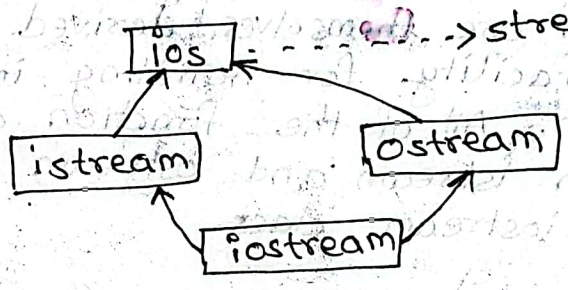
Chapter-8
Stream Computation for console and File Input/Output.

→ The input/output operations which involve keyboard as input device and screen (i.e monitor) as output device, are known as console I/O operation.

→ The input/output operations which read/write data from/to files are known as File I/O operations.

Stream class Hierarchy for console Input/Output.



→ ios is base class for istream and ostream which are in turn base class for iostream.

→ The class ios is declared as virtual base class so that only one copy of ios is copied to iostream.

→ class ios provides basic support for formatted and unformatted I/O operations.

Different stream classes associated with console input/output are discussed below.

ios

ios class is the top class of the stream class hierarchy. The ios class has data and function members that are essential for formatted input/output, error checking and status information related with stream input/output. It contains the pointer to a class streambuf which contains the actual memory buffer into which data is read or written and low level routine for handling this buffer.

istream

→ The class istream is derived from ios. It has member function like get(), getline(), read() to read data from standard input device. It also defines an extraction operator >>, as an overloaded operator function, to read data from a standard input to variables.

Ostream

→ The class ostream is derived from ios. It has member functions like put() and write(). The ostream class also defines an insertion operator << function by overloading the operator two write data from variable constants to the standard output device.

iostream.

→ The iostream class is derived from istream and ostre classes which are themselves derived from ios class. It provides facility for handling input as well as output stream. All of the function and operators available with istream and ostream classes are available in iostream class.

Streambuf.

→ provides an interface to physical devices through buffer.

Testing Stream Errors.

There can be a several abnormal situations in I/O process.

Some of the member function of ios class to test state of stream:

a) bool ios::good()

→ This function returns true when everything is okay that is when there is no error condition.

b) bool ios::eof()

→ This functions return true if the input operation reached end of input sequence.

c) bool ios::bad()

→ This function returns true if the stream is corrupted and no read/write operation can be performed.

d) bool ios::fail()

→ This function returns true if the input operation failed to read the expected characters, or that an output operation failed to generate the desired characters.

Unformatted Input/Output.

Unformatted operations do not allow user to read or display data in desired format. They read or write output data in default setting and we can't change it according to our choice.

The unformatted operations supported by c++ are:

1) Overloaded Operators >> and <<.

Eg: cout << message;
cin >> variable;

2) put() and get() functions.

→ get() assigns the input character and put() is used to output a character.

Eg:
```
#include <iostream.h>
int main()
{
char c;
cout << "Enter character";
cin.get(c);
cout << "The entered character is :";
cout.put (c);
return 0;
}
```

3) getline() and write() functions.

→ The function getline() reads single line or multiline text that ends with new line or specified delimiter until maximum limit is met.

Formatted Input/Output

Formatted I/O operations allow the input to read or the output displayed to be formatted according to our requirements. The features are:

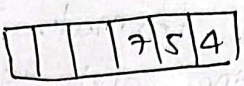
1) Member functions of ios class and flags.

→ Some important member functions of ios class that are used for formatting are:

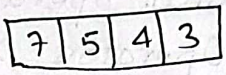
width()

→ to specify required field size for displaying an output value.

Eg:
```
cout.width(6);
cout << 754;
```

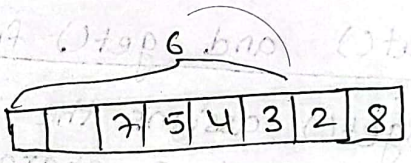



```
cout.width(3);
cout << 7543
```

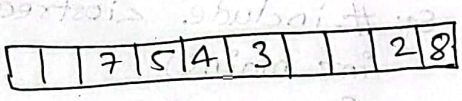


If the field width specified is smaller than the required width, the field is expanded.

```
cout.width(6);
cout << 7543;
cout << 28;
```




```
cout.width(6);
cout << 7543;
cout.width(4)
cout << 28;
```



precision()

→ to specify number of digits to be displayed after decimal point of a float value.

