UNIT 5 STREAMS AND TEMPLATES

Structure Nos. **Page**

5.0 INTRODUCTION

In this Unit, we will discuss about C++ streams library. We have already used "<<" and ">>" for standard input and output. In this Unit, we will discuss the way of using the same operators for user defined types. Also, we will discuss the ways of reading data from and writing data to other files. We will discuss ways of opening files in different modes and closing them. In the context, we will be discussing about istream and ostream classes. In addition, we will also look into Exception Handling using C++.

5.1 OBJECTIVES

After going through this Unit, you should be able to:

- Write programs which perform input and output from built in data type;
- Write programs which perform input and output from user defined data types, and
- Write programs which open other files and perform operations on the data in those files.

5.2 OUTPUT

As we have seen in a number of previous programs, the standard operation for output is left shift operation "<<".

We have already seen as to how to apply this operator for built-in types. For example,
consider

```
int x = 5;
cout << x; // will print 5
```

Now, we will consider as to how to apply this operator for user defined types:

The following program demonstrates the application of this operator for user defined types:

```
#include <iostream.h>
class output

{
        int i ;
public:
        output (int j = 0)
        {
        i = j;
        }
        friend int show (output& a) {return a.i;}
}
ostream& operator << (ostream& s, output o)
{       return s << show (o);
}

void main ( )
{       output x (1);
        cout << "x=" << x;
}
```

5.3 INPUT

As previously seen, the standard operator for input is right shift operator ">>". We have already applied in it for built in data types. For example,
int i;
cin >>i; //reads the value into i.

Now, we shall apply it to user defined data types. For an input operations, it is essential that the second argument is of reference type.

The following program demonstrates use of operator ">>" for user defined types:

```
#include <iostream.h>
class output {
int i;
public :
        output (int j=0)
        {
        i=j;
        }
        friend int show (output& a )
        {
        return a.i;
        }
};
ostream&operator << (ostream& s, output o)
{
        return s << show (o);
}
istream& operator>>(istream& s, output& o)
{
        int i = 0;
        s >> i;
        if (s) o = output (i);
        return s;:
}

void main ( )
```

```
{
output x;
cin>>x;
cout << "x=" << x;
}
```

5.4 FILES AND STREAMS

In this Section, we will learn how to open files, close files and attaching files to streams.

The following example demonstrates those concepts. This is a program which copies the data from one file and copies it to another. The program received the names of the files as command line arguments.

```
#include <fstream.h>
void main (int argc, char* argv[])
{
if (argc ! = 3)
        {       cout << "The no. of arguments should be 3'';
                return;
        }
ifstream read(argv [1]);
ofstream write(argv [2]);
char c;
while(read.get(c)) write.put(c);
}
```

<fstream.h> is a library which declares the C++ stream classes that support file input and output. It also includes iostream.h

ifstream is a class that provides an input stream to input from a file using a buffer.

Now, hereafter, "read" will be the handle to the file presented as second argument. So, whenever we have to refer to the second argument, we will be using "read". So, the second argument on the command line will be opened from which data is read in future operations.

ofstream is a class which provides an output streams to extract from a file using a buffer.

Now, hereafter, "write" will be the handle to the file presented as third argument. So, whenever we have to refer to the third argument, we will be using "write". So, the third argument on the command line will be opened for writing the data in future operations.

Here "read" and "write" are two names which we have chosen on our own. The user can have any other names as handles to files.

So,finally read.get (c) will read a character from file argv [1] and write.put (c) will write it to file argv [2]. This is automatically done until the end of file is encountered.

An "ofstream" is opened for writing by default and "ifstream" is opened for reading by default.
Also, we can open a file in other modes. In this case, the above classes will accept a second argument.

The different modes are defined in the following class ios

```
class ios       {
```

```
public:
        enum open_mode
                {       in=1, out=2, ate = 4, app =010, trunc = 020, nocreate = 040,
                noreplace =0100  };
};
```

in mean “Open for reading”
out means “Open for output”
ate means “Open and seek to end of file”
app means “Append”
trunc means “truncate file to 0-length”
nocreate means “Fail if file does not exist”
noreplace means “ fail if file exists”.

Consider,

```
ofstream ex (file, ios::out | ios :: nocreate);
```

This means, “Open the file identified by variable “file” or output mode. If the file does not exist, the operation should fail.

We can also open a file for both input and output. For example, consider
```
fstream ex(name, ios::in | ios::out);
```

A file can be closed by calling the function close( ) on its stream. For example, consider

```
ex.close( );
```

5.5 TEMPLATES

Templates are also referred to as Parameterised types. It enables you to define generic classes. It defines a family of classes and functions. For example, stack of various data types such as int, float, etc. Similarly, function template for sort function. It enables function of classes and function with parameter.

