C++ Orientado a Objetos

Miguel Ángel Martín López

Grupo Universitario de Informática

Hora del Código
6 de Noviembre 2024

Contenidos

Introducción a C++

- Lenguaje de Programación Orientado a Objetos (y mucho más)
- Uso en muchísimos ámbitos
  - Modelado 3D y Videojuegos (Unreal Engine 5)
  - Navegadores Web (Google Chrome)
  - Bases de Datos (MySQL o MongoDB)
  - Sistemas Operativos (Windows o MacOS)
  - Editores de código (Visual Studio Code)
- Lenguaje Totalmente Compilado
- Control **casi** total de los recursos del sistema
- Compatibilidad con bibliotecas externas (y también con el lenguaje C!)

Hello World

```cpp
#include <iostream>

int main(int argc, char **argv)
{
    // Comentarios de una linea
    std::cout << "Hola Mundo!" << std::endl;
    return 0;
    /*
        Comentario
        multilinea
    */
}
```

¿Cómo compilamos?

- Usamos el compilador `g++`.
- `g++ fichero.cpp -o fichero`
- La opción `-o` sirve para dar nombre al archivo de salida
- El archivo que nos devuelve el compilador es un ejecutable, que podemos ejecutar directamente escribiendo `./fichero`
- Si usamos la opción `-g` nos permitirá depurar el archivo posteriormente (con `gdb`) si ocurre algún error que detenga el programa.

¿Qué es la Programación Orientada a Objetos?

Es una forma de programar que nos permite no reutilizar código, de manera que sea más sencillo su mantenimiento.
Podemos agrupar el código en módulos e independizar unos módulos de otros.
La forma más común de agrupar módulos es mediante **Clases**.
Es muy importante conocer la diferencia entre los distintos modificadores de acceso.

Modificadores de acceso

Tenemos 3 modificadores:

- `public`: Se puede acceder desde cualquier parte.
- `protected`: Se puede acceder desde una clase derivada (más adelante vemos que es).
- `private`: Solo se puede acceder desde una misma clase.

Hay una excepción, que es que si declaramos una clase con la palabra clave `friend`, referenciando a otra clase, esta otra clase sí podrá acceder a los elementos que sean privados. No suele utilizarse mucho.
Si no especificamos un modificador de acceso, el compilador lo interpretará como privado.

Declaración de clase

En el archivo vehiculo.h

```
1  #ifndef VEHICULO_H
2  #define VEHICULO_H
3
4  #include <string>
5
6  class Vehiculo {
7    private:
8      int velocidad;
9      std::string marca;
10
11   public:
12     Vehiculo(std::string marca);
13     void acelerar();
14     void acelerar(int velocidadAñadida);
15     void frenar();
16     void frenar(int velocidadReducida);
17     int getVelocidad() const;//const indica que no modifica
18     std::string getMarca() const;//los atributos del objeto.
19 }; // Este punto y coma es obligatorio!!
20
21 #endif
```

Implementación de la clase

En el archivo vehiculo.cpp

```
1  #include "vehiculo.h"
2  #include <iostream>
3
4  Vehiculo::Vehiculo(std::string m) : marca(m), velocidad(0) {
5    // Otras funciones que haya que hacer
6  };
7
8  void Vehiculo::acelerar(){
9    velocidad += 10;
10   std::cout << "Acelerando: " << velocidad << " km/h" << std::
         endl;
11 }
12
13 void Vehiculo::frenar(){
14   if(velocidad <= 0) return;
15   velocidad -= 10;
16   std::cout << "Frenando: " << velocidad << " km/h" << std::endl
         ;
17 }
18
19 int Vehiculo::getVelocidad(){
20   return velocidad;
21 }
```

Creación del objeto

En el archivo main_vehiculo.cpp

```cpp
#include "vehiculo.h"
#include <iostream>

int main() {
    Vehiculo miCoche("Toyota");

    std::cout << "Marca del vehículo: " << miCoche.
        getMarca() << std::endl;

    miCoche.acelerar();
    miCoche.acelerar();
    miCoche.frenar();

    std::cout << "Velocidad final: " << miCoche.
        getVelocidad() << " km/h" << std::endl;

    return 0;
}
```

¿Cómo ampliamos la modularidad en C++?

