UNIT III – C++ PROGRAMMING ADVANCED FEATURES

Abstract class – Exception handling – Standard libraries - Generic Programming – templates – class template – function template – STL – containers – iterators – function adaptors – allocators – Parameterizing the class – File handling concepts.

Abstract class.

- Class that contains at least one pure virtual function is known as abstract class.
- Abstract classes cannot be instantiated, but can be subclassed.
- Classes that are derived from abstract classes must implement pure virtual functions of the abstract class. The derived class are also known as *concrete* classes.

```
#include <iostream.h>

class shape                     // Abstract class
{
public:
        virtual void display() = 0;
};

class triangle : public shape       // Concrete class
{
public:
        void display()
        {
                cout << "Triangle shape";
        }
};

class rectangle : public shape      // Concrete class
{
public:
        void display()
        {
                cout << "Rectangle shape";
        }
};

int main()
{
        triangle t1;
        t1.display();
        rectangle       r1;
        r1.display();
        return 0;
}
```

Exception

- Exceptions are runtime anomalies that a program may encounter while executing.
  - *Synchronous* exceptions *out-of-range index, divide by zero,* etc., could be handled
  - *Asynchronous* exceptions such as keyboard interrupt are beyond user control.

control is transferred and handled.

- *C++* handles synchronous exception using keywords try, throwand catch.

1. Keyword tryis used to preface a block of statement that may throw exception.
2. When exception is detected, it is raised by a throwstatement in the tryblock.
3. Control is transferred to the catchblock, which handles the exception.

```
try
{
        throw exception
}
catch (type1   arg) {     }
catch (type2   arg) {     }
…
```

- There can be multiple catch blocks for a try block. The catch block that matches the exception type is executed. Control is transferred after end of the last catch block.
- If no catch block matches, then program is terminated with a call to exit()or abort()

```
#include <iostream.h>

const int max=5;
int main()
{
        int a, b, x;
        cout << "Enter values   : ";
        cin >> a >> b;
        try
        {
                if (b == 0)
                        throw(b);
                  else
                        cout << float(a)/b;
        }
        catch(int i)
        {
                cout << "Divide by zero";
        }
        return 0;
}
```

stack operation using exception handling.

```
#include <iostream.h>
const int size = 3;
class mystack
{
        int st[size];
        int top;
public:
        class full { };              //exception class class empty { };
        //exception class mystack()
        {
                top = -1;
        }
        void push(int var)
        {
                if(top >= size-1)
                        throw full();      //throw exception for stack full st[++top] = var;
        }
        int pop()
        {
                if(top < 0)
                        throw empty();      //throw exception for stack empty return st[top--];
        }
};
int main()
{
        mystack s1;
        try
        {
                s1.push(11);
                s1.push(22);
                s1.push(33);
                cout << s1.pop(); cout
                << s1.pop(); cout   <<
                s1.pop();
                cout  << s1.pop();   // stack empty error
        }
        catch(mystack::full)
        {
                cout << "Exception: Stack Full";
        }
        catch(mystack::empty)
        {
                cout << "Exception: Stack Empty";
        }
        return 0;}
```

Generic Catch Block

- A catch block that handles all exceptions is known as generic catch.
- The ellipsis argument matches any type of data.

```
catch(...)
{

}
```

Restrict Exceptions Thrown Out Of A Function

- A throwclause can be added to a function definition to restrict the exceptions thrown.
- The following function test can throw only exceptions of type int, char or double.

```
void test() throw(int, char, double)
{

}
```

standard libraries.

- C++ libraries are broadly classified into two types.
- *Standard Function Library* consisting of general-purpose, stand-alone functions inherited from
  C. Some of them are:
    - String and character handling
    - Mathematical
    - Date and time
    - Dynamic allocation
- *Object Oriented Class Library* which is a collection of classes and associated functions. Some of them are:
    - Standard C++ I/O Classes
    - String Class
    - STL Container Classes, Algorithms, Function Objects, Iterators, Allocators
    - Exception Handling Classes

String Class

- Standard C++ provides a new class called stringby including <*string*> header file.

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

- The stringclass takes care of memory management, allows use of overloaded operators and provides a set of member functions
- A string object can be defined by using default/parameterized/copy constructor

List operators applicable to C++ string class.

