

## SESSION 1-5: ABSTRACT DATA TYPES

An abstract data type may be taken to mean a user defined extension to the native data types available in C++. In simple terms, it is a programmer defined user data type. Abstract data types usually consist of a set of values and a collection of operations to be performed on the values. Below are some of the main user defined data types.

### 1-5.1 Enumeration Types

The reserved word, *enum*, is used in C++ to declare a distinct integer data type for a set of named constants called enumerators. The enumerators are by default assigned the values 0, 1, 2, etc according to the number of items in the set but it is also possible to initialise them to arbitrary **integer** (int) constants. The general format of enum declaration is as follows:

```
enum identifier {enumerators};
```

where identifier (also known as tag name) is the name of the data type being defined as enum and enumerators is a set of named constants separated by commas. As an example, let us define a variable that can store someone's marital status. Let the variable be MaritalStatus and let its possible values be Single, Married, Divorced, Widow, Widower and Separated. We can define MaritalStatus as enum as follows:

```
enum MaritalStatus={Single, Married, Divorced, Widow, Widower, Separated};
```

The enumerators, Single, Married, Divorced, Widow, Widower and Separated will have the values 0, 1, 2, 3, 4 and 5 respectively. Note that a variable declared to be of MaritalStatus data type cannot have any other values apart from its allowable ones, that is, its enumerators. Enumerators can be implicitly converted to ordinary integer types but not vice versa.

In the above example, if one initialises Widow to say 10, that is  

```
enum MaritalStatus {Single, Married, Divorced, Widow=10, Widower, Separated};
```

  
then the Widower and Separated will have the values 11 and 12 respectively. The others will have their usual default values. This is known as the default rule. The enumerators need not necessarily have consecutive numbers.

### 1-5.2 Structures

So far, we have considered only data structures or types that were included by the C++ developers such as the int, char and float. Variables of these data types can store only one type of data in a program. C++ allows a programmer to define his own data types in addition to those provided by the C++ developers. Such data types are referred to as user-defined data structures or types. The user defined data structure allows programmers to define a collection of variables of different data types. This is achieved by using the struct statement. The general format of the struct statement is as follows:

```

struct UserDataType
{
    dataType1 variable1;
    dataType2 variable2;
    :
    dataTypen variablen;
};

```

where the UserDataType is the name the programmer wishes to give to the new data type being defined. This is usually referred to as **structure tag**. The dataType1, dataType2, etc are either C++ data types or user defined data types. The variable1, variable2, etc are user defined variables and are usually referred to as member names. Note that member names can be declared to be protected, public or private, however, it is better to assume their default scope, public hence we will assume for this work that member names are all public variables. By default all member names of a struct are public. See below for more information on protected, public and private member names.

Generally, a user-defined data type is best placed before the start of the main function, that is before the int main() statement so that the main function and other user defined functions can declare variables to be of this new user defined data type.

Once a user defined data type is declared, it is used in the same way as the other data types such as the int, char and double within the main function or user defined functions.

As an example, let us assume that in a certain company, the company has the following details about her employees: employee number (integer), name (string), age (integer), address (string), rank (char) and salary (integer) where the items in bracket define the data type. We can declare a data structure for the above as follows:

```

struct EmpRecord
{
    int EmployeeNumber;
    char Name[50];
    int Age;
    char Address[80];
    char Rank;
    integer Salary;
};

```

In the above structure definition, the variables EmployeeNumber, Name, Age, etc are called member names. The figures 50 and 80 are arbitrary numbers chosen to represent the maximum number of characters that a name and an address can have respectively.

Now that we have defined a new data structure, we can declare variables to be of this type. For example, we can declare the variable `StaffRecord` to be of `EmpRecord` data type as follows:

```
EmpRecord StaffRecord;
```

**Example 2:** Assume that in a program in which student records are to be processed, if for each student we have student ID number, year of programme, age, mark1, mark2, mark3 and the weighted average, then we can define a data type as follows if we use `Student_ID`, `year`, `Age`, `mark1`, `mark2`, `mark3` and `WeightAve` as the variables to store the students' details:

```
struct studentRecType
{   char Student_ID[8];
    int year;
    int Age, mark1, mark2, mark3;
    float WeightAve:real
}
```

## Referencing a member name

Having defined variables to be of a user defined data type, one can reference the member names using the following syntax:

```
variableName.memberName;
```

where *variableName* is the name of the user-defined variable type and the *memberName* is the name of a member in the *variableName*.

As an example, let's define a data type, *EmpRecord* using the struct statement and then define a variable *StaffRecord* to be of the `EmpRecord`. If *salary* is a member name of *EmpRecord* then one can assign the data 1,235,000 to the member salary using the following statement:

```
StaffRecord.salary = 1235000;
```

## Initializing structures

Just as variables of a predefined data type can be initialised during declarations, one can also initialise variables of a user defined data type. The general format for doing this is as follows:

```
dataType variable = {value1, value2, ..., valuen};
```

where the *dataType* is the user-defined data type, *variable* is the name of the variable of the data type *dataType*. The *Value1*, *value2*, etc are the values to be assigned to the member names. The values must be specified in the order that corresponds to the order of the member names or variables as they appear in the structure type definition.

As an example, let's define a data type, EmpRecord using the struct statement and then define a variable StaffRecord to be of the EmpRecord. If EmployeeNumber, name, year, age, rank and salary are member names then these can be initialised to 123450, Maame Obu, 1990, 30, S and 2020000 respectively using the following statements:

```
EmpRecord staffRecord = {123450, 'Maame Obu', 1990, 31, 'S', 2020000};
```

**Question:** Write a C++ program that will allow name, description, quantity and the unit cost of items to be entered into memory and then to generate an output showing the item names, quantity, the unit cost and the total cost. You are required to use make of the struct statement.

**Solution:**

We shall assume that an item name and the description will be at most 40 characters long each. The program may be as follows

```
#include<iostream>
using namespace std;
// declare a data type for the items
struct good
{
    char name[40];
    char desc[40];
    int quantity;
    int cost;
};
int main()
{
    char space[10]="    ";
    char answer[2] = "N";
    int j,i=1;
    good item[100]; // variable item is an array of the data type good
    cout <<"Please provide the following information for each product \n ";
    cout <<"Name, description, quantity and unit cost \n";
    // read all item information and store in memory
    do
    {
        cout <<"For item " <<i <<space;
        cin >> item[i].name >>item[i].desc >>item[i].quantity >>item[i].cost;
        i=i+1;
        cout <<"Please Enter Y to end loop";
        cin>>answer;
```

```

        } while (answer=="N");
// output the needed information
    cout <<space <<"Item Name" <<space <<"Quantity" <<"Cost" <<space;
    cout <<"Total Cost \n";
    for(j=1;j<i;j++)
    {
        cout <<space <<item[j].name <<space <<item[j].quantity <<space
        cout <<item[j].cost <<space; <<(item[j].cost *item[j].quantity) ;
        cout <<endl;
    }
    return 0;
}

```

The above program illustrates how the struct statement is used and how a user defined data type can also be used to declare arrays of that data type.

## STRUCTURE WITHIN A STRUCTURE

C++ allows for a structure to be defined within another structure, that is nested structures. Let us consider a typical car as an example and then we will see how we can define a structure within a structure. For any given car, the information available are the car make, model, year (day, month, year), owner (name, address), registration number and insurers (name, address). The items in brackets indicate the breakdown of the main items that they follow. This means that for a car owner, we have the name and the address of the owner. Note that the order in which structures are listed in a program are very important. If a structure A is to be used within another structure B, then the declaration of the structure A must come before that of B. If a structure is used as a data type before it is defined then the (Bloodshed Dev C++) compiler will display the following error messages:

```

error C2146: syntax error : missing ';' before identifier 'structure'
error C2501: 'structure' : missing storage-class or type specifiers

```

where *structure* is the name of the structure being used in another structure but whose definition comes later.

For our car example, if we are to define a structure for a car then we will have a structure within a structure. The following is a typical example of the car structure within a structure:

```

struct date //to be used as data type for year in car
{
    int day,month,year;
}

```

```

};

struct owner //to be used for owner information in car
{
    char name[50];
    char address[80];
};

struct insurer //to be used for insurers information in car
{
    char name[50];
    char address[80];
};

struct car
{
    char make[20];
    char model[20];
    date year;
    owner Owner;
    char RegNo[10];
    insurer insurers;
};

```

In the body of the *car* structure, the bolded data types are the structures within the car structure. The values in square brackets are arbitrary and are used to specify the number of characters that the string members can contain.

## ACCESSING MEMBER NAMES OF STRUCTURE WITHIN STRUCTURE

Accessing or referencing member names of structures within a structure is similar to accessing a member name within a structure. The syntax for accessing a member of a structure within another structure is as follows:

MotherStruct.ChildStruct.ChildMemberName

where *MotherStruct* is the name of the structure that contains another structure. The *ChildStruct* is the name of the structure that is within another structure. The *ChildMemberName* is the name of the member of the structure that is within another structure. As an example, let us assume that in

a program that contain the car structure above, we declare a variable CarInfo to have the data type car, then we can refer to the car owner's name as follows:

```
CarInfo.owner.name
```

It can be seen from above that the *CarInfo* structure has the *owner* structure within it hence the *CarInfo* is a 'mother' to the *owner* and hence why that owner's name is accessed as such.