→ By default, float values are printed with six digits after decimal points.

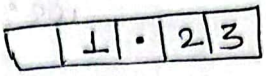
```
cout.precision(3);
cout << sqrt(2); // 1.412
cout << 1.2356; // 1.235
cout << 2.500032; // 2.5
```

Unlike width(), precision() retains settings in effect until it is reset.

```
cout.precision(3);
cout << sqrt(2);
cout.precision(5);
cout << sqrt(3);
cout.precision(0); //sets to default.
```

Combining field specifier.

```
cout.precision(2);
cout.width(5);
cout<<1.2345
```



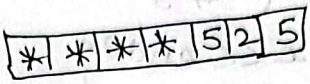
fill()

→ to specify a character that is used to fill unused portion of field.

```
cout.fill(character)
```

Eeg:

```
cout.fill('*');
cout.width(7);
cout<<525;
```



→ like precision(), fill() stays in effect till we change it.

```
cout.fill(); // change to default.
```

setf()

→ to specify format flag that can control the form of output display (such as left justification or right justification)

Syntax:

```
cout.setf(arg1, arg2);
```

arg1 → One of the formatting flags defined in the class ios.

arg2 → known as bit field that specifies the group to which formatting flag belongs.

→ The flag once set will remain as it is unless it is unset or new value is set.

The following table shows various flags and bit fields used for formatting output.

Flag value	Bit field value	Effect on output
ios::left	ios::adjustfield	Justifies o/p to left.
ios::right	ios::adjustfield	Justifies o/p to right.
ios::internal	ios::adjustfield	Filling character is displayed between sign and number or base indicator and number.
ios::dec	ios::basefield	Integer no. is displayed in decimal.
ios::oct	ios::basefield	Integer no is displayed in octal.
ios::hex	ios::basefield	Integer no is displayed in hexadecimal.

ios::scientific	ios::floatfield	Floating point number is displayed in exponential format.
ios::fixed	ios::floatfield	Floating point number is displayed in normal integer part dot fractional part format.

Eg:
```
int x = -456;
cout.setf (ios::internal, ios::adjustfield);
cout.width (6);
cout.fill ('#');
cout << x << endl;
```

Output:

```
-##456.
```

Following table shows different flags that do not have bit fields are are passed as argument to the one argument version of setf() function.

Flag value.	Effect on output.
ios::showbase	Prefix '0' on octal number and 0x on hexadecimal numb to indicate the base of number
ios::showpoint	show point and trailing zeros on fraction part.
ios::showpos	show + sign for positive number
ios::uppercase	show letter in number in uppercase like X in 0x, E in exponential format and digits a-e in hexadecimal number.
ios::skipws	skip whitespace character on input.
ios::boolalpha	display 'true' or 'false' instead of 1 or 0.
ios::unitbuf	flush output stream after each output operations.

Eg:
```
int x = 118;
cout.setf (ios::hex, ios::basefield);
cout.setf (ios::showbase);
cout << x;
```
Output 0x76:

Eg:
```
int x = 255
cout.setf (ios::hex, ios::basefield);
cout.setf (ios::uppercase);
cout << x;
```
Output : FF.

Unsetf()

→ Used to clear the flag specified.

Eg:
```
cout.unsetf (ios::showpos);
```
⇒ After this statement the output will not display '+' sign in displaying positive number.

Standard Manipulators.

→ To call ios function for formatting, we need to write separate statement and call the function through stream objects cin and cout. Manipulators are the formatting function for input/output that are embedded directly into the c++ input/output statements so that extra statements are not required.

As according to the number of arguments to be supplied, manipulators are categorized in the following types.

→ Non-parameterized Manipulators ⇒ eg. endl.

→ parameterized Manipulators. ⇒ eg. setw().

Manipulators.

setw(int)
setprecision(int)
setfill(char)
setiosflags(long)
resetiosflags(long)

Member Function.

width()
precision()
fill()
setf()
unsetf()

User-defined manipulators.

→ We can define our own manipulator according to the requirement.

Syntax:

```
ostream &manipulator_name (ostream &os)
  {
    //body
    return output;
  }.
```

Eg: WAP to create manipular equivalent to 'it'.

```
# include <iostream.h>
# include <iomanip.h>

ostream &tab (ostream &o)
    {
    o<< "\t";
    return o.
    }
int main()
    {
    int num1 = 32, num2 = 52, num3 = 75;
    cout << num1 << tab<< num2 << tab <<num3;
    }.
```

Output:

32 52 75.

