Object-Oriented Programming  
Tutorial 03 - Classes and Objects

## Introduction

The process you have used so far to design and implement your programs using functions is called **procedural** **programming**. With this methodology you break down a task into functions to process the data, and identify the required variables and data structures. In **object-oriented programming** you identify data types and the relationships between them required to fulfil a task, then for each data type you combine both data structures and functions that operate on that data into a single entity called a **class**. The new data type that is defined by a class can be used to declare specific instances of a class, called **objects**. Although both procedural and object-oriented are both valid methodologies, object-oriented programming has largely superseded procedural in many software development industries due to the ease of implementation of features such as **encapsulation** (combining data and functionality) , **abstraction** (hiding implementation detail), **modularity** (separation into interchangeable components), and **inheritance** (extension of data types).

### Classes

A **class** can be thought of as a combination of two programming concepts that you have already come across - functions and structures. A class defines member variables in the same way as a structure, these are called class **attributes**, and **in addition** it includes functions definitions called class **methods**. A class is usually designed to represent the properties, state and behaviour of a specific type of entity. For instance, a Car class might have attributes representing model, engine size and speed, and methods such as accelerate and brake:

Class: Car

Attributes: Model

Engine Size

Speed

Methods: Accelerate()

Brake()

### Objects

Like structures, the declaration of a class creates a new data type, and you can declare variables using this data type in the same way as any other type. Variables declared with a class data type are called **objects**. Objects are individual **instances** of a class in exactly the same way as a variable is a specific instance of a data type, or to put it another way a class can be thought of as a **blueprint** for creating objects:

Object: marks\_Car

Model :BMW 323

Engine Size :2.5

Speed :70

Accelerate()

Brake()

Class: Car

Attributes: Model

Engine Size

Speed

Methods: Accelerate()

Brake()

Object: marks\_Next\_Car

Model :Bugatti Veyron

Engine Size :8

Speed :250

Accelerate()

Brake()

Every object declared from a class contains all of the attributes and methods defined in the class.

### Declaring **Classes**

A Car class might be declared like this:

class Car

{

private:

int m\_speed; // private attribute

void turnOnBrakeLight(void); // private method

public:

char m\_model[50]; // public attribute (would normally be private)

float m\_engine\_size; // public attribute (would normally be private)

void accelerate(void); // public method

void brake(void); // public method

}; // don't forget the semicolon

Notice the format is very similar to a structure, but with the addition of functions, i.e. methods. There are also two new keywords public: and private: whichare needed when using classes, these will be discussed in the next section.

### Encapsulation

The idea of combining data and the operations performed on that data into a class is called **encapsulation**. Encapsulation allows programmers to treat the state and behaviour of an object as a single entity, as well as restricting access to the attributes in the object. Generally, classes are designed so that the **attributes are only accessible via the methods** in the class, so that when objects are created each object will have its own copy of each attribute, and only that object's methods can affect those attributes. Restricting access in this way allows the creator of a class to control what can access the data contained in an object - This can improve the robustness of a software system. It also has the advantage that the way the data in a class is represented can be changed without affecting those using the class.

### **public:** and **private:** Access Specifiers

In order to provide encapsulation you use the public: and private: **access specifiers** inside the class declaration. The public: access specifier determines those attributes and methods that are accessible outside of the class, while private: covers those accessible from within the class. In the above car class the m\_speed attribute is defined as private and therefore can only be accessed from the methods in the class. The turnOnBrakeLight() method is also private, and only accessible from other class methods. The m\_model and m\_engine\_size attributes, and the accelerate() and brake()methods, are set as public and are accessible from any code outside of the class. However, these two attributes would normally be declared in the private section; they are shown as public just for demonstration purposes.

### Defining Methods

If you declare a method within a class you will need to provide a method definition that specifies what the method does. This is exactly the same as defining a function at the time of declaration. Defining a method outside its class is in fact very similar to defining a function, so for the accelerate() and brake()methods:

void Car::accelerate(void)

{

m\_speed+=3; // Part of Car class so can access **private** attribute

}

void Car::brake(void)

{

m\_speed-=5; // Part of Car class so can access **private** attribute

}

In the case of class methods you need to prefix the method name with the class name and the scope resolution operator '::'. This tells the compiler which definition belongs to which class. This is important as different classes may have methods with the same name, and without the class name prefix the compiler wouldn't be able to tell which belonged to which. For instance, a Truck class might have the same methods as the Car class:

class Truck

{

private:

int m\_speed; // private attribute

public:

void accelerate(void); // public method

void brake(void); // public method

};

void Truck::accelerate(void)

{

m\_speed++;

}

void Truck::brake(void)

{

m\_speed--;

}

in this case the method definitions are prefixed by the Truck class, so these methods will be associated with the correct class.

