The 10 Minute Guide to Object Oriented Programming

Why read this?

Because the main concepts of object oriented programming (or OOP), often crop up in interviews, and all programmers should be able to rattle them off and feel comfortable talking about them.

If you come from a low-level, or even machine-level programming background, it’s still A Good Thing to know what this object oriented lark is all about.

OOP deals with objects

Unfortunately there is no standard definition of “object oriented programming” that everyone agrees on, aside from the fact that it uses objects, which is possibly one of the reasons why so many people who use it day-in and day-out have trouble defining or explaining what they are actually doing.

OOP requires a particular way of thinking. Instead of planning out a sequence of instructions in your program, you need to think about what you might want it to do, break it up into useful objects, and then allow those objects to interact with each other. There is no set order or route through the code (although all paths should be testable, but that’s a different subject entirely!).

An analogy I often use is to imagine a room full of people talking to each other. Each person is an object, and each object reacts to what other objects around them are saying.

How is OOP implemented?

An object is a concept that is supported by an OOP language, such as Java or C++. Each language varies slightly in it’s approach, but the basic idea is that the language provides you with a way to define an object that you want to use, and then you can generate as many, or few, of those objects as you need.

In C++, you create (or instantiate) objects by declaring a class. A class acts as a template for your object creation. A class contains methods that dictate how the object can be used and how it will respond to stimuli.

The main principles of OOP

This is a subject open to debate, but there are essentially 3 core principles of OOP, which rather memorably spell out the word PIE:

1. Polymorphism

The hardest one to define.

Polymorphism describes the ability of different objects to be accessed by a common interface.

If that makes no sense, don't worry, I'll be covering this below, and providing a simple example, so when someone asks you in an interview, "Can you explain polymorphism to me?" you will never be at a loss for words again.

2. Inheritance

Inheritance describes the ability of objects to derive behaviour patterns from other objects.

This is an amazing concept which opens up the ground for easily creating interfaces/APIs, and reducing the duplication of code in your program. I'll take a closer look at this shortly.

3. Encapsulation

Encapsulation describes the ability of objects to maintain their internal workings independently from the program they are used in.

This basically means that you define object-related behaviour *inside* your object, and it therefore becomes self-reliant - you don't need to have supporting code in place for your object to work correctly. Again, I will cover this in more detail very soon.

So there you have it. The three cornerstones of OOP, and a nice handy definition of each.

Now let's look at each one in more detail.

Polymorphism

I was once asked in an interview, “Can you tell me what polymorphism is?”

A straightforward enough question. Or is it? This can be exactly the kind of question that catches you off guard. You’ve used objects for years and of course you know what polymorphism is. But can you summarize it concisely? Let's lay it all out clearly, so the next time someone asks, you can say, “Yes, let me explain it to you.”

The word

The word is greek in origin and translates literally to ‘many’ + ‘forms’.

In programming

Polymorphism is a descriptive term used in sciences such as biology and chemistry, but it has also been adopted by the object oriented programming methodology.

In a single sentence, polymorphism describes the ability of different objects to be accessed by a common interface.

What does that mean though?

At its most basic, an object is polymorphic if it is related to one or more other objects that can ALL be utilized using the same methods.

To illustrate this with a trivial (and very non-scientific) example, consider the following.

In a room stands a mouse, a cat and a dog. From the far corner of the room, a speaker emits the sudden sound of a balloon popping. The mouse, the cat and the dog all immediately turn toward the direction of the sound. The dog barks, the cat twitches its tail and the mouse sniffs the air.

Programmatically, we can summarize this bizarre scenario as follows:

The mouse, cat and dog are 3 unique objects. They are not the same, but they have some characteristics that are the same. Together, they descend from a more generic superclass, which we will call a mammal. Mammals have ears, which allow them to process sound. The sound of the balloon popping causes each ‘object’ to process the sound in its own way: barking, tail-twitching and sniffing.

These ‘objects’ are polymorphic. Why?

