OBJECT ORIENTED PROGRAMMING (PC-CS203A)

UNIT-IV

Prepared By:

Er. Tanu Sharma

(A.P. CSE Deptt., PIET)

Syllabus (Unit-4)

Text Streams and binary stream, Sequential and Random Access File, Stream Input/ Output Classes, Stream Manipulators.

Basics of C++ Exception Handling, Try, Throw, Catch, multiple catch, Re-throwing an Exception, Exception specifications.

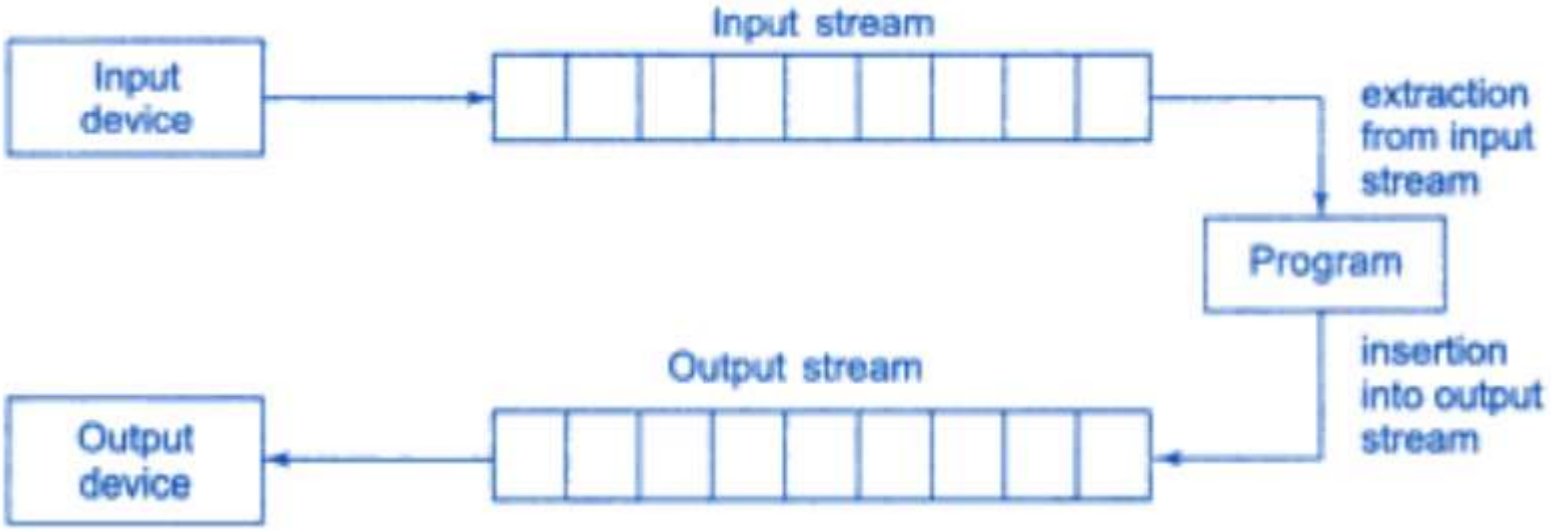
Templates: Function Templates, Overloading Template Functions, Class Template, Class Templates and Non- Type Template arguments.

FILE HANDLING

C++ Streams

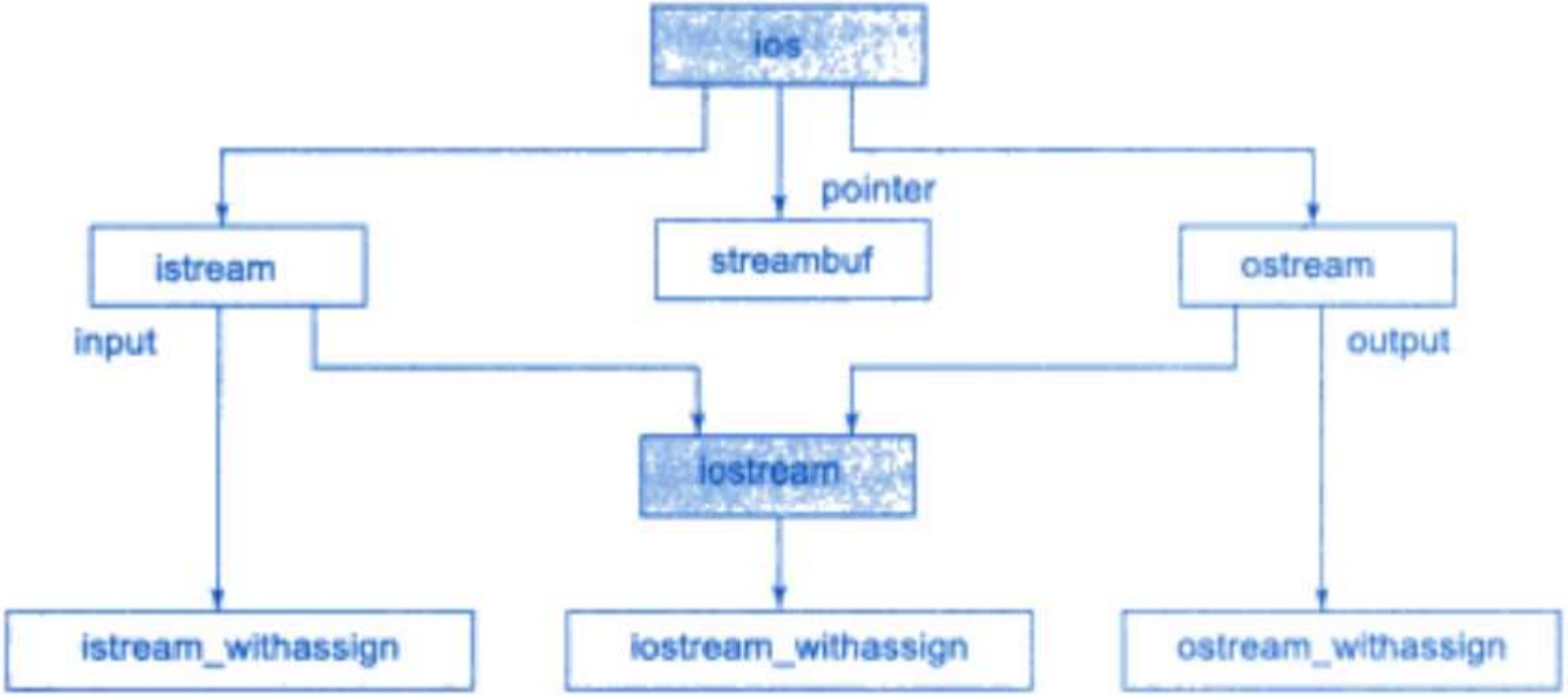
- A stream is an interface between I/O system and the actual hardware device being used.
- It is a sequence of bytes.
- Acts as either source or destination from/to which input/output data can be received or send.
  - The source stream that provides data to the program is called **input stream**.
  - The destination stream that receives output from the program is called **output stream**.

Data Streams



Stream Classes

- C++ I/O system contains a hierarchy of classes used to define various streams to deal with console and disk files.
- These classes are called stream classes.



Unformatted I/O Operations

- Unformatted I/O functions are used for performing I/O operations at console and the resulting data is left unformatted and untransformed i.e. in its raw and original form.

- istream class
  - get() //single character input.
  - getline() //read line of text
- ostream class
  - put() //single character output
  - write() //display line of text

get()

- get(char *)
  - assigns input character to its argument
  - E.g.

    ```
    char ch;
    cin.get(ch);        //single character typed will be read
                        and stored in character variable ch.
    ```

- get() or get(void)
  - returns the input character
  - E.g.

    ```
    char ch;
    ch=cin.get();       //value returned by get() is assigned to ch.
    ```

put()

- Can be used to display single character

```
char ch;
cout.put(ch);     //display value of ch
cout.put('m')     //display character m
cout.put(68);     //displays character whose ASCII
                  value is 68 i.e. D
```

Example Program

```
#include<iostream.h>
#include<conio.h>
int main()
{
        int count=0;
        char c;
        cout<<"\nInput Text\n";
        cin.get(c);
        while (c != '\n')
        {
                cout.put(c);
                count++;
                cin.get(c);
        }
        cout<<"\nNumber of characters: "<<count;
        getch();
        return 0;
}
```

```
Input
     Object Oriented Programming
Output
     Object Oriented Programming
     Number of characters = 27
```

getline()

- Reads a whole line of text that ends with a newline character.

- Can be invoked by:

  cin.getline(line,size);

- Reading is terminated when:
  - Newline character in detected
  - (Size) characters are read

- E.g.

  char name[20];

  cin.getline(name,20);

```
int main()
{
    int size = 20;
    char city[20];

    cout << "Enter city name: \n";
    cin  >> city;
    cout << "City name:" << city << "\n\n";

    cout << "Enter city name again: \n";
    cin.getline(city, size);
    cout << "City name now: " << city << "\n\n";

    cout << "Enter another city name: \n";
    cin.getline(city, size);
    cout << "New city name: " << city << "\n\n";

    return 0;
}
```

The output of Program

```
First run
     Enter city name:
     Delhi
     City name: Delhi

     Enter city name again:
     City name now:
     Enter another city name:
     Chennai
     New city name: Chennai

Second run
     Enter city name:
     New Delhi
     City name: New

     Enter city name again:
     City name now: Delhi

     Enter another city name:
     Greater Bombay
     New city name: Greater Bombay
```

write()

- Displays the entire line
- General form:

  cout.write(line,size)

  – *line*- represents the name of string to be displayed
  – *size*- indicates the number of characters to be displayed
- If size of string is greater than the length of line, it displays beyond the bounds of line.

