



Module 818

Encapsulation of Objects and Methods in C++

Aim

After working through this module you should be able to create and use objects in C++, incorporating encapsulation and information hiding

Learning objectives

After working through this module you should be able to:

1. Identify the advantages of using encapsulated objects in C++
2. List the advantages of using information hiding in C++
3. Describe the process of object-oriented messaging in C++
4. Define and use constructors and destructors in C++
5. List the correct methods for using classes for object packaging in C++
6. Define and use abstract data types in C++
7. Use objects within C++ programs
8. List the advantages of using operator overloading in C++

Content

Definition of objects in C++

Encapsulation of data and methods

Constructors and destructors

Abstract data types

Operator overloading

Learning Strategy

Read the printed module and the assigned readings and complete the exercises as requested.

Assessment

Completion of exercises and the CML test at the end of the module.

Objective 1 After working through this module you should be able to identify the advantages of using encapsulated objects in C++

Encapsulation is the basic process of forming objects. An encapsulated object is often called an abstract data type. Without encapsulation, which involves the use of one or more classes, there is really no basis for object-oriented programming.

Why use encapsulation?

When we fully encapsulate code, we actually build an impenetrable wall to protect the contained code from accidental corruption due to the silly little errors that we are all prone to make. We also tend to isolate errors to small sections of code to make them easier to find and fix. We will have a lot more to say about the benefits of encapsulation as we progress through this Module.

Encapsulation is the basic method for producing *information hiding* in object-oriented methodology. The following program [OPEN.CPP] is a 'conventional' program that incorporates *no* information hiding, largely because of the use of global identifiers.

```
#include <iostream.h>
struct one_datum {
int data_store;
};
void main()
{
one_datum dog1, dog2, dog3;
int piggy;
dog1.data_store = 12;
dog2.data_store = 17;
dog3.data_store = -13;
piggy = 123;
cout << "The value of dog1 is " << dog1.data_store << "\n";
cout << "The value of dog2 is " << dog2.data_store << "\n";
cout << "The value of dog3 is " << dog3.data_store << "\n";
cout << "The value of piggy is " << piggy << "\n";
}
```

A very simple structure is defined in lines 2 to 4 which contains a single int type variable within the structure. Three variables are declared in line 7, each of which contains a single int type variable, and each of the three variables are available anywhere within the main function. Each variable can be assigned, incremented, read, modified, or have any number of operations performed on it; a few of these operations are illustrated in lines 9 to 16.

Exercise 1

Describe in simple terms the process of encapsulation.

Objective 2 After working through this section you should be able to list the advantages of using information hiding in C++

The next program [CLAS.CPP] has limited information hiding. This program is identical to the last one except for the way it does some of its operations. We will take the differences one at a time and explain what is happening. Keep in mind that this is a trivial program and the safeguards built into it are not needed for such a simple program but are used here to illustrate how to use these techniques in a larger and more complicated program.

```
#include <iostream.h>
class one_datum {
    int data_store;
public:
    void set(int in_value);
    int get_value(void);
};
void one_datum::set(int in_value)
{
    data_store = in_value;
}
int one_datum::get_value(void)
{
    return data_store;
}
void main()
{
    one_datum dog1, dog2, dog3;
    int piggy;
    dog1.set(12);
    dog2.set(17);
    dog3.set(-13);
    piggy = 123;
    // dog1.data_store = 115;          This is illegal in C++
    // dog2.data_store = 211;          This is illegal in C++
    cout << "The value of dog1 is " << dog1.get_value() << "\n";
    cout << "The value of dog2 is " << dog2.get_value() << "\n";
    cout << "The value of dog3 is " << dog3.get_value() << "\n";
    cout << "The value of piggy is " << piggy << "\n";
}
```

The first difference is that we have the definition of a *class* instead of a structure beginning in line 2. The only difference between a class and a structure is that a class begins with a private section whereas a structure has no private section automatically defined. The keyword *class* is used to declare a class.

The class named *one_datum* is composed of the single variable named *data_store* and two functions, one named *set()* and the other named *get_value()*. A more complete definition of a class is a group of variables and one or more functions that can operate on that data.

The PRIVATE section

A private section of a class is a section of data which cannot be accessed outside of the class; it is hidden from any outside access. Thus

the variable named `data_store`, which is a part of the object (an object will be defined completely later) named `dog1` declared in line 18, is not available for use anywhere in the main function. It is as if we have built a 'brick wall' around the variables to protect them from accidental corruption by outside programming influences. It seems a little strange to declare a variable in the main program that we cannot use, but that is exactly what we did.

The PUBLIC section

A new keyword, *public*, is introduced in line 4 which states that anything following this keyword can be accessed from outside of this class. Because the two functions are defined following the keyword *public*, they are both public and available for use in the calling function or any other function that is within the scope of the calling function. This opens two small peepholes in the solid wall of protection. You should keep in mind that the private variable is not available to the calling program. Thus, we can only use the variable by calling one of the two functions defined as a part of the class. These are called member functions because they are members of the class.

Since we have declared two functions, we need to define them by saying what each function will actually do. This is done in lines 8 to 15 where they are each defined in the normal way, except that the class name is prepended onto the function name and separated from it by a double colon. These two function definitions are called the implementation of the functions. The class name is required because we can use the same function name in other classes and the compiler must know with which class to associate each function implementation.

One of the key points to be made here is that the private data contained within the class is available within the implementation of the member functions of the class for modification or reading in the normal manner. You can do anything with the private data within the function implementations which are a part of that class, but the private data of other classes is hidden and not available within the member functions of this class. This is the reason we must prepend the class name to the function names of this class when defining them.

Note that it is legal to include variables and functions in the private part and additional variables and functions in the public part. In most practical situations, variables are included in only the private part and functions are included in only the public part of a class definition. Occasionally, variables or functions are used in the other part. This sometimes leads to a very practical solution to a particular problem, but in general, the entities are used only in the places mentioned.

In C++ we have three scopes of variables: local, file and class. Local variables are localised to a single function and file variables are available anywhere in a file following their definition. A variable with class scope is available anywhere within the scope of a class and nowhere else.

To clarify the use of the various OOP terms we have introduced, Table 1 presents definitions of the four terms we have used so far in this Module, together with their use in C++.

Class	A grouping of data and methods (functions). A class is similar to a type in ANSI-C.
Object	An instance of a class, similar to a variable defined as an instance of a type.
Method	A function contained within the class, designed to operate on the data within the class.
Message	Equivalent to a function call in ANSI-C. In object-oriented programming, we send messages instead of calling functions.

Table 1: OOP Terminology in C++

Note that lines 5 and 6 of this program are actually the prototypes for the two methods within a class. The method named set requires one parameter of type int and returns nothing, so the return type is void. The method named get_value () has no input parameters but returns an int type value to the caller.

Exercise 2

Define the difference between a *class* and an *object*.

Modify CLAS.CPP by adding a method to calculate the square of the stored value. Amend the main program to read and display the resulting squared values.

Add a constructor to CLAS.CPP to initialise the stored value to 10, and amend the main program to display the values immediately after the object definition.

Objective 3 After working through this section you should be able to describe the process of object-oriented messaging in C++

How do we use the class `one_datum` that is defined in `CLAS.CPP`? The class is made up of one variable and two methods. The methods operate on the variable contained in the class when they receive messages to do so. In line 18 we declare three objects of the class `one_datum` and name the objects `dog1`, `dog2`, and `dog3`. Each object contains a single data point which we can set through use of one method, or read its value through use of the other method, but we cannot directly set or read the value of the data point because it is hidden within the "block wall" around the class. In line 21, we send a message to the object named `dog1` instructing it to set its internal value to 12, and even though this looks like a function call, it is properly called sending a message to a method. Remember that the object named `dog1` has a method associated with it called `set()` that sets its internal value to the actual parameter included within the message. You will notice that the form is very much like the means of accessing the elements of a structure. You mention the name of the object with a dot connecting it to the name of the method. In a similar manner, we send a message to each of the other two objects `dog2` and `dog3` to set their values to those indicated.

Lines 24 and 25 have been commented out because the operations are illegal because the variable named `data_store` is private and not available to the code outside of the object itself. It should be clear that the data contained within the object named `dog1` is not available within the methods of `dog2` or `dog3` because they are different objects.

The other method defined for each object is used in lines 26 to 29 to illustrate how it can be used. In each case, another message is sent to each object and the returned result is output to the monitor via the stream library.

Using a normal variable

There is another variable named `piggy` declared and used throughout this program that illustrates that a standard C variable can be intermixed with the objects and used in the normal manner. It would be a good exercise for you to remove the comments from lines 24 and 25 to see what kind of error message your compiler issues.