## STRUCTURES AS FUNCTION ARGUMENTS

It is possible for a function to have a call-by-value parameters and/or call-by-reference parameters of a structure type. It is also possible for a function to have a structure type as the return value. The general syntax for using a structure as an argument and/or a return value type is as follows:

```
structureType functionName (structureArguments)
{
    function statement;
    return structureResult;
}
```

where *structureType* is a previously defined structure data type, *structureArguments* is the list of arguments defined to be of a structure type. The *structureReturn* is also a value to be return and is of a structure type. Note that if the return type is a structure type then it does not imply that all variables used within the function should also be of the structure type.

## UNIONS

The use and declaration of unions are same as that of structures however their functionalities are different. One major advantage in the use of a union is that all its members are stored at the same memory location. The syntax of a union is as follows:

```
union UserDataTypeName
{
    dataType1 variable1;
    dataType2 variable2;
    :
    dataTypen variablen;
};
```

where *UserdataTypeName* is the name of the union and *variable1,...,variablen* are its members. Just like the struct, once a union has been defined other variables can be defined to be of this type for later use in a program. Even though all members of a union share the same memory location,

modifying the value of any of them does affects all the others. Let us consider the following example:

```
#include <iostream>
#include <cstdlib>
using namespace std;
union newtype
{int amount;
 int balance;
};

int main()
{   newtype tryme;
    cout <<"Please enter two integers ";
    cin >> tryme.amount >>tryme.balance;
    cout<<"The two values add up to " <<tryme.amount+tryme.balance
    <<endl;
    system("PAUSE");
    return 0;
}
```

### Program run

```
Please enter two integers 200 144
The two values add up to 288
Press any key to continue . . .
```

It can be seen from the above program run that instead of the program displaying the sum of 200 and 144 as 344, it rather displayed the value 288. A careful look at the result will tell you that the 288 is actually  $2 \times 144$  or  $144 + 144$ . What happened in memory was that since the two variables are declared as union to share the same memory location, they initially had the value in the shared memory location to be 200 (by the value of amount) and immediately changed to 144 (by the value of balance), meaning that both values are storing 144 hence summing them gives the 288.

Let us consider the following example where the output is a bit confusing as different values are printed for the different variables sharing the same memory locations.

```
#include <iostream>
#include <cstdlib>
using namespace std;
union type
{
    float f;
    int i;
    double d;
};

int main()
```



```

{
    type MyType;
    MyType.d=20.88;
    MyType.i=100;
    MyType.f=double(MyType.i);
    cout <<MyType.i <<"\t" <<MyType.f <<"\t" <<MyType.d <<endl;
    system("PAUSE");
    return 0;
}

```

In the above example, a data type, type is declared as a union. The variable myType is then defined to be of type type. The myType, by the definition of type, can be used to store values of the type int, double, float or long int as myType.i, myType.d, myType.f and myType.l respectively. If the above program is run the following output is obtained.

```

1120403456    100    20.88
Press any key to continue . . .

```

As an exercise, find out why this is so. As a hint, run the program a few times by changing the order of the statements highlighted in the above program.

## ANONYMOUS UNION

Unions can be defined to be anonymous. A union is said to be anonymous if it is not given a name. This makes it possible to refer to the elements directly in the main or other user defined function. Anonymous union has the same syntax or structure as ordinary union except that the name of the union is omitted to make it anonymous. The syntax for defining anonymous union is therefore as follows:

```

union
{
    dataType1 variable1;
    dataType2 variable2;
    :
    dataTypen variablen;
};

```

where the variables have the same meaning as above.

For example

```

struct car
{
    char make[20];
    char model[20];
    date year;
    own owner;
    char RegNo[10];
    insure insurers;
    union price

```

```

struct car
{
    char make[20];
    char model[20];
    date year;
    own owner;
    char RegNo[10];
    insure insurers;
    union

```

Figure 5.1a A named Union

Figure 5.1b An anonymous union

The Figure 5.1a and Figure 5.1b show two identical car structures. Figure 5.1b shows a structure that uses anonymous union while the Figure 5.1a does not. If we define a variable OpelCar to be of the type car, then to refer to the SellPrice and the CostPrice will be specified as follows:

For car structure using a non anonymous union

OpelCar.price.SellPrice;

OpelCar.price.SellPrice;

and for the structure using anonymous union we will have

OpelCar.SellPrice;

OpelCar.SellPrice;

### 1-5.3 Classes

Under this topic, we will be using the terms objects, classes and member functions. We shall first define these terms and then move on to how they can be defined and used in a program. In object oriented programming, everything is object. An object (sometimes called an instance of a class) is a class variable. Thus a class is simply a data type whose variables are objects. An object as a variable has member function and also has the ability to store values. Generally, a class consist of a collection of variables called **fields** and a collection of functions called **methods**. The methods operate on the fields. Methods and fields are usually refered to as members of a class.

A class is similar to a structure. The main difference between the two is that by adding member functions to a structure one obtains a class. The general format for defining a class is as follows:

```
class className
{
    Public:
}
```

```

        memberName1;
        memberName2;           public members
        :
        memberNamen;
private:
        memberName1;           }
        memberName2;          } private members
        :
        memberNamen;
};

```

where the memberName1, memberName2, etc are either member variables or member functions. Note that the same variable cannot be under both public and private. Thus a member variable or a member function can be declared to be public or private and not both. By default all members of a class are private. We now consider a typical example of a class. Let us consider a class called date. The date is made up of three parts namely the day, month and the year. Assume this date has a member function DisplayDate that is called whenever it is needed to display the values stored in the member variables day, month and year. Also, assume that date has all its members to be public.

The above example means we have three member variables namely day, month and year. We also have one member function called DisplayDate. The date class can be defined as follows:

```

Class date
{
    Public:
        int day;
        char month[10];
        int year;
        void DisplayDate();
};

```

The member function, DisplayDate, as it appears above means that it does not return a value when called and it does not also require argument when called.

To use the above class in a program, it must be placed before the main function, that is before the main function header, int main(). Note that in listing the member functions within a class, only the function prototype is required for each member function. Note also that there is no specific order for listing member variables and the member functions but we will stick to grouping member variables and member functions together as has been done in the above example. For consistency, we shall also adopt listing all member functions after the member variables. A member function is defined in the same way as any other user defined function. The main difference between

member functions and ordinary user defined function is that the function header of member function contains the class name followed by the scope resolution operator (::) and then the name of the member function. In the above class, that is the date class, the member function can be defined as follows:

```
void date::DisplayDate()
{
    function_definition;
};
```

where the function\_definition is the necessary statements that make up the function.

Generally, a member function is defined as follows:

```
dataType ClassName::memberFunctionName(argumentsList)
{
    Function_definition;
};
```

where the dataType defines the type of data to be returned, ClassName is the name of the class that has the function as a member function and memberFunctionName is the name of the member function being defined.

Once a class has been defined, it can be used to define the data type of other variables just as if you are to use the int, float, double, etc.

## **PUBLIC AND PRIVATE MEMEBERS OF A CLASS**

As you might have noticed above, member functions and member names or variables can be declared as public or private. By default all member functions and variables are private. Private member variables can be accessed only within member functions of the class itself. As such, when you declare member variables as private, it implies that the main function and other functions cannot access such variables of a class and only member function can access the variables. However, public member variables can be accessed through any object of the class. I will advise that in all your programs you declare member variables as private to prevent them being access in a program except within the definition of a member function. You should also declare your member functions as public. Note that when accessing member variables in member functions you need not have to specify the name of the class. For public member variables, if they are used outside member functions then you must specify the name of the class when accessing such variables. The following example throws more light on public and private members.

```
#include<iostream>
using namespace std;
class account
{
    private:
```

```

        char Name[40];
        int sortCode;
        char branchName[40];
        int amount,Balance;
        char TransCode;
    public:
        void UpdateAccount();
};

int main()
{
    account acc;
    cout <<"Please enter tranaction amount and type ";
    cin >>TransCode
    cin >>amount;
    return 0;
}

void account::UpdateAccount()
{
    if (TransCode == 'W')
        Balance=Balance-amount;
    else if (TransCode == 'D')
        Balance = Balance+amount;
    cout <<Name <<"\t" <<Balance;
}

```

wrong!!!, these are private member variables

all references to the member variables are valid.

In the above example, during compilation, the compiler will indicate that the variables TransCode and amount in the main function have not been declared. Note that these member variables have been declared in the account class as private hence they can only be used in member functions and that explains why in the function definition of UpdateAccount the compiler will not signal any error in reference to the member variables. The correct form should have been `cin>>acc.TransCode` and `cin>>acc.amount`.

If the member variables were to be declared as public the compiler will still display a message to indicate that the references to the two member variables are incorrect since they are undeclared. They should therefore be preceded by a dot and the class name, `acc`. This is because once they are public the main function can access them but they have to be qualified by the class names. I guess the above explanation has given you a clear insight as to how member variables and member functions can be accessed when they are declared as public or private members.

The member functions that have access to the values of private member variables are called **accessor** functions. It is very important to have accessor functions in all programs that you use

classes. This makes it possible to test for objects equality. Accessor functions also make it possible for other functions to have access to member functions of a class.

## PROTECTED MEMBERS OF A CLASS

In addition to declaring member functions and member variables as private and public, one can also declare them as protected. A member declared to be protected can only be used by members of the same class. In other words derived classes cannot access protected members. Protected members cannot also be referenced outside the class in which they are declared. I will advise that where possible, avoid the use of protected members as they are a little unsafe to use and can be very tricky. The table below summarises the differences between the public, private and protected members.