Concatenation of strings

- Strings can be concatenated using write() function.
- E.g.
- cout.write (string1, m).write (string2, n);

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    char * string1 = "C++ ";
    char * string2 = "Programming";
    int m = strlen(string1);
    int n = strlen(string2);

    for(int i=1; i<n; i++)
    {
        cout.write(string2,i);
        cout << "\n";
    }

    for(i=n; i>0; i--)
    {
        cout.write(string2,i);
        cout << "\n";
    }
    // concatenating strings
    cout.write(string1,m).write(string2,n);

    cout << "\n";

    // crossing the boundary
    cout.write(string1,10);

    return 0;
}
```

Look at the output

```
P
Pr
Pro
Prog
Progr
Progra
Program
Programm
Programmi
Programmin
Programming
Programmin
Programmi
Programm
Program
Progra
Progr
Prog
Pro
Pr
P
C++ Programming
C++ Progr
```

Formatted I/O Operations

- There are certain functions which help us to format output in various ways.
- Few of them are:

Table 10.2 ios *format functions*

Function	*Task*
Width ()	To specify the required field size for displaying an output value
precision ()	To specify the number of digits to be displayed after the decimal point of a float value
fill()	To specify a character that is used to fill the unused portion of a field
setf()	To specify format flags that can control the form of output display (such as left-justification and right-justification)
unsetf()	To clear the flags specified

Equivalent Manipulators

Table 10.3 *Manipulators*

Manipulators	*Equivalent ios function*
setw()	width()
setprecision()	precision()
setfill()	fill()
setiosflags()	setf()
resetiosflags()	unsetf()

Table 10.4 *Flags and bit fields for setf() function*

Format required	*Flag (arg1)*	*Bit-field (arg2)*
Left-justified output	ios :: left	ios :: adjustfield
Right-justified output	ios :: right	ios :: adjustfield
Padding after sign or base	ios :: internal	ios:: adjustfield
Indicator (like +##20)		
Scientific notation	ios :: scientific	ios :: floatfield
Fixed point notation	ios :: fixed	ios :: floatfield
Decimal base	ios :: dec	ios :: basefield
Octal base	ios :: oct	ios :: basefield
Hexadecimal base	ios :: hex	ios :: basefield

Flags used in C++

Flag	Meaning
boolalpha	Boolean values can be input/output using the words "true" and "false".
dec	Numeric values are displayed in decimal.
fixed	Display floating point values using normal notation (as opposed to scientific).
hex	Numeric values are displayed in hexidecimal.
internal	If a numeric value is padded to fill a field, spaces are inserted between the sign and base character.
left	Output is left justified.
oct	Numeric values are displayed in octal.
right	Output is right justified.
scientific	Display floating point values using scientific notation.
showbase	Display the base of all numeric values.
showpoint	Display a decimal and extra zeros, even when not needed.
showpos	Display a leading plus sign before positive numeric values.
skipws	Discard whitespace characters (spaces, tabs, newlines) when reading from a stream.
unitbuf	Flush the buffer after each insertion.
uppercase	Display the "e" of scientific notation and the "x" of hexidecimal notation as capital letters.

width()

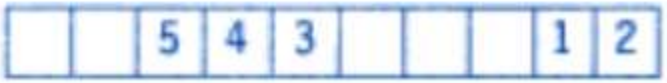
cout.width(w);

```
cout.width(5);
cout << 543  <<  12 << "\n";
```

produce the following output:


```
cout.width(5);
cout << 543;
cout.width(5);
cout << 12  << "\n";
```

This produces the following output:



precision()

cout.precision(d);

Note: Default precision is 6 digits

```
cout.precision(3);
cout << sqrt(2) << "\n";
cout << 3.14159 << "\n";
cout << 2.50032 << "\n";
```

produce the following output:

```
1.141   (truncated)
3.142   (rounded to the nearest cent)
2.5     (no trailing zeros)
```

```
#include <iostream>
#include <cmath>

using namespace std;

int main()
{
        cout << "Precision set to 3 digits \n\n";
        cout.precision(3);

        cout.width(10);
        cout << "VALUE";
        cout.width(15);
        cout << "SQRT_OF_VALUE" << "\n";

        for(int n=1; n<=5; n++)
        {
                cout.width(8);
                cout << n;
                cout.width(13);
                cout << sqrt(n)  << "\n";
        }
        cout << "\n Precision set to 5 digits \n\n";
        cout.precision(5);              // precision parameter changed
        cout << " sqrt(10) = " << sqrt(10) << "\n\n";

        cout.precision(0);              // precision set to default
        cout << " sqrt(10) = " << sqrt(10) << " (default setting)\n";

        return 0;
}
```

```
Precision set to 3 digits
       VALUE    SQRT_OF_VALUE
           1                1
           2             1.41
           3             1.73
           4                2
           5             2.24

Precision set to 5 digits

sqrt(10) = 3.1623

sqrt(10) = 3.162278 (default setting)
```

fill()

- Unused positions of the field are by default filled with white spaces.
- Any other character can also be filled in the unused positions as:
  - cout.fill(ch);
  - *ch* represents the character used for filling unused space.

```
cout.fill('*');
cout.width(10);
cout << 5250  << "\n";
```

The output would be:

```cpp
#include <iostream>

using namespace std;

int main( )
{       cout.fill('<');

        cout.precision(3);
        for(int n=1; n<=6; n++)
        {
            cout.width(5);
            cout << n;
            cout.width(10);
            cout << 1.0 / float(n) << "\n";
            if (n == 3)
               cout.fill ('>');
        }
        cout << "\nPadding changed \n\n";
        cout.fill ('#');     // fill( ) reset
        cout.width (15);
        cout << 12.345678    << "\n";

        return 0;
}
```

The output of Program

```
<<<<1<<<<<<<<<1
<<<<2<<<<<<<0.5
<<<<3<<<<<0.333
>>>>4>>>>>>0.25
>>>>5>>>>>>>0.2
>>>>6>>>>>0.167

Padding changed

########12.346
```

setf()

- It is a member function of ios class.
- Provides various formatting options.
- setf() stands for set flags

- General form:

  cout.setf (*arg1*, *arg2*)
  - arg1- formatting flag, specifies the format action required for output.
  - arg2- bit field, specifies the group to which formatting flag belongs.
- E.g.

```
cout.setf(ios::left, ios::adjustfield);
cout.setf(ios::scientific, ios::floatfield);
```

Example:

```
cout.fill('*');
cout.setf(ios::left, ios::adjustfield);
cout.width(15);
cout << "TABLE 1" <<  "\n";
```

This will produce the following output:

T	A	B	L	E		1	*	*	*	*	*	*	*	*

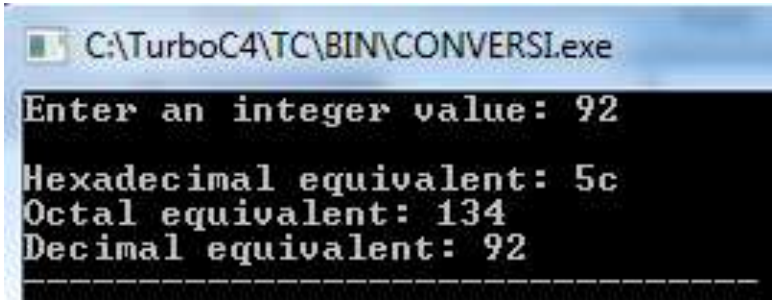
```
cout.fill ('*');
cout.precision(3);
cout.setf(ios::internal, ios::adjustfield);
cout.setf(ios::scientific, ios::floatfield);
cout.width(15);

cout << -12.34567  <<'"\n";
```

produce the following output:

-	*	*	*	*	*	1	·	2	3	5	e	+	0	1

Manipulation of Flag and Bit fields

```
#include<iostream.h>
void main()
{
     int num;
     clrscr();
     cout<<"Enter an integer value: ";
     cin>>num;
     cout<<"\nHexadecimal equivalent: ";
     cout.setf(ios::hex, ios::basefield);
     cout<<num;
     cout<<"\nOctal equivalent: ";
     cout.setf(ios::oct, ios::basefield);
     cout<<num;
     cout<<"\nDecimal equivalent: ";
     cout.setf(ios::dec, ios::basefield);
     cout<<num;
}
```



Displaying Trailing Zeros and Plus Sign

- cout.setf(ios::showpoint); //display trailing zeros
- cout.setf(ios::showpos); //show + sign
- In above functions, setf() is used with single argument.

- E.g.

```
cout.setf(ios::showpoint);
cout.setf(ios::showpos);
cout.precision(3);
cout.setf(ios::fixed, ios::floatfield);
cout.setf(ios::internal, ios::adjustfield);
cout.width(10);
cout << 275.5 << "\n";
```

produce the following output:

+			2	7	5	·	5	0	0

Flags that do not have bit fields

Flag	Meaning
ios :: showbase	Use base indicator on output
ios :: showpos	Print + before positive numbers
ios :: showpoint	Show trailing decimal point and zeroes
ios :: uppercase	Use uppercase letters for hex output
ios :: skipus	Skip white space on input
ios :: unitbuf	Flush all streams after insertion
ios :: stdio	Flush **stdout** and **stderr** after insertion

Formatting with Manipulators

- *iomanip.h*- header file to be used to manipulate the output formats.
- Provides same features as that of **ios** member functions and flags.

```
cout << manip1 << manip2 << manip3 << item;
cout << manip1 << item1  << manip2 << item2;
```

- Concatenation can be done in order to display several columns of output.

Manipulators

Manipulator	Meaning	Equivalent
setw (int w)	Set the field width to w	width()
setprecision (int d)	Set the floating point precision to d	precision()
setfill (int c)	Set the fill character to c	fill()
setiosflags (long f)	Set the format flag f	setf()
resetiosflags (long f)	Clear the flag specified by f	unsetf()
endl	Insert the new line and flush stream	“\n”

Difference between ios class and manipulators

ios class	Manipulators
• ios functions returns values	• Manipulators does not return value.
• ios functions cannot be created.	• Own manipulators can be created
• Can be applied single and cannot be concatenated	• Can be concatenated.
• Includes **iostream.h**	• Includes **iomanip.h**

Designing own manipulators

- Manipulators can also be designed for special purposes.
- General Form:

```
ostream & manipulator (ostream & output)
{
    .....
    .....        (code)
    .....
    return output;
}
```

- Here, *manipulator*– name of manipulator under construction

Example

```
ostream & unit (ostream & output)
{
    output<<" inches";
    return output;
}
```

```
cout << 36 << unit;
```

produce the following output

```
36 inches
```

USER-DEFINED MANIPULATORS

```
#include <iostream>
#include <iomanip>

using namespace std;

// user-defined manipulators

ostream & currency(ostream & output)
{
        output << "Rs";
        return output;
}

ostream & form(ostream & output)
{
        output.setf(ios::showpos);
        output.setf(ios::showpoint);
        output.fill('*');
        output.precision(2);
        output << setiosflags(ios::fixed)
               << setw(10);
        return output;
}

int main()
{
        cout << currency << form << 7864.5;