Problems

The next program [`OPENPOLE.CPP`] has a few serious problems that will be overcome in the next example program by using the principles of encapsulation.

```
#include <iostream.h>
int area(int rec_height, int rec_width);
struct rectangle {
    int height;
    int width;
};
struct pole {
    int length;
    int depth;
};
int area(int rec_height, int rec_width)     //Area of a rectangle
{
    return rec_height * rec_width;
}
void main()
{
    rectangle box, square;
    pole flag_pole;
    box.height = 12;
    box.width = 10;
    square.height = square.width = 8;
    flag_pole.length = 50;
    flag_pole.depth = 6;
    cout << "The area of the box is " <<
        area(box.height, box.width) << "\n";
    cout << "The area of the square is " <<
        area(square.height, square.width) << "\n";
    cout << "The funny area is " <<
        area(square.height, box.width) << "\n";
    cout << "The bad area is " <<
        area(square.height, flag_pole.depth) << "\n";
}
```

We have two structures declared, one a rectangle and the other a pole. The data fields should be self-explanatory with the exception of the depth of the flagpole, which is the depth it is buried in the ground. The overall length of the pole is therefore the sum of the length and the depth.

Based on your experience with ANSI-C, you should have no problem understanding exactly what this program is doing, but you may be a bit confused at the meaning of the result found in lines 28 and 29, where we multiply the height of the square with the width of the box. This is perfectly legal in ANSI-C or C++, but the result has no meaning because the data are for two different entities. Likewise, the result calculated in lines 30 and 31 is even sillier because the product of the height of the square and the depth of the flagpole has absolutely no meaning in any real-world physical system we can think up.

Would it not be convenient if we had some way to prevent such things from happening in a large production program? If we had a program that defined all of the things we can do with a square and another program that defined everything we could do with a pole, and if the data could be kept mutually exclusive, we could prevent these things from happening. The next program will do just those things for us and do it in a very elegant way.

Objects protect data

The next program [CLASPOLE.CPP] is an example of data protection in a very simple program.

```
#include <iostream.h>
class rectangle {           // A simple class
int height;
int width;
public:
int area(void);           // with two methods
void initialize(int, int);
};
int rectangle::area(void)  //Area of a rectangle
{
return height * width;
}
void rectangle::initialize(int init_height, int init_width)
{
height = init_height;
width = init_width;
}
struct pole {
int length;
int depth;
};
void main()
{
rectangle box, square;
pole flag_pole;
// box.height = 12;
// box.width = 10;
// square.height = square.width = 8;
box.initialize(12, 10);
square.initialize(8, 8);
flag_pole.length = 50;
flag_pole.depth = 6;
cout << "The area of the box is " << box.area() << "\n";
cout << "The area of the square is " << square.area() << "\n";
// cout << "The funny area is " <<
//      area(square.height, box.width) << "\n";
// cout << "The bad area is " <<
//      area(square.height, flag_pole.depth) << "\n";
}
```

In this program the rectangle is changed to a class with the same two variables (which are now private) and two methods to handle the private data. One method is used to initialise the values of the objects created and the other method to return the area of the object. The two methods are defined in lines 7 to 17 in the manner described previously. The pole is left as a structure to illustrate that the two can be used together and that C++ is truly an extension of ANSI-C.

In line 24 we declare two objects, once again named box and square, but this time we cannot assign values directly to their individual components because they are private elements of the class. Lines 26 to 28 are commented out for that reason and the messages are sent to the objects in lines 29 and 30 to tell them to initialise themselves to the values input as parameters. The flag_pole is initialised in the same manner as in the previous program. Using the class in this way prevents us from making the silly calculations we did in the last program. The

compiler is now being used to prevent the erroneous calculations. The end result is that the stupid calculations we did in the last program are not possible in this program so lines 35 to 38 have been commented out. Once again, it is difficult to see the utility of this in such a simple program, but in a large program, using the compiler to enforce the rules can really pay off.

Even though the square and the box are both objects of class rectangle, their private data is hidden from each other such that neither can purposefully or accidentally change the others data. This is the abstract data type mentioned earlier in this section, a model with an allowable set of variables for data storage and a set of allowable operations that can be performed on that stored data. The only operations that can be performed on the data are those defined by the methods which prevents many kinds of erroneous or silly operations. Encapsulation and data hiding bind the data and procedures, or methods, tightly together and limit the scope and visibility of each. Once again, we have the “divide and conquer” technique in which an object is separated from the rest of the code and carefully developed in complete isolation from it. Only then is it integrated into the rest of the code with a few very simple interfaces.

A good example of the use of this technique is in file access in ANSI-C. The data in the file are only available through the predefined functions provided by your compiler writer. You have no direct access to the actual data because it is impossible for you to address the actual data stored on the disk. The data are therefore private data, as far as you are concerned, but the available functions are very much like methods in C++. There are two aspects of this technique that really count when you are developing software. First, you can get all of the data you really need from the file system because the interface is complete, but secondly, you cannot get any data that you do not need. You are prevented from getting into the file handling system and accidentally corrupting some data stored within it. You are also prevented from using the wrong data because the functions available demand a serial access to the data.

Another example is monitor and keyboard handling routines. You are prevented from getting into the workings of them and corrupting them accidentally (or even on purpose), but once again, you are provided with all of the data interfaces you need.

Suppose you are developing a program to analyse some characteristics of flagpoles. You would not wish to accidentally use some data referring to where the flagpole program was stored on your hard disk as the height of the flagpole, nor would you wish to use the cursor position as the flagpole thickness or colour. All code for the flagpole is developed alone, and only when it is finished, is it available for external

use. When using it, you have a very limited number of operations which you can do with the class. The fact that the data is hidden from you protects you from accidentally doing such a thing when you are working at midnight to try to meet a schedule. Once again, this is referred to as information hiding and is one of the primary advantages of object oriented programming over procedural techniques.

Based on the discussion given above you can see that object-oriented programming is not really new, since it has been used in a small measure for as long as computers have been popular. The newest development, however, is in allowing the programmer to partition her programs in such a way that she too can practice information hiding and reduce the debugging time.

Cost of encapsulation

It should be clear that this technique has an efficiency cost because every access of the elements of the object will require a call to a method. In building a large program, however, this could easily be saved in debug time. This is because a program made up of objects that closely match the application are much easier to understand than a program that does not.

In a real project, however, there could be considerable savings in development time (and therefore cost) if someone developed all the details of the rectangle, programmed them, and made them available to you to use.

This is exactly what has been done if you regard the video monitor (screen) as an 'object'. There is a complete set of pre-programmed and debugged routines you can use to make the monitor do anything you wish it to do, all you have to do is study the interface to the routines and use them, expecting them to work. It is impossible for you to multiply the size of your monitor screen by the depth of the flag pole because that information is not available to you to use in a corruptible or meaningless way.

Exercise 3

Describe how a message is sent to a method.

Objective 4 After working through this section you should be able to define and use constructors and destructors in C++

The next program [CONSOLE.CPP] introduces *constructors* and *destructors*.

```
#include <iostream.h>
class rectangle {           // A simple class
int height;
int width;
public:
    rectangle(void);       // with a constructor,
    int area(void);        // two methods,
    void initialize(int, int);
    ~rectangle(void);      // and a destructor
};
rectangle::rectangle(void) // constructor
{
    height = 6;
    width = 6;
}
int rectangle::area(void)  // Area of a rectangle
{
    return height * width;
}
void rectangle::initialize(int init_height, int init_width)
{
    height = init_height;
    width = init_width;
}
rectangle::~~rectangle(void) // destructor
{
    height = 0;
    width = 0;
}
struct pole {
    int length;
    int depth;
};
void main()
{
    rectangle box, square;
    pole flag_pole;
    cout << "The area of the box is " << box.area() << "\n";
    cout << "The area of the square is " << square.area() << "\n";
    // box.height = 12;
    // box.width = 10;
    // square.height = square.width = 8;
    box.initialize(12, 10);
    square.initialize(8, 8);
    flag_pole.length = 50;
    flag_pole.depth = 6;
    cout << "The area of the box is " << box.area() << "\n";
    cout << "The area of the square is " << square.area() << "\n";
    // cout << "The funny area is " <<
    //         area(square.height, box.width) << "\n";
    // cout << "The bad area is " <<
    //         area(square.height, flag_pole.depth) << "\n";
}
```

This program is identical to the last except that a constructor has been added as well as a destructor. The constructor always has the same name as the class itself and is declared in line 8 then defined in lines 11 to 15. The constructor is called automatically by the C++ system when

the object is declared and can therefore be of great help in preventing the use of an uninitialised variable. When the object named `box` is declared in line 36, the constructor is called automatically by the system. The constructor sets the values of `height` and `width` each to 6 in the object named `box`. This is printed out for reference in lines 38 and 39. Likewise, when the square is declared in line 36, the values of the `height` and the `width` of the square are each initialised to 6 when the constructor is called automatically.

