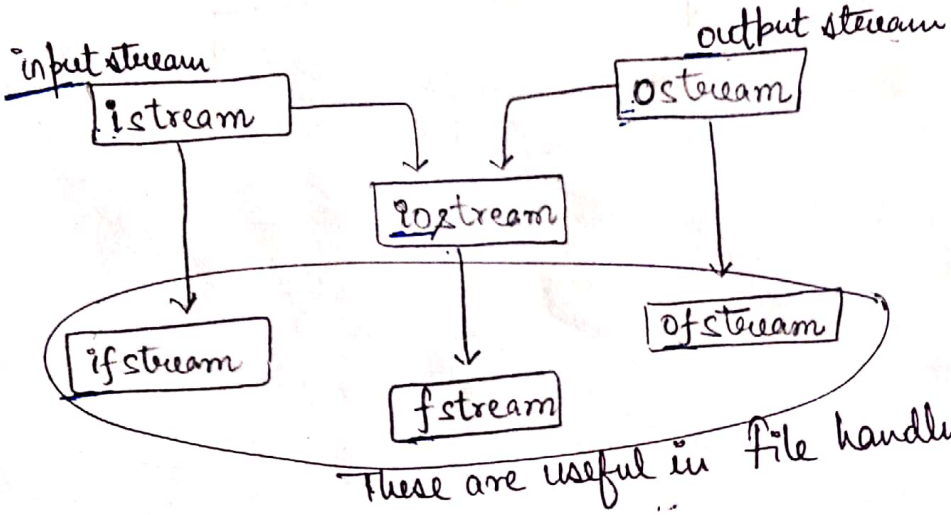
# FILES AND STREAMS

iostream standard library, which provides Cin & Cout methods. Stream is a sequence of bytes

→ fstream :- supports for simultaneous i/p and o/p operations on files. (File can be used for both i/p & o/p)

## Stream Classes




istream = input stream, deals with all inputs, e.g. Cin >>

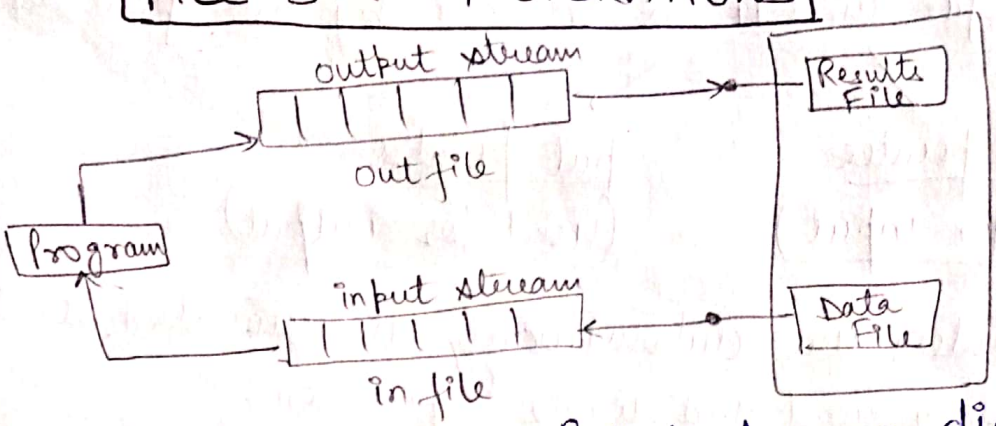
ostream = output stream, deals with all output, e.g. Cout <<

ifstream – represents the input file stream and used to read the information from files.

ofstream – represents the output file stream and used to create files and to write information to files.

fstream – represents the file stream generally and has the capabilities of both ofstream & ifstream which means it can create files, write information to files, & read information from files.

# FILE STREAM OPERATIONS




A C++ program can take input from a disk file and also write data to it.

It uses streams (file streams), (streams are sequence of bytes) as an interface between programs and files.

Above figure shows a C++ program that reads data from one file and writes output to another file.

Input stream: The stream that supplies data to the program is called "input stream"

Output stream: The stream that receiving data from program for writing to file is called output stream

# FILE POINTERS AND OPERATIONS

→ A file pointer is not a normal C++ pointer.
→ " " " will not store address of any variable.
→ Rather, A file pointer will tell the current location in the file where the reading or writing will take place.
→ A file pointer is like a text cursor in a text file.
→ Hence a file pointer will tell us the position of the cursor in the file.

position of cursor
↑ ← file pointer

→ Every file in C++ is marked by two pointers.

- Get pointer (used for input)
- put pointer (used for output)

→ These pointers are automatically incremented every time after every read and write operation.

→ Every time we open a file for input, the input pointer is automatically set to the beginning of the file.

seekg(), seekp(), tellg(), tellp() are commonly used functions.

→ seekg() and tellg() → used for input streams for get pointer.

