Chapter Six

6. Structures and File Management

6.1.Record: - Structure

Thus far you have worked with variable's whose data types are very simple: they are a numbers of either integer or floating-point format with a specific range. These types of variables, who only have a single value to their name, are known as basic variables or primitives. Everything in computers is built on numbers, but not necessarily singular types. Sometimes it's advantageous to group common variables into a single collection. For example a date would require a day, month, and year. From what we have discussed currently explained, you could create three separate variables for each, like so:

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

This isn't so bad, but what happens if you want to store two dates and not one? You'd have to create three more variables and give them unique names:

```
int day1, month1, year1;
int day2, month1, year2;
```

This begins to become a hassle. Not only do you have to create many variables, but you have to keep giving them unique names. C++ provides a way to collect similar variables into a single *structure*.

An array is a data structure which holds multiple numbers of objects having the same basic property (data type) in a contiguous memory slots. Programmer can also reserve contiguous memory space for aggregates of elements of arbitrary data types each. A data type which is created to reserve such type of memory space is called user defined data type.

User defined data types can be equally used like predefined data types to declare variable identifiers. For example we can define simple, array, or pointer variable identifier from this user defined data type.

6.1.1. Structure

The term *structure* in C++ means both a user-defined type which is a grouping of variables as well as meaning a variable based on a user-defined structure type. For the

purpose of distinction we will refer to the user-defined type side as *structure definition* and the variable side as *structure variable*.

A structure definition is a user-defined variable type which is a grouping of one or more variables. The type itself has a name, just like '`int`', '`double`', or '`char`' but it is defined by the user and follows the normal rules of identifiers. Once the type has been defined through the C++ '`struc`t' keyword, you can create variables from it just like you would any other type.

Since a structure definition is a grouping of several types: it is a group of one or more variables. These are known as elements or member variables as they are members of the structure definition they are part of. Following through with our hinted example, a structure definition could be a 'date' which might be made up of three 'int' member variables: 'day', 'month', and 'year'.

Before creating a structure variable you must create a structure definition. This is a blue print for the compiler that is used each time you create a structure variable of this type. The structure definition is a listing of all member variables with their types and names.

When you create a structure variable based on a structure definition, all of the member variables names are retained. The only name you have to give is that of the new structure variable. The element names within that variable will be the same as in the structure type. If you create two structure variables from 'date', both will have all three member variables with the same name in both: 'day', 'month', and 'year'.

Member variables are distinguished by the structure variable they are part of. You wouldn't simply be able to use 'day' by itself; instead you'd have to refer to both the structure variable's name (that you gave when you created it) as well as 'day'.

6.1.2. struct Specification: Defining Structures

Defining a structure is giving the compiler a blue print for creating your type. When you create a variable based on the structure definition, all of the member variables are created automatically and grouped under the name you gave. Note that when you create a variable of any kind, you must give it a unique name that is different than its type. Below is the syntax for a structure definition:

```
struct structname
{
      datatype1 variable1;
      datatype2  variable2;

 };
```

Discussion: Writing a structure definition begins with the word 'struct' followed by the type-to-be, and ended with a structure block that is ultimately terminated with a semicolon. **Do not forget this trailing semi-colon when defining a structure**!

The name of a structure definition is known as the structure *tag*. This will be the name of the type that you create, like 'int' or 'float'. It is the type that you will specify when creating a structure variable. This structure block is remarkably similar to a statement block since it starts and ends with curly braces. But don't forget that it ultimately ends with a semi-colon. Within the structure block you declare all the member variables you want associated with that type. Declare them as you would normal variables, but do not try to initialize them. This is simply a data blue print, it is not logic or instructions and the compiler does not execute it.

The *data members* (synonym for member variables) of a structure won't actually be created until a variable b*ased* on the structure is created. Technically an '`int`' is just this as well. It's a description of a storage unit. That storage unit isn't reserved until you create a variable with it.

Example defining a student struct,

```
struct student
 {
   int id;
   char name[15];
 };
```

Example: The follow defines a structure called 'date' which contains three 'int' member variables: 'day', 'month', and 'year':

```
struct date {
   int day;
   int month;
   int year;
};
struct date
{
    int day, month, year;
```

```
};
```
Note: You *cannot* initialize member variables in a structure definition. The following is *wrong* and will *not* compile:
```
struct date{
    int day = 24, month = 10, year = 2001;
};
```

A structure definition has the same type of scoping as a variable. If you define a structure in a function, you will only be able to use it there. If you define it in a nested statement block, you will only be able to use it inside there and any statement blocks nested within it. But the most common place is defining it globally, as in outside of any functions or blocks. Typically, you'll want to be able to create variables of the defined structure anywhere in your program, so the definition will go at the top. The following is an empty program that defines a 'date' structure globally:

```
#include <iostream.h>

    struct date{
        int day, month, year;
    };
    int main(){
        return 0;
    }
```

6.1.3. Declaring and using sturct data types

Once you have defined a structure you can create a variable from it just as you would any other variable.

```
student std1;
date birthday;
```

The above declaration statements would create a variable called '`birthday`' whose type is the structure '`date`'. The variable contains three parts: '`day`', '`month`', and '`year`'. What this actually does is set aside a whole block of memory that can contain all of the member variables. Each member variable then occupies a chunk of it for their individual storage units. The member variables are not actually created one at a time.

Storage for member variables exist at some *offset* from the beginning of the glob of memory reserved by the entire structure. For example, in our 'date' structure variable the first member variable is 'day' so it exists at offset 0. The next member variable, 'month' in this case, will exist at the next available offset. If 'day' uses 4 bytes (32 bits), then 'month' will be at offset 4:

```
int  i;
student std1;
```

The above statements are similar in nature because both declare a variable of a given type. The former is a built in type, which is integer while the later declares a variable type of user-defined type. std1 will have the characteristics of representing id and name of a student.

6.1.3.1. Initializing Structure Variables

You cannot initialize member variables in the structure definition. This is because that definition is only a map, or plan, of what a variable based on this type will be made of. You can, however, initialize the member variables of a structure *variable*. That is, when you create a variable based on your structure definition you can pass each member variable an initializer.