A constructor is defined as having the same name as the class itself. In this case both are named `rectangle`. The constructor cannot have a return type associated with it since it is not permitted to have a user defined return type. It actually has a predefined return type, a pointer to the object itself, but we will not be concerned about this until much later in this tutorial. Even though both objects are assigned values by the constructor, they are initialised in lines 45 and 46 to new values and processing continues. Since we have a constructor that does the initialisation, we should probably rename the method named `initialize()` something else but it illustrates the concept involved here.

The destructor is very similar to the constructor except that it is called automatically when each of the objects goes out of scope. You will recall that automatic variables have a limited lifetime since they cease to exist when the enclosing block in which they were declared is exited. When an object is about to be automatically deallocated, its destructor, if one exists, is called automatically. A destructor is characterised as having the same name as the class but with a tilde prepended to the class name. A destructor has no return type.

A destructor is declared in line 9 and defined in lines 26 to 29. In this case the destructor only assigns zeros to the variables prior to their being deallocated, so nothing is really accomplished. The destructor is only included for illustration of how it is used. If some blocks of memory were dynamically allocated within an object, a destructor should be used to deallocate them prior to losing the pointers to them. This would return their memory to the free store for further use later in the program.

It is interesting to note that if a constructor is used for an object that is declared prior to the main program, otherwise known as globally, the constructor will actually be executed prior to the execution of the main program. Similarly, if a destructor is defined for such a variable, it will execute after the main program has been executed. This will not adversely affect your programs, but it is worth noting.

Exercise 4

Define the terms *constructor* and *destructor*.

Modify CONSOLE.CPP by adding appropriate output statements to the constructor and destructor of the rectangle object to verify how and when they are called.

Objective 5 After working through this section you should be able to list the correct methods for using classes for object packaging in C++

The program below [BOXES1.CPP] is an example of how *not* to package an object for universal use. This packaging is actually fine for a very small program but is meant to illustrate to you how to split your program up into smaller more manageable files when you are developing a large program or when you are part of a team developing a large system. The next three example programs in this section will illustrate the proper method of packaging a class.

```
#include <iostream.h>
class box {
    int length;
    int width;
public:
    box(void);           //Constructor
    void set(int new_length, int new_width);
    int get_area(void) {return (length * width);}
    ~box(void);         //Destructor
};
box::box(void)         //Constructor implementation
{
    length = 8;
    width = 8;
}
// This method will set a box size to the two input parameters
void box::set(int new_length, int new_width)
{
    length = new_length;
    width = new_width;
}
box::~~box(void)       //Destructor
{
    length = 0;
    width = 0;
}
void main()
{
    box small, medium, large;           //Three boxes to work with
    small.set(5, 7);
        // Note that the medium box uses the values supplied
        // by the constructor
    large.set(15, 20);
    cout << "The small box area is " << small.get_area() << "\n";
    cout << "The medium box area is " << medium.get_area() << "\n";
    cout << "The large box area is " << large.get_area() << "\n";
}
```

This program is very similar to the last one with the pole structure dropped and the class renamed box. The class is defined in lines 2 to 10, the implementation of the class is given in lines 11 to 26, and the use of the class is given in lines 29 to 36. With the explanation we gave of the last program, you should have no problem understanding this program in detail.

Inline implementation

The method in line 8 contains the implementation for the method as a part of the declaration because it is very simple, and because it introduces another new topic which you will use often in C++ programming. When the implementation is included in the declaration, it will be assembled *inline* wherever this function is called. This leads to much faster code because there is no overhead to accomplish the call to the method. In some cases this will lead to code that is both smaller and faster. This is yet another illustration of the efficiency built into the C++ programming language.

The class header file

If you examine BOX.H carefully, you will see that it is only the class definition.

```
class box {
    int length;
    int width;
public:
    box(void);                //Constructor
    void set(int new_length, int new_width);
    int get_area(void) {return (length * width);}
    ~box(void);              //Destructor
};
```

No details are given of how the various methods are implemented except of course for the inline method named `get_area()`. This gives the complete definition of how to use the class with no implementation details. You would be advised to keep a hardcopy of this file available as we study the next two files. You will notice that it contains lines 2 to 8 of the previous example program named BOXES1.CPP. This is called the *class header file* and cannot be compiled or executed.

The Class Implementation File

Examine the next program file [BOX.CPP] for the implementation of the methods declared in the class header file. Notice that the class header file is included into this file in line 1 which contains all of the prototypes for its methods.

```
#include "box.h"
box::box(void)                //Constructor implementation
{
    length = 8;
    width = 8;
}
// This method will set a box size to the two input parameters
void box::set(int new_length, int new_width)
{
    length = new_length;
    width = new_width;
}
box::~~box(void)              //Destructor
{
```

```
length = 0;  
width = 0;  
}
```

The code from lines 11 to 26 of BOXES1.CPP is contained in this file which is the implementation of the methods declared in the class named box. This file can be compiled but it cannot be executed because there is no main entry point which is required for all ANSI-C or C++ programs. When it is compiled, the object code will be stored in the current directory and available for use by other programs. It should be noted here that the result of compilation is usually referred to as an object file because it contains object code. This use of the word object has nothing to do with the word object as used in object-oriented programming. It is simply a matter of overloading the use of the word. The practice of referring to the compiled result as an object file began long before the method of object-oriented programming was ever considered.

The separation of the definition and the implementation is a major step forward in software engineering. The definition file is all the user needs in order to use this class effectively in a program. She needs no knowledge of the actual implementation of the methods. If she had the implementation available, she may study the code and find a trick she could use to make the overall program slightly more efficient, but this would lead to non-portable software and possible bugs later if the implementor changed the implementation without changing the interface. The purpose of object-oriented programming is to hide the implementation in such a way that the implementation can not affect anything outside of its own small and well defined boundary or interface. Compile this implementation file now and we will use the resulting object file with the next example program.

Using the box object

Examine the next program [BOXES2.CPP] and you will find that the class we defined previously is used within this file. In fact, these last three programs taken together are identical to the program named BOXES1.CPP studied earlier.

```
#include <iostream.h>  
#include "box.h"  
void main()  
{  
    box small, medium, large;           //Three boxes to work with  
    small.set(5, 7);  
        // Note that the medium box uses the values  
        // supplied by the constructor  
    large.set(15, 20);  
    cout << "The small box area is " << small.get_area() << "\n";  
    cout << "The medium box area is " << medium.get_area() << "\n";  
    cout << "The large box area is " << large.get_area() << "\n";  
}
```

The BOX.H file is included here, in line 2, since the definition of the box class is needed to declare three objects and use their methods. You

should have no trouble seeing that this is a repeat of the previous program and will execute in exactly the same way. There is a big difference between BOXES1.CPP and BOXES2.CPP as we will see shortly.

A very important distinction must be made at this point. We are not merely calling functions and changing the terminology a little to say we are sending messages; there is an inherent difference in the two operations. Since the data for each object are tightly bound up in the object, there is no way to get to the data except through the methods, and we send a message to the object telling it to perform some operation based on its internally stored data. However, whenever we call a function, we take along the data for it to work with as parameters since it doesn't contain its own data.

You can compile and execute this program, but when you come to the link step, you will be required to link this program along with the result of the earlier compilation of class named box, named BOX.OBJ. This is where a make or project file will be required, as described in Module 812. We will be linking several more multi-file C++ programs in the remainder of this, and the next, Module.

Information hiding again

The last three example programs have illustrated a method of information hiding that can have a significant impact on the quality of software developed for a large project. Since the only information the user of the class really needs is the class header, that is all she needs to be given. The details of implementation can be kept hidden from him to prevent her from studying the details and possibly using a quirk of programming to write some rather obtuse code. Since she doesn't know exactly what the implementor did, she must follow only the definition given in the header file. This can have a significant impact on a large project. As mentioned earlier, accidental corruption of data is prevented also.

Another reason for hiding the implementation is economic. The company that supplied you with your C++ compiler gave you many library functions but did not supply the source code to the library functions, only the interface to each function. You know how to use the file access functions but you do not have the details of implementation, nor do you need them. Likewise a class library industry can develop which supplies users with libraries of high quality, completely developed and tested classes, for a licensing fee of course. Since the user only needs the interface defined, she can be supplied with the interface and the object (compiled) code for the class and can use it in any way she desires. The supplier's source code is protected from

accidental (or intentional) compromise and she can maintain complete control over it.

It is very important that you understand the principles covered in this section before proceeding on to the next section. It requires considerable analysis and planning to effectively use classes.