        return 0;
}
```

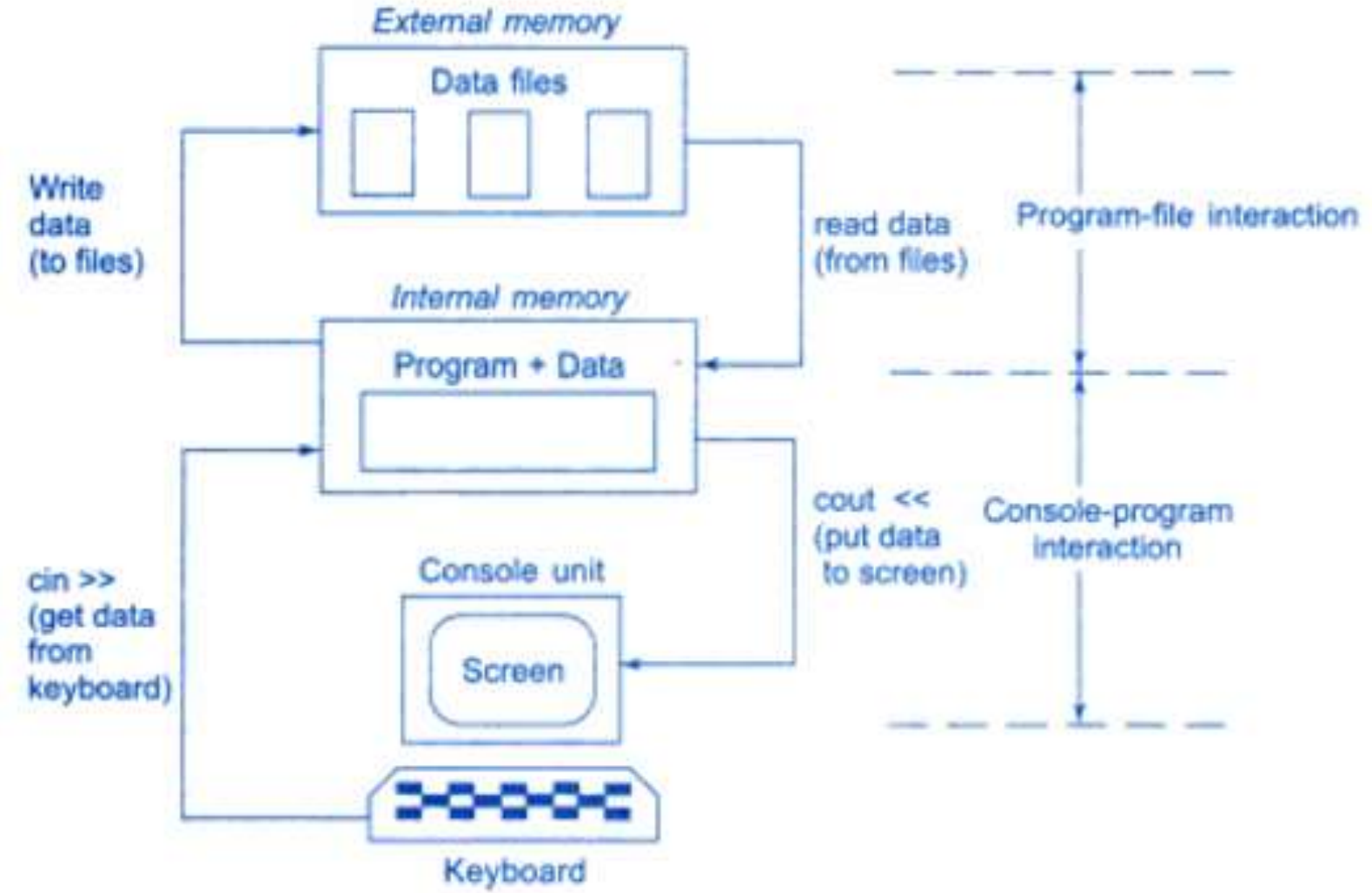
The output of Program

```
Rs**+7864.50
```

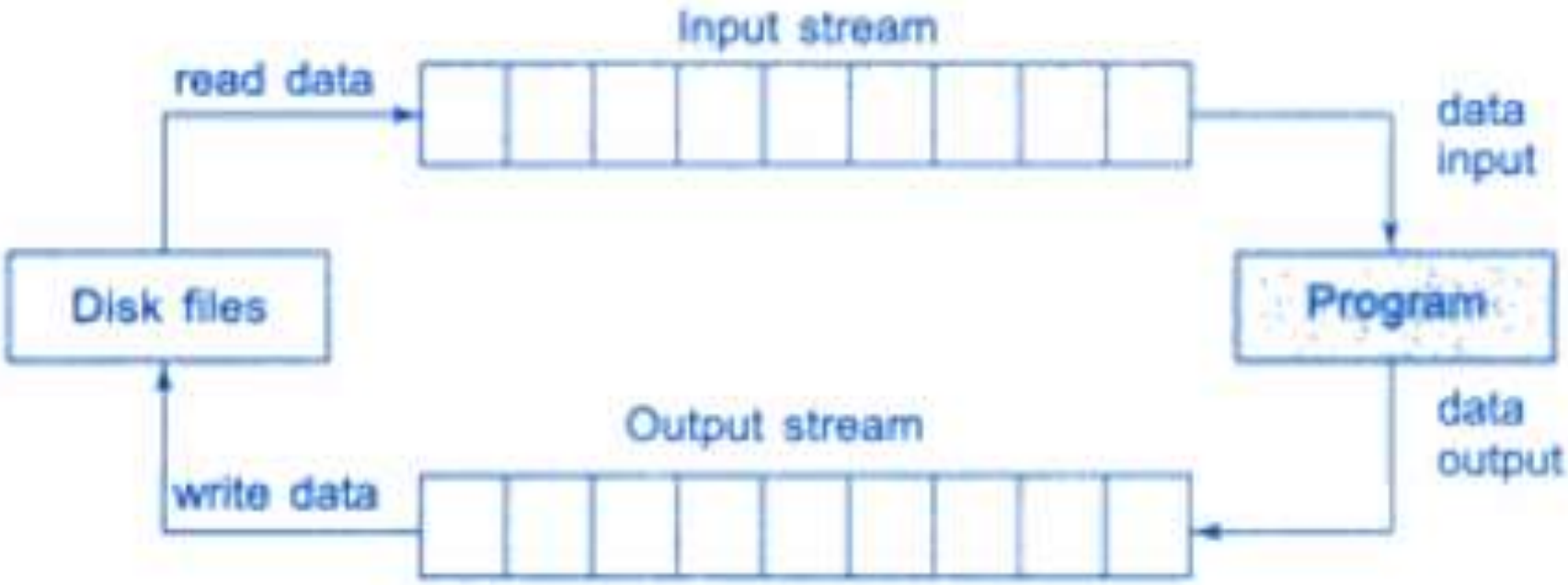
Files

- A file is a collection of related data stored in a particular area on the disk.
- These files can be used by the programs to performs read and write operations:
  - to transfer data between the console unit and the program
  - to transfer between program and disk file

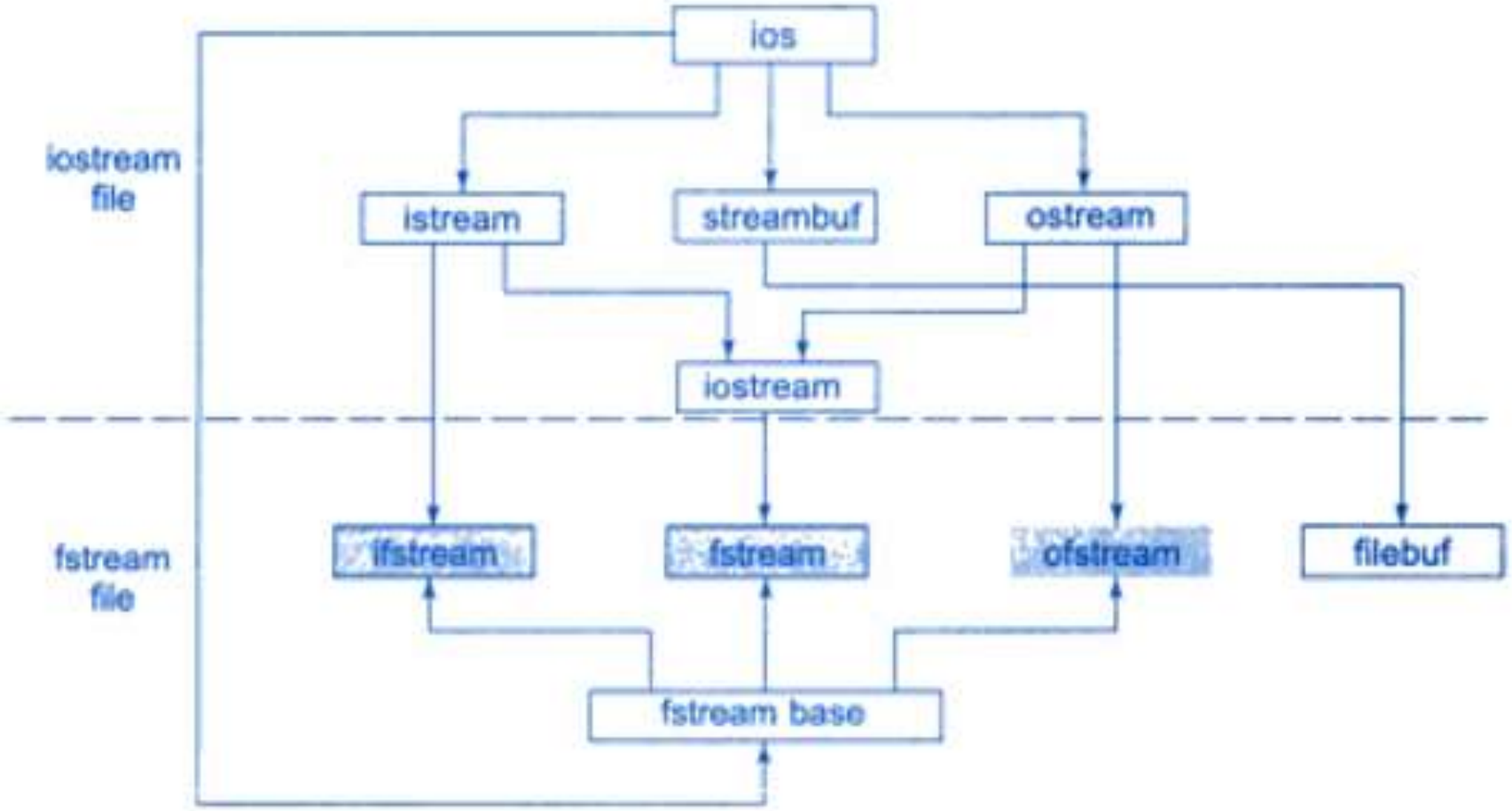
Console-Program-File Interaction



File I/O Streams



Classes for file stream operations



File stream classes

- Filebuf
  - To set the file buffers to read and write.
  - Contain **open()** and **close()** as members.

- Fstreambase
  - Provides operations common to file streams.
  - Acts as a base class for **fstream**, **ifstream** and **ofstream** class.
  - Contains **open()** and **close()** functions.

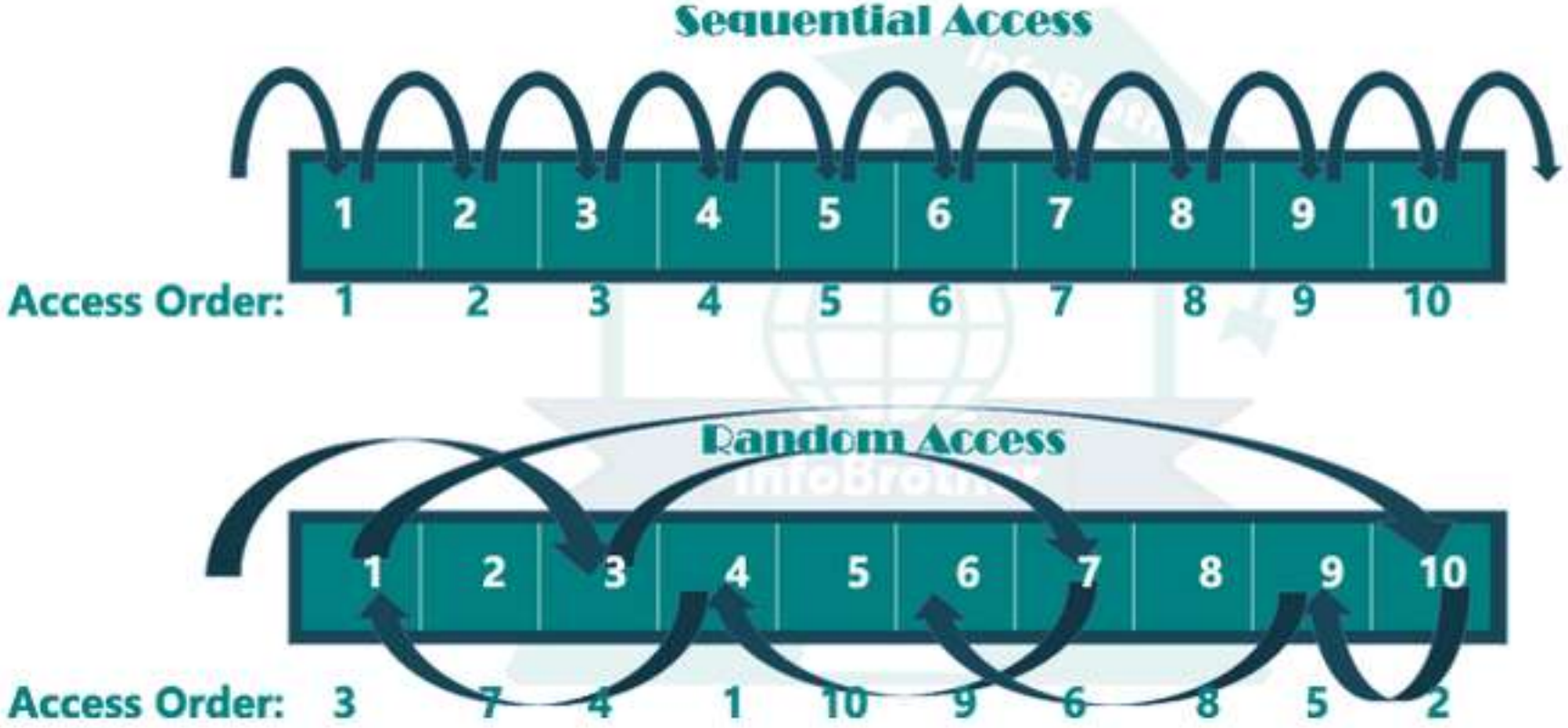
- if stream
  - Provides input operations.
  - Contains **open()** with default input mode.
  - Inherits **get()**, **getline()**, **read()**, **seekg()** and **tellg()** functions from **istream**.

- Ofstream
  - Provides output operations.
  - Contains **open()** with default output mode. Inherits **put()**, **seekp()**, **tellp()** and **write()** functions from **ostream**.

- Fstream
  - Provides support for simultaneous input and output operations.
  - Inherits all the functions from **istream** and **ostream** classes through **iostream**

Accessing Files

- A program can access files in two ways:
  - Sequential Access
    - It has to be accessed in the same order as file was written i.e. sequentially.
    - It is not possible to add or delete any record from the middle of a sequential file