A stack template:

```
template <class T> class stack
{

}
```

Template <class T> prefix in the class declaration states that you are going to declare a class template and you would use T as a class name in the declaration. Thus, stack is a parameterised class with the type T as its parameter. With this definition of the stack class template, you can create stacks for different data types, such as:
```
stack<int> istack;
```

```
stack<float> fstack;
```

You could similarly define a generic array class as follows:

```
Template <class T> class Array
{
        :
        :
        :
}
```

You can then create instance of different Array types in the following manner:

```
Array <int> iarray (128);
Array <float> farray (32);
```

Function templates

Like class templates, function templates define a family of functions parameterised by a data type. For example, you could define a parameterised sort function for sorting any type of array like this.

```
template <class T > void sort (Array <T>)
{
// Body of function (do the sorting)
:
:
:
}
```

You can invoke the sort function just like any ordinary function. The C++ compiler will analyse the arguments to the function and call the proper function.

Advantage of templates

- It helps you to define classes that are general in nature.

A Simple Stack Template

```
template <class T>  class stack {
T*v;
T*p; // Stack Pointer

int SZ;
public:
 stack (int s ) {v=p=new T [SZ = s] }
~stack ( ) {delete [ ] v;}
void Push (T a) {*p++ =a;}
T Pop () {return * --p;}
}
```

The template <class T> prefix specifies that a template is being declared and that an argument T of type <type> will be used in the declaration.
Template <class T> says that T is a type name, it need not actually be the name of a class. The name of a class template follow by a type bracketed by <T is the name of a class (as defined by template) and can be used exactly like other class names. For example,

```
Stack <char> SC (100); // Stack of characters defines an object SC of a class
                                //stack <char>
```

Except for the special syntax of its name, stack <char> works exactly as if it had been defined.

```
class Stack-char {
char*v;
 char *p;
int SZ;
public:
Stack-char (int s) {v=p=new char [SZ=s];
          :
          :
          :
```

};

One can think of a template as a clever kind of macro that obeys the scope, naming and types rules of C++.

It is important to write templates so that they have a few dependencies on global information as possible. The reason is that a template will be used to generate function and classes based on unknown types and in unknown contexts. Any context dependency will surface as a debugging tool.

5.6 EXCEPTION HANDLING

Exception means unusual condition while execution of a program. They may cause programs to fail or may lead to errors. Some exceptions can be “array out of bound”, null pointer assignment”, “file does not exist”, etc. The exception handling provides a uniform way of handling errors in C++ class libraries and programs. Let us discuss exception handling with the help of an example.

Let us assume a class
ReportIOException as:

```
Class ReportIOException {
Public:
ReportIOException (Char* filename):
_filename (filename) { }
Private:
        char* _filename;
}
```

In case a Report class any output related problem such as there is no space on the storage device can be thrown as ReportIOException as:

```
void Report:: write (const char* filename)
ofstream fs1 (filename)// open a file for output
if (!fs1)           // If cannot open a new file as no space
{                                       // throw exception
        throw ReportIOException (filename);
}
// continue normal processing
.
.
.
}
```

The code for catching the exception is provided by the programmer using the Report class. It may be as:

```
// prepare report and try to output it, but be ready for any error.
try
{
mis.write (“Report 1.rep”);
// In case of any error, the exception will
// be transferred to catch block
}
catch (ReportIOException fileio)
{
        //Display an error message
        cout << “ERROR! Cannot open file! Disk is full.”;
}
```

☞ Check Your Progress

1) "<<" and ">>" can be used for input and output on built-in data types as well as user defined types. True ☐ False ☐

2) "<<" and ">>" should be _____if they have to be applied for user defined types. True ☐ False ☐

3) For an input operation, it is essential that the second argument is of _____type. True ☐ False ☐

4) A close () function should be applied if we want to close the file before reaching the end of scope in which the stream was declared. True ☐ False ☐

5) It is possible to open a file in more than one mode simultaneously. True ☐ False ☐

5.7 SUMMARY

So, we can overload operators "<<" and ">>" for input and output on user defined types. For an input operation, it is essential that the second argument is of reference type. A file is opened for output for creating an object of class of stream and a file is opened for reading by creating an Object of class ifstream. Other than for reading and writing, we can also open a file in more than one mode simultaneously. Though, you can close a file with close () function, it is not necessary to close a file, because the conceived object will contain a destructor which closes a file after the execution of that particular programme. Anyway, the close () function can be applied if we want to close the file before reaching the end of the scope in which the stream was declared. We have also discussed about the exception handling in C++.
In a C++ programme, we can freely use any I/O functions, which are defined in C.

5.8 SOLUTIONS/ANSWERS

Check Your Progress

1) True
2) Overloaded
3) Reference
4) True
5) True.