Exercise 5

Explain the distinction between method declaration and implementation, and how these are managed in C++.

Objective 6 After working through this section you should be able to define and use abstract data types in C++

An *abstract data type* is a group of data, each of which can store a range of values, and a set of methods or functions that can operate on that data. Since the data are protected from any outside influence, it is protected and said to be encapsulated. Also, since the data is somehow related, it is a very *coherent* group of data that may be highly interactive with each other, but with little interaction of its class outside the scope.

The methods, on the other hand, are coupled to the outside world through the interface, but there are a limited number of contacts with the outside world and therefore a weak coupling with the outside. The object is therefore said to be *loosely coupled* to the outside world. Because of the tight coherency and the loose coupling, ease of maintenance of the software is greatly enhanced. The ease of maintenance may be the greatest benefit of object-oriented programming.

It may bother you that even though the programmer may not use the private variables directly outside of the class, they are in plain sight and she can see what they are and can probably make a good guess at exactly how the class is implemented. The variables could have been hidden completely out of sight in another file, but because the designers of C++ wished to make the execution of the completed application as efficient as possible, the variables were left in the class definition where they can be seen but not used.

Friend functions

A function outside of a class can be defined to be a *friend* function by the class which gives the friend free access to the private members of the class. This in effect, opens a small hole in the protective shield of the class, so it should be used very carefully and sparingly. There are cases where it helps to make a program much more understandable and allows controlled access to the data. Friend functions will be illustrated in some of the example programs later in this tutorial. It is mentioned here for completeness of this section. A single isolated function can be declared as a friend, as well as members of other classes, and even entire classes can be given friend status if needed in a program. Neither a constructor nor a destructor can be a friend function.

The STRUCT in C++

The struct is still useable in C++ and operates just like it does in ANSI-C, but with one addition. You can include methods in a structure that operate on data in the same manner as in a class, but all methods and

data are automatically defaulted to be public in a structure. Of course you can make any of the data or methods private by defining a private section within the structure. The structure should be used only for constructs that are truly structures. If you are building even the simplest objects, you are advised to use classes to define them.

A Practical Class

The examples of encapsulation used so far in this Module have all been extremely simple in order to focus attention on the mechanics of encapsulation. In this section we will study the development of a noticeably larger example, based upon the date class. The date class is a complete, nontrivial class which can be used in any program to get the current date and print it as an ASCII string in any of four predefined formats. It can also be used to store any desired date and format it for display. The next program file [DATE.H] is the header file for the date class.

```
// This date class is intended to illustrate how to write a non-
// trivial class in C++. Even though this class is non-trivial,
// it is still simple enough for a new C++ programmer to follow
// all of the details.
#ifndef DATE_H
#define DATE_H
class date {
protected:
    int month;                // 1 through 12
    int day;                  // 1 through max_days
    int year;                 // 1500 through 2200
    static char out_string[25]; // Format output area
    static char format;       // Format to use for output
    // Calculate how many days are in any given month
    // Note - This is a private method which can be called only
    //         from within the class itself
    int days_this_month(void);
public:
    // Constructor - Set the date to the current date and set
    //               the format to 1
    date(void);
    // Set the date to these input parameters
    // if return = 0 ---> All data is valid
    // if return = 1 ---> Something out of range
    int set_date(int in_month, int in_day, int in_year);
    // Get the month, day, or year of the stored date
    int get_month(void) { return month; };
    int get_day(void)   { return day;   };
    int get_year(void)  { return year;  };
    // Select the desired string output format for use when the
    // get_date_string is called
    void set_date_format(int format_in) { format = format_in; };
    // Return an ASCII-Z string depending on the stored format
    // format = 1   Aug 29, 1991
    // format = 2   8/29/91
    // format = 3   8/29/1991
    // format = 4   29 Aug 1991   Military time
    // format = ?   Anything else defaults to format 1
    char *get_date_string(void);
    // Return Jan Feb Mar Apr etc.
    char *get_month_string(void);
};
#endif
```

There are one or two new features in this code. The reserved word *protected* in line 8 will be defined shortly; for now, think of it as meaning the same as *private*. The keyword *static* in lines 12 and 13 will be explained later, as will the code in lines 5, 6 and 43. For the present, simply pretend those lines (5, 6 and 43) are not there.

You should spend the time necessary to completely understand this class header, with the exception of the new things added, before going on to the implementation for this class.

Using the data class

The following program [DATE.CPP] is the implementation of the date class.

```
// This file contains the implementation for the date class.
#include <stdio.h>          // Prototype for sprintf
#include <time.h>          // Prototypes for the current date
#include "date.h"
char date::format;       // This defines the static data member
char date::out_string[25]; // This defines the static string
                        // Constructor - Set date to current date, and
                        // set format to the default of 1
date::date(void)
{
    time_t time_date;
    struct tm *current_date;
    time_date = time(NULL);           // DOS system call
    current_date = localtime(&time_date); // DOS system call
    month = current_date->tm_mon + 1;
    day = current_date->tm_mday;
    year = current_date->tm_year + 1900;
    format = 1;
}

// Set the date to these input parameters
// if return = 0 ---> All data is valid
// if return = 1 ---> Something out of range
int date::set_date(int in_month, int in_day, int in_year)
{
    int temp = 0;
    int max_days;
    // The limits on the year are purely arbitrary
    if (in_year < 1500) { // Check that the year is between
        year = 1500; // 1500 and 2200
        temp = 1;
    } else {
        if (in_year > 2200) {
            year = 2200;
            temp = 1;
        } else
            year = in_year;
    }
    if(in_month < 1) { // Check that the month is between 1 and 12
        temp = 1;
    } else {
        if (in_month > 12) {
            month = 12;
            temp = 1;
        } else
            month = in_month;
    }
    max_days = days_this_month();
    if (in_day < 1) { // Check that the day is between
```

```

    day = temp = 1;           // 1 and max_days
} else {
    if (in_day > max_days) {
        day = max_days;
        temp = 1;
    } else
        day = in_day;
}
return temp;
}
static char *month_string[13] = {" ", "Jan", "Feb", "Mar", "Apr",
                                "May", "Jun", "Jul", "Aug",
                                "Sep", "Oct", "Nov", "Dec"};
    // Return Jan Feb Mar Apr etc.
char *date::get_month_string(void)
{
return month_string[month];
}
    // Return an ASCII-Z string depending on the stored format
    // format = 1   Aug 29, 1991
    // format = 2   8/29/91
    // format = 3   8/29/1991
    // format = 4   29 Aug 1991   Military time
    // format = ?   Anything else defaults to format 1
char *date::get_date_string(void)
{
switch (format) {
    // This printout assumes that the year will be between 1900
    // and 1999
case 2: sprintf(out_string, "%02d/%02d/%02d",
                month, day, year - 1900);
        break;
case 3: sprintf(out_string, "%02d/%02d/%04d", month, day, year);
        break;
case 4: sprintf(out_string, "%d %s %04d",
                day, month_string[month], year);
        break;
case 1:           // Fall through to the default case
default: sprintf(out_string, "%s %d, %04d",
                month_string[month], day, year);
        break;
}
return out_string;
}
int days[13] = {0, 31, 28, 31, 30, 31,
                30, 31, 31, 30, 31, 30, 31};
    // Since this is declared in the private part of the class
    // header is is only available for use within the class.
    // It is hidden from use outside of the class.
int date::days_this_month(void)
{
if (month != 2)
return days[month];
if (year % 4)           // Not leap year
return 28;
if (year % 100)        // It is leap year
return 29;
if (year % 400)        // Not leap year
return 28;
return 29;           // It is leap year
}

```

The program below [USEDATE.CPP] is a main program that uses the date class to list the current date and another date on the monitor. Once again, you should have no problem understanding this program.

```

// This is a very limited test of the date class
#include <iostream.h>
#include "date.h"

```

```
void main()
{
date today, birthday;
birthday.set_date(7, 21, 1960);
cout << "Limited test of the date class\n";
cout << "Today is " << today.get_date_string() << "\n";
cout << "Birthday is " << birthday.get_date_string() << "\n";
today.set_date_format(4);
cout << "Today is " << today.get_date_string() << "\n";
cout << "Birthday is " << birthday.get_date_string() << "\n";
}
```

This class will be extensively used, in conjunction with others, to illustrate single and multiple inheritance in Module 819. Even though you may not yet understand all the details of these files, spend enough time that you are comfortable with their structure and the major points that they illustrate.

Exercise 6

Describe how structures in C++ differ from structures in C.