= + += == != < <= > >= [] << >>

List the functions applicable to the string class. Explain with an example.

append() appends full/part of a string
empty() returns true if the string is empty
erase() removes no. of characters specified from the given position
find() searches for the occurrence of a substring
insert() inserts no. of characters specified at the given position
length() returns length of the string
replace() replaces characters of a string with another string/substring
swap() swaps the string contents with the invoked string
substr() returns a substring of the given string
find_first_of()returns the position of first occurrence of the given character
find_last_of() returns the position of last occurrence of the given character

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

int main()
{
        string s1("ANSI "),s2("Standard "),s3,s4;
        cin >> s3;                      // C++
        s4 = s3;
        if (s3 == s4)
                cout << "Strings s3 and s4 are identical";
        s4 = s1 + s2 + s3;
        cout << s4;                     // ANSI Standard C++
        string s5 = "ISO ";
        cout << s2.insert(0,s5);            // ISO Standard
        cout << s2.append(s3);              // ISO Standard C++ cout <<
        s4.erase(7,3);              // ANSI Stard C++ cout << s2.replace(0,4,s1);
        // ANSI Standard C++ s1.swap(s3);
        cout << s1 << s3;                   // s1: C++   s3: ANSI
        cout << s2.substr(7,3);             // and cout <<
        s2.find_first_of('a');      // 7 for(int i=0; i<s2.length();
        i++)
                cout << s2[i];
        return 0;
}
```

generic programming

- Generic programming enables to define generic classes and functions. It is implemented in C++ using templates.
- Compiler uses the template to generate different instances through template parameters.
- It improves programming efficiency and allows the programmer to defer more of the work to the compiler.

function template

- A generic function defines a set of operations that can be applied to various types of data.
- The *type* of data that the function will operate upon is passed as a parameter.
- Function template avoids code redundancy among structurally similar families of functions.
- Compiler uses function template to generate version of the function that is needed.
- A function that is an instance of a template is also called a template function.

```
template <class T>
returntype  functionname(arguments)
{

}
```

- The templatekeyword indicates that a function template is defined.
- The keyword classmeans generic type and parameter *T* is known as template argument, which can be substituted by any type.

```
#include <iostream.h>

template <class T>                  // Template function void arrange(T arr[],
int size)
{
      for(int i=0; i<size-1; i++)
      {
            for(int j=i+1; j<size; j++)
            {
                  if (arr[i] > arr[j])
                        {
                              T temp;
                              temp = arr[i];
                              arr[i] = arr[j];
                              arr[j] = temp;
                        }
            }
      }
}
```

```
int main()
{
        int x[6] = {1,-4,8,0,3,5};
        float y[6] = {2.2,-0.4,3.5,1.9,2.7,0.7};
        char z[7] = {'I' 'n','d','i','a','n'};

        arrange(x,
        6);
        arrange(y,
        6);
        arrange(z,
        6);

        cout << "Sorted Arrays \n Int \t Float \t Char \n";
        for (int i=0; i<6; i++)
                cout << x[i] << "\t" << y[i] << "\t" << z[i] << "\n";
        retur
        n 0;
}
```

- For each call, the compiler generates the complete function, replacing the type parameter with the argument type (int/float/char).

Class Template

- Like generic functions, generic classes can be created.
- A generic class that defines algorithms used by that class and the actual type of the data being manipulated will be specified as a parameter when objects of that class are created.
- Class templates are used for data storage (container) classes. Class templates are also called parameterized types.

```
template <classT>
class
{
    member of type T
};
```

- Classes are instantiated by defining an object using the template argument. A class created from a class template is called template class.

classname <type>objectname;

- Member functions of a class template are defined externally as function templates.

