

OBJECT CONSTRUCTION AND DESTRUCTION

In this chapter, the lifetime of an object will be discussed i.e. looking at when objects get created and when they get destroyed. This will re-introduce two special functions in C++ called *constructors* and *destructors* which can be used to provide automatic initialisation of an object when it is created and automatic tidying-up when an object is destroyed.

The other major topic is the simple free-store management using the *new* and *delete* operators which are similar (but not identical) to the functions *malloc ()* and *free ()*.

1. REVIEW - SCOPE AND STORAGE CLASS

The concepts of scope and storage class are very important. In C++, there are three different storage classes: static, automatic and dynamic (which behave in exactly the same way as in C). A recap of the first two are shown below :

	Storage Class	Scope
MyClass GlobalVal;	static	global
static int FileScopeVal;	static	file
int Func (MyClass ArgVal) {	auto	function
static int Max = 20;	static	function
MyClass LocalVal;	auto	function
for (int i = 0; i < Max; i++) {	auto	function
int VLocalVal = 36;	auto	block
....		
}		
}		

Variables with *static* live to the end of the program, whilst variables with *automatic* storage are destroyed at the end of the enclosing block. The scope of an object is that part of the program where the object is visible.

If an object has *global* scope, it can be accessed in any file in the program. It must be defined globally in one file and can be referenced in any other file using an *extern* declaration as follows:

// FILE1.CPP

int GlobalVar;

// FILE2.CPP

extern int GlobalVal;

If necessary, an object can be defined global to a single file, thereby preventing access from

another file. This may be achieved by declaring the object with the *static* keyword.

2. FREE-STORE OBJECT

Dynamic storage is also known as “free-store”. Allocating objects from the free-store is similar to allocating objects from the heap in C. In C, the standard functions *malloc ()* and *free ()* are used to allocate and de-allocate memory dynamically. Although these functions are still available in C++, the preferred way of allocating and de-allocating objects dynamically is with the *new* and *delete* operators.

The *new* operator takes a single operand, specifying the type of object required. The compiler then tries to allocate sufficient memory (in a contiguous block) to hold the required type of object. If sufficient memory is available in the free-store, operator *new* returns the starting address of the block.

```
int *ptr_int = new int;           // Explicitly allocate an object which is
                                  // unnamed and un-initialised

*ptr_int = 0;                    // Manipulate through pointers

delete ptr_int;                  // Explicitly de-allocate
```

Care must be taken to preserve the value of the pointer since it is the only access to the newly created object. If this pointer is accidentally corrupted, all access to the object it points to is lost.

When the object is no longer required, operator *delete* is used to free up the memory used for the storage of the object. It is good practice to delete an object as soon as it is no longer required, so that the memory can be used elsewhere in the program. Operator *delete* is similar to the C function *free ()*.

If there is insufficient memory, a zero pointer is returned instead, similar to the way *malloc ()* works in C. The client program should check for this situation.

```
int *ptr_int = new int;

if ( ptr_int == 0 )
    cout << “Insufficient memory”;
else
    // Memory allocated successfully- so use object as needed.

delete ptr_int;
```

NOTE: It is not an error to supply a value of 0 to operator *delete*. It would simply have no effect.

3. OPERATORS *new* AND *delete*

Operator *new* can be used to allocate built-in data types such as *int* , *double*, etc but it can also be used to allocate whole “class objects”. It is also possible to allocate *arrays of objects*. The number of elements can be a constant or a run-time expression (such as *size* in the example below :

```
double *ptr_double = new double;      // Allocate double object
Account *ptr_class = new Account;     // Allocate class Account object
long *ptr_table = new long [size];    // Allocate array of long objects
                                        // size is run time expression
```

NOTE : Operator *new* returns a correctly-typed pointer to the new object. The compiler will complain if an incorrect pointer is used to store the address.

```
long *ptr_long = new int;             // Incorrect pointer type
```

The use of *new* is easier to read than the equivalent operation in C using the *malloc ()*.

Another reason for using *new* and *delete ()* rather than *malloc ()* and *free ()* is that they provide full object construction and destruction.

There is also a special notation for deleting an array of objects.

```
delete ptr_double;
delete ptr_class;
delete [ ] ptr_table;
```

4. OBJECTS LIFETIMES AND INITIALISATION

The three different storage classes available in C++ (*static*, *automatic* and *dynamic*) behave as in C.

Statically-allocated objects i.e. global and local *statics*, are active throughout the entire program and are placed in an area of memory holding zeroed data. It is important to remember that local *statics* really are local. They can only be accessed by name from within the block in which they are defined.

Automatic objects are always allocated on the runtime stack. They exist from the point of definition to the end of the block and their initial value is garbage, by default.