Write the header file [NAME.H] for a name class, similar to the date class. This class should store any name as three components (first name, middle name or initial, family name) and display them in any of several different forms (e.g. John Paul Jones; J.P.Jones; Jones, John Paul)

Objective 7 After working through this section you should be able to use objects within C++ Programs

This section looks at how to use some of the traditional aspects of C or C++ programming with classes and objects. Pointers to an object, as well as pointers within an object, will be illustrated, as will arrays embedded within an object, and arrays of objects. Since objects are simply another C++ data construct, all of these things are possible and can be used as needed.

We will use the BOXES1.CPP program from the last section as a starting point and we will add a few new constructs to it for each example program. You will recall that it was a very simple program with the class definition, the class implementation, and the main calling program all in one file. This was selected as a starting point because we will eventually make changes to all parts of the program and it will be convenient to have it all in a single file for illustrative purposes. It must be kept in mind however that the proper way to use these constructs is to separate them into the three files as was illustrated in BOX.H, BOX.CPP, and BOXES2.CPP in the last section. This allows the implementor of box to supply the user with only the interface, namely BOX.H. Not giving her the implementation file named BOX.CPP is to practice the technique of information hiding.

An array of objects

Examine the next program [OBJARRAY.CPP] for an example of an array of objects. This file is nearly identical to the file named BOX1.CPP until we come to line 33 where an array of 4 boxes is declared.

```
#include <iostream.h>
class box {
    int length;
    int width;
    static int extra_data;    // Declaration of extra_data
public:
    box(void);                //Constructor
    void set(int new_length, int new_width);
    int get_area(void);
    int get_extra(void) {return extra_data++;}
};
int box::extra_data;        // Definition of extra_data
box::box(void)              //Constructor implementation
{
    length = 8;
    width = 8;
    extra_data = 1;
}
// This method will set a box size to the two input parameters
void box::set(int new_length, int new_width)
{
    length = new_length;
    width = new_width;
}
```

```
    }
    // This method will calculate and return the area of a box
    instance
    int box::get_area(void)
    {
        return (length * width);
    }
void main()
{
    box small, medium, large, group[4];    //Seven boxes to work with
    small.set(5, 7);
    large.set(15, 20);
    for (int index = 1 ; index < 4 ; index++)
        //group[0] uses default
        group[index].set(index + 10, 10);
    cout << "The small box area is " << small.get_area() << "\n";
    cout << "The medium box area is " << medium.get_area() << "\n";
    cout << "The large box area is " << large.get_area() << "\n";
    for (index = 0 ; index < 4 ; index++)
        cout << "The array box area is " <<
            group[index].get_area() << "\n";
    cout << "The extra data value is " << small.get_extra() << "\n";
    cout << "The extra data value is " << medium.get_extra() << "\n";
    cout << "The extra data value is " << large.get_extra() << "\n";
    cout << "The extra data value is " << group[0].get_extra()
        << "\n";
    cout << "The extra data value is " << group[3].get_extra()
        << "\n";
}
```

Recalling the operation of the constructor, you will remember that each of the four box objects will be initialised to the values defined within the constructor since each box will go through the constructor as they are declared. In order to declare an array of objects, a constructor for that object must not require any parameters. (We have not yet illustrated a constructor with initialising parameters, but we will in the next program.) This is an efficiency consideration since it would probably be an error to initialise all elements of an array of objects to the same value. We will see the results of executing the constructor when we compile and execute the file later.

Line 36 defines a for loop that begins with 1 instead of the normal starting index for an array leaving the first object, named group[0], to use the default values stored when the constructor was called. You will observe that sending a message to one of the objects uses the same construct as is used for any object. The name of the array followed by its index in square brackets is used to send a message to one of the objects in the array. This is illustrated in line 36 and the operation of that code should be clear to you. The other method is called in the output statement in lines 39 to 41 where the area of the four boxes in the group array are listed on the monitor.

Another fine point should be pointed out. The integer variable named index is declared in line 36 and is still available for use in line 38 since we have not yet left the enclosing block which begins in line 32 and extends to line 51.

Declaration and definition of a variable

An extra variable was included for illustration, the one named `extra_data`. Since the keyword `static` is used to modify this variable in line 5, it is an external variable and only one copy of this variable will ever exist. All seven objects of this class share a single copy of this variable which is global to the objects defined in line 33.

The variable is actually only declared here which says it will exist somewhere, but it is not defined. A declaration says the variable will exist and gives it a name, but the definition actually defines a place to store it somewhere in the computers memory space. By definition, a static variable can be declared in a class header but it cannot be defined there, so it is defined somewhere outside of the header, usually in the implementation file. In this case it is defined in line 12 and can then be used throughout the class.

Line 17 of the constructor sets the single global variable to 1 each time an object is declared. Only one assignment is necessary so the other six are actually wasted code. To illustrate that there is only one variable shared by all objects of this class, the method to read its value also increments it. Each time it is read in lines 45 to 51 it is incremented, and the result of the execution proves that there is only a single variable shared by all objects of this class. You will also note that the method named `get_extra()` is defined within the class declaration so it will be assembled into the final program as inline code.

A string within an object

Look at the next program [OBJSTRNG.CPP] for our first example of an object with an embedded string (actually an embedded pointer, but the two work so closely together that we can study one and understand both).

```
#include <iostream.h>
class box {
    int length;
    int width;
    char *line_of_text;
public:
    box(char *input_line);           //Constructor
    void set(int new_length, int new_width);
    int get_area(void);
};
box::box(char *input_line)         //Constructor implementation
{
    length = 8;
    width = 8;
    line_of_text = input_line;
}
// This method will set a box size to the two input parameters
void box::set(int new_length, int new_width)
{
    length = new_length;
```

```
        width = new_width;
    }
    // This method will calculate and return the area of a box
    instance
    int box::get_area(void)
    {
        cout << line_of_text << "= ";
        return (length * width);
    }
}
void main()
{
    box small("small box "),      //Three boxes to work with
    medium("medium box "),
    large("large box ");
    small.set(5, 7);
    large.set(15, 20);
    cout << "The area of the ";
    cout << small.get_area() << "\n";
    cout << "The area of the ";
    cout << medium.get_area() << "\n";
    cout << "The area of the ";
    cout << large.get_area() << "\n";
}
```

You will notice that line 5 contains a pointer to a string named `line_of_text`. The constructor contains an input parameter which is a pointer to a string which will be copied to the string named `line_of_text` within the constructor. We could have defined the variable `line_of_text` as an actual array in the class, then used `strcpy()` to copy the string into the object and everything would have worked the same, but we will leave that as an exercise for you at the end of this section. It should be pointed out that we are not limited to passing a single parameter to a constructor. Any number of parameters can be passed, as will be illustrated later.

You will notice that when the three boxes are declared this time, we supply a string constant as an actual parameter with each declaration which is used by the constructor to assign the string pointer some data to point to. When we call `get_area()` in lines 38 to 42, we get the message displayed and the area returned. It would be prudent to put these operations in separate methods since there is no apparent connection between printing the message and calculating the area, but it was written this way to illustrate that it can be done. What this really says is that it is possible to have a method that has a side effect, the message output to the monitor, and a return value, the area of the box.

An object with an internal pointer

The next program [OBJINTPT.CPP] is our first example program with an embedded pointer which will be used for dynamic allocation of data.

```
#include <iostream.h>
class box {
    int length;
    int width;
    int *point;
public:
    box(void);                //Constructor
```

```
void set(int new_length, int new_width, int stored_value);
    int get_area(void) {return length * width;}           // Inline
    int get_value(void) {return *point;}                // Inline
    ~box();                                             //Destructor
};
box::box(void)                                         //Constructor implementation
{
length = 8;
width = 8;
point = new int;
*point = 112;
}
// This method will set a box size to the input parameters
void box::set(int new_length, int new_width, int stored_value)
{
length = new_length;
width = new_width;
*point = stored_value;
}
box::~box(void)                                       //Destructor
{
length = 0;
width = 0;
delete point;
}
void main()
{
box small, medium, large;                            //Three boxes to work with
small.set(5, 7, 177);
large.set(15, 20, 999);
cout << "The small box area is " << small.get_area() << "\n";
cout << "The medium box area is " << medium.get_area() << "\n";
cout << "The large box area is " << large.get_area() << "\n";
cout << "The small box stored value is " <<
    small.get_value() << "\n";
cout << "The medium box stored value is " <<
    medium.get_value() << "\n";
cout << "The large box stored value is " <<
    large.get_value() << "\n";
}
```

In line 5 we declare a pointer to an integer variable, but it is only a pointer: there is no storage associated with it. The constructor therefore allocates an integer type variable on the heap for use with this pointer in line 18. It should be clear to you that the three objects created in line 35 each contain a pointer which points into the heap to three different locations. Each object has its own dynamically allocated variable for its own private use. Moreover, each has a value of 112 stored in its dynamically allocated data because line 18 stores that value in each of the three locations, once for each call to the constructor.

