Advanced OOP Concepts

Introduction to Modern C++

Ryan Baker

March 8, 2025

"*Never trust a computer you can't throw out a window.*" - Steve Wozniak

Lecture Objectives

- To understand the four pillars of object-oriented programming.
- To become acquainted with `virtual` functions.
- To understand interfaces in C++.
- To know how and when to overload operators for class design in C++.
- To understand copy constructors.

Contents

1 Pillars of OOP

1.1 Encapsulation

Encapsulation is the bundling of data (variables) and methods (functions) that operate on the data into a single unit. Direct access to the data is restricted to prevent accidental interference or nefarious misuse.

In C++, encapsulation is demonstrated by marking data members of an object `private`. Consider the following, non-encapsulated implementation of an `Account class`:

```
class Account
{
public:
    Account(): balance(0) {}
    double balance;
    // ...
};
```

Reading the above code should make you shiver. Notice that everything sits in the `public` interface, meaning that all data associated with an `Account` is subject to accidental modification or nefarious intervention:

```
int main()
{
    Account ryans_account();
    // Yours truly is the world's first trillionaire
    ryans_account.balance = 1e12;
}
```

A wiser architect may design the `Account class` like this:

```
class Account
{
private:
    double balance;

public:
    Account(): balance(0) {}
    // ...
};
```

"But now we cannot access the account's balance!" I hear you cry. A fair critique indeed, our class, while encapsulated, is now entirely useless. To fix

this, we can use *getters* and *setters*.

1.1.1 Getters and Setters

Getters and setters are methods defined within a `class` used to get or set data members. They can be implemented with more sophisticated and safe logic than standard member access:

```
class Account
{
private:
    double balance;

public:
    Account(): balance(0) {}

    // getter for account balance
    double get_balance() { return this->balance; }

    // setter for account balance
    void set_balance(double bal)
    {
        if (/* some authentication logic */)
        {
            this->balance = bal;
        }
    }
};
```

Now, access to our account balances can be controlled. As somewhat of a stylistic aside, I cannot help but cringe when I see a `class` method prefixed with `get_`. I prefer to alter the name of my member variable in some manner, perhaps by assigning it a unit, then have my getter take on the member's old name:

```
class Account
{
private:
    double balance_usd;

public:
    // ...
    double balance() { return balance_usd; }
};
```

Often times, developers choose to prefix their data member names with `m_` for "member": `m_balance`. I don't. You can.

I believe that it is in everyone's best interest, except perhaps Lucifer himself, to mark all of your data members as `private`. First of all, it aids in consistency. Users of your `class` need not pontificate about whether or not to access your members with parentheses or not, because only functions sit in the `public` interface.

```
// ... class List defined above

List l(1, 2, 3);

// eliminate this confusion
std::cout << l.length << std::endl;
std::cout << l.length() << std::endl;
```

A more important reason to place data members in the private section is that it allows you to implement custom access. If members sit in the public interface, read/write access is always granted. If you mark them private, you can implement read-only, read-write, or neither read nor write access:

```
class MyClass
{
private:
    int a; // hidden
    int b; // read-only
    int c; // read-write

public:
    int read_b() { return a; }
    int read_c() { return b; }
    void write_c(int c) { this->c = c; }
};
```

Finally, hiding data members allows you to implement *functional abstraction*. This means that you can change the implementation of your data members or the calculations you perform on them while your class user can use the same interface:

```
class Point
{
private:
    double x_in, y_in, z_in; // position in inches

```

```
public:
    double x() { return x_in; }
    double y() { return y_in; }
    double z() { return z_in; }
};
```

Perhaps you'd like to alter your `Point` `class` to store data in meters rather than inches. Rather than needing to modify all the code that interacts with a point, you need only alter the three getter methods:

```
class Point
{
private:
    double x_m, y_m, z_m; // position in meters

public:
    double x() { return x_m * 39.37; } // return in
    double y() { return y_m * 39.37; } // inches like
    double z() { return z_m * 39.37; } // before
};
```

1.1.2 `friend` Functions

`friend` functions declared within a class or struct allow external functions access to private members:

```
class Point
{
private:
    int x, y;

public:
    Point(int x, int y): x(x), y(y) {}

    // declare print_point to be a 'friend'
    friend print_point(const Point& p);
};

void print_point(const Point& p)
{
    // ...
}
```

1.2 Inheritance

Inheritance allows a new class to acquire the properties and methods of an existing class. The new class can also have additional features or `override` existing ones. This promotes code reuse and establishes a hierarchical relationship between classes.

