16

Starting Classes

In this Chapter:

- ❑ Identifying Classes and Objects
- ❑ Class Properties: Attributes and Operations
- ❑ Class Diagrams
- ❑ Creating Classes in C++
- ❑ Constructors and Destructors
- ❑ Encapsulation
- ❑ this
- ❑ Inline Methods
- ❑ Class Constants
- ❑ Static Attributes and Methods
- ❑ Overloading << and >>
- ❑ Friends
- ❑ Classes and Pointers
- ❑ Const Methods
- ❑ Classes and Arrays
- ❑ Designing Classes

Classes and Objects

Introduction

C++ is similar in many respects to its parent language C but the obvious difference is that C++ is an object-oriented language.

The idea behind object-orientation has been around for many decades now, but each language implements the concept in its own unique style. In this chapter we will discover the basic ideas behind object-oriented programming and how those ideas are implemented in C++.

What is an Object?

Real World Objects

Our lives are populated with objects. Typical examples include the digital camera, TV, cell phones, books, clouds, people, customers, etc.

Some objects, such as cameras, TVs and books have a physical existence; other objects represent roles or jobs (e.g. customer, shopkeeper) and a final group of objects are incidents or events such as a date, a traffic jam or joining a website.

One way to describe an object is to write down its characteristics and the operations that the object can perform. For example, we could describe a camera's characteristics by giving its weight, dimensions, lens type, and sensor size while its operations could be listed as take a picture, zoom, focus, set aperture, view stored images, adjust image size, delete recorded images, download images to another device, etc.

Non physical objects such as a bank account could be described in a similar way: with information such as account holder's name and address, account number, current balance and operations: list transactions, deposit money in account, withdraw money, set up a standing order.

Activity 16.1

List the physical characteristics of a beach ball and typical tasks performed on or with a beach ball.

Classes and Objects

It is important to differentiate between a general description of a group of identical objects and an individual object. For example, an architect may create a blueprint for a house and then a builder would use that design to build a physical house. In programming, the blueprint is known as a **class** while the specific item is known as an **object** (or **instance**) of that class. It's very similar to defining a `struct` and then creating a variable of that type.

Activity 16.2

Identify each of the following as either a class or an object:

a) Dogs b) Lassie c) Integers d) 27

Object-Oriented Programming

An object-oriented approach to software design views a system as containing two main elements: a collection of objects and the relationships between these objects.

If we take the example of a User Interface (UI), the design will suggest a set of objects such as buttons, edit boxes, menus, etc. In other applications objects might include things like a customer's account, a purchase order, a data file, or a character in a video game.

Once the objects required by the system have been identified, the relevant characteristics and operations of the classes to which these objects belong are defined.

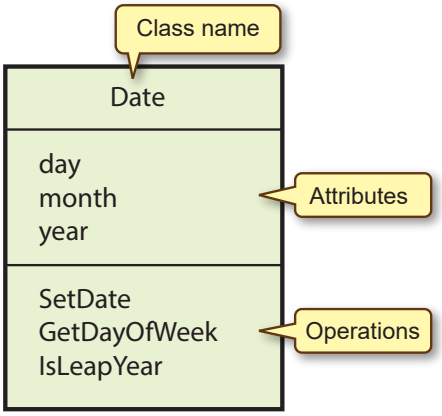
In the world of object-oriented design, a class's characteristics are known as its **attributes** while the procedures that can be performed are known as the **operations**. Collectively, the attributes and operations are known as the **features**, **properties**, or **members** of the class.

By far the easiest way to get the hang of all this is to look at a simple example. Back in Chapter 12 we created a set of functions for use with the *DateType* structure, so let's see how we could make use of this to create a class called *Date*.

Having identified our class and its properties, we can create a visual representation of this as shown in FIG-16.1.

FIG-16.1

A Basic Class Diagram



Note that for the sake of clarity, only a few of the functions we created in the earlier chapter are included above.

Activity 16.3

In the Imperial measuring system (still in use in the USA) short distances are measured in yards, feet and inches. There are 12 inches in a foot and 3 feet in a yard. So if we identified ImperialDistance *as a class within a system we were creating, we would list these three values (*yards, feet *and* inches*) as the attributes of that class. For the moment we can identify only two operations: one to set the value of the attributes (*SetDistance*) and one to convert the distance to metres (*ConvertToMetric*).*

Draw a class diagram similar to that in FIG-16.1 for a class called *ImperialDistance*.

Taking the design for *Date* further, each attribute must be defined as a C++ data type (or another class). In the case of the attributes *day*, *month* and *year*, all three of these can be defined as `int` types.

Next we need to add parameters and return types to the operations list. Since we are working exclusively in C++, we'll make use of the types available in that language. A more generic diagram might use other terms when describing attribute and operation types.

The updated class diagram is shown in FIG-16.2.

FIG-16.2

A Detailed Class Diagram

There are several things to notice in this version of the diagram:

- the type is given after the attribute or parameter name
- the type of value returned by an operation is given at the end of the operation (operations which do not return a value have no return type)
- the *DateType* parameter which was specified for each of these functions when we last used them in Chapter 12 has been removed.

It is this last point that needs to be explained in some detail.

Those of us of a certain age may remember cassette recorders. These devices performed record and play operations using magnetic tape cassettes. Just pop a tape into the machine and record or play. Replace the cassette with a new one and record or play other sounds. But without a cassette tape in place the device was useless. This is exactly how our *DateType* functions we created back in Chapter 12 worked: in order to use the functions, we had to supply a *DateType* variable (just as our recorder needed a cassette) that the function could operate on.

Jump forward to the 21st century and we now have digital recorders. Like the old cassette players, these can also record and play audio. But this time the storage is built into the device. All we do is select record or play and the internal storage is accessed automatically. This is the approach taken by classes. Classes are designed to contain both the storage requirements (attributes) and the operations that will make use of that storage. Combing these two elements into a single unit is known **encapsulation**.

This means that when an operation accesses *day*, *month* or *year*, it automatically assumes that this refers to the *day month* and *year* attributes held within that particular object. The attributes of a class are said to have **class scope**, meaning they can be freely accessed by any operation in the same class.

Activity 16.4

Add type details to your *ImperialDistance* class diagram.

A class diagram might also specifically state the range of values which an attribute may have. For example, *day* must always have a value in the range 1 to 31 while *month* must be between 1 and 12. These would be shown in the class diagram as:

```
day     : {1..31}
month   : {1..12}
year    : int
```

Activity 16.5

In your *ImperialDistance* class diagram, specify ranges for *inches* and *feet*.

The operations of a class can be specified using a set of mini-specs in a similar fashion to those we created in earlier chapters. This time we would also list any attributes of the class which are read from or written to by the operation. An example of an operation's mini-spec is given in FIG-16.3.

FIG-16.3

A Mini-Spec for the *SetDate* Operation

Class	: Date
Operation	: SetDate
Parameters	
In	: d : int
	m : int
	y : int
Out	: None
Attributes	
Read	: None
Written	: day, month, year
Pre-condition	: *d,m,y* forms a valid date
Post-condition	: *day* = *d*
	month = *m*
	year = *y*
Description	: Sets the date to *d/m/y*.

Activity 16.6

Create a mini-spec similar to that shown above for the *SetDistance* operation in the *ImperialDistance* class.

For some classes it is more appropriate to have a separate *Set* operation for each attribute, but when those attributes are closely linked such as those in a distance, a date, or a time object, then it is quite valid to have a single *Set* operation such as *SetDate* or *SetDistance* set the value of more than one attribute.

Classes in C++

Actually, classes can also be constructed using the term `struct`, but we'll ignore that option here.

Our design is now ready to be coded. In C++ a class's definition begins with the keyword `class`. We can think of this definition as a blueprint for later variables or objects. In this respect, a `class` definition is similar to that of a `struct`.

The keyword `class` is followed by the class name (e.g. *Date*, *ImperialDistance*). Although there are no fixed rules, generally class names start with a capital letter and if the name is constructed from several words, each new word starts with a capital. Underscores are not normally used in a class name.

After the class name, enclosed in braces, are an **access specifier**, the attributes and operations of the class. The operations are given in the form of function prototypes.

A first attempt at a definition of the *Date* class is given in below:

```
class Date
{
   public:
      int day;
      int month;
      int year;

      void SetDate(int,int,int);
      int GetDayOfWeek();
      bool IsLeapYear();
};
```

Let's look at the purpose of each section of the code:

`class Date`	This tells the compiler we are declaring a class and gives the name of the class. By convention, class names are capitalised.
`public:`	This keyword is the access specifier and tells the compiler that all the following properties of the class may be freely accessed by any function - even functions which are not part of the *Date* class.

`int day;`
`int month;`
`int year;`
	These are the attributes of the class and are similar to fields within a record.

`void SetDate(int,int,int);`
`int GetDayOfWeek();`
`bool IsLeapYear();`
	These are the prototypes for the various functions that will be used to implement the operations of the class.

Activity 16.7

Start a new project called *DateClass* (with `#include` and `using` statements) and implement the code for class *Date* as shown above. Save your code.

After the class declaration, we add the code for each of the class operations. Since the code for the class operations is not placed within the braces of the class declaration, it is necessary to tell the compiler to which class a function belongs. This is done in the first line of the function. For example, the first line of *SetDate()* is coded as

```
void Date::SetDate(int d, int m, int y)
```

Notice that this differs from a normal function heading only in that the class name and scope resolution operator (`Date` and `::`) precedes the function name. The complete code for the *Date* class is shown in FIG-16.4.

FIG-16.4

Implementing the Date Class

```
#include <iostream>
using namespace std;

class Date
{
```

FIG-16.4
(continued)

Implementing the Date Class

```
   public:
      int day;
      int month;
      int year;

      void SetDate(int, int, int);
      int GetDayOfWeek();
      bool IsLeapYear();
};


//*************************************
//***        Date Class Methods      ***
//*************************************

// *** Sets the date to d/m/y ***
void Date::SetDate(int d, int m, int y)
{
   // *** Days in each month of year ***
   int daysinmonth[] = {0, 31,28,31, 30,31,30, 31,31,30, 31,30,31};
   //*** If month invalid, exit ***
   if (m < 1 || m > 12)
      return;
   //*** Add 1 to days in February if leap year ***
   daysinmonth[2] += (y%400 == 0)||(y%4 == 0 && y%100 != 0);
   //*** If days in month invalid, exit ***
   if (d <1 || d > daysinmonth[m])
      return;
   //*** If year less than 1, exit ***
   if (y < 1)
      return;
   //*** Assign date ***
   day = d;
   month = m;
   year = y;
}


// *** Returns the day of the week of a date ***
int Date::GetDayOfWeek()
{
   int M, modifiedyear, C, Y;
   //*** Calculate M ***
   M = (month + 9) % 12 + 1;
   //*** Calculate modified year ***
   modifiedyear = year - M / 11;
   //*** Calculate C ***
   C = modifiedyear / 100;
   //*** Calculate Y ***
   Y = modifiedyear % 100;
   //*** Calculate day of week ***
   int dayofweek = ((static_cast<int>(2.6 * M - 0.2) + day + Y +
   ↳Y/4 + C/4 - 2*C)%7+7)%7;
   return dayofweek;
}


// *** Returns true if the date's year is a leap year ***
bool Date::IsLeapYear()
{
   return ((year%400 == 0)||(year%4 == 0 && year%100 != 0));
}
```

We can't call *IsLeapYear()* to check for a leap year since that method checks the value stored in the *Date* attribute *year* whereas we wish to check the parameter *y*.

The coded version of a class operation is referred to as a **method**. Think of the code as the method by which we implemented the operation.

Activity 16.8

Modify your *DateClass* project to match the code given above. To get the code to compile, add the following line to the end of your program:

```
int main(){}
```

Check that your program compiles.

There are 2.54 cm in an inch.

Activity 16.9

Start a new project called *ImperialDistanceClass* and create code for the *ImperialDistance* class operations using the *Date* class as a guideline.

Compile and save your program.

In a large software project our job might be over at this point after we have created (and tested) a class. It would be other programmers who make use of objects of this class when writing their own section of the project. We can identify these two groups of programmers as the **class programmers** and the **application programmers**.

The application programmer uses objects from one or more classes, created by the class programmer, to implement the application design.

One goal of the class designer/programmer is to create a sufficient range of error-free methods within a class to allow the application programmer to easily achieve the results required by their project. For example, the present version of the *Date* class would be of little use to an application programmer since it does not allow for basic operations such as adding days to a date, finding out the number of days between two dates, or comparing two dates for equality.

A second goal of the class programmer is to have written a class in such a way that the application programmer cannot mishandle objects of that class. We'll see exactly what this means in a moment.

Normally, the application programmer will not have access to the source code for a class but will, instead, have a description of the public operations of that class. This is exactly the situation we met when using C++'s standard functions such as `sin()`, `cos()`, etc: we had no access to the original code, but nevertheless we have sufficient knowledge of the purpose, parameters and return value of each function to call the function appropriately.

If our mythical application programmer requires a *Date* class object, this is created like any other variable:

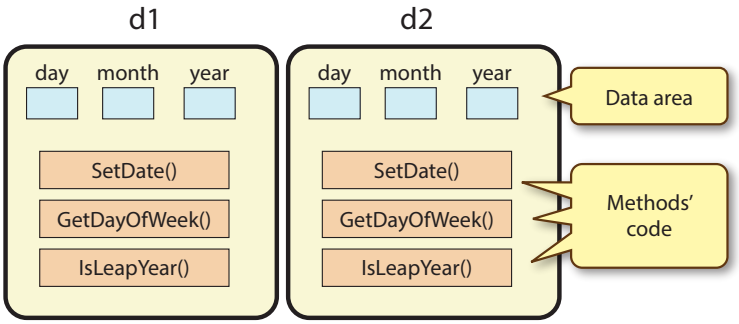
```
Date d1, d2;
```

d1 and *d2* are now objects of class *Date*.

Unlike a standard `struct` construct, the objects *d1* and *d2* contain not only data fields, but also their own copies of the routines defined for the class (see FIG-16.5).

FIG-16.5

A Visual Representation of Two *Date* Objects



Within the methods' code we will see that the properties *day*, *month* and *year* are accessed in exactly the way a standard function might access global variables (they have class scope). This is because the attributes of any class are always freely available to the methods of that class.

The methods in each object are designed to operate only with the attributes of that specific object. So, when we execute *SetDate()* in object *d1*, it is the contents of the *day*, *month* and *year* attributes in *d1* that will be modified.

The application programmer can access properties embedded in each object using the same notation as we did with record structures. So expressions such as

`d1.day` and `d2.month`

give access to a data element in each object while expressions such as

`d1.SetDate()` and `d2.GetDayOfWeek()`

give access to methods in each object.

The term

`d1.GetDayOfWeek()`

will return the day of the week of the date stored within object *d1*; the term

`d2.GetDayOfWeek()`

will return the day of the week of the date stored within object *d2*.

The program in FIG-16.6 demonstrates how the properties of an object are used to set a date and display which day of the week the date falls on.

FIG-16.6

Using a Date Object

```
#include <iostream>
using namespace std;

class Date
{
   public:
      int day;
      int month;
      int year;

      void SetDate(int, int, int);
      int GetDayOfWeek();
      bool IsLeapYear();
};
```

FIG-16.6
(continued)

Using a Date Object

```
//************************************
//***        Date Class Methods      ***
//************************************

// *** Sets the date to d/m/y ***
void Date::SetDate(int d, int m, int y)
{
   // *** Days in each month of year ***
   int daysinmonth[] = {0, 31,28,31, 30,31,30, 31,31,30, 31,30,31};
   //*** If month invalid, exit ***
   if (m < 1 || m > 12)
      return;
   //*** Add 1 to days in February if leap year ***
   daysinmonth[2] += (y%400 == 0)||(y%4 == 0 && y%100 != 0);
   //*** If days in month invalid, exit ***
   if (d <1 || d > daysinmonth[m])
      return;
   //*** If year less than 1, exit ***
   if (y < 1)
      return;
   //*** Assign date ***
   day = d;
   month = m;
   year = y;
}

// *** Returns the day of the week of a date ***
int Date::GetDayOfWeek()
{
   int M, modifiedyear, C, Y;
   //*** Calculate M ***
   M = (month + 9) % 12 + 1;
   //*** Calculate modified year ***
   modifiedyear = year - M / 11;
   //*** Calculate C ***
   C = modifiedyear / 100;
   //*** Calculate Y ***
   Y = modifiedyear % 100;
   //*** Calculate day of week ***
   int dayofweek = ((static_cast<int>(2.6 * M - 0.2) + day + Y +
   ↳Y/4 + C/4 - 2*C)%7+7)%7;
   return dayofweek;
}

// *** Returns true if the date's year is a leap year ***
bool Date::IsLeapYear()
{
   return ((year%400 == 0)||(year%4 == 0 && year%100 != 0));
}


int main()
{
   // *** Day names ***
   char daynames[7][10] = {"Sunday","Monday","Tuesday",
   ↳"Wednesday","Thursday","Friday","Saturday"};

   Date d1; // Date object

   // *** Set date ***
   d1.SetDate(23,11,1963);
```

FIG-16.6
(continued)

Using a Date Object

```
    // *** Display date details ***
    cout << d1.day << '/' << d1.month << '/' << d1.year <<
    ↳" was a " << daynames[d1.GetDayOfWeek()] <<endl;
}
```

Activity 16.10

Modify *DateClass* to match the code in FIG-16.6 and observe the output produced.

Although we have a class that will operate successfully as long as the application programmer makes valid calls to the class methods, there are a few problems with this code.

The first problem is that should the application programmer attempt a call to the `SetDate()` method using invalid parameters, then the contents of the attributes *day*, *month* and *year* will remain undefined.

Activity 16.11

In *DateClass,* change the call to *SetDate()* so that the parameters are 29, 2, and 2021. How does this affect the display when the program is run?

Activity 16.12

Modify *ImperialDistanceClass* creating an object of the class, assigning it a valid distance and then displaying the equivalent distance in metres.

What happens if an invalid distance (e.g. 2,3,10) is assigned to the object?

Overloading Methods

We saw, when creating traditional functions back in Chapter 12 that we could overload functions. That is, we could create multiple functions with the same name as long as their parameter lists differed in some way. We can also do this with the methods of a class.

To demonstrate overloaded methods we'll create a second version of *SetDate()*. This new version takes two integer values. The first of these is the number of days into the year (1 to 365/366). The second parameter is the year. For example, 32,2021 would represent 1/2/2021 (32 days into the year 2021).

The method's logic creates a date of the form *d/m/y* by subtracting the days in each month until we end up with a zero or negative value and then adding back the number of days in the last month to have been deducted. The methods code is:

```
// *** Sets date using days-into-year (diy) and year (y) ***
void Date::SetDate(int diy, int y)
{
    // *** Days in each month of year ***
    int daysinmonth[] = { 0, 31,28,31, 30,31,30, 31,31,30, 31,
    ↳30,31 };
    //*** If leap year add one day to February ***
    daysinmonth[2] += (y % 400 == 0) || (y % 4 == 0 &&
```

```
      y % 100 != 0);
      //*** Calculate month ***
      int remaining = diy;
      int m = 0;
      while (remaining > 0)
      {
         m++;
         remaining -= daysinmonth[m];
      }
      remaining += daysinmonth[m];
      //*** Set date ***
      SetDate(remaining, m, y);
}
```

Activity 16.13

In *DateClass,* add the second version of *SetDate()* and test this new method with the values 60,2020, and 60,2021.

Constructors and Destructors

Constructors

When an object is first created a special method known as a **constructor** is executed. A default constructor is automatically created by C++ for every class, but this default constructor is of little use since it contains no code! Luckily, we can replace the default constructor with a version of our own.

Typically, a coded constructor is used to initialise each of the attributes of an object to a specific value. This will solve the problem we encountered in the last two Activities.

A constructor method must be given the same name as the class and have no return type (not even `void`) but other than that it is prototyped and defined in the same way as any other method in a class.