Objects allocated using *new* will exist until *deleted* and are located on the free-store. By default, dynamically-created objects have an unknown initial value, just like automatic objects.

These issues of lifetimes and initial values are closely related in C++ because it is possible to provide a special member function called a *constructor*, whose role in life is to initialise an object when it is first created. Constructors are a very important part in C++.

In a similar vein it is possible to provide a special member function called a *destructor*, which can be used to perform any shutdown operations upon an object just before it dies.

5. INITIALISATION CONTROL

A constructor is a special member function belonging to a class. It is easily recognisable because it has the same name as the class :

```
class Point                // Point class definition
{
public:

    Point();                // Constructor for Point class
    // Other member functions

private:
    int x,
        y;
};
```

Constructors are not allowed to specify a return type, not even void. The reasons for this are technical - most C++ compilers already return an internal value from a constructor that is invisible to the programmer.

The definition of the constructor is similar to other member functions, except of course that there is no return type specified. Thus in the following example, the first ‘*Point*’ tells the compiler that this is a member function of the *Point* class, and the second ‘*Point*’ is the name of the function itself. Since the name of the function is the same as the name of the class, this must be the constructor for the *Point* class :

```
Point :: Point ( )
{
    x = 0;
    y = 0;
}
```

6. PROVIDING CONSTRUCTORS

Constructors are allowed to take arguments, just like any other function in C++. It is also possible to provide overloaded constructors; in fact, it is often useful to supply maybe two or three different constructors for a class so that objects can be constructed in a variety of different ways.

Each constructor specified in a class definition must have a unique signature. In other words, the number of parameters or the data types of the parameters in each constructor must be distinguishable by the compiler. A constructor that takes no arguments is called a *default constructor*.

As pointed out earlier, a constructor is automatically invoked when an object is created. Depending on its signature, the appropriate constructor is picked by the compiler when it sees the object definition. The object definition needs to specify the arguments to be passed into the required constructor.

```

class Point
{
public:
    Point ();
    Point ( int x0, int y0 );

private:
    int x,
        y;
};

```

```

Point :: Point ()
{
    x = 0;
    y = 0;
}

Point :: Point ( int x0, int y0 )
{
    x = x0;
    y = y0;
}

```

```

Point origin; //Automatically calls Point constructor
              //Point :: Point (). No parameters needed

Point Top_Right ( 640, 480 ); //Automatically calls Point constructor
                              //Point :: Point ( int x0, int y0 ), passing
                              //the value 640 into x0 and 480 into y0

```

Since the default constructor *Point :: Point ()* has no arguments, all it can do is perform some kind of default initialisation.

NOTE :

1. The use of default arguments is encouraged. The two *Point* constructors above can be replaced by a single constructor taking default arguments :

```

Point ( int x0 = 0, int y0 = 0 ); // Constructor prototype.

Point :: Point ( int x0, int y0 ) // Constructor implementation
{
    x = x0;
    y = y0 ;
}

```

- Constructors should be limited to straightforward data initialisation if possible. Any processing that might fail (e.g. opening a file, etc) should be placed in a separate function. This is due to the fact that constructors are incapable of returning a value to indicate if the operation succeeded or not.

7. TIMING OF CONSTRUCTORS

It is important to know the exact time in a program when a constructor is invoked. The constructor is called when the object is created, and the timing depends on the storage class of the object.

```
Point Top_Right ( 640, 480 );      // Global statics - constructor called
                                   // before main ( )

void Func ( void)
{
    static Point Border ( 600, 400 ); // Local static - constructor called before
                                       // first call

    Point Graph_Origin ;           // Automatic objects - constructor called
                                   // at point of definition

                                   // Dynamic objects - constructor called
                                   // at point of creation

    Point *Ptr_Mouse = new Point (320, 240);

    Point *Ptr_Array = new Point [20];
}
```

The object called *Top_Right* is defined as a *global* variable and therefore has *static* storage class by default. This object will have its constructor called right at the beginning of the program, even before the start of the *main ()* function. The object has to be initialised this early in case it is used immediately inside *main ()* :

```
Point Top_Right ( 640, 480 );

void main ( )
{
    Top_Right.Move (-10, -10 );
    .....
}
```

The second object *Border* is local to *Func ()* function. This object can only be referenced inside *Func ()*, and therefore does not need to be initialised until the function is first invoked. Since *Border* has been declared static, the object will only need to be constructed once. Therefore the object will remain permanently allocated in memory until the program terminates.

The local object *Graph_Origin* will be constructed each time the function *Func ()* is called. The constructor will be invoked at the exact point where the object is defined, and this is quite useful

because the situation often arises where the function has to do some preliminary work before it is ready to create the object.

The free-store object pointed to by *Ptr_Mouse* is constructed at the point of the *new* statement. Operator *new* allocates the memory for the object; providing enough memory is available, the *Point* constructor is then invoked with the parameters 320 and 240 to initialise the object.

