UNIT-III

Templates

Template:

It is a method for writing a single function for a class for family of similar function or class in a generic manner.

Types of templates.

1. Function template
2. Class Template

when a single function is written for a family of similar function, it is called function template. In this function atleast one formal argument is generic.

class template:

A class template is a class definition that describes a family of related classes.

For eg: A class template for an array class would enable us to create arrays of various data types such as int, float, char array. we can define a template

for a function also.

Eg:- multiplication function. The multiplication function is used to multiply integer value, floating point value and double values

A template is considered as a kind of macro when an object of a specific type is defined for actual use, the template definition for that class is substituted with the required data type. Since the template is defined with a parameter that would be replaced by a specified data type at the time of actual use of the class (or) function. The templates are sometimes called as parameterised classes or, function.

Syntax for function template is

```
template <class T>
T fnname (T formalargs)
{
return (T);
}
```

Syntax of class template

```
template <class T>
class userdefined name
{
data members of class
};
```

class template :

```
template <class T>
class sample
{
 T value, value1, value2;
 public:
    void getdata();
    void sum();
 };
template <class T>
void sample <T> :: getdata()
{cin >> value1 >> value2;
 }
template <class T>
void sample <T> :: sum()
{
 value = value1 + value2;
 cout << value;
   }
void main()
{
 Sample <int> obj1;
 Sample <float> obj2;
 obj1. getdata();
 obj1. sum();
 obj2. getdata();
 obj2. sum();
 }
```

class template with default constructor.

```
template <class T>
class sample
{
 T value;
 public:
    sample(T=0) //(or) Sample ()
    {
    }
    void display()
    {
     cout << " Default constructor called " << value;
    }
 };
void main()
{
 sample <int> obj1;
 obj1. display();
 sample <float> obj2;
 obj2. display();
 }
```

class template with parameterised constructor & destructor

```
template <class T>
class sample
{
 T value;
 public:
   sample (T n) : value(n)
   {
    }
   ~sample ()
    {
     }
   void display()
   {
    cout << " Default constructor called " << value;
     }
 };
void main()
{
 sample <int >obj1(10);
  obj1. display();
 sample <float > obj2(22.14);
 obj2. display();
}
```

Function template:

```
template <class T>
T swap( T first, T second)
{
 T temp;
  temp = first ;
  first = second;
  second = temp;
 }
void main()
{
  int ix, iy;
  float fx, fy;
  cin >>ix>> iy>>fx>>fy;
  swap(ix, iy);
  swap (fx, fy);
 }
```

Function template with multiple parameters:

we can use more than one generic datatype in the template statement using a comma separated list. The syntax is

```
template <class T1, class T2 . . . .>
returntype fnname (args of types T1, T2...)
 {
  }
```

```
template <class T1, class T2>
void display (T1 x, T2 y)
{
cout << x << y;
 }
 void main()
 {
  display (1000, "ABC");
  display (12.34, 1234);
  }
```

overloading of template function:

A template function may be overloaded either by its template function or ordinary function of its name. In such cases overloading done by (1) call an ordinary function that has an exact match

(2) call a template function that could be created with an exact match.

(3) Try normal overloading to ordinary function and call the one of that matches

An error is generated if no matches is found. No automatic conversions are applied to arguments on the template function.


```
template <class T>
void display (T x)
{
 cout << "Template display" << x;
 }
 void display (int x)
 {
  cout << "Explict display" << x;
  }
 void main()
 {
  display (100);
  display (12.34);
  display ('c');
```

Non-type Template arguments

```
template <class T, int size>
class array
{
T a[size]
};
array <int, 10> a1;
array <float, 5> a2;
```

It is also possible to use non-type arguments in the templates that is addition to the type argument T we can also use other arguments such as strings, function names, constant expression and built in types

Exception handling

There are two common types of error

1. Logical error
2. syntactic error.

LOGIC ERROR

This occurs due to the poor understanding of problem and solution procedure.

SYNTACTIC ERROR

This occurs due to the poor understanding of language itself.

we can detect errors by debugging mechanisms and testing procedures. The problems other than logic or syntactic errors are known as exceptions. Exceptions are runtime anomalies (or) unusual conditions that a program may encounter while executing. The

anomalies includes division by zero, access to an array outside its boundary (or) running out of memory or disk space

Basics of Exception handling:

There are two types of exceptions

1. synchronous exceptions
2. Asynchronous exceptions

synchronous:

Errors such as out of range index and overflow error belongs to the synchronous type.