To initialize a structure variable's members, you follow the original declaration with the assignment operator (=). Next you define an initialization block which is a list of initializers separated by commas and enclosed in curly braces. Lastly, you end it with a semi-colon. These values are assigned to member variables in the order that they occur. Let's look at an example:

```
date nco_birthday = { 19, 8, 1979 };
student std1={"Ababe", "Scr/2222/22"};
```

This creates a variable called '`nco_birthday`' and initializes it to a list of values. The values are assigned to the member variables in the order they are declared in the structure definition. Remember what is mentioned about each member variable having an offset. The same order in which each member is given an offset is the order in which each is assigned a value in an initialization. So the first initializer is used to initialize the first member variable in the structure, next is the second, and so on and so forth. This order of initialization continues until the values run out.

If you try to assign more values than are member variables, you will get a compiler error. However, you can assign *fewer* values than there are member variables. If there are no more values to be assigned, the assignment will simply end. For example, if we had omitted the last value, '1979', then no value would be assigned to '`year`'.

It is possible to use any expression that you normally would. But remember that the expression must result in a value. Here is an example of initialization with things other than literals:

```
int myday = 19;
int mymonth = 5;
date nco_birthday = { myday, mymonth + 3, 2001 - 22 };
```

Although you can assign a value to a variable in the same way you initialize it, the same is not true with structures. So while this works:

```
int x;
x = 0;
```

This doesn't:

```
date nco_birthday;
nco_birthday = { 19, 8, 1979 };
```

Assigning values to multiple members of a structure variable is only possible when that variable is first created. Once a structure variable has been declared, you must access each member individually.

6.1.3.2. Accessing members of a structure variable

There's not much you can do with the structure itself; much of your time with structure variables will be spent using its members. You can use a member variable in any place you'd use a normal variable, but you must specify it by the structure variable's name *as well as* the member variable's name using the member operator.

To specify that you want a member of a specific structure variable, you use the structure member operator which is the period (also known as a "dot"). Simply use the structure's name, follow with the period, and end with the member:

```
structure.member
```

Example: to reading and displaying values to and from structure s1.

```
cin>>s1.id; //storing to id item of s1
cin>>s1.name; //storing a name to s1
cout<<s1.id;  //displaying the content of id of s1.
cout<<s1.name; //displaying name
```

Example:-a program that creates student struct and uses it to store student information.

```
#include<iostream.h>
#include<conio.h>
struct student
   {
```

```
  int id;
  char name[15];
  };

void main()
  {
   //creating three student variables
   student s1,s2;
   cout<<"\n Enter Student Id";
   cin>>s1.id;
   cout<<"\nEnter Name";
   cin>>s1.name;
   cout<<"\n Enter Student Id";
   cin>>s2.id;
   cout<<"\nEnter Name";
   cin>>s2.name;

   cout<<"\nStudents Information";
   cout<<"\n Student id\t Student Name";
   cout<<endl<<s1.id<<"\t"<<s1.name;
   cout<<endl<<s2.id<<"\t"<<s2.name;
   getch();
  }
```

The above program shows you how to capture data in student variables.

Example 2: This program demonstrates using member variables for user input, output, and mathematical operations.

```
#include <iostream.h>
struct date
{
    int day, month, year;
};
int main()
{
    date birth;
    cout << "Enter your birth date!" << endl;
    cout << "Year: ";
    cin >> birth.year;
    cout << "Month: ";
    cin >> birth.month;
    cout << "Day: ";
    cin >> birth.day;
    cout << "You entered " << birth.month << "/"<< birth.day
      << "/" << birth.year << endl;
    cout << "You were born in the "<< ((birth.year / 100) + 1)
      << "th Century!"<< endl;
    return 0;
}
```

6.1.3.3. Variables with Definition

The syntax of '`struct`' also allows you to create variables based on a structure definition without using two separate statements:

```
struct tag
{
    member(s);
} variable;
```

The structure variables created in this way will have the same scope as their structure definition. This is a nice thing to use when you want to group some variables in one place in your program without it affecting other things. You can create a structure definition as well as variables from it in that one local place and not have to ever use it again.

Example: A 'point' variable right after the 'pointtag' structure is defined:

```
struct pointtag{
    int x, y;
} point;
```

In this, 'point' is a variable just as if we had declared it separately. In fact, this statement is identical to:

```
struct pointtag{
    int x, y;
};
pointtag point;
```

Rather than defining a structure under a name and then creating variables which refer to the named definition, the definition becomes part of the variable declaration. It is even possible to omit the structure tag which may make more sense in this situation:

```
struct{
    int x, y;
} point;
```

The above creates a structure variable called 'point' which has two member variables. Because the structure definition is not named, it cannot be used elsewhere to create variables of the same structure type. However, like in any variable declaration, you can create multiple variables of the same type by separating them with commas:

```
struct{
    int x, y;
} point1, point2;
```

Now we have two structure variables of the same nameless structure type. Even more fun we can initialize these structure variables as we normally would:

```
struct{
    int x, y;
} point1 = { 0, 0}, point2 = {0, 0};
```

This creates the two structure variables, ‘point1’ and ‘point2’, and initializes both members, ‘x’ and ‘y’, on each to zero (0). You don’t need to make a name-less definition to do this. You could still have a structure tag *and* initialize the variables created after the definition.

Note: Not only can you write name-less structures, but you can write them without declaring any variables. This will cause a warning on smarter compilers, but is perfectly legal:

```
struct
{
    int x, y;
};
```

The above structure definition can’t be used to create any variables because it has no name. Likewise, no variables are declared after it either. So this is a nice way to fill up your programs with useless source code. But why stop there? You could make an empty, name-less definition:

```
struct { };
```

6.1.4. Array of structs

```
#include<iostream.h>
#include<conio.h>
struct student {
       int id;
       char name[15];
   };

 void main()
   {
    clrscr();
    //creating 5 student using an array
    student s[5];
    int i;
   for(i=0; i < 5; i ++)
    {
      cout<<"\n Enter Student Id";
      cin>>s[i].id;
      cout<<"\nEnter Name";
      cin>>s[i].name;
    }
    cout<<"\n Displaying student Info";
    cout<<"\nStudent Id \t Student Name";
    cout<<"\n============================";

  for( i = 0; i < 5; i++)
      cout<<endl<<s[i].id<<"\t\t\t"<<s[i].name;

    getch();     }
```

Memory map of the above struct declaration.