```
template <classT>
returntype classname <T>::functionname (arguments)
{

}
```

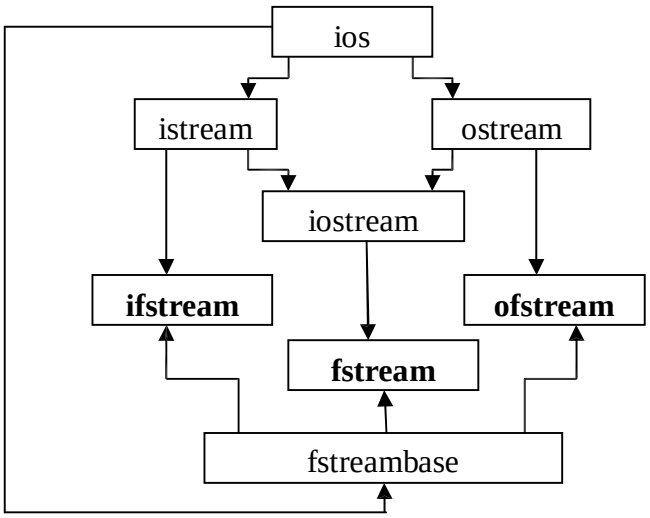
```
#include <iostream.h>
const int size = 5;
template <class T>
class mystack
{
        T st[size];
        int top;
public:
        mystack()
        {
                top = -1;
        }
        void push(T var)
        {
                st[++top] = var;
        }
        T pop()
        {
                return st[top--];
        }
        bool empty()
        {
                if (top < 0)
                        return true;
                else
                        return false;
        }
        bool full()
        {
                if (top == size-1)
                        return true;
                  else
                        return false;
        }
};
int main()
{
        mystack <char>          s1;     //  Character Stack
        mystack <int>           s2;     //  Integer stack

        for (char x='a'; x<='e'; x++)
        {
                if (s1.full())
                        cout << "Char Stack Full";
                  else
                        s1.push(x);
        }
```

```
        while (!s1.empty())
                cout << s1.pop();
        return 0;
}
```

File Handling

- A file is a collection of related data stored on an auxiliary storage device.
- In file I/O, *input* refers to reading from a file and *output* is writing contents to a file.
- <fstream.h>must be included for file I/O.
  - C++ provides the following classes for high-level file handling.



ifstream	Provides input operations. Contains functions such as open(), get(), read(), tellg() seekg(), getline() inherited from classes *fstreambase* and *istream*
ofstream	Provides output operations. Contains functions such as open(), put(), write(), tellp(), seekp() inherited from classes *fstreambase* and *ostream*
fstream	Provides input / output operations and inherits functions from *iostream* and *fstreambase*. Contains open()with default input mode.

syntax of open( ) method in file handling.

- A file should be opened prior to any I/O operations. It is done using open()method.

  filestreamclass fsobj;
  fsobj.open("*filename*", *mode*);

- The *mode* parameter is optional if file object is of either *ifstream* or *ofstream* class.
- It outlines purpose for which the file is opened. It is a constant defined in **ios** class. Some are:

ios::in Open for reading (default for *ifstream*)
ios::out Open for writing (default for *ofstream*)
ios::ate Start reading or writing at end of file
ios::app Start writing at end of file ios::trunc
Truncate file to zero length if it exists
ios::nocreate Error when opening if file does not already exist
ios::noreplace Error when opening for output if file already exists
ios::binary Open file in binary (not text) mode

Detect End-Of-File

- When a file is read continuously, eventually an end-of-file (EOF) condition will be encountered.
- The EOF is a signal sent to indicate that there is no more data to read.
- Detecting EOF is necessary to prevent any further reading using a loop.
- The *fileobject* returns 0when end-of-file is reached causing the loop to terminate.

```
while(fsobj)
{

}
```

List the errors that may occur in file I/O

- Errors may occur during file handling due to the following:
  - To open a non-existent file
  - To create a file with a name that already exists
  - Reading past end-of-file
  - Insufficient disk space
  - Attempt to perform operation on a file for which it is not opened
- Error status function can be used to check file stream status for specific error.

eof() returns true if end-of-file is encountered
fail() returns true if the input or output operation has failed
bad() returns true if an invalid operation is attempted
good() returns true if no error has occurred

closing files

- When a program terminates, the *fileobjects* goes out of scope.
- This calls the destructor that closes the file.
- However, it is recommended to close the files using close()method.

```
fsobj.close();
```

- A closed *fileobject* can be opened in another mode.

Character I/O in file operations

- The get()and put()function, members of *istream* and *ostream*, handle single character at a time.
- The get()function reads a single character from the associated stream whereas the put() function is used to write a single character onto the stream.
- Both functions are usually executed within a loop to perform sequential I/O operation.

charvar = *fsobj*.get();
fsobj.put(*charvar*);

- The following program writes user input onto a file

```
#include <iostream.h>
#include <fstream.h>
int main()
{
        ofstream fp; char ch;
        fp.open("sample.txt");

        cout << "Ctrl + Z to terminate: ";
        while( (ch = cin.get()) != EOF)
                fp.put(ch);

        fp.close();
        return 0;
}
```

- The following program reads the file character-by-character and determines number of vowel present.

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

int main()
{
        ifstream fp;
        char ch;
        int vc = 0;
        fp.open("sample.txt");

        while(fp)
        {
                ch = fp.get();
                switch(tolower(ch))
                {
```

```
                        case 'a' : case 'e' : case 'i' : case 'o' : case 'u' :


                                                        }
        }
        vc++;
        break;
        cout << " No. of vowels : " << vc;

        fp.close();
        return 0;
}
```

Write a program in C++ to perform file copy.

