Object-Oriented Programming  
Tutorial 06 - Dynamic Objects. Destructors. Composition and Aggregation

## Introduction

In this tutorial you will extend your understanding of classes and objects by considering how to **dynamically allocate objects**, as well as the use of dynamically created objects within structures, arrays and other objects. To this end a mechanism called a **destructor** will be introduced that allows the safe deletion of dynamically allocated memory used by objects. Finally, class **associations** will be discussed, notably **composition** and **aggregation**, which describe the relationship between a class and any classes it has as attributes.

## Dynamically Allocating and De-Allocating Objects

As has been discussed previously, it is often very useful to be able to allocate and de-allocate the memory for variables at run-time. This is achieved using the new and delete operators, which can be used to allocate memory for any data type. Therefore, as classes are data types, you can use these operators to dynamically create and destroy objects, using pointers to objects of that class to store the returned address of the newly allocated memory for the objects:

Car normal\_car; // normal Car object, memory reserved at compile time

Car \*dynamic\_car = NULL; // declare a pointer to a Car object

dynamic\_car = new Car; // dynamically create Car object (using default constructor)

Car \*init\_dynamic\_car = new Car(10); //dynamically **initialise** car object (using a

//different constructor)

Dynamic objects are accessed using the same pointer syntax used for other data types:

normal\_car.accelerate(); // dot operator for normal objects

dynamic\_car->accelerate(); // arrow operator for dynamic objects accessed via pointer

Also remember that, unlike compile time created objects, the programmer is responsible for explicitly de-allocating the memory for dynamically created ones once the object is no longer required, or before it goes out of scope:

delete dynamic\_car;

dynamic\_car = NULL; // good practice to set pointer to NULL after deallocating memory

**NOTE:** There are potential consequences of not correctly managing memory allocation and de-allocation, i.e. dangling pointers and memory leaks.

## Using Dynamically Created Objects in Structures and Arrays

You can declare member objects of a structure to be dynamically allocated by using a member variable that is a pointer to a class:

struct my\_struct\_pointer

{

Car \*m\_p\_A\_Car; // this member is a pointer, constructor is **NOT** called

Car m\_Another\_Car; // this member is an object, default constructor **IS** called

};

my\_struct\_pointer Test; // declare a structure called Test

Test.m\_p\_A\_Car = new Car; // dynamically allocate memory for member object. You d**on't** need to

// for m\_Another\_Car

It's important to realise that the member object remains **uninitialised** after the structure Test has been created, and the m\_p\_A\_Car member needs to be allocated using new. The same is true for arrays of objects, the array is declared using pointer to a class type, so you have an array of pointers to a class all of which will need to be allocated using new:

const int SIZE = 10;

Car \*car\_array[SIZE] = {0}; // array of pointers to Cars, clever use of init syntax to set all to NULL

for(int i = 0; i < SIZE; i++)

{

car\_array[i] = new Car;

}

Again it is the programmer's responsibility to de-allocate the dynamically created memory in structures and arrays once they are no longer needed:

delete Test.m\_p\_A\_Car; // delete dynamically allocated Car member. **Don't** need to delete **m\_Another\_Car**

Test.m\_p\_A\_Car = NULL; // good practice to set to NULL

for(int i = 0; i < SIZE; i++ )

{

delete car\_array[i]; // delete each element

car\_array[i] = NULL; // good practice to set to NULL

}

In this code **the array itself wasn't created dynamically** so the array delete [] operator isn't appropriate, the array will be automatically de-allocated when it goes out of scope just like any other non-pointer variable. It is only the **dynamically allocated array elements** themselves that require deletion.

## Dynamic Objects Within Objects

The previous use of dynamically created objects leads naturally to the use of dynamic objects that are declared inside other objects. In the simplest case this works in the same way as for structures:

// my\_simple\_class.h

class my\_simple\_class

{

public:

Car \*m\_p\_A\_Car; // this member is a pointer, constructor is **NOT** called

Car m\_Another\_Car; // this member is an object, default constructor **IS** called

};

// main.cpp

my\_simple\_class TestObject;

TestObject.m\_p\_A\_Car = new Car;

delete TestObject.m\_p\_A\_Car;

TestObject.m\_p\_A\_Car = NULL;

However, by using the special features provide by classes you can improve on this. Instead of relying on the person using the class to initialise the dynamic objects, you could use constructors to initialise them. In this way you can guarantee that the dynamic objects get memory allocated to them:

// my\_class.h

class my\_class

{

private:

Car \*m\_p\_A\_Car; // good practice to make attributes private (encapsulation)

Car m\_Another\_Car;

public:

my\_class(void); // constructor declaration

};

// my\_class.cpp

my\_class::my\_class(void) // constructor definition

{

m\_p\_A\_Car = new Car; // **Don't** need to dynamically allocate for **m\_Another\_Car**

}

// main.cpp

my\_class TestObject;

Now every time a my\_class object is created the constructor is called which automatically allocates memory for the dynamic object attribute.