Stream Operator Overloading.

→ To read and display user defined data types like the built in data types, we have to overload stream extraction (>>) and insertion (<<) operator to accept the user defined data.

Without stream operator overloading, we need to define member function like display() for a class and call them as:

```
obj1. display();
obj2. display();
```

But with overloading of insertion operator both the objects can be displayed conveniently as:

```
cout << obj1 << obj2 << endl;
```

Insertion Operator (<<) overloading.

Syntax:

```
ostream & operator <<(ostream &os, Myclass & myobj)
  {
  //....
  }.
```

For eg: class Time

```
  {
  //....
  friend ostream & operator <<(ostream & os, Time & tm)
  };
ostream & operator << (ostream & os , Time & tm)
  {
  os << tm.hour << ":" << tm.min << ":" << tm.sec;
  return os;
  }
```

Here, the first parameter os of overloaded insertion operator is the reference to ostream object particularly cout and the second parameter is the object to be printed. This overloaded insertion operator function must be declared as friend in the user defined class to access the private data member of the class whose data is to be displayed.

Extraction Operator (>>) overloading.

Syntax:

```
istream & operator >> (istream&is, Myclass & myobj)
  {
  //...
  }
```

Eg: class Time

```
  {
  //...
  friend istream & operator >> (istream & is, Time & tm);
  };
istream & operator >> (istream & is, Time & tm)
  {
  is >> tm.hour >> tm.min >> tm.sec;
  return is;
  }.
```

Eg: WAP to illustrates the overloading of stream Operators.

```
#include <iostream.hs
class Time.
{
private:
  int hour, min, se;
public:
  Time()
  {
  hour = 0, min = 0, sec = 0;
  }
friend istream & operator >> (istream &is, Time &tm)
friend ostream & operator << (ostream &os, Time &tm)
};
istream & operator >> (istream & is, Time &tm)
{
 is >> tm.hour >> tm.min >> tm.sec;
 return is;
 }
ostream & operator << (ostream &os, Time &tm)
 {
 os << tm.hour << ":" << tm.min << ":" << tm.sec <<
                                          flush;
  return os;
  }
int main()
  {
  Time t1;
  Cout << "Enter current time:";
  Cin >> t1;
  Cout << "Time entered is :" << t1;
  return 0;
  }
```

Output:

```
Enter current time : 2 57 35
Time entered is : 2 : 57 : 35.
```

File Input/Output with Streams.

→ Cout and Cin are used for console o/p and input.
→ They are used because it is not necessary to store information for further use.
→ Many application may require a large amount of data to be read, processed and also saved for later use.
→ Such information is stored on the auxiliary memory device in the form of data file.
→ File manipulation is done through stream classes ifstream, ofstream and fstream.

For file manipulation, object of file stream class is created and associated with the file. The file stream classes are declared in header file <fstream>.