The second free-store object deals with dynamic arrays. Arrays are used much less in C++ than you might expect as many developers tend to use specialised *collection classes* such as *List* and *Map*.

NOTE : C++ only allows a *default constructor* to be called to initialise dynamic arrays. Operator *new* allocates enough memory for 20 *Point* objects, and the default constructor is then invoked for each object.

8. CONSTRUCTORS AS CONVERTERS

As well as being seen as an “initialisation” function, a constructor can also be thought of as a “conversion” function, which can be used like a cast operation in C.

```
class Money
{
public:
    Money (double Amount );
private:
    int dollars,
        cents;
};
```

```
Money Cash ( 0.0);

Cash = ( Money ) 10.50;    // Cast style conversion

Cash = Money ( 10.50 );    // Function style conversion

Cash = 10.50;              // Implicit conversion (preferred)
```

In the example above, the *Money* constructor takes a *double* argument and presumably uses it to initialise the *dollars* and *cents* data members when a *Money* object is created.

When the *Cash* object is created, the constructor will be called to initialise the object with the value *0.0*. This is the traditional meaning of a constructor where the constructor initialises an object when it is first created.

The statement :

```
Cash = (Money) 10.50;
```

looks like a cast conversion; in fact that is exactly what is happening. The value *10.50* is

converted into a *Money* object, which is then assigned to the *Cash* object. The exact sequence of events is as follows :

1. The *Money* constructor is called with the *double* value *10.50* as a parameter. The compiler checks the *Money* class definition to ensure the the class has a constructor with a double argument.
2. The constructor fabricates a temporary *Money* object with the values (*dollars* = 10, *cents* = 50) and returns this object. This can be seen as “converting *10.50* into a *Money* object”.
3. The temporary *Money* object is assigned to the *Cash* object, thereby overwriting the previous contents of *Cash*. The temporary *Money* object will be destroyed by the compiler at some point before the end of the function (the exact time is not strictly defined in the C++ language).

The second “conversion” statement

Cash = Money (10.50);

has exactly the same effect. It looks more like a function call which of course is exactly what is happening; *Money :: Money (double)* is called to convert *10.50* to a *Money* object.

Finally in the last “conversion” statement

Cash = 10.50;

the value *10.50* is implicitly converted into a *Money* object. This conversion occurs automatically when the compiler realises that *10.50* cannot be assigned directly to *Cash*; the value must first be converted into a *Money* object to preserve type consistency in the statement. This is the preferred approach since it avoids unnecessary clutter - the compiler makes the necessary conversion without the programmers’ intervention.

9. THE DEATH OF AN OBJECT

A destructor is a special member function in a class and can be thought of as the opposite of a constructor. Whereas a constructor is called to initialise an object as it is created, a destructor is called just as an object is about to die to perform any tidying up operations that might be appropriate before the object disappears.

The exact timing of the destructor call depends on the storage class of the object. The destructor is called automatically by the compiler; thus the programmer does not have to worry about calling the destructor. This *automatic* initialisation and tidying up objects is one of the nice features that makes C++ a more robust language than C.

10. PROVIDING A DESTRUCTOR

Destructors must have the same name as the class, with a leading tilda (~) at the front. Destructors are not allowed to specify a return value, nor are they allowed to receive any parameters. Thus it is not possible to have more than one destructor in a class (the compiler only allows function overloading if the functions have unique argument lists).

```
class String
{
    public
        String ( char *);    // Constructor function
        ~String ();        // Destructor function

    private:
        char *Text;

};
```

It may not be necessary to have a destructor in every class, but most classes will need one. If a class has no destructor, this simply means that there is no specific tidying up operation to perform when the object is destroyed. The object is de-allocated in the usual way, whether the class has a destructor or not.

Some companies have C++ coding standards which insist that every class has a destructor, even if there is nothing to do. The absence of a destructor might be misconstrued by some class users as being an oversight. The presence of a destructor, even an empty destructor, assures the class user that destruction operations have been fully taken into account.

There are no simple rules about what is required in a destructor, because each class is different and has its own requirements. One common situation is where an object has allocated some secondary storage during its lifetime. In these cases, it is usually desirable to make sure this extra memory is de-allocated before the object dies. The destructor is an ideal place to perform this de-allocation but it is not responsible for de-allocating the memory of the object; it de-initialises the object prior to the normal de-allocation of the object's storage when it dies.

The *String* class shown above is one such example; the constructor allocates a buffer to hold a copy of some string of text and the destructor has to make sure that this buffer is deleted before the *String* object dies. Otherwise the buffer would never be deleted and its memory would never be released.

