Unit - 5

❖ I/O STREAMS

❖Every program takes some data as input and generates processed data as an output following the familiar input process output cycle.

❖It is essential to know how to provide the input data and present the results in the desired form.

❖The use of the cin and cout is already known with the operator >> and << for the input and output operations.

❖we will discuss how to control the way the output is printed.

❖C++ supports a rich set of I/O functions and operations.

❖These functions use the advanced features of C++ such as Classes, derived classes, and virtual functions.

It also supports all of C's set of I/O functions and therefore can be used in C++ programs, but their use should be restrained due to two reasons.

- ✓I/O methods in C++ support the concept of OOPs.
- ✓I/O methods in C cannot handle the user-define data type such as class and object.

It uses the concept of stream and stream classes to implement its I/O operations with the console and disk files.

C++ Stream

The I/O system in C++ is designed to work with a wide variety of devices including terminals, disks, and tape drives. Although each device is very different, the I/O system supplies an interface to the programmer that is independent of the actual device being accessed. This interface is known as the **stream.**

❖A stream is a sequence of bytes.

❖The source stream that provides the data to the program is called the input stream.

❖The destination stream that receives output from the program is called the output stream.

❖The data in the input stream can come from the keyboard or any other input device.

❖The data in the output stream can go to the screen or any other output device.

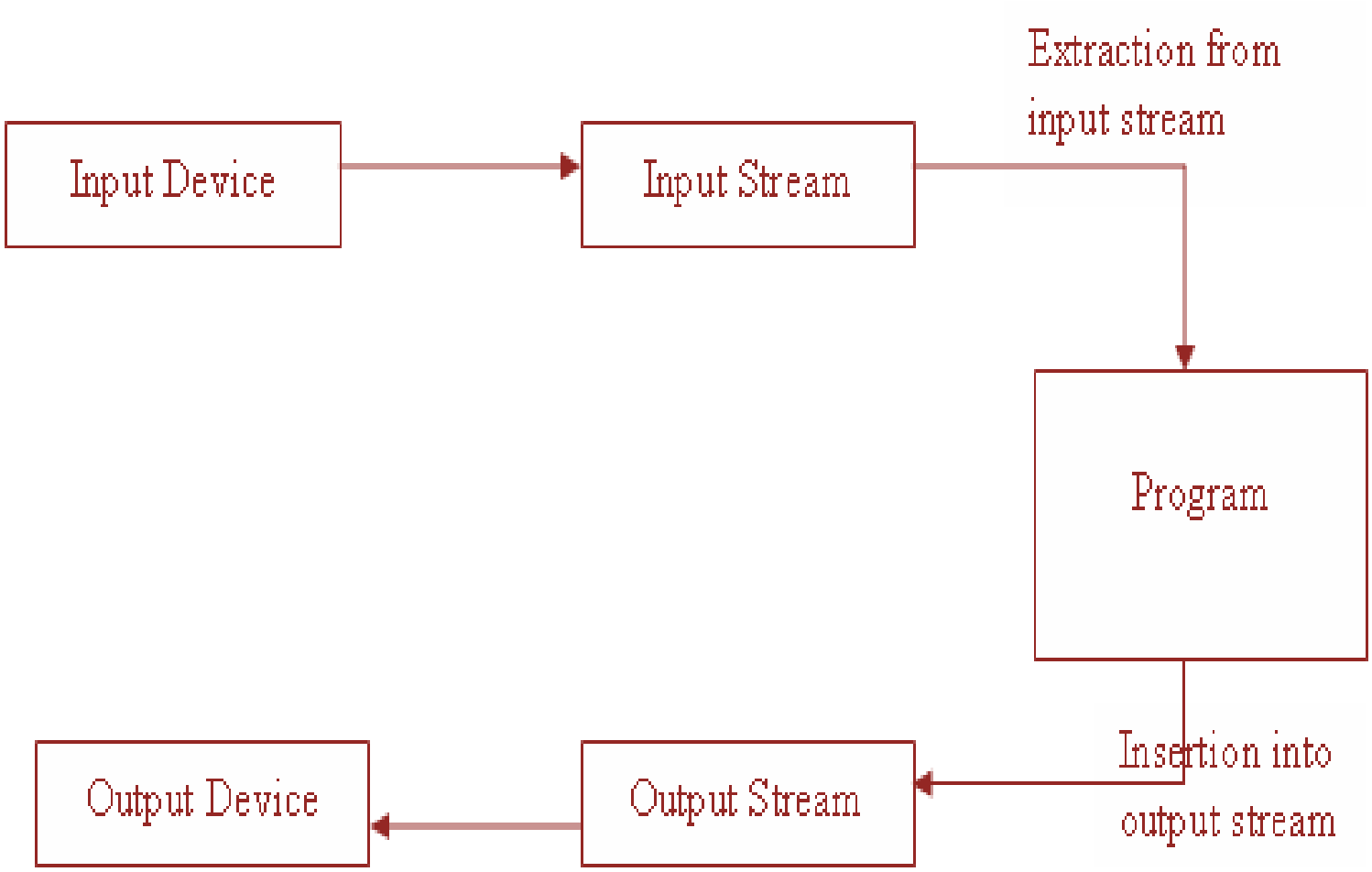
C++ contains several pre-defined streams that are automatically opened when a program begins its execution.