A constructor for the *Date* class which sets *day*, *month* and *year* to 1/1/2001 would have the prototype

```
Date();
```

This is known as a **zero-argument constructor** since it requires no parameters.

and be coded as

```
Date::Date()
{
   day = 1;
   month = 1;
   year = 2001;
};
```

The coded constructor will automatically replace the default one created by the compiler.

Activity 16.14

Modify *DateClass,* so that the *Date* class contains the constructor code given above. What happens this time when you call *SetDate()* with parameters 29, 2, and 2021?

Activity 16.15

Modify *ImperialDistanceClass,* so that *ImperialDistance* class contains a constructor which sets the distance to zero.

What happens this time when you call *SetDistance()* with parameters are 2, 3, and 10?

Overloading Constructors

Like other methods of a class, the constructor can be overloaded. For example, we could create a second *Date* class constructor which takes three `int` parameters and assigns to the *Date* object being created.

This would require the prototype (in the class code)

```
Date(int,int,int);
```

and the code

```
Date::Date(int d, int m, int y)
{
   // *** Days in each month of year ***
   int daysinmonth[] = {0, 31,28,31, 30,31,30, 31,31,30,
   ↳31,30,31};

   bool valid = true;  // Parameters valid

   //*** If month invalid, invalid parameters ***
   if (m < 1 || m > 12)
      valid = false;
   //*** Add 1 to days in February if leap year ***
   daysinmonth[2] += (y%400 == 0)||(y%4 == 0 && y%100 != 0);
   //*** If days in month invalid, invalid parameters ***
   if (d <1 || d > daysinmonth[m])
      valid = false;
   //*** If year less than 1, invalid parameters ***
   if (y < 1)
      valid = false;
   //*** Assign date ***
   if (valid)
   {
      day = d;
      month = m;
      year = y;
   }
   else
   {
      day = 1;
      month = 1;
      year = 2001;
   }
}
```

As we can see, the code is similar to that for the first version of *SetDate()*. However, rather than exit from the routine when invalid parameters are offered, the method sets the date attributes to 1/1/2001 rather than leaving them undefined.

In `main()` we could make use of the new constructor when declaring *d1* as a *Date*

object with the line:

```
Date d1(23,11,1963);
```

When a *Date* object is declared in `main()`, with the line

```
Date d1;
```

this will still cause the zero-argument constructor to be executed and the date set to 1/1/2001.

Activity 16.16

Modify *DateClass,* adding the second constructor given above to the *Date* class.

Test the new constructor by initialising *d1* to 23/11/1963 and checking that the appropriate value is assigned.

We can even make use of default values for the parameters to a constructor (and other methods in a class). For example, if we rewrote the prototype for the second *Date* constructor to read

```
Date(int=1,int=1,int=2001);
```

we end up removing the need for a coded zero-argument constructor since this will be performed automatically:

```
Date d1;          // Uses default values to set d1 to 1/1/2001
Date d2(23,11,1963); // Overrides default values to set d2 to
↳23/11/1963
```

Activity 16.17

Modify *DateClass,* so that the zero-argument constructor can be removed.

Check that the default value works to create a *Date* object set to 1/1/2001.

Activity 16.18

Modify *ImperialDistanceClass,* adding a three-argument constructor (`int, int, int`) with default values 0,0,0.

Test the constructor, checking that the appropriate value is assigned to any *ImperialDistance* object created.

The Copy Constructor

C++ creates not one but two default constructors. We've already seen that the first default constructor does not perform any task, but the second constructor – known as the **copy constructor** – allows a new object to be initialised with a copy of an existing object of the same type.

For example, if we have previously created a *Date* object with the line

```
Date d1(1,1,2001);
```

then we can create a new *Date* object containing a copy of the first date using one of the following lines

```
Date d2{d1};
Date d2(d1);
Date d2 = d1;
```

Each of these options makes use of the copy constructor to copy the values held in the attributes of *d1* into the corresponding attributes of *d2*.

The copy constructor's code for the *Date* class is equivalent to:

```
Date::Date(const Date& d)
{
   day = d.day;
   month = d.month;
   year = d.year;
};
```

Note that the parameter to the copy constructor is a `const` reference. This is a requirement of all copy constructors. For example, if we were to write a copy constructor for the *ImperialDistance* class, its first line would be

```
ImperialDistance::ImperialDistance(const ImperialDistance& d)
```

As with the zero-argument constructor, we are free to override the copy constructor but for simple classes, where none of the attributes are pointers, this is usually unnecessary.

Activity 16.19

In *DateClass*, check out the copy constructor of the *Date* class by first creating a *Date* object, *d1*, set to 23/1/2001 and then creating a new *Date* object, *d2*, which is initialised with a copy of the values in *d1*. The value of *d2* should be displayed.

The Class Assignment Operator

Although not a constructor, this is an appropriate point to mention the other piece of code that C++ generates for free when we create a class - the assignment operator.

If we have two class objects, then we can copy the contents from one object to the other using the standard assignment operator:

Here we are assuming *d1* and *d2* are *Date* objects.

```
d1 = d2;
```

This default assignment operator, like the default copy constructor, copies the contents of every attribute (in this case, *day*, *month* and *year*) from the right-hand object (*d2*) to the corresponding fields in the left-hand object (*d1*).

For regular type attributes this logic will normally be fine, but when any of the attributes involved are pointers, then we may want to override this version of the assignment operator with our own code. We'll see an example of this later.

Note that in the following example, the second line will use the copy constructor while the third makes use of the assignment operator:

```
Date d1(23,11,1963), d2;
Date d3(d1);      //Copy constructor
d2 = d1;          //Assignment operator
```

Even if we think we are using the assignment operator as in the second line below

```
Date d1(26,10,1961);
Date d2 = d1;
```

we're not! Any assignment to an object in the line in which that object is declared always makes use of a constructor.

Typecasting and Constructors

As we saw in an earlier chapter, C++ can automatically convert values from one type to another. So the line

```
double x = 21;
```

will convert the `int` value 21 to a `double` before storing that value in *x*.

The same automatic conversion is possible with class objects. For example, if we were to write

```
Date d1;
d1 = 21;
```

then the `int` value 21 would automatically be converted to a date.

Activity 16.20

In *DateClass*, modify *main()* so that *d2* is assigned the value 21 after previously having been created holding a copy of *d1*. What value is displayed?

The reason for the result obtained by Activity 16.20 is that when presented with a value which is not of the same class as the object being assigned the value, C++ examines the constructors available for that class to see if any of these can accept the value being assigned as a valid parameter. If it can, then that constructor is executed and the *Date* object created is used in the assignment. In the case of the statement above, the *Date* class has a constructor whose prototype is

```
Date(int=1, int=1, int=2001);
```

Since the parameters have default values, when presented with a single value, 21, C++ uses this as the first parameter to the constructor and uses the default values (1 and 2001) for the remaining two parameters.

We can also use a class constructor explicitly along with the class assignment operator to assign a value to an object. Hence,

```
Date d1;
d1 = Date(23,11,1963);
```

creates an unnamed *Date* object and assigns its value to *d1*.

explicit

The fact that C++ will make use of a constructor to automatically convert a value to an object of a class is not always desirable. For example, it's a bit of a stretch to think that a program converting the value 21 to the date 21/1/2001 would be acceptable.

To control this sort of situation, we can tell our constructor not to indulge in these automatic conversions by adding the keyword `explicit` at the start of a constructor's prototype. In the case of the *Date* class constructor, this would mean that the prototype

would now read

```
explicit Date(int = 1, int = 1, int = 2001);
```

There is no need to make any changes to the constructor's definition.

Now, if we were to attempt the line

```
d2 = 21; //d2 is a Date object
```

the compiler would throw up an error. The only way to have this line accepted is to explicitly cast 21 to a *Date* object:

```
d2 = static_cast<Date>(21);
```

With this explicit request to convert 21 to a *Date* object, the compiler will again make use of the constructor to perform the conversion from `int` to *Date*.

Activity 16.21

Modify *DateClass,* to create an explicit constructor. What error do you get when you attempt to convert 21 automatically to a date?

Change the assignment line to explicitly convert 21 to a *Date* object. What happens when you compile and run the program this time?

Destructors

A **destructor** is a function whose code is automatically executed when an object has its space deallocated. Again, C++ automatically creates a class's destructor, but we can override this by creating our own.

A class's destructor is named after the class to which it belongs. However, to distinguish it from the constructor, the name must begin with a tilde (~) character. A destructor cannot return a value nor take any parameters.

We are only likely to need a destructor when a class contains a raw pointer attribute which references dynamically allocated space (smart pointers will handle the release automatically). Other reasons a class may require a destructor is when it needs to release some resource such as a file, or internet connection.

In these situations the destructor's code would free up the space referenced by the raw pointer, close the file, or disconnect from the internet. We'll see an example of a class which makes use of dynamic space in the next chapter.

For the moment, let's create a trivial destructor for the *Date* class which simply displays a "*Date object deleted*" message.

This requires the class to include the following prototype

```
~Date();
```

and the code

```
Date::~Date()
{
   cout << "Date object deleted\n";
};
```

Activity 16.22

Modify *DateClass,* so that the *Date* class includes the destructor coded above.

Run the program. Is the destructor message displayed?

After testing, remove the destructor's code from the *Date* class.

Data Hiding

A problem with the *Date* class is that the application programmer is free to bypass the `SetDate()` method entirely and just set the values held in attributes *day*, *month* and *year* directly with code such as

```
Date d1;
d1.day = 2
d1.month = 1;
d1.year = 1950;
```

If they use this approach, then there are no safeguards to stop an invalid value being stored. For example the line

```
d1.month = 13;
```

will execute without a complaint and any other methods which assume the date is valid will return unpredictable results.

To overcome this problem we can restrict the application programmer's access to some of the properties in a class. For example, by stopping access to *day*, *month* and *year*, we remove any chance of invalid values being stored within those attributes.

Hiding properties in this way is known as **data hiding**.

private

The application programmer has direct access to the attributes *day, month* and *year* in any *Date* objects he creates because those attributes are marked as `public` in the class definition. However, properties can also be labelled as `private`. Private properties in objects created by the application programmer's code cannot be accessed directly but may still be accessed via the methods of the class.

So to implement this change in *Date* we re-code the class declaration as

```
class Date
{
   private:
      int day;
      int month;
      int year;
   public:
      Date(int=1, int=1, int=2001);
      void SetDate(int,int,int);
      void SetDate(int,int);
      int GetDayOfWeek();
      bool IsLeapYear();
};
```

With this new version of the class, the application programmer can no longer gain

direct access to the three attributes of the class. Attempts to use statements such as

```
d1.month = 2;
```

will now fail to compile. If an object's attributes are to have their contents changed, then the `SetDate()` method must be employed:

```
d1.SetDate(23,11,1963);
```

Activity 16.23

Modify *DateClass,* so that the attributes of the class are private. What happens when you try to compile the program?

And now the side-effect of marking attributes as `private` becomes apparent. Not only have we stopped the application programmer from storing values directly in *day*, *month* and *year*, but we have also stopped him from discovering what values are already stored there! `private` means absolutely no access - even if we only want to look at the value held.

To get round this we need to add new functions which do nothing more than return copies of the values in the class's attributes. For example, if we want to find out what value is stored in the *days* attribute we would write a new method as shown below:

```
int Date::GetDay()
{
   return day;
}
```

With this code (and the method prototype) embedded in the class, the application programmer, who has discovered that

```
cout << d1.day;
```

causes a compilation error, can now write

```
cout << d1.GetDay();
```

Activity 16.24

Modify *DateClass,* adding new methods *GetDay()*, *GetMonth()* and *GetYear()* to the class.

Uses these new methods in `main()` to allow the contents of the *Date* object *d2* to be displayed.

It is not only attributes that can be labelled as private – methods can be too. It is only appropriate to make helper functions private. A helper function is designed only to help in the coding of some other method and not as a function we wish the application programmer to gain access to. To stop that access, the helper function should be marked as private. Going back to the functions we created for *DateType* in Chapter 11, *DateToJDN()* and *JDNToDate()* are good examples of helper functions since they were only needed so that the functions *AddDays()* and *DaysDifference()* could be coded.

We will start by adding these two helper functions to our *Date* class - but this time giving them slightly different names. The class declaration now reads:

```
class Date
{
```

```
   private:
      int day;
      int month;
      int year;

      long ToJDN();
      Date FromJDN(long);

   public:
      Date(int=1,int=1,int=2001);
      void SetDate(int, int, int);
      void SetDate(int,int);
      int DayOfWeek();
      bool IsLeapYear();
      int GetDay();
      int GetMonth();
      int GetYear();
};
```

And the new methods are coded as:

```
// *** Returns the JDN of the date ***
long Date::ToJDN()
{
   int a = (14 - month)/12;
   int m = month + 12*a - 3;
   int y = year + 4800 - a;
   long result = day + (153*m + 2) / 5 + 365*y + y/4 - y/100 +
   ↳y/400  - 32045;
   return result;
}

// *** Sets date to eqivalent of JDN ***
Date Date::FromJDN(long jdn)
{
   int f = jdn + 1401+(((4 * jdn + 274277)/146097)*3)/4 -38;
   int e = 4 * f + 3;
   int g = (e % 1461)/4;
   int h = 5 * g + 2;
   Date d;
   d.day = (h % 153)/5 + 1;
   d.month = (h/153 + 2) % 12 + 1;
   d.year = e/1461 - 4716 + (12 + 2 - d.month)/12;
   return d;
}
```

Notice that this second method returns a *Date* object. This is no different in implementation than our previous version of the function, *JDNToDate(),* which returned a *DateType* structure.

Activity 16.25

Add the two private functions *ToJDN()* and *FromJDN()* to *Date* class. Check that your code compiles.

Overloading Operators

All the things we could achieve with standard functions can also be done within a class. This includes overloading operators. Implementing `operator+` in the *Date* class would require the following code:

```
//*** Add d days to date ***
Date Date::operator+(int d)
{
   Date dt = FromJDN(ToJDN()+d);
   return dt;
}
```

Notice that it makes use of the two private functions in calculating the result.

As before, we have two ways of using an overloaded operator. We can treat it as just a standard method allowing the format

```
Date dt2 = dt1.operator+(7); //Add one week to date
```

or we can treat it in the same way as standard operators when dealing with normal numeric values:

```
Date dt2 = dt1 + 7; //Add one week to date
```

Activity 16.26

Add `operator+` as a public method to *Date* class. To check that your code is correct, create a new version of `main()` which sets a *Date* object *d1* to 30/5/2023 then adds 7 days to the date and displays the result.

Activity 16.27

Add the following operators to the *Date* class:

- `-` (subtraction: returns days between two dates)
- `-` (subtraction: returns a date after subtracting a number of days)
- `==` (test for equality: returns *true* or *false*).
- `!=` (test for inequality: returns true or false).

Modify `main()` to test each operation.

The Call Operator and Functors

The opening and closing parentheses (known as the **call operator** since it used whenever we call a function) can also be overloaded. The code associated with the call operator is executed when parentheses are given immediately after an existing object. We'll demonstrate this by creating a simple class A with a single feature which overloads the call operator. (see FIG-16.7).

FIG-16.7

The Call Operator

```
#include <iostream>
using namespace std;

class A
{
   public:
      void operator()();
};
void A::operator()()
{
   cout << "Call operator executed\n";
}
int main()
{
   A test;     //Create object test
   test();     //Execute the call operator
}
```

Notice that the object, *test*, begins to look a lot like a function. If we saw the program's

```
test();
```

line in isolation from the remainder of the code, we would, quite reasonably, think *test* was the name of a function rather than an object. Because of this, any class that implements a call operator method is classified as a **functor** – a class whose objects act like functions.

Like any method, the call operator can accept parameters. For example, we could change the code for class A's call operator to

```
void A::operator()(int v)
{
   cout <<"Call operator executed. Parameter value "<<v<< endl;
}
```

Activity 16.28

Start a new project called *TestCallOperator* and implement the code given in FIG-16.7.

Modify the call operator code to accept an `int` parameter and include the value of the parameter in the `cout` statement as shown above.

Modify the call parameter again to accept two parameters and to return the product of the two values. Also modify the code in `main()` to display the returned value.

Although this represents a trivial example of a functor, we'll discover in a later chapter how widely used this option is.

Inline Methods

Defining a function as inline only acts as a request to the compiler. The final decision on whether to do so is left up to the compiler.

In Chapter 12 we learned that inline functions can be used to create a more efficient program by replacing the call to a function with the function's code thereby saving the overhead of parameter passing and saving a return address. A class method can be defined as **inline** in one of two ways. The first option is to use the `inline` keyword in the method definition, the second is simply to embed the method's code within the class declaration. For example, if we were to make *IsLeapYear()* and inline function we could write

```
inline bool IsLeapYear()
{
   return ((year%400 == 0)||(year%4 == 0 && year%100 != 0));
}
```

or we could move the code for the function into the class:

```
class Date
{
   private:
      int day;
      int month;
      int year;
      long ToJDN();
      Date FromJDN(long);
   public:
      Date(int=1,int=1,int=2001);
      void SetDate(int, int, int);
```

```
    void SetDate(int, int);
    int DayOfWeek();
    bool IsLeapYear()
    {
       return ((year%400 == 0)||(year%4 == 0 &&
       ↳year%100 != 0));
    }
    int GetDay();
    int GetMonth();
    int GetYear();
    Date operator+(int);
    int operator-(Date);
    Date operator-(int);
    bool operator==(Date);
    bool operator !=(Date);
    void operator+=(int);
};
```

The *this* Pointer

Returning to the *Date* class, another operator we could overload is `+=` allowing us to add a number of days to the current date. For example, with this operator overloaded the application programmer could write lines such as

```
Date d1(23,11,1963);
d1 += 10;
```

However, when we come to code such a routine for our class, we run into an unexpected problem:

```
void Date::operator+=(int d)
{
    ??? = FromJDN(ToJDN()+d);
}
```

We want the value calculated by the method to be assigned to the *Date* object which has called this method, but how do we refer to that object since the actual object executing the routine can be different each time the operation is used? Luckily, C++ solves this problem by maintaining a pointer to whatever object is currently executing a method of a class.

The pointer is called `this` and is available for use within every standard method of a class. So, when we are writing a method within a class and wish to refer to the object which is currently executing that method, we can dereference the `this` pointer. This allows us to overcome our problem in writing the code for the `+=` operator since it can now be coded as:

```
void Date::operator+=(int d)
{
    *this = FromJDN(ToJDN()+d);
}
```

Activity 16.29

Add `operator+=` as a public method to *Date* class.

To check that the method operates correctly set a *Date* object to 23/12/1980 and add 10 days to the date, displaying the new date.