Consider the following *Field* class which might be useful in data entry programs such as those used by mortgage brokers, where a large amount of data is entered into forms on the screen :

```

class Field
{
public:
    Field (int Length = 80);    //Default constructor
    Field ( char *Def);        //Additional constructor
    ~Field ();                 //Destructor

    void Display ();

private:
    char *Text;
    int Length;
};

```

The implementation of these constructors could be :

```

Field::Field ( int Field_Length)
{
    Length = Field_Length;
    Text = new char [ Length + 1 ];    //Allocate buffer for input

    memset ( Text, ' ', Length);      //Fill field with spaces and
    Text[ Length ] = '\0';            //null terminate

    Display ();                        //Display field on screen
}

Field::Field ( char *Def )
{
    Length = strlen ( Def );
    Text = new char [ Length + 1 ];    //Allocate buffer for input
    strcpy ( Text, Def );              //Copy in default reply

    Display ();                        //Display on screen
}

```

When a *Field* object is destroyed, the destructor will be automatically called to tidy up the object. One of the tasks for the destructor here is to free the secondary storage used to hold the text entered by the user. The destructor also clears the field from the screen :

```

Field::~Field ( )
{
    memset ( Text, ' ', Length ); //Fill field with spaces and
    Text [ Length ] = '\0';      //null terminate

    Display ( );                //Display blank field on screen

    delete [ ] Text;           //Release buffer char array
}

```

11. TIMING OF DESTRUCTORS

The following example highlights the exact moment in a program when a destructor is invoked upon an object. The destructor is called when the object is about to die, and the timing depends on the storage class of the object.

```

String Err_Mess ("\\n Invalid name ");

void Func ( )
{
    static String Prompt ("\\n Name ");

    String Title ("Mr.");

    String *Ptr_String = new String ("C++");

    ::::::::::::::::::::

    delete Ptr_String;

}

```

Err_Mess is defined globally and therefore has static storage by default. This variable will have its destructor called at the end of the program (even after the *main ()* function has terminated).

Prompt is defined local to the *Func ()* function, and has also been defined with static storage space. Therefore just like a global variable, *Prompt* will be destroyed at the end of the program.

Title is defined with automatic storage space, local to the function. It will be destroyed when it goes out of scope at the end of the enclosing block i.e. at the end of the function *Func ()*.

The free-store object pointed to by *Ptr_String* is de-allocated in the delete statement and the destructor is invoked at this point. When dealing with dynamically-created objects, care must be taken to ensure that the object is deleted when the object is no longer needed. Otherwise the memory used to hold the object will never be released for some other program to use.

Consider the following example :

```

void Func ( )
{

String *Ptr_String = new String ("Text");
    .....
    .....
}

```

The *String* object is dynamically created at the start of the function, but the object is never *deleted*. Unfortunately, the local pointer *Ptr_String* goes out of scope and is automatically destroyed. The object it pointed to, however, is not destroyed (because it must be explicitly *deleted*). Once, the program leaves the *Func ()* function, the object address is lost and therefore all access to it is lost and thus it can never be deleted. This is often referred to as a *memory leak*.

In the concept of class association, it is possible to dynamically create an object in one function and destroy it in another function.

12. DYNAMIC ARRAYS AND DESTRUCTORS

There is a special notation when deleting an array of objects.

```

Money *Pay_Roll = new Money [Num_Of_Staff];
    ....
    ....
    ....
delete [ ] Pay_Roll;

```

In this example, an array of *Money* objects (*Num_Of_Staff* of them) is allocated and the address of the first object is held in the pointer *Pay_Roll*.

When the time comes to destroy the array, the delete operator must clearly be used to free up the memory. But how many times must the *Money* destructor be called ? It is reasonable to expect the destructor to be called *Num_Of_Staff* times, once for each element of the array. To ensure that this happens, the following notation must be used:

```

delete [ ] Pay_Roll;

```

This reminds the compiler that *Pay_Roll* points to an array of *Money* objects, not just the single *Money* object. The compiler remembers internally how many elements are in the array, so that it knows how many times the *Money* destructor needs to be called.

13. SUMMARY

In this section, it has been seen that objects in C++ obey certain rules, depending on where and how they are defined. There are three categories of storage : *static*, *automatic* and *dynamic*. The storage class of an object determines when the object is created and when it is destroyed.

Dynamic objects are created and destroyed using the *new* and *delete* operators.

Whatever the storage class of an object, its initialisation and shutdown can be controlled through the provision of constructors and destructors.

Most classes have at least one constructor to guarantee that an object is initialised when it is created. Any number of overloaded constructors may be provided for a class, so that the class implementor can supply a number of alternative ways to create an object.

A class can only have a single destructor, and it is considered good programming practice to provide one for each class, even if there is no tidying-up operation for the object when it dies. The destructor is called automatically when an object is in its death throes, just before the object's memory is de-allocated.