These include

- ✓ **cin**
- ✓ **cout**.

cin represents the input stream connected to the standard input device (usually the keyboard) and

cout represents the output stream connected to the standard output device (screen).


Input Device
Input Stream
Extraction from input stream
Program
Insertion into output stream
Output Stream
Output Device

C++ Stream Classes

The C++ I/O system contains a hierarchy of classes that are used to define various streams to deal with both the console and disk files.

These classes are called stream classes.

The hierarchy of stream classes used for input and output operations is with the console unit.

These classes are declared in the **header file iostream**.

This file should be included in all the programs that communicate with the console unit.

❖The **ios** are the base class for

- ✓ **istream** (input stream) and
- ✓ **ostream** (output stream)

which are in turn base classes for **iostream** (input/output stream).

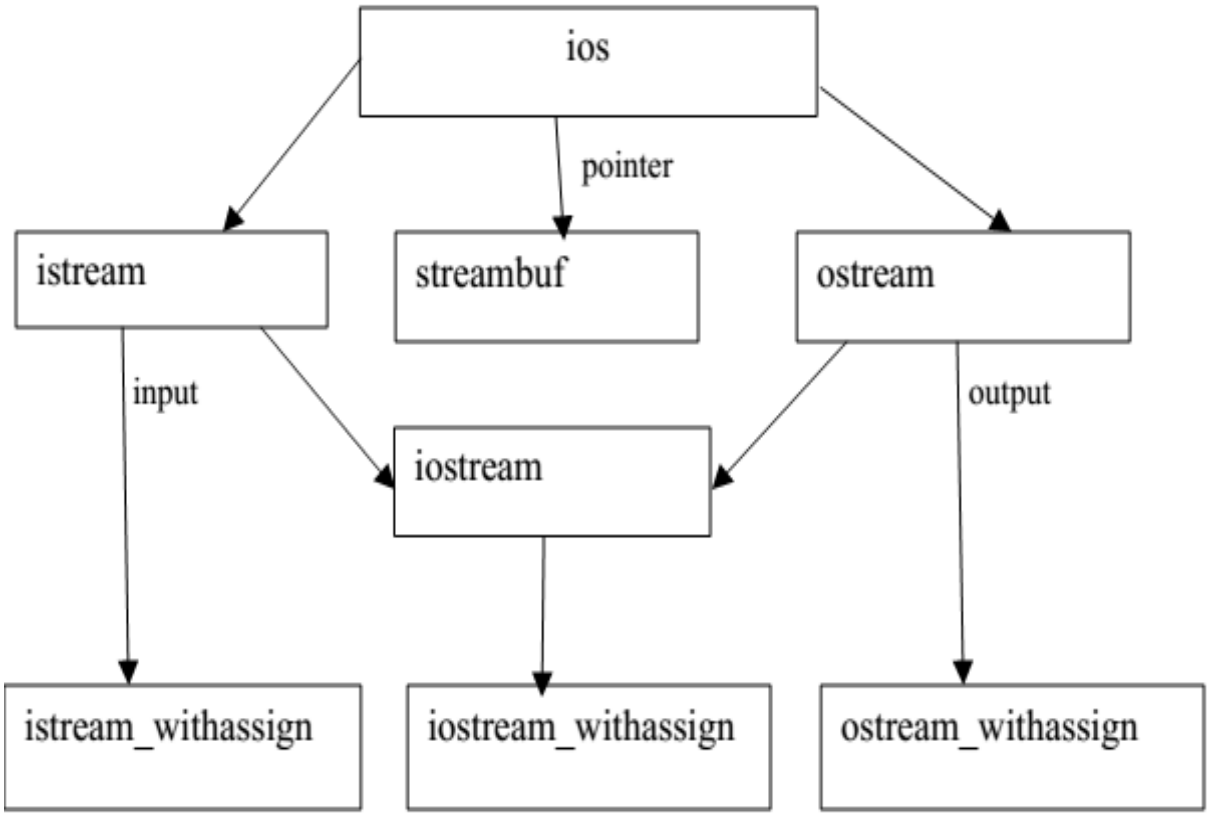
❖The class **ios** are declared as the virtual base class so that only one copy of its members is inherited by the iostream.