Asynchronous:

The errors that are caused by events beyond the control of the program are called asynchronous exceptions.

Eg:- keyboard interrupt

The purpose of exception handling mechanism is to provide to detect and report an exceptional circumstance so that appropriate action can be taken.

Exception handling task :

1. Find the problem i.e., Hit the exception
2. Inform that an error has occured i.e., throw exception.
3. Receive the error information i.e., catch the exception.
4. Take corrective actions i.e., handle the exception

The error handling code basically consists of two segments. one to detect errors and throw the exception and the other to catch the exception and take appropriate actions.

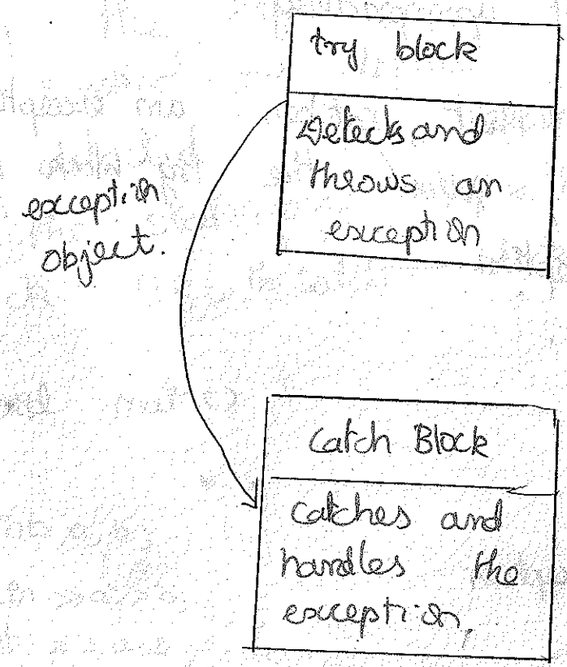
Keywords in exception error handling mechanism

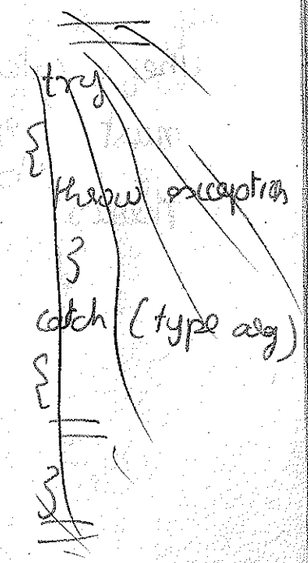
The keywords in error handling mechanisms

1) Try
2) Throw
3) catch

syntax of error handling mechanism

```
try (expression)
{
==
}
throw (expression)
{
error message;
}
```





The exception handling mechanism has 3 keywords

1. Try
2. Throw
3. catch

The keyword try is used to preface a block of statements which may generate exceptions. This block of statement is known as try block. When an exception is detected, it is thrown using throw statement in the try block. The catch block defined by the keyword catch, catches the exception thrown by the throw statement in the try block and handles it appropriately.

The catch block that catches an exception must immediately follow the try block that throws the exception.

```
---
try
{
 throw exception
 }
catch (type arg)
{
 --
 }
---
```

when the try block throws an exception, the program control leaves the try block and enters the catch statement of the catch block. If the type of object thrown matches the argument type in the catch statement, then the catch block is executed for handling the exceptions. If they do not match the program is aborted with the help of abort() function which is invoked by default. when no exception is detected and thrown, the control goes to the statement immediately after the catch block i.e; the catch block is skipped.

```
void main()
{
 int a,b;
 cin >>a>>b;
 int x = a-b;
 try
 {
   if (x!=0)
    {
     cout << a/x;
     }
```

```
        else
        {
         throw (x);
         }
        }
        Catch (int i)
        {
        cout << x;
         }
      }
```

2) void divide (int x, int y, int z)

```
{
 if ((x-y) != 0)
 {
 int R = z/(x-y);
 cout <<R;
 }
else
 {
 throw (x-y); // throw point
 }
}
void main()
{
 try
  { divide (10,20,30);
   divide(10,10,20).
```

```
 catch (int i)
 {
 cout << " caught the exception" << x-y ;
  }
}
```

Throwing mechanism :

when an exception that is desired to be handled is detected, it is thrown using the throw statement in one of these following forms.

```
throw (exception);
throw exception;
throw ;
```

catching mechanism :