En C++ tenemos una palabra reservada llamada `namespace`.
Esta palabra reservada permite clasificar por 'colecciones'.
Podemos agrupar tanto funciones independientes como clases.
El `namespace` más importante es **std**. Se trata de la libreria estándar.

```
1  #include <...>
2
3  namespace Espacio {
4
5    class UnaClase {
6      private:
7        UnaClase() : {}
8      public:
9        void unMetodo();
10     protected:
11       ...
12   };
13
14   void metodoFuera();
15 }
16 // Ahora, para referenciar a esta clase:
17 int main(){
18   Espacio::UnaClase clase; // Aqui el objeto ya se crea
19   clase.unMetodo();
20   Espacio::metodoFuera();
21 }
```

¿Podemos evitar escribir 'Espacio::'?

- Sí, pero no es conveniente. Puede haber conflicto con funciones o clases con el mismo nombre.
  Además, puede llegar a dificultar la lectura del código si tenemos varios namespaces distintos.
- Para no tener que escribirlo escribimos lo siguiente:
  `using namespace Espacio;`

Control de errores

- Tenemos que asegurar que controlamos los errores que afecta a la lógica del objeto.
- Preguntas ⇒
  - ¿Todos los valores de entrada de un método son válidos?
  - ¿Puede llegar un objeto a un estado inválido?
  - ¿Qué es realmente el estado de un objeto?
- En el ejemplo anterior, ¿tiene sentido que un coche tenga la marca = '11111111$\cdots$'?
- ¿Tiene sentido una velocidad negativa en el caso del Vehículo?

Control de errores

- Para que el código sea más seguro ⇒ añadimos excepciones.
- Imaginad que en la declaración añadimos funciones que reciben argumentos.

```
1  void acelerar(int velocidadAnadida){
2      // Hay que asegurarse que la velocidad no sea negativa
3      if(velocidadAnadida < 0){
4          throw std::invalid_argument("El aumento de velocidad no
               puede ser negativo!");
5      }
6      velocidad += velocidadAñadida;
7  }
```

Para capturar una excepción, con try - catch

```
1  try{
2      // código que puede lanzar excepción
3  } catch(std::exception) {
4      // Recuperacion de la excepcion
5  } catch(...) {
6      // Cualquier otra excepcion que no controlemos
7  }
```

Herencia

- La herencia es un mecanismo de la POO que nos permite generalizar o especificar las clases.
- Tenemos clases padres y clases hijas. (También super-clase y sub-clase).
- En Java $\rightarrow$ Herencia Simple
- En C++ $\rightarrow$ Herencia Múltiple
- Las clases hijas heredan los atributos y métodos de las clases padres.
- Si una clase padre tiene un método con la keyword `virtual`, en la clase hija podremos reescribir ese método.

Herencia Simple

```cpp
class Vehiculo {
private:
    int velocidad;
public:
    Vehiculo() : velocidad(0) {}
    void acelerar() { velocidad += 10; }
};

class Automovil : public Vehiculo {
/*
  Esta clase heredará los atributos de Vehiculo y sus métodos
*/
public:
    void pitar() {
        std::cout << "Bip Bip!" << std::endl;
    }
};
```

Herencia Múltiple

```cpp
class Vehiculo {
private:
    int velocidad;
public:
    Vehiculo() : velocidad(0) {}
    void acelerar() { velocidad += 10; }
};
class Persona {
private:
    float altura;
    std::string nombre;
public:
    Persona(float a, std::string n) : altura(a), nombre(n) {}
    void saludar() { std::cout << "Hola!" << std::endl; }
};
class Transformer : public Vehiculo, public Persona {
/*
  Esta clase heredará los atributos y métodos de Vehiculo
  y Persona.
*/
public:
    void transformarse() { ... }
};
```

Métodos y clases virtuales

- Para que una clase sea virtual, debe tener al menos un método virtual sin implementar. Ejemplo:

```
1  class Vehiculo {
2  public:
3      virtual void describir() = 0;  // Método puramente virtual
4  };
5
6  class Automovil : public Vehiculo {
7  public:
8      void describir() override { std::cout << "Soy un automóvil."
           << std::endl; }
9  };
```

Polimorfismo

- Es la capacidad de un objeto de poder reconocer de forma dinámica qué método debe ejecutar.
- Es muy importante saber la diferencia entre el tipo que nosotros declaramos y el que realmente tenemos.
- Ejemplo:

```
1  // En este vector almacenamos todo tipo de vehiculos
2  std::vector<Vehiculo> vehiculos;
3  ... // Insertamos muchos vehiculos...
4
5  std::cout << vehiculos[20].pitar() << std::endl;
6  // Qué vamos a imprimir realmente??
```

¡GRACIAS POR VENIR!