Object Oriented Programming

Raul P. Pelaez

October 31, 2024

Contents

1 Introduction

One of the paradigms supported by C++ is Object Oriented Programming (OOP). As a matter of fact, C++ was originally designed as an extension of C to support OOP. In this chapter we will introduce the basic concepts of OOP and how they are implemented in C++.
OOP allows developers to structure code around objects, which are instances of classes. This approach enables you to organize your programs into reusable, modular components that combine data and functionality. In C++, OOP is built on four main principles: encapsulation, abstraction, inheritance, and polymorphism. By leveraging these concepts, you can create more maintainable, flexible, and scalable software.
In today's lesson we will cover the basic syntax of classes and objects in C++, focusing on the encapsulation principle.

Info

Encapsulation
Encapsulation is the process of bundling data and methods that operate on that data into a single unit, called a class. The data is hidden from the outside world and can only be accessed through the class's public interface. This allows you to protect the data from being modified in unexpected ways and ensures that the class's internal state remains consistent.
Encapsulation is a powerful way of separating implementation and interface.

Info

Nomenclature

- An object is an instance of a type.
- A class is a type.
- A structure is a class where all members have public visibility by default.
- A method is a function that belongs to a class.
- A member refers to a data member or a method of a class.

2 New basic syntax

Before getting into OOP, let me show you a bit of new core C+ syntax.

2.1 Returning multiple values from a function

In C++ you can only return one value from a function. The classic trick in C to overcome this is to use a structure:

Source code

```
// The old C way
struct Pair{
  int a;
  int b;
};

Pair foo(){
  return {1, 2};
}

void bar(){
  Pair p = foo();
}
```

In C++, you can return multiple values from a function using the `std::tuple` class. This class is defined in the `<utility>` header. You can think of a `std::tuple` as an "anonymous structure".

Source code

```cpp
#include <tuple>
#include <string>
  std::tuple<int, double, std::string> foo(){
    return std::make_tuple(1, 2.0, "hello");
  }

  void bar(){
    // Latest C++ standards allow to "unpack" the tuple into new variables like
    ↪  this.
    auto [a, b, c] = foo();
    // This also works when the variables are already defined.
    int d;
    double e;
    std::string f;
    std::tie(d,e,f) = foo();
  }
```

The notation `something<Type1, Type2, ...>` is called a template argument list. We will learn more about templates in a later lesson.

2.2 Key new concepts

Info

Returning multiple values
`std::tuple` allows you to return multiple values from a function.
There is a special syntax to "unpack" the tuple into new variables:

Source code

```cpp
auto [a, b] = foo();
//Equivalent
//int a;
//std::string b;
//std::tie(a, b) = foo();
```

3 User-defined types: Class and struct

Any type that is not built-in (like `int`,`double`,`char`,...) is called "user-defined".
You have already seen many instances of user-defined types, like `std::string`, `std::vector`. In this section we will learn how to define our own user-defined types using the `class` and `struct` keywords.
In C, we could define a `struct` like this:

Source code

```
struct Vector2D{
  double x;
  double y;
};

void foo(){
  Vector2D p;
  p.x = 1.0;
  p.y = 2.0;
}
```

In C, a `struct` is a just collection of data members (called a plain-old-data (POD) type).

Info

In C++, we are allowed to customize our types further, a type can have:

- Data members (variables)
- Member functions (methods)
- Control over member visibility (`public`, `private`, `protected`)

3.1 Adding methods to types

Say we need to compute the magnitude of a 2D vector. We could define a function that takes a `Vector2D` as an argument:

Source code

```
// The old C way
struct Vector2D{
  double x;
  double y;
};

double magnitude(Vector2D v){
  return sqrt(v.x*v.x + v.y*v.y);
}

void foo(){
  Vector2D p;
  p.x = 1.0;
  p.y = 2.0;
  double m = magnitude(p);
}
```

In C++, we can define a method that belongs to a type, so we can do this instead:

Source code

```
// The C++, OOP way
struct Vector2D{
  double x;
  double y;

  double magnitude(){
    // The method has access to the data members of the object
    return sqrt(x*x + y*y);
  }
};

void foo(){
  Vector2D p;
  p.x = 1.0;
  p.y = 2.0;
  // We can call the method on the object using the dot operator
  double m = p.magnitude();
}
```

A class "knows" how to represent the data needed in an object and what operations can be applied to it.