0	id	1
	name	Tameru
1	id	2
	name	Hassen
2	id	3
	name	Selamawit
3	id	4
	name	Asia
4	id	5
	name	Micheal

If you use s[0].id you will be referring to 1, and s[0].name will refer to Tameru.

The following program declares and uses a book struct. Also swaps the content of two book struct variables.

```
#include<iostream.h>
#include<conio.h>
struct Book {
      int id;
      char title[15];
};
void main(){
       //creating three Book variables
       Book b1,b2,temp;
       cout<<"\n Enter Book Id";
       cin>>b1.id;
       cout<<"\nEnter Title";
       cin>>b1.title;
       cout<<"\n Enter Book Id";
       cin>>b2.id;
       cout<<"\nEnter Title";
       cin>>b2.title;
       cout<<"\n Book Information";
       cout<<"\n Before Changing Contents";
       cout<<"\n Book id\t Title";
       cout<<"\n=========================";
       cout<<endl<<b1.id<<"\t\t\t"<<b1.title;
       cout<<endl<<b2.id<<"\t\t\t"<<b2.title;
      //swapping content
       temp=b1;
       b1=b2;
       b2=temp;
       cout<<"\nAfter swapping contents";
       cout<<"\n Book Information";
       cout<<"\n Book id\t Title";
       cout<<"\n=========================";

       cout<<endl<<b1.id<<"\t"<<b1.title;
       cout<<endl<<b2.id<<"\t"<<b2.title;
       getch();
      }
```

6.1.5. Declaring struct types as part of a struct

A structure definition contains multiple variables, but not necessarily just primitives. You can define a structure to have *structure member variables*.

Now if you have data's like birth of day of an employee, published year of a book, address of a person. What are you going to do? You must be able to incorporate this type of data's in other structs. The following program declares two structs one for address and other for student.

```
#include<iostream.h>
#include<conio.h>
struct Address{
    int kebele;
    char Kefle_ketema[20];
    char roadname[20];
  };
struct Student {
   int id;
   char name[15];
   char section[6];
   //declaring address type within student
   Address studaddress;
   };
 void main(){
    clrscr();
    //creating Student type that encapsulates Address
    Student s1;
    cout<<"\n Enter Student Id";
    cin>>s1.id;
    cout<<"\nEnter Student Name";
    cin>>s1.name;
    cout<<"\n Enter Section";
    cin>>s1.section;
    //reading address attributes
    cout<<"\nEnter Kebele";
    cin>>s1.studaddress.kebele;
    cout<<"\nEnter Street Name";
    cin>>s1.studaddress.roadname;
    cout<<"\nEnter Kefle Ketema";
    cin>>s1.studaddress.Kefle_ketema;
    cout<<"\n Student Information";
    cout<<"\n id\t Name\t Section \t Kebele";
    cout<<" \t Street Name \t Kefele Ketema ";
    cout<<"\n================================";
    cout<<endl<<s1.id<<"\t"<<s1.name;
    cout<<"\t"<<s1.section<<"\t"<<s1.studaddress.kebele;
    cout<<"\t"<<s1.studaddress.roadname<<"\t";
    cout<<s1.studaddress.Kefle_ketema;
    getch();
}
```

The memory map of student s;

id		1
name		Asefa
section		RDDOB02
studaddress	kebele	21
	Kefle_ketema	Arada
	roadname	Gahandi

Using the above figure. You should be able to understand how data is accessed. Example s.id refers id of student, but s.studadress.kebele refers kebele of student where the kebele is referenced by studaddress, intern part of student.

Example: This assumes that 'date' and 'time' are structures that have already been declared.

```
struct moment
{
    date theDate;
    time theTime;
};
```

These structure members can then be accessed as normal member variables, but then *their* variables must be accessed as well. If there is a variable based on 'moment' called 'birth', then we would need to write the following to assign '19' to "birth's date's day":

```
birth.theDate.day = 19;
```

A structure only has to be declared before it can be used in another structure's definition. The following is perfectly acceptable:

```
struct date;
struct time;
struct moment
{
    date theDate;
    time theTime;
};
struct date
{
    int day, month, year;
};
struct time
{
    int sec, min, hour;
};
```

To be able to define a structure you only must know the types and the names of the member variables declared inside. With the above we declare the structures 'date' and 'time' but do not define them until later. This simply acknowledges that they exist and they can therefore be used within 'moment'.

What if ‘date’ and ‘time’ hadn’t defined? It would still be legal, but then I would not be able to *use* ‘moment’ at all. Why? Since ‘date’ or ‘time’ have not been defined, the compiler does not know how big they are supposed to be or what kind of data they contain. You couldn’t then create a variable based on ‘moment’ because the compiler doesn’t know how big of a memory block to allocate. Likewise if you try to use a structure that has been declared *before* it has been defined, you will encounter the same problem.

6.1.6. Defining Structure in Structure

It is possible to define a structure inside a structure definition and create variables from it at the same time. For example:

```
struct moment
{
    struct date
    {
        int day, month, year;
    } theDate;

    struct time
    {
        int second, minute, hour;
    } theTime;
};
```

