Object Oriented Programming III

Raul P. Pelaez

October 31, 2024

Contents

1 Introduction

OOP is based on three principles: Encapsulation, Inheritance and Polymorphism. In the previous lessons we have seen how to use classes and objects to encapsulate data and methods.
In this lesson we will explore Inheritance and Polymorphism.

1.1 Key new concepts

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.

2 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.

2.1 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:

Source code

```
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;
}
```

We also had to specify a visibility, in this case `public`. Lets see what this means in a future section. 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.

Info

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

Source code

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

2.1.1 Overriding methods

If a subclass defines a method with the same name as a method in the base class, the subclass method will "override" the base class method, meaning that the subclass method will be called instead of the base class method.
Example

Source code

```
class Base{
public:
  virtual void print(){
    std::cout << "Base" << std::endl;
  }
};
class Derived : public Base{
public:
  void print() override{
    std::cout << "Derived" << std::endl;
  }
};
int main(){
  Derived d;
  d.print(); // Derived
}
```

The keyword `virtual` tells the compiler (and the reader) that the method can be overridden by a subclass.
The keyword `override` tells the compiler (and the reader) that the method is overriding a method in the base class.

Advice

The keywords `virtual` and `override` are not strictly necessary but highly recommended.
Using them can help catch errors and also manages expectations in the code. For instance, the compiler will complain if the keyword `override` is used but the method in the base class is not `virtual`. This might mean that the programmer is missing something in the signature of the method that they are trying to override, like an argument, or just a typo.

2.1.2 Calling base class methods

Sometimes we need to call a method from the base class from the derived class. We can do this by using the scope resolution operator `::`.

Example

Source code

```
class Base{
public:
  virtula void print(){
    std::cout << "Base" << std::endl;
  }
};
class Derived : public Base{
public:
  void print() override{
    Base::print(); // Calls the Base class version of print
    std::cout << "Derived" << std::endl;
  }
};
int main(){
  Derived d;
  d.print(); // Base\nDerived\n
}
```

2.2 Abstract classes

An abstract class (also called an interface) is one that has at least one pure virtual method. A pure virtual method is one that is marked with the keyword `virtual` and the specifier `=0`. An abstract class cannot be instantiated, but it can be used as a base class for other classes, in which cases the derived class must implement the pure virtual methods.

Example

Source code

```
class Shape{
public:
  virtual double getArea() = 0;
};
class Circle : public Shape{
  double radius;
public:
  Circle(double r) : radius(r) {}
  double getArea() override{
    return 3.14159 * radius * radius;
  }
};
int main(){
  Circle c(5);
  std::cout << c.getArea() << std::endl;
}
```

The compiler will complain if the method `getArea()` is not implemented in the derived class.

Advice

Pure virtual methods make sense when the base class cannot provide a meaningful implementation of the method. For instance, in the case of the `Shape` class, it is not clear how the area of a generic shape would be computed.

2.3 Visibility

In C++, we can specify how the visibility of the members of the base class will transmit to the derived class. The visibility can be `public`, `protected` or `private`.

Info

The `protected` visibility is new for us, it means that the members of the base class are accessible from the derived class, but not from the outside.
Example

Source code

```
class Base{
protected:
  int x;
public:
  int public_thing;
  Base(int x) : x(x) {}
};
class Derived : public Base{
public:
  Derived(int x) : Base(x) {}
  void print(){
    // x is only accessible because it is protected
    std::cout << x << std::endl;
  }
};
int main(){
  Derived d(5);
  //d.x; // Error: x is not public
  d.public_thing; // OK
  d.print();
}
```

When we inherit from a class, we can specify how the visibility of the base class members will manifest in the newly created derived class.

Info

Modes of inheritance
Here is a summary of how the visibility of the inheritance (the specifier when declaring the inheritance) affects the visibility of the base class members in the derived class:

Visibility in base class	Public inheritance	Protected inheritance	Private inheritance
public	public	protected	private
protected	protected	protected	private
private	Unaccessible	Unaccessible	Unaccessible

For instance, if a class is inherited with `public` visibility (the second column), the public members of the base class will be public in the derived class, the protected members will be protected and the private members will not be inherited.
On the other hand, private inheritance (the last column) will make all the members of the base class private in the derived class and thus to the outside world.

