Object-Oriented Programming  
Tutorial 05 - Using Objects, Static Attributes and Methods

## Introduction

This tutorial will demonstrate how to correctly assign and initialise objects with other objects, how to use objects when creating and initialising structures and arrays, as well as the use of objects with functions and methods. Finally, **static attributes** and **static methods** will be introduced which allow information about a class to be stored and manipulated independently from any actual objects created by the class.

### Assigning Objects

It is often useful to be able to **copy** objects, and this is done in the same way as any other variable using the **assignment operator**. By default, the **assignment operator** copies all attributes of one object into another. So, for instance:

Car marks\_Car(10); // constructor initialises speed to 10

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

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

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

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

Car copy\_of\_marks\_Car;

copy\_of\_marks\_Car = marks\_Car;

cout << copy\_of\_marks\_Car.m\_model << ", " << copy\_of\_marks\_Car.GetSpeed() << endl;

would result in the output:

BMW, 10

as all of the attributes of marks\_Car have been copied into copy\_of\_marks\_Car.

### Initialising Objects with Objects

It is also possible to **initialise** objects using an **existing** object. The following code demonstrates this:

Car marks\_Car(10); // constructor initialises speed to 10

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

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

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

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

Car copy\_of\_marks\_Car(marks\_Car);

cout << copy\_of\_marks\_Car.m\_model << ", " << copy\_of\_marks\_Car.GetSpeed() << endl;

The output will be the same as the previous example, as this operation initialises all of the attributes of copy\_of\_marks\_Car with those in marks\_Car. Notice that this syntax is the same as any other constructor, except the type of the single parameter is the **same type as the class**. Whenever you create a class the compiler automatically creates this special **copy constructor** for this initialisation to work.

You can use the normal initialisation assignment syntax used for other types instead of the copy constructor syntax used above, i.e.

Car copy\_of\_marks\_Car = marks\_Car; // calls copy constructor, same as copy constructor above

### Objects Within Structures

As a class is a data type it can be a **member of a structure**. Here is a simple structure that contains a single Car object to demonstrate:

struct my\_struct

{

Car m\_A\_Car;

};

You can access the Car member in the same way as usual with the structure member access operators, i.e.

my\_struct test;

my\_struct \*p\_test = &test;

test.m\_A\_Car.accelerate(); // adds 5 to speed attribute

p\_test->m\_A\_Car.m\_model[0] = 'B';

cout << test.m\_A\_Car.m\_model << ", " << p\_test->m\_A\_Car.GetSpeed() << endl;

When an object is a part of a structure in this way, then the default constructor (one with no parameters) is called for the class whenever a variable of the structure type is created, so in this case the default Car constructor would have been called and after the operations the output would be:

Bone, 5

assuming the accelerate() method adds 5 to m\_speed.

You can use the structure initialisation syntax with constructors if you want to initialise the embedded object with a different constructor, so:

my\_struct test = { Car(11) };

would call the constructor that will initialise the m\_speed attribute to 11.

### Object Pointers and Arrays of Objects

Like other data types you can declare **pointers to objects**, which work the same way as any other pointer. Because classes are data types they can also be used to declare **arrays of objects**, which work in the same way as arrays of any other type:

Car car\_array[10];

Car \*p\_car\_array = &car\_array[9]; // point at tenth element of array

cout << car\_array[0].m\_model << ", " << p\_car\_array->m\_model << endl;

Notice here that you can declare and use a **pointer to an object** in the same way as any other data type pointer, and in this case p\_car\_array is being pointed at the last element of car\_array. When accessing attributes and methods via a pointer to an object, the same syntax as the structure pointer member access '->' is used.

When an array of objects is created, the default constructor is called for **every** element of the array, so the above code will display

None, None

as the first and tenth elements have been initialised using the default constructor, along with all the others.

You can use the array initialisation syntax to call a different constructor for some or all of the elements. In the following code two different constructors are being used to initialise the first two elements, the rest will be initialised with the default constructor:

Car car\_array[10] = { Car(22), Car(3.0, 33) };

**NOTE** - A class with **no** constructors has a default constructor (that does nothing) created for it automatically by the compiler. If you declare **one or more** constructors in a class then the default constructor is no longer automatically created. If you want to embed a class in a structure or create arrays from it then you need to create a default constructor for it, otherwise the program will not compile. It's usually a good idea to **always** declare a default constructor - you would normally want to use it to initialise the attributes, as the automatically created constructor does not do anything and the attributes would be undefined.