In the following code example, inheritance is demonstrated as both the `Dog` and the `Cat` `class` *inherit* from the `Animal` `class`:

```
#include <iostream>
#include <string>


class Animal
{
public:
    Animal(std::string name): name(name) {}

    virtual void speak() { std::cout << name << " is
        speaking" << std::endl; }
private:
    std::string name;
};


class Cat: public Animal
{
public:
    Cat(std::string name): Animal(name) {}

    void speak() override { std::cout << "meow" <<
        std::endl; }
};


class Dog: public Animal
{
public:
    Dog(std::string name): Animal(name) {}

    void speak() override { std::cout << "woof" <<
        std::endl; }
};


```

```
34 int main()
35 {
36     Animal* cat = new Cat("Jade");
37     Animal* dog = new Dog("Jenny");
38 
39     cat->speak();
40     dog->speak();
41 }
```

1.2.1 The `virtual` Keyword

In C++, the `virtual` keyword is used to declare a method in a base class that can be overridden in derived classes. A virtual function allows for dynamic (or runtime) polymorphism, meaning that the method call is resolved at runtime rather than compile-time.

1.2.2 `protected` Members

`protected` members are visible within their own class and classes that inherit from it:

```
1  #include <iostream>
2  
3  
4  class A
5  {
6  private:
7      int a = 1;
8  protected:
9      int b = 2;
10 public:
11     int c = 3;
12 
13     A() = default;
14 };
15 
16 class B: public A
17 {
18 public:
19     B() = default;
20 
21     void foo()
22     {
23         std::cout << a << std::endl; // not ok
24         std::cout << b << std::endl; // ok
```

```
25          std::cout << c << std::endl; // ok
26      }
27  };
28
29
30  int main()
31  {
32      B* ptr = new B;
33      ptr->foo();
34  }
```

1.3 Polymorphism

Polymorphism allows objects to be treated as instances of their parent `class`, enabling the same operation to behave differently based on the object.

```
1  #include <iostream>
2
3  class Shape
4  {
5  public:
6      virtual void draw() { /* ... */ }
7  };
8
9  class Circle: public Shape
10 {
11 public:
12     // draw circle
13     void draw() { cout << "()" << endl; }
14 };
15
16 class Rectangle: public Shape
17 {
18 public:
19     // draw rectangle
20     void draw() { cout << "[]" << endl; }
21 };
22
23 class Triangle: public Shape
24 {
25 public:
26     // draw triangle
27     void draw() { cout << "|>" << endl; }
28 };
```

1.4 Abstraction

Abstraction focuses on exposing only the essential features of an object while hiding the implementation details. In C++, abstraction is typically achieved using abstract classes (classes with at least one pure virtual function) or interfaces (classes with only virtual functions).

Abstract Classes: An abstract class allows you to define methods that must be implemented by subclasses, while also providing a base for shared functionality. These classes provide a way to define common behavior that can be shared, while allowing subclasses to implement their own specific behavior.

Virtual Functions and Inheritance: In C++, a virtual function is a member function that you expect to override in derived classes. The keyword `virtual` ensures that the function call is resolved at runtime, using dynamic dispatch (polymorphism). This enables the use of the same function name across different classes, while each class can provide its own specific behavior.
When a base class declares a function as virtual, the derived class can override it, and when a pointer or reference to the base class is used to call the function, the appropriate derived class function is executed.

```cpp
#include <iostream>

class Shape {
public:
    // Shape will not implement draw(), but subclasses
       must implement it
    virtual void draw() = 0;
    virtual ~Shape() = default;
};

class Circle : public Shape
{
public:
    void draw() override
    {
        std::cout << "()" << std::endl;
    }
};

class Square : public Shape
{
public:
    void draw() override
    {
        std::cout << "[]" << std::endl;
```

```
    }
};

int main()
{
    Shape* shape1 = new Circle();
    Shape* shape2 = new Square();

    shape1->draw();  // ()
    shape2->draw();  // []

    delete shape1;
    delete shape2;
}
```

In this example, `Shape` is an abstract class with a pure virtual function `draw()`. The derived classes `Circle` and `Square` provide their specific implementations of `draw()`. By using `virtual` in the base class, we ensure that when `draw()` is called on a base class pointer (e.g., `Shape*`), the correct derived class function is invoked. This is an example of polymorphism in action.

1.4.1 Interfaces

An *interface* in C++ is a class that contains only pure virtual functions. These are functions that are declared but not defined in the base class, essentially acting as a blueprint that derived classes must follow. An interface defines a contract of behavior but does not implement it. The purpose of interfaces is to provide a common set of operations that various classes can implement in their own specific ways.