3.1.1 Separating declaration and definition

Similarly as with functions, we can separate the declaration of the method from its definition.

Source code

```
struct Vector2D{
  double x;
  double y;
  // Notice the () denoting a method
  double magnitude(); // Declaration
};

void foo(){
  Vector2D p;
  p.x = 1.0;
  p.y = 2.0;
  double m = p.magnitude();
}

// Definition
// We could even define the method in another file
double Vector2D::magnitude(){
  return sqrt(x*x + y*y);
}
```

3.2 The constructor method

A constructor is a special method that is called when an object is created. It is used to initialize the object's data members. The constructor has the same name as the class and no return type.
Example

Source code

```
struct Vector2D{
  double x;
  double y;

  // Constructor, check for valid input and initialize
  Vector2D(double in_x, double in_y){
    x = in_x;
    y = in_y;
  }

  double magnitude(){
    return sqrt(x*x + y*y);
  }
};

void foo(){
  // We can now create a Vector2D object like this
  Vector2D p(1.0, 2.0); // The constructor is called
  double m = p.magnitude();
}
```

3.2.1 The initializer list

The constructor can also use an initializer list to initialize the data members. This is the preferred way to initialize data members, as it is more efficient and allows you to initialize const members (see below for information about const).
The syntax is as follows:
Example

Source code

```
struct Vector2D{
  double x;
  double y;

  Vector2D(double in_x, double in_y): x(in_x), y(in_y){
  }
};
```

3.3 Referring to the current object: The `this` pointer

All classes hold a special hidden member called `this`. This member is a pointer to the object itself. Besides any other use that might require such a pointer, we can use `this` to make explicit that we are referring to a member of the object. Sometimes we might want to do this for clarity, but other times it is necessary, like in the following example:
Example

Source code

```
struct Vector2D{
  double x;
  double y;

  Vector2D(double x, double y){
    // We can use the this pointer to refer to the object's data members
    ↪  instead of the arguments of the constructor
    //  that happen to have the same name
    this->x = x;
    this->y = y;
  }
};
```

3.4 Controlling member visibility

The power of encapsulation comes from the ability to separate the interface of a class from its implementation. We can hide the implementation details of a class by making some members private. This way, we can change the implementation of a class without affecting the code that uses it.
We can use the `public`, `private`, and `protected` keywords to control the visibility of members of a class.

Advice

The only distinction between a `class` and a `struct` is that members of a `class` are private by default, while members of a `struct` are public by default.
That means that the following two definitions are equivalent:

Source code

```
struct Vector2D{
  double x;
  double y;
};

class Vector2D{
public:
  double x;
  double y;
};
```

Advice

Interface and implementation
A class should typically look like this:

Source code

```
class X{ // this class is called X
public:
  // the interface to users (accessible by all)
  //functions, types, and data members (data is often best kept private)
private:
  // the implementation details (used by members of this class only)
  //functions, types, and data members
};
```

Encapsulation allows us to make sure that the internal data of a class is not modified in unexpected ways. For example, we can make the data members of a class private and provide public methods to access and modify them in a controlled way.
Example

Source code

```
// A vector that is always normalized
class UnitVector2D{
  // The data members are private by default in a class
  double x;
  double y;
public:
  double magnitude() const{
    return sqrt(x*x + y*y);
  }
  // Constructor
  UnitVector2D(double in_x, double in_y){
    double m = sqrt(in_x*in_x + in_y*in_y);
    x = in_x/m;
    y = in_y/m;
  }
  // Getter
  auto get(){
    return std::make_pair(x,y);
  }
  // Setter
  void set(double in_x, double in_y){
    double m = sqrt(in_x*in_x + in_y*in_y);
    x = in_x/m;
    y = in_y/m;
  }
};

void foo(){
  UnitVector2D p(1.0, 2.0);
  // double x = p.x; // Error: cannot access private members
  // We can access the data members using the getter
  auto [x,y] = p.get();
  // We can modify the data members using the setter
  p.set(2.0, 1.0);
}
```

3.5 Key new concepts

Info

`class` and `struct`
Classes and structures are used to define user-defined types. Types can have data members and member functions (methods).
The only difference is that members of a `class` are private by default, while members of a `struct` are public by default.