	<b>public</b>	<b>Private</b>	<b>protected</b>
Members of the same class	Yes	Yes	Yes
Members of derived classes	Yes	No	Yes
References from outside class	Yes	No	No

## VIRTUAL FUNCTIONS

Virtual functions are member functions declared with a base class and redefined with a derived class. A class containing a virtual when inherited, the derived class redefines the virtual function relative to the derived class and not the base class. Virtual functions are declared with the keyword **virtual** but when a derived class redefines a virtual function the keyword virtual is not needed. Virtual functions are called in the same way as member functions but they are called through a pointer during runtime.

By way of an example, we will consider a program with a base class and two derived class. One of the derived classes will use the virtual function definition of the base while the other will be made to overwrite the base virtual function definition.

### Program listing

```
#include <iostream>
#include <cstdlib>
using namespace std;
class baseFunction
{
public:
    int i;
```

```

inline baseFunction(int x){i=x;}
virtual void BaseVirFunc()
{
    cout <<"\nI used the virtual function of base: " <<i;
    cout <<endl;
}
};

class deriveBase1:public baseFunction
{
public:
    inline deriveBase1(int x):baseFunction(x) {}
    void BaseVirFunc()
    {
        cout <<"\nI used function of derived class 1 "
            <<"hence will increase by 10 fold: " <<10*i <<endl;
    }
};

class deriveBase2:public baseFunction
{
public:
    inline deriveBase2(int x):baseFunction(x) {}
};

```

### Program run:

I used the virtual function of base: 20

I used function of derived class 1 hence will increase by 10 fold: 200

I used the virtual function of base: 20

Press any key to continue . . .

As can be seen from the program listing and the program run, the base class, baseFunction has a virtual function BaseVirFunc whose function is to display the value passed to it. The deriveBase1 and deriveBase2 are two derived classes from the baseFunction class. Both of the derived classes can access the virtual function of the base but since derived class 1 has its own function definition to replace the virtual function of the base class, any call to the deriveBase1 will use its own function definition and not that of the base. Since the deriveBase2 has no function of its own, it will make use of the base class' virtual function and hence why it displayed the same value as the base class. Note that the redefinition of the virtual function within the derived class does not imply function overloading as the two functions are completely different. You will recall that for overloading

functions, they must differ in type and/or the number of arguments or parameters. Note also that overloaded functions are not class members but virtual functions are. One point to note is that constructors cannot be virtual functions but destructors can.

## **FRIEND FUNCTIONS**

A friend function is just like an ordinary member function but has access to both public and private members of objects of that class. To make a function a friend of a class, you must list the function prototype of the friend function in the class definition preceded by the keyword friend. The prototype can appear in the private or the public section but irrespective of where it is placed, it is considered public hence it is better to place it under the public section. The general syntax is as follows:

```
class ClassName
{
    private:
        private members;
    public:
        public members;
        friend prototypeOfFreind1;
        friend prototypeOfFreind2;
        :
}
```

Please note that a friend function is not a member function hence the dot operator or the type qualifier is not required in the definition or the header of a friend function.

## **CONSTRUCTOR AND DESTRUCTORS**

### **CONSTRUCTORS**

Generally, there are two ways of declaring and initialising variables. The following examples show the two ways:

```
int x;
```

```
x=10;
```

or

```
int x = 10;
```

The first example declares the variable x to be of integer type and then assigns it a value of 10. The second example is equivalent to the first as it also declares x as integer and assign it the value 10 simultaneously. We can do the same with member variables. Member classes have special functions call constructors. A constructor is a member function that can take parameters as needed, but cannot have a return value and cannot also use the return data type void. Normally, a



constructor is a class method that has the same name as the class itself. As such a constructor is automatically called when the object of that class is declared. Basically, a constructor is used to initialise the values of some or all member variables. The general form of a constructor (prototype) is as follows:

```
className(dataType1 MemVariable1, dataType2 MemVariable2, ...);
```

where className is the name of the constructor. As mentioned above, the name must be the same as that of the class for which the constructor is being defined. The dataType1, dataType2 are data types and the MemVariable1, MemVariable2 etc are also member variables. The definition of a constructor is given in the same way as member functions. The general form of a constructor's definition is as follows:

```
className::constructorsName(dataType1 MemVariable1, dataType2 MemVariable2, ...)
{
    constructDefinition;
}
```

where className and constructorsName are the same but represent the name of the class and the constructor respectively. The constructDefinition is the definition of the constructor. Other items have their usual meanings. Note that it is possible for a constructor not to have arguments. In this case the opening and the closing brackets should not be specified when called. A constructor with no argument is called a default constructor. It must also be noted that a constructor is called in the same way as member functions and other functions are called. It is also possible to overload a constructor just like any other user defined functions. A typical call of a constructor may be as follows.

Let the constructor's prototype be MyClass(int x, int y, double z) and the variable, OurClass be of the type MyClass then any of the following are valid calls to the constructor.

1. MyClass OurClass(100, 200, 10.2);

Or

2. MyClass OurClass;

OurClass=MyClass(100,200,10.2);

Note that the above two calls to the constructor MyCall are equivalent. The first declares OurClass to be of the type MyClass and at the same time call the constructor to initialise the member variables. The second call first declares OurClass to be of MyClass type before calling the constructor to initialise the member variables. You can make use of any of the two forms or both in your programs.

For a program example, let us write a small program making use of constructors for Forex bureaux. We will assume that the Forex Bureau has three categories of customers namely Customers (those

who have accounts with them), Visitors (nationals with no accounts with them) and others (other nationalities). The exchange rates for these three groups are different and that the commission charges are also different. We have used arbitrary figures. Making use of constructors in a program that can allow a certain amount to be converted to its equivalent in local currency we can have a typical coding such as follows:

```
#include<iostream>
using namespace std;
class forexBee
{
public:
//these three member functions are overload constructors
    forexBee(double amount, double ExRate, double ComRate);
    forexBee(double amount, double ExRate);
    forexBee(); //Is the default constructor
//this member function writes output of equivalent amount
    void Equiv();
//the following are private member variables of the class forexBee
private:
    double amount, yours;
    double ExRate,ComRate;
};
double amtToChange;
int main()
{
    double code;
    cout <<"Please enter the amount to Change ";
    cin>>amtToChange;
    cout <<"Enter 1 for Customer, 2 for Visitor and 3 for other Nationals ";
    cin >> code;
    if (code==1)
    {
        forexBee Customer(amtToChange,16000.0, 3.0);
        Customer.Equiv();
    }
    else if (code ==2)
    {
        forexBee Visitor(amtToChange,15800.0);
        Visitor.Equiv();
    }
    else
    {
        forexBee others;
        others.Equiv();
    }
}
```

```

        return 0;
    }

void forexBee::Equiv()
{
    double amt = amount*ExRate;
    yours=amt-amt*ComRate;
    cout<<"Your money £" <<amount <<" is equivalent to ";
    cout << int(yours) <<"cedis" <<endl;
}

forexBee::forexBee(double amout, double ExRat, double ComRat)
{
    amount=amout;
    ExRate=ExRat;
    ComeRate=ComRat/100;
}

forexBee::forexBee(double amout, double ExRat)
{
    amount=amout;
    ExRate=ExRat;
    ComRate=0.04;
}

forexBee::forexBee()
{
    amount=amtToChange;
    ExRate = 15400.00;
    ComRate=0.045;
}

```

In each overload of the constructor, three member variables amount, ExRate and ComRate are initialised. The values assigned to these member variables are dependent on the number of arguments passed to the constructor. The actual conversion of the amount is done by the member function Equiv().

## DESTRUCTORS

Anytime a constructor is declared there is the need to create a destructor. A destructor is a member function that cleans up after object and frees any memory that might have been allocated to a

constructor. It is called automatically when an object of the class goes out of scope. Just as a constructor has the same name as a class, a destructor should always have the same name as the class but preceded by a tilde (~). Destructors do not take arguments and have no return value. By default if you don't create a constructor or a destructor, the C++ compiler automatically creates one for you. These default constructors and destructors basically take no arguments and do nothing. It is advisable to include destructors in your program once you use a constructor to ensure that the constructors are destroyed in the end. If you are quite sure that your program may run to a successful completion then you can forget about using a destructor.

## 1-5.4 Inheritance

Inheritance is the mechanism of deriving a new class from an existing or old one. In a real world, children normally inherit traits from their parents, grand parents or ancestors. Some of the traits inherited may be the colour of the eyes, shape of the nose, way of walking, etc. Thus a child can be said to be "derived" from his/her parents. A child may have in addition to traits inherited from parents certain traits that are unique to him/her. The human or animal inheritance concept is same as in C++. Thus, in C++, one can add or alter an existing class to create the derived class. If a class B is a derived class of the class A, then A is said to be the **base class** of B, in other words class A is the parent of class B. Generally, a derived class has all the member variables and functions of its base class but also has additional features. To derive a new class from an existing one the syntax is as follows:

```
class DeriveClassName: (public|private|protected) BaseClassName
{
    member declarations
}
```

where the *DeriveClassName* is the name of the class being derived from an existing one. The *BaseClassName* is the name of the existing or the base class. The options public, private and protected have their usual meanings. The member declarations are the member names and functions.

