de miembros.

- `longitud{lon}` inicializa la variable miembro `longitud` con el valor del parámetro `lon`
- `altura{alt}` inicializa la variable miembro `altura` con el valor del parámetro `alt`.

Cuando creamos un objeto de la clase `Wall`, pasamos los valores de las variables miembro como argumentos. El código para esto es:

```
Wall wall1(10.5, 8.6);
Wall wall2(8.5, 6.3);
```

Con las variables miembro inicializadas de esta manera, ahora podemos calcular el área de la pared con el método `calculaArea()`.

Nota: Un constructor se utiliza principalmente para inicializar objetos. También se utilizan para ejecutar un código predeterminado cuando se crea un objeto.

Lista de inicializadores de miembros de C++

Considere el constructor:

```
Wall(double lon, double alt)
  : longitud{lon}
  , altura{alt} {
}
```

Aquí,

```
: longitud{lon}
,  altura{alt}
```

es la lista de inicializadores de miembros.

La lista de inicializadores de miembros se utiliza para inicializar las variables miembro de una clase.

El orden o la inicialización de las variables miembro se realiza según el orden de su declaración en la clase, en lugar de su declaración en la lista de inicializadores de miembros.

Dado que las variables miembro se declaran en la clase en el siguiente orden:

```
double longitud;
double altura;
```

La variable de `longitud` se inicializará primero incluso si definimos nuestro constructor de la siguiente manera:

```
Wall(double alt, double lon)
  : altura{alt}
  , longitud{lon} {
}
```

Constructor de copia de C++

El constructor de copia de C++ se utiliza para copiar datos de un objeto a otro. Por ejemplo,

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

// declarar una clase
class Wall {
  private:
    double longitud;
    double altura;

  public:

    // inicializar variables con un constructor parametrizado
    Wall(double lon, double alt)
      : longitud{lon}
      , altura{alt} {
    }

    // constructor de copias con un objeto Wall como parámetro
    // copia datos del parámetro obj
    Wall(const Wall& obj)
```

```
      : longitud{obj.longitud}
      , altura{obj.altura} {
    }

double calculaArea() {
      return longitud * altura;
    }
};

int main() {
  // Crea un objeto de la clase Wall
  Wall wall1(10.5, 8.6);

  // Copiar el contenido de wall1 a wall2
  Wall wall2 = wall1;

  // áreas de impresión de wall1 y wall2
  cout << "Área de Wall 1: " << wall1.calculaArea() << endl;
  cout << "Área de Wall 2: " << wall2.calculaArea();

  return 0;
}
```

Salida

```
Área de Wall 1: 90.3
Área de Wall 2: 90.3
```

En este programa, hemos utilizado un constructor de copia para copiar el contenido de un objeto de la clase `Wall` a otro. El código del constructor de copia es:

```
Wall(const Wall& obj)
    : longitud{obj.longitud}
    , altura{obj.altura} {
}
```

Observe que el parámetro de este constructor tiene la dirección de un objeto de la clase `Wall`.

Luego asignamos los valores de las variables del objeto `obj` a las variables correspondientes del objeto, llamando al constructor de copia. Así es como se copia el contenido del objeto.

En `main()`, creamos dos objetos `wall1` y `wall2` y luego copiamos el contenido de `wall1` a `wall2`:

```
// Copiar el contenido de wall1 a wall2
Wall wall2 = wall1;
```

Aquí, el objeto `wall2` llama a su constructor de copia pasando la referencia del objeto `wall1` como argumento.

Constructor de copia predeterminado de C++

Si no definimos ningún constructor de copia, constructor de movimiento o asignación de movimiento en nuestra clase, entonces el compilador de C++ creará automáticamente un constructor de copia predeterminado que realiza la asignación de copia miembro por miembro. Esto es suficiente en la mayoría de los casos. Por ejemplo,

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

// declare a class
class Wall {
  private:
    double longitud;
    double altura;

  public:

    // inicializar variables con un constructor parametrizado
    Wall(double lon, double alt)
      : longitud{lon}
```

```
      , altura{alt} {
    }

    double calculaArea() {
      return longitud * altura;
    }
};

int main() {
  // Crea un objeto de la clase Wall
  Wall wall1(10.5, 8.6);

  // Copiar el contenido de wall1 a wall2 mediante el constructor de copia predeterminado
  Wall wall2 = wall1;

  // impresión de áreas de wall1 y wall2
  cout << "Área de Wall 1: " << wall1.calculaArea() << endl;
  cout << "Área de Wall 2: " << wall2.calculaArea();

  return 0;
}
```

Salida

```
Área de Wall 1: 90.3
Área de Wall 2: 90.3
```

En este programa no hemos definido un constructor de copia. El compilador utilizó el constructor de copia predeterminado para copiar el contenido de un objeto de la clase `Wall` a otro.