File stream class Hierarchy

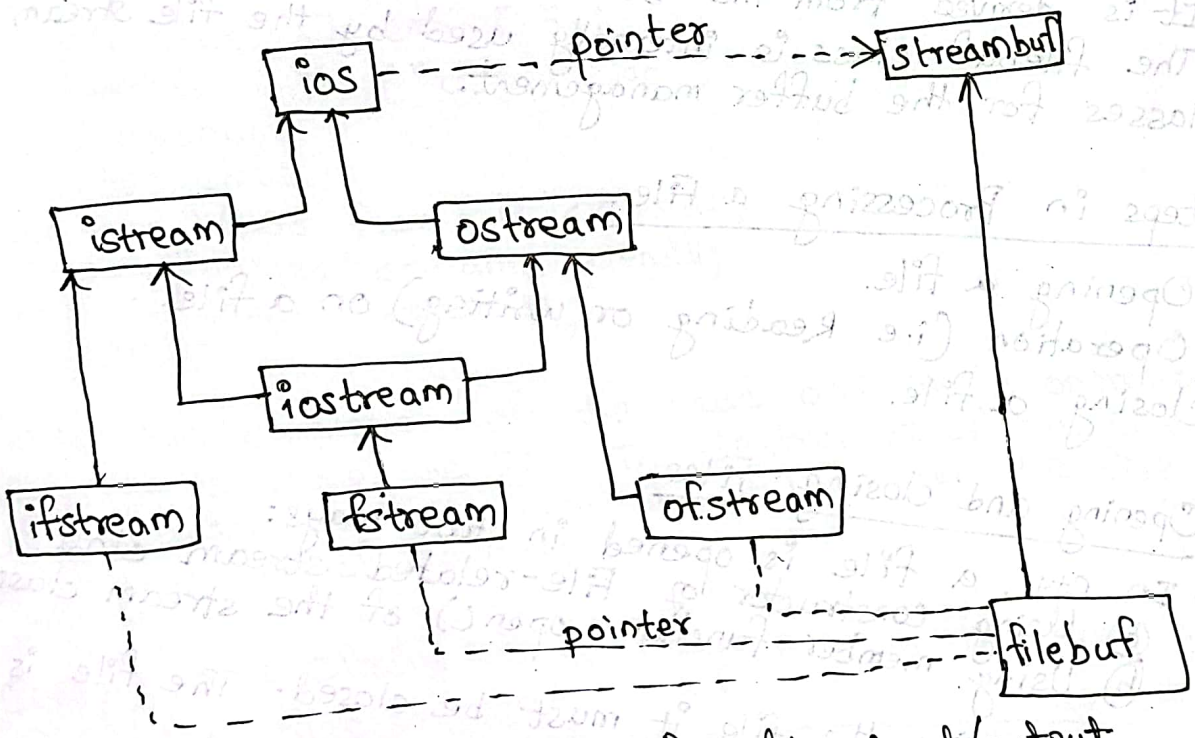


Fig: Stream class hierarchy for file input/output

ifstream

→ provides input operations for file.
→ Contains function open() with default input mode to open a file for input.
→ inherits input functions such as get(), getline(), read(), seekg() and tellg() from istream class.

ofstream.

→ provides output operations for file.
→ contains function open() with default output mode to open a file for output.
→ inherits output function such as put(), write(), seekp() and tellp() from ostream class.

fstream

→ It supports both input and output operations for file.
→ Contains function open() with default input and output mode to open a file for input and output.
→ inherits all functions from iostream class which actually inherits features from istream and ostream classes.

filebuf.

→ The filebuf class is used to set the file buffer to read and write.
→ It is derived from the base streambuf.
→ The filebuf class is internally used by the file stream classes for the buffer management.

Steps in Processing a File.

1. Opening a file.
2. Operation (i.e Reading or writing) on a file.
3. Closing a file.

Opening and closing Files.

→ In C++, a file is opened in two ways:
   (a) Using constructor of File-related stream class.
   (b) Using member function open() of the stream class.

→ After using the file it must be closed. The file is closed in two ways:
   (a) Using destructor of file stream class.
   (b) Using member function close() of the stream class.

Opening and closing Files Using Constructor and Destructor.

Eg: WAP to illustrates how data can be written to file.

```
#include <iostream>
#include <fstream>
```

```
Using namespace std;
int main()
{
char name[25], email[25];
ofstream outfile ("contact.doc");
cout <<"Enter name:";
cin >> name;
outfile << name << endl;
cout << "Enter email:";
cin >> email;
outfile << email;
return 0;
}
```

Eg: WAP to illustrates how data can be read from file.

```
#include <iostream>
#include <fstream>
Using namespace std;
int main()
{
char name[25], email[25];
ifstream infile ("contact.doc");
infile >> name;
infile >> email;
cout << "Data from file" << endl;
cout << "name:" << name << endl;
cout <<"email:" << email;
return 0;
```

After the completion of the read and write operation it is better to close the file. The destructors of the file stream classes automatically close the file when the file stream object goes out of the scope.

Opening and closing files Using member function / Opening and closing files explicitly.

→ The stream classes ifstream, ofstream and fstream contain a member function open(). We can use this function to open files. This approach of using file stream members function open() and close() is generally used when different files are to be associated with the same object at different times.

→ After the operation on the file, we have to close the file. The file stream class destructors close the file when their

objects go out of the scope. But, when we need to use the same file stream to be associated with different data files, the files should be opend and closed explicitly.