The seekg() func. moves the input (get) pointer to a specified location. tellg() tells the current position of get pointer

→ seekp() and tellp() → for output streams for put pointers.

seekp() function moves to output (put) pointer to a specified location. tellp() tells the current position of put pointer

- seekg() - this func. is member of ifstream. It is used to move the get pointer to the specific location.
- seekp() - member of ofstream. It is used to moves the put pointer to the specific location.
- tellg() - this func. is member of ifstream. It gives the current position of get pointer.
- tellp() - this func. is member of ofstream. It gives the current position of the put pointer.

# Reference pointer for seekg() and seekp()

- ios : beg ← for beginning
- ios : cur ← current
- ios : end ← ending

**Syntax**

```
seekg (Number of Bytes, reference_ptr);
seekp (Number of Bytes, reference_ptr);
```

---

# TEMPLATES

Templates are powerful features of C++ which allows you to write generic programs. In simple terms, you can create a single function or a class to work with different data types using templates.

The concept of templates can be used in 2 different ways-

- Function Templates
- Class Templates

## FUNCTION TEMPLATE -

→ Function templates are special functions that can operate with generic types.

→ This allows us to create a function template whose functionality can be adopted to more than one type or class without repeating the entire code for each type.

→ The simple idea is to pass data type as a parameter so that we donot need to write same code for different data types.

→ We write a generic function that can be used for different data types.

## Function Template Syntax -

```
template <typename T>
T add (Ta , Tb)
{ return a+b ;
}
```

Example :-

```
# include <conio.h>
using namespace std;
template < class X >
X big (Xa , Xb)
{
  if (a>b)
        return (a);
  else
        return (b);
}
int main ()
{
  cout << big (4,5);
  cout << big (5.6, 3.4);
  getch ();
}
```

Place holder 'X' replaced by data types

place holder

int

double data type

## CLASS TEMPLATE – make it easy to reuse the same code for all data types.

→ Like function templates, we can also create class templates for generic class operations.

→ Sometimes, we need a class implementation that is same for all classes, only the data types used are different.

→ Normally, we need to create a different class for each data type OR create different member variables & functions within a single class.

→ In Class Template, we write a CLASS that can be used for different data types.

Syntax :-

```
template <class T1, class T2>
Class classname
{
  attributes;
  methods;
};
```

```
#include <iostream>
template <class T1, class T2>
class Example
{
 T1 x;
 T2 y;
 Public:
  Example (T1 a, T2 b)
  { x=a;
    y=b;
  }
  void show()
  {
   cout<<x<<"and"<<y<<"\n";
  }
};
  int main()
{
```

continue →

```
Example <float, int>
        test1(3.45, 345);
Example <int, char> test2
        (100, 'm');
test1.show()
test2.show()
return(0);
};
```

# EXCEPTION HANDLING

The process of converting system error message into user friendly error message is known as Exception Handling.

→ It is an event, which occurs during the ~~exception~~ execution of program, that disrupts the normal flow of the program instruction.

→ ERRORS can be classified into 2 types –

① Compile time error – error caught during compile time
~~Exe~~ – Syntax error, missing braces {}, missing semicolon
~~etc.~~ incorrect class import etc.

② Runtime error – That is occur during program execution after successful Compilation.
– It is also known as Exception. An exception caught during runtime creates serious issues.

Example - user divides a no. of zero, this will x/0
Compiles successfully but an Exception or Runtime error will occur due to which our application will be crashed.

In order to avoid this we will introduce Exception Handling technique in our code.

# Exception Handling Mechanism

1. Find problem (HIT the exception)
2. Inform about its occurance (THROW the exception)
3. Receive error information (CATCH the exception)
4. Take proper action (HANDLE the exception)

| try block |
| --- |
| Detects and throw Exception |

```
try
{
  code.....
  throw parameter;
}
```

exception

| Catch Block |
| --- |
| Catches and Handles the Exceptions |

```
Catch (type argument)
{
  //code to handle exception
}
```

⇒ TRY BLOCK is intended to throw exception which is followed by catch block. (ONLY 1 Try Block)

⇒ CATCH BLOCK is intended to catch the error and handle the exception. (We can have multiple catch blocks)

⇒ THROW BLOCK - throw keyword, throw an exception encountered inside try block. It
It mainly communicate information about error.

## Example :-

Here the prog. compiles successfully BUT fails during runtime.

```
# include <iostream>
  using namespace std;
  int main ()
{
  int a=10, b=0, c;
  c = a/b;
  return 0;
}
```

← error occurs during runtime (10/0 = x)

∴ the prog. will crash.

## Implementation of try-catch-throw statements.

```
# include <iostream>
  using namespace std;
  int main ()
{ int a=10, b=0, c;
try   //activates exception handling
  {
  if (b==0)
   throw "division by zero is not possible";
   c = a/b;
  }
catch (char *ex) //catch the exception//
  { cout << ex;
  }
  return 0;
}
```

output

| |
|---|
| 0P = 0 |