In such a small program, there is no chance that we will exhaust the heap, so no test is made for unavailable memory. In a real production program, it would be essential to test that the value of the returned pointer is not NULL to ensure that the data actually did get allocated.

The method named set() has three parameters associated with it and the third parameter is used to set the value of the new dynamically allocated variable. There are two messages passed, one to the small box and one to the large box. As before, the medium box is left with its default values.

The three areas are displayed followed by the three stored values in the dynamically allocated variables, and we finally have a program that requires a destructor in order to be completely proper. If we simply leave the scope of the objects as we do when we leave the main program, we will leave the three dynamically allocated variables on the heap with nothing pointing to them. They will be inaccessible and will therefore represent wasted storage on the heap. For that reason, the destructor is used to delete the variable which the pointer named `point` is referencing as each object goes out of existence. In this case, lines 29 and 30 assign values to variables that will be automatically deleted. Even though these lines of code really do no good, they are legal statements.

Actually, in this particular case, the variables will be automatically reclaimed when we return to the operating system because all program cleanup is done for us at that time. If this were a function that was called by another function however, the heap space would be wasted. This is an illustration of good programming practice, that of cleaning up after yourself when you no longer need some dynamically allocated variables.

One other construct should be mentioned again, that of the inline method implementations in line 9 and 10. As we have already seen, inline functions can be used where speed is of the utmost in importance since the code is assembled inline rather than by actually making a method call. Since the code is defined as part of the declaration, the system will assemble it inline, and a separate implementation for these methods is not needed. If the inline code is too involved, the compiler is allowed to ignore the inline request and will actually assemble it as a separate method, but it will do it invisibly to you and will probably not even tell you about it.

Remember that we are interested in using information hiding and inline code prevents hiding of the implementation, putting it out in full view. Many times you will be more interested in speeding up a program than you are in hiding a trivial implementation. Since most inline methods are trivial, you should feel free to use the inline code construct.

A dynamically allocated object

Examine the next program [OBJDYNAM.CPP] for our first look at a dynamically allocated object.

```
#include <iostream.h>
class box {
    int length;
    int width;
public:
    box(void);                //Constructor
    void set(int new_length, int new_width);
```

```
    int get_area(void);
};
box::box(void) //Constructor implementation
{
    length = 8;
    width = 8;
}
// This method will set a box size to the two input parameters
void box::set(int new_length, int new_width)
{
    length = new_length;
    width = new_width;
}
// This method will calculate and return the area of a box
instance
int box::get_area(void)
{
    return (length * width);
}
void main()
{
    box small, medium, large; //Three boxes to work with
    box *point; //A pointer to a box
    small.set(5, 7);
    large.set(15, 20);
    point = new box;
        // Use the defaults supplied by the constructor
    cout << "The small box area is " << small.get_area() << "\n";
    cout << "The medium box area is " << medium.get_area() << "\n";
    cout << "The large box area is " << large.get_area() << "\n";
    cout << "The new box area is " << point->get_area() << "\n";
        point->set(12, 12);
    cout << "The new box area is " << point->get_area() << "\n";
        delete point;
}
}
```

In line 30 we declare a pointer to an object of type box and since it is only a pointer with nothing to point to, we dynamically allocate an object for it in line 33, with the object being created on the heap just like any other dynamically allocated variable. When the object is created in line 33, the constructor is called automatically to assign values to the two internal storage variables. Note that the constructor is not called when the pointer is declared since there is nothing to initialise; it is called when the object is allocated.

Reference to the components of the object are handled in much the same way that structure references are made, through use of the pointer operator as illustrated in lines 38 to 41. Of course you can use the pointer dereferencing method without the arrow such as

```
(*point).set (12, 12);
```

as a replacement for line 39 but the arrow notation is much more universal and should be used. Finally, the object is deleted in line 41 and the program terminates. If there were a destructor for this class, it would be called as part of the delete statement to clean up the object prior to deletion.

An object with a pointer to another object

The program below [OBJLIST.CPP] contains an object with an internal reference to another object of its own class. This is the standard structure used for a singly linked list and we will keep the use of it very simple in this program.

```
#include <iostream.h>
class box {
    int length;
    int width;
    box *another_box;
public:
    box(void); //Constructor
    void set(int new_length, int new_width);
    int get_area(void);
    void point_at_next(box *where_to_point);
    box *get_next(void);
};
box::box(void) //Constructor implementation
{
    length = 8;
    width = 8;
    another_box = NULL;
}
// This method will set a box size to the two input parameters
void box::set(int new_length, int new_width)
{
    length = new_length;
    width = new_width;
}
// This method will calculate and return the area of a box
instance
int box::get_area(void)
{
    return (length * width);
}
// This method causes the pointer to point to the input parameter
void box::point_at_next(box *where_to_point)
{
    another_box = where_to_point;
}
// This method returns the box the current one points to
box *box::get_next(void)
{
    return another_box;
}
void main()
{
    box small, medium, large; //Three boxes to work with
    box *box_pointer; //A pointer to a box
    small.set(5, 7);
    large.set(15, 20);
    cout << "The small box area is " << small.get_area() << "\n";
    cout << "The medium box area is " << medium.get_area() << "\n";
    cout << "The large box area is " << large.get_area() << "\n";
    small.point_at_next(&medium);
    medium.point_at_next(&large);
    box_pointer = &small;
    box_pointer = box_pointer->get_next();
    cout << "The box pointed to has area " <<
        box_pointer->get_area() << "\n";
}
```

The constructor contains the statement in line 17 which assigns the pointer the value of NULL to initialise the pointer. This is a good idea

for all of your programming: don't allow any pointer to point off into space, but initialise all pointers to something. By assigning the pointer within the constructor, you guarantee that every object of this class will automatically have its pointer initialised. It will be impossible to overlook the assignment of one of these pointers.

Two additional methods are declared in lines 9 and 10 with the one in line 10 having a construct we have not yet mentioned. This method returns a pointer to an object of the box class. As you are aware, you can return a pointer to a struct in ANSI-C, and this is the parallel construct in C++. The implementation in lines 36 to 39 returns the pointer stored within the object. We will see how this is used when we get to the actual program.

An extra pointer named `box_pointer` is declared in the main program for use later and in line 50 we make the embedded pointer within the small box point to the medium box, and in line 50 we make the embedded pointer within the medium box point to the large box. We have effectively generated a linked list with three elements. In line 52 we make the extra pointer point to the small box. Continuing in line 53 we use it to refer to the small box and update it to the value contained in the small box which is the address of the medium box. We have therefore traversed from one element of the list to another by sending a message to one of the objects. If line 53 were repeated exactly as shown, it would cause the extra pointer to refer to the large box, and we would have traversed the entire linked list which is only composed of three elements.

The keyword THIS

Another new keyword available in C++ is *this*. The word is defined within any object as a pointer to the object in which it is contained. It is implicitly declared as

```
class_name *this;
```

and is initialised to point to the object for which the member function is invoked. This pointer is most useful when working with pointers and especially with a linked list when you need to reference a pointer to the object you are inserting into the list. The keyword *this* is available for this purpose and can be used in any object. Actually the proper way to refer to any variable within a list is through use of the predefined pointer *this*, by writing `this->variable_name`, but the compiler assumes the pointer is used, and we can simplify every reference by omitting the pointer. The keyword *this* will be used in one of the larger example programs later in this Module.

A linked list of objects

The next example program [OBJLINK.CPP] is a complete example of a linked list written in object-oriented notation.

```
#include <iostream.h>
class box {
    int length;
    int width;
    box *another_box;
public:
    box(void); //Constructor
    void set(int new_length, int new_width);
    int get_area(void);
    void point_at_next(box *where_to_point);
    box *get_next(void);
};
box::box(void) //Constructor implementation
{
    length = 8;
    width = 8;
    another_box = NULL;
}
// This method will set a box size to the two input parameters
void box::set(int new_length, int new_width)
{
    length = new_length;
    width = new_width;
}
// This method will calculate and return the
// area of a box instance
int box::get_area(void)
{
    return (length * width);
}
// This method causes the pointer to point
// to the input parameter
void box::point_at_next(box *where_to_point)
{
    another_box = where_to_point;
}
// This method returns the box that this one points to
box *box::get_next(void)
{
    return another_box;
}
void main()
{
    box *start = NULL; // Always points to the start of the list
    box *temp; // Working pointer
    box *box_pointer; // Used for box creation
    // Generate the list
    for (int index = 0 ; index < 10 ; index++ ) {
        box_pointer = new box;
        box_pointer->set(index + 1, index + 3);
        if (start == NULL)
            start = box_pointer; // First element
        else
            temp->point_at_next(box_pointer);
            // Additional element
        temp = box_pointer;
    } // Print the list out
    temp = start;
    do {
        cout << "The area is " << temp->get_area() << "\n";
        temp = temp->get_next();
    } while (temp != NULL); // Delete the list
```

```
temp = start;
do {
    temp = temp->get_next();
    delete start;
    start = temp;
} while (temp != NULL);
}
```

This program is very similar to the last one; in fact it is identical until we get to the main program. You will recall that in the last program the only way we had to set or use the embedded pointer was through use of the two methods named `point_at_next()` and `get_next()` which are listed in lines 33 to 39 of the present program. We will use these to build up our linked list then traverse and print the list. Finally, we will delete the entire list to free the space on the heap.