The drawback of the above is that the ‘date’ and ‘time’ definitions cannot be used elsewhere without also referring to the parent structure. If the ‘date’ definition isn’t going to be used elsewhere anyway, the structure tag can simply be omitted. Thus we could remove the ‘date’ and ‘time’ structure type identifiers and it would work fine. You can write any number of variables in a structure definition and a valid variable declaration statement (minus initialization of course) is valid inside a structure definition. Let’s say we want to be able to use ‘date’ elsewhere, but not ‘time’. The following program demonstrates how the structure definitions would be written as well as uses the defined structure types in an extended “birth date” sample:

```
#include <iostream.h>
struct date { int day, month, year; } ;
struct moment
{
    date theDate;
    struct
    {
```

```
        int sec, min, hour;
    } theTime;
};
int main()
{
    moment birth;
    cout << "Enter your birth moment!" << endl;
    cout << "Year: ";
    cin >> birth.theDate.year;
    cout << "Month: ";
    cin >> birth.theDate.month;
    cout << "Day: ";
    cin >> birth.theDate.day;
    cout << "Hour (military): ";
    cin >> birth.theTime.hour;
    cout << "Minute: ";
    cin >> birth.theTime.min;
    cout << "Second: ";
    cin >> birth.theTime.sec;

    cout << "You entered " << birth.theDate.month << "/"
         << birth.theDate.day << "/" << birth.theDate.year
         << " @ " << birth.theDate.hour << ":"
         << birth.theDate.min << ":" << birth.theDate.sec
         << endl;

    if (birth.theTime.hour > 20 || birth.theTime.hour < 8)
        cout << "You were born early in the morning!"
             << endl;

    return 0;
}
```

Any number of structure definitions can be nested; you can get extremely complex with structures, which is why they are sometimes known as *complex types*.

6.1.7. Structure, Reference and Pointer

References to structure variables, work the same way as normal references. To create one you would write out the name of the structure type, followed by the reference name which is preceded by the reference operator (`&`):

```
struct_name &reference;
```

The following creates a structure variable based on ‘date’ and a reference to it:

```
date birth;
date &mybirth = birth;
```

Both of these, ‘birth’ and ‘mybirth’, would have access to the same member variables and their values. Thus they can be used interchangeably:

```
birth.year = 1981;
mybirth.year -= 2;
```

Can you guess what the value of 'birth.year' would be from the above? It would be '1979'. The reference 'mybirth' is just an alias to 'birth' and its group of member variables. Remember that a reference is not a real variable (it has no value), but simply a nickname for another.

Utilizing pointers with structures, unfortunately, adds a previously unseen complexity. A pointer to a variable cannot be used to modify the variable's value until it has been dereferenced. This case is no different from structures. But it affects the way you access the structure variable's members. Recall that a pointer simply contains a memory address. The pointer knows nothing of what this memory address is used for, which is why you have to dereference a pointer to a specific type. Thus, you cannot access a structure variable's members until you dereference a pointer to the structure type:

```
date birth;
date *p = &birth;
(*p).year = 1979;
```

The pointer 'p' above, had to be dereferenced before the member variable 'year' could be accessed through it. It was surrounded in parenthesis because the indirection operator (asterisk '*') has a lower precedence than the member operator. Thus the following would not work:

```
*p.year = 1979;
```

This would be seen as "get the value pointed to by '`p.year`'" and '`p.year`' is an invalid identifier; hence the parenthesis around the indirection operation. This method of accessing a structure variable's members is cumbersome and requires two operations simply to get at the member variable: indirection and then member. For this purpose, there is an operator specifically for accessing a structure variable's members through a pointer. This is a member operator known specifically as a pointer-to-member operator or a dash followed by a greater than sign '`->`' (also known as an "arrow"):

```
p->year = 1979;
```

This operator only works on pointers to structure variables. You cannot use it on normal structure variables for members. The following would not work:

```
birth->year = 1979;
```

The left operand must be a pointer to a variable with members, and the right operand must be the name of a member variable within that.

6.2.File Management

File handling is an important part of all programs. Most of the applications will have their own features to save some data to the local disk and read data from the disk again. Files which are on the secondary storage device are called physical files. In order to process file through program, logical file must be created on the RAM. This logical file is nothing but an object having file data type. As an object there should be a variable identifier that points to it. This variable is called file variable and some times also called file handler. C++ File I/O classes simplify such file read/write operations for the programmer by providing easier to use classes.

6.2.1. Streams and Files

The I/O system supplies a consistent interface to the C++ programmer independent of the actual device being accessed. This provides a level of abstraction between the programmer and the device. This abstraction is called stream. The actual device is called a file.

6.2.1.1. Streams

The C++ file system is designed to work with a wide variety of devices, including terminals, disk drives, and tape drives. Even though each device is very different, the C++ file system transforms each into a logical device called stream. There are two types of streams: text and binary.

a. Text Streams

A text stream is a sequence of characters. In a text stream, certain character translations may occur as required by the host environment. For example a new line may be converted to a carriage return/linefeed pair. There may not be a one-to-one relationship between the characters that are written (or read) and those on the external device. Because of possible transformations, the number of characters written (or read) may not be the same as those on the external device.

b. Binary streams

A binary stream is a sequence of bytes with a one-to-one correspondence to those in the external device i.e., no character translations occur. The number of bytes written (or read) is the same as the number on the external device. However, an implementation-

defined number of null bytes may be appended to a binary stream. These null bytes might be used to pad the information so that it fills a sector on a disk, for example.

6.2.1.2. Files

In C++, a file can be anything from a disk file to a terminal or printer. You associate a stream with a specific file by performing an *open* operation. Once a file is open, information can be exchanged between it and a program. *All streams* are the *same* but all files are not. If the file can support position requests, opening that file also initializes the *file position indicator* to the start of the file. As each character is *read* from or *written* to the file, the position indicator is *incremented*. You disassociate a file from a specific stream with a *close* operation. If you close a file opened for output, then contents, if any, of its associated stream are written to the external device. -- this process is referred to as *flushing* the *stream*. All files are closed automatically when the program terminates normally. Files are not closed when a program terminates abnormally. Each *stream* that is associated with a file has a *file control structure* of *type FILE*. This structure FILE is defined in the header stdio.h.

The File Pointer

A file pointer is a pointer to information that defines various things about the file, including its name, status, and the current t position of the file. In essence, the file pointer identifies a specific disk file and is used by the associated stream to direct the operation of the I/O functions. A file pointer is a pointer variable of type FILE.

```
FILE * fp;
```

6.2.1.3. The standard streams

When ever a program starts execution, three streams are opened automatically.

stdin --- standard input.

stdout -- standard output

stderr -- standard error

Normally, these streams refer to the console. Because the standard streams are file pointers, they can be used by the ANSI C file system to perform I/O operations on the console.