In fact, C++ makes extensive use of the `this` pointer, secretly adding it to any code we write when referencing attributes of a class within one of its methods. For example, when we write

```
bool Date::IsLeapYear()
{
   return ((year%400 == 0)||(year%4 == 0 && year%100 != 0));
}
```

The C++ compiler secretly changes this to:

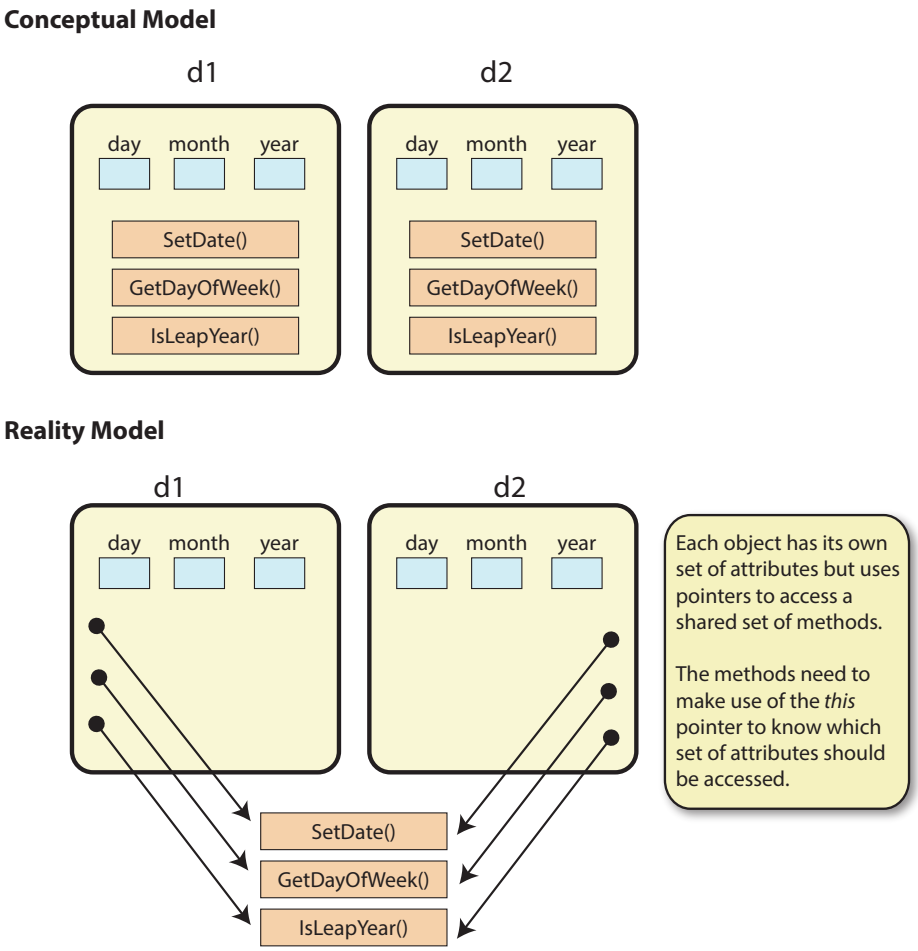
```
bool Date::IsLeapYear(Date *this)
{
   return ((this->year%400 == 0)||(this->year%4 == 0 &&
   this->year%100 != 0));
}
```

This is necessary because, although we may imagine each object having its own copy of every member defined for that class, the reality is that, in order to optimise storage requirements, each object has its own copy of every attribute but shares use of a communal copy of the code for the class's methods with every other object of that class.

So, when executing a method, some mechanism is needed to make sure the attributes of the appropriate object are accessed. This requirement is fulfilled by the `this` pointer. (see FIG-16.8).

FIG-16.8

Why the *this* Pointer is Required when Executing Methods



And although C++ automatically adds the `this->` prefix to any attribute accessed within a class's methods, we may also add this prefix explicitly without it objecting.

Activity 16.30

Modify the *IsLeapYear()* method in the *Date* class so that it explicitly uses `this ->` when accessing attributes of the class.

Check that this has no effect on the program's compilation or execution.

Return the code for *IsLeapYear()* to its original version.

We need to make use of the `this` pointer again if we want to overload the `++` and `--` operators for our *Date* class. For example, the pre-fix version of `++` would be coded as

```
Date Date::operator++()
{
   *this = FromJDN(ToJDN()+1);
   return *this;
}
```

while the post-fix version would be written as

```
Date Date::operator++(int)
{
   DateType result = *this;
   *this = FromJDN(ToJDN()+1);
   return result;
}
```

Activity 16.31

Add both versions of `++` and `--` as public methods in *Date* class. Check that all four new methods operate correctly.

Class Constants

A class declaration can contain named constants. For example, we might begin our *Date* class declaration by adding a constant for the number of days in a standard year:

```
class Date
{
   private:
      const int DAYSINYEAR = 365;
```

Activity 16.32

Add the DAYSINYEAR constant given above to the *Date* class and compile your program. What error messages are produced?

Although the compiler has not objected to the new line of code we've added, it suddenly objects to all the assignment statements that attempt to copy one *Date* value to another.

When we copy the contents of one *Date* object to another as in the line

```
Date dt1(23,11,1963),dt2;
dt2 = dt1;
```

every attribute of *dt1* is copied to the corresponding attribute of *dt2*. However, when a class contains a constant value, this approach is no longer appropriate since we cannot attempt to assign a constant a value (even if it is the same value as it already contains).

Because of this, we are forced to define a replacement version of the `=` operator within the *Date* class so that no attempt to copy the `const` value is included. Such an operation has the prototype

```
Date& operator=(const Date&);
```

and is coded as

```
//*** Copies d to current object ***
Date& Date::operator=(const Date& d)
{
   day = d.day;
   month = d.month;
   year = d.year;
   return *this;
};
```

Notice that the method is designed to return a reference to the object being assigned the new value. Although not necessary for simple statements such as

```
dt1 = dt2;
```

the return value is required for more complex statements such as

```
dt1 = dt2 = dt3;
```

Activity 16.32

Add the necessary code for the = operator to the *Date* class. Does the program compile?

Add another method to the *Date* class called *GetDaysInYear()* which returns the number of days in the year (using the value held in *year* to determine if it's a leap year). The method should make use of the DAYSINYEAR constant.

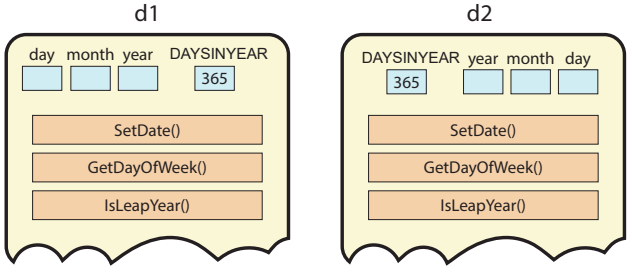
Modify `main()` to display the number of days in the year of the date within object *d1*.

Static Properties

When a standard `const` value is defined within a class, every object of that class acquires its own copy of that `const` (see FIG-16.9).

FIG-16.9

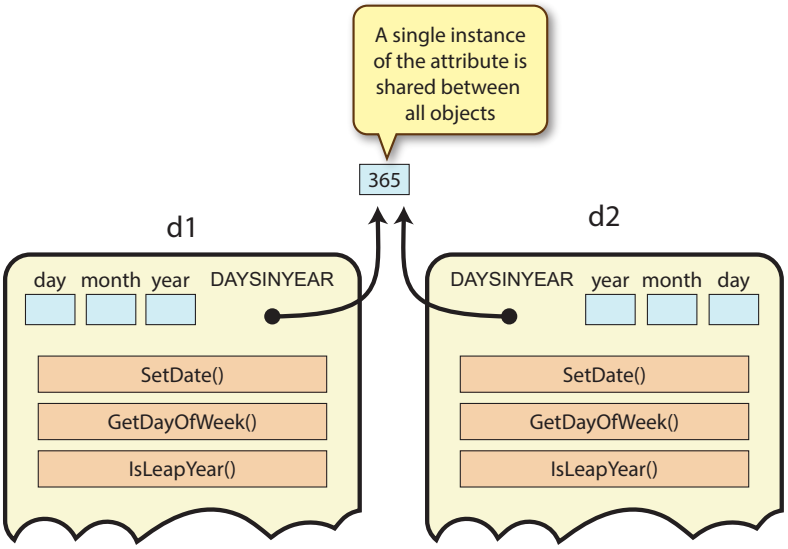
How a Class `const` Value is Stored



This situation is rather inefficient and also unnecessary. We've already seen how objects share the same method code for reasons of efficiency. We can do the same thing with constant values declared within a class, creating only a single instance of that constant and sharing it between all objects of that class (see FIG-16.10).

FIG-16.10

How a Class **`static const`** Value is Stored



To achieve this effect we must declare the constant as a **`static const`** as in the line

```
class Date
{
   private:
      static const int DAYSINYEAR = 365;
```

Another advantage of using a static constant is that it removes the need to redefine the = operator for the class since the default assignment operator code has no effect on static attributes.

If we change the declaration of the *DAYSINYEAR* attribute making it public, and then create two *Date* class objects, we could access the attribute in the usual fashion:

```
Date d1, d2;
cout << d1.DAYSINYEAR << endl;
cout << d2.DAYSINYEAR << endl;
```

Static properties of a class have another strange property: they come into existence the moment a program begins execution. This means they can exist before any objects of that class are declared:

```
int main()
{
   // DAYSINYEAR exists now
   cout << "Hello world\n";
   int no = 12;
   Date d1; //Date object exists now
      ...
```

To allow the application programmer to access the static access before creating a *Date* object, the following syntax can be used:

```
class_name::static_value_name
```

As we can see, the class name is used rather than an object name and the scope resolution operator replaces the normal member selector. So, if we wanted to access DAYSINYEAR before *Date* objects *d1* or *d2* are created, we would use the expression

```
Date::DAYSINYEAR
```

Activity 16.33

Change *Date* class's DAYSINYEAR to a public static constant then remove the prototype and definition of the `=` operator from the *Date* class.

Write a `main()` function which first displays the value held in DAYSINYEAR using the term *Date::DAYSINYEAR* and then declares a *Date* object called *d1* before displaying the value in *DAYSINYEAR* for a second time using the terms *d1.DAYSINYEAR*.

If we were to return the DAYSINYEAR class constant to private status, the application programmer could only gain access to its constants if a public *get* method is included to retrieve the constant's value.

Activity 16.34

Change *Date* class's *DAYSINYEAR* to a private static constant.

Add a public method called *GetDAYSINYEARconst()* which returns a copy of the value in *DAYSINYEAR*.

In `main()`, declare a *Date* object called *d1* and then attempt to display the value held in *DAYSINYEAR* using the terms *d1.GetDAYSINYEARconst()* and *Date::GetDAYSINYEARconst()*.

The trouble now is that although it is fine to call the new method when accessed via a *Date* object as in the expression `d1.GetDAYSINYEARconst()`, it is not valid to attempt to call it using the class name. This means that although the static constant DAYSINYEAR exists before any *Date* object is created, there is no way to access it until such an object is brought into existence.

Static Methods

Methods, as well as attributes can be defined as static. When this is done, the method, like the attribute, exists even before any objects of that class are declared. One reason for creating a static method is to allow access to private static attributes before any class objects are created. For example, if we change the prototype for *Date*'s *GetDAYSINYEARconst()* to

The term `static` is not required in the function's heading.

```
static int GetDAYSINYEARconst();
```

then, now that we have a static method, we are free to use the expression

```
Date::GetDAYSINYEARconst()
```

in our program.

Activity 16.35

Add the word `static` to the start of *Date* class's Get*DAYSINYEARconst()* prototype.

Gaining access to private static attributes is not the only reason to use static methods. Sometimes we want to collect together a set of functions which have a common

theme (just as we might sit all our programming books on the same shelf).

These functions are not designed to manipulate a common set of attributes but are placed together to give a neat and organised approach. For example, the Java programming language holds all of its maths functions within a class called *Math*.

In this *Math* class we will find functions to perform such mathematical operations as square root, sine, cosine, tangent, raise to a power, etc. Now, if we want to find, say, the square root of a number, it wouldn't make sense to have to create a *Math* object in order to access the square root method. Instead, since all the methods are static, we would just write the Java equivalent to:

```
cout << Math::Sqrt(25.78) << endl;
```

Static methods are also referred to as **class methods**.

Since a static method can be called without an object of its class being created, such methods cannot access standard (non-static) attributes of the class.

Static Variables

It's not just constant values that can be declared as static. Regular attributes can also be static, meaning that the only one instance of that property is created irrespective of the number of objects that exist.

One common use for such an attribute is to keep a count of the number of objects of a class that currently exist. For example, let's assume we wish to keep a count of the number of *Date* objects that exist in a program. We would start by adding a new static attribute and operation to the class:

```
class Date
{
   private:
      static const int DAYSINYEAR = 365;
      static int count;
      int day;
      int month;
      int year;

      long ToJDN();
      Date FromJDN(long);

   public:
      Date(int=1,int=1,int=2001);
      void SetDate(int, int, int);
      void SetDate(int, int);
      int GetDayOfWeek();
      inline bool IsLeapYear();
      int GetDay();
      int GetMonth();
      int GetYear();
      Date operator+(int);
      int operator-(Date);
      Date operator-(int);
      bool operator==(Date);
      void operator+=(int);
      Date operator++();
      Date operator++(int);
      Date operator--();
      Date operator--(int);
```

```
        int GetDaysInYear();
        static int GetDAYSINYEARConst();
        static int GetCount();
        ~Date();
};
```

In the *Date* class constructor we will increment *count:*

```
//*** Initialises new date object to d/m/y ***
Date::Date(int d, int m, int y)
{
   // *** Days in each month of year ***
   int daysinmonth[] = {0, 31,28,31, 30,31,30, 31,31,30,
   ↳31,30,31};

   bool valid = true;  // Parameters valid

   //*** If month invalid, invalid parameters ***
   if (m < 1 || m > 12)
      valid = false;

   //*** Add 1 to days in February if leap year ***
   daysinmonth[2] += (y%400 == 0)||(y%4 == 0 && y%100 != 0);

   //*** If days in month invalid, incalid parameters ***
   if (d <1 || d > daysinmonth[m])
      valid = false;

   //*** If year less than 1, invalid parameters ***
   if (y < 1)
      valid = false;

   //*** Assign date ***
   if (valid)
   {
      day = d;
      month = m;
      year = y;
   }
   else
   {
      day = 1;
      month = 1;
      year = 2001;
   }
   //*** Add one to count of Date objects **
   count++;
}
```

and in the destructor, we will decrement *count:*

```
//*** Class destructor ***
Date::~Date()
{
   //*** Decrement count of Date objects ***
   count--;
}
```

The other new method, *GetCount()* returns the value held in *count*:

```
//*** Returns the number of Date objects existing ***
int Date::GetCount()
{
   return count;
}
```

There's one last thing we need to do and that is to set the initial value for our new static attribute. This is done using the line

```
int Date::count = 0;
```

which is traditionally placed immediately after the class declaration.

Activity 16.36

Change the code for *Date* class to include the attribute *count* as defined above along with the *GetCount()* method.

Modify the constructor to increment *count* and create a destructor which decrements *count*.

Test the new class using the following code:

```
int main()
{
   cout << "There are " << Date::GetCount() << " Date objects\n";
   Date dt1(23,11,1963);
   cout << "There are " << Date::GetCount() << " Date objects\n";
   {
      Date dt2;
      cout << "There are "<<Date::GetCount() << " Date objects\n";
   }
   cout << "There are " << Date::GetCount() << " Date objects\n";
}
```

Overloading << and >>

By now you must be tired of always writing code such as

```
cout << d1.GetDay() << '/' << d1.GetMonth() << '/' <<
↳d1.↳GetYear() << endl;
```

How much nicer it would be to simply write

```
cout << d1 << endl;
```

and get the same result.

To do this, we need to overload the `<<` operator.

Our first guess at how to do this might be to add the overloaded operator to the *Date* class in just the same way as we added previous operators. But unfortunately, things are a little more complicated than that.

We'll be covering the ostream class in detail in a later chapter.

We have made use of the command `cout` since the very first program back in Chapter 3, so perhaps it will come as something of a surprise to realise that `cout` isn't a command at all, but is in fact an object of the class `ostream`. When we write a line of code such as

```
cout << 12;
```

we are actually making use of the `<<` operator as defined within the `ostream` class. We could just as validly rewrite the line as

```
cout.operator<<(12);
```

This should make it clearer that if we want to write a line such as

```
cout << d1; //d1 is a Date object
```

which could be rewritten as

```
cout.operator<<(d1);
```

that it is actually the code for the `ostream` class that needs to be changed to accommodate outputting a *Date* class object and not, as we first thought, the *Date* class.

Now, since we do not have access to the source code for the `ostream` class, modifying its code is not an option. This means we have to resort to creating a standard function, which is not part of any class, to overload the `<<` operator.

The function will take an `ostream` object reference and a *Date* object reference as its parameters:

```
void operator<<(ostream& ct, Date& d)
```

Our new function has no access privileges to private properties of either class so its code needs to make use of methods such as *GetDay()* to access the required attributes.

The code for the complete routine is

```
void operator<<(ostream& ct, Date& d)
{
   ct << d.GetDay() << '/'<< d.GetMonth() << '/'<< d.GetYear();
}
```

Now, rather than write the correct but unwieldy

```
operator<<(cout,d1)  //Assuming d1 is a Date object
```

we can write

```
cout << d1;
```

The function isn't quite complete yet, because it needs to return an `ostream&` value to allow it to be used in longer output statements. For example, if we wanted to write

```
cout << d1 << d2;
```

there would be a problem. This is more clearly seen if we rewrite the statement as

```
operator<<(operator<<(cout,d2),d1);
```

From this we can see that the value returned from the inner function call is required as the first parameter to the outer function call. To achieve the required results, our function needs to return a value of type `ostream&`. The updated version of the function is

```
ostream& operator<<(ostream& ct, Date& d)
{
   ct << d.GetDay() << '/'<< d.GetMonth() << '/'<< d.GetYear();
   return ct;
}
```

Friends

Writing functions which are not part of the *Date* class and yet are obviously logically part of that class seems a bit disjointed. We can overcome this problem by defining our new function as a friend of the *Date* class.

```
class Date
{
   private:
```

```
        .
        .
    public:
        .
        .
        friend ostream& operator<<(ostream&,Date&);
};
```

The advantage of declaring the overloaded operator as a friend of the *Date* class is that it now has access to the private attributes, meaning we can use a term such as

```
d.day
```

rather than

```
d.GetDay()
```

allowing us to rewrite the code for `operator<<()` as:

```
ostream& operator<<(ostream& ct, Date& dt)
{
   ct << dt.day <<'/' << dt.month << '/' << dt.year;
   return ct;
}
```

Activity 16.37

Add `operator<<` as a friend in the *Date* class and test that the new operator works correctly by using it to display the contents of a *Date* object.

At this point, it won't now come as too much of a surprise to learn that `cin` is also an object. It is an instance of the class `istream`.

This time we will cheat a little and use `scanf()` within the new function rather than `cin` since we can then enter the / symbols between each part of the date. The new function has the following code.

```
istream& operator>>(istream& cn, Date& dt)
{
   scanf("%d/%d/%d",&dt.day, &dt.month, &dt.year);
   return cn;
}
```

Notice that we still need an `istream` parameter which is returned by the function even though the parameter is not otherwise used within the code.

Activity 16.38

Add `operator>>` as a friend in the *Date* class.

Test the new operator by reading and displaying the value of a *Date* object.

This latest function highlights the problem with friends - they get to override the privacy of the attributes and unless they take the same precautions as regular class methods, there is the danger that they may allow the contents of an object to be corrupted.

Activity 16.39

Modify the code for `operator>>` so that only a valid date will be stored in the *Date* object. Where the date entered is invalid, the *Date* object's values should remain unchanged.

More Friends

Another reason to create functions which are not part of a class is highlighted by the following example.

If we create a *Date* object and then add a few days to the value, we could do this with the following code:

```
Date d1(22,11,1963), d2;
d2 = d1 + 1;   // d2 is 23/11/1963
```

This works because we defined the `+` operator in the *Date* class. But remember what is really happening here is more accurately reflected when we rewrite the last line as

```
d2 = d1.operator+(1);
```

so it should come as no surprise that rearranging the assignment statement to read

```
d2 = 1 + d1;
```

won't work. This time the `+` is not the operator defined in *Date* – the elements are in the wrong order. And because of this, the program will not compile.

However, we can solve our problem by writing a separate, standard function:

```
Date operator+(int days, Date& d)
{
   Date dt = d.FromJDN(d.ToJDN() + days);
   return dt;
}
```

Since we made the methods `FromJDN()` and `ToJDN()` private, this function will only compile if we make it a friend of the *Date* class, thereby giving access to the two routines.

Activity 16.40

Add the new `operator+` function described above as a friend of the *Date* class and test it using a second *Date* object which is set to the first *Date* object's date + 10 days.

More on Class Diagrams

We first saw class diagrams at the start of this chapter, but since then we have covered many additional class features which it would be useful to identify within a diagram.

The following features can be added:

+	at the start of a property indicates that it is public.
-	at the start of a property indicates that it is private.
#	at the start of a property indicates that it is protected.

We'll talk about protected attributes in the next chapter.