The code for handling the exception is included in the catch block. The catch block looks like a function definition. The syntax is

```
catch (type arg)
{
==
}
```

The type indicates the type of exception

that catch block handles. The parameter arg is an optional parameter name. The catch statement catches an exception whose type matches with the type of catch argument. when an exception is caught, the code in the catch block is executed

Multiple catch statements:

The syntax is

```
try
{
}
catch (type1 arg)
{
}
catch (type2 arg)
{
}
catch(typeN arg)
{
}
```

when an exception is thrown, the exception handlers are searched inorder for an appropriate match. The first handler that

yields a match is executed. After executing the handler, the control goes to the first statement after the last catch block for that try. when no matches found, the program is terminated. It is possible that the arguments of several catch statements match the type of exception. In such cases the first handler that matches the exception type is executed

```
void test (int x)
{
 try
  {
   if (x==1)
      throw x;
   else
      if (x==0)
       throw 'x';
      else
       if(x==-1)
        throw 1.0;
  }
 catch (char c)
 {
  cout <<" caught a character";
 }
```

```
    catch (int m)
    {
     cout << " caught an integer ";
     }
     catch (double d)
     {
      cout << " caught a double ";
      }
    }
void main()
{
 test(1);
 test(0);
 test(-1);
 test(2);
 }
```

Catch all exceptions:

The syntax is

```
catch (...)
{
=
}
```

```
void test (int x)
{
 try
 {
  if (x==1)
    throw x;
  else
   if (x==0)
    throw 'x';
   else
    if (x==-1)
     throw 1.0;
  }
 catch (...) //ellipse statement
 {
  cout << " caught the exception";
  }
}
 void main()
 {
  test(1);
  test(0);
  test(-1);
  }
```

Rethrowing an exception

A handler may decide to rethrow the exception caught without processing it. In such situations we may simply invoke throw without any arguments.

The syntax is

```
throw ;
```

This statement throw causes the current exception to be thrown to the next closing try catch sequence and caught by the catch statement listed after the enclosing try block. when an exception is rethrown it will not caught by the same catch statement (or) any other catch in that group rather it will be caught by an appropriate catch in the outer try or catch sequency only

```
void divide(double x, double y)
{try
 {
   if (y==0.0)
     throw y;
   else
     cout << x/y;
 }
catch (double)
{
 cout <<"caught double inside function";
 throw;
 }
 cout <<"End of function";
 }
void main()
{
 try
  {
   divide(10.5, 2.0);
   divide (20.0, 0.0);
  }
 catch (double)
  {
   cout <<" caught double inside main";
  }
}
```

Specifying an exception

It is possible to restrict a function to throw only certain specified exceptions. This is achieved by adding throw clist to the

function definition. The syntax is.

type fnname (arg list) throw (type list)

~~throw (type list)~~

{

fn body

}

The type list specifies the type of exceptions that may be thrown. Throwing anyother type of exceptions will lead to abnormal program termination. If we wish to prevent a function from throwing any exceptions we may do the type list as empty one i.e throw ();

A function can only be restricted in what type of exceptions it throws back to the try & block that called it. The restriction applies only when throwing an exception out of the function

void test (int) throw (int, double)

{

if (x==0) throw 'x';

else

if (x==1) throw x;

else

if (x==-1) throw 1.0;

}

void main()

{

try

{

test(0);

test(1);

test(-1);

test(2);

}

catch (char c)

{ cout << " caught a char";

}

catch (int m)

{

cout << " caughtan int";

}

catch (double d)

{ cout << " caught a double";

}

}

Managing Console I/O operations

Every program takes some data as inputs and generates process data as output. cin and cout operators are used for input/output operations. C++ supports a rich set of input/output functions. C++ use the concept of stream and stream classes to implement its I/O operations with the console and disk file. A stream is a sequence of bytes. It acts either as source from which the input data can be obtained or destination to which output data can be sent. The interface between the programmer and the actual device being accessed is known as stream.