Info

Constructors
A constructor is a special method that is called when an object is created. It is used to initialize the object's data members.
The constructor has the same name as the class and no return type.

Info

Visibility
The `public` and `private` keywords are used to control the visibility of members of a class, i.e. whether the members are accessible from outside the class.

Info

`this` pointer
The `this` pointer is a pointer to the object itself. It is used to refer to the object's data members.

4 Nested classes

A class can be defined inside another class. This is called a nested class. A nested class can access the private members of the enclosing class.

Example

Source code

```
class Outer{
  int x;
public:
  class Inner{
    int y;
  public:
    Inner(int y): y(y){}
    int get_x(Outer& o){
      return o.x;
    }
  };
};

void foo(){
  Outer o;
  Outer::Inner i(1);
  int x = i.get_x(o);
}
```

Advice

Nested classes having access to private members should not be used to just break encapsulation like we are doing here, but to provide a meaningful relationship between the classes.

4.1 Key new concepts

Info

Nested classes
Its possible to define a class inside another class. A nested class can access the private members of the enclosing class.

5 `const` in classes

The `const` keyword has two main uses:

- To declare that a function does not modify the object (const member functions)
- To declare that a data member is constant (const data members)

Example

Source code

```
class Person{
  // Const here means that the name cannot be changed
  //  after the object is created
  const std::string name;
  int age;
public:
  Person(std::string name, int age): name(name), age(age){
    // We can't change the name here either
    // It can only be set in the initializer list
  }
  // Const here makes the compiler and the reader know
  //  that this function does not modify the state of the object
  std::string get_name() const{
    return this->name;
  }

  //Adding const here would produce an error
  void set_age(int new_age){
    this->age = new_age;
  }
};
```

5.1 Key new concepts

Info

`const` member functions

- **Methods**: A const member function is a member function that does not modify the object. It is declared by adding the `const` keyword after the function declaration.
- **Data**: A const data member can only be initialized in the initializer list of the constructor.

6 `static` members

A `static` member is a member of a class that is shared by all instances of the class. It is not associated with any particular object, but with the class itself.
Example

Source code

```
class Counter{
  static int count; // This is a static member
public:
  Counter(){
    count++;
  }
  // Methods that make use of only static members can be static themselves
  static int get_count(){
    return count;
  }
};

int Counter::count = 0; // We need to define the static member outside the class

void foo(){
  Counter c1;
  Counter c2;
  int n = Counter::get_count(); // We can access the static member using the
  ↪  class name
  // We can also access it using any object
  int also_n = c1.get_count();
}
```

6.1 Key new concepts

Info

`static` members
A static member (either a method or data) is a member of a class that is shared by all instances of the class. It is not associated with any particular object, but with the class itself.
It is similar to defining the member inside a namespace.

7 The `friend` keyword

Sometimes we want a function or another class to have access to the private members of a class. We can achieve this by declaring the function or class as a friend of the class. Friends are entities that for some reason or another need to break encapsulation and access the implementation of another class.
Example

Source code

```
class Secret{
  int x;
public:
  Secret(int x): x(x){}
  // This function is a friend of Secret
  friend void print_x(Secret s);
};

void print_x(Secret s){
  std::cout << s.x << std::endl; // OK: x is private but print_x is a friend
}
void foo(Secret s){
  std::cout << s.x << std::endl; // Error: x is private
}
```

Advice

A typical use of the `friend` keyword is to control where a particular object can be created. For example, we might want to make the constructor of a class private to make sure that objects of that class can only be created by a factory function.
Example

Source code

```
class HardToConstruct{
  int x;
  HardToConstruct(int x): x(x){
    // Imagine getting x is really complex
  }
public:
  // The factory function is a friend
  friend HardToConstruct create();
  int get_x(){
    return x;
  }
};

HardToConstruct create(){
  int x = something_really_complex();
  return HardToConstruct(x);
}
```

7.1 Key new concepts

Info

`friend` functions and classes
It is possible to allow an external function or class to access the private members of a class by declaring it as a friend.