= value	after a property shows its initial value. OR after a parameter, giving it's default value.
underlined	below a static property.
«constructor»name	for constructors.
«destructor»name	for destructors.
«friend»name	for functions which are friends of the class.
in, **out**, **inout**	one of these terms can be used in front of each operation parameter to indicate which direction a parameter's value is being passed.

When naming the constructors and the destructor in a class diagram, we can use the C++ convention of naming them after the class or you can use other terms such as *Create* and *Destroy* or *New* and *Delete*.

There are no class diagram conventions for showing finer details such as the fact that a method uses `const` parameters or does not modify any of the attributes of the class. In any case, the decision to use `const` parameters or call by reference variables is often a decision made at a later stage in the development process.

Of course, since the purpose of a class diagram is an aid to our task, we are really free to add whatever detail we might find helpful later when we come to develop our code. FIG-16.11 shows a class diagram for the latest version of the *Date* class using the new features described above.

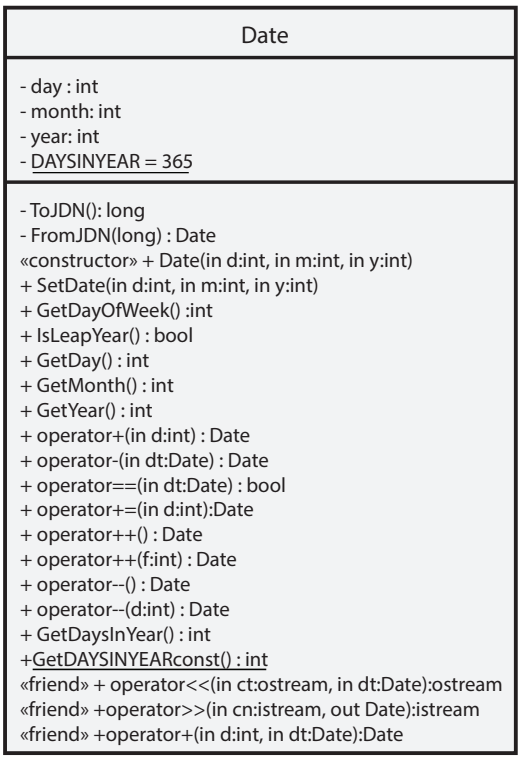


FIG-16.11

Date Class Diagram

In addition, we would need to create mini-specs for each operation mentioned in the class diagram.

Visualising An Object

The class diagram is an essential tool when designing a new class, but a more informal way of looking at a *Date* object, which, at this early stage in the learning process, may give us a greater insight into how objects operate is shown in FIG-16.12.

FIG-16.12

How an Object Operates

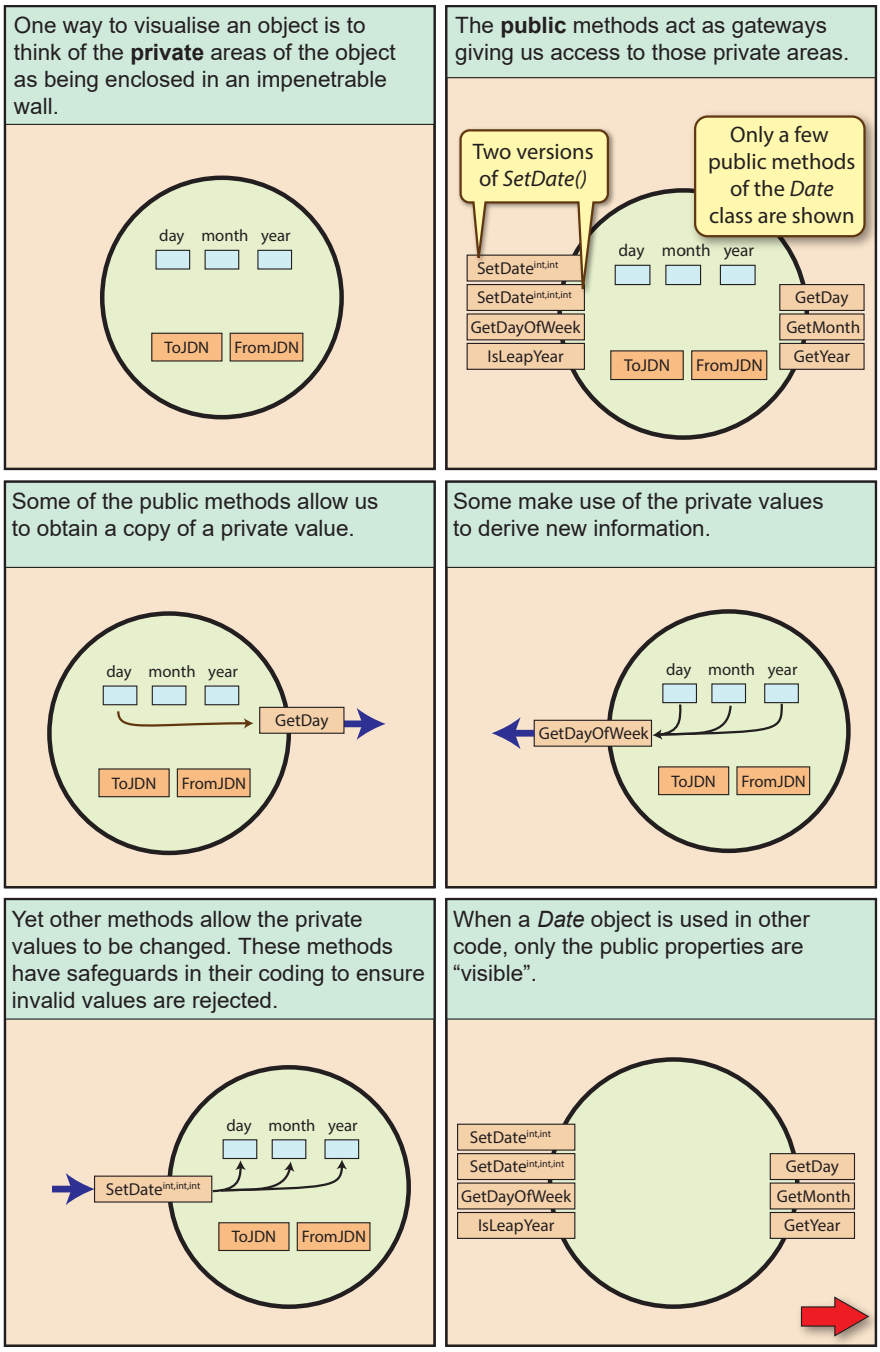
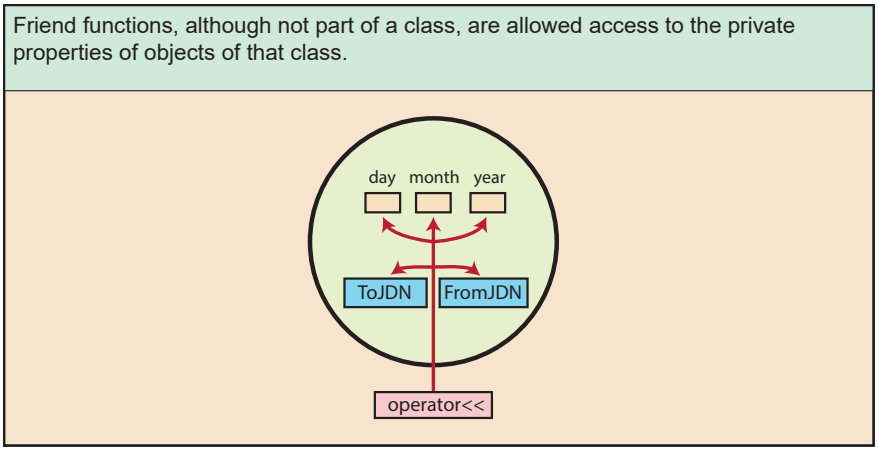


FIG-16.12
(continued)

How an Object Operates



Student Class

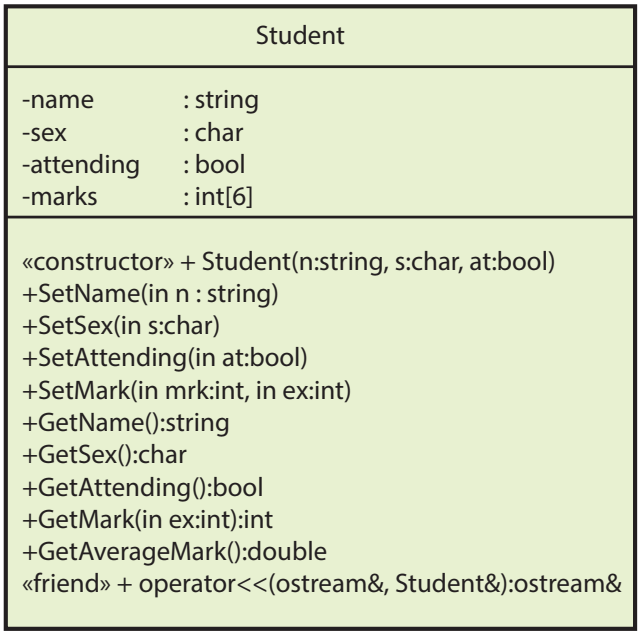
We've spent most of this chapter constructing the *Date* class. Obviously, as an introduction to classes, we've made use of a very simple class. And, although we've looked at various types of operations we can add to a class, we've done little with the type of attributes that can be used within a class.

In fact, so far we've only used `int` attributes (one of which was static). But of course, we can have `float`, `char`, `bool` attributes as well as arrays, pointers, structs, and even objects.

To demonstrate a larger range of attribute types, we'll create another class, *Student*, which contains details of a college student along with their marks for six exams. The class diagram for *Student* is shown in FIG-16.13.

FIG-16.13

Student Class Diagram



The code for most of the new class is shown in FIG-16.14.

FIG-16.14

Student Class Code

```
#include <iostream>
using namespace std;

class Student
{
   private:
      char name[21]{ "" };
      char sex{ '?' };
      bool attending{ true };
      int marks[6]{ 0 };
   public:
      Student(const char*, char, bool = true);
      void SetName(char*);
      void SetSex(char);
      void SetAttending(bool);
      void SetMark(int, int);
      char* GetName();
      char GetSex();
      bool GetAttending();
      int GetMark(int);
      double GetAverageMark();

      friend ostream& operator<<(ostream&, Student&);
};

//************************************
//***    Student class Operations   ***
//************************************

Student::Student(const char* n, char s, bool at)
{
   strcpy(name, n);
   s = toupper(s);
   if (s == 'M' || s == 'F')
      sex = s;
   attending = at;
}


//*** Set the student's name to n ***
void Student::SetName(char * n)
{
   strcpy(name, n);
}


//*** Set student's sex to s ***
void Student::SetSex(char s)
{
   s = toupper(s);
   if (s == 'M' || s == 'F')
      sex = s;
}


//*** Set student's attendence status ***
void Student::SetAttending(bool at)
{
   attending = at;
}
```

FIG-16.14
(continued)

Student Class Code

```
//*** Set student's mark for specified exam ***
void Student::SetMark(int mrk, int ex)
{
   //*** If not valid exam, return ***
   if (ex < 0 || ex > 5)
      return;
   //*** If not valid mark, return ***
   if (mrk < 0 || mrk > 100)
      return;
   //*** Store mark ***
   marks[ex] = mrk;
}

//*** Return student's name ***
char* Student::GetName()
{
   return name;
}

//*** Return student's attendence status ***
bool Student::GetAttending()
{
   return attending;
}

//*** Return student's exam mark ***
int Student::GetMark(int ex)
{
   if (ex < 0 || ex > 5)
      return -1;
   return marks[ex];
}

//*** Return student's average mark ***
double Student::GetAverageMark()
{// TO BE CODED}

//*** Display contents of student object ***
ostream& operator<<(ostream& ct, Student& st)
{//TO BE CODED}

int main()
{
   Student st1("Madeline Bray", 'F');
   Student st2("Nicholas Nickleby",'M');
   Student st3("Ada Clare", 'F', false);
   Student st4("Richard Carstone", 'M');
   //*** Assign random marks ***
   for (int ex = 0; ex < 6; ex++)
      st1.SetMark(rand() % 101, ex);
   for (int ex = 0; ex < 6; ex++)
      st2.SetMark(rand() % 101, ex);
   for (int ex = 0; ex < 6; ex++)
      st3.SetMark(rand() % 101, ex);
   for (int ex = 0; ex < 6; ex++)
      st4.SetMark(rand() % 101, ex);
   //*** Display object contents ***
   cout << st1 << endl;
   cout << st2 << endl;
   cout << st3 << endl;
   cout << st4 << endl;
}
```

Activity 16.41

Start a new project, *StudentClass*, and implement the code given above.

Add the code for the two missing methods, *GetAverageMark()* and *operator<<()*. *GetAverageMark()* should return the average of the six marks.

operator<<() should display the contents of the complete object and finish with the average mark. The attribute *attending* should be displayed as "*attending*" or "*not attending*".

The first array in our *Student* class is the *name* attribute which makes use of an array of type `char`.

In any modern C++ code we would use a `string` class object (covered in Chapter 20) for storing data such as the student's name but if we ever encounter existing code from a few years back it may well use an array of `char` or a `char` pointer to store strings. If you're never going to encounter older C++ code, feel free to skip over this topic.

Using *char* Pointers

Although very useful, standard arrays can be a bit problematic. Make them too small and we cannot store all our data; make them too large and we waste space. In the case of string constants, C++ solves this problem by reserving exactly the space required.

Older code might use this same approach when storing a string value by employing a pointer in conjunction with the `new` command to reserve the exact amount of space required. For example, in the *Student* class we could use a `char` pointer rather than a `char` array for the name, it would be:

```
class Student
{
   private:
      char* name;
      char sex;
         ...
```

A function which sets the *name* attribute in objects of this class would be coded as:

```
void Student::SetName(const char* n)
{
   //*** If no new name, exit ***
   if (n == nullptr)
      return;
   //*** Delete any space already being used ***
   if (name != nullptr)
      delete[]name;
   //*** Create enough space for the new name ***
   name = new char[strlen(n)+1]; //Extra byte for final null
   //*** Copy name into this new space ***
   strcpy(name,n);
}
```

When we use a pointer as an attribute or parameter, it's important that any method which accesses that attribute checks for the value `nullptr` (meaning the pointer does not reference a data area) and handles that situation appropriately. In the above method we can see a parameter guard to check against *n* containing null and that space is only deallocated if *name* is not set to null.

When a class makes use of dynamically allocated space, it becomes important that we add a destructor to the class which deletes any allocated space.

```
Student::~Student()
{
   if(name != nullptr)
      delete [] name;
}
```

We might be asking at this point, why not use a smart pointer? That way we could avoid having to delete the allocated space in the destructor. However, we must remember, we're only likely to find this way of doing things in older code before the `string` class and smart pointers were introduced.

Activity 16.42

Modify *StudentClass* so that the *name* attribute is implemented as a `char` pointer (which should have an initial value of `nullptr`).

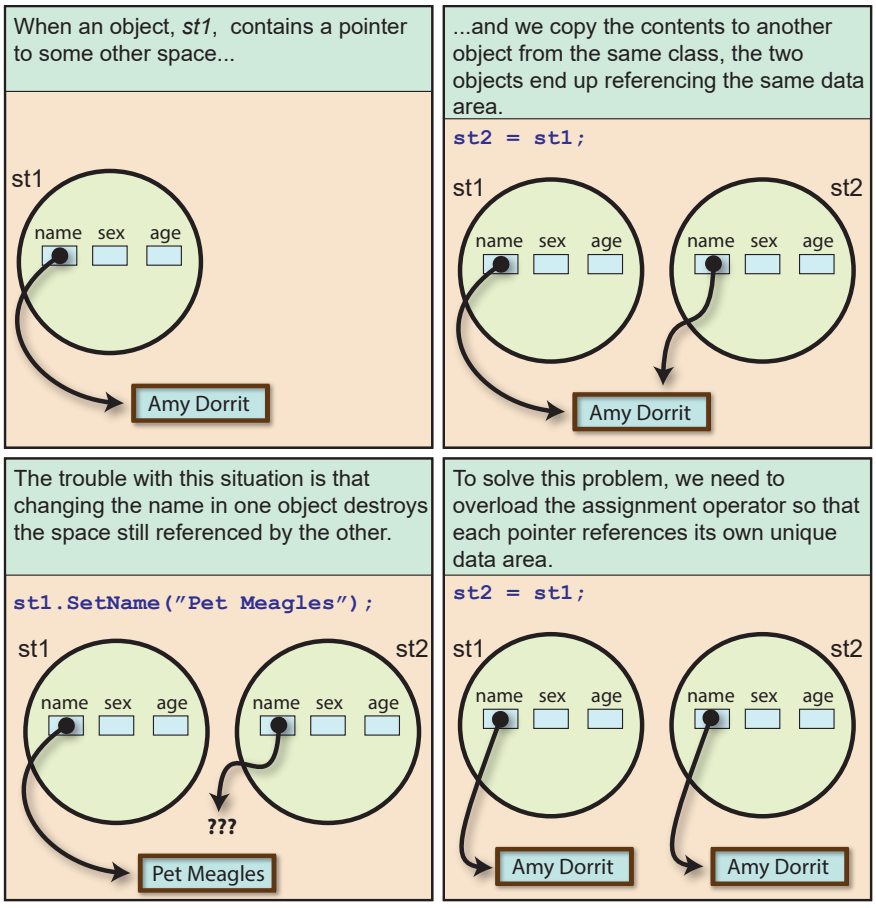
Make any other necessary changes. Check that the program operates exactly as before.

Overloading the Assignment Operator

Classes containing pointers to dynamically allocated space present another problem when we try to copy the contents of one object to another (see FIG-16.15).

FIG-16.15

The Trouble with Pointers



Activity 16.43

Modify *StudentClass* so that the program logic is:

Create Student objects st1 to st4 (as before)
Copy st1 to object st2
Display the contents of st1 and st2
Change the name in st1 to "Pet Meagles"
Display the contents of st1 and st2

What happens when you run the program?

Expect the program to crash!

Now that we've experienced the problems of the default assignment operator, we can correct the problem with the following code:

```
//*** Copies a Details object to current Details object ***
Student& Student::operator=(Student& st)
{
   //*** Delete any space already being used ***
   if (name != nullptr)
      delete[]name;
   //*** Set up space for copy of name ***
   int size = strlen(st.name)+1;
   name = new char[size];
   //*** Copy name ***
   strcpy(name, st.name);
   //*** Copy sex ***
   sex = st.sex;
   //*** Copy attending ***
   attending = st.attending;
   //*** Copy marks ***
   for(int c = 0; c < 6; c++)
      marks[c] = st.marks[c];
   //*** Return reference to updated object ***
   return *this;
}
```

The function returns a reference to the updated object. This allows the operator to be used in situations such as

```
st3 = st2 = st1;  //All Detail objects
```

Activity 16.44

Modify *StudentClass,* adding the = operator to the class.

How does this affect the results produced?

The assignment operator is closely related to the copy constructor: if we need to create code for one of these methods, we will almost certainly have to create near identical code for the other.

For example, in the *Details* class, with the *name* attribute now implemented as a pointer, the code

```
Student st2(st1);
```

will require exactly the same logic to copy the contents of *st1* to *st2* as we employed in the *operator=()* method.

Activity 16.45

Modify *StudentClass,* adding a copy constructor to the class. Test the new code by copying the contents of *st1* to a new *Student* object using the line

```
Student st5(st1);
```

There are even situations where we may be forgiven for thinking we are using the assignment operator when, in fact, we are calling the copy constructor. For example, in the line

```
Details st2 = st1;
```

it is the copy constructor that is called to copy the contents of *dt1* to *dt2.* This is because the assignment is being made as the object is being created. On the other hand, the lines

```
Details st2;
st2 = st1;
```

first use the zero-argument constructor to create *st2* and then the assignment operator to copy *st1* to *st2*.

const Methods

Back in Chapter 11, we saw that reference parameters are used in a function when we want to update that parameter within the function. But a second reason for using a reference parameter is that no copy is made of the actual parameter's value. This can save significant time and space when that parameter is a complex record structure or object.