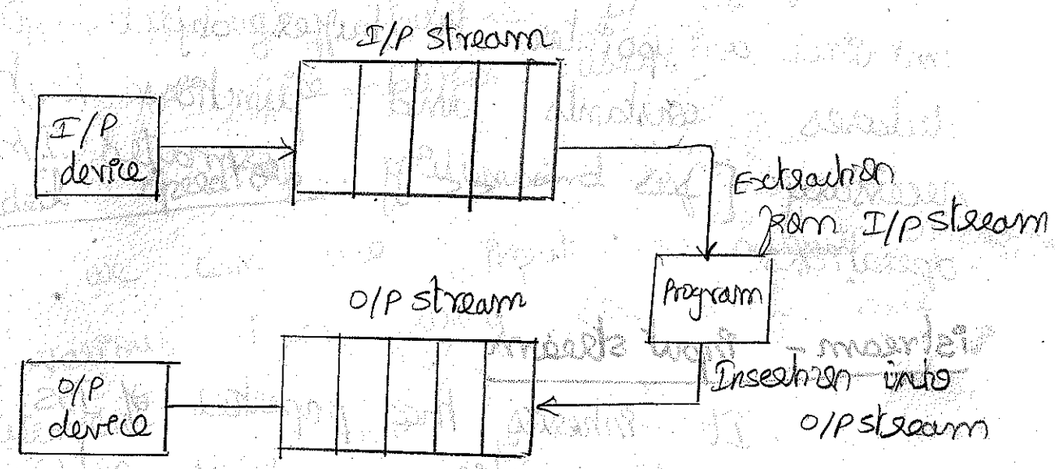
Input stream

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

output stream

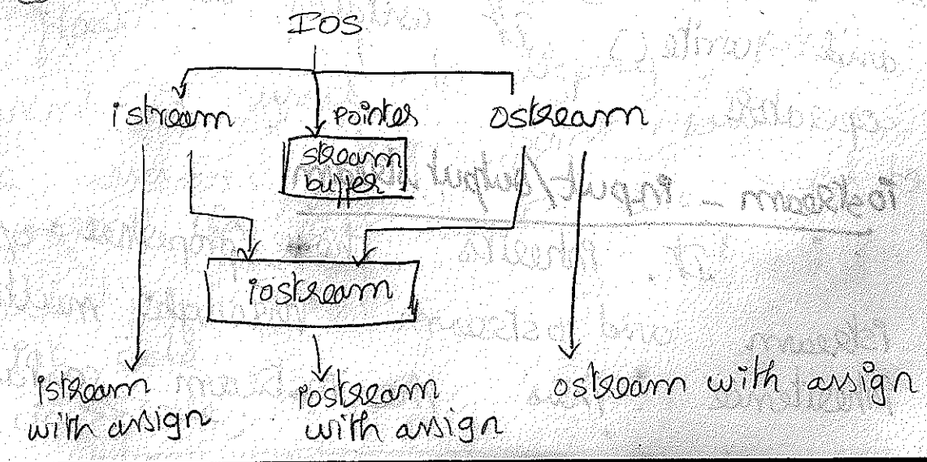
The destination stream that receives output from the program is called output stream.

The program extracts the bytes from input stream and inserts bytes into the output stream.



stream classes :

The C++ input/output system contains a hierarchy of classes that are used to define various streams to deal with both console and disk file. These classes are called stream classes.



Ios - Input output stream

It contains basic facilities that are used by all other I/O classes. It also contains a pointer to buffer object. It declares constants and functions that are necessary for handling formatted I/O operations

istream - input stream

It inherits the properties of Ios. It declares input functions such as get(), getline() and read(). It also contains overloaded extraction operator.

ostream - output stream

It also inherits the properties of Ios. It declares output functions such as put() and write(). It contains overloaded insertion operator

iostream - input/output stream

It inherits the properties of Ios, istream and ostream through multiple inheritence. Thus the iostream contains all the input/output functions.

stream buffer

It provides an interface to the physical devices through buffers. It act as base for file buffer used in Ios files

Overloaded operators : [>> and <<]

We can use input (or) output of data of various types using cin and cout statement.

```
int a;
float b;
cin >>a >>b;
cout <<a <<b;
```

A single << input statement i.e., cin is used to read an integer value and also floating point value and cout statement is used to display integer value and floating point value. so the operators cin and cout are said to be overloaded.

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

This cin statement is used to read the input value of the code. The cin statement reads character by character and assign the indicated location. The reading for a variable will be terminated at the encounter of a ~~code~~ white space or <enter> character that does not match the destination type

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

input: 4258D

Accepts 4258 only

put() and get()

The get() to fetch a character including blank space, tab and ~~newline~~ newline character

```
char c;
cin.get(c);
while (c!='\n')
{
cout << c;
cin.get(c);
}
```

The difference between cin() and cin statement

The cin statement is used to read a character but it will skip the widespaces and newline characters.