However, there is still the question of de-allocating the memory before TestObject goes out of scope. You *could* write another method that calls delete on the Car attribute, e.g. TestObject.DeleteCar(), but this would still require the programmer to remember to call this method before the object goes out of scope. Fortunately there is another special feature of C++, called a **destructor**, that can help here.

## Destructors and Memory Management

A destructor can be thought of as the opposite to a constructor. It is a method that is **automatically called when an object is being deleted**, that is when a normal object goes out of scope or when delete is called on a dynamically created object. Destructors are used to help manage the process of deleting objects. This is especially useful when it comes to classes that have dynamically allocated attributes, though there are often other useful housekeeping duties that they can perform. The memory that a destructor can clear includes both dynamically allocated objects as well as any other type of dynamically allocated variable.

There can only be one destructor for any class, and it needs to be declared in a particular way. The destructor needs to have the same name as the class, but is preceded by a tilde '~' character. Like a constructor it has no return value but, unlike a constructor, it **cannot** have any parameters. The following code demonstrates the declaration and definition of a destructor for my\_class:

// my\_class.h

class my\_class

{

private:

Car \*m\_p\_A\_Car; // good practice to make attributes private (encapsulation)

Car m\_Another\_Car;

public:

my\_class(void); // constructor declaration

~my\_class(void); // destructor declaration

};

// my\_class.cpp

my\_class::~my\_class(void) // destructor definition

{

delete m\_p\_A\_Car; // **Don't** need to de-allocate **m\_Another\_Car**

m\_p\_A\_Car = NULL; // not strictly necessary, as object is about to be deleted

}

You cannot **explicitly** call destructors, the only time a destructor will be called is when an object is being deleted. As you can see, the destructor allows a class to free up memory allocated inside an object automatically, without any intervention required from the user of the class. Finally, although in this class it is an object that is being deleted, the same principle also applies to **any** dynamic data created for the class, e.g. a dynamically created array.

## Other Uses for Destructors

Destructors can be used for other 'housekeeping' purposes, in addition to the role of correctly de-allocating memory. For instance, you might want to keep track of how many objects of a class are currently active. To do this you might have an integer **static attribute** in a class called number\_of\_active\_objects. You could increment this amount by one in the constructor(s) to count the number of times an object is created, and then decrement the amount by one in the destructor whenever an object is destroyed.

## Class Associations - Composition and Aggregation

As you have seen classes may be associated with other classes, that is to say objects may contain other objects (or pointers to objects) as attributes, and may pass objects to their methods as parameters in just the same way as they use and pass other data types. There are two types of associations in object oriented programming - **composition** and **aggregation**.

### Composition

Composition is a **tight** **association** between parent and child objects. In a composite relationship the child object **has no existence outside of the relationship with the parent** and may not be referenced by other objects while it is referenced by the parent. The parent can be said to ‘**contain’** a child object and the child object is **'part of'** the parent. Also, when the parent object is destroyed the child object ceases to exist. Composition is indicated on class diagrams using a line between the two classes with a **solid diamond** shape representing the parent class, i.e. the class that contains the other class.

In the following diagram Class1 will contain Class2, in other words it will have Class2 as an attribute:

**Class1**

**Attributes**

**Methods()**

**Class2**

**Attributes**

**Methods()**

As an example, let's say that we are redesigning the Car class to include more information about the engine. In this case it might be appropriate to design an Engine class to store this information. Here's a partial class diagram that shows this:

**-m\_the\_engine: Engine**

**Methods()**

**Car**

**-m\_speed: int**

**...**

**-m\_cylinders: int**

**Methods()**

**Engine**

**-m\_size: float**

**...**

As you can see, with this relationship an Engine object is an integral part of a Car object. As such it should only be referenced by the Car object and be deleted when the Car is deleted. This would happen automatically in this case as the Engine has not been dynamically allocated. The association can clearly be seen as the Car **'contains’** an Engine and the Engine is **'part of'** the Car.