### Objects with Functions and Methods

Objects can be passed to functions (and methods) via parameters, and can also be set as the return type. Exactly the same syntax and rules apply when using objects with functions, so you can pass an object in by value:

void Car::display\_car\_pass\_by\_value(Car c)

{

cout << c.m\_model << ", " << c.m\_engine\_size << ", " << c.GetSpeed() << endl;

}

Car marks\_Car;

marks\_Car.display\_car\_pass\_by\_value(marks\_Car);

Remember that passing in by value causes an **entirely new local object to be created** and the contents of the variable being passed in to be copied into the temporary local object, which can be slow for large objects and can have side effects if you wish the function to change the data contained in the object passed in. Usually you would use pointers or references to pass objects in to functions:

void Car::display\_car\_pass\_by\_pointer(Car \*c)

{

cout << c->m\_model << ", " << c->m\_engine\_size << ", " << c->GetSpeed() << endl;

}

void Car::display\_car\_pass\_by\_reference(Car &c)

{

cout << c.m\_model << ", " << c.m\_engine\_size << ", " << c.GetSpeed() << endl;

}

Car marks\_Car;

marksCar.display\_car\_pass\_by\_pointer(&marks\_Car);

marksCar.display\_car\_pass\_by\_reference(marks\_Car);

Here's an example of using a pointer to an object as a return value:

Car\* Car::fastest\_car(Car \*car1, Car \*car2)

{

if(car1->GetSpeed() > car2->GetSpeed())

{

return car1;

}

else

{

return car2;

}

}

Car marks\_Car(10);

Car marks\_Next\_Car(99);

Car \*p\_fastest = fastest\_car(&marks\_Next\_Car, &marks\_Car); // will point at marks\_Next\_Car because it is the fastest

### Static Data and Methods

Sometimes it's useful to be able to store information that applies to the class, as a whole. For example, imagine that it would be useful to store the number of car objects that have been created in a program. This could be done by incrementing a variable each time a car object is created:

int number\_of\_cars = 0;

...

Car marks\_Car;

number\_of\_cars++;

but this method does rely on the programmer remembering to increment the variable for every object created. A better method would be for the class to do this automatically, and this can be done using a **static variable** in the class. When you define a static variable in a class it is treated as a single instance that represents data about the **whole class**, not any individual object, and is accessible by all objects created with that class. Here is how a variable representing the number of car objects created would be declared in the Car class using a static variable:

// inside class declaration in .h file

class Car

{

...

public:

static int m\_number\_of\_cars; // Accessible by all Car objects

...

};

The static keyword before the type - integer in this case - indicates that this is to be a single instance of a variable for the class. It doesn't belong to any specific object in a given class and exists independently of all objects, even if there have been no objects created. All objects of the class have access to static variables.

In order to initialise this attribute, you would need the following line of code in the **global scope**, i.e. **not** inside any functions or methods; one good place would be just after the #include directives of **Car.cpp**:

// declare as a global variable in Car.cpp

int Car::m\_number\_of\_cars = 0;

Here you can see the scope resolution operator is used to tell the compiler which class to associate the attribute with. This way of accessing the attribute is needed because it exists before any objects are created.

The static variable can now be used in the constructor(s) of a class to count how many instances have been created:

// Car.cpp

Car::Car(void)

{

m\_number\_of\_cars++;

...

}

Car::Car(int speed)

{

m\_number\_of\_cars++;

...

}

Car::Car(float engine\_size, int speed)

{

m\_number\_of\_cars++;

...

}

Now the constructors are set up to increment the variable every time an object is created, we can query the attribute to get this value. You simply use the class name and scope resolution operator:

cout << Car::m\_number\_of\_cars << endl; // No Car objects created, will output 0

Car car1;

Car car2(15);

Car car3(3.2, 25);

cout << Car::m\_number\_of\_cars << endl; // Three Car objects created, will output 3

Car car\_array[10];

cout << car2.m\_number\_of\_cars << endl; // Ten more Car objects created in array, will output 13

Notice in the final line of code that, if you do have an object instance of a class, you can access the attribute **via the object** instead if you wish.