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

int main()
{
        ifstream src;
        ofstream dst;
        char file1[10], file2[10];
        char ch;
        cout << "Enter source   and destination file : ";
        cin >> file1 >> file2;

        try
        {
                src.open(file1);
                if(src.fail())
                        throw file1;     // Source file non-existent dst.open(file2);
                if(!dst.fail())
                        throw file2;    // Destination file exists while(src)
                {
                        ch = src.get();
                        dst.put(ch);
                }
                src.close();
                dst.close();
        }
        catch (char *name)
        {
                if (strcmp(name,file1) == 0)
                        cout << "Source file not available";
                  else
                        cout << "Destination file already exists";
        }
```

file pointer

- Each file object has two associated file pointers namely *get* (input) and *put* (output) pointer.
- For each file I/O, the appropriate pointer is automatically advanced.
- The default action for pointers on opening a file are:
  - ios::in get pointer is moved to beginning of the file
  - ios::out put pointer is moved to beginning of the file
  - ios::app put pointer is moved to end of the file
- File pointers can be moved randomly using functions seekg()and seekp().

Function	**Description**
seekg(*abspos*) seekg(*offset,refpos*)	o Moves the get pointer to the specified byte. o The positional value may be *absolute* or *relative*. o If relative, *refpos* is a constant ios::beg, ios::cur, or ios::end referring to start, current, and end of the file. fobj.seekg(10,ios::cur);
seekp(*abspos*) seekp(*offset,refpos*)	o Like seekg, seekp( ) moves put pointer. fobj.seekg(100);
tellg()	o Returns current position of the get pointer.
tellp()	o Returns current position of the put pointer.

random access file I/O

```
#include <fstream.h>
#include <iostream.h>
class student
{
        int rollno; char name[20];
        int arrear; float cgpa;
public:
        void getdata()
        {
                cin >> rollno >> name >> arrear  >> cgpa;
        }
        void display()
        {
                cout << rollno << name << arrear << cgpa;
        }
};

int main()
{
        fstream fobj; student s1; int
        ch, rno;
        cout << "1.Append  2.Display"; cout << "Enter
```