    - For this, a new file is to be created containing both new and old records.
  - Random Access
    - Random access file enables us to read and write any data in our disc file in any order.


Sequential Access
1 2 3 4 5 6 7 8 9 10
Access Order: 1 2 3 4 5 6 7 8 9 10
Random Access
1 2 3 4 5 6 7 8 9 10
Access Order: 3 7 4 1 10 9 6 8 5 2

Opening and Closing a File

- If a disk file is to be used, following information is needed:
  - Suitable name for the file.
  - Data type and structure
  - Purpose
  - Opening method

- Filename is a string of characters that make a valid filename.
- It may contain one or two parts:
  - A primary name
  - Optional period(.) with extension.
  - E.g.
    - input.data
    - test.doc
    - student
    - employee

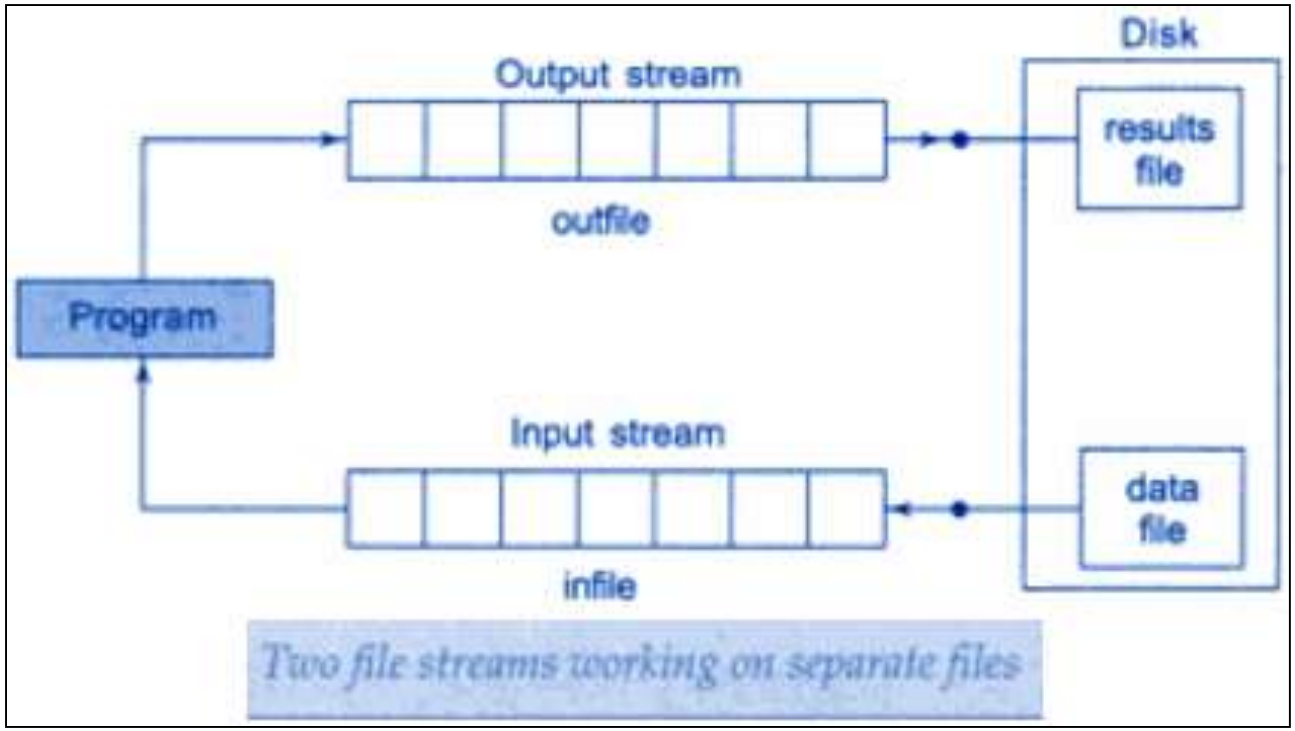
- Initially a file stream is created and linked to the filename.

- Either **ifstream**, **ofstream** or **fstream** files can be included as header file.

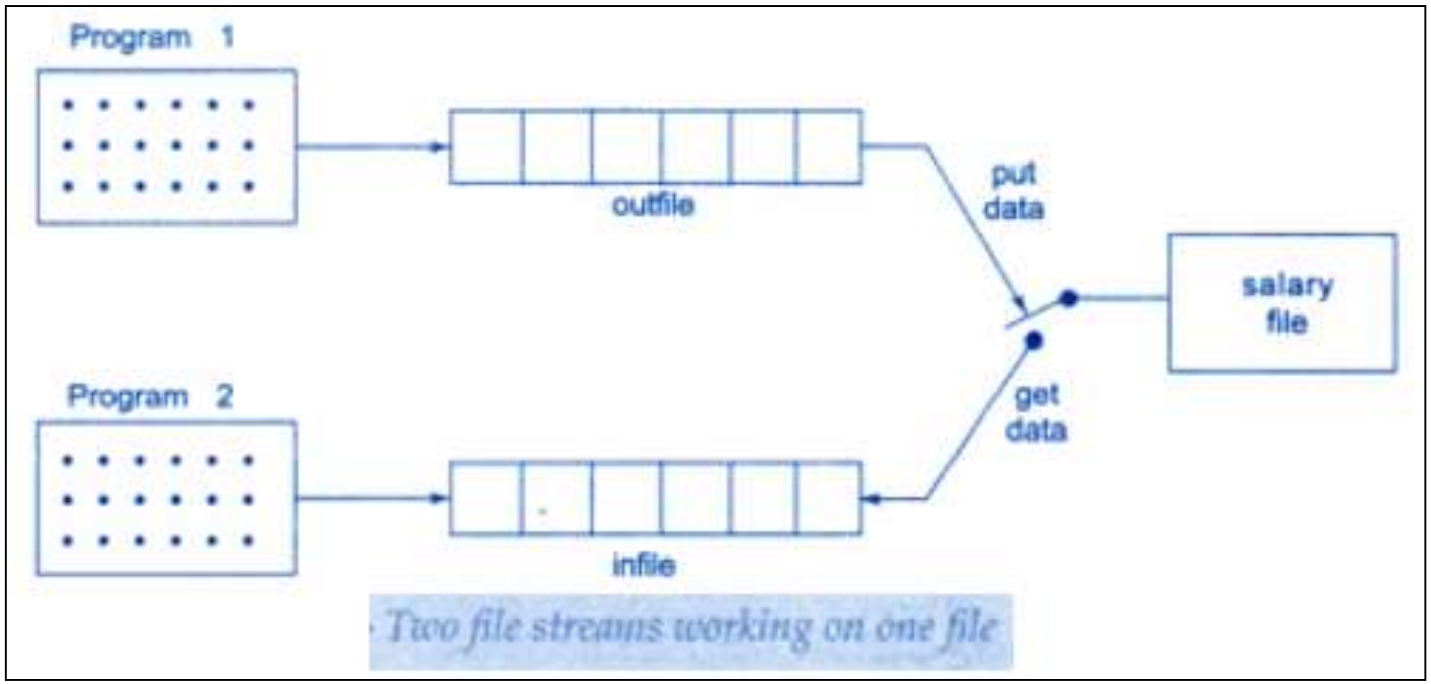
- File can be opened in two ways:
  - Using constructor function of the class.
    - If only one file in stream is used.
  - Using member function **open()** of the class.
    - If multiple files need to be managed using one stream.

Opening Files using Constructors

- Following are the steps to initialize the file stream object:
  - Create a file stream object to manage the stream using appropriate class.
  - Initialize the file object with desired filename.
- E.g.
  - ofstream outfile (“results”);
  - Where,
    - *outfile* is the name of object of class ofstream
    - *results* is the name of file to be opened



Two file streams working on separate files



Two file streams working on one file

WORKING WITH SINGLE FILE