6.2.1.4. C++ File I/O Classes and Functions

To perform file I/O, the header file *fstream.h* is requied. fstream.h defines several classes, including *ifstream, ofstream,* and *fstream*. These classes are derived form *istream* and *ostream,* repectively. istream and ostream are derived form ios.

Three file I/O classes are used for File Read/Write operations:

a. ifstream - Can be used for File read/input operations
b. ofstream - Can be used for File write/output operations
c. fstream - Can be used for both read/write c++ file I/O operations

These classes are derived directly or indirectly from the classes istream, and ostream. We have already used objects whose types were these classes: cin is an object of class istream and cout is an object of class ostream. Therefore, we have already been using classes that are related to our file streams. And in fact, we can use our file streams the same way we are already used to use cin and cout, with the only difference that we have to associate these streams with physical files. Let's see an example:

6.2.2. Text and Binary Files

In file processing, files are generally classified into two as

- Text file and
- Binary file

Text file is a file in which its content is treated as a sequence of characters and can be accessed sequentially. Where as binary file is a file in which its content is treated as record sequence in a binary format. Binary format refers to the actual format the data is going to be placed and processed in the memory which is directly related to its data type.

For example, the value int count 321 will be stored in three byte if it is written in text file considering the digit sequence ‘3’, ‘2’, ‘1’. It will be stored in two byte if it is written in binary file since int requires two byte to store any of its value. When you open the binary file you will see the character equivalence of the two bytes.

321 in binary equals 0000 0001 0100 0001

The first byte holds the character with ASCII value equals to one and the second byte a character with ASCII value equals 65 which is ‘A’. Then if you open the binary file you will see these characters in place of 321.

6.2.3. Text File processing

File processing involves the following major steps

1. Declaring file variable identifier
2. Opening the file
3. Processing the file
4. Closing the file when process is completed.

6.2.3.1. Opening and Closing a file

An open file is represented within a program by a stream object and any input or output operation performed on this stream object will be applied to the physical file associated to it. In C++, you open a file by linking it to a stream. Before you can open a file, you must first obtain a stream. There are *three types* of *streams:* input, output, and input/output. To create an *input stream*, you must **declare** the stream to be of *class ifstream.* To create an *output stream*, you must **declare** it as *class ofstream.* Streams that will be performing both input and output operations must be declared as *class fstream.*

```
ifstream in ; //input stream
ofstream out ; // output stream
fstream io ; // input and output
```

Once you have declared a stream, you associate it with a file by using the method open(). The method open ( ) is a member of each of the three stream classes. Its prototype is:

void open (const char *filename, int mode, int access = filebuf::penprot);

Where:

Filename is the name of the file.

The value of the mode determines how the file is opened. It must be one (or more) of these values:

Mode	**Description**
ios::app	Write all output to the end of the file
ios::ate	Open a file for output and move to the end of the file (normally used to append data to a file). Data can be written anywhere in the file.
ios::binary	Cause the file to be opened in binary mode.
ios::in	Open a file for input
ios::nocreate	If the file does not exist, the open operation fails.
ios::noreplace	If the file exists, the open operation fails.

ios::out	Open a file for output
ios:trunc	Discard the file's content if it exists (this is also the default action ios::out)

You can combine two or more of these values by using them together.

```
ofstream out ;
out.open ( "test", ios::out); // correct statement
ofstream out1 ;
out.open ( " test"); // the default value of mode is ios::out –
                     // correct statment
```

To open a stream for input and output, you must specify both the ios::in and the ios::out mode values. (Noe default value for mode is supplied in this case.)

```
fstream myStream;
myStream.open ( "test", ios::in | ios::out );
If open ( ) fails, myStrream will be zero
if (myStream){
cout << "Cannot open a file.\n";
// handle error
}
```

Each one of the open() member functions of the classes ofstream, ifstream and fstream has a default mode that is used if the file is opened without a second argument:

class	default mode parameter
ofstream	ios::out
ifstream	ios::in
fstream	ios::in | ios::out

For ifstream and ofstream classes, ios::in and ios::out are automatically and respectively assumed, even if a mode that does not include them is passed as second argument to the open() member function. The default value is only applied if the function is called without specifying any value for the mode parameter. If the function is called with any value in that parameter the default mode is overridden, not combined.

File streams opened in binary mode perform input and output operations independently of any format considerations. Non-binary files are known as text files, and some translations may occur due to formatting of some special characters (like newline and carriage return characters).

Since the first task that is performed on a file stream object is generally to open a file, these three classes include a constructor that automatically calls the open() member function and has the exact same parameters as this member. Therefore, we could also

have declared the previous myfile object and conducted the same opening operation in our previous example by writing:

```
ofstream myfile ("example.bin", ios::out | ios::app | ios::binary);
```

Combining object construction and stream opening in a single statement. Both forms to open a file are valid and equivalent.

To check if a file stream was successful opening a file, you can do it by calling to member is_open() with no arguments. This member function returns a bool value of true in the case that indeed the stream object is associated with an open file, or false otherwise:

```
if (myfile.is_open()) { /* ok, proceed with output */ }
ifstream myStream ( "myfile" ); // open file for input
```

When we are finished with our input and output operations on a file we shall close it so that its resources become available again. In order to do that we have to call the stream's member function close(). This member function takes no parameters, and what it does is to flush the associated buffers and close the file:

```
myfile.close();
```

Once this member function is called, the stream object can be used to open another file, and the file is available again to be opened by other processes.

In case that an object is destructed while still associated with an open file, the destructor automatically calls the member function close(). The close method takes no parameters and returns no value.

6.2.3.2. Reading and writing text files

Simply use the << and >> operators in the same way you do when performing console I/O except that instead of using cin and cout, you substitute a stream that is linked to a file.

```
ofstream out ("inventory");
out <<"Radios" << 39.95 << endl;
out << "Toastors" << 19.95 << endl;
out.close ( );
```

Example: Basic file operations

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

```
int main () {
  ofstream myfile;
  myfile.open ("example.txt");
  myfile << "Writing this to a file.\n";
  myfile.close();
  return 0;
}
```

This code creates a file called example.txt and inserts a sentence into it in the same way we are used to do with cout, but using the file stream myfile instead.

