Object Oriented Programming

Raul P. Pelaez

November 26, 2025

Contents

1 Introduction **1**

2 New basic syntax **2**
2.1 Returning multiple values from a function . . . 2
2.2 Key new concepts . . . 3

3 User-defined types: Class and struct **3**
3.1 Adding methods to types . . . 4
3.1.1 Separating declaration and definition . . . 4
3.2 The constructor method . . . 5
3.2.1 The initializer list . . . 5
3.3 Destructors . . . 6
3.3.1 Key new concepts . . . 6
3.4 Referring to the current object: The `this` pointer . . . 6
3.5 Controlling member visibility . . . 7
3.6 Key new concepts . . . 8

4 Inheritance **9**
4.1 Basic syntax . . . 9

5 Polymorphism **10**
5.1 Runtime polymorphism . . . 10

6 Exercises **11**
6.1 The Student class . . . 11

1 Introduction

Today, we will dip our toes into C++ a bit to learn about Object Oriented Programming (OOP). As we have seen throughout the course, C and C++, while both having a shared past, are different languages with different philosophies. C++ is a multi-paradigm programming language that supports procedural, object-oriented, and generic programming styles, while C is primarily a procedural programming language. Although many times similar concepts and syntax are shared between the two languages, C++ and C are written with vastly different styles and approaches in mind.

One of the paradigms supported by C++ is 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

Inheritance
Inheritance is a mechanism that allows a class to inherit properties and behavior from another class. The class that inherits is called a subclass or derived class, and the class that is inherited from is called a superclass or base class.

Info

Polymorphism
Polymorphism is a feature that allows objects of different classes to be treated as objects of a common superclass.

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:

```
// 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".

```
#include <tuple>
#include <string>
```

```cpp
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, a powerful C++ feature that allows you to define generic types and functions. We will not go into details about templates today, but keep in mind that many C++ standard library classes and functions are defined using templates.

Info

The namespace access operator: `::`
The `::` operator is used to access members of a namespace or a class. In this case, we are using it to access the `tuple` class and the `make_tuple` function from the `std` namespace. In C, we do not have namespaces, so we just use the names directly. The classic C way of doing this is to use a prefix in the names, like `std_` for standard library functions and types.

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:

```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".

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:

```cpp
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:

```
// 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:

```
// 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.

```
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

```
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. The syntax is as follows:

Example

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

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

3.3 Destructors

A destructor is a special method called when an object is destroyed (like when it goes out of scope). The main purposes of destructors are:

- To release resources acquired by the object during its lifetime.
- To perform any necessary cleanup operations.
- To ensure proper handling of object destruction.

Destructors are declared with a tilde (`~`) before the class name. They have no return type and take no arguments.

Example

```
class FileHandler {
  FILE* file;// Note: in C++ we normally use std::fstream instead.
public:
  FileHandler(const char* filename) {
    // Acquaire the resource
    file = fopen(filename, "r");
  }
  ~FileHandler() {
    // Release the resource
    if (file) {
      fclose(file);
    }
  }
};
void read_file(const char* filename) {
  FileHandler handler(filename); // The constructor is called here
} // handler goes out of scope, destructor is called
```

Advice

RAII: Resource Acquisition Is Initialization
Perhaps the most powerful idiom in C++ is RAII. The idea is that resources should be acquired in constructors and released in destructors. This way, the lifetime of the resource is tied to the lifetime of the object, ensuring proper cleanup and avoiding resource leaks. This is the reason why we can just create an `std::vector` without worrying about memory leaks, its destructor makes sure to release the memory.

3.3.1 Key new concepts

Info

Destructors
Destructors are called when an object is destroyed, like when it goes out of scope. They clean up resources and perform finalization. Destructors are declared with a tilde (`~`) before the class name.

Info

RAII
Whenever a class manages a resource (a file, a pointer), it should make sure to release it in the destructor.

3.4 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

```
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;
  }
};
```

3.5 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:

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

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

Advice

Interface and implementation
A class should typically look like this:

```
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

```
// 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);
}
```

3.6 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 Inheritance

Inheritance allows a class to inherit properties and behavior from another class. This is useful, for instance, in cases where:

- We want to define an interface (called abstract class) that other classes must adhere to. For instance, we could define a class `Shape` with a method `draw()` that all subclasses (like `Circle` or `Rectangle`) must implement.
- We want to specialize a class. For instance, we could define a class `Clock` that represents the time of the day in UTC+0. We could then define a subclass `LocalClock` that behaves in the same way but has an additional method to set the time zone.

4.1 Basic syntax

In C++, inheritance is declared by specifying the base class in the class definition. For instance, to define a class `Square` that inherits from `Rectangle` we would write:

```
#include <iostream>

class Rectangle{
  double width, height;
public:
  Rectangle(double w, double h) : width(w), height(h) {}
  double getArea(){
    return width * height;
  }
};

class Square : public Rectangle{
public:
  Square(double s) : Rectangle(s, s) {}
};

int main(){
  Square s(5);
  std::cout << s.getArea() << std::endl;
}
```

Output

```
25
```

We also had to specify a visibility, in this case `public`. The visibility here determines which members and methods the derived class will expose. For instance, private inheritance would make all public and protected members of the base class private in the derived class.

Notice how we can call the method `getArea()` from a `Square` object, even though it is defined in the `Rectangle` class. Inheritance allows us to "inherit" methods and properties from the base class.

Advanced

The constructor of the base class is called automatically when the derived class is instantiated, even if it is not explicitly called in the derived class constructor. It will be initialized before the derived class constructor is called. For destruction, the opposite is true: the derived class destructor is called first, and then the base class destructor.

Advanced

Multiple inheritance
We can inherit from more than one class at once:

```
class Derived: public Serializable, public Drawable{
  //...
};
```

5 Polymorphism

Inheritance on its own only enables us to encode conceptual hierarchies. Polymorphism works on top of it by giving us tools to treat objects of different classes as objects of a common superclass. For instance, we can make a function that takes a `Shape` object and call the `getArea()` method on it, even if the object is actually a `Circle` object.

5.1 Runtime polymorphism

Runtime polymorphism is achieved by using pointers or references to the base class. When a method is called on a pointer or reference to a base class, the method that is called is the one that is defined in the most derived class.

In other words, pointers to a base class can be used to refer to objects of derived classes.

Example

```
#include <iostream>

class Base {
public:
    virtual void print() const {
        std::cout << "Base" << std::endl;
    }
};

class Derived : public Base {
public:
    void print() const override {
        std::cout << "Derived" << std::endl;
    }
};

void print_with_ptr(const Base* b) {
    b->print();
}

int main() {
    Base base;
    Derived derived;

    print_with_ptr(&base); // Prints "Base"

    print_with_ptr(&derived); // Prints "Derived"

    // Pointers to Derived are automatically casted
    // to pointers to Base
    Base* b = &derived;
    b->print(); // Prints "Derived"
    return 0;
}
```

Output

```
Base
Derived
Derived
```

6 Exercises

6.1 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` or `char*`)
  - `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(const char* 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.