Q. WAP for opening file using open function for writing.

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
char name[25], email[25];
ofstream outfile;
outfile.open("contact.doc");
cout<<"Enter name:";
cin>>name;
cout<<"Enter email:";
cin>>email;
outfile<<name<<endl;
outfile<<email;
return 0;
```

Eg: WAP for opening file using open function for reading

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
 {
 char name[25], email[25];
 ifstream infile;
infile.open("contact.doc");
infile>>name;
infile>>email;
cout<<"Data from file"<<endl;
cout<<"name:"<<name<<endl;
cout<<"email:"<<email;
return 0;
 }
```

Eg:

```
ofstream fout;
fout.open("data.txt");
  //some operations.
  //.....
fout.close();
// closes the file first before opening different file
fout.open("mydata.doc");
      //opens another file by the same object fout.
```

File Modes:

→ While opening a file, we specified only a single argument in both constructor or member function open().

In this case, the default modes are used. With ifstream class default mode is input and with ofstream class the default mode is output, and with fstream class the default mode is both input and output.

→ But we can explicitly specify the mode while opening the file. The file mode is passed in second argument explicitly.

Syntax:

```
fstream stream_obj;
stream_obj.open("filename", mode);
```

Different file modes are:

1. ios::in

→ This mode opens a file for reading. The file open will be unsuccessful if we try to open a non-existing file in read mode.

2. ios::out.

→ This mode opens a file for writing. When a file is opened in this mode, it also opens in the ios::trunc mode by default. If specified file already exists, it is truncated.

3. ios::ate

→ When a file is opened in this mode, file access pointer is set at the end of file.

4. ios::app

→ When a file is open in this mode, file is opened in write mode with the file access pointer at the end of file. This mode can be used only with output files.

5. ios::trunc

When a file is opened in this mode, the file is truncated if a file with specified name already exists.

6. ios::binary

→ when a file is opened in this mode, the file is opened as a binary file and not an ASCII text file.

Detecting End of file.

→ we can either use an object of istream class which returns 0 on an end of file or we can use eof() which is a member function of the ios class.

Let us consider fin is object of class ifstream, the following code detects end of file.

```
while (fin)
  { // code to read content of file when
    not end of file.
  }
```

or

```
while (fin.eof() ==0)
  {
  // code to read content of file when
not end of file.
  }.
```

Reading / writing a character from a file.

→ The stream member function get() of istream class and put() of ostream class are designed to read and display a character at a time.

→ The function get() is used to read a single character from the file and the function put() is used to write a single character to the file.

Eg: WAP to illustrate the use of function put()

```
#include <iostream>
# include <fstream>
# include <cstring>
 using namespace std;
 int main ()
   {
```

```
char name[25], email[25];
int i;
fstream file;
file.open("name_list.txt", ios::out);   // file mode
cout<<"Enter name";
cin>>name;
cout<<"Enter email";
cin>>email;
for (i=0; i<strlen(name); i++)
        file.put(name[i]);
        file.put('\n');
for (i=0; i<strlen(email); i++)
    file.put(email[i]);
file.put('\n');
return 0;
}
```

Eg: WAP to illustrates the use of function get()

```
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;
int main()
{
char name[25], email[25];
int i;
fstream file;
file.open("name_list.txt", ios::in);
while (file)
{
i=0;
while (name[i] != '\n')
    file.get(name[i++]);
cout<<name;
i=0;
while (email[i] != '\n')
    file.get(email[i++]);
cout<<email;
} return 0;}
```

Machines uses binary format to store information rather than ASCII format. Binary format makes storing and retrieval faster. To store or retrieve binary data member functions write() and read() of istream and ofstream classes are used respectively.

Syntax:

```
file_obj.write (reinterpret_cast<char *> (&variable),
                sizeof (variable));
```

or

```
file_obj.write ((char *)(&variable), sizeof (variable));
file_obj.read (reinterpret_cast<char *>(&variable),
               sizeof (variable));
```

or

```
file_obj.write ((char *)(&variable), sizeof (variable));
```

Eg:

Writing data to file (binary)

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
  {
  char name[25];
  int age;
  float salary;
  ofstream outfile ("employee.dat", ios::binary);
  cout << "Enter Name:";
  cin >> name;
  cout << "Enter Age:";
  cin >> age;
  cout << "Enter Salary:";
  cin >> salary;
  outfile.write (reinterpret_cast<char *>(name), sizeof (name));
  outfile.write (reinterpret_cast<char *>(&age), sizeof (age));
  outfile.write (reinterpret_cast<char *>(&salary), sizeof(salary));
  return 0;
  }
```

Reading data from file.

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
 {
char name[25];
int age;
float salary;
fstream infile ("employee.dat", ios::binary);
infile.read (reinterpret_cast <char *> (name), sizeof(name));
infile.read (reinterpret_cast <char *> (& age), sizeof(age));
infile.read (reinterpret_cast <char *> (& salary), sizeof(salary));
cout << " Data from file" << endl;
cout << " name: " << name << endl;
cout << "age:" << age << endl;
cout << " salary:" << salary;
return 0;
}.
```

File Access Pointers and their Manipulators.