### Aggregation

Aggregation is a **loose association** between parent and child objects in which the parent object in the relationship simply contains a reference to the child. In an aggregate relationship the child object has an existence outside of the relationship with the parent and may be referenced by other objects. Also, when the parent object is destroyed the child object is not. This type of association can be described as the parent object **'refers to'** the childor the child is **'linked to'** the parent.

Aggregation is represented using an unfilled diamond as below:

**Class1**

**Attributes**

**Methods()**

**Class2**

**Attributes**

**Methods()**

Class1 will still contain an attribute of type Class2, but the relationship is different from composition as the object represented by Class2 can exist independently of Class1, and can also be referenced by other classes. For instance, a Person object clearly exists independently of a Car object, so it is sensible to link a Car to a Person using aggregation. A Car can contain a Person but a Person can also exist without a car:

**-m\_p\_driver: Person\***

**Methods()**

**Car**

**-m\_speed: int**

**...**

**-m\_height: float**

**Methods()**

**Person**

**-m\_weight: float**

**...**

In this example the Car **'refers to'** a driver, a Person object, which also implies the driver can be changed, or in fact there could be no driver.

The use of aggregation implies the use of a **pointer to the child object**, as it wouldn't make sense to copy the Person into the Car data structure, just to provide a reference to an independent entity.

A Person could also be a member of a Car drivers’ club, a **'linked to'** relationship, so would be associated with both classes via aggregation:

**Car**

**Club**

**Person**

The Person can get out of the Car and/or leave the Club, and they will still exist. Then they, or another Person, could be added as driver to the Car again. If the Car is crushed or the Club disbanded then, again, the Person will continue to exist independently of these events, i.e. they would still exist even if the Car or Club is deleted.

## Using Composition and Aggregation

With a relationship between classes based on composition you will generally find that the child object is created and managed by the parent class. So, for the Car and Engine example, the Engine could be declared using a normal non-pointer member attribute like so:

// Engine.h

class Engine

{

private:

float m\_size;

... etc

};

// Car.h

class Car

{

private:

Engine m\_the\_engine; // composition using normal member attribute

... etc

};

Here the Engine attribute contained in Car will be initialised with the Engine default constructor, when the Car object is destroyed it will be automatically deleted. Alternatively, you might need to use dynamic memory allocation in which case you would declare the attribute as a pointer, allocate the memory in the constructor(s), and clear the memory in the destructor:

// Car.h

class Car

{

private:

Engine \*m\_p\_the\_engine; // composition using pointer variable

... etc

};

// Car.cpp

Car::Car(void)

{

m\_p\_the\_engine = new Engine;

}

Car::~Car(void)

{

delete m\_p\_the\_engine;

m\_p\_the\_engine = NULL;

}

When the relationship is based on aggregation then the child object is created separately and assigned to the parent object via a pointer. For the Person and Car classes it would be more appropriate to use a pointer representation as this can be used to point at different Person objects as required:

// Car.h

class Car

{

private:

Person \*m\_p\_driver; // aggregation using pointer variable

... etc

};

The class would then have methods to add and remove a driver from the relationship, e.g.

// Car.cpp

void Car::set\_driver(Person \*driver)

{

m\_p\_driver = driver;

}

void Car::remove\_driver(void)

{

m\_p\_driver = NULL;

}

// main.cpp

Car my\_car;

Person mark;

my\_car.set\_driver(&mark);

...

my\_car.remove\_driver();

## 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 01

1. Create a new project, with the **Car.cpp** and **Car.h** copied from Tutorial 03 Exercise 01.
2. Create a structure, named how you like, in **main.cpp** that has two members - a Car object and a pointer to a Car object.
3. Create a variable using this structure in main().
4. Use new to dynamically allocate memory to the pointer member of the structure.
5. Put breakpoints just before and just after the memory allocation and examine the structure. Write in your notebook the results and explain why the structure members have different values at these different times.
6. Put a breakpoint in the Car constructor and step through the code. Make note in your logbook of when the constructor is called, and explain why the constructor is called at that time.
7. Create a five element array of the structure. Consider how you would initialise the Car pointers for every element, and implement.
8. Delete all dynamically allocated memory from the structure and the array. Ensure you set all the pointers to NULL after deletion.
9. Examine the contents of the structure and the array just after the deletion but before the pointers are set to NULL. What does this tell you about the validity of the pointer after delete is called?

