OOP with C++ (MCA 111)
Unit 4

File Handling in C++

- File Handling is used for store a data permanently in computer. Using file handling we can store our data in Secondary memory (Hard disk).
- **File:** The information / data stored under a specific name on a storage device, is called a file.
- **Stream:** It refers to a sequence of bytes.

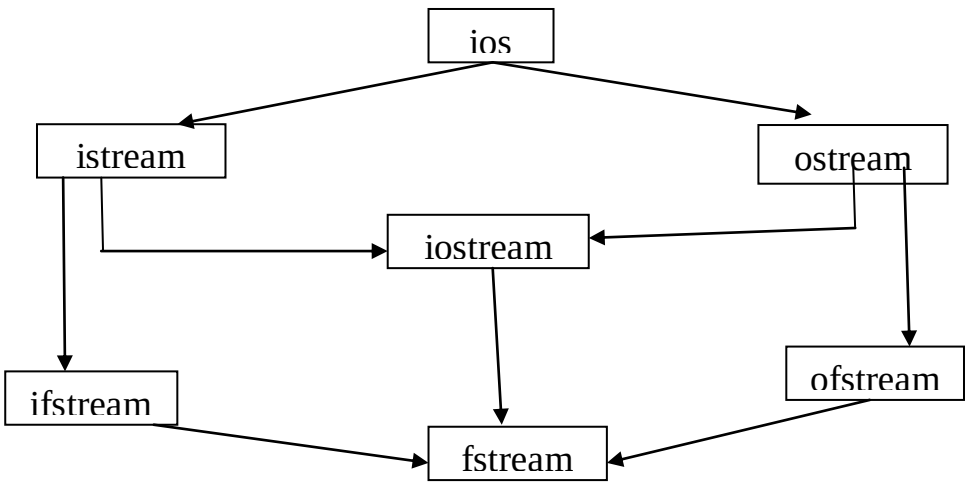
Why use File Handling in C++

- Memory is volatile
- For permanet storage.
- The transfer of input - data or output - data from one computer to another can be easily done by using files.

Classes for File Stream Operations:

- We have been using the **iostream** standard library, which provides **cin and cout** methods for reading from standard input and writing to standard output respectively.
- The another C++ standard library called **fstream** used to read and write from a file.

Datatype	Description
ofstream #include<ofstream>	This is used to create a file and write data on files. (Output file stream)
ifstream #include<ifstream>	This is used to read data from files. (Input file stream)
fstream #include<fstream>	This is used to both read and write data from/to files. (Both Input and Output)



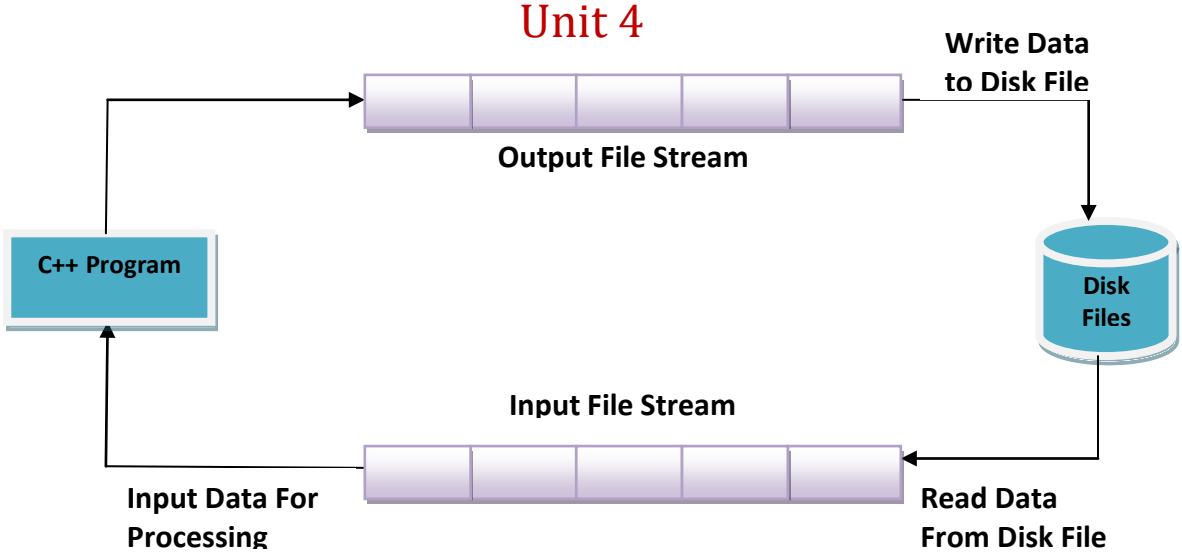


Fig : File Handling in c++

How to achieve File Handling

For achieving file handling in C++ we need follow following steps:

- Naming a file
- Opening a file
- Reading data from file
- Writing data into file
- Closing a file

Functions use in File Handling:

Function	Operation
open()	To create a file
close()	To close an existing file

Sequential input and output operations in file: The file stream classes support a number of member functions for performing the input and output operations on files.