→ Till now the programs are sequential. While reading data from a file, the data items are read from beginning of the file in sequence until end of the file. Similarly, while writing data to a file, data items are placed one after another in a sequence. But it is necessary to access a particular data item placed in any location without starting from the beginning. Such type of access can be obtained by the concept of file pointer.

→ There are two types of file access pointers. One for input called get pointer and another for output called put pointer.

→ The getpointer facilities the movement in file while reading and put pointer faciliates the movement while writing.

⇒ The getpointer specifies a location from where the next reading operation is performed.

⇒ The putpointer specifies a location from where the next writing operation is performed.

⇒ When a file is opened in read mode, the i/p pointer 'get' is initialized to the beginning of file such that it can be read from start.

⇒ When a file is set in write mode, the o/p pointer 'put' is set to the beginning of file to write from start.

If the filename specified already exists, the newfile with same name is created and the old file is overwritten.

⇒ When the file is opened in append mode, the o/p pointer 'put' is set to point to the end of file so that data can be added from end of existing content.

There are functions for manipulation of these file pointers such as: seekg(), seekp() tellg() and tellp(). These functions are used to set a file access pointer to any desired position inside file or to get current file access pointer.

seekg()

⇒ The seekg() is a member function of istream class and this function moves get file pointer to a specific location.

Syntax:

```
stream_object.seekg(offset, reference_position);
```

where,

offset specifies the number of bytes the file pointer to be moved from the location specified by the other argument reference_position. The reference_position takes one of the following constants defined in ios class:

ios::beg → beginning of the file
ios::cur → current position of file
ios::end → end of the file

Eg:

Eg:→ seekg(10, ios::beg) i.e moves the getpointer by 10 bytes from beginning of the file.
→ seekg(10, ios::cur) i.e. moves the getpointer by 10 bytes from current position of the pointer.
→ seekg(-5, ios::end) i.e. moves the get pointer by 5 in backward direction from the end of the file.
→ seekg(0, ios::end) i.e moves the get pointer at the end of the file.
→ seekg(0, ios::beg) i.e moves the get pointer at the beginning of the file.

```
char ch;
ifstream infile("myfile.txt", ios::binary);
infile.seekg(5);
infile.read(&ch, sizeof(ch));
infile.seekg(-3, ios::cur);
infile.read(&ch, sizeof(ch));
```

read a character from 5th character of the file "myfile.txt" and another character from 3rd position of the file because first read moves the file access pointer to 6th position.
Attempting to seek beyond the beginning or end of a file puts the stream into the bad() state.

2) seekp()

⇒ The seekp() is a member function of ostream class and this function moves put file pointer to a specific location.

Syntax:

```
stream_object.seekp(offset, reference_position);
```

Eg: seekp(20, ios:beg) i.e moves the put pointer by 20 bytes from beginning of the files.

Eg: ofstream outfile ("myfile.txt");
outfile.seekp(15);
outfile << "**";
outfile.seekp(-5, ios::cur);
outfile << "#";

Write ** to 15th and 16th character positi
of the file "myfile.txt". This causes the file
access pointer to move to 16th position. The next
seekp() moves the access pointer to 5 position
back. So, "#" is written at position 11.

3) tellg()

→ The tellg() is a member function of istream
class and this function returns the current
file access pointer position of the get pointer.

Syntax:

int_variable = stream_object.tellg()

4) tellp()

→ The tellp() is a member function of ostream
class and this function returns the current
file access pointer position of the put pointer.