8 Exercises

Each milestone is intended to reflect at least one of the "Key new concepts" laid out in the notes for today. Please read the notes if the milestone's description or intent is not immediately clear to you. Consider the advanced milestones as optional as far as submission is concerned, but please always read them anyway even if you decide you do not have time to go through them.

8.1 The Student class

Goal

Let us write an Student class to put the basics of OOP into practice.
Milestone by milestone we will add more features to the class.

Learning goal

Familiarize with the basic C++ OOP syntax.

Milestone

Returning multiple values
Write a function 'generateStudentInfo' that returns a tuple with the following values:

- A random name (from a list of names)
- A random age (between 18 and 25)
- A random grade (between 0.0 and 10.0)

Learning goal

Familiarize with the C++ idiom to return multiple values from functions.
Get to know the `std::tuple` class.

Milestone

Basic Class Creation
Create a class called ‘Student‘ with the following specifications:

- Private data members:
  - ‘name‘ (std::string)
  - ‘age‘ (int)
  - ‘grade‘ (double)
- Public member functions:
  - ‘getName()‘, ‘getAge()‘, and ‘getGrade()‘ accessor methods (getters)
  - ‘setAge(int newAge)‘ and ‘setGrade(double newGrade)‘ methods (setters)
  - Add a ‘setName(std::string newName)‘ method that ensures the name is not empty

Write a ‘main()‘ function to create a Student object and test its methods.

Learning goal

- Familiarize with the concept of visibility.
- Familiarize with class method definitions.

Milestone

Initialization and Data Validation
Modify the ‘Student‘ class to include data validation by adding a Constructor:

- The constructor should take name, age, and grade as parameters
- Ensure that age is between 0 and 120
- Ensure that grade is between 0.0 and 10.0
- In the setters and the constructor, print an error message if an invalid value is provided

Learning goal

Familiarize with the concept of constructors.

Milestone

Separate definition and declaration
Add a new method to the ‘Student‘ class, ‘printInfo()‘, that prints a summary of the student. Declare this method inside the body of the class and define it outside the class definition.

Learning goal

Similar to functions, we can separate the declaration and definition of methods.
Understand that methods can be declared inside the class definition and defined outside.

Milestone

Static Members
Add static members to the ‘Student‘ class:

- Add a static member variable ‘totalStudents‘ that keeps track of how many Student objects have been created.
- Add a static method ‘getTotalStudents()‘ that returns the current count.

Modify the constructor to increment the count.

Hint

Static member variables must be declared inside the class definition, but defined outside.

Learning goal

Understand the concept of static members. Variables shared by all instances of a class.

Milestone

Const Correctness
Modify the ‘Student‘ class for const correctness:

- Mark all the methods that do not modify the object as `const`

Learning goal

Understand the usage of `const` in method definitions as both a promise to the compiler and to the reader.

Advanced milestone

Mutable members
Sometimes we need members to be mutable even if the object, or a method inside it, is const. For instance, we might want to register whether someone has accessed the age of a student.

Add a mutable bool variable to the ‘Student‘ class called ‘ageAccessed‘ that is set to false by default.
Modify getAge to set this variable to true when called.
Modify printInfo to print whether the age has been accessed or not.

Make sure the method getAge is still marked as const.
Make sure you can still call getAge from a const object like this:

Source code

```
const Student s("Alice", 20, 8.0);
std::cout << s.getAge() << std::endl;
s.printInfo();
```

Hint

You can use the mutable decorator to declare a member variable that can be modified even in a const method or object.

Advanced milestone

Friend Function access
Add a friend function to the ‘Student‘ class:

- Declare a friend function ‘compareGrade(const Student& a, const Student& b)‘ that returns true if student a’s Grade is higher than student b’s.
- Implement this function outside the class definition

Hint

You must declare the function as a friend of the class inside the class definition.

Advanced milestone

Nested classes
Lets Add the concept of a Course to the Student class:
Create a nested class within ‘Student‘ called ‘Course‘:

- ‘Course‘ should have public members ‘courseName‘ (std::string) and ‘year‘ (int)
- Add a vector of ‘Course‘ objects as a private member to the ‘Student‘ class, starts empty.
- Add a method ‘enroll‘ that takes a ‘Course‘ object and adds it to the vector.
- Add a method ‘printCourses‘ that prints the name and year of each course currently enrolled.