2.4 Key new concepts

Info

`virtual` and `override`
The `virtual` keyword is used to declare a method that can be overridden by a subclass.
The `override` keyword is used to declare that a method is overriding a method in the base class.
When a method is invoked on an object, the method that is called is the one that is defined in the most derived class.
When calling a method from the base class in the derived class, we can use the scope resolution operator `::`.

Info

Abstract classes
An abstract class is a class that has at least one pure virtual method.
A pure virtual method is declared with the `virtual` keyword and the specifier `=0`.
An abstract class cannot be instantiated, but it can be used as a base class for other classes.

Info

Visibility
The visibility of the members of the base class in the derived class depends on the visibility of the inheritance. The visibility can be `public`, `protected` or `private`.

3 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.

3.1 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, references and pointers to a base class can be used to refer to objects of derived classes.
Example

Source code

```
#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();
}

void print_with_ref(const Base& b) {
    b.print();
}

int main() {
    Base base;
    Derived derived;

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

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

    // Pointers and references to Derived are automatically casted
    // to pointers and references to Base
    Base* b = &derived;
    b->print(); // Prints "Derived"
    Base& b_ref = derived;
    b_ref.print(); // Prints "Derived"

    return 0;
}
```

3.1.1 Dynamic cast

Converting a pointer to a derived class as one to a base class is in general safe and works as one expects. The opposite indirection is however not safe and dangerous.

When we are presented with a pointer (or reference) to a base class, we do not in principle know if this pointer is actually pointing to an object of the base class or a derived class. For this reason, C++ makes it harder for us to perform this operation, forcing us to use the `dynamic_cast` operator.
The `dynamic_cast` operator can be used to cast a pointer or reference to a base class to a pointer or reference to a derived class. If the cast is not possible, the result will be a null pointer.
Example

Source code

```cpp
#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;
    }
};

int main() {
    Base base;
    Derived derived;

    Base* b = &derived;
    Derived* d = dynamic_cast<Derived*>(b);
    if (d) {
        d->print(); // Prints "Derived"
    } else {
        std::cout << "This is not a pointer to Derived" << std::endl;
    }
    return 0;
}
```

3.2 Key new concepts

Info

Runtime polymorphism
Pointers and references to a base class can be used to refer to objects of derived classes.
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.

Info

Dynamic cast
The `dynamic_cast` operator can be used to cast a pointer or reference to a base class to a pointer or reference to a derived class.
If the cast is not possible, the result will be a null pointer.

4 User defined type conversion

In C++, we can define how a class is converted to another class. This is done by defining a conversion operator. For instance, we could define a conversion operator in the `Circle` class that converts the object to a `double` representing the area of the circle.
Example

Source code

```
#include <iostream>

class Circle {
  double radius;
public:
  Circle(double r) : radius(r) {}
  operator double() const {
    return 3.14159 * radius * radius;
  }
};
int main() {
  Circle c(5);
  double area = c; // Calls the conversion operator
  return 0;
}
```

Advice

Implicit conversion operators are dangerous and can easily result in unexpected behavior. For instance, it is arguably an strange idea that a Circle object when converted to a double should represent its area. One way to avoid this is to mark the conversion operator as `explicit`, which will prevent implicit conversions.
Example

Source code

```
class Circle {
  double radius;
public:
  Circle(double r) : radius(r) {}
  explicit operator double() const {
    return 3.14159 * radius * radius;
  }
};
int main() {
  Circle c(5);
  //double area = c; // Error: no viable conversion from Circle to double
  double area = static_cast<double>(c); // OK
  return 0;
}
```

4.1 Key new concepts

Info

Conversion operators
A conversion operator is a method that allows a class to be converted to another class.
Conversion operators can be used to define implicit conversions, which can be dangerous.
To prevent implicit conversions, the `explicit` keyword can be used.