Syntax:

int_variable = stream_object.tellp();

Sequential Access to File.

```
#include <iostream>
#include <fstream>
using namespace std;
class student
{
private:
    char name[25];
    int rollno;
public:
```

```
Void read_data()
  {
  cout <<" Enter name :";
  cin>> name;
  cout<<" Enter rollno : ";
   cin >> rollno;
   cout << endl;
   }
  Void show_data()
   {
   Cout <<" Name :" <<name <<endl;
   cout <<" Roll.No :" <<rollno <<endl;
    Cout <<endl;
    }
   };
 Void write2file()
     {
     student stu;
     ofstream outfile ("record.dat", ios::binary |
                                     ios::app);
     stu.dread_data();
     outfile.write (reinterpret_cast <char*>
                  (&stu), sizeof (stu);
      }
 Void readfromfile()
  {
  student stu;
  cout <<"\n Data from file" <<endl;
  ifstream infile ("record.data", ios::binary);
  while ( !infile.eof())
  {
  if (infile.read (reinterpret_cast <char*>
                  (&stu), sizeof (stu))>0)
       stu.show_data();
  }
  }
```

```
int main()
  {
   int choice;
   cout <<" Student Record System";
   cout <<" Select one option below ";
   cout <<"\n t1 Write Records to file";
   cout <<" t2 Read from file";
   cout <<" t3 Exit from program";
  while (true)
    {
    cout<< " Enter your choice: ";
    cin >> choice;
    switch (choice)
      {
      case 1:
        write2file();
        break;
      case 2:
          readfromfile();
          break;
      case3:
          exit(0);
          break;
      default:
        cout <<" \n Choice not available ";
        exit(0);
        }
      }
    return 0;
    }
```

Random Access to File.

```
#include <iostream>
#include <fstream>
using namespace std;
class student
    {
```

```
private:
    char name[25];
    int rollno;
public:
  Void read_data()
   {
   cout << " Enter name :";
   cin >> name;
   cout << " Enter rollno :";
   cin >> rollno;
   cout << endl;
   }
 Void show_data()
   {
   cout << "Name:" << name << endl;
   cout << "Roll No:" << rollno << endl;
   cout << endl;
  }
};
Void write2file()
   {
   student stu;
   ofstream outfile("record.dat", ios::binary |
                                  ios::app);
   stu.read_data();
   outfile.write(reinterpret_cast<char *>(&stu), sizeof(stu));
   }
  Void readfromfile()
  {
  student stu;
  cout << "Data from file" << endl;
  ifstream infile("record.dat", ios::binary);
  while (!infile.eof())
  {
  if(infile.read(reinterpret_cast<char *>(&stu),
                 sizeof(stu))>0)

   stu.show_data();
         }}
```

```
void readonerec()
{
student stu;
int n;
ifstream infile("record.dat", ios::binary);
cout << "Enter Record Number:";
cin>>n;
infile.seekg((n-1) *sizeof(stu));
infile.read(reinterpret_cast<char *>(&stu),
                    sizeof(stu));
stu.show_data();
}
int main()
{
int choice;
cout << "Student Record System";
cout << "Select One option below";
cout << " 1 Write Records to file";
cout << " 2 Read All Records from File";
cout << " Read One Record";
cout << " Exit from Program";
while (true)
    {
    cout << "Enter your choice:";
    cin >> choice;
    switch (choice)
    {
    case 1 :
        write2file();
        break;
    case 2:
        readfromfile();
        break;
    case 3;
        readonerec();
        break;
    default:
        cout<<" choice not available";
        exit(0); } }
        return 0; }
```

Testing Errors during file Operations.

→ Some potential situations where errors may arise during file manipulations and the mechanism for checking errors during those situations are given below:

While attempting to open a non existing file in read mode ifstream infile ("data.txt");

```
if (!infile)
  {
   // file does not exist.
  }
```

while attempting to open a file marked as read only

```
ofstream outfile ("data.txt");
if (!outfile)
   {
    // file for read only.
   }
```

While trying to open a file by specifying filename not permitted by operating system.

```
infile.open ("#_/?.dat";
  if (!infile)
    {//invalid file name
    }
```

while attempting invalid operation such as read a file beyond the end of the file.

```
while (1)
    {
     // if (infile.eof ())
      break;
     }
```

while attempting to manipulate an unopened file

```
infile.read (reinterpret_cast <char *> (obj), sizeof
                                              (obj)).
    {
     // file not opened
     }
```

while attempting to manipulate a file stored in malfunctioning media and while attempting to access the file through file object which is not yet associated to any file.

```
infile.read (reinterpret_cast<char *>(obj), sizeof
                                        (obj))
if (infile.bad ())
   {
   // file cannot be read.
   }
```

while attempting to write a file where there is insufficient disk space.

```
ofstream outfile;
outfile.open ("data.txt");
if (outfile.bad ())
{
  // Writing not possible.
}
```