#### Exercise 02

1. Create a new project with the **Car.cpp** and **Car.h** copied from Exercise 01.
2. Create an array of Car pointers, that has a number of elements equal to the number of constructors you have in your Car class.
3. Dynamically create every element with a different constructor.
4. Step through the code noting how the elements of the array change after each allocation.
5. Don't forget to delete every element before the program finishes.

#### Exercise 03

1. Create a new project with the **Car.cpp** and **Car.h** copied from Exercise 01.
2. Create a new class that represents a Garage in new **.cpp** and **.h** files.
3. The Garage class should have two public Car pointer objects, called whatever you wish.
4. Create a Garage object in main(), and dynamically allocate the memory for the two cars.
5. Use the object access operators '.' and '->' to correctly access the Car object methods in order to change some of the values contained in the objects.
6. Output all the attributes of the Car objects.
7. De-allocate the Car objects from memory before the program exits.

#### Exercise 04

1. Create a new project with the classes copied from Exercise 03.
2. Update the Garage class to use the default constructor to initialise the Car objects.
3. Add a destructor to the Garage class to de-allocate the memory.
4. Put breakpoints in the Garage constructor and destructor.
5. Create a **global** Garage object and a **local** Garage object in main().
6. Create a local Garage **pointer** object and initialise it using new in main().
7. **D**elete the dynamically created Garage object before the end of main().
8. Run the code and find out using breakpoints where the constructors and destructors are called for each object. Explain why you think they get called in the order they do.

#### Exercise 05

1. Create a new project with the classes copied from Exercise 04.
2. Update the Car destructor so that the static variable m\_number\_of\_cars gets decremented every time an object is deleted.
3. Add a similar static variable to the Garage class, along with increment and decrement code in the constructor and destructor.
4. Initialise the Garage static counter in the same way as the Car class counter.
5. Create several Cars, Garages, structures of both, and arrays of both, using both normal objects and pointers with dynamic allocation and de-allocation.
6. Add several basic functions that get called from main(). Each function should have local variables of all the various types mentioned in the previous step.
7. Monitor the counter variables as the program runs by stepping through one line at a time, and seeing how they change over time. Check that they correctly track the number of objects created - at the very least the variables should end up as zero by the time the program ends.

#### Exercise 06

1. Create a new project, with the **Car.cpp** and **Car.h** copied from Exercise 01.
2. Create a class diagram for an Engine class, using the design in the tutorial as a basis. Add several more meaningful attributes, along with constructors, and Get() and Set() methods. Also add a static class object instance counter, the same as those in exercise 5, along with updates in the constructors and destructor.
3. Use the design to create code for the Engine class in **.cpp** and **.h** files.
4. Remove the existing engine size variable from Car, and add an Engine. Choose between adding the Engine as a normal attribute, or a pointer to an Engine. Explain why you think your choice is best.
5. If you decide on using a pointer make sure you correctly allocate and de-allocate memory for the Engine in the Car constructors and destructor.
6. Initialise the Engine attributes as needed, either in the constructors or with other methods.
7. Add code to the Car display() method to output the Engine attributes.
8. Test the new Car and Engine combination by creating a few Cars in various ways, updating the Engine values and outputting them. Check that the Cars and Engines both get destroyed together by monitoring the static counter attributes for each class as you step through the program.

#### Exercise 07

1. Create a new project, with the **Car.cpp** and **Car.h** copied from Exercise 01.
2. Create class diagrams for basic Person and Club classes so a Person can be a driver of a Car and a member of a Club in aggregate relationships, as mentioned in the tutorial. You might want to consider that a Club can have more than one member.
3. Implement the designs.
4. Update the Car class to contain a private Person pointer attribute, along with methods to add and remove a driver.
5. Test the design work by creating several dynamic Car, Person and Club objects, and setting the Person objects to be Car drivers and Club members. Add and remove Person objects from Cars and Clubs, and delete Cars and Clubs to demonstrate the Person objects are independent from the other objects.

#### Exercise 08

Think of one example each of an aggregate and composite relationship in a real world example and draw the class diagrams showing these relationships. Implement and test the classes if you can.