When we want to pass a parameter as a *pass-by-reference* value for the sake of efficiency but without the intention to change any values held within the parameter, we had the option to add the term `const` in the parameter list:

```
GetDayOfWeek(const Date& d)
```

This same option is open to us when creating methods within a class. For example, if we wanted to make the parameter to the *Detail* class's *operator=()* method a `const`, we would start the method's code with the line

```
Student& Student::operator=(const Student& dt)
```

Activity 16.46

Modify *StudentClass,* so that the `=` operator now uses a `const` parameter (two lines of code need to be changed). Check that the program operates exactly as before.

Although making a parameter to a class method a `const` isn't a problem, the same can't be said when the application programmer tries to write a function in the same way. For example, let's say we want to write a function that returns the number of characters in a *Student* object's name field. We could try writing this as

```
int GetNameSize(const Student& st)
{
   return strlen(st.GetName());
}
```

Remember this is not a method of the *Details* class, but a separate function written by the application programmer.

When we call this function in `main()` with a line such as

```
cout << GetNameSize(st1) << endl;
```

and try to compile the code, we would be presented with a compilation error for the line within *GetNameSize()*.

Activity 16.47

Modify *SudentClass,* adding *GetNameSize()* as a standard function and using the function to display the size of the name in *st1*.

Try compiling your code *(there will be an error)* .

The reason for our problem is relatively simple: we have stated that the parameter to *GetNameSize()* is a `const` parameter. That means that *st* (the parameter) cannot be modified in any way within the function. But the function calls a method of the *Student* class, *GetName()*, and the compiler has no way of knowing if that method modifies the contents of *st*. Because it does not know this, the compiler throws up an error.

To solve this general problem, we need to tell the compiler which methods defined within the *Student* class modify attributes of that class and which don't.

For example, we know that the methods

```
SetName()
SetSex()
SetAttending()
```

all modify *Student* class attributes. On the other hand,

```
GetName()
GetSex()
GetAttending()
```

only retrieve the current values of attributes but do not change them.

To let the compiler know that a method does not modify any attribute within its own class, we add the term `const` to the end of the function prototype. Hence, the *Details* class code would now begin with:

```
class Student
{
   private:
      char* name{ nullptr };
      char sex{ '?' };
      bool attending{ true };
      int marks[6]{ 0 };
   public:
      Student(const char*, char, bool = true);
      Student(const Student&);
      void SetName(const char*);
      void SetSex(char);
      void SetAttending(bool);
      void SetMark(int, int);
      char* GetName() const;
      char GetSex() const;
      bool GetAttending() const;
      int GetMark(int) const ;
      double GetAverageMark() const;
```

```
        Student& operator=(const Student&);
        ~Student();

        friend ostream& operator<<(ostream&, Student&);
};
```

The need for `const` does not apply to *friend* functions.

The functions themselves must also include the `const` term in their heading as in the code

```
char* Student::GetName()const
{
   return name;
}
```

Activity 16.48

Modify *StudentClass,* so that the appropriate methods are defined with a `const` ending and check that the program now executes.

const Methods and *char* Arrays Attributes

There is one situation where declaring a `const` method can cause a slight problem. Let's assume we have declared a *Trivial* class and made use of it as shown in FIG-16.16.

FIG-16.16

Trivial Class

```
#include <iostream>

using namespace std;

class Trivial
{
   private:
      char name[31];
   public:
      void SetName(const char*);
      char* GetName() const;
};

void Trivial::SetName(const char* n)
{
   if(n != nullptr)
      strncpy(name, n, 30);
}


char* Trivial::GetName() const
{
   return name;
}


int main()
{
   Trivial tv;
   tv.SetName("Libby");
   cout << tv.GetName() << endl;
}
```

Activity 16.49

Create a new project called *TrivialClass*, and create a source file containing the code given above.

What happens when you attempt to compile the code?

The problem is that, since we have declared *GetName()* as a `const` method, C++ will not return a pointer to the data in case the pointer is dereferenced and used to modify the data - hence the compilation error.

However, if we declare the return value to be a pointer to a `const` value, then C++ will be mollified and allow the compilation. So, to have the program compile we need to change the prototype and heading for *GetName()* to read

```
const char* GetName() const
```

Activity 16.50

Change the appropriate two lines in *TrivialClass* and rerun your project.

Object Pointers

Just as we can create pointers which reference basic data types and record structures, so we can create pointers to class objects. For example, we could create a pointer to a *Student* object using the line:

```
Student* stptr;
```

The most likely use of such a pointer is to have it reference a dynamically allocated object as in:

```
stptr = new Student;
```

The statement above will create a new object, executing its zero-argument constructor to initialise the object's attributes, and assign the start address of the object to *stptr*. If we have written a constructor that accepts arguments, that can be used in the creation of the object:

```
stptr {new Student("Henry Gowan",'M')};
```

Once the object has been created and referenced by the pointer, we use the same syntax to access the public properties of the object as we employed earlier with record (struct) pointers. Hence, we could retrieve the name held in the object with the expression

```
stptr->GetName()
```

which dereferences the pointer to gain access to the object's elements.

Activity 16.51

Modify *StudentClass,* so that *st1* and *st2* are pointers to *Student* objects (with *st3* and *st4* remaining as standard objects). Change the output statements where necessary to display the contents of all four objects.

Objects and Arrays

An Array of Objects

If we require several objects of the same type, we are free to create an array of objects. For example, we might write

```
Student group[4];
```

When we create an array of objects in this way, it is the zero-argument constructor that is used to initialise each of the objects. If the class does not have a zero-argument constructor, then attempting to create the array will produce a compilation error.

Accessing public properties of the object requires a combination of array element access and record field access. Hence, to execute the *GetName()* method of the third object in the array we would use the term

```
group[2].GetName()
```

Activity 16.52

Modify *StudentClass* so that all four *Student* objects are stored in a *Student* class array called *group*. Assign the same values as before to the objects and display their contents.

Arrays of Class Pointers

Another option when handling several objects from the same class is to have an array of class pointers. For example, we could create space for five *Date* pointers with the line:

```
Date* event[5];
```

Of course, if we do this, then we must also create space for each object before using it:

```
for(int c = 0; c <=4; c++)
   event[c] = new Details;
```

Activity 16.53

Modify *StudentClass* so that it uses an array of pointers rather than an array of objects.

We have already seen that the main advantage of such a system is to ensure, by writing robust methods, that the data held is not corrupted by allowing invalid values to be assigned.

Time Class

Like most new concepts it takes a bit of experience before we become fully understanding of all the subtleties behind those ideas.

To help gain a better grasp of coding classes as well as setting up code that we will revisit in later chapters, the Activity that follows will get you to develop a new class, *Time*, from scratch.

The class diagram for the new class is shown in Activity 16.53.

Of course, to accompany a class diagram, we would expect to see mini-specs for each of the operations stated in the diagram. However, in this case, since much of what is required shares a close similarity to methods already coded for the *Date* class, we'll show just two mini-specs here.

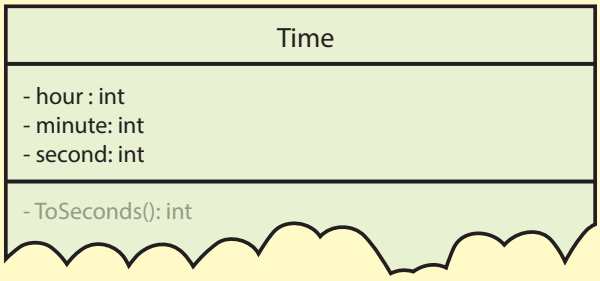
Those are for the private methods, *ToSeconds()* and *FromSeconds()*.

Class	: Time
Operation	: ToSeconds
Parameters	
In	: None
Out	: result : int
Attributes	
Read	: hour, minute, second
Written	: None
Pre-condition	: None
Post-condition	: *result* = *hour* *3600 + *minute*60+*second*
Description	: Sets *result* to the number of seconds since midnight to the current time.

Class	: Time
Operation	: FromSeconds
Parameters	
In	: s : int
Out	: result : Time
Attributes	
Read	: None
Written	: None
Pre-condition	: None
Post-condition	: *result.hour* = *s*/3600 *result.minute* = (*s* - *result.hour**3600)/60 *result.second* = *s*%60
Description	: Converts *s* seconds into hours, minutes and seconds storing the value in *result*.

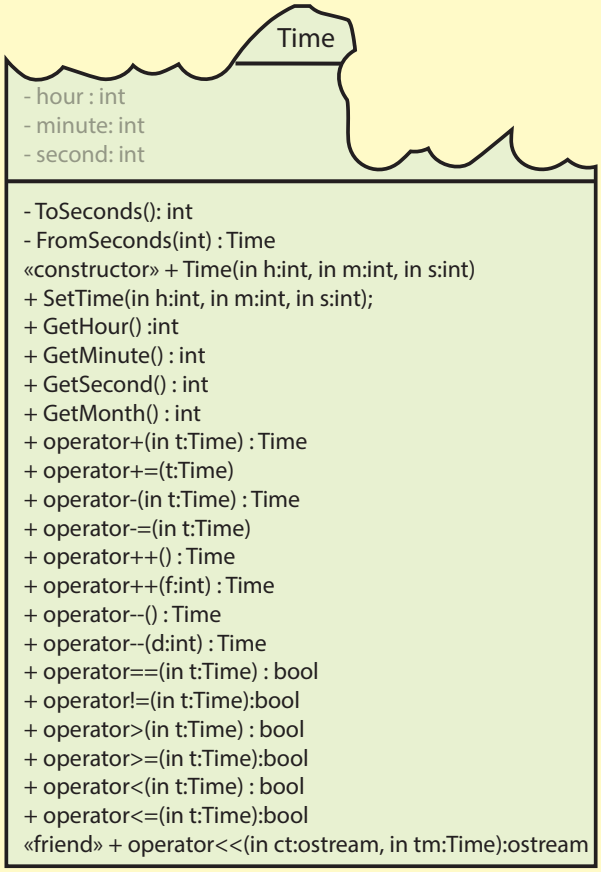
Activity 16.54

Start a new project called *TimeClass* and develop all of the code necessary to define a *Time* class as shown below. Do not code the methods yet.



Remember a UML class diagram doesn't tell us everything about a method's coding. For example, it does not tell us if we should use const values, or reference values or a pointer. It does not tell us if a method should be declared as const. These are things that are normally decided at a later stage or in further documentation.

Activity 16.54 (continued)



Make your own decision on the use of *const*, and references based on the classes covered previously in this chapter.

Activity 16.55

In project *TimeClass* implement all of the operations for the *Time* class and create a `main()` function to test one arithmetic operator, one relational operator and the << operator.

The constructor and *SetTime()* method should allow the *hour* attribute to be set to values above 23 but not below zero. No upper limit will be useful when an object of this type is used to represent a duration time rather than a clock time.

Summary

- A class is a blueprint for a structure containing both data and related operations.
- This combining of the data and operations is known as encapsulation.
- The elements of a class are collectively known as properties, features or members.

- The data elements of a class are known as attributes.
- The tasks of a class are known as operations.
- An object is a program component which realises a class specification.
- An object is also called an instance of a class.
- A class diagram is a rectangle split into three spaces. These contain: the class name; the attributes' names and types, and the operations' names, parameters and return type.
- When designing a class, use:

+	to mark a public feature
-	to mark a private one
= value	to assign a value to a named constant
_	underline to mark a static feature
«constructor»	to mark a constructor
«destructor»	to mark a destructor.

- In C++, a class is defined using the term `class`.
- When coded, the operations of a class are known as methods.
- There are two parts to coding a class: the class declaration and the methods' code.
- The call operator is the name given to the set of open and close parentheses.
- A functor is any class that defines a method for the call operator.
- Inline methods can be coded within the class declaration or marked with the keyword `inline`.
- Properties of a class can be marked as public or private.
- When coding the methods of a class, the class name and the scope resolution operator must precede the method name.
- C++ automatically adds a zero-argument constructor, a copy constructor, an assignment operator, and a destructor to every class.
- The default copy constructor and assignment operator copy the attribute values in one object to the corresponding attributes in another object.
- The default constructors, the assignment operator and destructor can be overridden by defining new operations within the class.
- A constructor cannot return a value.
- A destructor cannot return a value or take parameters.
- All methods within a class can be overloaded.
- A program using an object can access only its public properties.
- A class constructor is run automatically when an object is created.
- If an object is created with the value of an existing object, the copy constructor is executed.
- The class destructor is run automatically when an object is deleted.
- A program will, where possible, automatically make use of a class's

constructors to convert another type to an object of that class.

- To stop a constructor being used automatically to convert other values, start the definition of each constructor with the term `explicit`.
- When constructors are declared as explicit, then conversions to an object must be made explicit.
- Data hiding limits access to the features of a class when using an object of that class.
- C++ maintains a pointer called *this* which references the current object whose features are being accessed.
- Constants can be defined within a class.
- Attributes of a class can be marked as static.
- There is only a single copy of each static attribute (not one in each object).
- Methods can also be declared as static.
- Static features in a class exist even before any object of that class has been created.
- Public static features can be accessed using the format

```
class name :: static feature name
```

- The code within a static methods cannot access non-static attributes of its class.
- When overloading the insertion (`>>`) and extraction (`<<`) operators, to display an object, the code must create standard functions rather than operations of the class.
- Standard functions can have access to the private features of a class by declaring the functions as friends of the class.
- When an attribute of a class is a pointer, make sure that the copy constructor and assignment operator handle the pointer appropriately.
- A method which does not modify the value of any class attribute should add the term `const` to the end of its declaration and heading.
- Only methods marked as `const` have access to the attributes of an object or parameter marked as `const`.

Solutions

Activity 16.1

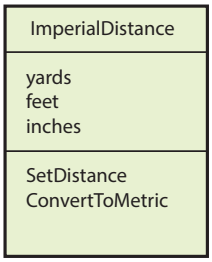
Beach ball characteristics include:
- Size
- Colour
- Weight
- Material used

Operations include:
- Inflate
- Deflate
- Throw
- Catch
- Bounce

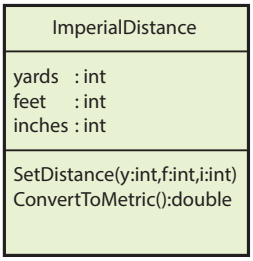
Activity 16.2

a) Class
b) Object
c) Class
d) Object

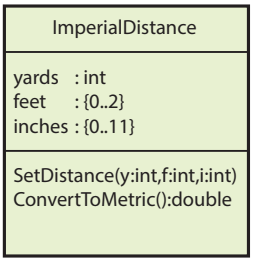
Activity 16.3



Activity 16.4



Activity 16.5



Activity 16.6

Class	: ImperialDistance
Operation	: setDistance
Parameters	
In	: y : int
	f : int
	i : int
Out	: None
Attributes	
Read	: None
Written	: yards, feet, inches
Pre-condition	: y,f,i forms a valid Imperial distance
Post-condition	: yards = y
	feet = f
	inches = i
Description	: Sets the distance to *y* yards, *f* feet, *i* inches.

Activity 16.7

Code for *DateClass*:

```
#include <iostream>
using namespace std;

//*** Class declaration ***
class Date
{
  public:
     int day;
     int month;
     int year;

     void SetDate(int, int, int);
     int GetDayOfWeek();
     bool IsLeapYear();
};
```

Activity 16.8

The code should compile.

Activity 16.9

Code for *ImperialDistanceClass*:

```
#include <iostream>
using namespace std;

class ImperialDistance
{
  public:
     int yards;
     int feet;
     int inches;

     void SetDistance(int,int,int);
     double ConvertToMetric();
};



//*************************************
//*** ImperialDistance Class Methods ***
//*************************************

// *** Sets distance to y yds, f ft, and i in ***
void ImperialDistance::SetDistance(int y, int f,
↳int i)
{
  //*** If any parameter invalid, exit ***
  if (i < 0 || i > 11 || f < 0 || f > 2 || y < 0)
     return;
  //*** Assign distance ***
  yards = y;
  feet = f;
  inches = i;
}

// *** Returns equivalent distance in metres ***
double ImperialDistance::ConvertToMetric()
```

```
{
  return ((yards*36 + feet*12 + inches)*0.0254);
}

int main(){}
```

Activity 16.10

The supplied date fell on a Saturday.

Activity 16.11

Since the parameters to *SetDate()* are invalid, the attributes of *d1* are never assigned values and so the random values within the memory assigned to those attributes are displayed.

Activity 16.12

Coding for `main()` in *ImperialDistanceClass*:

```
int main()
{

  ImperialDistance dist;  // ImperialDistance object

  // *** Set distance ***
  dist.SetDistance(0,0,0);

  // *** Display distance ***
  cout << dist.yards << " yards  " << dist.feet <<
  ↳" feet  " << dist.inches << " inches " << endl;
}
```

When we change the call to *SetDistance()* to use the invalid parameters 2,3,10, we have a similar situation to that of the last Activity. The parameters to *SetDistance()* are invalid and hence the attributes of the distance objects retain their original random values which are displayed on the screen.

Activity 16.13

Modified code for *DateClass*:

```
#include <iostream>
using namespace std;

class Date
{
  public:
    int day;
    int month;
    int year;

    void SetDate(int, int, int);
    void SetDate(int,int);
    int GetDayOfWeek();
    bool IsLeapYear();
};


//*************************************
//***       Date Class Methods        ***
//*************************************
// *** Sets the date to d/m/y ***
void Date::SetDate(int d, int m, int y)
{
  // *** Days in each month of year ***
  int daysinmonth[] = {0, 31,28,31, 30,31,30,
  ↳31,31,30, 31,30,31};
  //*** If month invalid, exit ***
  if (m < 1 || m > 12)
    return;
  //*** Add 1 to days in February if leap year ***
  daysinmonth[2] += (y%400 == 0)||(y%4 == 0 &&
  ↳y%100 != 0);
  //*** If days in month invalid, exit ***
  if (d <1 || d > daysinmonth[m])
    return;
  //*** If year less than 1, exit ***
  if (y < 1)
    return;
  //*** Assign date ***
  day = d;
  month = m;
  year = y;
}

// *** Sets date using days-into-year (diy) and year
↳(y) ***
void Date::SetDate(int diy, int y)
{
  // *** Days in each month of year ***
  int daysinmonth[] = { 0, 31,28,31, 30,31,30,
  ↳31,31,30, 31,30,31 };
  //*** If leap year add one day to February ***
  daysinmonth[2] += (y % 400 == 0) || (y % 4 == 0
  ↳&& y % 100 != 0);
  //*** Calculate month ***
  int remaining = diy;
  int m = 0;
  while (remaining > 0)
  {
    m++;
    remaining -= daysinmonth[m];
  }
  remaining += daysinmonth[m];
  //*** Set date ***
  SetDate(remaining, m, y);
}

// *** Returns the day of the week of a date ***
int Date::DayOfWeek()
{
  int M, modifiedyear, C, Y;
  //*** Calculate M ***
  M = (month + 9) % 12 + 1;
  //*** Calculate modified year ***
  modifiedyear = year - M / 11;
  //*** Calculate C ***
  C = modifiedyear / 100;
  //*** Calculate Y ***
  Y = modifiedyear % 100;
  //*** Calculate day of week ***
  int dayofweek = ((static_cast<int>(2.6 * M - 0.2)
  ↳+ day + Y + Y/4 + C/4 - 2*C)%7+7)%7;
  return dayofweek;
}


// *** Returns true if year is a leap year ***
bool Date::IsLeapYear()
{
  return ((year%400 == 0)||(year%4 == 0 &&
  ↳year%100 != 0));
}

int main()
{
  // *** Day names ***
  char daynames[7][10] =
  ↳{"Sunday","Monday","Tuesday","Wednesday",
  ↳"Thursday","Friday","Saturday"};

  Date d1;  // Date object

  // *** Set date ***
  d1.SetDate(60,2021);

  // *** Display date details ***
  cout << d1.day << '/' << d1.month << '/' <<
  ↳d1.year << " was a " << daynames[d1.DayOfWeek()]
  ↳<< endl;
}
```

Note that the final line of our new method uses a call to the original *SetDate()* to store the date. This makes sure only valid dates can be stored.