5 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.

5.1 The Expression Problem

Goal

Today, I want to introduce you to one of the main weaknesses of inheritance, known as the expression problem.
Let me borrow from this great write up about it (please, refrain from reading that until you have worked on the milestones for today):

> Imagine that we have a set of data types and a set of operations that act on these types. Sometimes we need to add more operations and make sure they work properly on all types; sometimes we need to add more types and make sure all operations work properly on them. Sometimes, however, we need to add both - and herein lies the problem. Most of the mainstream programming languages don't provide good tools to add both new types and new operations to an existing system without having to change existing code. This is called the "expression problem". Studying the problem and its possible solutions gives great insight into the fundamental differences between object-oriented and functional programming and well as concepts like interfaces and multiple dispatch.

After finishing your implementation, this code should run:

Source code

```
#include "expression.h"
#include <iostream>
#include <cassert>
int main() {
  // Inheritance in action.
  //Constant and BinaryPlus are both children of Expression
  Constant c1(1);
  Constant c2(2);
  Constant c3(3);
  BinaryPlus bp(c1, c2);
  BinaryPlus bp2(bp, c3);
  std::string c1plusc2plusc3 = bp2.toString();
  double result = bp2.evaluate();
  std::cout << c1plusc2plusc3 << " = " << result << std::endl; // ((1 +
  ↪  2) + 3) = 6
  assert(result == 6);
  // Polymorphism in action
  std::vector<const Expression*> expressions = {&c1, &c2, &c3, &bp, &bp2};
  for (const Expression* e_ptr : expressions) {
    const Expression& e = *e_ptr;
    printExpression(e);
  }
  return 0;
}
```

Learning goal

Familiarize yourself with inheritance and polymorphism.

Milestone

Write the `Expression` abstract class. It should have two pure virtual methods:

- `std::string toString() const` which should return a string representation of the expression.
- `double evaluate() const` which should return the result of evaluating the expression.

Remember that a class is abstract when it has at least one pure virtual method.

hint

- A `virtual` method can be made pure by adding the `= 0` at the end of the declaration.
- Pure virtual methods do not have a body.

Learning goal

Understand how to create an abstract class.

Milestone

Write the `Constant` class. It should inherit from `Expression` and have a constructor that takes a double and stores it.
Implement the `virtual` methods from the `Expression` class.
The evaluate method should return the stored value and the toString method should return the stored value as a string.

hint

- Remember to use the `override` keyword when implementing a virtual method.
- Use the `std::to_string()` function to convert numbers to strings.

Learning goal

Understand the basic syntax of inheritance.

Milestone

Implement the `BinaryPlus` class. It should inherit from `Expression` and have two members of type `const Expression&` that represent the two expressions being added.
At this point, all the code up until the assert in the main function should compile and run.

hint

- The constructor should take two references to expressions and store them.
- The evaluate method should return the sum of the results of evaluating the two expressions.

Milestone

Implement the `printExpression` function, which should print the result of calling the toString method of the expression passed to it and the result of calling the evaluate method.

hint

- You will need to make use of polymorphism, as the function should work with any expression.
- Polymorphism means that a reference or a pointer to a base class can refer or point to a derived class object.

Learning goal

Understand how the basics of polymorphism.

Advanced milestone

Instead of having a method in `Expression` called toString(), implement this functionality using an `explicit` overload of the `std::string()` conversion operator.
This way, you could write:

Source code

```
Constant c1(1);
Constant c2(2);
BinaryPlus bp(c1, c2);
std::string c1plusc2 = static_cast<std::string>(bp);
//Instead of
//std::string c1plusc2 = bp.toString();
```

Advanced milestone

Add a method to the `Constant` class that is called `double setValue(double new_value)` that changes the value stored in the constant.
Write a function that takes a pointer to an expression and a double and changes the value of the constant if the expression is a constant, otherwise does nothing.
Try the function with a `Constant` and a `BinaryPlus` expression.

hint

- You will need to use `dynamic_cast` to check if the expression is a `Constant`.