One common mistake is to forget to include the class prefix, e.g.:

void accelerate(void) // missing Car::

{

m\_speed+=3;

}

This is interpreted by the compiler as a normal function rather than a method of a class, and the build will probably fail. If your build fails after adding new method definitions to your code, your first check should always be if you have missed out the class prefix.

### Creating and Accessing Objects

The declaration of a class creates a new data type, so you can declare variables of this type. Variables that are declared using a class are called objects, and are declared in exactly the same way as other variables:

Car marks\_Car; // Create a variable named m**arks\_Car** of type **Car**

Car marks\_Next\_Car;

Each object has its own set of attributes that are totally separate from other objects. You can access **public** attributes using the same **dot operator** used to access structure member variables:

marks\_Car.m\_engine\_size = 2.5f;

marks\_Next\_Car.m\_engine\_size = 8.0f;

In addition, you can use the same operator to access the **public methods** of a class:

marks\_Car.accelerate();

marks\_Next\_Car.brake();

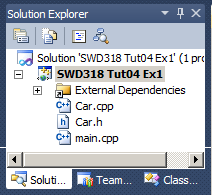
Notice, however, that you cannot access private attributes or private methods using the dot access operator:

marks\_Car.m\_speed = 100; // **not valid**, speed is **private**

### Using Classes in C++

Although you can declare and define classes withinyour main program, e.g. **main.cpp**, as you have done so far with all of your exercises, it is in fact common practice to put each class in separate files. These files are then imported into your program. A class normally consists of two separate files - the **header** file (with a **.h** suffix) that contains the class declaration, and the **source** file (with a **.cpp** suffix) that contains the class method definitions.

For the above Car class you would first of all add **Car.cpp** and **Car.h** files to your project, in exactly the same way as you do for **main.cpp**. Your project Solution Explorer should look something like this:



**Car.h** would consist of the class declaration:

#pragma once

class Car

{

private:

int m\_speed;

void turnOnBrakeLight(void);

public:

char m\_model[50];

float m\_engine\_size;

void accelerate(void);

void brake(void);

};

The line at the start, #pragma once, is a special **compiler directive** that tells the compiler to only compile this file once. Header files can be included from several different source files, as you will see when defining **main.cpp** and c**ar.cpp**, and major problems can occur as your programs get larger unless you begin every class header file with this. **Get into the habit of always adding #pragma once at the start of your header files when you create them.**

**Car.cpp** would then contain the class method definitions:

#include "Car.h"

void Car::accelerate(void)

{

m\_speed++;

}

void Car::brake(void)

{

m\_speed--;

turnOnBrakeLight();

}

void Car::turnOnBrakeLight(void)

{

// turn on brake light

}

Notice the use of another compiler directive #include "Car.h". You've used this type of directive before to include C/C++ libraries, but in this case it is being used to include the header file for the Car class. Without this line the compiler wouldn't know where to find the Car declaration and compilation would fail. You should normally put your own header files after the C/C++ libraries that you include.

Finally, here is **main.cpp**:

#include <iostream>

#include "Car.h"

using namespace std;

// DECLARE functions

void wait\_for\_keypress(void);

int main()

{

Car marks\_Car;

Car marks\_Next\_Car;

marks\_Car.m\_model[0] = 'B';

marks\_Car.m\_model[1] = 'M';

marks\_Car.m\_model[2] = 'W';

marks\_Car.m\_model[3] = '\0';

marks\_Next\_Car.m\_model[0] = 'B';

marks\_Next\_Car.m\_model[1] = 'U';

marks\_Next\_Car.m\_model[2] = 'G';

marks\_Next\_Car.m\_model[3] = 'A';

marks\_Next\_Car.m\_model[4] = 'T';

marks\_Next\_Car.m\_model[5] = 'T';

marks\_Next\_Car.m\_model[6] = 'I';

marks\_Next\_Car.m\_model[7] = '\0';

marks\_Car.m\_engine\_size = 2.5f;

marks\_Next\_Car.m\_engine\_size = 8.0f;

marks\_Car.accelerate();

marks\_Next\_Car.brake();

wait\_for\_keypress();

}

// DEFINE functions

void wait\_for\_keypress(void)

{

cout << "Press any key to continue" << endl;

cin.get();

}

Notice that **main.cpp** also needs to include the **Car.h** header if it wants to be able to use the Car class.

