Object Oriented Programming: Principles & Concepts

# 1 Introduction

Object Oriented Programming (OOP) is a programming paradigm that moves the focus of program design away from just methods (functions) and towards classes, objects, and the way they interact.

OOP brings forth many advantages, but also some disadvantages, as can be expected from any approach. Three of the major hallmarks of OOP are encapsulation, inheritance, and polymorphism. These functionalities relate to the way objects are constructed, and the way they interact.

Below, I will talk about the concepts behind these three topics, how they are implemented in C++, and their benefits and drawbacks.

# 2 Encapsulation

## What is it?

Encapsulation is an idea within OOP design which requests the isolation of data members (variables) within objects, meaning they can be freely accessed by methods of the same object but are not visible to any methods outside of the object. They can however be made available via so-called ‘getter’ and ‘setter’ methods which are public but can restrict changes made to the data members.

The aim of encapsulation is to protect an object’s data members so that other objects in the program cannot set their values to meaningless or problematic values or read their values when not desired or required.

One of the main benefits of encapsulation is program design – if thought about and implemented properly, encapsulation helps a class-oriented program to naturally achieve readable and maintainable code as it encourages careful class design and only giving access to data members when necessary.

## Implementation

In C++, the header file for a class defines its structure – all methods and data members are declared here. In addition to outlining what methods and data members will make up the class, the programmer can specify the levels of protection assigned to each part – public, protected, or private. See Figure 1 for a demonstration of how this is done in C++.

While is it possible to set data members to public or protected, there is very little to no reason to do so in a well-constructed program that conforms to good practises. Getter and setter methods can be created to allow controlled access to the data members, and these have almost no performance impact and do not hinder the readability of the code.

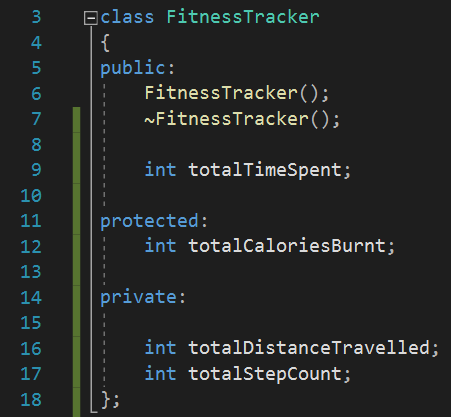


Figure 1 Some data members in a class with different levels of protection

# 3 Inheritance

## What is it?

Inheritance is the idea that, when a class is defined in a program, it can be set to inherit from another class. This results in an exact duplication of structure (and thus functionality) from the benefactor to the inheritor.

The idea here is that so-called ‘base classes’ can be constructed at a more abstract level, for example “Animal”, and child classes (classes which inherit from that class) , for example “Dog” or “Cat”, can take on its structure, and extend or specify its functionality.

Imagine the Animal class has an abstract method named “MakeNoise” – within the Animal class this has no implementation; if called it would do nothing. In the Dog class, which inherits from Animal, the MakeNoise method can be implemented to provide the functionality of barking. Respectively, in the Cat class the MakeNoise method would be implemented to meow.

## Implementation

When declaring a class in C++, a small amount of code can be added to implement inheritance:

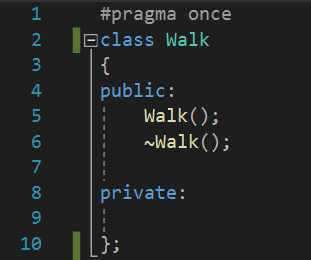
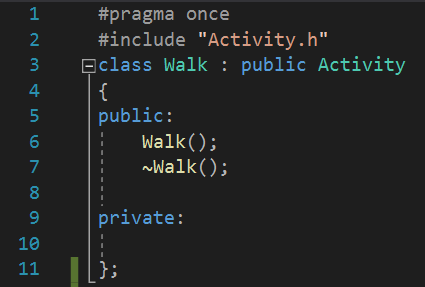


Figure A basic class declaration in C++

Figure A class declaration that inherits from "Activity" in C++

The colon specifies inheritance, the public keyword specifies that all public members inherited from the base class will be public in the child class, and the Activity keyword is specifying the class to inherit from.