Example 2: writing on a text file

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

int main () {
  ofstream myfile ("example.txt");
  if (myfile.is_open())
  {
    myfile << "This is a line.\n";
    myfile << "This is another line.\n";
    myfile.close();
  }
  else cout << "Unable to open file";
  return 0;
}
```

Example 3: reading a text file

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

int main () {
  string line;
  ifstream myfile ("example.txt");
  if (myfile.is_open())
  {
    while (! myfile.eof() )
    {
      getline (myfile,line);
      cout << line << endl;
    }
    myfile.close();
  }

  else cout << "Unable to open file";

  return 0;
}
```

This last example reads a text file and prints out its content on the screen. Notice how we have used a new member function, called eof() that returns true in the case that the end of the file has been reached. We have created a while loop that finishes when indeed myfile.eof() becomes true (i.e., the end of the file has been reached).

Checking state flags

In addition to eof(), which checks if the end of file has been reached, other member functions exist to check the state of a stream (all of them return a bool value):

Function	Description
bad()	Returns true if a reading or writing operation fails. For example in the case that we try to write to a file that is not open for writing or if the device where we try to write has no space left.
fail()	Returns true in the same cases as bad(), but also in the case that a format error happens, like when an alphabetical character is extracted when we are trying to read an integer number.
eof()	Returns true if a file open for reading has reached the end.
good()	It is the most generic state flag: it returns false in the same cases in which calling any of the previous functions would return true.

In order to reset the state flags checked by any of these member functions we have just seen we can use the member function clear(), which takes no parameters.

get and put stream pointers

All I/O streams objects have, at least, one internal stream pointer:

- ifstream, like istream, has a pointer known as the get pointer that points to the element to be read in the next input operation.
- ofstream, like ostream, has a pointer known as the put pointer that points to the location where the next element has to be written.
- Finally, fstream, inherits both, the get and the put pointers, from iostream (which is itself derived from both istream and ostream).

These internal stream pointers that point to the reading or writing locations within a stream can be manipulated using the following member functions:

tellg() and tellp()

These two member functions have no parameters and return a value of the member type pos_type, which is an integer data type representing the current position of the get stream pointer (in the case of tellg) or the put stream pointer (in the case of tellp).

seekg() and seekp()

These functions allow us to change the position of the get and put stream pointers. Both functions are overloaded with two different prototypes. The first prototype is:

```
seekg ( position );
seekp ( position );
```

Using this prototype the stream pointer is changed to the absolute position position (counting from the beginning of the file). The type for this parameter is the same as the one returned by functions tellg and tellp: the member type pos_type, which is an integer value.

The other prototype for these functions is:

```
seekg ( offset, direction );
seekp ( offset, direction );
```

Using this prototype, the position of the get or put pointer is set to an offset value relative to some specific point determined by the parameter direction. offset is of the member type off_type, which is also an integer type. And direction is of type seekdir, which is an enumerated type (enum) that determines the point from where offset is counted from, and that can take any of the following values:

ios::beg	offset counted from the beginning of the stream
ios::cur	offset counted from the current position of the stream pointer
ios::end	offset counted from the end of the stream

The following example uses the member functions we have just seen to obtain the size of a file:

```
Example:  obtaining file size
#include <iostream>
#include <fstream>
using namespace std;
int main () {
  long begin,end;
  ifstream myfile ("example.txt");
  begin = myfile.tellg();
  myfile.seekg (0, ios::end);
  end = myfile.tellg();
  myfile.close();
  cout << "size is: " << (end-begin) << " bytes.\n";
```

```
  return 0;
}
```

6.2.4. Binary File processing

In binary files, to input and output data with the extraction and insertion operators (<< and >>) and functions like getline is not efficient, since we do not need to format any data, and data may not use the separation codes used by text files to separate elements (like space, newline, etc...).

File streams include two member functions specifically designed to input and output binary data sequentially: write and read. The first one (write) is a member function of ostream inherited by ofstream. And read is a member function of istream that is inherited by ifstream. Objects of class fstream have both members. Their prototypes are:

```
write ( memory_block, size );
read ( memory_block, size );
```

Where memory_block is of type "pointer to char" (char*), and represents the address of an array of bytes where the read data elements are stored or from where the data elements to be written are taken. The size parameter is an integer value that specifies the number of characters to be read or written from/to the memory block.

There are *two ways* to write and read binary data to and from a file.

- get ( ) and put ( )
- read ( ) and write ( )

If you will be performing binary operations on a file, be sure to open it using the *ios::binary mode specifier.*

6.2.4.1. get ( ) and put ( )

These functions are byte-oriented.

- get ( ) will read a byte of data.
- put ( ) will write a bye of data.

The get ( ) method has many forms.

```
istream & get( char & ch );
ostream & put ( char ch);
```

The get ( ) method read a single character from the associated stream and puts the value in ch, and returns a reference to the stream. The put ( ) method writes ch to the stream and returns a reference to the stream.

```
char in;
ifstream in ( "test", ios::in | ios::binary);
if (!in){
cout <<"Cannot open file";
return 1;
}
while (in) //inn will be 0 when eof is reached
{  in.get ( ch );
    cout << ch;
}
```

When the end-of-file is reached, the stream associated with the file becomes zero.

```
ofstream out ( "chars", io::out | ios::binary);
for (int i= 0; i < 256; i++)
  out.put ( (char ) i ) ; //write all characters to disk
out.close ( );
```

6.2.4.2. read ( ) and write ( )

The read ( ) method reads num bytes from the associated stream, and puts them in a memory buffer (pointed to by buf).

```
istream & read ( unsigned char * buf, int num );
```

The write ( ) method writes num bytes to the associated stream from the memory buffer (pointed to by buf).

```
ostream & write ( const unsigned char * buf, int num );
```

If the end-of-file is reached before num characters have been read, then read ( ) simply stops, and the buffer contains as many characters as were available. You can find out how many characters have been read by using another member function, called gcount ( ), which has the prototype:

```
int gcount ( );
```

6.2.4.3. More get ( ) functions

The method get ( ) is overloaded in several ways.

istream &get (char *buf, int num, char delim = '\n');

This get ( ) method reads characters into the array pointed to by the buf until either num characters have been read, or the character specified by delim has been encountered. The array pointed to by buf will be null terminated by get ( ). If the delimiter character is encountered in the input stream, it is not extracted. Instead, it remains in the stream until the next input operation.

a. int get ( )

It returns the next character from the stream. It returns EOF if the end of file is encountered.

b. getline ( )

```
istream & getline ( char *buf, int num, char delim ='\n');
```

This method is virtually identical to the get (buf, num, delim) version of get ( ). The difference is getline ( ) *reads and removes the delimiter* from the input stream.