If you are designing your classes with encapsulation in mind it is an excellent idea to set any static attributes as **private** in the class. This means that you will need methods to access them, and to access private static variables that exist apart from any class objects you will need **static methods** that also exist independently. This is very straightforward to do, just put the static keyword in front of the return type in the same way as you did for the attribute. Here is how you would modify the class to do this:

// inside class declaration in .h file

class Car

{

...

private:

static int m\_number\_of\_cars;

...

public:

static int get\_number\_of\_cars(void);

};

// inside class definition in .cpp file

int Car::get\_number\_of\_cars(void)

{

return m\_number\_of\_cars;

}

This method can now be used to access the private attribute instead of directly accessing it:

cout << Car::get\_number\_of\_cars() << endl; // static method called using scope resolution operator

cout << car2.get\_number\_of\_cars() << endl; // static method called using an object

## Exercises

**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 01

1. Create a new project, with a **main.cpp**, **Car.cpp** and **Car.h**.
2. Add the code for the Car class from tutorial 3 exercise 1 to the appropriate files.
3. In main() function create several (at least three) Car objects, and initialise or set the values of the attributes as you wish.
4. Create an equal number of new Car objects with the same name as the original set, but with '\_copy' appended, e.g. marks\_Car\_copy.
   1. Use assignment to copy one original object's values into the copy object after the copy has been initialised.
   2. Use the two different forms of initialisation to initialise copies with the originals.
5. Use the debugger and output values to confirm that the copies have correctly occurred.

#### Exercise 02

1. Create a new project, with the **Car.cpp** and **Car.h** copied from exercise 01.
2. In **main.cpp** declare a structure that represents a garage that has space for two cars and the name of the owner.
3. Create two Car objects and set the attributes as you see fit.
4. Create a garage structure variable called garage1 and copy the Car objects into the structure, as well as setting the name of the owner.
5. Creature another variable called garage2, but this time use initialisation syntax to set the member variables.

#### Exercise 03a

1. Create a new project, with the **Car.cpp** and **Car.h** copied from exercise 01.
2. In **main.cpp** declare an array of several Car objects, don't use any initialisation.
3. Use a loop to go through the array and set the values of the speed of each object to the **array index times ten**, as well as setting all the other values as you wish.
4. Create a Car pointer and point it at the last element of the array.
5. Use a second loop to go through the array backwards, outputting all the data for each object using the pointer variable.

#### Exercise 03b

1. Create another array of Car objects, this time initialising the first few objects with as many different constructors as you have created for your Car class (if you still have only three create at least two more).
2. Output all the data for every object in the array to confirm the different constructors were called, as well as the default constructor for the non-initialised elements.

#### Exercise 04

1. Create a new project, with the **Car.cpp** and **Car.h** copied from exercise 01.
2. Add the functions to display the Car data from the tutorial, adding in the additional information that your Car object now stores.
3. Create a Car object and pass it in to each of the three functions in turn.
4. Put a breakpoint in each function and examine the address of the Car object, comparing it to the address of the Car object being passed in. Explain in comments and your notebook what the values mean (if you don't already know, if you want to get the address of an object in the watch window, type in &variable, e.g. if your car object is called my\_car you would use &my\_car).
5. Create a function that swaps all the values of two Car objects passed in, and test it with two Car objects. What sort of parameter passing mechanism should you use, and why?

#### Exercise 05

1. Create a new project, with the **Car.cpp** and **Car.h** copied from exercise 01.
2. Add a private static variable to count the number of car objects to your Car class.
3. Add code to every constructor to increment this value.
4. Add a static method to access this variable.
5. Initialise the variable in **car.cpp**.
6. Create several Car objects with all the different constructors you have.
7. Output the number of objects created using the class name and the scope resolution operator. Check this matches the actual number created.
8. Copy your garage structure from exercise 02 into **main.cpp** and create a garage. Output the number of objects created again. Explain why the value has gone up by this amount.
9. Create an array of Car objects, using all the constructors you have as initialisers.
10. Use one of these objects to output the number of objects created, and check again that it is correct.