Because although they all behave in different ways, the same 'method' (ears) can be used to interact with them, and that method derives from a superclass (all mammals have ears). And thus here is the crux of the whole thing:

You don’t need to know which object you are talking to in order to utilize the method to talk to it.

Polymorphism allows you to talk to an object, without knowing exactly what the object is. If an object is polymorphic, it could be one of many different forms defined by you, the programmer, and descended from a generic base class. All you need to know is what generic type you are talking to and the available methods to talk to it.

Once you’ve called upon the method, the polymorphic type of the object will determine exactly how it behaves in response.

Polymorphism Example in C++

I've written up the example above in C++ code, so you can see it here with the actual nuts and bolts of declaring and using objects polymorphically.

I've kept the code in header files only for brevity, but as you well know, in the real world, objects would be split out into .h and .cpp files. The program consists of the main.cpp file, a base class called Mammal, and three derived classes called Dog, Cat and Mouse.

1. main.cpp

This is where it all comes together and you see polymorphism in action. I've declared an array of pointers to the Mammal base class, and for each one have called the SendLoudNoise() method.

The output of the program should look like this:

```
Woof woof!
--- Twitch my tail ---
--- Sniff the air ---
```

Which means that although I am calling the method on the base class, it is the derived class that is responding, as it overrides the SendLoudNoise() method in each case.

Here is the main.cpp file:

```
#include <iostream>
#include "Mammal.h"
#include "Dog.h"
#include "Cat.h"
#include "Mouse.h"

const int cNumAnimals = 3;

int main()
{
    Mammal* pAnimal[cNumAnimals] = { new Dog, new Cat, new
Mouse };

    for (int i = 0 ; i < cNumAnimals ; ++i)
    {
        std::cout << pAnimal[i]->SendLoudNoise() << std::endl;
        delete pAnimal[i];
    }

    return 0;
}
```

2. Mammal.h

The Mammal class, our base class, defines the virtual method SendLoudNoise(), but we never see the text that this method outputs. This proves that our polymorphism example is working correctly.

Note that methods you want to override *must* be declared as virtual. If you remove this one keyword, the output of the program would be 3 lines of “I am a generic mammal”!

```
#include <string>

#ifndef MAMMAL_H_
#define MAMMAL_H_

class Mammal {
public:
    Mammal() {};
    virtual ~Mammal() {};

    virtual std::string SendLoudNoise()
    {
        std::string str("I am a generic mammal");
        return str;
    }
};

#endif /* MAMMAL_H_ */
```

3. Dog.h

The next three files are very similar. Each one inherits from the Mammal class and then overrides the SendLoudNoise() method. Aside from that, they are almost identical.

You don't have to override a method in a derived class. If you wanted default behaviour, you could specify that in your base class and not mention it in your derived class. Then when that method is called, the object would default to using the base class version.

```
#include "Mammal.h"

#ifndef DOG_H_
#define DOG_H_

class Dog : public Mammal {
public:
    Dog() {};
    virtual ~Dog() {};

    std::string SendLoudNoise()
    {
        std::string str("Woof woof!");
        return str;
    }
};

#endif /* DOG_H_ */
```

4. Cat.h

```
#include "Mammal.h"

#ifndef CAT_H_
#define CAT_H_

class Cat : public Mammal {
public:
    Cat() {};
    virtual ~Cat() {};

    std::string SendLoudNoise()
    {
        std::string str("--- Twitch my tail ---");
        return str;
    }
};

#endif /* CAT_H_ */
```

5. Mouse.h

```
#include "Mammal.h"

#ifndef MOUSE_H_
#define MOUSE_H_

class Mouse : public Mammal {
public:
    Mouse() {};
    virtual ~Mouse() {};

    std::string SendLoudNoise()
    {
        std::string str("--- Sniff the air ---");
        return str;
    }
};

#endif /* MOUSE_H_ */
```

And that's all there is to it – a very basic example of using polymorphism to enable interaction with different objects via a base class pointer.

Inheritance

Earlier in this guide, I described inheritance as the ability of objects to derive behaviour patterns from other objects. But what does this mean in practice?