Once inheritance has been achieved, it must be utilised – simply linking classes together will not achieve anything. Below is an example of an abstract method declared within a base class and its implementation in a child class:

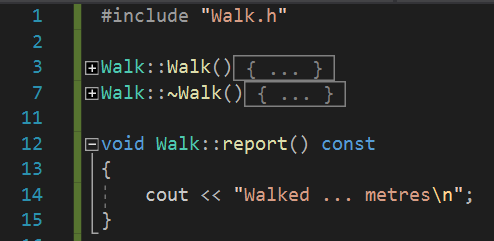
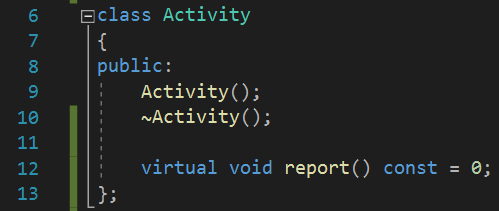


Figure 4 The Activity class, with an abstract method

Figure 5 An implementation of the abstract method in a child class

# 4 Polymorphism

## What is it?

Polymorphism somewhat extends on inheritance; it is the concept that numerous different classes can inherit from one ‘base class’, and while the different classes have different functionalities, they can all be interfaced with via the base class.

Allowing for complex class structures to have very simple interfaces, polymorphism is an extremely powerful tool

## Implementation

After an inheritance structure is set up in a class-oriented C++ program, virtual methods in base classes that have been implemented in child classes can be used to call the child class’ methods through the base class – this is polymorphism in practise.

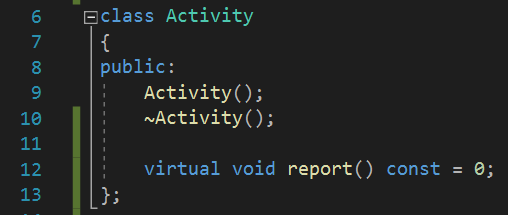


Figure 4 A base class with an abstract method

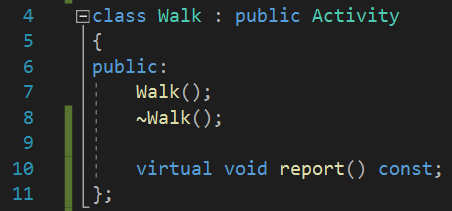


Figure 5 The abstract method is given a definition in a child class

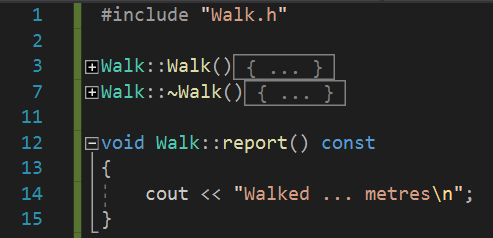


Figure 6 One implementation of the abstract method

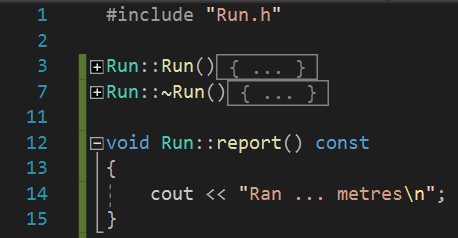


Figure 7 Another implementation of the same abstract method

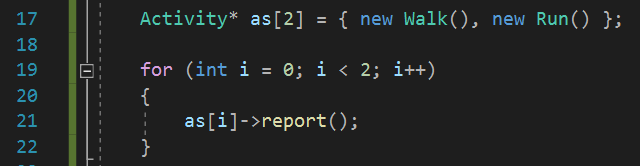


Figure 8 Instances of Walk and Run are created, and placed into an array of Activities

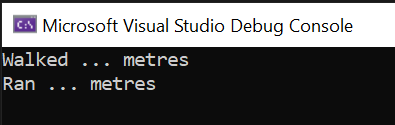


Figure 9 When the code is run, the same abstract method within Activity is called, but because it has been implemented differently in Walk and Run it produces different outputs.

# 5 Conclusion

The features and tools of OOP covered in this report have drawbacks such as added complexity in program design and slower runtimes than procedural programs due to the large size of objects in computer memory, but for many modern solutions OOP provides far more benefits than drawbacks, creating well-structured, readable, maintainable and powerful code bases.