The get() is used to read a character but it will not skip the widespaces and new line characters.

```
void main()
{
int count = 0;
char c;
cin.get(c);
while (c!='\n')
{
cout.put(c);
count++;
}
cin.get(c);
cout << count;
}
```

input: CSE\n

cout<<count<<endl;

0123 | CSE

Difference between put() and cout statement

cout statement display all the characters It will skip wide spaces and newline characters.

put() displays character by character on the screen.

```
cout.put('x');
cout.put(ch);
cout.put(68);
```

The first put() is used to display the character
The second put() is used to display the value of the variable ch.
The third put() is used to display the ASCII value of 68.

getline() and write():

The getline() function reads the whole line of text that ends with newline character transmitted by return key. The function is invoked using object cin. Syntax is

```
cin.getline(line, size)
```

The function call invoke the function getline() which reads character input into the variable line. The reading is terminated as soon as either newline character is

\n or size-1 characters are read whichever occurs first.

Eg:
```
char name[10];
cin.getline(name, 10);
```

The syntax for write() is

```
cout.write(line, size)
```

The first argument, line represents the name of the string to be displayed. The second argument, size indicates the number of characters to display. It does not stop the displaying characters automatically. when the null character is encountered if the size is greater the length of line then it displays beyond the bounce of line.

Formatted console Input-output operation

C++ supports a number of features that could be used for formatting the output. This features include

1. Ios class functions and flags
2. manipulators
3. user-defined output functions

1. Ios class functions for formatting

(i) width()

To specify the required field size for displaying an output value.

(ii) precision()

To specify the number of digits to be displayed after the decimal point of a floating value.

(iii) fill()

To specify a character that is used to fill the unused portion of a field.

(iv) setf()

setf() i.e., setflag() to specify the format flags that control the form of output display either left justification or right justification.

(v) unsetf()

unsetf() i.e., unsetflag() to clear the flag specify.

2. manipulators:

Manipulators are special functions that can be included in the IO statements to alter the format parameters of a stream. For accessing manipulators the file iomanip should be included in the program.

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

width()

We can use the width() to define the width of a field necessary for the output of the variable. The syntax is

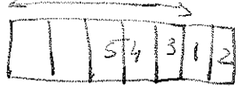
cout.width(w)

where w is a field width i.e; no of columns the output will be printed in a field of w characters wide at the right end of the field. The width() can specify the fieldwidth only one item. After printing the one

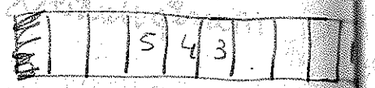
items, it will set back to the default

Eg:1
```
cout.width(5);
cout<<543<<12;
```

(right to left)
width



Eg:2
```
cout.width(5);
cout<<543
cout.width(5);
cout<<12
```



precision()

By default, the floating numbers are printed with six digits after the decimal point. The precision function can specify the number of digits to be displayed after the decimal point while printing the floating point numbers. The syntax is

```
cout.precision(d)
```

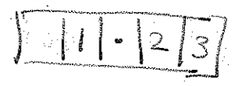
where d is the number of digits to the right of decimal point.

Eg:
```
cout.precision(3);
cout<<sqrt(2);
cout<<3.14159;
cout<<2.50032;
```

displays:
1.414
3.142
2.5

Unlike the function width, the precision retains the setting in effect until it is reset. That is any we have declared only one statement for the precision setting which is used by all the three outputs. We can also combine precision with width()

Eg:
```
cout.precision(2);
cout.width(5);
cout<<1.2345;
```

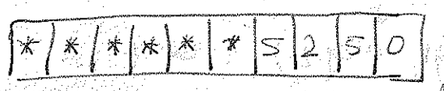


~~fill()~~ Filling and Padding

We can use the fill() to fill the unused positions by any desire character

Syntax is
```
cout.fill(ch);
```

Eg:
```
cout.fill('*');
cout.width(10);
cout<<5250;
```

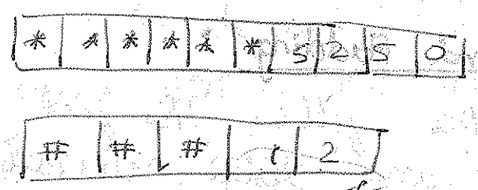


Financial institutions and banks use this kind of padding while printing cheques so that no one can change the amount easily.

like previous, fill() stay in effect till we change it.