```
// Creating files with constructor function

#include <iostream.h>
#include <fstream.h>

int main()
{
        ofstream outf("ITEM");          // connect ITEM file to outf

        cout << "Enter item name:";
        char name[30];
        cin >> name;                    // get name from key board and

        outf << name << "\n";           // write to file ITEM

        cout << "Enter item cost:";
        float cost;
        cin >> cost;                    // get cost from key board and

        outf << cost << "\n";           // write to file ITEM

        outf.close();                   // Disconnect ITEM file from outf

        ifstream inf("ITEM");           // connect ITEM file to inf

        inf >>  name;                   // read name from file ITEM
        inf >> cost;                    // read cost from file ITEM

        cout << "\n";
        cout << "Item name:" << name << "\n";
        cout << "Item cost:" << cost << "\n";

        inf.close();                    // Disconnect ITEM from inf

        return 0;
}
```

```
Enter item name:CD-ROM
Enter item cost:250

Item name:CD-ROM
Item cost:250
```

Opening files using open()

- The function open() can be used to open multiple files using same stream object.
- The files are processed sequentially.

- General form:

  file-stream-class stream-object;

  stream-object.open (“filename”);

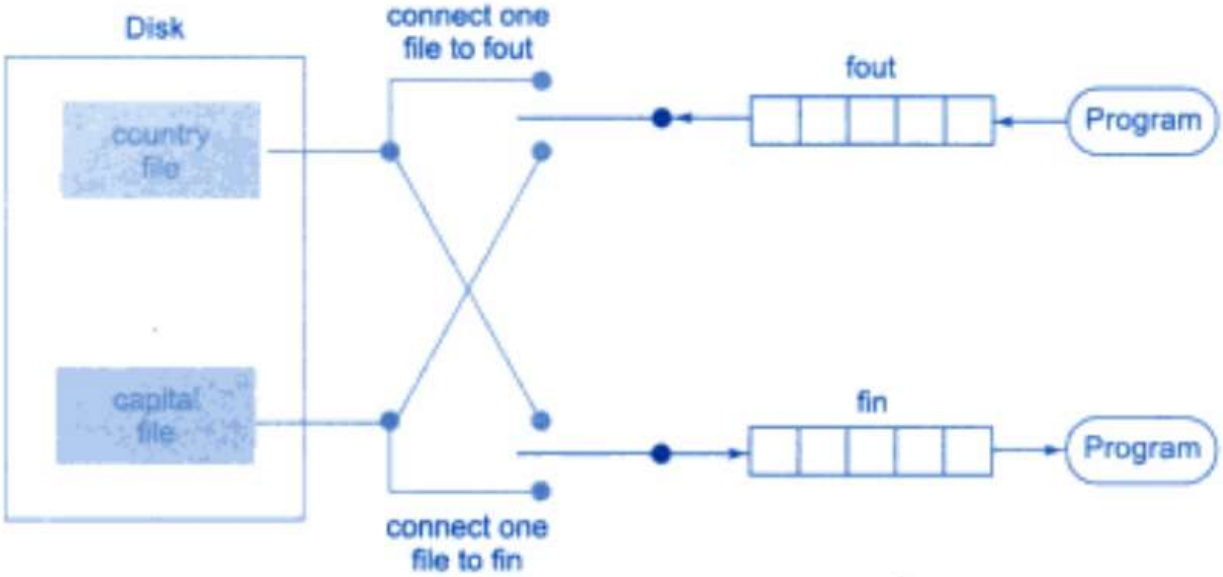
- While opening multiple files, one files must be closed before opening another file.

Example:

```
ofstream outfile;           // Create stream (for output)
outfile.open("DATA1");      // Connect stream to DATA1
.....
.....
outfile.close();            // Disconnect stream from DATA1
outfile.open("DATA2);       // Connect stream to DATA2
.....
.....
outfile.close();            // Disconnect stream from DATA2
.....
.....
```

Reading multiple files simultaneously

- To read/write two or more files at the same time, separate input and output streams for handling input files and output files need to be created.



Streams working on multiple files

READING FROM TWO FILES SIMULTANEOUSLY

```
// Reads the files created in Program 11.2

#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>              // for exit() function

int main()
{
        const  int SIZE = 80;
        char line[SIZE];

        ifstream fin1, fin2;          // create two input streams
        fin1.open("country");
        fin2.open("capital");

        for(int i=1; i<=10; i++)
        {
              if(fin1.eof() != 0)
              {
                    cout << "Exit from country \n";
                    exit(1);
              }
        fin1.getline(line, SIZE);
        cout << "Capital of "<< line ;

        if(fin2.eof() != 0)
        {
              cout << "Exit from capital\n";
              exit(1);
        }

        fin2.getline(line,SIZE);
        cout << line << "\n";
  }
  return 0;
}
```

The output of Program

```
Capital of United States of America
Washington
Capital of United Kingdom
London
Capital of South Korea
Seoul
```

Detecting End-of-File

- Detecting end-of-file is required to prevent any further attempt to read data from file.
- Can be done by using eof().
  - eof() is a member function of ios class.

- General form:
  - if(fin.eof()!=0) {exit(1);}
    - Terminates the program on reaching EOF.
    - Returns a non zero value if end-of-file(EOF) condition is encountered.
    - Returns zero if error

Open(): File Modes

- If ifstream and ofstream classes are used to open files, we used only one argument that was the filename.
- Thus the files can be opened in default mode.
- Two arguments can be specified, second one for the file mode.
- General form:
- Stream-object.open(“filename”, mode);

File Mode Parameters

File Mode Parameter	Meaning	Explanation
ios::in	Read	Open the file for Reading Purpose Only: if the file don't exist, it will generate an error.
ios::out	Write	Open the file for Writing Purpose Only: if the file already exist, then this mode will open the file and format it. and if there is no file exist, then this mode will create New file.
ios::app	Appending	Open the file for Appending data to end of the file. Using this mode we can Open an already existing file in this mode we can write only at the end of the file.
ios::ate	Appending	It's same like ios::app mode. Open file using this mode will take us to end of the file. but we can write any where in file.
ios::binary	Binary	Open file in Binary mode:
ios::trunc	Truncate/Discard	When we will open the file using this Mode. this mode will delete all data if the file already existing.
ios::nocreate	Don't Create	This Mode can just open an already existing file. if the file not exist, it will show an error.
ios::noreplace	Don't Replace	This mode is used to open new file. if the file is already exist, it will show an error.

File Mode Parameters

- Opening a file in **ios::out** mode opens it in **ios::trunc** mode by default.

- **ios::app** and **ios::ate**
  - both allow take us to the end of file when it is opened.
  - File is created if it does exist.
  - **ios::app** allow us to add data only at the end.
  - **ios::ate** permits us to add data or modify existing data anywhere in the file.

- **ifstream** and **ofstream**
  - does not require mode parameters.
  - implies input and output mode by default.

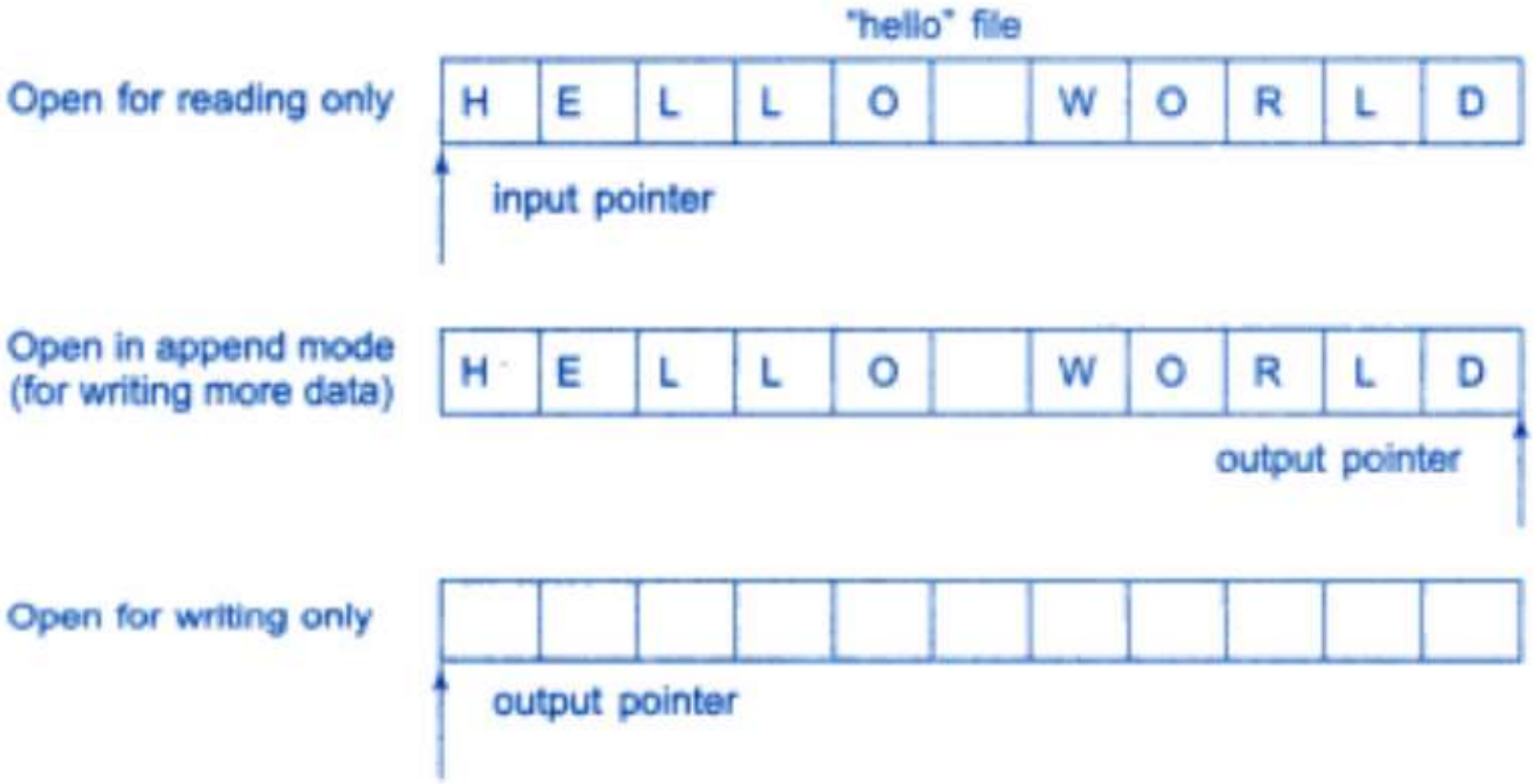
- **fstream**
  - does not provide default mode.
  - Mode must be provided explicitly when using object of fstream class.

- Two modes can also be combined using bitwise OR operator as:
  - **Fout.open("data", ios::app|ios::create)**

File Pointers Manipulations

- Each file has two associated pointers.
- One is the input pointer and other is the output pointer.
- These pointers are moved through the files while reading or writing.

Actions on file pointers while opening



Functions for File Pointer Manipulations

- seekg()
  - Moves get pointer (input) to a specified location
- seekp()
  - Moves pointer (output) to a specified location.
- tellg()
  - Gives current position of get pointer.
- tellp()
  - Gives current position of put pointer.

Consider the following statements:

```
ofstream  fileout;
fileout.open("hello", ios::app);
int p = fileout.tellp();
```

- On execution, output pointer is moved to the end of file “hello”.
- Value of **p** represents the number of bytes in the file.

- Seek function can also be used with two arguments as:
  - Seekg (offset, refposition);
  - Seekp (offset, refposition);
  - Where,
    - *Offset* represents the number of bytes the files pointer is to be moved.
    - *Refposition* specifies the location of file pointer.

  - Refposition takes one of the following constants defined in ios class.

ios::beg	start of the file
ios::cur	current position of the pointer
ios::end	End of the file

Pointer Offset calls

Seek Call	Action
fout.seekg (0, ios::beg);	Go to start
fout.seekg (0, ios::cur);	Stay at the current position
fout.seekg (0, ios::end);	Go to the end of the file
fout.seekg (m, ios::beg);	Move to (m+1)th byte in the file
fout.seekg (m, ios::cur);	Go forward by m byte 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

Sequential Input/ Output Operations

- get()
- put()

```
#include <iostream.h>
#include <fstream.h>
#include <string.h>

int main()
{
        char string[80];

        cout<<"Enter a string: ";
        cin>>string;

        int len=strlen(string);

        fstream file;                    //input and output stream
        cout<<"\nOpening the 'TEXT' file and storing the string in
           it.\n\n";

        file.open("TEXT", ios::in | ios::out);

        for(int i=0;i<len;i++)
           file.put(string[i]);          //put a character to file

        file.seekg(0);                   //go to the start

        char ch;
        cout<<"Reading the file contents:  ";
        while(file)
        {
          file.get(ch);                  //get a character from file
          cout<<ch;                      //display it on screen
        }

        return 0;
}
```

Binary Files

- A binary file is stored in a binary format.
- It is computer readable but not human readable.
- Executable files are in binary format.
- At the time of opening the file, ios::binary mode is used with other modes.
- Binary format is more accurate for storing the numbers as they are stored in exact internal representation.

write() and read() functions

- Handles the data in binary form
- Values are stored in the same format in which they are stored in internal memory.

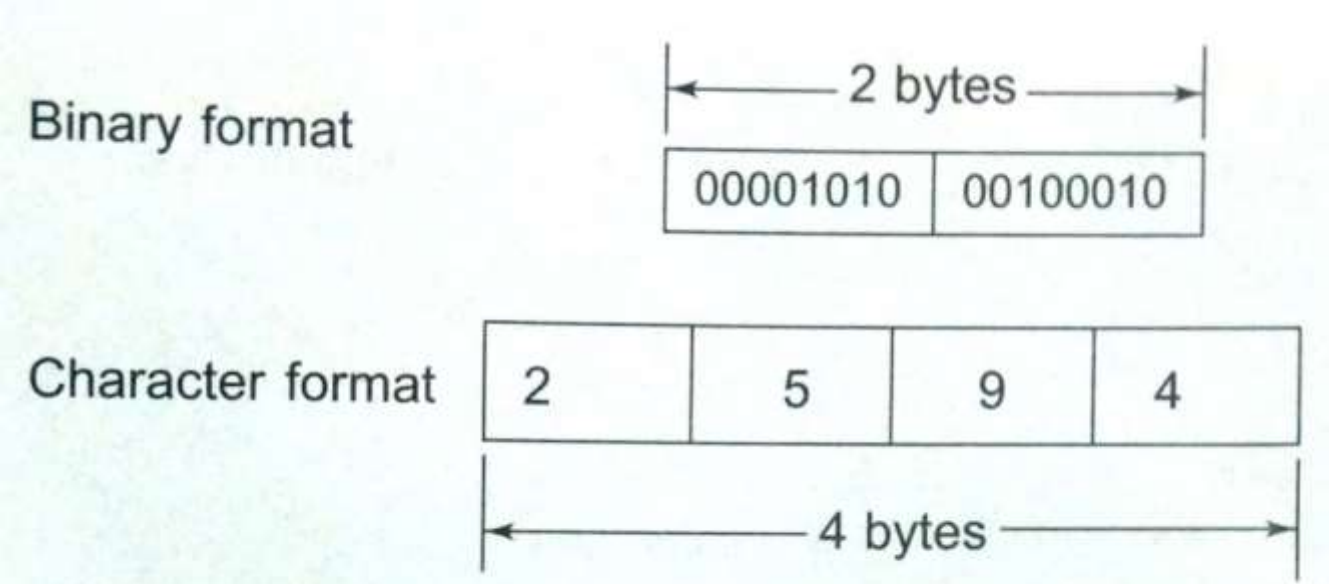


Fig. 11.9 *Binary and character formats of an integer value*

Error Handling during File Operations

- While dealing with files, following things may happen:
  - File which we are attempting to open does not exist.
  - File name used for a new file may already exist.
  - Attempting invalid operation i.e. reading past the end of file.
  - There may not be any space in the disk for storing more data.
  - Invalid file name
  - We may attempt to perform an operation while the file is not opened for that purpose.

- The class ios supports several member functions that can be used to read the status recorded in a file stream.
- These function are as follows:

Function	Return value and meaning
eof()	Returns (True) non zero value if end of file is encountered, other returns zero (False)
fail()	Returns True if input or output operation has failed
bad()	Returns True if an invalid operation is attempted or any unrecoverable error has occurred
good()	Returns True if no error has occurred. When it returns False, no further operations can be carried out.

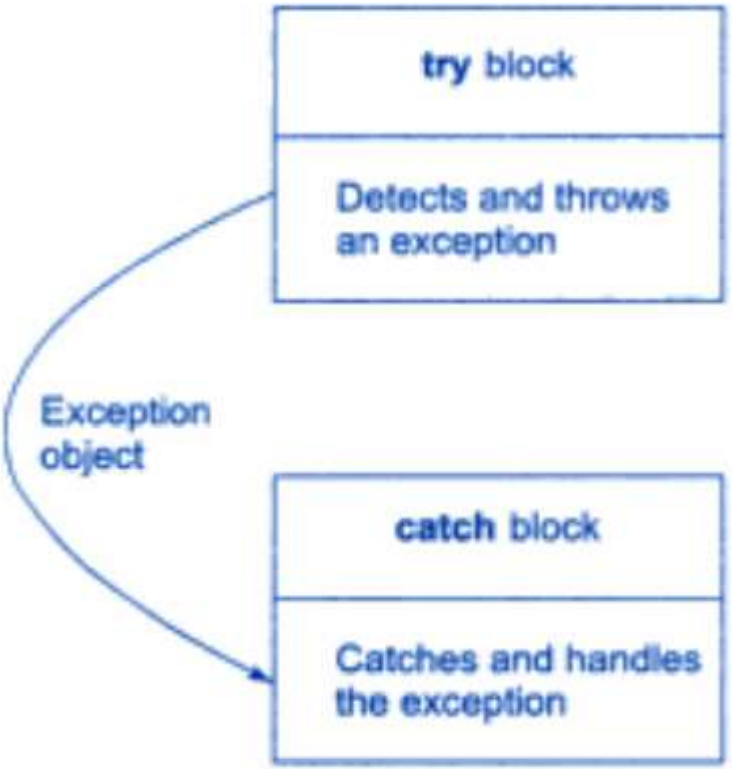
EXCEPTION HANDLING

Exception Handling

- Exceptions are the run time anomalies or unusual conditions that a program may encounter while executing.
- Exceptions are of two kinds:
  - Synchronous exceptions
    - Includes error such as:
    - out-of-range index
    - Over-flow
  - Asynchronous exceptions
    - Caused by events beyond the control of program.
- Exception handling mechanism handles only synchronous exceptions.

Exception Handling Mechanism

- Find the problem
  - (**Hit** the exception)
- Inform the error
  - (**Throw** the exception)
- Receive the error information
  - (**Catch** the exception)
- Take corrective actions
  - (**Handle** the exception).



NOTE: catch block must immediately follow the **try** block that **throws** the exception.

General Form

```
.....
.....
try
{
        .....
        throw exception;          // Block of statements which
        .....                     // detects and throws an exception
        .....
}
catch(type arg)                   // Catches exception
{
        .....
        .....                     // Block of statements that
        .....                     // handles the exception
        .....
}
.....
.....
```

```
#include <iostream>

using namespace std;
int main()
{
    int a,b;
    cout << "Enter Values of a and b \n";
    cin >> a;
    cin >> b;
    int x = a-b;
    try
    {
        if(x != 0)
        {
            cout << "Result(a/x) = " << a/x << "\n";
        }
        else                    // There is an exception
        {
            throw(x);   // Throws int object
        }
    }
    catch(int i)                // Catches the exception
    {
        cout << "Exception caught: x = " << x << "\n";
    }

    cout << "END";

    return 0;
}
```

The output of Program

First Run

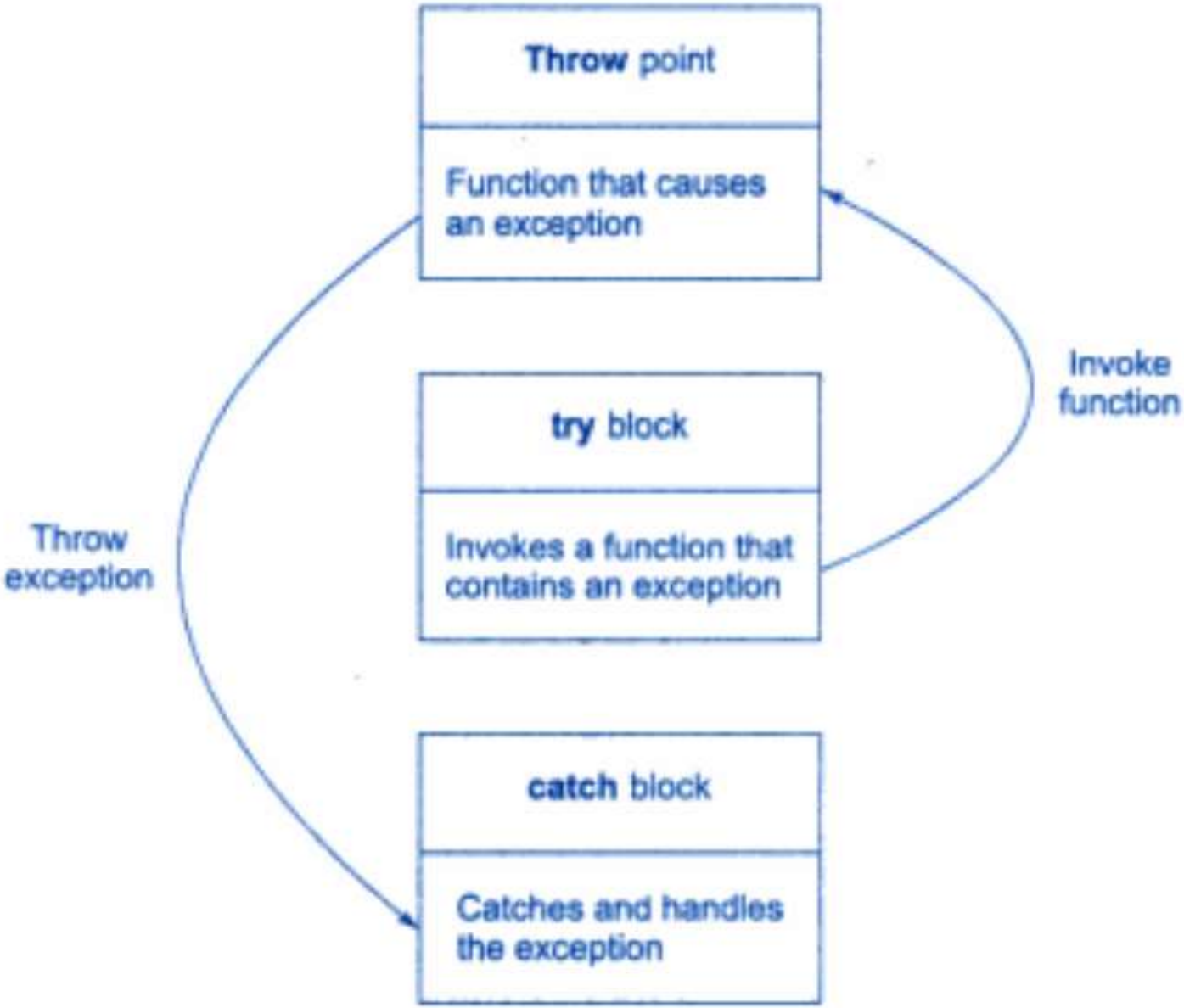
```
Enter Values of a and b
20 15
Result(a/x) = 4
END
```

Second Run