Let us now define a typical inheritance or derived class. First, we define the parent or the base class:

```
class car
{
    Private:
        char make[20];
        char model[20];
        date year;
        own ownerName[40];
        char RegNo[10];
        insure insurers[40];
}
```

```

public:
    void getCarInfo();
    void storeCarData();
    void ProcessCarInfo();
};

```

Usually, cars in this country are either commercial or private. Assume we want to derive a class for commercial vehicles where we might want to know which GPRTU is the car registered with, the route it applies and the registration fee. This means that we need three extra member names for the commercial class. Let us call the class, Commercial and let RegFee, Station and Route be the member names representing the registration fee, GPRTU station and route respectively. Then we can define our derived class Commercial as follows:

```

Class Commercial: public car
{
    private:
        double RegFee;
        char Station[40];
        char Route[40];
    public:
        void CommeCar();
};

```

The above derived class assumes the base class, car to be public. Any object defined to be of the class Commercial will have nine member names and four member functions since the Commercial class has inherited the member names and member functions of the car class. The additional member function is the CommeCar(). Note that even though a derived class inherits all member functions and member variables, it cannot access the private members of its base class. In our example above, the member variables RegFee, Station and Route can be accessed by the member function (CommeCar) of the Commercial class however, it cannot directly access the make and model for example of the car class. The private member variables of car can only be accessed via the public member functions of the base class, that is the car class. It can be seen from the above example that an inheritance allows a great deal of reuse of C++ code.

**Question:** The following question should give you a good understanding of how to handle classes in a program. The question is to design and implement an Advance Data Type (ADT) to represent a specified date. A date consist of the day, month and year and the ADT for date will be defined in the form month day year (e.g. 12 25 2009 to mean December 25, 2009). Define or construct a class called Date. Thus, the private member variables of Date will be month, day and year. The Date should have member functions that can allow the following operations to

be carried out. Each operation should be defined as a public member function. Use the bolded name at the end of each of the following as the function name.

1. A function to accept a date from the keyboard. This date may or may not be valid but the function should accept the date entered as it is. **SetDate**.
2. A function that can check whether or not the day entered for the date in (1) is a valid one. If its invalid then this function should set the day to 1. **inspectDay**
3. A function that can check whether or not the month entered for the date in (1) is a valid month. If not, the month should be set to January, that is 1. **inspectMonth**
4. A function that can check whether or not the year entered for the date in (1) is a valid year. A year is said to be valid if it is after the year 1800. If year is invalid then set the year to 1800. **inspectYear**
5. A function to increase the year in the current set date by one year. **addYear**
6. A function capable of increasing the month in the set date by one month. **addMonth**
7. A function capable of increasing the day in the set date by one day. **addDay**
8. A function that will allow a new date and the set date to be compared. The outcome of the comparison should indicate that the set date is before, after or equal to the new date. **compare**
9. A function to print the current set date. **print**

Now design a menu shows appropriate prompts for each of the above operations excluding the function to print. When the corresponding number of an option is selected, the program should be able, where necessary, display the old date and the new date. You are to put in place proper checks so that the **compare** function compares valid dates.

To ensure that your program listing and that of ours are not too different in terms of the outputs displayed to the screen, we will advise you to take a look at our various outputs first.

### Program listing

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

class Date
{
public:
    Date(int=1 , int =1 , int = 1853); // default constructor
    int getMonth();
    int getDay();
    int getYear();
    void inspectDay();
    void inspectMonth();
    void inspectYear();
```

```

Date Date::inspectDate(Date yr);
void print();
int addDay();
int addMonth();
int addYear();
void SetDate(int mm,int dd,int yyyy);
void compare(Date date1);
private:
    int month; // 1-12
    int day;    // 1-31 based on month
    int year;   // any year
// int checkDay(int);
};

```

```

Date::Date(int m, int d, int y)
{
    month=m;
    day=d;
    year=y;
}

```

```

void Date::inspectDay()
{
    int daysPerMonth[ 13 ] = {0,31,28,31,30,31,30,31,31,30,31,30,31};
    if ( day > 0 && day <= daysPerMonth[month])
        cout <<"The day " <<day <<" in " <<month <<"/" <<day <<"/" <<year
            <<" is a valid\n";
    else if ( month == 2 && day == 29 &&
        ( year % 400 == 0 || (year % 4 == 0 && year % 100 != 0 ) ) )
        cout <<"The day " <<day <<" in " <<month <<"/" <<day <<"/" <<year <<" is a valid\n";
    else
    {
        cout <<"The day " <<day <<" in " <<month <<"/" <<day <<"/" <<year
            <<" is invalid\nThe day will now be set to 1st, i.e. 1 ";
        day=1;
    }
}

```

```

void Date::inspectMonth()
{
    if (month>0 && month <13)
        cout <<"The month " <<month <<" in " <<month <<"/" <<day <<"/" <<year
            <<" is valid\n";
    else

```

```

{
    cout <<"The month " <<month <<" in " <<month <<"/" <<day <<"/" <<year
        <<" is invalid\nThe month will now be set to January, i.e. 1";
    month=1;
}
}

```

void Date::inspectYear()

```

{
    if(year>1582)
        cout <<"The year " <<year <<" in " <<month <<"/" <<day <<"/" <<year <<" is valid\n";
    else
    {
        cout <<"The year " <<year <<" in " <<month <<"/" <<day <<"/" <<year
            <<" is invalid\nThe year will now be set to 1853.";
        year=1853;
    }
}

```

Date Date::inspectDate(Date yr)

```

{
    Date newdate;
    int vd, vm,vy;
    vd=vm=vy=0;
    if(yr.year>1582)
    {
        newdate.year=yr.year;
        vy=1;
    }
    else
    {
        cout <<"The year " <<yr.year <<" in " <<yr.month <<"/"
            <<yr.day <<"/" <<yr.year <<" is invalid. The year will now be set to 1853. ";
        newdate.year=1853;
    }
    if (yr.month>0 && yr.month <13)
    {
        newdate.month=yr.month;
        vm=1;
    }
    else
    {
        cout <<"The month " <<yr.month <<" in " <<yr.month <<"/"
            <<yr.day <<"/" <<newdate.year <<" is invalid. The month will now be set to 1. ";
    }
}

```



```

    newdate.month=1;
}

int daysPerMonth[ 13 ] = {0,31,28,31,30,31,30,31,31,30,31,30,31};
if ( yr.day > 0 && yr.day <= daysPerMonth[newdate.month])
{
    newdate.day=yr.day;
    vd=1;
}
else if (newdate.month == 2 && yr.day == 29 &&
        ( newdate.year % 400 == 0 || (newdate.year % 4 == 0 && newdate.year % 100 != 0 ) ) )
{
    newdate.day=yr.day;
    vd=1;
}
else
{
    cout <<"\n\nThe day " <<yr.day <<" in " <<newdate.month <<"/"
    <<yr.day <<"/" <<newdate.year <<" is invalid. The day will now be set to 1. ";
    newdate.day=1;
}
if(vd==0||vm==0||vy==0)
cout <<"\n\nThus, the date entered " <<yr.month <<"/" <<yr.day <<"/"
    <<yr.year <<" is incorrect hence date is now set to \n"
    <<newdate.month <<"/" <<newdate.day <<"/" <<newdate.year <<endl;
return newdate;
}

void Date::SetDate(int mm,int dd,int yyyy)
{
    month=mm;
    day=dd;
    year=yyyy;
    cout <<"The date has now been SET to ";
    print();
}

int Date::getMonth() { return month; }
int Date::getDay() { return day; }
int Date::getYear() { return year; }

void Date::print()
{ cout << month << '/' << day << '/' << year;}

```

```

int Date::addDay()
{
    int daysPerMonth[ 13 ] = { 0,31,28,31,30,31,30,31,31,30,31,30,31};
    if (month == 2 && (year % 400 == 0 || (year % 4 == 0 && year % 100 != 0 ) ) )
        daysPerMonth[2]++;
    if (day<daysPerMonth[month]-1)
        day++;
    else
    {
        day=1;
        addMonth();
    }
}

```

```

int Date::addMonth()
{
    if (month<12)
        month++;
    else
    {
        month=1;
        addYear();
    }
}

```

```

int Date::addYear()
{
    year++;
}

```

```

void Date::compare(Date yr1)
{
    int comp=0;
    if(yr1.year > year) //year 1 comes after year 2
        comp=1;
    else if (yr1.year<year) // year 1 comes before year 2
        comp=-1;
    else //the years are the same so we check the months
    {
        if(yr1.month > month) //month 1 comes after month 2

```

```

        comp=1;
    else if (yr1.month<month)//month 1 comes before month 2
        comp=-1;
    else //the year and the months are the same so we check the day
    {
        if(yr1.day>day)//day 1 is after day 2
            comp=1;
        else if (yr1.day<day) //day 1 is before day 2
            comp=-1;
    }
}
switch(comp)
{
    case 1:
    {
        print();
        cout <<" is before ";
        month=yr1.month; day=yr1.day;year=yr1.year;
        print(); break;
    }
    case -1:
    {
        print();
        cout <<" is after ";
        month=yr1.month; day=yr1.day;year=yr1.year;
        print(); break;
    }
    default:
    {
        print();
        cout <<" is equal ";
        month=yr1.month; day=yr1.day;year=yr1.year;
        print(); break;
    }
}
} //end of switch statement
cout <<endl;
}

```

