Object-Oriented Programming

Tutorial 4 – Constructors & Class Diagrams

## Introduction

The first exercise from the previous tutorial demonstrated that the attributes of a Car object are uninitialised after the object is created, just like any other variable. Like other variables, it is usually a good idea to initialise the attributes of a class to sensible values when creating an object of that class type. To do this, there is a special type of method called a **constructor** available in classes to allow object initialisation.

**Class Diagrams** are a useful design tool to help visualise what elements are needed to form a class.

### Constructors

The initialisation syntax that you have used before for arrays and structures does not work for classes. To initialise variables that have been declared inside a class you need to define a special method inside your class called a **constructor**. A constructor allows you to do all the appropriate initialisation of a newly created object as soon as that object is created; in other words, when you create a new object the appropriate constructor method is called automatically after the memory for the object has been allocated. Although you could set the initial values of the attributes of an object using the dot access operator - for public - and methods - for public and private - after an object is created, it is considered best practice in object-oriented programming to initialise them to sensible values in a constructor. This means that an object will be guaranteed to have valid data, assuming the constructor is correctly written, so there is no chance of forgetting to initialise values.

Constructors are declared inside a class like other methods, and **must** be named with the same name as the class. Additionally, whilst they can have parameters like other methods they **do not have a return type** declared. Here is how a constructor with no parameters, called a **default constructor**, for the Car class might be declared:

class Car

{

private:

int m\_speed;

void turnOnBrakeLight(void); // private method

public:

Car(void);

char m\_model[50];

float m\_engine\_size;

void accelerate(void);

void brake(void);

};

Notice that the attributes have been prefixed with 'm\_', as usual, the importance of this in relation to class constructors will be discussed shortly. The constructor could be defined like this:

Car::Car(void) // notice no return type defined, not even void

{

m\_speed = 0;

m\_engine\_size = 0.0f;

m\_model[0] = 'N';

m\_model[1] = 'o';

m\_model[2] = 'n';

m\_model[3] = 'e';

m\_model[4] = '\0';

}

The constructor, in this case, is simply being used to initialise the attributes of the object.

If you created a Car object then immediately displayed its attributes:

Car marks\_Car; // object created, no parameters so default constructor called, if one exists

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

then the output would be:

None, 0, 0

**NOTE** - You can't **explicitly** call the constructor. The constructor for an object is called automatically, just once when the object is created.

### Using Constructors with Parameters

The above default constructor with no parameters is great for initialising the object attributes to sensible values, but sometimes it is useful to initialise an object to specific values when it is created. In this case you can use **constructors with parameters** to pass values in to the constructor, which can be used to initialise some or all the object attributes. Here's an example for the Car class that initialises the speed attribute:

**// declaration in .h**

class Car

{

...

public:

Car(int speed);

...

};

**// definition in .cpp**

Car::Car(int speed)

{

m\_speed = speed;

m\_engine\_size = 0.0f;

m\_model[0] = 'N';

m\_model[1] = 'o';

m\_model[2] = 'n';

m\_model[3] = 'e';

m\_model[4] = '\0';

}

To initialise the object using this constructor you need to use the following syntax:

Car marks\_Car(55); // call constructor with int parameter, just like other method parameters

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

then the output would now be:

None, 0, 55

**NOTE** - Remember that all the class attribute names should by convention be prefixed with 'm\_'. This is shorthand for **'member attribute'** in the same way as 'p\_' is used to indicate a pointer variable. The main reason for this is to help avoid accidental naming conflicts when passing values in as parameters, whilst still enabling useful and relevant names to be given to both attributes and parameters. For instance, if the attribute was still called speed, then a common error for beginners is to use the same name as the parameter, e.g.:

Car::Car(int speed)

{

speed = speed; // compiles, but is meaningless, doesn't set speed attribute

...

}

Because speed is in the parameter list, it becomes a local variable name and would override access to the attribute of the same name. Adding the 'm\_' helps differentiate between the two variables.

### Overloading Constructors

It is perfectly valid for there to be several different constructors in a class, each of which can be used to initialise an object in a different way:

// declaration in .h

class Car

{

...

public:

Car(void);

Car(int speed);

Car(float engine\_size, int speed);

...

};

//-------------------------------------------------------

// definition in .cpp

Car::Car(float engine\_size, int speed) // this constructor has float and int parameters

{

m\_speed = speed;

m\_engine\_size = engine\_size;

m\_model[0] = 'N';

m\_model[1] = 'o';

m\_model[2] = 'n';

m\_model[3] = 'e';

m\_model[4] = '\0';

}

If a class has overloaded constructors you can use any of the constructors when initialising an object:

Car marks\_Car\_1; // call default constructor

Car marks\_Car\_2(55); // call constructor with int parameter

Car marks\_Car\_3(2.5f, 55); // call constructor with float and int parameter

The only rule when overloading constructors is that there cannot be two constructors with the same types in the same order in the parameter list, so you couldn't declare another constructor with a different implementation for this class with a float then an integer in the parameter list, e.g. Car(float x, int y); wouldn't be allowed.

**NOTE** - when using a class constructor with **no parameters**, i.e. a **default** constructor, you might be tempted to use the following syntax to create an object:

Car marks\_Car(); // Do not use, incorrect syntax

This **seems** logical as constructors with parameters look like normal function calls, and you need to specify brackets for a function call with no parameters. However, this is not true for default constructors with no parameters - to the compiler this looks like you are trying to declare a local function called marks\_Car(). The above line would compile just fine, but any later attempts to use the object will not, e.g.

marks\_Car.model[0] = 'B';

would give the following compiler error:

error C2228: left of '.model' must have class/struct/union

as an object hasn't been created. If you come across errors like this always check the constructor is being correctly called if it has no parameters, i.e.