Activity 16.14

Modified code for *DateClass*:

```
#include <iostream>
using namespace std;

class Date
{
  public:
    int day;
    int month;
    int year;

    Date();
```

```
    void SetDate(int, int, int);
    void SetDate(int,int);
    int GetDayOfWeek();
    bool IsLeapYear();
};

//*************************************
//***       Date Class Methods        ***
//*************************************

//*** Initialise new Date objects to 1/1/2001 ***
Date::Date()
{
  day = 1;
  month = 1;
  year = 2001;
}

// *** Sets the date to d/m/y ***
void Date::SetDate(int d, int m, int y)
{
  // *** Days in each month of year ***
  int daysinmonth[] = {0, 31,28,31, 30,31,30,
  ↳31,31,30, 31,30,31};
  //*** If month invalid, exit ***
  if (m < 1 || m > 12)
    return;
  //*** Add 1 to days in February if leap year ***
  daysinmonth[2] += (y%400 == 0)||(y%4 == 0 &&
  ↳y%100 != 0);
  //*** If days in month invalid, exit ***
  if (d <1 || d > daysinmonth[m])
    return;
  //*** If year less than 1, exit ***
  if (y < 1)
    return;
  //*** Assign date ***
  day = d;
  month = m;
  year = y;
}

// *** Sets date using days-into-year (diy) and year
↳(y) ***
void Date::SetDate(int diy, int y)
{
  // *** Days in each month of year ***
  int daysinmonth[] = { 0, 31,28,31, 30,31,30,
  ↳31,31,30, 31,30,31 };
  //*** If leap year add one day to February ***
  daysinmonth[2] += (y % 400 == 0) || (y % 4 == 0
  ↳&& y % 100 != 0);
  //*** Calculate month ***
  int remaining = diy;
  int m = 0;
  while (remaining > 0)
  {
    m++;
    remaining -= daysinmonth[m];
  }
  remaining += daysinmonth[m];
  //*** Set date ***
  SetDate(remaining, m, y);
}



// *** Returns the day of the week of a date ***
int Date::DayOfWeek()
{
  int M, modifiedyear, C, Y;
  //*** Calculate M ***
  M = (month + 9) % 12 + 1;
  //*** Calculate modified year ***
  modifiedyear = year - M / 11;
  //*** Calculate C ***
  C = modifiedyear / 100;
  //*** Calculate Y ***
  Y = modifiedyear % 100;
  //*** Calculate day of week ***
  int dayofweek = ((static_cast<int>(2.6 * M - 0.2)
  ↳+ day + Y + Y/4 + C/4 - 2*C)%7+7)%7;
  return dayofweek;
}


// *** Returns true if year is a leap year ***
bool Date::IsLeapYear()
{
  return ((year%400 == 0)||(year%4 == 0 &&
  ↳year%100 != 0));
}

int main()
{
  // *** Day names ***
  char daynames[7][10] =
  ↳{"Sunday","Monday","Tuesday","Wednesday",
  ↳"Thursday","Friday","Saturday"};

  Date d1;  // Date object

  // *** Set date ***
  d1.SetDate(29,2,2001);

  // *** Display date details ***
  cout << d1.day << '/' << d1.month << '/' <<
  ↳d1.year << " was a " << daynames[d1.DayOfWeek()]
  ↳<< endl;
}
```

Since all *Date* objects default to 1/1/2001 the moment they are created, when the attempt to assign an invalid date to `d1` fails, the object retains the date 1/1/2001 which is displayed and reported as being a Monday.

Activity 16.15

Modified code for *ImperialDistanceClass*:

```
#include <iostream>
using namespace std;

class ImperialDistance
{
  public:
    int yards;
    int feet;
    int inches;

    ImperialDistance();
    void SetDistance(int,int,int);
    double ConvertToMetric();
};

//*************************************
//*** ImperialDistance Class Methods ***
//*************************************

//*** Zeroises newly created distance objects ***
ImperialDistance::ImperialDistance()
{
  yards = 0;
  feet = 0;
  inches = 0;
}

// *** Sets the distance to y yards, f feet, and i
inches ***
void ImperialDistance::SetDistance(int y, int f,
↳int i)
{
  //*** If any parameter invalid, exit ***
  if (i < 0 || i > 11 || f < 0 || f > 2 || y < 0)
    return;
  //*** Assign distance ***
  yards = y;
  feet = f;
  inches = i;
}


// *** Returns equivalent distance in metres ***
double ImperialDistance::ConvertToMetric()
{
  return ((yards*36 + feet*12 + inches)*0.0254);
}



int main()
{
  ImperialDistance dist;  // ImperialDistance object

  // *** Set distance ***
  dist.SetDistance(2,3,10);
```

```
  // *** Display distance ***
  cout << dist.yards << " yards  " << dist.feet
  ↳<< " feet  " << dist.inches << " inches "<<endl;
}
```

When the invalid parameters are used with *SetDistance()*, the object's value remains set to zero.

Activity 16.16

Modified code for *DateClass*:

```
#include <iostream>
using namespace std;

class Date
{
  public:
    int day;
    int month;
    int year;

    Date();
    Date(int,int,int);
    void SetDate(int, int, int);
    void SetDate(int,int);
    int GetDayOfWeek();
    bool IsLeapYear();
};

//*************************************
//***        Date Class Methods      ***
//*************************************

//*** Initialise new Date objects to 1/1/2001 ***
Date::Date()
{
  day = 1;
  month = 1;
  year = 2001;
}

//*** Initialise new Date objects to d/m/y ***
Date::Date(int d, int m, int y)
{
  // *** Days in each month of year ***
  int daysinmonth[] = {0, 31,28,31, 30,31,30,
  ↳31,31,30,31,30,31};

  bool valid = true;  // Parameters valid

  //*** If month invalid, invalid parameters ***
  if (m < 1 || m > 12)
    valid = false;
  //*** Add 1 to days in February if leap year ***
  daysinmonth[2] += (y%400 == 0)||(y%4 == 0 &&
  ↳y%100 != 0);
  //*** If days invalid, invalid parameters ***
  if (d <1 || d > daysinmonth[m])
    valid = false;
  //*** If year less than 1, invalid parameters ***
  if (y < 1)
    valid = false;
  //*** Assign date ***
  if (valid)
  {
    day = d;
    month = m;
    year = y;
  }
  else
  {
    day = 1;
    month = 1;
    year = 2001;
  }
}


// *** Sets the date to d/m/y ***
void Date::SetDate(int d, int m, int y)
{
  // *** Days in each month of year ***
  int daysinmonth[] = {0, 31,28,31, 30,31,30,
  ↳31,31,30, 31,30,31};
  //*** If month invalid, exit ***
  if (m < 1 || m > 12)
    return;
  //*** Add 1 to days in February if leap year ***
  daysinmonth[2] += (y%400 == 0)||(y%4 == 0 &&
  ↳y%100 != 0);
  //*** If days in month invalid, exit ***
  if (d <1 || d > daysinmonth[m])
    return;
  //*** If year less than 1, exit ***
  if (y < 1)
    return;
  //*** Assign date ***
  day = d;
  month = m;
  year = y;
}


// *** Sets date using days-into-year (diy) and year
↳(y) ***
void Date::SetDate(int diy, int y)
{
  // *** Days in each month of year ***
  int daysinmonth[] = { 0, 31,28,31, 30,31,30,
  ↳31,31,30, 31,30,31 };
  //*** If leap year add one day to February ***
  daysinmonth[2] += (y % 400 == 0) || (y % 4 == 0
  ↳&& y % 100 != 0);
  //*** Calculate month ***
  int remaining = diy;
  int m = 0;
  while (remaining > 0)
  {
    m++;
    remaining -= daysinmonth[m];
  }
  remaining += daysinmonth[m];
  //*** Set date ***
  SetDate(remaining, m, y);
}


// *** Returns the day of the week of a date ***
int Date::DayOfWeek()
{
  int M, modifiedyear, C, Y;
  //*** Calculate M ***
  M = (month + 9) % 12 + 1;
  //*** Calculate modified year ***
  modifiedyear = year - M / 11;
  //*** Calculate C ***
  C = modifiedyear / 100;
  //*** Calculate Y ***
  Y = modifiedyear % 100;
  //*** Calculate day of week ***
  int dayofweek = ((static_cast<int>(2.6 * M - 0.2)
  ↳+ day + Y + Y/4 + C/4 - 2*C)%7+7)%7;
  return dayofweek;
}

// *** Returns true if year is a leap year ***
bool Date::IsLeapYear()
{
  return ((year%400 == 0)||(year%4 == 0 &&
  ↳year%100 != 0));
}



int main()
{
  // *** Day names ***
  char daynames[7][10] =
  ↳{"Sunday","Monday","Tuesday",
  ↳"Wednesday","Thursday","Friday","Saturday"};

  Date d1(23,11,1963); // Date object

  // *** Display date details ***
  cout << d1.day << '/' << d1.month << '/' <<
  ↳d1.year << " was a " << daynames[d1.DayOfWeek()]
  ↳<< endl;
}
```

Activity 16.17

In *DateClass*, the class declaration needs to be changed to:

```
class Date
{
  public:
```

```
    int day;
    int month;
    int year;

    Date(int=1,int=1,int=2001);
    void SetDate(int, int, int);
    void SetDate(int,int);
    int GetDayOfWeek();
    bool IsLeapYear();
};
```

Note that the zero-argument constructor has been removed.

In `main()`, the parameters need to be removed when declaring *d1*:

```
int main()
{
  // *** Day names ***
  char daynames[7][10] =
  ↳{"Sunday","Monday","Tuesday",
  ↳"Wednesday","Thursday","Friday","Saturday"};

  Date d1;  // Date object

  // *** Display date details ***
  cout << d1.day << '/' << d1.month << '/' <<
  ↳d1.year << " was a " << daynames[d1.DayOfWeek()]
  ↳<< endl;
}
```

Without a line to change the value in *d1*, the date displayed is 1/1/2001.

Activity 16.18

Modifications to *ImperialDistanceClass*:

Class declaration:

```
class ImperialDistance
{
public:
  int yards;
  int feet;
  int inches;

  ImperialDistance(int=0, int=0, int=0);
  void SetDistance(int, int, int);
  double ConvertToMetric();
};
```

Code for new constructor:

```
//*** Initialises newly created distance objects ***
ImperialDistance::ImperialDistance(int y, int f, int
i)
{
  bool valid{ true };
  //*** If yards invalid, invalid parameters ***
  if (y < 0 )
    valid = false;

  //*** If feet invalid, invalid parameters ***
  if (f <0 || f > 2)
    valid = false;
  //*** If inches invalid,  invalid parameters ***
  if (i < 0 || i > 11)
    valid = false;
  //*** Assign distance ***
  if (valid)
  {
    yards = y;
    feet  = f;
    inches = i;
  }
  else
  {
    yards = 0;
    feet = 0;
    inches = 0;
  }
}
```

Modified code for *main()*:

```
int main()
{

  ImperialDistance dist(2,3,10); // ImperialDistance
                                 ↳object

  // *** Display distance ***
  cout << dist.yards << " yards  " << dist.feet
  << " feet  " << dist.inches << " inches " << endl;
}
```

When an invalid distance is attempted, the distance remains set at zero.

Activity 16.19

Code for *DateClass*'s *main()*:

```
int main()
{
  Date d1(23,1,2001);

  //*** Create new Date object containing same date
  ↳***
  Date d2(d1);

  //*** Display value of new Date object ***
  cout << d2.day << '/' << d2.month << '/'
  ↳<< d2.year << endl;
}
```

Activity 16.20

Code for *DateClass*'s *main()*:

```
int main()
{
  Date d1(23,1,2001);

  //*** Create new Date object containing same date
  ↳***
  Date d2(d1);

  d2 = 21;

  //*** Display value of new Date object ***
  cout << d2.day << '/' << d2.month << '/'
  ↳<< d2.year << endl;
}
```

The value displayed is 21/1/2001. The *month* and *year* values are taken from the constructor's defaults.

Activity 16.21

Modified code for *DateClass* declaration:

```
class Date
{
  public:
    int day;
    int month;
    int year;

    explicit Date(int=1,int=1,int=2001);
    void SetDate(int, int, int);
    void SetDate(int, int);
    int GetDayOfWeek();
    bool IsLeapYear();
};
```

This version creates a compilation error since 21 cannot be automatically converted to a *Date* object.

Modified code for *main()* :

```
int main()

{
  Date d1(23,1,2001);

  //*** Create new Date object containing same date
  ↳***
  Date d2(d1);

  d2 = static_cast<Date>(21);
```

```
  //*** Display value of new Date object ***
  cout << d2.day << '/' << d2.month << '/'
  ↳<< d2.year << endl;
}
```

This version compiles and runs successfully.

Activity 16.22

The class declaration in *DateClass* should be changed to

```
class Date
{
  public:
    int day;
    int month;
    int year;

    explicit Date(int=1,int=1,int=2001);
    ~Date();
    void SetDate(int, int, int);
    void SetDate(int, int);
    int GetDayOfWeek();
    bool IsLeapYear();
};
```

and the code for the new destructor added:

```
Date::~Date()
{
  cout << "Date object deleted\n";
};
```

The message *Date object deleted* should be displayed when the program is run.

Activity 16.23

The new code for the *Date* declaration is:

```
class Date
{
  private:
    int day;
    int month;
    int year;
  public:
    explicit Date(int=1,int=1,int=2001);
    ~Date();
    void SetDate(int, int, int);
    void SetDate(int, int);
    int GetDayOfWeek();
    bool IsLeapYear();
};
```

No other changes are required.

The compilation fails. Error messages from the compiler should be of the form:

'Date::day' : cannot access private member declared in class 'Date'

Activity 16.24

The class declaration for *Date* becomes:

```
class Date
{
  private:
    int day;
    int month;
    int year;
  public:
    explicit Date(int=1,int=1,int=2001);
    void SetDate(int, int, int);
    void SetDate(int, int);
    int GetDayOfWeek();
    bool IsLeapYear();
    int GetDay();
    int GetMonth();
    int GetYear();
};
```

The code for the new methods is:

```
//*** Returns the value in attribute day ***
int Date::GetDay()
{
  return day;
}

//*** Returns the value in attribute month ***
int Date::GetMonth()
{
  return month;
}

//*** Returns the value in attribute year ***
int Date::GetYear()
{
  return year;
}
```

Modified code for *main()*:

```
int main()
{
  Date d1(23,1,2001);

  //*** Create new Date object containing same date
  ↳***
  Date d2(d1);

  d2 = static_cast<Date>(21);

  //*** Display value of Date object ***
  cout << d2.GetDay() << '/' << d2.GetMonth() << '/'
  ↳<< d2.GetYear() << endl;
}
```

Activity 16.25

The class declaration for *Date* becomes:

```
class Date
{
  private:
    int day;
    int month;
    int year;

    long ToJDN();
    Date FromJDN();

  public:
    explicit Date(int=1,int=1,int=2001);
    void SetDate(int, int, int);
    void SetDate(int, int);
    int DayOfWeek();
    bool IsLeapYear();
    int GetDay();
    int GetMonth();
    int GetYear();
};
```

The code for the new methods is:

```
// *** Returns the JDN of the date ***
long Date::ToJDN()
{
  int a = (14 - month)/12;
  int m = month + 12*a - 3;
  int y = year + 4800 - a;
  long result = day + (153*m + 2) / 5 + 365*y + y/4
  ↳- y/100 + y/400  - 32045;
  return result;
}


// *** Sets date to eqivalent of JDN ***
Date Date::FromJDN(long jdn)
{
   int f = jdn + 1401+(((4 * jdn + 274277)/
  ↳146097)*3)/4 -38;
   int e = 4 * f + 3;
   int g = (e % 1461)/4;
   int h = 5 * g + 2;
   Date d;
   d.day = (h % 153)/5 + 1;
   d.month = (h/153 + 2) % 12 + 1;
   d.year = e/1461 - 4716 + (12 + 2 - d.month)/12;
   return d;
}
```

Activity 16.26

The new code for the *Date* declaration is:

```
class Date
{
  private:
    int day;
    int month;
    int year;

    long ToJDN();
    Date FromJDN(long);

  public:
    explicit Date(int=1,int=1,int=2001);
    void SetDate(int, int, int);
    void SetDate(int, int);
    int GetDayOfWeek();
    bool IsLeapYear();
    int GetDay();
    int GetMonth();
    int GetYear();
    Date operator+(int);
};
```

The code for the new method is:

```
//*** Add d days to date ***
Date Date::operator+(int d)
{
  Date dt = FromJDN(ToJDN()+d);
  return dt;
}
```

The code for `main()` is:

```
int main()
{

  Date d1(30,5,2023);

  //*** Add 7 days ***
  d1=d1+7;

  //*** Display date ***
  cout << d1.GetDay() << '/' << d1.GetMonth() <<
  ↳'/' << d1.GetYear() << endl;
}
```

Activity 16.27

The new code for the *Date* declaration is:

```
class Date
{
  private:
    int day;
    int month;
    int year;

    long ToJDN();
    Date FromJDN(long);

  public:
    explicit Date(int=1,int=1,int=2001);
    void SetDate(int, int, int);
    void SetDate(int, int);
    int GetDayOfWeek();
    bool IsLeapYear();
    int GetDay();
    int GetMonth();
    int GetYear();
    Date operator+(int);
    int operator-(Date);
    Date operator-(int);
    bool operator==(Date);
    bool operator!=(Date);
};
```

The code for the three new methods is:

```
//*** Returns a Date calculated by subtracting d
days to date ***
Date Date::operator-(int d)
{
  Date dt = FromJDN(ToJDN() - d);
  return dt;
}


//*** Returns the difference in days between two
↳dates ***
int Date::operator-(Date dt)
{
  int result = ToJDN() - dt.ToJDN();
  return result;
}


//*** Returns true if two dates are equal ***
bool Date::operator==(Date dt)
{
  return (ToJDN() == dt.ToJDN());
}

//*** Returns true if two dates are not equal ***
bool Date::operator!=(const Date dt) const
{
  return (ToJDN() != dt.ToJDN());
}
```

One possible version of `main()` used to test these methods is:

```
int main()
{
  Date d1(30,5,2013);

  // *** New date 7 days before first date ***
  Date d2= d1 - 7;

  //*** Test if dates are equal ***
  if (d1 == d2)
    cout << "The dates are the same\n";
  if (d1 != d2)
  {
    cout << "The dates are different\n";
    cout << "There are " << d1 - d2 <<
    ↳" days between the two dates\n";
  } //*** Display dates ***
  cout << "d1: "<< d1.GetDay() << '/' <<
  ↳d1.GetMonth() << '/' << d1.GetYear() << endl;
  cout << "d2: "<< d2.GetDay() << '/' <<
  ↳d2.GetMonth() << '/' << d2.GetYear() << endl;
}
```

Of course, we could have used `else` rather than a second *if* statement in the code given above – but that would not test the `!=` operator.