```

void ShowMenu();
//list of global variables
int mm=1,dd=1,yyyy=1000,choice=9;
Date CheckDate,FirstDate;
int main()
{

```

```

do
{
    ShowMenu();
    cout <<endl;
    system("pause");
} while (choice!=9);
cout <<endl;
system("pause");
return 0;
}

void ShowMenu()
{
do
{
    system("cls");
    cout<<"\n\n\n\n";
    cout <<"\t\t\t M A I N      M E N U \n";
    cout <<setw(80)<<setfill('_') <<"_" <<setfill(' ') <<endl;
    cout <<"\t\t\t1. SET the date \n";
    cout <<"\t\t\t2. Inspect DAY in SET Date\n";
    cout <<"\t\t\t3. Inspect MONTH in SET Date\n";
    cout <<"\t\t\t4. Inspect YEAR in SET Date\n";
    cout <<"\t\t\t5. Increase DAY\n";
    cout <<"\t\t\t6. Increase MONTH\n";
    cout <<"\t\t\t7. Increase YEAR\n";
    cout <<"\t\t\t8. Compare Dates\n";
    cout <<"\t\t\t9. Exit\n";
    cout <<setw(80)<<setfill('_') <<"_" <<setfill(' ') <<endl;
    cout <<"\t\t\t\bMake a VALID choice [1-9]: ";
    cin>>choice;cout <<"\n\n";
} while(choice<1||choice>9);

    switch(choice)
    {
        case 1:
        {
            cout <<" Please enter a SET date [format: mm dd yyyy]: ";
            cin >>mm >>dd >>yyyy;cout <<"\n\n";
            CheckDate.SetDate(mm,dd,yyyy);
            break;
        }
        case 2:
        {

```

```

        CheckDate.inspectDay();
        break;
    } //end of case 2
case 3:
    {
        CheckDate.inspectMonth();
        break;
    } //end of case 3
case 4:
    {
        CheckDate.inspectYear();
        break;
    } //end of case 4
case 5:
    {
        cout << "The current date is ";
        CheckDate.print();
        CheckDate.addDay(); cout << "\n";
        cout << "Increasing day by 1 will set the new date to ";
        CheckDate.print(); break;
    } //end of case 5
case 6:
    {
        cout << "The current date is ";
        CheckDate.print();
        CheckDate.addMonth(); cout << "\n";
        cout << "Increasing the month by 1 will set the new date to ";
        CheckDate.print(); break;
    } //end of case 6
case 7:
    {
        cout << "The current date is ";
        CheckDate.print();
        CheckDate.addYear(); cout << "\n";
        cout << "Increasing the year by 1 will set the new date to ";
        CheckDate.print(); break;
    } //end of case 7
case 8:
    {
        cout << " Please enter a valid date [format: mm dd yyyy]: ";
        cin >> mm >> dd >> yyyy;
        FirstDate = Date(mm, dd, yyyy);
        FirstDate = CheckDate.inspectDate(FirstDate);
        CheckDate.compare(FirstDate);
    }

```