Eg:

```
cout.fill('*');
cout.width(10);
cout<<5250;
cout.fill('#');
cout.width(5);
cout<<12;
```



Formatting flags bit field and setf()

The width() is used to display the value in right justified manner. But in usual conditions the characters are displayed in left justified only. The setf function can be used for printing the value in right justified (or) left justified. The syntax is

cout.setf(arg1, arg2)

where arg1 is one of the formatting flag defined in the class ios. The arg2 is known as bitfield to specify the group to which the formatting flag belongs.

Format required	flag (arg1)	Bitfield (arg2)
left justified output	ios::left	ios::adjustfield
right justified	ios::right	ios::adjustfield
Padding after sign (or) base indicator	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.

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

Displaying trailing zeros and plus sign

```
cout.width(8);
cout.precision(2);
cout << 10.75;
cout << 25.00;
cout << 15.50;
```

			1	0	.	7	5
						2	5
			1	5	.	5	

Here trailing zeros are avoided. The setflag can be used with the flag ios::showpoint as a single argument to achieve the output with trailing zeros

```
cout.width(8);
cout.precision(2);
⇒ cout.setf(ios::showpoint);
cout << 10.75;
cout << 25.00;
cout << 15.50;
```

			1	0	.	7
			2	5	.	0
			1	5	.	5

plus sign can be printed before a positive number using the statement

```
cout.setf(ios::showpos);
```



```
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;
```

+			2	7	5	.	5	0	0

ios::showbase
use base indicator on output

ios::showpos
print + before positive numbers

ios::showpoint
show trailing decimal point and zeros

ios::uppercase
use uppercase letters for hexadecimal output

ios::skipws
skip white space on input

ios::unitbuf
flush all the streams after insertion

ios::stdio

flush standard put and standard error after insertion.

Managing output with manipulators:

cout << setw(5) << 12345

1	2	3	4	5

cout << setw(10) << setiosflags (ios::left) << 12345;

1	2	3	4	5					

Designing our own manipulators

We can design our own manipulators for certain special purposes. Syntax is

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

Here the manipulator is the name of manipulator under creation

Eg:

```
ostream & unit (ostream & output)
{
output << "inches";
return output;
}
cout << 36 << unit;
}
```

36 inches

```
ostream & show (ostream & output)
{
output.setf (ios::showpoint);
output.setf (ios::showpos);
output << setw(10);
return output;
}
```

+	7	8	5	6	.	5	0	0	0

```
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;
    }
void main()
{
 cout << currency << form << 7864.5;
}
```

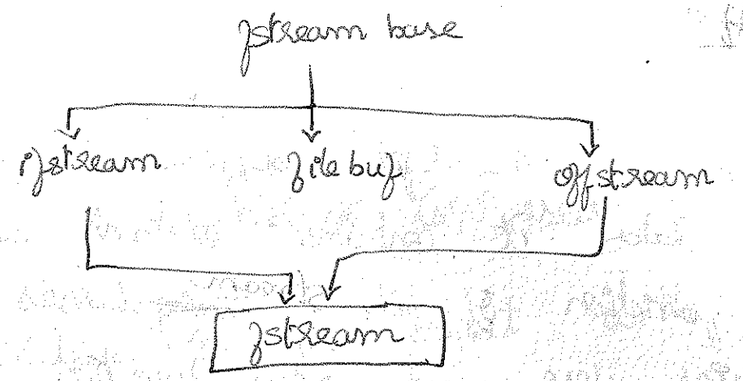
Rs	*	+	7	8	6	4	.	5	0

FILE

A file is a collection of ~~relate~~ related data stored in a particular area on the disk. Programs can be designed to perform read and write operation on these files. A program involves either or both of the following kinds of data communication.