❖The class **ios** provide the basics support for formatted and unformatted I/O operations.

❖The class istream provides the facilities formatted and unformatted input while the class ostream (through inheritance) provides the facilities for formatted output.

The class iostream provides the facilities for handling both input and output streams. Three classes add an assignment to these classes:

- **istream_withassign**
- **ostream_withassign**
- **iostream_withassign**



Stream classes for console I/O operations

stream classes for Console Operation:

Class name	Content
ios (General input/output stream class)	•Contains basic facilities that are used by all other input and output classes •Also contains a pointer to a buffer object (streambuf object) •Declares Constants and functions that are necessary for handling formatted input and output operations
Istream (Input stream)	•It inherits the properties of ios •It declares input functions such as get(), getline(), and read() •It contains overload extraction operator >>
Ostream (Output Stream)	•It inherits the properties of ios. •It declares output functions such as put() and write(). •It contains an overload insertion operator <<.
iostream (Input/output stream)	•Inherits the properties of ios stream and ostream through multiple inheritances and thus contains all the input and output functions.
Streambuf	•It provides an interface to physical devices through buffers. •It acts as a base for filebuf class used ios file.

Unformatted input/output operations In C++

Using objects **cin** and **cout** for the input and the output of data of various types is possible because of overloading of operator **>>** and **<<** to recognize all the basic C++ types.

The operator **>>** is overloaded in the **istream class** and

operator **<<** is overloaded in the ostream class.

The general format for reading data from the keyboard:

cin >> var1 >> var2 >> …. >> var_n;

❖Here, **var_1**, **var_2**, ……, **var_n** are the variable names that are declared already.

❖This statement will cause computer to stop execution and look for I/P data from keyboard. The input data for this statement would be

data1 data2 ……. datan

❖The input data must be separated by white spaces and the data type of user input must be similar to the data types of the variables which are declared in the program.

❖The **operator >>** reads the data character by character and assigns it to the indicated location.

❖Reading of variables terminates when white space occurs or character type occurs that does not match the destination type.

❖Consider the following code

```
int code;
cin>>code;
```

Suppose the following data is given as input

4258D

operator will read characters upto 8 and value 4258 is assigned to code.

Character D remains in I/P stream and will be input to next cin statement.

put() and get() functions:

❖The class **istream** and **ostream** have predefined functions get() and put(), to handle single character input and output operations.

❖There are 2 types of **get()** functions.

❖we can use both **get(char*)** and **get(void)** to fetch characters including blank spaces, newline characters, and tab.

❖The function **get(char*)** assigns the input character to its argument and

get(void) to return the input character.

Syntax:

```
char data;
data = cin.get();     // get() return the character value and assign to data variable
cout.put(data);       // Display the value stored in data variable
```

Example:

char c;

cin.get(c); value returned by function get () assigned to variable c

For Example

cout.put('x');

Displays character x.

cout.put(ch);

Displays the value of variable ch. variable ch must contain a character value.

We can also use a number as an argument to the function put.

Cout.put(68);

Displays Character D. converts int value to char value and display the character whose ASCII value is 68.

C++ program to illustrate the input and output of data using get() and put()

```
#include <iostream>
using namespace std;
int main()
{
    char c;
    int count = 0;
    cout << “Input text: ";
    // Get the data
    cin.get(c);
while (c != '\n')
{
        // Print the data
        cout.put(c);
        count++;
        // Get the data again
        cin.get(c);
}
cout << “number of characters = “ << count << “\n”;
return 0;
}
```

getline() and write() functions:

❖In C++, the function getline() and write() provide a more efficient way to handle line-oriented inputs and outputs.

❖**getline()** function reads the complete line of text that ends with the new line character. This function can be invoked using the **cin object**.

Syntax: cin.getline(line, size);

❖The reading is terminated by the **'\n' character or size-1 characters are read**. The new line character is read but not saved.

❖ instead, it is replaced with a NULL character.