```
Enter Values of a and b
10 10
Exception caught: x = 0
END
```

Invoking function in try block

- Exception can also be thrown by functions that are invoked from within the try block.
- The point where **throw** is executed is called **throw point**.
- Once an exception is thrown to the catch block, control cannot return to the throw point.


Throw point
Function that causes an exception
try block
Invokes a function that contains an exception
catch block
Catches and handles the exception
Invoke function
Throw exception

General Form

```
type function(arg list)      // Function with exception
{
        ......
        ......
        throw(object);       // Throws exception
        ......
        ......
}
......
......
try
{
        ......
        ...... Invoke function here
        ......
}
catch(type arg)              // Catches exception
{
        ......
        ...... Handles exception here
        ......
}
......
```

```
#include <iostream>

using namespace std;

void divide(int x, int y, int z)
{
        cout << "\nWe are inside the function \n";
        if((x-y) != 0)          // It is OK
        {
              int R = z/(x-y);
              cout << "Result = " << R << "\n";
        }
        else                    // There is a problem
        {
              throw(x-y);       // Throw point
        }
}

int main()
{
        try
        {
              cout << "We are inside the try block \n";
              divide(10,20,30);   // Invoke divide()
              divide(10,10,20);   // Invoke divide()
        }
        catch(int i)      // Catches the exception
        {
              cout << "Caught the exception \n";
        }
        return 0;
}
```

Throwing Mechanism

- An exception can be thrown using one of the following forms:
  - throw (exception);
  - throw exception;
  - throw;
- *exception* may be of any type, including constants also.

Catching Mechanism

- To handle the exception, code is included in **catch** block.
- Its general form is same as that of function definition.

```
Catch (type arg)
{
  //statements for managing exception
}
```

  - *Type* indicates the type of exception that catch block handles
  - Parameter *arg* is an optional parameter name
- Catch statement catches an exception whose type matched with the type of catch argument.
- After execution, control goes to the statement immediately following the catch block.
- If exception is not caught, program will be terminated abnormally.

Multiple Catch Statements

```
try
{
     // try block
}
catch(type1 arg)
{
     // catch block1
}
catch(type2 arg)
{
     // catch block2
}
......
......
catch(typeN arg)
{
     // catch blockN
}
```

- When an exception is thrown, exception handlers are searched in order.

- Only one handler is executed even if multiple catch statements match the type.

- Then the control goes to the first statement following the catch block.

```
#include <iostream>
using namespace std;
void test(int x)
{
        try
        {
            if(x == 1) throw x;          // int
            else
                if(x == 0) throw 'x';    // char
            else
                if(x == -1) throw 1.0;   // double
            cout << "End of try-block \n";
        }
        catch(char c)                    // Catch 1
        {
            cout << "Caught a character \n";
        }
        catch(int m)                     // Catch 2
        {
            cout << "Caught an integer \n";
        }
        catch(double d)                  // Catch 3
        {
            cout << "Caught a double \n";
        }
        cout << "End of try-catch system \n\n";
}
int main()
{
        cout << "Testing Multiple Catches \n";
        cout << "x == 1 \n";
        test(1);
        cout << "x == 0 \n";
        test(0);
        cout << "x == -1 \n";
        test(-1);
        cout << "x == 2 \n";
        test(2);
        return 0;
}
```

Catch All Exceptions

- A catch statement can be used to catch all exceptions instead of a certain type alone using ellipses as:

```
Catch(…)
{
  //statements for processing all exceptions
}
```

- Catch(…) can be placed last in the list of handlers as a default statement to catch all those exceptions which are not handled explicitly.

- If placed before, prevent all other blocks from catching exceptions.

```cpp
#include <iostream>

using namespace std;

void test(int x)
{
    try
    {
        if(x==0) throw x;
        if(x==-1) throw 'x';
        if(x==1) throw 1.0;
    }
    catch(...)
    {
        cout<<"\nCaught an exception";
    }
    cout<<"\nExits Catch Block";
}
int main()
{
    cout<<"Testing Generic Catch";
    test(-1);
    test(0);
    test(1);
    test(2);
    return 0;
}
```

```
Testing Generic Catch
Caught an exception
Exits Catch Block
Caught an exception
Exits Catch Block
Caught an exception
Exits Catch Block
Exits Catch Block
```

Re-throwing an Exception

- Sometimes an exception handler may pass the exception to another outer try catch block from the current catch block without processing.
- This can be done by using:

```
throw;          //rethrowing an exception without using any argument
```

- This is known as **Re-throwing and exception**.

```
void divide(double x, double y)
{
        cout << "Inside function \n";
        try
        {
                if(y == 0.0)
                        throw y;        // Throwing double
                else
                        cout << "Division = " << x/y << "\n";
        }
        catch(double)                   // Catch a double
        {
                cout << "Caught double inside function \n";
                throw;                  // Rethrowing double
        }
        cout << "End of function \n\n";
}

int main()
{
        cout << "Inside main \n";
        try
        {
                divide(10.5,2.0);
                divide(20.0,0.0);
        }
        catch(double)
        {
                cout << "Caught double inside main \n";
        }
        cout << "End of main \n";

        return 0;
}
```

The output of the Program

```
Inside main
Inside function
Division = 5.25
End of function

Inside function
Caught double inside function
Caught double inside main
End of main
```

Specifying Exceptions

- A function can be restricted to throw only certain specified exceptions.

- a throw list can be added to the function definition.
  - ***Throw-list/ type-list*** specifies the type of exceptions that may be thrown
  - General Form:

```
type function(arg-list) throw (type-list)
{
        ......
        ...... Function body
        ......
}
```

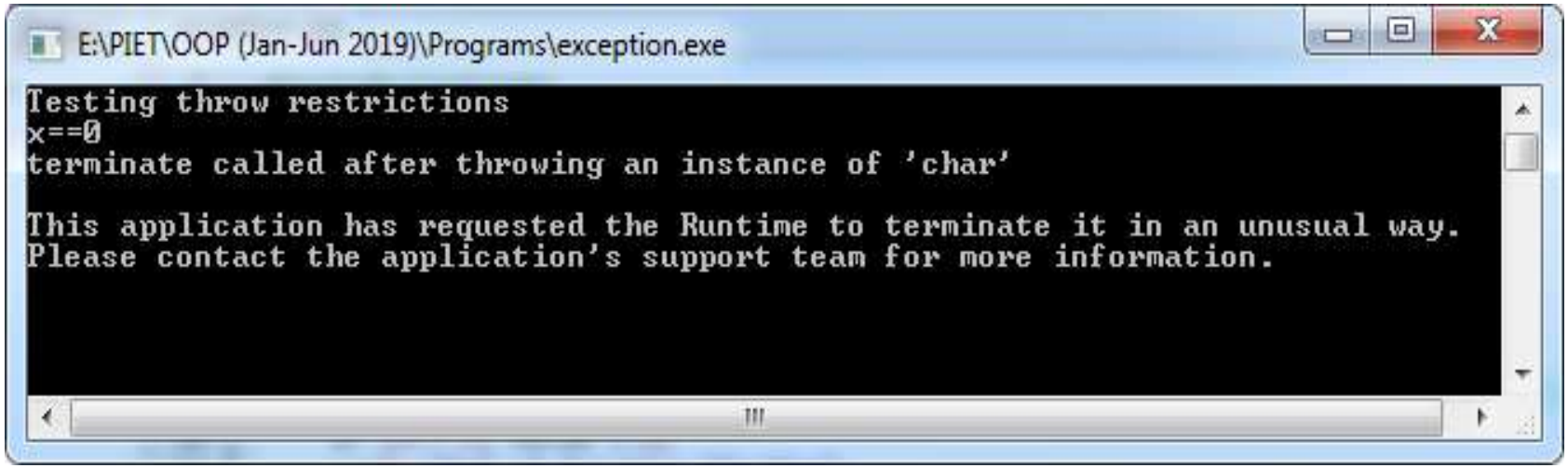
- If type does not match with the type-list, program is terminated abnormally.

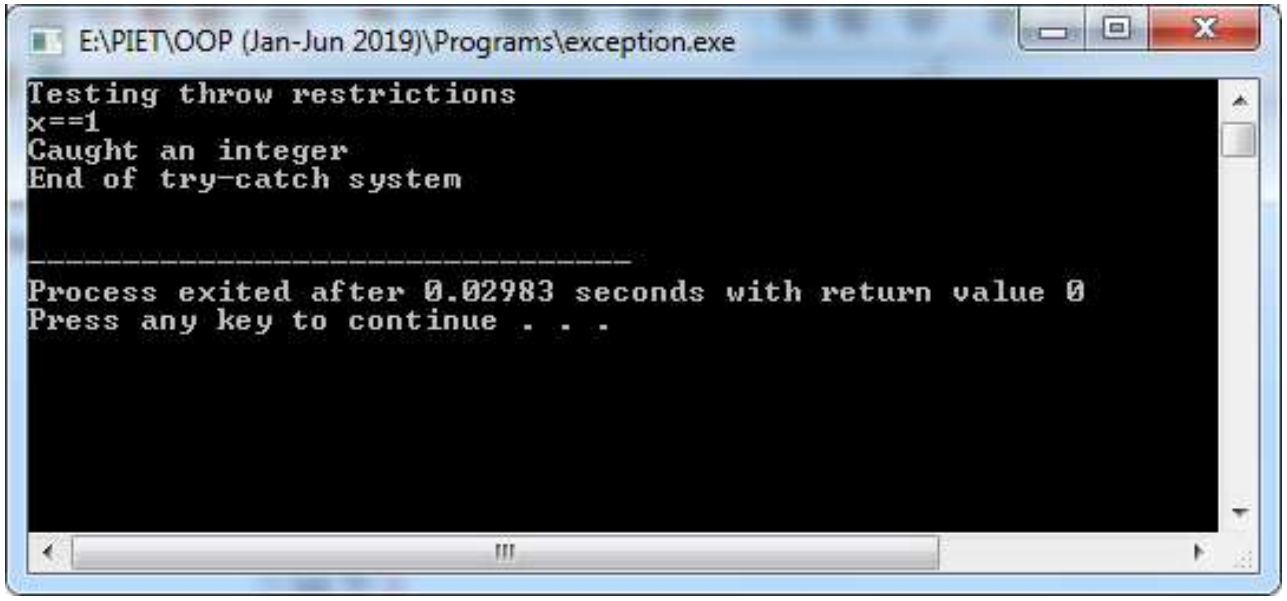
- A function can also be prevented from throwing any exception by adding empty type-list as:

  throw();