```
        break;
    } //end of case 8
    default: exit(1);
} //end of switch block
}
```

**Program run:** We shall first show the main menu and then go on to show the output for each of the options. We will show the various options in the order presented in the main menu.

## MAIN MENU

---

1. SET the date
  2. Inspect DAY in SET Date
  3. Inspect MONTH in SET Date
  4. Inspect YEAR in SET Date
  5. Increase DAY
  6. Increase MONTH
  7. Increase YEAR
  8. Compare Dates
  9. Exit
- 

Make a VALID choice [1-9]:

Option 1: (prompts the user to enter date to set)

Please enter a SET date [format: mm dd yyyy]: 13 12 2004

The date has now been SET to 13/12/2006

Press any key to continue . . .

Option 2: (checks the validity of the day)

The day 12 in 13/12/2006 is a valid

Press any key to continue . . .

Option 3: (checks the validity of the month)

The month 13 in 13/12/2006 is invalid

The month will now be set to January, i.e. 1

Press any key to continue . . .

Option 4: (checks the validity of the year)

The year 2006 in 1/12/2006 is valid  
Press any key to continue . . .

Option 5: (increases the current set date by one day)

The current date is 1/12/2006  
Increasing day by 1 will set the new date to 1/13/2006  
Press any key to continue . . .

Option 6: (increases the current set date by one month)

The current date is 1/13/2006  
Increasing the month by 1 will set the new date to 2/13/2006  
Press any key to continue . . .

Option 7: (increases the current set date by one year)

The current date is 2/13/2006  
Increasing the year by 1 will set the new date to 2/13/2007  
Press any key to continue . . .

Option 8: (compare a date to be entered with the current set date)

Please enter a valid date [format: mm dd yyyy]: 13 10 900  
The year 900 in 13/10/900 is invalid. The year will now be set to 1853.  
The month 13 in 13/10/1853 is invalid. The month will now be set to 1.

Thus, the date entered 13/10/900 is incorrect hence date is now set to  
1/10/1853  
2/13/2007 is after 1/10/1853  
Press any key to continue . . .



## Self Assessment 1-5

1. Write a class, *Banking* with two constructors. One of the constructors is parameterless and will be required to set *interestRate* and amount to 0.15 and 100 respectively and the other parameterised to initialise the *interestRate* and the amount using values of the parameters.
2. add a member function, *Balance* to the *Banking* class that is capable of displaying the current values in *amount* and *interestRate*,

3. Add a member function, *interest* to Banking to calculate the interest earned by a customer by multiplying the *interestRate* by the *amount* by the *period*. The period has a constant value of 0.5 as interest are calculated every six months
4. Add a member function, *update*, to automatically update a customer's account balance as soon as interest is calculated by adding the calculated interest to the value in *amount*.

## SESSION 2-5: SORTING AND SEARCHING TECHNIQUES

### 2-5.1 Sorting Techniques

One of the common data processing operations that is performed by the Computer is SORTING. Sorting is the process of arranging data in numerical or alphabetical order. The order of the arrangement may be ascending or descending order. With respect to records in a file, the ordering or sorting is based on one or more record fields usually referred to as record key(s) or key field(s). Example, if records in a file have the following fields: IdNUMBER, Name, Address and Salaries, then to arrange the records in ascending order of IdNUMBER, our key field or record key will be IdNUMBER. Record keys must be unique. Note that each time it becomes necessary to interchange any two IdNUMBERS, all corresponding record fields (IdNUMBER, Name, Address and salaries) of the two records must be interchanged as well otherwise one person's details may in the end be given to a different person.

Sorting is very important if for example the problem at hand is of 'control break' in nature. What does this mean? Let us examine or look at a big institution that is made up of a number of departments. Each department may be given a unique code or may have a unique name. Let us assume that the management of this institution wants to find out the amount spent on each department in terms of wages and salaries paid to the employees in order to make future decisions. If data on employees are captured randomly, then there is the need to sort these records otherwise for each department the Computer will have to scan the entire (personnel) file from the beginning to the end to pick all those in that department. With sequential files, the file will have to be opened and closed a number of times otherwise the program will have to contain a lot of IF (decision making) statements which will actually make the program inefficient in terms of Computer time. Sorting is used in other fields such as Mathematics (e.g. in finding the median of a set of unordered numbers), Book writers and Businesses.



There are a number of techniques that can be used in sorting data but for this course we shall look at BUBBLE, SELECTION, INSERTION, MERGE SORT, SHELL and QUICK sorts.

## **BUBBLE SORT**

This method of sorting works by comparing adjacent items. If they are found to be out of order (out of place) they are interchanged. Two items may be out of place depending on the sorting order. For example, if we want to sort the numbers 20, 40 and 5 in ascending order, then we will say 20 and 40 are in place while the 40 and 5 are out of place. If the sorting order is descending then 20 and 40 will be out of place while 40 and 5 will be in place. Comparing adjacent items and interchanging them when necessary should be repeated until we have all items in place. The steps involved in bubble sort can be as follows. Let us assume that we will want the items to be sorted in ascending or increasing magnitude.

- step 1: Initialize the number of passes to 1
- step 2: Compare the first and the second elements in the set.
- step 3: If the 1st element is greater than the second, interchange the positions of the two elements so that the first becomes the second and the second the first otherwise maintain their positions.
- step 4: Next compare the second and the third elements, third and fourth, etc. until the last but one element has been compared with the last, interchanging positions where necessary. This completes the first pass. At this stage the largest value will be filtered to the end of the list. For the next pass we will assume this last entry is not part of the input list. Note that the result will not be affected if it is included but the sorting will take a longer time to complete especially for a large list of items.
- step 5: Increase the number of passes by 1.
- step 6: Repeat steps 1 to 5 until the number of passes is one less than the number of data elements in the list.

At the end of the sorting process the data elements will be in ascending order. For sorting in descending, you will have to interchange the elements being compared when the 1st element is smaller than the 2nd, 2nd smaller than the 3rd, etc. as explained in steps 1 to 4. In sorting, a 'pass' is a term used to mean 'a round'. If we compare adjacent items from the first item through to the last item then we have completed a pass or a round of comparisons. In bubble sort, each pass is meant to place one number in its correct position. Thus, for a list of 10 numbers or data items, one will need at least 9 passes in order to have the items fully sorted. If less than 9 passes are used to fully sort a list, then that is a coincidence.

Let N and PASS be the number of keys to sort and the number of passes required respectively. To save time, the comparisons of the data elements within each PASS should be made to end at the entry at (N - PASS)th position. For example, at the end of the first pass, the largest entry in the list will have been filtered to its correct position and it will therefore be time consuming to include this entry as part of the input in the second pass. Since PASS is 2 for the second pass, it implies one entry is at its correct position hence the number of comparisons can also be reduced by 1. Looking at the example below, the largest entry 9 is filtered to its correct position at the end of the first pass. We can therefore assume that the entry 9 is not part of the input during the second pass by reducing the number of comparisons by 1. At the end of each pass we need to reduce the number of comparisons further by 1 and this will save a considerable amount of time especially when the number of keys to be sorted is very large.

Let us assume that we have been given the following keys to sort : 8, 6, 5, 4, 9, 2, 1 and we are to sort them into increasing order of magnitude. Figure 4.1 illustrates the comparisons required for each of the passes. The bolded and slightly bigger numbers in each column indicates the numbers being compared. If they are in italics then it means the comparison requires that the two numbers be interchanged.

<b>First pass</b> 8 6 6 6 6 6 6 <b>6</b> 8 5 5 5 5 5 5 <b>5</b> 8 4 4 4 4 4 4 <b>4</b> 8 8 8 8 9 9 9 <b>9</b> 9 2 2 2 2 2 2 <b>2</b> 9 1 1 1 1 1 1 <b>1</b> 9	<b>Second pass</b> <b>6</b> 5 5 5 5 5 5 5 <b>6</b> 4 4 4 4 4 4 <b>4</b> 6 6 6 6 6 8 8 <b>8</b> 8 2 2 2 2 2 2 <b>2</b> 8 1 1 1 1 1 1 <b>1</b> 8 8 9 9 9 9 9 9 9	<b>Third pass</b> 5 4 4 4 4 4 4 <b>4</b> 5 5 5 5 5 5 6 <b>6</b> 6 2 2 2 2 2 2 <b>2</b> 6 1 1 1 1 1 1 <b>1</b> 6 6 6 8 8 8 8 8 8 8 9 9 9 9 9 9 9
<b>Fourth pass</b> <b>4</b> 4 4 4 4 4 4 5 5 2 2 2 2 2 2 2 5 1 1 1 1 1 1 <b>1</b> 5 5 5 5 6 6 6 6 6 6 6 8 8 8 8 8 8 8 9 9 9 9 9 9 9	<b>Fifth pass</b> <b>4</b> 2 2 2 2 2 2 2 <b>4</b> 1 1 1 1 1 1 <b>1</b> 4 4 4 4 4 5 5 5 5 5 5 5 6 6 6 6 6 6 6 8 8 8 8 8 8 8 9 9 9 9 9 9 9	<b>Sixth pass</b> 2 1 1 1 1 1 1 <b>1</b> 2 2 2 2 2 2 4 4 4 4 4 4 4 5 5 5 5 5 5 5 6 6 6 6 6 6 6 8 8 8 8 8 8 8 9 9 9 9 9 9 9
<b>Final Result</b> 1 1 1 1 1 1 1 2 2 2 2 2 2 2 4 4 4 4 4 4 4 5 5 5 5 5 5 5 6 6 6 6 6 6 6 8 8 8 8 8 8 8		

Figure 4.1: Example of Bubble sort

Let us consider the first pass of figure 4.1. We can see that at the end of the first pass, the largest key or number (9) is filtered down to the last position and the number of comparison made in that pass is 6 (that is the number of items in the list minus 1). Now, having filtered the largest entry to the last position, we assume that the largest entry is no longer part of the input when considering the second pass. This does not mean that the largest entry should be deleted (physically) from the list at the end of the first pass but rather we should make sure that we reduce the number of comparisons done in the first pass by one in the second pass just to save unnecessary comparisons. Similarly, in the third pass, we would exclude the last two entries as part of the input, and so forth.

**Question:** Draw a flowchart for the binary sort method

## SELECTION SORT

As the name suggests, selection sort involves selecting either the largest or the smallest entry in a list and interchanging it with the last or the first position depending on the programmer. The next largest or smallest entry is selected and interchange with the element at the second or the last but one position, etc. The following steps therefore explain further the processes involved in this type of sorting. Let us assume that we want to sort in ascending order.

- (1) Let  $pass = 1$
- (2) For the current  $pass$ , start from the element at position  $pass$  and scan through to the end of the entire data elements for the minimum entry.
- (3) Interchange the positions of the minimum element and the element at the  $pass$  position such that the minimum entry selected in step 2 is moved to location  $pass$  and the element at location  $pass$  to the location where the minimum entry was found.
- (4) Increase the value of  $pass$  by 1. If the current value of  $pass$  is less than the total number of items in the original list then repeat step 2 otherwise stop as the list will be sorted by now.

Let us use the list 8, 6, 5, 7, 9, 1, 2 to see how the selection sort actually works. Where two number are bolded in a list, it implies the two were the numbers that were interchanged at the end of the pass in question.

- (i) For  $pass=1$ , that is the first pass and starting with the first element, the number 9 will be selected as the largest entry. Five is therefore the position at which the 9 is found. We will therefore have to interchange the elements at position 1 and 5, that is the numbers 8 and 9. This means after the first pass, the list will now be as follows:  
           9, 6, 5, 7, **8**, 1, 2
- (ii) For  $pass=2$ , we start from the second element and look for the largest entry. This will be the number 8 at position 5 hence we will interchange the elements at positions 2 and 5, meaning we are interchanging the numbers 6 and 8. Our new list will now be  
           9, **8**, 5, 7, **6**, 1, 2
- (iii) For  $pass = 3$ , we start from the third element and look for the largest entry. We will get the number 7 at position 4, as such we will interchange the elements at positions 3 and 4, that is the elements 5 and 7. This will give a new list as follows:  
           9, 8, **7**, **5**, 6, 1, 2
- (iv) At the end of the fourth pass, we will have our list as 9, 8, 7, **6**, **5**, 1, 2
- (v) For  $pass = 5$ , we will notice if we start from the fifth item and look for the largest item we will end up with the fifth item being the largest. Here, there is therefore no need to interchange any item. In actual fact, we interchange 5 with itself. Our list will now be 9, 8, 7, 6, **5**, 1, 2
- (vi) Since the elements are 7 in number, we will need six passes. At the sixth pass 2 will be the largest entry hence we have to interchange 1 and 2. This will give the final list as 9, 8, 7, 6, 5, **2**, **1**

**Question:** Draw two flowcharts for the selection sort method, one for sorting in ascending order and the other for descending order.

## INSERTION SORT

This method of sorting is not very much different from the selection sort described above. The steps involved in insertion sort for sorting in ascending order are as follows:

- (i) Set a variable  $pass$  to 1. Assume the first entry in the list to be at its correct position when the entire list is sorted. In other words, we are assuming that to start with we consider only the

first item in the list. We shall refer to this part of the list as sorted segment, hence we are assuming that the first entry is a sorted segment.

- (ii) Increase the value of *pass* by 1 so that it points to the element at position *pass*.
- (iii) Compare the *pass*th entry with the entries in the sorted segment starting with element at the first location to that at the (*pass*-1)th location and insert the *pass*th element in its proper position of the sorted segment. For example, with *I*=2, the 2nd item is compared with the first item and will be placed either before or after the first item depending on the sorting order.
- (iii) Repeat steps (ii) and (iv) until the counter *pass* has a value equal to *N*, where *N* is the number of entries to be sorted.

Let us consider a list of numbers to explain the above algorithm. We will use the list 15, 6, 25, 14, 7 and 10 as the entries required to be sorted. Since the number of entries is 6, we will set *N* to 6. In the following steps, any entry or entries underlined will constitute the sorted segment.

- (i) For *pass* = 1, we assume 15 to be the sorted segment of the entries, so for the start we have  
15, 6, 25, 14, 7, 10.
- (ii) We will increase *pass* by 1 to 2 to point to the second entry, 6 in the list.
- (iii) Since there is only one entry in the sorted segment, we compare the element 6 with the element 15. Six (6) is less than 15, so 6 is placed before 15 by moving 15 to the second location and inserting 6 into the first location. Our new list is now:  
6, 15, 25, 14, 7, 10 with 6 and 15 as the sorted segment.
- (iv) We increase the counter *pass* by 1 to 3 so as to point to the third element, 25.
- (v) We now compare 25 with entries in the sorted segment (6 and 15). Since 25 is greater than these two entries, we will place 25 after the entry 15. Hence our list will be  
6, 15, 25, 14, 7, 10
- (vi) Increase *pass* by 1 to point to the fourth entry 14.
- (vii) Since 14 is greater than 6 but less than 15, we have to insert it between 6 and 15. This is done by moving the entries 15 and 25 to the right by one location each. The list will now be  
6, 14, 15, 25, 7, 10
- (viii) Increase *pass* to 5.
- (ix) The fifth entry is to be inserted between 6 and 14 hence we will have  
6, 7, 14, 15, 25, 10 as the new list
- (x) Increase *pass* by 1 to point to the sixth entry, 10.
- (xi) Since the sixth entry is greater than 7 but less than 14, we will insert it between the entries 6 and 14. This modifies our entries as follows: 6, 7, 10, 14, 15, 25
- (xii) Increasing *pass* by 1 will result in *pass* being greater than *N* (*N*=6) meaning that we have dealt with all the entries in the list hence we terminate program execution. Finally the selection sort will return 6, 7, 10, 14, 15, 25 as the sorted list.

**Question:** Draw a detailed flowchart for the selection sort method.

## SHELL SORT

This method of sorting sorts elements by diminishing increments  $h_t > h_{t-1} > h_{t-2} > \dots > h_2 > h_1$  where  $h_1 = 1$  and  $t$  is the number of diminishing increments required. The diminishing increments should be integers. There is no limit to the number of the diminishing increments that can be used but it is a must that the last increment  $h_1$  be 1 else the list may not be fully sorted. The increments can be chosen at random but experience has shown that the shell sort algorithm works well when the increments are of the form  $2^x 3^y$  such that the largest increment is less than the number of entries to be sorted, and  $x$  and  $y$  are integers. For example, if you have a list of 15 entries to be sorted, then the following increments will be preferred:

- (i)  $x = 1$  and  $y = 1 \implies 2^x 3^y = 2 \times 3 = 6$
- (ii)  $x = 2$  and  $y = 0 \implies 2^x 3^y = 4 \times 1 = 4$
- (iii)  $x = 0$  and  $y = 1 \implies 2^x 3^y = 1 \times 3 = 3$
- (iv)  $x = 1$  and  $y = 0 \implies 2^x 3^y = 2 \times 1 = 2$
- (v) the increment 1,

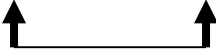
Apart from the combination of values used for  $x$  and  $y$  in the above, any other combination of values will give a result that is greater than 6,  $2^x 3^y$  hence the full incremental set will be (6, 4, 3, 2 and 1). It is not a must that all increments that can be derived from  $2^x 3^y$  and are less than  $N$  be used. Some may be ignored and the program will still work. In our example above, we can decide to ignore increment 4 and either 2 or 3 since the difference between them is very small. Note that although the difference between 2 and 1 is just like that between 3 and 2, 1 cannot be ignored since it is a must that 1 be included in the increments.

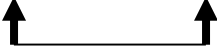
Having chosen the required increments, the program works by sorting the  $h_t$ th entries, followed by  $h_{t-1}$ th entries and finally  $h=1$  entries. When  $h=1$ , it implies adjacent items are to be compared.


To fully understand how this method works, let us consider a list of eight entries with the diminishing increments 4, 2 and 1. Let the list consists of the entries 80, 91, 16, 78, 25, 20, 1, 93. We will sort in ascending order. Thus we have  $h_3 = 4$ ,  $h_2 = 2$  and  $h_1 = 1$ .


- (i) With  $h_3$ , we sort every fourth entry in the list, thus for our list we sort as follows:  
We start by comparing the first with 5th entry. This requires swapping/interchanging. The 2nd entry is compared with sixth entry. This requires swapping. The 3rd entry compared with 7th

entry also requires swapping. The 4th entry compared with 8th entry. Requires no swapping. Pictorially, this step can be represented as follows:

That is:      80 91 16 78 25 20 1 93 (compares 80 and 25, requires swapping)  


To give      25 91 16 78 80 20 1 93 (compares 91 and 20, requires swapping)  


to give      25 20 16 78 25 91 1 93 (compare 16 and 1, requires swapping)  


to give      25 20 1 78 80 91 16 93 (compare 78 and 93, no swapping required)  


finally      25 20 1 78 80 91 16 93      at the end of the comparisons

- (ii) We repeat the above processes this time with  $h=2$  to sort every 2nd entry in the partially sorted list, that is (1<sup>st</sup> and 3<sup>rd</sup>, 2<sup>nd</sup> and 4<sup>th</sup>, 3<sup>rd</sup> and 5<sup>th</sup>, 4<sup>th</sup> and 6<sup>th</sup>, 5<sup>th</sup> and 7<sup>th</sup>, 6<sup>th</sup> and 8<sup>th</sup>). Before we start with  $h_2$ , the partially sorted list is 25 20 1 78 80 91 16 93

After sorting with  $h=2$ , the newly partially sorted list will be as follows:

1 20 16 78 25 91 80 93

- (iii) As can be seen, the list is not fully sorted hence we apply  $h = 1$ . When  $h=1$ , it means each element should be compared with the element immediately following and interchange them if they are out of order. From the result in step (ii), if we apply the  $h=1$ , we will end up with the following as our final list.

1 16 20 25 78 80 91 93.

### Question

Draw a detailed flowchart for the shell sort method.

## QUICK SORT

As the name implies, this method of sorting is very fast especially when the number of keys involved to be sorted is quite large. The processes involved in this method can be broken down as follows:

- (i) Let  $q$  and  $p$  points respectively to the first and the last entries in the list to be sorted.
- (ii) Store the entry pointed to by  $q$  (that is the first entry in the segment to be sorted) in a certain variable,  $X$  say.
- (iii) Start scanning the entries in the list from the entry pointed to by  $p$  upwards (or to the left) until an entry is found to be less than  $X$ .
- (iv) Placed the element obtained in step (iii) at the location currently pointed to by  $q$  and let  $p$  points to the location just above the location at which the entry in step (iii) was found.
- (v) Start scanning the entries in the list again this time from the entry pointed to by  $q$  and then downwards (or to the right) until an entry is found to be greater or equal to that stored in  $X$ .
- (vi) Place the entry selected in step (v) at the position currently pointed to by  $p$ , and let  $q$  points to the location following that at which the entry in step (iv) was found.
- (vii) Repeat steps (iii) through to (vi) until  $p$  and  $q$  point to the same location
- (viii) Store  $X$  at the location pointed to by both  $p$  and  $q$

In summary, the above steps push all entries less than the first entry to be moved above the first entry and those greater than or equal to the first entry to be moved below the first entry. This therefore results in the first entry being placed at its correct position with the whole list being divided into two parts (those that are less than  $X$  and those that are greater than or equal to  $X$ ). Each part is then assumed to be the new input and the above steps are applied to it. This is repeated until each part contains only one entry. When that is done, each entry in the original list will be at its correct position and hence a sorted list is obtained. Let us use the following keys to understand the steps given above: 16, 4, 25, 30, 14, 2, 18 and 17.

We start with  $X = 16$

16 ← $q$	2 ← $q^*$	2	2	2	2	2
4	4	4	4	4	4	4
25	25 $s$	25 ← $q$	14 ← $q^*$	14	14	14
30	30	30	30	30 ← $q$	30 $p=q$	<b>16</b>
14	14	14 $s$	14 ← $p$	30 ← $p^* s$	30	30
2 $s$	2 ← $p$	25 ← $p^*$	25	25	25	25
18	18	18	18	18	18	18
17 ← $p^*$	17	17	17	17	17	17



An asterisk after the pointer  $p$  or  $q$  indicates the pointer to be moved for the search of the next entry and an  $s$  also indicates the entry being searched for. Where there is no  $s$  indicates that no entry satisfies the condition for search and hence no entry is found. When  $p$  and  $q$  are pointing to the same location, we have to place the entry that was stored in  $X$  at this location and the result is as

2   4   14   16   30   25   18   17.

From the above our two parts will have the following entries: part 1: 2, 4 and 15 and part 2: 30, 25, 18 and 17, with the entry 16 being at its correct position. We then repeat the above processes for each of the parts and each time we repeat the processes, a part is further broken down into two parts with one entry being placed at its correct position. When each part has only one entry, the entire parts will be found to be sorted when joined together.

## MERGE SORT

This method of sorting uses two sorted lists as input and sort them into one merged list. The processes involved in this method of sorting are as follows:

Let  $X$  and  $Y$  be the two sorted lists to be merged with sizes of  $N$  and  $M$  respectively. Let  $C$  be the array to store the merged list items. This means the array  $C$  will have a size of  $(N+M)$ . We will sort in ascending order.

- step 1:        Set two counters or pointers, say  $I$  and  $J$  to point to the first entries in arrays  $X$  and  $Y$  respectively. Initialize a third pointer say  $K$  to 1 to point to the first location in  $C$ .
- step 2:        Compare the two entries pointed to by  $I$  and  $J$  and move the smaller of the two entries to the location in  $C$  pointed to by  $K$ . If it happens that the two entries are equal move any one of them to array  $C$ .
- step 3:        Increase the pointer ( $I$  or  $J$ ) that points to the entry that was moved to array  $C$  in step 2 by 1 so that it points to the next entry in the list to which it has been assigned to. Increase also the pointer  $K$  by 1 to point to the next available free location in  $C$ .
- step 4:        Again, compare the two entries pointed to by  $I$  and  $J$  and move the smaller of the two to array  $C$ .
- step 5:        Repeat steps 3 and 4 above until either  $I$  is greater than  $N$  or  $J$  is greater than  $M$ .
- step 6:        If  $I$  is greater than  $N$ , then move the entries in  $Y$  starting from current position of  $J$  to the end of the array  $C$  else (that is,  $J$  greater than  $M$ ) move entries in array  $X$  starting from the current position of  $I$  to the end of the array  $C$ .

Note that when the value of  $I$  is greater than that of  $N$  it implies that all entries in  $X$  have been moved to array  $C$  hence all that is left for us to do is to move the remaining entries in  $Y$  to  $C$ . If on the other hand the value of  $J$  is greater than that of  $M$  then we will move the remaining entries in  $X$  to the end of the array  $C$ .

Let us use the following as an example to illustrate the above steps:

array X: 10, 20, 30, 40, 50 and

array Y: 15, 25, 30, 60, 70, 80, 90, 100

Here  $N=5$  and  $M=8$  since the list  $X$  and  $Y$  contain 5 and 8 elements respectively.

- step 1.  $I = J = K = 1$
- step 2. Compare  $X(I)$  and  $Y(J)$
- step 3. Since  $X(I)$  is smaller than  $Y(J)$  we will move  $X(I)$  to  $C(K)$ , hence at this point array  $C$  will contain the entry 10.
- step 4. We increase  $I$  and  $K$  by one each. If  $X(I)$  was to be greater than  $Y(J)$  in step 3, we would have increase  $J$  and  $K$  instead of  $I$  and  $K$ .
- step 5. If  $I \leq N$  and  $J \leq M$  then repeat step 2
- step 6. If  $I > N$  and  $J \leq M$  then copy remaining elements in  $Y$  to  $C$  starting from the  $J$ th position else copy remaining elements in  $X$  to  $C$  starting at the  $I$ th position.
- Step 6. Print array  $C$
- Step 7. Stop

**Question:** Draw a detailed flowchart for merging any two sorted input lists into one.

### Question 1

A long time ago, the Romans used letters as symbols for numbers as follows:

Roman Numeral Symbol	I	V	X	L	C	D	M
Equivalent Number	1	5	10	50	100	500	1000

If one has to convert a number in Roman Numeral to its equivalent in base 10, the following rules are used:

- 1. When a letter is repeated, their equivalents are added  
e.g.  $CC=100+100=200$        $XXX=10+10+10=30$

2. When a letter of a smaller value is placed to the immediate right of another letter, their equivalents are added  
e.g.  $CX = 100 + 10 = 110$                        $CCXV = 100 + 100 + 10 + 5 = 215$
  
3. When a letter of a smaller value is placed to the immediate left of another letter, the smaller value is subtracted from the larger value  
e.g.  $XC = 100 - 10 = 90$                        $CMX = (1000 - 100) + 10 = 910$
  
4. A single letter is used no more than three times in a row. In such a case, use one letter of the smaller value to the immediate left on the next letter of larger value.  
e.g.               $XXXX$  should be written as  $XL$   
                     $CCCCIII$  should be written as  $CDIV$

Use the above rules and the table above, write a detailed algorithm that accept as input a Roman numeral and output its equivalent in the decimal system.

### Question 2

Refer to question 1 to understand the conversion of Roman Numeral to decimal. Write a detailed algorithm that accepts as input an integer number in base 10 and output its equivalent in Roman Numeral.

### Question 3

Considering your solutions to questions 1 and 2, write a program that accepts as input two Roman Numerals and an arithmetic operator (+, -, \* or /) and output the result obtained by applying the arithmetic operator to the two Roman Numerals in Roman Numeral. For example, if the inputs are **XXIX**, **CXX** and **\***, your output should be **XXIX \* CXX = MMMCDLXXX** (that is  $120 * 29 = 3480$ ).

## 2-5.2 Searching Techniques

Searching is a method of locating a specific item in a larger collection of items. Most of the time when a program is written it does not only store and manipulate items stored in an array but in addition will have to locate specific items within the array. In this session we shall consider two searching algorithms namely the Linear and the Binary searches.

### The LINEAR SEARCH

Linear search is a method of locating an item in a larger collection of items by beginning the search with the first item of the collections and then through to the last item. During the search, the item being search for may either be found or may not be on the list. If the items are already sorted then the search can be stopped as soon as the item is found or an

item that is greater or smaller than the item being search for is encountered depending on the order in which the item were arranged. The program below illustrates a linear search. It first allows a set of numbers to be stored in an array and later allows for a number to be searched for.

### Program listing