❖After reading a particular string the **cin** automatically adds the newline character at end of the string.

The **write() function** displays the entire line in one go and its syntax is the same as the getline() function.

Syntax: cout.write(line, size);

1st argument line represents name of string to be displayed.
2nd argument indicates no. of characters to display.

The key point to remember is that the **write() function** does not stop displaying the string automatically when a **NULL character** occurs. If the size is greater than the length of the line then, the **write() function** displays beyond the bound of the line.

It is possible to **concatenate** 2 strings using **write() function.**

cout.write(string1,m).write(string2,n); equivalent to

cout.write(string1,m);

cout.write(string2,n);

```
#include <iostream>
#include <string>
using namespace std;
// Driver Code
int main()
{
    char line[100];

    // Get the input
    cin.getline(line, 10);
    // Print the data
    cout.write(line, 5);
    cout << endl;
    // Print the data
    cout.write(line, 20);

    cout << endl;

    return 0;
}
```

```
i/p: abcdefghij
o/p:?
```

Formatted console I/O operation

C++ support a number of features that could be used for formatting output.

They include the following.

- ❖**ios class functions and flags**
- ❖**manipulators**
- ❖**User defined output functions**

The ios class, contains a large number of member function to assist in formatting the output in a number of ways.

The most important ones among them are

- ✓**width()**
- ✓**precision()**
- ✓**fill()**
- ✓**setf()**
- ✓**unsetf()**

IOS FORMAT FUNCTIONS

width() function a specifies the required field size for displaying output value.

Precision() function specifies the number of digits to be display after the decimal point of a float value.

fill() specifies a character that is used to fill the unused portion of a field.

Setf() to specify format flags that control the form of output display.

unsetf() clear the specified flag.

Manipulators are special function that can be included in the i/o statements to modify the format parameters of a stream. To access these manipulators the file iomanip should be included in the program.

Some important manipulator functions that are frequently used are shown in below table.

Manipulators	Equivalent ios function
Setw()	**width()**
setprecision()	**Precision()**
setfill()	**fill()**
setiosflags()	**Setf()**
resetiosflags()	**unSetf()**

In addition to these functions supported by c++ library we can create our own manipulator functions to provide any special output formats.

Defining Field width

The function width( ) is used to define the width of the field necessary for the output of an item. It must be accessed using objects of the ios class (commonly accessed using cout object).

Syntax for the width : cout.width(w);

- ✓Where w is the field width i.e number of columns to be used for displaying output.
- ✓The output will be printed in a field of w characters wide at the right end of the field.
- ✓The width function can specify the field width for only one item.
- ✓After printing one item it will revert back to the default.

For instance the statements

```
cout.width(4);
cout << 20 << 123;
```

produce the following statements output

		2	**0**	**1**	**2**	**3**

The first value is printed in right-justified form in four columns. The next item is printed immediately after first item without any separation; width(4) is then reverted to the default value, which prints in left-justified form with default size.

It can be overcome by explicitly setting width of every item with each cout statement as follows :

```
cout.width(4);
cout<<20;
 cout.width(4);
cout<<123;
```

These statements produce the following output :

		2	**0**		**1**	**2**	**3**

So that the field width should be specified for each item immediately if a width other than the default is desired for output.

Setting Precision :

✓ The function precision() is a member of the ios class and is used to specify the number of digits to be displayed after the decimal point while printing a floating point number.

✓ By default, the floating point numbers are printed with six digits after decimal point.

✓ This function must be accessed using objects of the ios class (commonly accessed using cout object)

Syntax : cout.precision(d);

Where d is the number of digits to the right of the decimal point.

For example the statements

cout.precision(2);	will produce the following output
cout<<2.23<<endl;	2.23 (perfect fit)
cout<<5.169<<endl;	5.17 (rounded)
cout<<3.5055<<endl;	3.51 (rounded)
cout<<4.003<<endl;	4 (no trailing zeros , truncated)

After displaying an item,the user defined precision will not revert to the default value.

Filling & Padding :

We have been printing the values using much larger field widths than required by the values. Unused positions of the field are filled with white spaces, by default. we can use the fill() function to fill the unused positions by any desired character. It is used in following form

cout .fill(ch);

where ch is the character to be filled in the unused portion.