In C++, an interface is typically created by defining a class with pure virtual functions. A pure virtual function is declared by appending `= 0` to the function declaration. A class containing at least one pure virtual function is considered an abstract class, and it cannot be instantiated directly.
Here's an example of an interface in C++:

```
#include <iostream>

class Shape
{
public:
    virtual void draw() = 0;  // pure virtual
    virtual double area() = 0; // pure virtual
};

class Circle : public Shape
```

```
{
private:
    double radius;
public:
    Circle(double r) : radius(r) {}

    void draw() override {
        std::cout << "()\n";
    }

    double area() override {
        return 3.14159 * radius * radius;
    }
};

class Square : public Shape
{
private:
    double side;
public:
    Square(double s) : side(s) {}

    void draw() override {
        std::cout << "[]\n";
    }

    double area() override {
        return side * side;
    }
};

int main()
{
    Shape* shape1 = new Circle(5.0);
    Shape* shape2 = new Square(4.0);

    shape1->draw();
    std::cout << "Area: " << shape1->area() << "\n";

    shape2->draw();
    std::cout << "Area: " << shape2->area() << "\n";

    delete shape1;
    delete shape2;
}
```

In this example, the class `Shape` is an interface with two pure virtual functions: `draw()` and `area()`. The classes `Circle` and `Square` both implement the `Shape` interface, providing specific implementations for the `draw()` and `area()` functions.

An important aspect of interfaces is that they allow for polymorphic behavior, similar to abstract classes, but they focus purely on the contract that derived classes must follow. In this case, both `Circle` and `Square` can be treated as `Shape` objects, even though they have different internal data and behavior. This allows for writing flexible and extensible code that can work with any class that implements the `Shape` interface.

While C++ does not have a built-in interface keyword like some other languages, it is common practice to define interfaces using abstract classes with pure virtual functions. An interface in C++ essentially serves as a type for polymorphic behavior, ensuring that classes adhere to a common structure and can be used interchangeably in certain contexts.

2 Operator Overloading

Operator overloading in C++ allows you to define how operators (such as `+`, `-`, `*`, `==`, etc.) behave for user-defined types (classes or structs). By overloading operators, you can make your custom types more intuitive and natural to use.

Operator overloading enables you to redefine the way operators work for your classes. For instance, instead of having to call a function to add two objects of your class, you can use the `+` operator directly, just as you would with built-in types like `int` or `double`. To overload an operator, you need to define a special member function or a non-member function with the `operator` keyword, followed by the operator symbol being overloaded.

```
1  class Point
2  {
3  private:
4      int x, y;
5
6  public:
7      Point(int x, int y): x(x), y(y) {}
8
9      // overload addition for Points
10     Point operator+(const Point& other)
11     {
12         return Point(
13             this->x + other.x,
14             this->y + other.y
```

```
        );
    }
};

int main()
{
    Point p1(1, 2), p2(3, 4);
    Point p3 { p1 + p2 }; // p3 == { 4, 6 }
}
```

In this example, the `operator+` function takes another `Point` object as an argument and returns a new `Point` object whose parts are the sums of the corresponding parts of the two operands. This is a typical use of operator overloading in C++.

2.1 Overloading Non-Member Functions

Here is an example of overloading the `<<` operator as a non-member function:

```
friend std::ostream& operator<<(
    std::ostream& os, const Point& p
)
{
    os << '{' << p.x << ',' << p.y << '}';
    return os;
}
```

In this case, the `<<` operator is defined outside the class and is marked as a `friend` to allow access to private members. The function returns a reference to the `std::ostream` object to allow for chaining of the output stream:

```
int main()
{
    Point p(13, 0), q(144,10);
    std::cout << p << std::endl << q << std::endl;
}
```

Output:

```
{13,0}
{144,10}
```

2.2 Commonly Overloaded Operators

Some operators are commonly overloaded, depending on the nature of the class and the types of operations it needs to support. Here are a few of the most frequently overloaded operators:

- `+`, `-`, , `/` Arithmetic operators
- `==`, `!=`, `<`, `>` Comparison operators
- `[]`, `()` Array subscript and function call operators
- `<<`, `>>` Stream insertion and extraction operators
- `=`, `+=`, `-=`, `=`, etc. Assignment operators

```
1  struct Point
2  {
3      int x, y;
4
5      bool operator==(const Point& other)
6      {
7          return (this->x == other.x)
8              && (this->y == other.y);
9      }
10 }
```

2.3 Functors

A functor is an object that can be called as if it were a function. This is done by overloading the function call `operator()`:

```
1  class RNG
2  {
3  private:
4      unsigned seed;
5
6  public:
7      RNG(unsigned seed): seed(seed) {}
8
9      unsigned operator()(unsigned min, unsigned max)
10     {
11         const unsigned a { 1664525 };
12         const unsigned c { 1013904223 };
13         seed = (a * seed + c);
14
```

```
        return min + (seed % (max - min + 1));
    }
};
```

3 Copy Constructors

A *copy constructor* is a constructor that creates an object using another object of the same `class`. It does so by taking a `const` reference to another object:

```
class MyClass
{
public:
    MyClass() = default; // default constructor
    MyClass(const MyClass& o); // copy constructor
};
```

One use case of a copy constructor is tracking object copies within a project to optimize performance:

```
class C
{
private:
    static int copy_cnt;

public:
    C() = default;
    C(const C& c)
    {
        ++C::copy_cnt;
    }

    static void print_copies()
    {
        std::cout << C::copy_cnt << std::endl;
    }
};

int C::copy_cnt { 0 };
```