1. Data transfer between console unit and the program.
2. Data transfer between program and disk files.

classes for file stream operations



fstream base

It is similar to ios. It provides the operations common to file streams. It act as base for fstream, ifstream and ofstream. It contains open and close function

ifstream

It is similar to istream. It provides input operations. It contains open function with default input mode. It inherits the functions get(), getline() and read() and seekg() and tellg(). Functions from istream

ofstream

It is similar to ostream. It provides output operations. It contains open function with default output mode. It inherits the properties from ostream i.e; put(), seekp(), tellp(), write() from ostream file

filebuffer:

It is similar to streambuffer. It's purpose is to set the file buffers to read and write data. It contains constant used in open function of file stream ~~string~~ classes. It also contains close and open function as member and member fns

opening and closing of a file:

For opening of a file we must create a filestream and then link it to file name A file stream can be defined using the classes ifstream, ofstream and fstream that are included in the header file fstream. The file can be opened in two ways

1. using constructor function of a class
2. using the member function open() of the class

opening files using constructor:

It is useful if we use only one file in the stream. The constructor is used initialise an object while it is being created. Here the filename is used to initialise the file stream object. This involves the following steps.

1. create a filestream object to manage the stream using the appropriate class i.e. the class ofstream is used to create the output stream and the class ifstream is used to create input stream.
2. Initialize the file object with the desired filename

```
void main()
{
ofstream outf("Item");
char name[30];
cin>>name;
outf<<name;
float cost;
cin>>cost;
outf<<cost;
outf.close();
ifstream inf("Item");
inf>>name>>cost;
cout<<name<<cost;
inf.close();
}
```

outf → to store data in file
inf → to fetch data from file

when a file is opened for writing only a new file is created if there is no file of that name. If a file by that name

exists already, then its contents are deleted and the file is presented as a clean file i.e; without any data

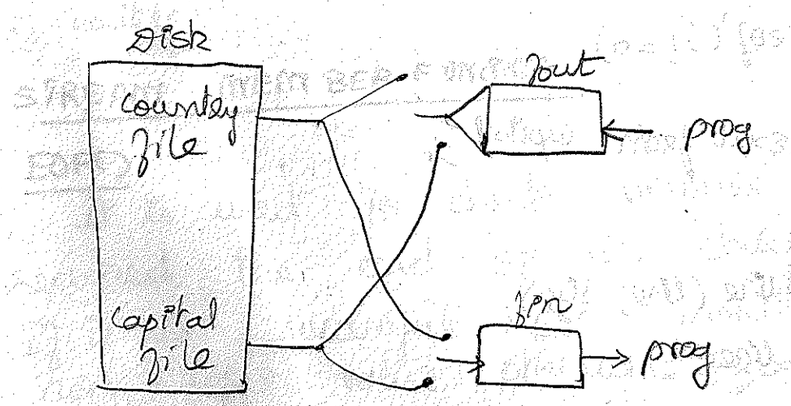
opening files using open()

If we want to manage multiple files using a single ~~string~~ stream, the open() member function is used for file opening. The function open can be used to open multiple files that use the same stream object. Syntax is

```
filestreamclass streamobject;
streamobject.open("filename");
```

```
void main()
{
 ofstream fout;
 fout.open("country");
 fout<<"US"<<"UK";
 fout.close();
 fout.open("capital");
 fout<<"washington"<<"London";
 fout.close();
 const int n=80;
 char line[n];
  ifstream fin;
```

```
fin.open("country");
while(fin)
{
 fin.getline(line,n);
 cout<<line;
}
fin.close();
fin.open("capital");
while(fin)
{
fin.getline(line,n);
cout<<line;
}
fin.close();
}
```



```
void main()
{
const int size = 80
char line[size];
ifstream fin1, fin2;
fin1.open ("country");
fin2.open("capital");
for(int i=1; i<=10; i++)
{
if(fin1.eof()!=0)
{
cout << "Exit from country";
exit(1);
}
fin1.getline(line, size)
cout << line;
if(fin2.eof()!=0)
{
cout <<"exit from capital";
exit(1);
}
fin2.getline(line, size);
cout << line;
}
fin1.close();
fin2.close();
}
```

Reduction of end of file

An ifstream object such as fin returns a value of zero if an error occurs in the file operation including the end of file condition. Thus the while loop terminates when fin.eof() returns a value of zero on reaching the end of file condition. eof i.e., end of file is a member function of ios class. It returns a non-zero value if the end of file condition is encountered and zero otherwise.

The difference between exit(0) and exit(1):

exit(1) for successful operation. If any error occurs during operation, it returns a zero value i.e. exit(0)

STREAM MEMBER FUNCTION

EOF()

It is used to check whether file pointer has reached the end of file character or not. If it is successful EOF function returns a non-zero value otherwise zero.

The second function: **fail()** is used to check whether the file has been opened for input or output successfully or any invalid

operators are attempted or there is an unrecoverable error. If it fails, it returns a non-zero character.