## Exercises

**In order to aid understanding of how the exercises work, use the debugger to step through all of the code you write for the exercises.**

**You are advised to write notes on all aspects of the tutorial and exercises in your notebooks. This can then be used to help with your assignments.**

#### Exercise 1a

1. Create a new project, with a **main.cpp**, **Car.cpp** and **Car.h**. Copy the code from the tutorial into these files, and compile and run to ensure that it works.
2. Step through the code using the debugger and the watch windows to see what is happening in the code. You can see the contents of each class by clicking on the **+**. You can step into and out of class methods in the same way as into functions by using **F11** and **SHIFT-F11**.
3. You might notice that the speed attribute starts as a large negative number. Write in comments and in your notebook why you think this is.
4. In order to fix the problem with m\_speed you need to be able to write a value to it. However, because m\_speed is a private attribute you cannot do this directly.
   1. Add a method to Car called SetSpeed() that takes one integer parameter and has no return value. The method should set m\_speed to the value passed in via SetSpeed(). You may need to revisit the work you have done on functions, which work in the same way as methods.
   2. Add a call in main() for each of the Car objects to set their m\_speed to zero before any acceleration or deceleration.
   3. Check in the debugger that this works correctly.
5. Add several more Car objects to your code, and set the name, engine size and speed for all of them. Check all the values in the debugger.
6. [extra] For an easier way to set the value of char arrays with string values check out the C function strcpy().

#### Exercise 1b

1. Add a method to your Car class that displays the current speed of a car.
2. Create a loop that accelerates and/or decelerates a Car object, and for each iteration of the loop display the speed.

#### Exercise 1c

1. Add a Truck class to your project. This should have the same attributes and methods as the Car class, but should add a **private** integer attribute called m\_total\_load.
2. The acceleration and braking amounts should be less than for a Car object.
3. Add public methods to set and display the value of m\_total\_load.
4. Create several Truck objects, and initialise all the attributes. Display the attributes in a loop whilst accelerating or decelerating the Trucks.

#### Exercise 1d

Create a loop that randomly accelerates or decelerates a Car object, and have a Truck object attempt to match the speed of the Car. This will require new methods for both classes that return the object's speed.

#### Exercise 1e

Create a MotorBike class. Create private attributes that you think will be useful for such a class, and public methods to set, return and display the values of the attributes.

#### Exercise 2

1. Create a new project.
2. Imagine you are designing a system to store contact information for people. Design and implement a Contact class that will store relevant information as private attributes, along with public methods to set and display that information. Don't forget to create new **.cpp** and **.h** files for the class.
3. Create some objects of the Contact class type, and ask the user to input information for those contacts.
4. Display all of the information of all of the contacts.

#### Exercise 3

1. Create a new project.
2. Design and create a class that represents a game character that has private attributes of name, health and attack strength.
3. Add class methods to set the name, health and the attack strength.
4. Add a method called GetNextAttack() that returns a random integer value between 0 and the attack strength.
5. Add a method called TakeDamage() that takes an integer as parameter and subtracts that amount of damage from the health attribute.
6. Add a method called IsDead() that returns true if health is less than 0, or false otherwise.
7. Add a method that displays the name and health.
8. Create two (or more!) character objects and initialise the name, health and attack values.
9. In a loop
   1. Call the GetNextAttack()method for each character.
   2. Use that value to cause damage to another character using the TakeDamage() method.
   3. Output all the character states.
   4. End the loop when one or more of the characters is dead.

#### Exercise 4

Consider how you might improve exercises in previous tutorials by using objects instead of the original method. Write in comments and/or notebook why you think this might be better. Implement the changes if you think you know how.

#### Exercise 5

In order to become a good programmer you should be spending several hours a week in self-directed research. This can be as simple as following an online tutorial, or as complex as taking what you have learnt to design and implement programs of your own that solve problems that interest you. I can't stress enough that the more you program, the better you will get at it.