```
        your choice : "; cin >> ch;
        switch(ch)
        {
                case 1:
                        s1.getdata();
                        fobj.open("placement.dat", ios::app|ios::binary);
                        fobj.write((char *)&s1, sizeof(s1));
                        fobj.close();
                        break;
                case 2:
                        fobj.open("placement.dat", ios::in|ios::binary);
                        cout << "Enter roll number : ";
                        cin >> rno;
                        fobj.seekg((rno - 1) * sizeof(s1), ios::beg); fobj.read((char *)&s1,
                        sizeof(s1)); s1.display();
                        fobj.close();
                        break;
        }
        return 0;
}
```

STL.

- STL provides general-purpose, templatized classes and functions that implement many popular and commonly used algorithms and data structures.
- STL simplifies software development by decreasing development time, simplifying debugging and maintenance and increasing the portability of code.
- STL is exhaustive but the usage is very simplistic.

STL

Component	**Description**
Container	object that is able to keep and administer other objects
Algorithm	procedure that is able to work on different containers
Iterator	pointer that is able to iterate through the contents of a container
Function Object	a class that has the function-call operator() defined
Adaptor	encapsulates a component to provide another interface (e.g: stack out of a list)

- Containers are the STL objects that actually store data.
- Each container has defined for it an allocator.
- Allocators manage memory allocation for a container.
- Iterators are used to access elements within the containers.
- Once the data is stored in a container, it can be manipulated by any of the available algorithm.

STL Container Classes

- There are two types of STL containers: *Sequence Containers* and *Associative Containers*.
- A sequence is a kind of container that organizes a finite set of objects, all of the same type, into a strictly linear arrangement. Examples are *Vector*s, *List*s and *Deque*s.
- Associative containers provide ability for fast retrieval of data based on keys. If the key value must be unique in the container, *Set* and *Map* can be used.

Vector Class

- The vectorclass behaves like a dynamic array. The file <vector>should be included.
- It provides random access to its elements. The subscripting operator [ ] is also defined.
- Vectors offer great power, safety, and flexibility, but are less efficient than normal arrays.
- Some of the commonly used member functions of vector class are:

Member Function	**Description**
begin()	Returns an Iterator to the first element in the vector
clear()	Removes all elements from the vector
empty()	Returns true if the invoking vector is empty
end()	Returns an Iterator to the end of the vector
erase(*I*)	Removes an element pointed to by the Iterator *I*
erase(*st, end*)	Removes all elements in the range from *st* to *end*
insert(*I, val*)	Inserts *val* before Iterator *I*
pop_back()	Removes the last element in the vector
push_back(*val*)	Adds an element with value *val* at the end of the vector
size()	Returns number of elements currently in the vector
back()	Returns a reference to the last element

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

int main()
{
        vector <int> v1(5,0);              // Vector of size 5, each 0 int data, i;

        for (i=0; i<v1.size(); i++)
                cin >> v1[i];
        cout << "Enter an element : ";
        cin >> data;
        v1.push_back(data);                // Add element dynamically cout << "Current size: " <<
        v1.size();
        cout << "First element : " << v1.front();
        cout << "Last element : " << v1.back();
        if (!v1.empty())
                v1.pop_back();             // Deleting last element cout << "Current size: " <<
        v1.size();
```

```
        for (i=0; i<v1.size(); i++)
                cout << v1[i] << "   ";
        return 0;
}
```

Lists

- The listclass supports a bidirectional, linear list. The file <list>should be included.
- A list can be accessed sequential only whereas vector can be accessed randomly.
- Some member functions applicable to list are:

Member Fn	**Description**
begin()	Returns an Iterator to the first element in the list
clear()	Removes all elements from the list
empty()	Returns true if the invoking list is empty
end()	Returns an Iterator to the end of the list
erase(*I*)	Removes an element pointed to by the Iterator *I*
erase(*st, end*)	Removes all elements in the range from *st* to *end*
insert(*I, val*)	Inserts *val* before the Iterator *I* and returns an Iterator
pop_back()	Removes the last element in the list
pop_front()	Removes the first element in the list
push_back(*val*)	Adds an element with value *val* at the end of the list
push_front(*val*)	Adds an element with value *val* at the start of the list
size()	Returns number of elements currently in the list
back()	Returns a reference to the last element in the list
front()	Returns a reference to the first element in the list
remove(*val*)	Removes elements with value *val* from the list
reverse()	Reverses the invoking list
sort()	Sorts the list in ascending order
merge(*L*)	Merges two ordered list.

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

int main()
{
        list <int> l2,l3;
        list <int>::iterator first;

        for (int i=1; i<10; i+=2)
                l2.push_back(i);
        for (int i=2; i<10; i+=2)
                l3.push_front(i);
        first = l3.begin(); cout << "List L3 : ";
        while (first != l3.end())
```

```
        cout << *first++ << " ";
    cout << "\n L3 size " << l3.size();

    l3.sort();
    l2.merge(l3);
    cout << "\n Merged List " ; first = l2.begin(); while(first != l2.end())
        cout << *first++ << " ";
    return 0;
}
```

Maps

- The mapclass supports an associative container in which unique key is mapped with values
- The map class does not allow duplicate keys. The file <map>should be included.
- Some commonly used member functions of map class are:

Member Fn	**Description**
begin()	Returns an Iterator to the first element in the map
clear()	Removes all elements from the map
empty()	Returns true if the invoking map is empty
end()	Returns an Iterator to the end of the map
count(*K*)	Returns whether a key *K* occurs in the map
erase(*I*)	Removes an element pointed to by the Iterator *I*
erase(*I1*,*I2*)	Removes all elements in the range pointed to by the Iterators *I1* and *I2*
erase(*val*)	Removes from the map that have keys with value *val*
find(*K*)	Returns an iterator that points to element whose first component equals the key. If no such element is found, then the pointer points to end()
insert(*I*,*val*)	Updates value of the element pointed by Iterator *I* with *val*
insert(pair<*Kt*,*Vt*>, (*key*,*val*))	Inserts a key and its associated value.
size()	Returns number of elements currently in the map

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

int main()
{
    map <char, int> m1;
    map <char, int>::iterator ptr;
    char ch;

    for(int i=0; i<26; i++)
        m1.insert(pair<char, int>('A'+i, 65+i));

    cout << "Enter key: ";
```

```
    cin >> ch;
    ptr = m1.find(ch);
    if(ptr != m1.end())
            cout << "ASCII value : " << ptr->second;
    else

  cout << "Key not in map";
    return 0;
  }
```