Activity 16.28

The final version of *TestCallOperator*:

```
#include <iostream>
using namespace std;

class A
{
  public:
    A() { cout << "Constructor called\n"; };
    int operator()(int, int);
};

int A::operator()(int v, int w)
{
  return (v * w);
}

int main()
{
  A test;
  cout << test(3,5) << endl;
}
```

Activity 16.29

The new *Date* class declaration is:

```
class Date
{
  private:
```

```
    int day;
    int month;
    int year;

    long ToJDN();
    Date FromJDN(long);

  public:
    explicit Date(int=1,int=1,int=2001);
    void SetDate(int, int, int);
    void SetDate(int, int);
    int GetDayOfWeek();
    bool IsLeapYear();
    int GetDay();
    int GetMonth();
    int GetYear();
    Date operator+(int);
    int operator-(Date);
    Date operator-(int);
    bool operator==(Date);
    bool operator!=(Date);
    void operator+=(int);
};
```

The code for the new method is:

```
//*** Adds a given number of days to the date ***
void Date::operator+=(int d)
{
  *this = FromJDN(ToJDN()+d);
}
```

To test the new method, `main()` is rewritten as:

```
int main()
{
  Date d1(23,12,1980);

  //*** Add 10 days ***
  d1 += 10;

  //*** Display date ***
  cout << d1.GetDay() << '/' << d1.GetMonth() << '/'
  ↳<< d1.GetYear() << endl;
}
```

Activity 16.30

Modified coded for *IsLeapYear()*:

```
// *** Returns true if year is a leap year ***
bool Date::IsLeapYear()
{
  return ((this->year%400 == 0)||(this->year%4 == 0
  ↳&& this->year%100 != 0));
}
```

Activity 16.31

The new *Date* class declaration is:

```
class Date
{
  private:
    int day;
    int month;
    int year;

    long ToJDN();
    Date FromJDN(long);

  public:
    explicit Date(int=1,int=1,int=2001);
    void SetDate(int, int, int);
    void SetDate(int, int);
    int GetDayOfWeek();
    bool IsLeapYear();
    int GetDay();
    int GetMonth();
    int GetYear();
    Date operator+(int);
    int operator-(Date);
    Date operator-(int);
    bool operator==(Date);
    bool operator!=(Date);
    void operator+=(int);
    Date operator++();
    Date operator++(int);
    Date operator--();
    Date operator--(int);
};
```

The code for the new methods is:

```
//*** Adds one day to date, returns new date ***
Date Date::operator++()
{
  *this = FromJDN(ToJDN()+1);
  return *this;
}

//*** Adds one day to date, returns original date
↳***
Date Date::operator++(int)
{
  Date result = *this;
  *this = FromJDN(ToJDN()+1);
  return result;
}

//*** Subtracts one day from date, returns new date
↳***
Date Date::operator--()
{
  *this = FromJDN(ToJDN()-1);
  return *this;
}

//*** Subtracts one day from date, returns original
↳date ***
Date Date::operator--(int)
{
  Date result = *this;
  *this = FromJDN(ToJDN()-1);
  return result;
}
```

The code for `main()` is:

```
int main()
{
  Date d1(31, 12, 2000);

  //*** Add a day (prefix) ***
  Date d2 = ++d1;
  //*** Display dates ***
  cout << "PREFIX++\n";
  cout << "d1="<< d1.GetDay() << '/'
  ↳<< d1.GetMonth() << '/' << d1.GetYear() << endl;
  cout << "d2="<< d2.GetDay() << '/'
  ↳<< d2.GetMonth() << '/' << d2.GetYear() << endl;

  //*** Add a day (postfix) ***
  d1 = Date(31, 12, 2000);
  d2 = d1++;
  //*** Display dates ***
  cout << "POSTFIX++\n";
  cout <<"d1=" << d1.GetDay() << '/'
  ↳<< d1.GetMonth() << '/' << d1.GetYear() << endl;
  cout <<"d2=" << d2.GetDay() << '/'
  ↳<< d2.GetMonth() << '/' << d2.GetYear() << endl;

  //*** Subtract a day (prefix) ***
  d1 = Date(31, 12, 2000);
  d2 = --d1;
  //*** Display dates ***
  cout << "PREFIX--\n";
  cout << "d1=" << d1.GetDay() << '/'
  ↳<< d1.GetMonth() << '/'<< d1.GetYear() << endl;
  cout << "d2=" << d2.GetDay() << '/'
  ↳<< d2.GetMonth() << '/'<< d2.GetYear() << endl;

  //*** Subtract a day (postfix) ***
  d1 = Date(31, 12, 2000);
  d2 = d1--;
  //*** Display dates ***
  cout << "POSTFIX--\n";
  cout << "d1=" << d1.GetDay() << '/'
  ↳<< d1.GetMonth() << '/' << d1.GetYear() << endl;
  cout << "d2=" << d2.GetDay() << '/'
  ↳<< d2.GetMonth() << '/' << d2.GetYear() << endl;
}
```

Activity 16.32

When you attempt to compile the code, an error message such as

"Date::operator=(const Date &)" (declared implicitly) cannot be referenced -- it is a deleted function

Activity 16.33

Modified code for *Date* class:

```
class Date
{
  private:
    const int DAYSINYEAR = 365;
    int day;
    int month;
    int year;

    long ToJDN();
    Date FromJDN(long);

  public:

    explicit Date(int=1,int=1,int=2001);
    void SetDate(int, int, int);
    void SetDate(int, int);
    int GetDayOfWeek();
    inline bool IsLeapYear();
    int GetDay();
    int GetMonth();
    int GetYear();
    Date operator+(int);
    int operator-(Date);
    Date operator-(int);
    bool operator==(Date)
    bool operator!=(Date);;
    void operator+=(int);
    Date operator++();
    Date operator++(int);
    Date operator--();
    Date operator--(int);
    Date& operator=(const Date&);
    int GetDaysInYear();
};


//*** Initialises new date object to d/m/y ***
Date::Date(int d, int m, int y)
{
  // *** Days in each month of year ***
  int daysinmonth[] = {0, 31,28,31, 30,31,30,
  ↳31,31,30,31,30,31};

  bool valid = true;  // Parameters valid

  //*** If month invalid, invalid parameters ***
  if (m < 1 || m > 12)
    valid = false;
  //*** Add 1 to days in February if leap year ***
  daysinmonth[2] += (y%400 == 0)||(y%4 == 0 &&
  ↳y%100 != 0);
  //*** If days in month invalid, invalid parameters
  ↳***
  if (d <1 || d > daysinmonth[m])
    valid = false;
  //*** If year less than 1, invalid parameters ***
  if (y < 1)
    valid = false;
  //*** Assign date ***
  if (valid)
  {
    day = d;
    month = m;
    year = y;
  }
  else
  {
    day = 1;
    month = 1;
    year = 2001;
  }
}

//*** Copies date d to date ***
Date& Date::operator=(const Date& d)
{
  day = d.day;
  month = d.month;
  year = d.year;
  return *this;
}

//*** Returns number of days in year ***
int Date::GetDaysInYear()
{
  return DAYSINYEAR + IsLeapYear();
}
```

Modified code for *main()*:

```
int main()
{
  Date d1(31, 12, 2000);
  cout << d1.GetDay() << '/' << d1.GetMonth() << '/'
  ↳<< d1.GetYear() << endl;
  cout << "That year has " << d1.GetDaysInYear()
  ↳<< " days\n";

}
```

Activity 16.34

Modified code for *Date* class:

```
class Date
{
  private:
    int day;
    int month;
    int year;

    long ToJDN();
    Date FromJDN(long);

  public:
    static const int DAYSINYEAR = 365;
    explicit Date(int=1,int=1,int=2001);
    void SetDate(int, int, int);
    void SetDate(int, int);
    int GetDayOfWeek();
    inline bool IsLeapYear();
    int GetDay();
    int GetMonth();
    int GetYear();
    Date operator+(int);
    int operator-(Date);
    Date operator-(int);
    bool operator==(Date);
    bool operator!=(Date);
    void operator+=(int);
    Date operator++();
    Date operator++(int);
    Date operator--();
    Date operator--(int);
    int GetDaysInYear();
};
```

`main()` is coded as:

```
int main()
{
  cout << "DAYSINYEAR value is " << Date::DAYSINYEAR
  ↳<< endl;
  Date d1;
  cout << "DAYSINYEAR value is " << d1.DAYSINYEAR
  ↳<< endl;
}
```

Activity 16.35

Modified code for *Date*:

```
class Date
{
  private:
    static const int DAYSINYEAR = 365;
    int day;
    int month;
    int year;

    long ToJDN();
    Date FromJDN(long);

  public:
    explicit Date(int=1,int=1,int=2001);
    void SetDate(int, int, int);
    void SetDate(int, int);
```

```
    int GetDayOfWeek();
    inline bool IsLeapYear();
    int GetDay();
    int GetMonth();
    int GetYear();
    Date operator+(int);
    int operator-(Date);
    Date operator-(int);
    bool operator==(Date);
    bool operator!=(Date);
    void operator+=(int);
    Date operator++();
    Date operator++(int);
    Date operator--();
    Date operator--(int);
    int GetDaysInYear();
    int GetDAYSINYEARconst();
};
```

New method's code:

```
//*** Returns the value of DAYSINYEAR ***
int Date::GetDAYSINYEARconst()
{
  return DAYSINYEAR;
}
```

Although the first version of `main()`,

```
int main()
{
  Date d1;
  cout << "DAYSINYEAR value is " <<
  ↳d1.GetDAYSINYEARconst() << endl;
}
```

operates correctly, using the expression

```
Date::GetDAYSINYEARconst()
```

will not compile. You can only use the class name in accessing a property of a class if that property (in this case the *GetDAYSINYEARconst()*) is defined as static.

Activity 16.36

Modified code for *Date*:

```
class Date
{
  private:
    static const int DAYSINYEAR = 365;
    int day;
    int month;
    int year;

    long ToJDN();
    Date FromJDN(long);

  public:
    explicit Date(int=1,int=1,int=2001);
    void SetDate(int, int, int);
    void SetDate(int, int);
    int DayOfWeek();
    inline bool IsLeapYear();
    int GetDay();
    int GetMonth();
    int GetYear();
    Date operator+(int);
    int operator-(Date);
    Date operator-(int);
    bool operator==(Date);
    bool operator!=(Date);
    void operator+=(int);
    Date operator++();
    Date operator++(int);
    Date operator--();
    Date operator--(int);
    int GetDaysInYear();
    static int GetDAYSINYEARconst();
};
```

The program will now compile when using the term `Date::GetDAYSINYEARconst()`.

Activity 16.37

Modified code for *Date*:

```
//*** Class declaration ***
class Date
{
  private:
    int day;
    int month;
    int year;

    long ToJDN();
    Date FromJDN(long);

  public:
    static const int DAYSINYEAR = 365;
    static int count;
    explicit Date(int = 1, int = 1, int = 2001);
    void SetDate(int, int, int);
    void SetDate(int, int);
    int GetDayOfWeek();
    bool IsLeapYear();
    int GetDay();
    int GetMonth();
    int GetYear();
    Date operator+(int);
    int operator-(Date);
    Date operator-(int);
    bool operator==(Date);
    bool operator!=(Date);
    void operator+=(int);
    Date operator++();
    Date operator++(int);
    Date operator--();
    Date operator--(int);
    int GetDaysInYear();
    static int GetDAYSINYEARconst();
    static int GetCount();
    ~Date();
};
int Date::count = 0;
```

Modified code for *Date* constructor:

```
//*** Initialise new Date objects to d/m/y ***
Date::Date(int d, int m, int y)
{
  // *** Days in each month of year ***
  int daysinmonth[] = { 0, 31,28,31, 30,31,30,31,
  ↳31,30,31,30,31 };

  bool valid = true;  // Parameters valid

  //*** If month invalid, invalid parameters ***
  if (m < 1 || m > 12)
    valid = false;
  //*** Add 1 to days in February if leap year ***
  daysinmonth[2] += (y % 400 == 0) || (y % 4 == 0
  ↳&& y % 100 != 0);
  //*** If days invalid, invalid parameters ***
  if (d <1 || d > daysinmonth[m])
    valid = false;
  //*** If year less than 1, invalid parameters ***
  if (y < 1)
    valid = false;
  //*** Assign date ***
  if (valid)
  {
    day = d;
    month = m;
    year = y;
  }
  else
  {
    day = 1;
    month = 1;
    year = 2001;
  }
  //*** Add one to count of Date objects **
  count++;
}
```

New methods' code:

```
//*** Returns the number of Date object existing ***
int Date::GetCount()
{
  return count;
}
```

```
//*** Destructor decrements count ***
Date::~Date()
{
  //*** Decrement count of Date objects ***
  count--;
}
```

Code for *main()*:

```
int main()
{
  cout << "There are " << Date::GetCount()
  ↳<< " Date objects\n";
  Date dt1(23, 11, 1963);
  cout << "There are " << Date::GetCount()
  ↳<< " Date objects\n";
  {
    Date dt2;
    cout << "There are " << Date::GetCount()
    ↳<< " Date objects\n";
  }
  cout << "There are " << Date::GetCount()
  ↳<< " Date objects\n";
}
```

The program should display the following lines:

There are 0 Date objects
There are 1 Date objects
There are 2 Date objects
There are 1 Date objects

Activity 16.38

Modified code for *Date*:

```
class Date
{
  private:
    static const int DAYSINYEAR = 365;
    static int count;

    int day;
    int month;
    int year;

    long ToJDN();
    Date FromJDN(long);

  public:
    explicit Date(int=1,int=1,int=2001);
    void SetDate(int, int, int);
    void SetDate(int, int);
    int GetDayOfWeek();
    inline bool IsLeapYear();
    int GetDay();
    int GetMonth();
    int GetYear();
    Date operator+(int);
    int operator-(Date);
    Date operator-(int);
    bool operator==(Date);
    bool operator!=(Date);
    void operator+=(int);
    Date operator++();
    Date operator++(int);
    Date operator--();
    Date operator--(int);
    int GetDaysInYear();
    static int GetDAYSINYEARconst();
    static int GetCount();
    Date::~Date();

    friend ostream& operator<<(ostream&, Date&);
};
int Date::count = 0;
```

Code for *operator<<*:

```
//*** Displays a Date ***
ostream& operator<<(ostream& ct, Date& dt)
{
  ct << dt.day <<'/' << dt.month << '/' << dt.year;
  return ct;
}
```

Code for *main()*:

```
int main()
{
  Date d1(23,11,1963);
  cout << d1 << endl;
}
```

Activity 16.39

Modified code for *Date*:

```
class Date
{
  private:
    static const int DAYSINYEAR = 365;
    static int count;

    int day;
    int month;
    int year;

    long ToJDN();
    Date FromJDN(long);

  public:
    explicit Date(int=1,int=1,int=2001);
    void SetDate(int, int, int);
    void SetDate(int, int);
    int GetDayOfWeek();
    inline bool IsLeapYear();
    int GetDay();
    int GetMonth();
    int GetYear();
    Date operator+(int);
    int operator-(Date);
    Date operator-(int);
    bool operator==(Date);
    bool operator!=(Date);
    void operator+=(int);
    Date operator++();
    Date operator++(int);
    Date operator--();
    Date operator--(int);
    int GetDaysInYear();
    static int GetDAYSINYEARConst();
    static int GetCount();
    Date::~Date();

    friend ostream& operator<<(ostream&, Date&);
    friend istream& operator>>(istream&, Date&);

};
int Date::count = 0;
```

Code for *operator>>*:

```
//*** Accepts a Date value from the keyboard ***
istream& operator>>(istream& cn, Date& dt)
{
  scanf("%d/%d/%d",&dt.day, &dt.month, &dt.year);
  return cn;
}
```

Code for *main()*:

```
int main()
{
  Date d1;
  cout << "Enter date : ";
  cin >> d1;
  cout << d1 << endl;
}
```

Activity 16.40

Modified code for *operator>>*:

```
//*** Accepts a Date value from the keyboard ***
istream& operator>>(istream& cn, Date& dt)
{
  int d,m,y;
  //*** Read date ***
  scanf("%d/%d/%d",&d, &m, &y);
  //*** Transfer values to date object ***
  dt.SetDate(d,m,y);
  return cn;
}
```

Activity 16.41

Modified code for *DateClass*:

```
class Date
{
  private:
    static const int DAYSINYEAR = 365;
    static int count;

    int day;
    int month;
    int year;

    long ToJDN();
    Date FromJDN(long);

  public:
    explicit Date(int=1,int=1,int=2001);
    void SetDate(int, int, int);
    void SetDate(int, int);
    int GetDayOfWeek();
    inline bool IsLeapYear();
    int GetDay();
    int GetMonth();
    int GetYear();
    Date operator+(int);
    int operator-(Date);
    Date operator-(int);
    bool operator==(Date);
    bool operator!=(Date);
    void operator+=(int);
    Date operator++();
    Date operator++(int);
    Date operator--();
    Date operator--(int);
    int GetDaysInYear();
    static int GetDAYSINYEARConst();
    static int GetCount();
    Date::~Date();

    friend ostream& operator<<(ostream&, Date&);
    friend istream& operator>>(istream&, Date&);
    friend Date operator+(int, Date&);
};
int Date::count = 0;
```

Code for *operator+*:

```
Date operator+(int days, Date& d)
{
  Date dt = d.FromJDN(d.ToJDN() + days);
  return dt;
}
```

Test code in *main()*:

```
int main()
{
  Date d1(23,11,1963);
  Date d2;
  d2 = d1 + 10;
  cout << d2 << endl;
}
```

Activity 16.42

Code for *StudentClass*:

```
#include <iostream>
using namespace std;

class Student
{
  private:
    char name[21]{ "" };
    char sex{ '?' };
    bool attending{ true };
    int marks[6]{ 0 };
  public:
    Student(const char*, char, bool = true);
    void SetName(char*);
    void SetSex(char);
    void SetAttending(bool);
    void SetMark(int, int);
    char* GetName();
    char GetSex();
    bool GetAttending();
    int GetMark(int);
    double GetAverageMark();
    friend ostream& operator<<(ostream&, Student&);
};

Student::Student(const char* n, char s, bool at)
{
  strcpy(name, n);
  s = toupper(s);
  if (s == 'M' || s == 'F')
    sex = s;
  attending = at;
}

//*** Set the student's name to n ***
void Student::SetName(char * n)
{
  strcpy(name, n);
}

//*** Set student's sex to s ***
void Student::SetSex(char s)
{
  s = toupper(s);
  if (s == 'M' || s == 'F')
    sex = s;
}

//*** Set student's attendence status ***
void Student::SetAttending(bool at)
{
  attending = at;
}

//*** Set student's mark for specified exam ***
void Student::SetMark(int mrk, int ex)
{
  //*** If not valid exam, return ***
  if (ex < 0 || ex > 5)
    return;
  //*** If not valid mark, return ***
  if (mrk < 0 || mrk > 100)
    return;
  //*** Store mark ***
  marks[ex] = mrk;
}

//*** Return student's name ***
char* Student::GetName()
{
  return name;
}

//*** Return student's sex ***
char Student::GetSex()
{
  return sex;
}

//*** Return student's attendence status ***
bool Student::GetAttending()
{
  return attending;
}

//*** Return student's exam mark ***
int Student::GetMark(int ex)
{
  if (ex < 0 || ex > 5)
    return -1;
  return marks[ex];
}

//*** Return student's average mark ***
double Student::GetAverageMark()
{
  double total{ 0.0 };
  for (auto v : marks)
    total += v;
  return total / 6.0;
}


//*** Display contents of student object ***
ostream& operator<<(ostream& ct, Student& st)
{
  ct << st.name << "  " << st.sex << "  ";
  if (st.attending)
    ct << "attending  ";
  else
    ct << "not attending  ";
```

```
  for (int ex = 0; ex < 6; ex++)
    ct << st.marks[ex] << "  ";
  ct << st.GetAverageMark() << endl;
  return ct;
}


int main()
{
  Student st1("Madeline Bray", 'F');
  Student st2("Nicholas Nickleby",'M');
  Student st3("Ada Clare", 'F', false);
  Student st4("Richard Carstone", 'M');
  //*** Assign random marks ***
  for (int ex = 0; ex < 6; ex++)
    st1.SetMark(rand() % 101, ex);
  for (int ex = 0; ex < 6; ex++)
    st2.SetMark(rand() % 101, ex);
  for (int ex = 0; ex < 6; ex++)
    st3.SetMark(rand() % 101, ex);
  for (int ex = 0; ex < 6; ex++)
    st4.SetMark(rand() % 101, ex);
  //*** Display object contents ***
  cout << st1 << endl;
  cout << st2 << endl;
  cout << st3 << endl;
  cout << st4 << endl;
}
```

Activity 16.43

Modified code in *StudentClass*:

Changes to *Student* class declaration:

```
class Student
{
  private:
    char* name{ nullptr };
    char sex{ '?' };
    bool attending{ true };
    int marks[6]{ 0 };
  public:
    Student(const char*, char, bool = true);
    void SetName(char*);
    void SetSex(char);
    void SetAttending(bool);
    void SetMark(int, int);
    char* GetName();
    char GetSex();
    bool GetAttending();
    int GetMark(int);
    double GetAverageMark();
    ~Student();

    friend ostream& operator<<(ostream&, Student&);
};
```

Changes to *Student* constructor:

```
Student::Student(const char* n, char s, bool at)
{
  name = new char[strlen(n) + 1];
  strcpy(name, n);
  s = toupper(s);
  if (s == 'M' || s == 'F')
    sex = s;
  attending = at;
}
```

Changes to *Student SetName()*:

```
//*** Set the student's name to n ***
void Student::SetName(char * n)
{
  //*** If no new name, exit ***
  if (n == nullptr)
    return;
  //*** Delete any space already being used ***
  if (name != nullptr)
    delete[]name;
  //*** Create enough space for the new name ***
  name = new char[strlen(n) + 1];
  //*** Copy name into this new space ***
  strcpy(n, name);
}
```

New code for *Student* destructor:

```
Student::~Student()
{
  if(name != nullptr)
    delete [] name;
}
```

Activity 16.44

The program is likely to crash or show unpredictable results.