Example: reading a complete binary file

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

ifstream::pos_type size;
char * memblock;

int main () {
  ifstream file ("example.txt", ios::in|ios::binary|ios::ate);
  if (file.is_open())
  {
    size = file.tellg();
    memblock = new char [size];
    file.seekg (0, ios::beg);
    file.read (memblock, size);
    file.close();

    cout << "the complete file content is in memory";

    delete[] memblock;
  }
  else cout << "Unable to open file";
  return 0;
}
```

In this example the entire file is read and stored in a memory block. Let's examine how this is done:

First, the file is open with the ios::ate flag, which means that the get pointer will be positioned at the end of the file. This way, when we call to member tellg(), we will directly obtain the size of the file. Notice the type we have used to declare variable size:

```
ifstream::pos_type size;
```

ifstream::pos_type is a specific type used for buffer and file positioning and is the type returned by file.tellg(). This type is defined as an integer type, therefore we can conduct on it the same operations we conduct on any other integer value, and can safely be

converted to another integer type large enough to contain the size of the file. For a file with a size under 2GB we could use int:

```
int size;
size = (int) file.tellg();
```

Once we have obtained the size of the file, we request the allocation of a memory block large enough to hold the entire file:

```
memblock = new char[size];
```

Right after that, we proceed to set the get pointer at the beginning of the file (remember that we opened the file with this pointer at the end), then read the entire file, and finally close it:

```
file.seekg (0, ios::beg);
file.read (memblock, size);
file.close();
```

At this point we could operate with the data obtained from the file. Our program simply announces that the content of the file is in memory and then terminates.

Function	**Description**
Detecting EOF **int eof ( );**	It returns nonzero when the end of the file has been reached; otherwise it returns zero.
Reading and discarding characters from the input stream. **istream & ignore (int num = 1, int delim = EOF);**	Reads and discards characters until either num characters have been nignored (1 by default) or until the charcter specified by delim is encounterdd (EOF by default). If the delimiting character is encountered, it is not removed from the input stream.
Obtain the next character in the input stream without removing it from that stream **int peek ( );** **istream & putback (char c);**	One can obtain the next character in the input stream without removing it from that stream by using peek ( ). It returns the next character in the stream or EOF if the end of file is encountered. One can return the last character read from a stream to that stream using putback ( ).

Forcing data to be physically written to the disk **ostream & flush ( );**	When the output is performed, data is not necessarily immediately written to the physical device linked to the stream. Instead, information is stored in an internal buffer until the buffer is full. Only then are the contents of that buffer written to disk. However, you can force the information to be physically written to the disk before the buffer is full by calling flush ( ).

6.2.5. Random Access Files

In C++'s I/O system, you perform random access by using seekg ( ) and seekp ( ) methods.

```
istream *seekg (streamoff offset, sek_dir origin);
ostream & seekp ( streamoff offet, seek_dir origin);
```

Here, streamoff is a type defined in iostream.h that is capable of containing the largest valid value that offset can have. Also seed-dir is an enumeration that has these values:

```
  ios::beg
ios::cur
ios::end
```

The C++ I/O system manages two pointers associated with a file. One is the *get pointer*, which specifies where in then file the next input operation will occur. The other is the put pointer, which specifies where in the file the next output operation will occur. Each time an input or output operation takes place the appropriate pointer is automatically sequentially advanced. The seekg ( ) method moves the associated file's current get pointer offset number of bytes from the specified origin, which must be one of three values. The seekp ( ) method moves the associated file's current put pointer offset number of bytes from the specified origin, which must be one of three values.

6.2.5.1. Obtaining the Current File Position

You can determine the current position of each file pointer by using these methods.

```
streampos tellg ( );
streampows tellp ( );
```

Here, streampos is a type defined in iostream.h that is capable of holding the largest value that either function can return.

6.2.5.2. I/O Status

The C++ I/O system maintains status information about the outcome of each I/O operation. The current state of the I/O system is held in an integer, in which the following flags are encoded.

- *eofbit* -- 1 when end-of-file is encountered; 0 otherwise
- *failbit* -- 1 when a (possibly) nonfatal I/O error has occurred; 0 otherwise
- *badbit* -- 1 when a fatal I/O error has ocurred; 0 otherwise

These flags are enumerated inside ios. Also defined in ios is *goodbit*, which has the value 0. There are two ways in which you can obtain I/O status information.

a. Use the rdstate function/method.

```
int rdstate ( );
```

rdstate function returns the current status of the error flags encoded into an integer. It returns zero, when no error has occurred. Otherwise, an error bit is turned on.

b. Use one or more of these methods.

Method	Description
int bad ( )	Returns true if badbit is set.
int fail ( )	Returns true if failbit is set.
int eof ( )	Returns true if there are no errors.
int good ( )	Otherwise they return false.

Once an error has occurred, it may need to be cleared before your program continues. to do this, use the clear ( ) method.