An object can inherit behaviour (in the form of methods), and even data variables, from a parent class, in much the same way as children inherit characteristics from their parents.

Why is this something that we might want to do in programming?

Inheritance accurately models relationships

Using inheritance can make a program easier to understand. The concept that an object derives from another object is quite intuitive. As long as you are deriving correctly, inheritance can help break a complex model down into workable objects.

You are deriving correctly when your inherited object **IS A** base class object. E.g. a glider **IS A** plane; a dog **IS A** mammal. If you can't say this, you should be questioning if inheritance is the correct approach.

Inheritance allows us to create polymorphic objects

By declaring methods as virtual, you have the option of overriding them in derived classes. This is how polymorphism is achieved.

Inheritance allows us to create interfaces

By declaring methods as pure virtual, you can create an abstract base class (ABC) that cannot be instantiated (i.e. it can't exist as an object in its own right). Consequently you end up with a class that defines a set of methods, but has no implementation of those methods – i.e. a lovely interface.

It is worth noting that the **IS A** relationship doesn't have to hold when you are creating an interface, since the interface cannot be instantiated itself.

Inheritance isn't a solution for reducing/reusing code

Inheritance *can* reduce the amount of code that needs to be written, but this isn't *why* it exists. It's important to remember this. If you find you are using inheritance to promote code-reuse, then you need to look carefully at whether the relationship between derived class and parent class is still correct.

Multiple inheritance

Multiple inheritance is the process of deriving a class from two or more base classes.

Eek! Why would anyone want to do that? Well, sometimes you need the functionality of two classes combined into one – for example, you may be working with a library and find that a class you have already derived from your own base class also needs to derive from one of the base classes provided in the library. Essentially, if you stick to deriving from abstract base classes, multiple inheritance shouldn't cause you too much pain.

I've worked in places that support opposite ends of the opinion spectrum on this topic. One company unanimously insisted that multiple inheritance was the work of the devil. Another company used multiple inheritance like it was going out of fashion.

Having seen its usefulness in live code, but also having been subjected to endless warnings of its dangers, I tend to take a more moderate view. Yes, there is a place for multiple inheritance, but you should be clear about why you need it.

There is no concrete way of deciding if multiple inheritance is the right solution to a particular problem, but if it seems like a simple and elegant way to solve an otherwise tricky issue, then it might just be the right thing to use.

This guide only discusses public inheritance, not private or protected.

Encapsulation

I defined this above as the ability of objects to maintain their internal workings independently from the program they are used in.

This is actually the simplest of the three concepts we've covered. With respect to C++, encapsulation describes the use of a class to contain both its relevant data, and the methods to manipulate its own data.

In practice

As a simple example, you might define a class called Message. This class can contain various data members, such as the sender's address, the date the message was written, the destination address, and of course, the actual message inside.

The class can also contain methods to manipulate a Message object. For example it might have a method to set the object status to 'read', and another method to print the message to the screen.

The fact that the Message class both contains its own data, and provides a way to interact with that data, is what encapsulation is all about.

It's such a fundamental part of C++ (and other OO languages), that it is often taken for granted.

Why is encapsulation a good thing?

The benefits of encapsulation are:

- It can restrict access to an object's data.

If data is declared as private and methods are generated to access that data, it can provide a good level of control over the data and how it is used.

- It allows the programmer to modify the object under-the-hood without disrupting the rest of the program.

If you want to change the internals of your class, you can do so without having to modify code elsewhere in the program, as long as you keep the methods to access that class the same.

Encapsulation is the class not the object

Note that encapsulation is provided by the class, not the instantiated object (there is a subtle difference). This is because a class can provide the ability to access information about every object that is instantiated from it, whereas a single object is just that.

Conclusion

That wraps up my 10 minute guide to object oriented programming; I hope you found it useful.

Remember the 3 cornerstones (PIE), and what each of them mean, and you are well on your way to creating better programs by using classes and objects in the way in which they were designed. Have fun!