For example the statements

```
cout . fill('*');
cout.precision(2);
cout.width(6);
cout<<12.53;
cout.width(6);
cout<<20.5;
cout.width(6);
cout<<2;
```

will produce the following output

*	1	2		5	3	*	*	2	0		5	*	*	*	*	*	2

Formatting with Flags and Bit-fields :

C++ provides a mechanism to set the printing of results in the leftjustified form, scientific notation etc. The member function of the ios class setf() is used to set flags and bitfields that control the output.

cout.setf(arg1,arg2);

Arg1 formatting flags defined in ios class. formatting flag specifies the format action required for the output.

Arg2 known as bitfield specifies the group to which the formatting flags belong.

There are 3 bit fields and each has a group of format flags which are mutually exclusive.

FLAGS AND BIT FIELDS FOR SETF() FUNCTION

Format	Flag	Bit Field
Left justification	ios::left	ios::adjustfield
Right justification	ios::right	ios::adjustfield
Padding after sign and base	ios::internal	ios::adjustfield
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

Note that the 1st argument should be one of the group members of the 2nd argument.

For example the statements

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

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";
```

will produce the following output

-	*	*	*	*	*	1	.	2	3	5	e	+	0	1

```
#include <iostream>
using namespace std;
int main()
{
int amount; //Variable declaration
amount = 564; //Variable initialization
//Converting base using showbase, hex, oct and dec
cout.setf(ios::showbase);
cout.setf(ios::hex,ios::basefield);
cout << "Hexadecimal =" << amount << endl;
cout.setf(ios::oct,ios::basefield);
cout << "Octal =" << amount << endl;
cout.setf(ios::dec,ios::basefield);
cout << "Decimal =" << amount << endl;
}
```

```
Output:
Hexadecimal =0x234
Octal =01064
Decimal =564
```

With out Bitfield argument we can also use a function with only flag value.

The list of flags without bitfield are

FLAGS THAT DO NOT HAVE BIT FIELDS

Flag	Purpose
Ios::showbase	Uses base indicator on output.
Ios::showpos	Displays + preceding positive number.
Ios::showpoint	Shows trailing decimal point and zeros.
Ios::uppercase	Uses capital case for output.
Ios::skipws	Skips white space on input.
Ios::unitbuf	Flushes all streams after insertion.
Ios::stdio	Adjusts the stream with C standard input and output.

Flags such as **showpoint** and **showpos** do not have any bitfields and therefore are used as single argument in setf(). This is possible because the setf() is declared as an overloaded function in the class ios.

❖The **manipulators** in C++ are special functions that can be used with insertion (<<) and extraction (>>) operators to manipulate or format the data in the desired way.

❖Certain manipulators are used with << operator to display the output in a particular format, whereas certain manipulators are used with >> operator to input the data in the desired form.

❖The manipulators are used to set field widths, set precision, inserting a new line, skipping white space etc.

❖ In a single I/O statement, we can have more than one manipulator, which can be chained as shown

```
Cout<<manip1<<manip2<<item;
Cout<<manip1<<item1<<manip2<<item2;
```

Manipulators are categorized into **2 types.**

i) Non-Parameterized manipulators

ii) Parameterized Manipulators

Non-Parameterized Manipulators :

endl: It is defined in ostream. It is used to enter a new line and after entering a new line it flushes (i.e. it forces all the output written on the screen or in the file) the output stream.

ws: It is defined in istream and is used to ignore the whitespaces in the string sequence.

ends: It is also defined in ostream and it inserts a null character into the output stream. It typically works with std::ostrstream, when the associated output buffer needs to be null-terminated to be processed as a C string.

flush: It is also defined in ostream and it flushes the output stream, i.e. it forces all the output written on the screen or in the file. Without flush, the output would be the same, but may not appear in real-time.

Parameterized Manipulators :

setw (val): It is used to set the field width in output operations.

setfill (c): It is used to fill the character ‘c’ on output stream.

setprecision (val): It sets val as the new value for the precision of floating-point values.

setbase(val): It is used to set the numeric base value for numeric values.

setiosflags(flag): It is used to set the format flags specified by parameter mask.

resetiosflags(m): It is used to reset the format flags specified by parameter mask.

User-defined Manipulators

C++ provides a feature of creating user defined manipulators as they do with built-in manipulators. Hence the user can design their own manipulators to control the appearance of the output depending on the requirement.

Syntax for creating a manipulator

```
Ostream  & manipulator(ostream & output)
{
set of statements;
return output;
}
```

```
#include <iostream>
 #include <iomanip>
ostream & curr(ostream &output)
{
cout << setprecision(2);
cout << “Rs.”
return output;
}
void main()
{
float amt = 4.5476;
cout << curr << amt ;
}
```

Output:

4.55