```
void clear ( int flags = 0);
```

If flag = zero (as it is by default), all error flags are cleared (reset to zero). Otherwise, set flags to the flags or values you want to clear.

6.2.6. Buffers and Synchronization

When we operate with file streams, these are associated to an internal buffer of type streambuf. This buffer is a memory block that acts as an intermediary between the stream and the physical file. For example, with an ofstream, each time the member function put

(which writes a single character) is called, the character is not written directly to the physical file with which the stream is associated. Instead of that, the character is inserted in that stream's intermediate buffer.

When the buffer is flushed, all the data contained in it is written to the physical medium (if it is an output stream) or simply freed (if it is an input stream). This process is called synchronization and takes place under any of the following circumstances:

- **When the file is closed**: before closing a file all buffers that have not yet been flushed are synchronized and all pending data is written or read to the physical medium.
- **When the buffer is full**: Buffers have a certain size. When the buffer is full it is automatically synchronized.
- **Explicitly, with manipulators**: When certain manipulators are used on streams, an explicit synchronization takes place. These manipulators are: flush and endl.
- **Explicitly, with member function sync()**: Calling stream's member function sync(), which takes no parameters, causes an immediate synchronization. This function returns an int value equal to -1 if the stream has no associated buffer or in case of failure. Otherwise (if the stream buffer was successfully synchronized) it returns 0.

Exercise

1. Write a program that accept N student record from the keyboard & store the list on a file "D:\\ Test.txt" in a text file format
2. Write a program that reads students record from the text file "D:|\ Test.txt" and display on the screen.
3. Do Q_1 in binary format.
4. Do Q_2 in binary format.

Note Student record consists of first name, last name, gender, age and Id.

Solution

Consider the file **Header .h** which contains the basic preprocessing include files, structure definition and functions other than the main function.

Header. h *// this file is saved at D: and it has the following structure.*

```
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
struct studList
{
    char firstName[12];
    char lastName[12];
    int age;
    char gender;
    char Id[12];
};
studList getStudent()
{
    studList std;
    cout<<"Enter student first name ===>";cin>>std.firstName;
    cout<<"Enter student last name ===>";cin>>std.lastName;
    cout<<"Enter student age ===>";cin>>std.age;
    cout<<"Enter student gender ===>";cin>>std.gender;
    cout<<"Enter student Id ===>";cin>>std.Id;
    return std;
}
void DisplayStudent(studList std)
{
    cout<<"Student first name:\t"<<std.firstName<<endl;
    cout<<"Student last name :\t"<<std.lastName<<endl;
    cout<<"Student age:\t\t"<<std.age<<endl;
    cout<<"Student gender:\t\t"<<std.gender<<endl;
    cout<<"Student Id:\t\t"<<std.Id<<endl;
}
```

Solution 1

```
  #include "D:\ header.h"
  int main() {
      studList std;
      fstream outf;
      outf.open("d:\\test.txt",ios::app);
      if(outf.fail())   {
            cout<<"unable to open the file d:\test.txt\n";
            return 1;
      }
      clrscr();
      int N;
      cout<<"Enter the number of students ===> ";cin>>N;
      for(int i = 0; i < N; i++)    {
            std = getStudent();
            clrscr();
      outf<<std.firstName<<" "<<std.lastName<<" "<<std.Id<<" "
<<std.gender<<"  "<<std.age<<endl;
      }
      getch();
      return 0;
  }
```

Solution 2

```
  #include "D:\ header.h"
  int main(){
      studList std;
      fstream inpf;
      inpf.open("d:\\test.txt",ios::in);
      if(inpf.fail())
      {
            cout<<"unable to open the file d:\test.txt\n";
      return 1;
      }
      clrscr();
      while (!inpf.eof())
      {

      inpf>>std.firstName>>std.lastName>>std.Id>>std.gender>>std.age;
            if(inpf.eof()) break;
            DisplayStudent(std);
  cout<<"====================================\n";
            getch();
      }
      inpf.close();
      return 0;
  }
```

Solution 3

```
        #include "D:\ header.h"
        int main()
        {
            studList std;
            fstream outf;
            outf.open("d:\\test2.txt",ios::app|ios::binary);
            if(outf.fail())
            {
```

```
            cout<<"unable to open the file d:\test.txt\n";
        return 1;
        }
        clrscr();
        int N;
        cout<<"Enter the number of students ===> ";cin>>N;
        for(int i = 0; i < N; i++)
        {
            std = getStudent();
            clrscr();
            outf.write((char *) &std, sizeof(std));
        }
        getch();
        return 0;
    }
```

Solution 4

```
#include "D:\ header.h"
int main() {
    studList std;
    fstream inpf;
    inpf.open("d:\\test2.txt",ios::in|ios::binary);
    if(inpf.fail())
    {
        cout<<"unable to open the file d:\test.txt\n";
        return 1;
    }
    clrscr();
    while (!inpf.eof()){
    {
        inpf.read((char*)&std, sizeof(std));

    //inpf>>std.firstName>>std.lastName>>std.Id>>std.gender>>std.age;
        if(inpf.eof()) break;
        DisplayStudent(std);
        cout<<"=====================================\n";
        getch();
    }
    inpf.close();
    return 0;
}
```

6.3.Annex

6.3.1. Prototypes of C++ file I/O methods

Void open (const char * filename, int mode, int access = filebuf::openprot) ;

File open modes:

ios::app	Write all output to the end of the file
ios::ate	Open a file for output and move to the end of the file (normally used to append data to a file). Data can be written anywhere in the file.
ios::binary	Cause the to be opened in binary mode.
ios::in	Open a file for input
ios::out	Open a file for output
ios::trunc	Discard the file's content if it exists (this is also the default action ios::out)
ios::nocreate	If the file does not exist, the open operation fails.
ios::noreplace	If the file exists, the open operation fails.

istream & get (char & ch) ;
ostream & put (char ch) ;
istream & read (unsigned char * buf, int num) ;
ostream & write (const unsigned char * buf, int num) ;
int gcount ( ) ;
istream & get (char * buf, int num, char delim = '\n') ;
int get ( ) ;
istream & getline (char * buf, int num, char delim = '\n') ;
int eof ( );
istream & ignore (int num = 1, int delim = EOF) ;
int peek ( ) ;
istream & putback (char c) ;
ostream & flush ( ) ;
istream & seekg (streamoff offset, seek_dir origin) ;
ostream & seekp (streamoff offset, seek_dir origin) ;
origin must be one of three values:

ios::beg -- Beginning of file
ios::cur -- Current location
ios::end -- End of file

streampos tellg ( ) ;
stream pos tellp ( ) ;
istream & seekg (streampos pos) ;
ostream * seekp (streampos pos) ;
int rdstate ( ) ;
inr bad ( ) ;
int eof ( ) ;
int fail ( ) ;

```
int good ( );
void clear ( int flags = 0 ) ;
```