Car marks\_Car; // correct: NO brackets for default constructor with no parameters

### Class Diagrams

When designing a class, you would normally create a **class diagram** before beginning the implementation of the class. Class diagrams form part of what is called the **Unified Modelling Language** (**UML**) that is used to visually design object-oriented systems. Basic class diagrams consist of three sections; the name, the attributes and the methods. Here is an example for the Car class:

**Car**

**-m\_speed: int**

**+m\_engine\_size: float**

**+m\_model: char[]**

**+Car(void)**

**+Car(speed: int)**

**+ accelerate(void): void**

**+ brake(void): void**

Class name

Attributes

Methods

- means private

+ means public

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

**- turnOnBrakeLight(void): void**

Notice the class diagram specifies the name of each variable **before** its type for both attributes and parameters, similarly the name of a method is specified before the return type. This is because UML is a generic system applicable to a wide range of programming environments, C++ being just one of these. Just remember to swap the elements around and use the correct syntax when it comes to actual coding in C++.

### Designing **Classes**

Classes are normally designed with the concepts of encapsulation and abstraction in mind. A good class design should hide both the implementation and the data representation behind an **interface**. This simply means that you would make all the attributes private to the class, and only give access to them via the public methods. Restricting access to the attributes via interface methods means that you can validate any changes to those attributes before updating them, as well as being able to change the implementation of a system without affecting users of the system.

A typical design of a class would declare all the attributes as private and then provide access methods to read from and write to these values. For instance, you could provide **Get** and **Set** methods, or other methods that can access and manipulate the data. A simple Employee class might be designed as follows:

**Employee**

**-m\_ID: int**

**-m\_HourlyWage:float**

**+Employee(void)**

**+SetID(ID: int):void**

**+GetID(void): int**

**+Employee(ID: int, HourlyWage: float)**

**+SetHourlyWage(HourlyWage: float): void**

**+GetHourlyWage(void): float**

**+ChangeHourlyWage(ChangeAmount: float): void**

0onsider what the approprite type should be for the parameter. u need to use the following syntax:

As you can see, both the attributes are private and inaccessible outside the class, only the access methods can be used to view or change them.

A simple demonstration of the use of encapsulation to validate changes to the data is to examine an implementation of the SetHourlyWage() method:

void Employee::SetHourlyWage(float HourlyWage)

{

if(HourlyWage < 0.0f) // wage can't be negative

{

m\_HourlyWage = 0.0f;

}

else

{

m\_HourlyWage = HourlyWage;

}

}

This example shows how encapsulation is an extremely useful property for a class - in the above case it's impossible to set the hourly wage to a negative value. A more sophisticated system would also tell the calling code that it was trying to do an invalid operation.

## Exercises

**To aid understanding of how the exercises work, use the debugger to step through all 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 01a

1. Create a new project, with a **main.cpp**, **Car.cpp** and **Car.h**.
2. Using the code from Tutorial 3 Exercise 01 as a basis, create a new Car class ensuring all attributes are prefixed with 'm\_'.
3. Add the three constructors described in the tutorial, and test that they work by creating several objects with each constructor, passing in different values where appropriate.
4. Add a display() method that displays all the attributes. Use the debugger and the display() method to check that the code is working as expected.
5. Add a new constructor that only initialises the engine size. Consider what the appropriate type should be for the parameter. Test this new constructor.

#### Exercise 01b

Expand the Car class to include more useful data appropriate to a car, e.g. add a top speed. For each added attribute make sure it is private, initialised in all the constructors, and create Get and Set methods.

#### Exercise 01c

For all the constructors and methods that set values in the class add some element of validation, e.g. don't allow speed to exceed top speed.

#### Exercise 02

1. Create a new project, with a **main.cpp**, **Employee.cpp** and **Employee.h**.
2. Implement the employee class defined in the class diagram in the tutorial. Create the constructors and access methods, assuming that neither the wage nor the ID can be less than zero.
3. Test the new class by creating objects with all the constructors and displaying the attributes with the Get methods, then setting all the attributes with the Set methods and redisplaying using the Get methods.

#### Exercise 03

1. Create a new project, and using the class diagram below create a Monster class for a game

**Monster**

**-m\_MonsterID: int**

**-m\_Health: float**

**+** **Monster(void)**

**+SetMonsterID(MonsterID: int): void**

**+GetMonsterID(void): int**

**+** **Monster (MonsterID: int, Health: float)**

**+SetHealth(Health: float): void**

**+GetHealth(void): float**

Don't forget to validate values being passed in to the constructors and Set methods.

1. Redesign the Monster class, using a new class diagram, to make the class more useful. For instance, add attributes such as speed, strength, etc.
2. Make sure you include constructors that initialise the new attributes, as well as the appropriate get and set methods.
3. Add methods called TakeDamage() and IsDead() to your design. Use appropriate parameters and return values in the class diagram.
4. Implement your design, and test that it works.

#### Exercise 04

1. Design a class, using a class diagram, that represents a weapon. The weapon must have an ID as well as attributes representing the weapons reload time (measured in seconds), minimum and maximum damage, and damage bonus.
2. All the attributes for the class should be set when the object is created, meaning they are set in the constructor. They should not be changeable after the object is created, so no Set methods, but will be accessible via Get methods.
3. The class also needs a method called display() which outputs the details of the weapon and a method called shoot() which will calculate and return an integer representing the damage from a single shot of the weapon. This is calculated as a random value between minimum and maximum damage, plus the bonus value.
4. Implement the design in a new project, and then test it. If you need to adjust the design during implementation due to unforeseen problems make sure you document the changes in your notebook.