In lines 44 to 46 we declare three pointers for use in the program. The pointer named `start` will always point to the beginning of the list, but `temp` will move down through the list as we create it. The pointer named `box_pointer` will be used for the creation of each object. We execute the loop in lines 48 to 57 to generate the list where line 50 dynamically allocates a new object of the `box` class and line 52 fills it with nonsense data for illustration. If this is the first element in the list, the `start` pointer is set to point to this element, but if elements already exist, the last element in the list is assigned to point to the new element. In either case, the `temp` pointer is assigned to point to the last element of the list, in preparation for adding another element if there is another element to be added.

In line 58 the pointer named `temp` is pointed to the first element and it is used to increment its way through the list by updating itself in line 62 during each pass through the loop. When `temp` has the value of `NULL`, which it gets from the last element of the list, we are finished traversing the list.

Finally, we delete the entire list by starting at the beginning and deleting one element each time we pass through the loop in lines 64 to 68.

A careful study of the program will reveal that it does indeed generate a linked list of ten elements, each element being an object of class `box`. The length of this list is limited by the practicality of how large a list we desire to print out, but it could be lengthened to many thousands of these simple elements provided you have enough memory available to store them all. Once again, the success of the dynamic allocation is not checked as it should be in a well-written program.

Nesting objects

Examine the next program [NESTING.CPP] for an example of nesting classes which results in nested objects. A nested object could be illustrated with your computer in a rather simple manner. The computer

itself is composed of many items which work together but work entirely differently, such as a keyboard, a disk drive, and a power supply. The computer is composed of these very dissimilar items and it is desirable to discuss the keyboard separately from the disk drive because they are so different. A computer class could be composed of several objects that are dissimilar by nesting the dissimilar classes within the computer class.

If however, we wished to discuss disk drives, we may wish to examine the characteristics of disk drives in general, then examine the details of a hard disk, and the differences of floppy disks. This would involve inheritance because much of the data about both drives could be characterised and applied to the generic disk drive then used to aid in the discussion of the other three. We will study inheritance in the next Module, but for now we will look at the embedded or nested class.

```
#include <iostream.h>
class mail_info {
    int shipper;
    int postage;
public:
    void set(int in_class, int in_postage)
        {shipper = in_class; postage = in_postage; }
    int get_postage(void) {return postage;}
};
class box {
    int length;
    int width;
    mail_info label;
public:
    void set(int l, int w, int s, int p) {
        length = l;
        width = w;
        label.set(s, p); }
    int get_area(void) {return length * width;}
};
void main()
{
    box small, medium, large;
    small.set(2, 4, 1, 35);
    medium.set(5, 6, 2, 72);
    large.set(8, 10, 4, 98);
    cout << "The area is " << small.get_area() << "\n";
    cout << "The area is " << medium.get_area() << "\n";
    cout << "The area is " << large.get_area() << "\n";
}
```

This program contains a class named box which contains an object of another class embedded within it in line 13, the mail_info class. This object is available for use only within the class implementation of box because that is where it is defined. The main program has objects of class box defined but no objects of class mail_info, so the mail_info class cannot be referred to in the main program. In this case, the mail_info class object is meant to be used internally to the box class and one example is given in line 18 where a message is sent to the label.set() method to initialise the variables. Additional methods could be used as needed, but these are given as an illustration of how they can be called.

Of prime importance is the fact that there are never any objects of the `mail_info` class declared directly in the main program, they are inherently declared when the enclosing objects of class `box` are declared. Of course objects of the `mail_info` class could be declared and used in the main program if needed, but they are not in this example program. In order to be complete, the `box` class should have one or more methods to use the information stored in the object of the `mail_info` class. Study this program until you understand the new construct, then compile and execute it.

If the class and the nested classes require parameter lists for their respective constructors an initialisation list can be given. This will be discussed and illustrated later in this Module.

Exercise 7

Modify `OBJDYNAM.CPP` so that the objects *small* and *medium* are pointers, and dynamically allocate them prior to their use.

Modify the loop (from line 47) in `OBJLINK.CPP` so that it stores up to 1000 elements in the list. (Remove the list output if necessary to speed up the execution.) Add a test to display the amount of available memory as the list is built, and to terminate list construction when this becomes too small for a new list node to be added.

Objective 8 After working through this section you should be able to list the advantages of using operator overloading in C++

The example program below [OPOVERLD.CPP] contains examples of *overloading* operators. This allows you to define a class of objects and redefine the use of the normal operators. The end result is that objects of the new class can be used in as natural a manner as the predefined types. In fact, they seem to be a part of the language rather than your own add-on.

```
#include <iostream.h>
class box {
    int length;
    int width;
public:
    void set(int l, int w) {length = l; width = w;}
    int get_area(void) {return length * width;}

    friend box operator+(box a, box b);
    friend box operator+(int a, box b);
    friend box operator*(int a, box b); };
    // Add two boxes' widths together
    box operator+(box a, box b)
    {
    box temp;
    temp.length = a.length;
    temp.width = a.width + b.width;
    return temp;
    }
    box operator+(int a, box b)      // Add a constant to a box
    {
    box temp;
    temp.length = b.length;
    temp.width = a + b.width;
    return temp;
    }
    box operator*(int a, box b)      // Multiply a box by a constant
    {
    box temp;
    temp.length = a * b.length;
    temp.width = a * b.width;
    return temp;
    }
void main()
{
    box small, medium, large;
    box temp;
    small.set(2, 4);
    medium.set(5, 6);
    large.set(8, 10);
    cout << "The area is " << small.get_area() << "\n";
    cout << "The area is " << medium.get_area() << "\n";
    cout << "The area is " << large.get_area() << "\n";
    temp = small + medium;
    cout << "The new area is " << temp.get_area() << "\n";
    temp = 10 + small;
    cout << "The new area is " << temp.get_area() << "\n";
    temp = 4 * large;
    cout << "The new area is " << temp.get_area() << "\n";
}
```

In this case we overload the + operator and the * operator, with the declarations in lines 9 to 13, and the definitions in lines 16 to 35. The methods are declared as friend functions so we can use the double parameter functions as listed. If we did not use the friend construct, the function would be a part of one of the objects and that object would be the object to which the message was sent. Including the friend construct allows us to separate this method from the object and call the method with infix notation. Using this technique, it can be written as object1 + object2 rather than object1.operator+(object2). Also, without the friend construct we could not use an overloading with an int type variable for the first parameter because we can not send a message to an integer type variable such as int.operator+(object). Two of the three operator overloadings use an int for the first parameter so it is necessary to declare them as friend functions. There is no upper limit to the number of overloadings for any given operator. Any number of overloadings can be used provided the parameters are different for each particular overloading.

The header in line 9 illustrates the first overloading where the + operator is overloaded by giving the return type followed by the keyword *operator* and the operator we wish to overload. The two formal parameters and their types are then listed in the parentheses and the normal function operations are given in the implementation of the function in lines 22 to 28. You will notice that the implementation of the friend functions are not actually a part of the class because the class name is not prepended onto the method name in line 9. There is nothing unusual about this implementation, it should be easily understood by you at this point. For purposes of illustration, some silly mathematics are performed in the method implementation, but any desired operations can be done.

The biggest difference occurs where this method is called by using the infix notation instead of the usual message sending format. Since the variables small and medium are objects of the box class, the system will search for a way to use the + operator on two objects of class box and will find it in the overloaded operator+ method we have just discussed. The operations within the method implementation can be anything we need them to be, and they are usually much more meaningful than the silly math included here.

In line 48 we ask the system to add an int type constant to an object of class box, so the system finds the other overloading of the + operator beginning in line 22 to perform this operation. In line 50 we ask the system to use the * operator to do something to an int constant and an object of class box, which it satisfies by finding the method in lines 29 to 35. Note that it would be illegal to attempt to use the * operator the other way around, namely large * 4, since we did not define a method

to use the two types in that order. Another overloading could be given with reversed types, and we could use the reverse order in a program.