Sobrecarga de constructores en C++

Los constructores se pueden sobrecargar de forma similar a la **sobrecarga de funciones**.

Los constructores sobrecargados tienen el mismo nombre (nombre de la clase), pero una cantidad diferente de argumentos. Según la cantidad y el tipo de argumentos que se pasen, se llama al constructor correspondiente.

Ejemplo 1: Sobrecarga del constructor

```
// Programa en C++ para demostrar la sobrecarga del constructor
#include <iostream>
using namespace std;

class Person {
   private:
    int edad;

   public:
    // 1. Constructor sin argumentos
    Person() {
        edad = 20;
    }

    // 2. Constructor con un argumento
    Person(int a) {
        edad = a;
    }

    int getEdad() {
        return edad;
    }
};

int main() {
    Persona persona1, persona2(45);

    cout << "Persona1 Edad = " << persona1.getEdad() << endl;
    cout << "Persona2 Edad = " << persona2.getEdad() << endl;
```

```
    return 0;
}
```

Salida

```
Persona1 Edad = 20
Persona2 Edad = 45
```

En este programa, hemos creado una clase `Persona` que tiene una única variable `edad`.

También hemos definido dos constructores `Persona()` y `Persona(int a)`:

Cuando se crea el objeto `persona1`, se llama al primer constructor porque no hemos pasado ningún argumento. Este constructor inicializa edad en 20. Cuando se crea `persona2`, se llama al segundo constructor porque hemos pasado 45 como argumento. Este constructor inicializa `edad` en 45.

La función `getEdad()` devuelve el valor de `edad` y lo usamos para imprimir la edad de `persona1` y `persona2`.

Ejemplo 2: Sobrecarga de constructores

```
// Programa en C++ para demostrar la sobrecarga del constructor
#include <iostream>
using namespace std;

class Habitacion {
   private:
    double longitud;
    double ancho;

   public:
    // 1. Constructor sin argumentos
    Room() {
        longitud = 6.9;
        ancho = 4.2;
    }

    // 2. Constructor con dos argumentos
    Habitacion(double l, double b) {
        longitud = l;
        ancho = b;
    }
    // 3. Constructor con un argumento
    Habitacion(double lon) {
        longitud = lon;
        ancho = 7.2;
    }

    double calculaArea() {
        return longitud * ancho;
    }
};

int main() {
    Habitacion habitacion1, habitacion2(8.2, 6.6), habitacion3(8.2);

    cout << "Cuando no se pasa ningún argumento: " << endl;
    cout << "Área de la habitación = " << room1.calculaArea() << endl;

    cout << "\nCuando se pasa (8.2, 6.6)." << endl;
    cout << "Área de la habitación = " << room2.calculaArea() << endl;

    cout << "\nCuando el ancho se fija en 7,2 y se pasa (8,2):" << endl;
    cout << "Área de la habitación = " << room3.calculaArea() << endl;

    return 0;
}
```

Salida

```
Cuando no se pasa ningún argumento:
```

```
Área de la habitación = 28,98

Cuando se pasa (8,2, 6,6).
Área de la habitación = 54,12

Cuando el ancho se fija en 7,2 y se pasa (8,2):
Área de la habitación = 59,04
```

Cuando se crea habitacion1, se llama al primer constructor. longitud se inicializa a 6.9 y ancho se inicializa a 4.2.

Cuando se crea habitacion2, se llama al segundo constructor. Hemos pasado los argumentos 8.2 y 6.6. longitud se inicializa al primer argumento 8.2 y ancho se inicializa a 6.6.