Function	Operation
get()	Read a single character from a file
put()	write a single character in file.
read()	Read data from file
write()	Write data into file.

OOP with C++ (MCA 111)
Unit 4

Opening a File:

- A file must be opened before you can read from it or write to it. Either **ofstream or fstream** object may be used to open a file for writing. And **ifstream** object is used to open a file for reading purpose only.
- The function open() can be used to open multiple files that use the same stream object.

Syntax:

```
file-stream-class stream-object;
stream-object.open("filename");
```

File Opening mode:

Mode	Meaning	Purpose
ios :: out	Write	Open the file for write only.
ios :: in	read	Open the file for read only.
ios :: app	Appending	Open the file for appending data to end-of-file.
ios :: ate	Appending	take us to the end of the file when it is opened.

Example1:

```
ifstream  outfile;      // create stream
outfile . open ("data.txt", ios :: in);  // connect stream to data1
```

Example2:

```
fstream file;
file.Open("data . txt", ios :: out | ios :: in);
```

Closing a File:

- When a C++ program terminates it automatically close all the streams, release all the allocated memory and close all the opened files before program termination.
- A file must be close after completion of all operation related to file. For closing file we need close() function.

Syntax:

```
fstream outfile;
outfile.close();
```

OOP with C++ (MCA 111)
Unit 4

1. Opening a File

```
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
void main()
{
   fstream st; // Step 1: Creating object of fstream class
   st.open("E:\data.txt",ios::out);  // Step 2: Creating new file
   if(!st) // Step 3: Checking whether file exist
   {
      cout<<"File creation failed";
   }
   else
   {
      cout<<"New file created";
      st.close(); // Step 4: Closing file
   }
   getch();
}
```

2. Writing to a File

```
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
void main()
{
   fstream st; // Step 1: Creating object of fstream class
   st.open("E:\data.txt",ios::out);  // Step 2: Creating new file
   if(!st) // Step 3: Checking whether file exist
   {
      cout<<"File creation failed";
   }
   else
   {
      cout<<"New file created";
      st<<"Hello";    // Step 4: Writing to file
      st.close(); // Step 5: Closing file
   }
   getch();
}
```

3. Reading from a File

```
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
void main()
{
   fstream st; // step 1: Creating object of fstream class
   st.open("E:\data.txt",ios::in);   // Step 2: Creating new file
   if(!st) // Step 3: Checking whether file exist
   {
      cout<<"No such file";
   }
   else
   {
      char ch;
      while (!st.eof())
      {
         //st >>ch;  // Step 4: Reading from file
         //cout << ch;   // Message Read from file
        ch=st.get();
        cout<<ch;

      }
      st.close(); // Step 5: Closing file
   }
   getch();
}
```

File pointer in C++:

- Each file have two associated pointers known as the file pointers. One of them is called the input pointer (or get pointer) and the other is called the output pointer (or put pointer).
- The input pointer is used for reading the contents of a given file location and the output pointer is used for writing to a given file location.
- **Function for manipulation of file pointer** When we want to move file pointer to desired position then use these function for manage the file pointers.

Function	Operation
seekp()	moves put pointer (output) to a specified location.(put data from program to file)
tellp()	gives the current position of the put pointer.(put data from program to file)
seekg()	moves get pointer (input) to a specified location. (get data from file to program)
tellg()	gives the current position of the get pointer.(get data from file to program)
fout . seekg(0, ios :: beg)	go to start
fout . seekg(0, ios :: cur)	stay at current position
fout . seekg(0, ios :: end)	go to the end of file
fout . seekg(m, ios :: beg)	move to m+1 byte in the file
fout . seekg(m, ios :: cur)	go forward by m bytes from the current position
fout . seekg(-m, ios :: cur)	go backward by m bytes from the current position
fout . seekg(-m, ios :: end)	go backward by m bytes from the end

put() and get() function:

The function put() write a single character to the associated stream. Similarly, the function get() reads a single character from the associated stream.