```
#include<iostream>
using namespace std;
int main()
{
    int x[100],searchNum,n;
    char again;
    cout<<"Please enter total number of items to store ";
    cin>>n;
    cout<<"\nPlease Enter\n";
    //get the numbers into an array X
    for(int i=0;i<n;i++)
    {
        cout<<"X["<<i+1<<"]: ";
        cin>>x[i];
    }
    do //loop to search for a number of items until the user does not press 'y' or 'Y'
    {
        cout<<"Please enter the item to search for ";
        cin>>searchNum;
        int position=-1,i;
        for(i=0;i<n;i++)
            if(x[i]==searchNum)
            { position=i;
              break;
            }
        if(position<0)
            cout <<"The item "<<searchNum<<" is not on the list";
        else
            cout <<"The item "<<searchNum<<" is on the list at position " <<position+1;
        cout<<"\nDo you want to search for another item [Y/N]? ";
        cin>>(again);
    } while(toupper(again)=='Y');
    system("pause");
    return 0;
}
```

### Program run:

Please enter total number of items to store 10

Please Enter

X[1]: 21

X[2]: 23

X[3]: 26  
X[4]: 32  
X[5]: 35  
X[6]: 39  
X[7]: 48  
X[8]: 56  
X[9]: 66  
X[10]: 80  
Please enter the item to search for 50  
The item 50 is not on the list  
Do you want to search for another item [Y/N]? y  
Please enter the item to search for 48  
The item 48 is on the list at position 7  
Do you want to search for another item [Y/N]? n  
Press any key to continue . . .

## The Binary Search

The Binary Search is much more efficient than the Linear Search except that for the Binary search the collection of items to be searched must be sorted while for a Linear Search the items may or may not necessarily be sorted. For a Binary Search, anytime a comparison is made and the item being search for is not the desired item, the Binary Search eliminates half of the remaining items to be searched. For example, if we have 1,000,000 items to search through for a particular item, if the Binary Search fails to find the desired item at the first attempt or comparison, the number of items that will remain to be search in the second item is 500,000 and at the third attempt will be 250,000, etc., a This process will continue until the search is able to find the desired item or established that the desired item is not on the list. In the worst case of 1,000,000 items, the Binary Search will make no more than 20 comparisons in locating the desired item while the Linear Search will make approximately 1,000,000 comparisons. A rough method of finding the maximum number of comparisons that may be needed by the Binary Search is to find the smallest powers of 2 that is greater than or equal to the number of items. For example with the 1,000,000 items, the smallest powers of 2 greater than 1,000,000 is  $2^{20}$  (1,048,576). The following summerises how the Binary search works assuming that N, L and R are the number of items on the list, the leftmost and the rightmost items respectively. Let X be the array containing the collection of items.

Step 1: Let L and R point to the first and last items on the list, that is  $L=0$  and  $R=N-1$

Step 2: Calculate M, the mid point of L and R, that is  $M=0.5(L+R)$

Step 3: Compare search item, S with item at position M, that if  $S=X[M]$ . If  $S=X[M]$  then stop as the search item is found at position M.

Step 4: If  $S > X[M]$  then  $L=M+1$  else  $R=M-1$

Step 5: if  $L=R=M$  and S is not equal to  $X[M]$  then stop and display a message that the search item is not on the list else repeat step 2.

Converting the above to a C++ program gives the following program listing:

```
#include <iostream>
using namespace std;
int getInput();
char SearchItem(int S);
int List[1000],n,position;
int main()
{
    int search;
    search=getInput();
    if (SearchItem(search)=='T')
        cout<<"The item "<<search<<" is on the list at position "<<position+1<<endl;
    else
        cout<<"The item "<<search<<" is NOT on the list\n";
    system("pause");
    return 0;
}

int getInput()
{
    int s;
    system("cls");
    cout<<"Please, enter the total number of items ";
    cin>>n;
    cout<<"Please enter \n";
    for(int i=0;i<n;i++)
    {
        cout<<"List["<<i+1<<"]: ";
        cin>>List[i];
    }
    cout<<"Please, what item do you want to search for ";
    cin>>s;
    return s;
}

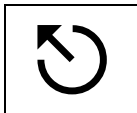
char SearchItem(int s)
{
    position=-1;int i=-1,left=0,right=n-1,mid;
    char notFound='F';
    do
    {
        mid=(int)(0.5*(left+right));
        position=mid;
        if(s==List[mid])
            return 'T';
        if(s>List[mid])
            left=mid+1;
        else
            right=mid-1;
    }while(left!=right && left!=mid && s!=List[mid]);
}
```

```
return notFound;  
}
```



## Self Assessment 2-5

1.
  - a) What is sorting?
  - b) Can you write brief notes on the different sorting techniques explained in this chapter, using typical examples to show how exactly the method works?
  - c) Given the set of numbers 21, 10, 5, 22, 21, 90, 10, 5, 8, 17 and 34 to sort into ascending order of magnitude, can you work out the number of passes and comparisons that are required by each of the different sorting methods (bubble, selection and insertion sorts) to completely sort the elements in the required order?
  - d. Rewrite the linear search program above using a do...while instead of the for loop.



## Unit Summary

1. We have learnt in this Unit that apart from the pre-defined data types one can also create and use his own data types. Structures make it possible for us to group a number of fields as a record. Classes are not too different from structures except that classes in addition to member variables have member functions. We have also learnt that we can write classes such that some of the classes can be derived from existing classes. To write a structure the reserved word *struct* is used and for classes we use *class*. Since structures and classes are referred to in a program they must be given names by which references can be made to them.
2. We have also learnt about a number of sorting techniques. Sorting a list basically involves arranging the members of the list into either ascending or descending order. Sorting can be done on numbers and string of characters. Unless specifically asked, any of the sorting techniques can be used even those some are faster than the other.