Cuando se crea habitacion3, se llama al tercer constructor. Hemos pasado un argumento 8.2. longitud se inicializa al argumento 8.2. ancho se inicializa a 7.2 de forma predeterminada.

Destructores de C++

Un destructor es una *función miembro* especial que se llama automáticamente cuando un objeto queda fuera del ámbito o cuando eliminamos el objeto con la expresión `delete`.

En C++, un destructor tiene el mismo nombre que el de la clase y no tiene un tipo de retorno. **~** precede al identificador para indicar destructor. Por ejemplo,

```
class  Wall {
  public:
    // create a destructor
    ~Wall() {
      // code
    }
};
```

Aquí, `~Wall()` es un destructor de la clase `Wall`.

Nota: Si no definimos ningún destructor, asignación de movimiento o constructor de movimiento en nuestra clase, entonces el compilador de C++ creará automáticamente un destructor predeterminado con un cuerpo vacío. Esto es suficiente en la mayoría de los casos.

Sin embargo, si nuestra clase implica el manejo de recursos como la asignación dinámica de memoria, tenemos que definir un destructor y desasignar los recursos en el cuerpo del destructor.

Asignación dinámica de memoria en la clase

Cuando nuestra clase tiene miembros punteros, el constructor de copia predeterminado simplemente asigna el valor de los punteros de miembro de un objeto a los punteros de miembro de otro objeto, en lugar de asignar diferentes direcciones de memoria y copiar el valor apuntado por los punteros de miembro.

Para asignar una nueva dirección de memoria para la variable y copiar los datos, tenemos que declarar un constructor de copia. Además, tenemos que desasignar la memoria utilizando el destructor.

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

// declarar una clase
class Wall {
  private:
    double* longitud;
    double* altura;

  public:

    // inicializar variables con un constructor parametrizado
    Wall(double lon = 1.0, double alt = 1.0)
      : longitud{new double{lon}}
      , altura{new double{alt}} {
    }
```

```
    // constructor de copias con un objeto Wall como parámetro
    // copia datos del parámetro obj
    Wall(const Wall& obj)
      : longitud{new double{*(obj.longitud)}}
      , altura{new double{*(obj.altura)}} {
    }

    void setlongitud(double lon) {
        *longitud = lon;
    }

    double calculaArea() {
      return *longitud * *altura;
    }

    // destructor para desasignar memoria
    ~Wall() {
        delete longitud;
        delete altura;
    }
};

int main() {
  // Crea un objeto de la clase Wall
  Wall wall1(10.5, 8.6);

  // Copiar el contenido de wall1 a wall2 mediante el constructor de copias
  Wall wall2 = wall1;

  // cambiar la longitud de wall2
  wall2.setlongitud(11.5);

  // print areas de wall1 y wall2
  cout << "Área de Wall 1: " << wall1.calculaArea() << endl;
  cout << "Área de Wall 2: " << wall2.calculaArea();

  return 0;
}
```

```
Área de Wall 1: 90.3
Área de Wall 2: 98.9
```

Aquí,

```
Wall(const Wall& obj)
  :longitud{new double{*(obj.longitud)}}
  , altura{new double{*(obj.altura)}} {
}
```

es el constructor de copia. Toma un objeto `obj` de `Wall` como referencia constante.

```
: longitud{new double{*(obj.longitud)}}
, altura{new double{*(obj.altura)}}
```

es la lista inicializadora que copia los datos a nuevas ubicaciones de memoria e inicializa los punteros de longitud y altura en consecuencia.

- `*(obj.longitud)` es el valor apuntado por el miembro puntero de longitud del objeto de argumento `obj`
- `new double*(obj.longitud)` asigna dinámicamente memoria para el tipo de datos `double` con el valor `*(obj.longitud)` y devuelve la dirección de memoria
- `longitudnew double *obj.longitud)` inicializa la variable `longitud` del nuevo objeto con la nueva dirección de memoria.

De manera similar,

```
altura{new double{*(obj.altura)}}
```

inicializa el miembro puntero `altura` del nuevo objeto.

Cuando los objetos `wall1` y `wall2` quedan fuera del ámbito, se invoca su destructor respectivo. El destructor desasigna la memoria dinámica adquirida por los objetos.