Activity 16.45

Modified code in *StudentClass*:

Changes to *Student* class declaration:

```
class Student
{
  private:
    char* name{ nullptr };
    char sex{ '?' };
    bool attending{ true };
    int marks[6]{ 0 };
  public:
    Student(const char*, char, bool = true);
    void SetName(const char*);
    void SetSex(char);
    void SetAttending(bool);
    void SetMark(int, int);
    char* GetName();
    char GetSex();
    bool GetAttending();
    int GetMark(int);
    double GetAverageMark();
    Student& operator=(Student&);
    ~Student();

    friend ostream& operator<<(ostream&, Student&);
};
```

New code for *Student operator==()*:

```
//*** Copies a Details object to current Details
object ***
Student& Student::operator=(Student& st)
{
  //*** Delete any space already being used ***
  if (name != nullptr)
    delete[]name;
  //*** Set up space for copy of name ***
  int size = strlen(st.name) + 1;
  name = new char[size];
  //*** Copy name ***
  strcpy(name, st.name);
  //*** Copy sex ***
  sex = st.sex;
  //*** Copy attending ***
  attending = st.attending;
  //*** Copy marks ***
  for (int c = 0; c < 6; c++)
    marks[c] = st.marks[c];
  //*** Return reference to updated object ***
  return *this;
}
```

The program should operate correctly this time.

Activity 16.46

Modified code in *StudentClass*:

Changes to *Student* class declaration:

```
class Student
{
  private:
    char* name{ nullptr };
    char sex{ '?' };
    bool attending{ true };
    int marks[6]{ 0 };
  public:
    Student(const char*, char, bool = true);
    Student(const Student&);
    void SetName(const char*);
    void SetSex(char);
    void SetAttending(bool);
```

```
    void SetMark(int, int);
    char* GetName();
    char GetSex();
    bool GetAttending();
    int GetMark(int);
    double GetAverageMark();
    Student& operator=(Student&);
    ~Student();

    friend ostream& operator<<(ostream&, Student&);
};
```

Code for copy constructor:

```
Student::Student(const Student& st)
{
  //*** Delete any space already being used ***
  if (name != nullptr)
    delete[]name;
  //*** Set up space for copy of name ***
  int size = strlen(st.name) + 1;
  name = new char[size];
  //*** Copy name ***
  strcpy(name, st.name);
  //*** Copy sex ***
  sex = st.sex;
  //*** Copy attending ***
  attending = st.attending;
  //*** Copy marks ***
  for (int c = 0; c < 6; c++)
    marks[c] = st.marks[c];
}
```

Modified code for *main()*:

```
int main()
{
  Student st1("Madeline Bray", 'F');
  Student st2("Nicholas Nickleby",'M');
  Student st3("Ada Clare", 'F', false);
  Student st4("Richard Carstone", 'M');
  //*** Assign random marks ***
  for (int ex = 0; ex < 6; ex++)
    st1.SetMark(rand() % 101, ex);
  for (int ex = 0; ex < 6; ex++)
    st2.SetMark(rand() % 101, ex);
  for (int ex = 0; ex < 6; ex++)
    st3.SetMark(rand() % 101, ex);
  for (int ex = 0; ex < 6; ex++)
    st4.SetMark(rand() % 101, ex);
  //*** Create new object equal to st1 ***
  Student st5(st1);
  //*** Display contents of both objects ***
  cout << st1 << endl;
  cout << st5 << endl;
}
```

Activity 16.47

Modified code in *StudentClass*:

Changes to prototype of *operator=():*

```
Student& operator=(const Student&);
```

Changes to the code for *operator=()*:

```
Student& operator=(const Student& st)
```

Activity 16.48

Modified code in *StudentClass*:

Prototype for *GetNameSize()*:

```
//*** Standard function prototypes ***
int GetNameSize(const Student&);
```

The above is placed after the code for all operations of the *Student* class.

Code for *GetNameSize()*:

```
//************************************
//***      Standard functions      ***
//************************************

//*** Returns number of characters in st.name ***
int GetNameSize(const Student& st)
{
  return strlen(st.GetName());
}
```

The above is placed after the code for `main()`.

Code for *main()*:

```
int main()
{
  Student st1("Madeline Bray", 'F');
  Student st2("Nicholas Nickleby",'M');
  Student st3("Ada Clare", 'F', false);
  Student st4("Richard Carstone", 'M');
  //*** Assign random marks ***
  for (int ex = 0; ex < 6; ex++)
    st1.SetMark(rand() % 101, ex);
  for (int ex = 0; ex < 6; ex++)
    st2.SetMark(rand() % 101, ex);
  for (int ex = 0; ex < 6; ex++)
    st3.SetMark(rand() % 101, ex);
  for (int ex = 0; ex < 6; ex++)
    st4.SetMark(rand() % 101, ex);
  //*** Display number of chars in st1's name ***
  cout << st1.GetName() << " contains "
  ↳<< GetNameSize(st1) << " characters\n";
}
```

The code fails to compile because the parameter to *GetNameSize()* is defined as a constant.

Activity 16.49

Modified code in *StudentClass*:

```
#include <iostream>
using namespace std;

class Student
{
  private:
    char* name{ nullptr };
    char sex{ '?' };
    bool attending{ true };
    int marks[6]{ 0 };
  public:
    Student(const char*, char, bool = true);
    Student(const Student&);
    void SetName(const char*);
    void SetSex(char);
    void SetAttending(bool);
    void SetMark(int, int);
    char* GetName() const;
    char GetSex() const;
    bool GetAttending() const;
    int GetMark(int) const ;
    double GetAverageMark() const;
    Student& operator=(const Student&);
    ~Student();

    friend ostream& operator<<(ostream&, Student&);
};

Student::Student(const char* n, char s, bool at)
{
  name = new char[strlen(n) + 1]; //Extra byte for
                                  ↳final null
  strcpy(name, n);
  s = toupper(s);
  if (s == 'M' || s == 'F')
    sex = s;
  attending = at;
}

Student::Student(const Student& st)
{
  //*** Delete any space already being used ***
  if (name != nullptr)
    delete[]name;
  //*** Set up space for copy of name ***
  int size = strlen(st.name) + 1;
  name = new char[size];
  //*** Copy name ***
  strcpy(name, st.name);
  //*** Copy sex ***
  sex = st.sex;
  //*** Copy attending ***
```

```
  attending = st.attending;
  //*** Copy marks ***
  for (int c = 0; c < 6; c++)
    marks[c] = st.marks[c];
}

//*** Set the student's name to n ***
void Student::SetName(const char * n)
{
  //*** If no new name, exit ***
  if (n == nullptr)
    return;
  //*** Delete any space already being used ***
  if (name != nullptr)
    delete[]name;
  //*** Create enough space for the new name ***
  name = new char[strlen(n) + 1]; //Extra byte for
                                  ↳final null
  //*** Copy n into this new space ***
  strcpy(name,n);

}

//*** Set student's sex to s ***
void Student::SetSex(char s)
{
  s = toupper(s);
  if (s == 'M' || s == 'F')
    sex = s;
}

//*** Set student's attendence status ***
void Student::SetAttending(bool at)
{
  attending = at;
}

//*** Set student's mark for specified exam ***
void Student::SetMark(int mrk, int ex)
{
  //*** If not valid exam, return ***
  if (ex < 0 || ex > 5)
    return;
  //*** If not valid mark, return ***
  if (mrk < 0 || mrk > 100)
    return;
  //*** Store mark ***
  marks[ex] = mrk;
}

//*** Return student's name ***
char* Student::GetName() const
{
  return name;
}

//*** Return student's sex ***
char Student::GetSex() const
{
  return sex;
}

//*** Return student's attendence status ***
bool Student::GetAttending() const
{
  return attending;
}

//*** Return student's exam mark ***
int Student::GetMark(int ex) const
{
  if (ex < 0 || ex > 5)
    return -1;
  return marks[ex];
}

//*** Return student's average mark ***
double Student::GetAverageMark() const
{
  double total{ 0.0 };
  for (auto v : marks)
    total += v;
  return total / 6.0;
}

//*** Copies a Details object to current Details
object ***
Student& Student::operator=(const Student& st)
{
  //*** Delete any space already being used ***
  if (name != nullptr)
    delete[]name;
  //*** Set up space for copy of name ***
  int size = strlen(st.name) + 1;
  name = new char[size];
  //*** Copy name ***
  strcpy(name, st.name);
  //*** Copy sex ***
  sex = st.sex;
  //*** Copy attending ***
  attending = st.attending;
  //*** Copy marks ***
  for (int c = 0; c < 6; c++)
    marks[c] = st.marks[c];
  //*** Return reference to updated object ***
  return *this;
}


Student::~Student()
{
  if (name != nullptr)
    delete[] name;
}


//*** Display contents of student object ***
ostream& operator<<(ostream& ct, Student& st)
{
  ct << st.name << "  " << st.sex << "  ";
  if (st.attending)
    ct << "attending  ";
  else
    ct << "not attending  ";
  for (int ex = 0; ex < 6; ex++)
    ct << st.marks[ex] << "  ";
  ct << st.GetAverageMark() << endl;
  return ct;
}

//*** Standard function prototypes ***
int GetNameSize(const Student&);


int main()
{
  Student st1("Madeline Bray", 'F');
  Student st2("Nicholas Nickleby",'M');
  Student st3("Ada Clare", 'F', false);
  Student st4("Richard Carstone", 'M');
  //*** Assign random marks ***
  for (int ex = 0; ex < 6; ex++)
    st1.SetMark(rand() % 101, ex);
  for (int ex = 0; ex < 6; ex++)
    st2.SetMark(rand() % 101, ex);
  for (int ex = 0; ex < 6; ex++)
    st3.SetMark(rand() % 101, ex);
  for (int ex = 0; ex < 6; ex++)
    st4.SetMark(rand() % 101, ex);
  //*** Display number of chars in st1's name ***
  cout << st1.GetName() << " contains "
  ↳<< GetNameSize(st1) << " characters\n";
}


//*** Returns number of characters in st.name ***
int GetNameSize(const Student& st)
{
  return strlen(st.GetName());
}
```

Activity 16.50

The compiler objects to the code trying to return a `char*` value from *Trivial*::*GetName()*.

Activity 16.51

The program now works as expected.

Activity 16.52

Changes to *StudentClass*:

Modified code for *main()*:

```
int main()
{
   Student *st1{ new Student("Madeline Bray", 'F') };
   Student * st2{ new Student("Nicholas Nickleby",
   'M') };
   Student st3("Ada Clare", 'F', false);
   Student st4("Richard Carstone", 'M');
   //*** Assign random marks ***
   for (int ex = 0; ex < 6; ex++)
      st1->SetMark(rand() % 101, ex);
   for (int ex = 0; ex < 6; ex++)
      st2->SetMark(rand() % 101, ex);
   for (int ex = 0; ex < 6; ex++)
      st3.SetMark(rand() % 101, ex);
   for (int ex = 0; ex < 6; ex++)
      st4.SetMark(rand() % 101, ex);
   //*** Display object contents ***
   cout << *st1 << endl;
   cout << *st2 << endl;
   cout << st3 << endl;
   cout << st4 << endl;
}
```

Activity 16.53

Changes to *StudentClass*:

Modified code for *main()*:

```
int main()
{
   Student group[4] {Student("Madeline Bray", 'F'),
   Student("Nicholas Nickleby",'M'),
   Student("Ada Clare", 'F', false),
   Student("Richard Carstone", 'M') };
   //*** Assign random marks ***
   for(int idx = 0; idx < 4; idx ++)
      for (int ex = 0; ex < 6; ex++)
         group[idx].SetMark(rand() % 101, ex);
   //*** Display objects' contents ***
   for (auto v: group)
      cout << v << endl;
}
```

Activity 16.54

Changes to *StudentClass*:

Modified code for *main()*:

```
int main()
{

   Student *group[4]
   {
      new Student("Madeline Bray", 'F'),
      new Student("Nicholas Nickleby",'M'),

      new Student("Ada Clare", 'F', false),
      new Student("Richard Carstone", 'M')
   };
   //*** Assign random marks ***
   for(int idx = 0; idx < 4; idx ++)
      for (int ex = 0; ex < 6; ex++)
         group[idx]->SetMark(rand() % 101, ex);
   //*** Display objects' contents ***
   for (auto v: group)
      cout << *v << endl;
}
```

Activity 16.55

Code for *TimeClass*'s definition of *Time*:

```
#include <iostream>
using namespace std;


class Time
{
  private:
     int hour;
     int minute;
     int second;

     int ToSeconds() const;
     Time FromSeconds(int) const;

  public:
     Time(int = 0, int = 0, int = 1);
     void SetTime(int, int, int);
     int GetHour() const;
     int GetMinute() const;
     int GetSecond() const;
     Time operator+(const Time&) const;
     Time& operator+=(const Time&);
     Time operator-(const Time&) const;
     Time& operator-=(const Time&);
     Time& operator++();
     Time operator++(int);
     Time& operator--();
     Time operator--(int);
     bool operator== (const Time&)const;
     bool operator!= (const Time&)const;
     bool operator<(const Time&)const;
     bool operator<=(const Time&)const;
     bool operator>(const Time&)const;
     bool operator>=(const Time&)const;

     friend ostream& operator<<(ostream&,
     const Time&);
};
```

Activity 16.56

Complete code for *TimeClass*:

```
#include <iostream>
using namespace std;


class Time
{
  private:
     int hour;
     int minute;
     int second;

     int ToSeconds() const;
     Time FromSeconds(int) const;

  public:
     Time(int = 0, int = 0, int = 1);
     void SetTime(int, int, int);
     int GetHour() const;
     int GetMinute() const;
     int GetSecond() const;
     Time operator+(const Time&) const;
     Time& operator+=(const Time&);
     Time operator-(const Time&) const;
     Time& operator-=(const Time&);
     Time& operator++();
     Time operator++(int);
     Time& operator--();
     Time operator--(int);
     bool operator== (const Time&)const;
     bool operator!= (const Time&)const;
     bool operator<(const Time&)const;
     bool operator<=(const Time&)const;
     bool operator>(const Time&)const;
     bool operator>=(const Time&)const;

     friend ostream& operator<<(ostream&,
     const Time&);
};

//************************************************
//***        Time Class Implementation         ***
//************************************************

//*********************************
//***      Public Methods       ***
//*********************************
//*** Initalise time to h:m:s if valid else 0:0:0***
Time::Time(int h, int m, int s)
{
   if (h < 0 || m < 0 || m > 59 || s < 0 || s > 59)
   {
      hour = 0;
      minute = 0;
      second = 0;
   }
   else
   {
      hour = h;
      minute = m;
      second = s;
   }
```

```
}


//*** Sets time to h:m:s if this is valid ***
void Time::SetTime(int h, int m, int s)
{
  if (h < 0 || m < 0 || m > 59 || s < 0 ||
  ↳s > 59)
  {
    return;
  }
  else
  {
    hour = h;
    minute = m;
    second = s;
  }
}


//*** Returns hour value ***
int Time::GetHour() const
{
  return hour;
}


//*** Returns hour value ***
int Time::GetMinute() const
{
  return minute;
}


//*** Returns hour value ***
int Time::GetSecond() const
{
  return second;
}


//*** Returns time of t + current time ***
Time Time::operator+(const Time& t) const
{
  return FromSeconds(ToSeconds() + t.ToSeconds());
}


//*** Adds time t to current time ***
Time& Time::operator+=(const Time& t)
{
  return *this = FromSeconds(ToSeconds() +
t.ToSeconds());
  return *this;
}


//*** Returns current time  - t ***
Time Time::operator-(const Time& t) const
{
  return FromSeconds(ToSeconds() - t.ToSeconds());
}


//*** Subtracts time t from current time ***
Time& Time::operator-=(const Time& t)
{
  return *this = FromSeconds(ToSeconds() -
  ↳t.ToSeconds());
  return *this;
}


//*** Adds 1 sec to time (pre) ***
Time& Time::operator++()
{
  *this = FromSeconds(ToSeconds() + 1);
  return *this;
}


//*** Adds 1 sec to time (post) ***
Time Time::operator++(int)
{
  Time result{ *this };
  *this = FromSeconds(ToSeconds() + 1);
  return result;
}


//*** Subtracts 1 sec from time (pre) ***
Time& Time::operator--()
{
  *this = FromSeconds(ToSeconds() - 1);
  return *this;
}


//*** Subtracts 1 sec from time (post) ***
Time Time::operator--(int)
{
  Time result{ *this };
  *this = FromSeconds(ToSeconds() - 1);
  return result;
}


//*** Returns true if times are equal ***
bool Time::operator==(const Time& t) const
{
  return (ToSeconds() == t.ToSeconds());
}


//*** Returns true if times are not equal ***
bool Time::operator!=(const Time& t) const
{
  return (ToSeconds() != t.ToSeconds());
}


//*** Returns true if time < t ***
bool Time::operator<(const Time& t) const
{
  return (ToSeconds() < t.ToSeconds());
}


//*** Returns true if time <= t ***
bool Time::operator<=(const Time& t) const
{
  return (ToSeconds() <= t.ToSeconds());
}


//*** Returns true if time > t ***
bool Time::operator>(const Time& t) const
{
  return (ToSeconds() > t.ToSeconds());
}


//*** Returns true if time >= t ***
bool Time::operator>=(const Time& t) const
{
  return (ToSeconds() >= t.ToSeconds());
}


//*********************************
//***     Private Methods       ***
//*********************************

//*** Converts current time to seconds ***
int Time::ToSeconds() const
{
  return hour * 3600 + minute * 60 + second;
}

//*** Converts seconds to hrs, mins, secs ***
Time Time::FromSeconds(int secs) const
{
  if (secs < 0)
    return Time(0, 0, 0);
  int h = secs / 3600;
  int m = (secs - h * 3600) / 60;
  int s = secs % 60;

  return Time(h, m, s);
}


//*********************************
//***         Friends           ***
//*********************************

//*** Displays the value of time t ***
ostream& operator<<(ostream& ct, const Time& t)
{
  char timestr[9];
```

```
  sprintf(timestr, "%02d:%02d:%02d", t.hour,
  ↳t.minute, t.second);
  ct << timestr;
  return ct;
}


int main()
{
  Time tm(2, 9, 6);
  Time tm2(6, 12, 57);
  tm += tm2;
  cout << tm << endl;
  if (tm < Time(9, 0, 0))
    cout << "Total time is less than nine hours\n";
  else
    cout << "Total time is nine hours or more\n";
}
```

The above code was executed twice with a total time less than 9 hours and again with a total time greater than 9 hours.