STL algorithms.

o STL algorithms are standalone template functions that can be used either with container classes or arrays.
o Algorithms are generic and are parameterized by iterator types. Hence separated from particular implementations of data structures.
o To have access to STL algorithms, include <algorithm>in the program.
o It is classified into four groups namely: mutating, non-mutating, sorting and numeric operations. Some of them are:

Algorithm	**Description**
count()	Returns no. of elements in the range that match a given value
count_if()	Returns no. of elements in the range for which the given function is true
find()	Finds first occurrence of a value in the given sequence
for_each()	Apply an operation to each element
copy()	Copies the given sequence
remove()	Deletes elements with the specified value
replace()	Replaces elements with the specified value.
reverse()	Reverses the order of elements in the specified range
swap()	Swap two elements
transform()	It transforms all the elements in the range according to given function
sort()	Sorts values in the given range

STL Function Objects

- o A function object is an object that has the function operator() defined/overloaded.
- o STL algorithms may have function object as an argument.
- o The header file <functional>must be included
- o STL provides a rich assortment of built-in function objects. Some are:

plus minus multipliesdivides
modulus not_equal_to greater greater_equal
less less_equal logical_orlogical_not
negate equal-to logical_and

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

int main()
{
        float fdata[] = { 19.2, 87.4, -33.6, 55.0, 42.2 };
        sort(fdata, fdata+5, greater<float>());        // sort the array for(int j=0; j<5; j++)
                cout << fdata[j] << " " ;
        return 0;
}
```

STL Adaptors

- Adaptors are template classes that provide interface mappings.
- Adaptors are classes that are based on other classes to implement a new functionality.
- *Stack*—A stack can be instantiated either with a vector, a list or a deque. The member functions empty, size, top, pushand popare accessible to the user.

  stack < deque<int> > s1;
- *Queue*—A queue can be instantiated with a listor a deque. Its public member functions are empty, size, front, back, push and pop. As with the stack, two queues can be compared using operator ==and operator <.

  queue < list <int> > q1;
- *Priority queue*—A priority queue can be instantiated with a vector. A priority queue holds the elements added by pushsorted by using a function object.

  priority_queue < vector <int>, less <int> > pq1;

iterator

- Iterators are smart pointers used to access / modify elements of a container class.
- Operators ++and *are overloaded
- STL implements five different types of iterators.
  - *random access* iterators↻Store and retrieve values with random access.
  - *bidirectional* iterators↻can store and write in both forward and backward
  - *forward* iterators↻can read, write, and move in forward direction
  - *input* iterators↻can only be used to read a sequence of values in forward direction
  - *output* iterators↻can only be used to write a sequence of values in forward direction
- For example, vector uses random access iterator, whereas list has only a bidirectional iterator.
- Iterators determine which algorithms can be used with which containers.
- Iterators allow generality. For example, an algorithm to reverse a sequence implemented using bidirectional iterators can be used on lists, vectors and deques.

manipulator.

- Manipulators are used to format I/O stream.
- Most manipulators have parallel member functions from iosclass.
- To access manipulators, <iomanip.h>must be included in the program.

Manipulators	**Description**
Endl	Output a newline character.
setfill(*ch*)	Set the fill character to *ch*.
setiosflags(*f*)	Turn on the flags specified in flg. Flag may be ios::right, ios::left ios::fixed, ios::hex, ios::oct, etc.
setprecision(*p*)	Sets the number of digits of floating point precision.
setw(*w*)	Sets the field width to *w*.

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

int main()
{
        cout << setprecision(2) << setw(10) << setfill('*')
                << setiosflags(ios::right|ios::fixed);
        cout << 2343.4 << endl;                         // ***2343.40 return 0;
}
```