```cpp
#include<iostream>
using namespace std;
void test(int x) throw(int,double)

{
    if (x==0) throw 'x';
    else
    if (x==1) throw x;
    else
    if (x==-1) throw 1.0;
    cout<<"End of function block\n";
}
int main()
{
    try
    {
        cout<<"Testing throw restrictions\n";
        //cout<<"x==0\n";
        //test(0);
        cout<<"x==1\n";
        test(1);
        cout<<"x==-1\n";
        test(-1);
        cout<<"x==2\n";
        test(2);
    }
    catch(char c)
    {
        cout<<"Caught a character\n";
    }
    catch(int m)
    {
        cout<<"Caught an integer\n";
    }
    catch(double d)
    {
        cout<<"Caught a double\n";
    }
    cout<<"End of try-catch system\n\n";
    return 0;
}
```

Output


E:\PIET\OOP (Jan-Jun 2019)\Programs\exception.exe
Testing throw restrictions
x==0
terminate called after throwing an instance of 'char'
This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.


E:\PIET\OOP (Jan-Jun 2019)\Programs\exception.exe
Testing throw restrictions
x==1
Caught an integer
End of try-catch system
Process exited after 0.02983 seconds with return value 0
Press any key to continue . . .

TEMPLATES

Templates

- A template is defined with a parameter that would be replaced by specified data type at the time of actual use of class or function.

- Templates enable us to define generic classes and functions.

- It provides support for generic programming.
  - **Generic Programming** is an approach where generic types are used as parameters in algorithms so that they can work for a variety of suitable data types and data structures.

- Can be used to create a family of classes or functions.

Class Templates

```
class vector
{
       int *v;
       int size;
    public:
      vector(int m)              // create a null vector
      {
           v = new int[size = m];
           for(int i=0; i<size; i++)
                v[i] = 0;
      }
      vector(int *a)             // create a vector from an array
      {
           for(int i=0; i<size; i++)
               v[i] = a[i];
      }
      int operator*(vector &y)        // scalar product
      {
      int sum = 0;
      for(int i=0; i<size; i++)
           sum += this -> v[i] * y . v[i];
      return sum;
      }
};
```

```
int main()
{
      int x[3] = {1,2,3};
      int y[3] = {4,5,6};
      vector v1(3);
      vector v2(3);
      v1 = x;
      v2 = y;
      int R = v1 * v2;
      cout << "R = " << R;
      return 0;
}
```

General Form of Class Template

```
template<class T>
class classname
{
      // ……
      // class member specification
      // with anonymous type T
      // wherever appropriate
      // ……
};
```

- Class created from class template is called **Template Class**.
- The process of creating a specific class from a class template is called **instantiation.**
- Syntax for defining an object of a template class:

```
classname<type> objectname(arglist);
```

```
#include <iostream>

using namespace std;

const size = 3;

template <class T>
class vector
{
      T* v;          // type T vector
public:
      vector()
      {
            v = new T[size];
            for(int i=0;i<size;i++)
                  v[i] = 0;
      }
      vector(T* a)
      {
            for(int i=0;i<size;i++)
                  v[i] = a[i];
      }
      T operator*(vector &y)
      {
            T sum = 0;
            for(int i=0;i<size;i++)
                  sum += this -> v[i] * y.v[i];
            return sum;
      }
};
```

```
int main()
{
      float x[3] = {1.1,2.2,3.3};
      float y[3] = {4.4,5.5,6.6};
      vector <float> v1;
      vector <float> v2;
      v1 = x;
      v2 = y;
      float R = v1 * v2;
      cout << "R = " << R << "\n";

      return 0;
}
```

The output of the Program

```
R = 38.720001
```

```
int main()
{
      int x[3] = {1,2,3};
      int y[3] = {4,5,6};
      vector <int> v1;
      vector <int> v2;
      v1 = x;
      v2 = y;
      int R = v1 * v2;
      cout << "R = " << R << "\n";
      return 0;
}
```

The output of the Program

```
R = 32
```

Class Templates with Multiple Parameters

General Form:

```
template<class T1, class T2, ...>
class classname
{
   .....
   .....     (Body of the class)
   .....
};
```

TWO GENERIC DATA TYPES IN A CLASS DEFINITION

```
#include <iostream>

using namespace std;

template<class T1, class T2>
class Test
{
        T1 a;
        T2 b;
  public:
        Test(T1 x, T2 y)
        {
                a = x;
                b = y;
        }
        void show()
        {
                cout << a << " and " << b << "\n";
        }
};

int main()
{
        Test <float,int> test1 (1.23,123);
        Test <int,char> test2 (100,'W');

        test1.show();
        test2.show();

        return 0;
};
```

The output of Program

```
1.23 and 123
100 and W
```

Using default data types in Class Definition

```
#include <iostream.h>

using namespace std;

template<class T1=int, class T2=int> //default dta types specified as int
class Test
{
        T1 a;
        T2 b;
     public:Test(T1 x, T2 y)
        {
            a = x;
            b = y;
        }
        void show()
        {
            cout<<a<<" and "<<b<<"\n";
        }
};

int main()
{
        Test <float,int> test1(1.23,123);
        Test <int, char> test2 (100, 'W');
        Test <> test3.('a', 12.983); //declaration of class object
        without types specification
        test1.show();
        test2.show();
        test3.show();
        return 0;
}
```

output of Program

```
1.23 and 123
100 and W
97 and 12
```

Function Templates

- Function templates can also be used to create a family of functions with different types of arguments.
- Syntax for function template is same as that of class template.
- General form:

```
template<class T>
returntype functionname (arguments of type T)
{
        // .....
        // Body of function
        // with type T
        // wherever appropriate
        // .....
}
```

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

template <class T>
void swap (T&x, T&y)
{
     T temp=x;
     x=y;
     y=temp;
}

void fun(int m, int n, float a, float b)
{
     cout<<"Before Swap:"<<m<<"&"<<n<<endl;
     swap(m,n);
     cout<<"After Swap:"<<m<<"&"<<n<<endl;
     cout<<"Before Swap:"<<a<<"&"<<b<<endl;
     swap(a,b);
     cout<<"After Swap:"<<a<<"&"<<b<<endl;
}
int main()
{
     fun(100,200,11.22,33.44)
     return 0;
}
```

The output of Program

```
m and n before swap: 100 200
m and n after swap:  200 100
a and b before swap: 11.22 33.439999
a and b after swap:  33.439999 11.22
```

```
#include <iostream>
#include <iomanip>
#include <cmath>

using namespace std;

template <class T>
void roots(T a,T b,T c)
{
        T d = b*b - 4*a*c;
        if(d == 0)                      // Roots are equal
        {
                cout << "R1 = R2 = " << -b/(2*a) << endl;
        }
        else if(d > 0)                  // Two real roots
        {
                cout << "Roots are real \n";
                float R = sqrt(d);
                float R1 = (-b+R)/(2*a);
                float R2 = (-b-R)/(2*a);
                cout << "R1 = " << R1 << " and ";
                cout << "R2 = " << R2 << endl;
        }
        else                            // Roots are complex
        {
                cout << "Roots are complex \n";
                float R1 = -b/(2*a);
                float R2 = sqrt(-d)/(2*a);
                cout << "Real part = " << R1 << endl;
                cout << "Imaginary part = " << R2;
                cout << endl;
        }
}

int main()
{
        cout << "Integer coefficients \n";
        roots(1,-5,6);
        cout << "\nFloat coefficients \n";
        roots(1.5,3.6,5.0);

        return 0;
}
```

The output of Program

```
Integer coefficients
Roots are real

R1 = 3 and R2 = 2

Float coefficients
Roots are complex
Real part = -1.2
Imaginary part = 1.375985
```

Function Template with Multiple Parameters

FUNCTION WITH TWO GENERIC TYPES

```
#include <iostream>
#include <string>

using namespace std;

template<class T1, class T2>
void display(T1 x, T2 y)
{
    cout << x << " " << y << "\n";
}

int main()
{
    display(1999, "EBG");
    display(12.34, 1234);
    return 0;
}
```

```
template<class T1, class T2, ...>
returntype functionname(arguments of types T1, T2,...)
{
        .....
        .....   (Body of function)
        .....
}
```

The output of Program

```
1999 EBG
12.34 1234
```

Overloading of Template Functions

- Templates can be overloaded by either template function or ordinary function.

- It can be done as:
  - Call an ordinary function that has an exact match.
  - Call a template function that could be created with an exact match.
  - Try normal overloading resolution to ordinary functions and call the one that matches.

- Automatic conversions are not applied to arguments on template functions.

- If no match is found, error is generated.

TEMPLATE FUNCTION WITH EXPLICIT FUNCTION

```
#include <iostream>
#include <string>

using namespace std;

template <class T>
void display(T x)
{
     cout << "Template display: " << x << "\n";
}
void display(int x)          // overloads the generic display()
{
     cout << "Explicit display: " << x << "\n";
}

int main()
{
     display(100);
     display(12.34);
     display('C');

     return 0;
}
```

The output of Program 12.9 would be:

```
Explicit display: 100
Overloaded Template Display 1: 12.34
Overloaded Template Display 2: 100, 12.34
Overloaded Template Display 1: C
```

Member Function Templates

- Class templates can also be defined outside the class.
- These functions must be defined by function templates.

Member Function Templates

```
// Class template ........

template<class T>
class vector
{
        T* v;
        int size;
    public:
        vector(int m);
         vector(T* a);
          T operator*(vector & y);
};
```

```
// Member function templates ........

template<class T>
vector<T> :: vector(int m)
{
        v = new T[size = m];
        for(int i=0; i<size; i++)
            v[i] = 0;
}

template< class T>
vector<T> :: vector(T* a)
{
       for(int i=0; i<size; i++)
       v[i] = a[i];
}

template< class T>
T vector<T> :: operator*(vector & y)
{
        T sum = 0;
        for(int i = 0; i < size; i++)
        sum += this -> v[i] * y.v[i];
        return sum;
}
```

Non-Type Template Arguments

- In addition to type argument T, other arguments such as strings, function names, constant expressions and built in types can also be used.
- General Form:

```
template<class T, int size>
class array
{
     T a[size];
     // .....
     // .....
};
```

- The arguments must be specified whenever a template is created.

```
array<int,10>  a1;
array<float,5> a2;
array<char,20> a3;
```

- The size is given as an argument to the template class.

END OF UNIT-IV