Example1:tellp() and seekp() function works

```
#include<iostream.h>
#include<fstream.h>
void main()
{
   fstream file;
   file.open("E:\data.txt",ios::out|ios::in);
   cout<<file.tellp()<<endl;
   file<<"Hello program";
   cout<< file.tellp()<<endl;
   file.seekp(-7,ios::end);
  cout<< file.tellp()<<endl;
```

```
  file<<"C++";
 file.seekp(0,ios::beg);
file<<"HELLO";
//Read file data
char ch;
while(!file.eof())
{
ch=file.get();
cout<<ch;
}
file.close();
}
```

Example2: tellg() and seekg() function works

```
#include<iostream.h>
#include<fstream.h>
void main()
{
fstream file;
file.open("E:\data.txt",ios::out|ios::in);
file<<"Hello C++ Programmig";
cout<<file.tellg()<<endl;
file.seekg(5,ios::beg);
cout<<file.tellg()<<endl;
//Read file data
char ch;
while(!file.eof())
{
ch=file.get();
cout<<ch;
}
cout<<file.tellg()<<endl;
file.close();
}
```

OOP with C++ (MCA 111)
Unit 4

C++ Templates

- A template is a powerful feature added to C++. It allows function and classes to operate with different data types (generic types).
- It allows a single template to deal with a generic data types.

Templates can be represented in two ways:

1. Function templates
2. Class templates

Class templates:

- Class can also be declared to operate on different data types are known as class template.
- **For example,** a class template can be created for the array class that can accept the array of various types such as int array, float array or double array.

Syntax:

```
template<class Ttype>
class class_name
{
  //body
}
```

Class template with multiple parameters: We can use more than one generic data type in a class template, and each generic data type is separated by the comma.

Syntax

```
template<class T1, class T2, ......>
class class_name
{
   // Body of the class.
}
```

we create an instance of a class:

```
class_name<type> ob;
```

Example:

```
#include <iostream>
 using namespace std;
 template<class T1, class T2>
class A
{
    T1 a;
    T2 b;
    public:
   A(T1 x,T2 y)
  {
     a = x;
     b = y;
  }
     void display()
    {
         cout << "Values of a and b are : " << a<<" ,"<<b<<std::endl;
     }
 };
 int main()
{
     A<int,float> d(5,6.5);
     d.display();
     return 0;
}
```

Function Template

- Generic functions use the concept of a function template. Generic functions define a set of operations that can be applied to the various types of data.
- The type of the data that the function will operate on depends on the type of the data passed as a parameter.

Syntax

```
template < class Ttype>
ret_type func_name(parameter_list)
{
   // body of function.
}
```

Non template Function:

```
#include <iostream>
using namespace std;
void add(int a,int b)
{
    int result;
    result = a+b;
    cout<<“result”<<result<<endl;
}
void add(float a, float b)
{
    float result;
    result = a+b;
    cout<<“result”<<result<<endl;  }
int main()  {
  int i =2;
  int j =3;
  float m = 2.3;
  float n = 1.2;
  add(i,j);
  cout<<'\n';
  add(m,n);
  return 0;  }
```

Template Function:

```
#include <iostream>
using namespace std;
template<class T>
void add(T a,T b)
{
    T result;
    result = a+b;
    cout<<“result”<<result<<endl;
}
int main()
{
  int i =2;
  int j =3;
  float m = 2.3;
  float n = 1.2;
  add(i,j);
  cout<<'\n';
  add(m,n);
  return 0;
}
```

Function Templates with Multiple Parameters:

Example:

```
#include <iostream>
using namespace std;
template<class X,class Y>
void fun(X a,Y b)
{
    cout << "Value of a is : " <<a<< endl;
    cout << "Value of b is : " <<b<< endl;  }
int main()
{
   fun(15,12.3);
   return 0;
}
```

Overloading of Template Function

- We can overload a template function either by a non-template function or by a another template function.
- The different template function can have same name but they must have different number of parameters.
- A template function cannot be overloaded if they have same number of arguments list.

Example:

```
#include <iostream>
using namespace std;
template<class X>
void fun(X a)
{
   cout << "Value of a is : " <<a<< endl;
}
template<class X,class Y>
void fun(X b ,Y c)
{
   cout << "Value of b is : " <<b<< endl;
   cout << "Value of c is : " <<c<< endl;
}
void fun(float w ,float z)
{
   cout << "Value of w is : " <<w<< endl;
   cout << "Value of z is : " <<z<< endl;
}
int main()
{
  fun(10);
  fun(20,30.5);
  fun(1.5,2.5);
  return 0;
}
```

OOP with C++ (MCA 111)
Unit 4

Member Function Templates:

The member function of a class template are called as member function template.

Syntax:

```
template < class T1,class T2,.....>
ret_type class_Name <T1,T2,.....> :: func_name(parameter_list)
{
   // body of function.
}
```

Example:

```
#include <iostream>
using namespace std;
template<class T>
class A{
T a,b;
public:
void set(T x,T y);
T add();
};

template<class T>
void A<T> :: set(T x,T y)
{
a=x;
b=y;
}

template<class T>
T A<T> :: add()
{
T n;
n=a+b;
return n;
}

int main()
{
A <int> n1;
A <float> n2;
n1.set(10,20);
n2.set(10.5,5.5);
int s1=n1.add();
float s2=n2.add();
cout<<"\n sum1="<<s1;
cout<<"\n sum2="<<s2;
}
```