You will notice that when using operator overloading, we are also using function name overloading since some of the function names are the same. When we use operator overloading in this manner, we actually make our programs look like the class is a natural part of the language since it is integrated into the language so well. C++ is therefore an extendible language and can be moulded to fit the mechanics of the problem at hand.

Operator overloading caveats

Operator overloading seems particularly prone to misuse. The overloading of operators is only available for classes, you cannot redefine the operators for the predefined simple types. This would probably be very silly anyway since the code could be very difficult to read if you changed some of them around.

The logical and (&&) and the logical or (||) operators can be overloaded for the classes you define, but they will not operate as short circuit operators. All members of the logical construction will be evaluated with no regard concerning the outcome. Of course the normal predefined logical operators will continue to operate as short circuit operators as expected, but not the overloaded ones.

If the increment (++) or decrement (--) operators are overloaded, the system has no way of telling whether the operators are used as pre-increment or post-increment (or pre-decrement or post-decrement) operators. Which method is used is implementation-dependent, so you should use them in such a way that it doesn't matter which is used.

Function overloading in a class

Examine the program below [FUNCOVER.CPP] for an example of function name overloading within a class. In this program the constructor is overloaded as well as one of the methods to illustrate what can be done.

```
#include <iostream.h>
class many_names {
    int length;
    int width;
public:
    many_names(void);           // Constructors
    many_names(int len);
    many_names(int len, int wid);
    void display(void);        // Display functions
    void display(int one);
    void display(int one, int two);
    void display(float number);
};
```

```
many_names::many_names(void)
{
length = 8;
width = 8;
}
many_names::many_names(int len)
{
length = len;
width = 8;
}
many_names::many_names(int len, int wid)
{
length = len;
width = wid;
}
void many_names::display(void)
{
cout << "From void display function, area = " <<
length * width << "\n";
}
void many_names::display(int one)
{
cout << "From int display function, area = " <<
length * width << "\n";
}
void many_names::display(int one, int two)
{
cout << "From two int display function, area = " <<
length * width << "\n";
}
void many_names::display(float number)
{
cout << "From float display function, area = " <<
length * width << "\n";
}
void main()
{
many_names small, medium(10), large(12, 15);
int gross = 144;
float pi = 3.1415, payroll = 12.50;
small.display();
small.display(100);
small.display(gross,100);
small.display(payroll);
medium.display();
large.display(pi);
}
```

This program illustrates some of the uses of overloaded names and a few of the rules for their use. You will recall that the function selected is based on the number and types of the formal parameters only. The type of the return value is not significant in overload resolution.

In this case there are three constructors. The constructor which is actually called is selected by the number and types of the parameters in the definition. In line 51 of the main program the three objects are declared, each with a different number of parameters and inspection of the results will indicate that the correct constructor was called based on the number of parameters.

In the case of the other overloaded methods, the number and type of parameters is clearly used to select the proper method. You will notice that one method uses a single integer and another uses a single float

type variable, but the system is able to select the correct one. As many overloads as desired can be used provided that all of the parameter patterns are unique.

Actually, we have already used an overloaded operator extensively without any problems. The `cout` operator works as an overloaded function since the way it outputs data is a function of the type of its input variable or the field we ask it to display. Many programming languages have overloaded output functions so you can output any data with the same function name.

Separate compilation

Separate compilation is available with C++ and it follows the rules for ANSI-C separate compilation. As expected, separately compiled files can be linked together. However, since classes are used to define objects, the nature of C++ separate compilation is considerably different from that used for ANSI-C. This is because the classes used to create the objects are not considered as external variables, but as included classes. This makes the overall program look different from a pure ANSI-C program. Your programs will take on a different appearance as you gain experience in C++.

Another practical example

In the last section we studied the date class, and now we will study a simple time class. You should begin by studying the file below [TIME.H] which will look very similar to the date class header.

```
// This is probably the minimum usable time class,
// but is intended as an illustration of a class rather
// than to build an all inclusive class for future use.
// Each student can develop his own to suit his own taste.
#ifndef TIME_H
#define TIME_H
class time_of_day {
protected:
    int hour;           // 0 through 23
    int minute;        // 0 through 59
    int second;        // 0 through 59
    static char format; // Format to use for output
    static char out_string[25]; // Format output area
public:
    // Constructor - Set time to current time and format to 1
    time_of_day(void);
    time_of_day(int H) {hour = H; minute = 0; second = 0; };
    time_of_day(int H, int M) {hour = H; minute = M; second = 0; };
    time_of_day(int H, int M, int S)
        {hour = H; minute = M; second = S; };
    // Set the time to these input values
    // return = 0 ---> data is valid
    // return = 1 ---> something is out of range
    int set_time(void);
    int set_time(int hour_in);
    int set_time(int hour_in, int minute_in);
    int set_time(int hour_in, int minute_in, int second_in);
    // Select string output format
```

```

void set_time_format(int format_in) { format = format_in; };
    // Return an ASCII-Z string depending on the stored format
    //   format = 1   13:23:12
    //   format = 2   13:23
    //   format = 3   1:23 PM
char *get_time_string(void);
};
#endif

```

The only major difference in this class from the date class is the overloaded constructors and methods. The program is a very practical example that illustrates graphically that many constructor overloadings are possible. The implementation for the time class is given below [TIME.CPP].

```

#include <stdio.h>           // For the sprintf function
#include <time.h>           // For the time & localtime functions
#include "time.h"          // For the class header
char time_of_day::format;
    // This defines the static data member
char time_of_day::out_string[25]; // This defines the string
    // Constructor - Set time to current time and format to 1
time_of_day::time_of_day(void)
{
time_t time_date;
struct tm *current_time;
    time_date = time(NULL);
    current_time = localtime(&time_date);
    hour = current_time->tm_hour;
    minute = current_time->tm_min;
    second = current_time->tm_sec;
    format = 1;
}
    // Set the time to these input values
    // return = 0 ---> data is valid
    // return = 1 ---> something out of range
int time_of_day::set_time(void)
{
return set_time(0, 0, 0);
};
int time_of_day::set_time(int H)
{
return set_time(H, 0, 0);
};
int time_of_day::set_time(int H, int M)
{
return set_time(H, M, 0);
};
int time_of_day::set_time
    (int hour_in, int minute_in, int second_in)
{
int error = 0;
if (hour_in < 0) {
    hour_in = 0;
    error = 1;
} else if (hour_in > 59) {
    hour_in = 59;
    error = 1;
}
hour = hour_in;
if (minute_in < 0) {
    minute_in = 0;
    error = 1;
} else if (minute_in > 59) {
    minute_in = 59;
    error = 1;
}
minute = minute_in;
}

```

```
if (second_in < 0) {
    second_in = 0;
    error = 1;
} else if (second_in > 59) {
    second_in = 59;
    error = 1;
}
second = second_in;
return error;
}
// Return an ASCII-Z string depending on the stored format
// format = 1    13:23:12
// format = 2    13:23
// format = 3    1:23 PM
char *time_of_day::get_time_string(void)
{
switch (format) {
case 2: sprintf(out_string, "%2d:%02d", hour, minute);
        break;
case 3: if (hour == 0)
        sprintf(out_string, "12:%02d AM", minute);
        else if (hour < 12)
        sprintf(out_string, "%2d:%02d AM", hour, minute);
        else if (hour == 12)
        sprintf(out_string, "12:%02d PM", minute);
        else
        sprintf(out_string, "%2d:%02d PM", hour - 12, minute);
        break;
case 1:
        // Fall through to default so the default is also 1
        default: sprintf(out_string, "%2d:%02d:%02d",
            hour, minute, second);
            break;
}
return out_string;
}
```

It should be pointed out that three of the four overloadings actually call the fourth so that the code did not have to be repeated four times. This is a perfectly good coding practice and illustrates that other member functions can be called from within the implementation.

The example program below [USETIME.CPP] is an application that uses the time class in a rudimentary way.

```
#include <iostream.h>
#include "date.h"
#include "time.h"
void main()
{
date today;
time_of_day now, lunch(12, 15);
cout << "This program executed on " << today.get_date_string() <<
    " at " << now.get_time_string() << "\n";
cout << "We are planning lunch at " << lunch.get_time_string() <<
    " tomorrow.\n";
lunch.set_time(13);
cout << "We decided to move lunch to " << lunch.get_time_string()
    << " due to a late meeting.\n";
}
```

It will be to your advantage to completely understand the practical example programs given at the end of the last section and the end of this section. As mentioned above, we will use the time class and the date class as the basis for both single and multiple inheritance in the next Module.

Exercise 8

List the potential problems to be considered when using operator and function overloading in C++.

Write a short program that uses both the date and time classes to add a basic 'time and date stamp' to simple operations (such as calendar or diary).