3) **bad()** : It is used to check whether any invalid file operations has been attempted or there is an unrecoverable error. It returns a non-zero value for true condition otherwise return zero

4) **good()** : It is used to check whether the previous file operation has been successful or not. It returns a non-zero if all stream state bits are zero.

Lab) write a c++ program to convert the lower case character to an upper case character using files.

```
#include <fstream.h>
#include <iostream.h>
#include <iomanip.h>
#include <ctype.h>
void main()
{
  ofstream outfile;   // stream object
  ifstream infile;
  char fname1[10], fname2[10];
  char ch, uch;
  cin >> fname1 >> fname2;
  infile.open(fname1)
  if (infile.fail())
  {
  cout << "no such file exist";
  exit(1);
  }
  outfile.open(fname2)
  if( outfile.fail())
  {
  cout << "unable to create a file";
  exit(1);
  }
  while (!infile.eof())
  {
  ch = (char) infile.get();
  uch = toupper(ch);
  outfile.put(uch);
  }
  infile.close();
  outfile.close()
  }
```

FILE MODES

The open() is used to create new files as well as to open the existing file. The open() takes two arguments

1. Filename
2. File mode

The syntax of open() with arguments is

```
stream object . open("filename", mode);
```

If we use only one argument in the open() the second argument is by default values. The default values are

ios::in ;

ios::out ;

Eg:-

infile.open("abc", ios::in);

The file abc is opened for input mode

infile.open("abc")

The file abc is opened in input mode by default

File mode parameters

1) **ios::app** Append to the end of file

2) **ios::ate** Go to the end of file ~~and~~ or opening

3) **ios::binary** Binary file

4) **ios::in** open file for reading only

5) **ios::nocreate** open ~~files~~ fails if the file does not exist

6) **ios::noreplace** open files if the file already exists

7) **ios::out** open file for writing only

8) **ios::trunc** delete the contents of file if exists

opening a file in ios::out mode also opens it in the ios::trunc mode by default

Both ios::app and ios::ate takes the end of file when it is opened. The difference between the two parameters is that ios::app allows to add data to the end of file only where ios::ate permits to add data or modify the existing data anywhere in the file. In both the cases the file is created by the specified name if it does not exist. The parameter ios::app can be used only the output file.

~~Creating a stream using ifstream~~

Creating a stream using ifstream implies input file and creating a stream using ofstream implies output. so in this case it is not necessary to provide the mode parameters. The fstream class does not provide a mode by default and therefore we must provide the mode explicitly when using the object of fstream class. The mode can be combined two or more parameters using bitwise OR operator as follows

fout.open("date", ios::app || ios::nocreate);

A set of general purpose template classes (data structure) and functions (algorithms) that could be use as a std approach for storing and processing data.

The collection of these generic classes and fns called standard template library (STL). The STL becomes a part of ANSI C++.

Using STL we can save a considerable time and effort and lead to high quality programs.

Components of STL :-

There are three basic key components available in STL.

i) Container

ii) Algorithms

iii) Iterator

Container :-

It is an object that actually stores data. It is a way a data is organised in memory. The containers are implemented by template classes and therefore, it can be easily customised to hold different types of data.

Algorithm :-

It is a procedure that is used to process the

deque	A double ended queue, allows insertion & deletions at both ends, permits direct access to the elements	<deque>	random access.
set	An associate container for storing non-unique quaeuset noduplicates are also allowed.	<set>	bidirectional
multiset	An associate container for storing non-unique sets duplicates allowed.	<set>	bidirectional
map	An associate container for storing unique key or unique value pairs each key is associated with one value (i.e) one to one mapping	<map>	bidirectional
multimap	An associate container for storing key pairs in which one key may be associated with more than one value (ie) one to many mapping.	<map>	bidirectional
stack	last in first out	<stack>	no iterator.

data contained in containers. It includes different kinds of algorithms to provide support to task such as initializing, searching, sorting. Algorithms are implemented by template functions. Iterator

Iterator:-

It is an object like pointer that points to an element in a container. We can use iterators to move through the contents of the containers. We can also increment / decrement the iterators. Iterator connects the algorithms with containers and play a key role in the manipulation of data stored in a container.

The containers supported by

Containes	Description	header file	iterator
Vector	It is a dynamic array, allow insertion & deletion at the back, permits direct access to an element	<vector>	random access
list	It is a bidirectional linear list, it allows insertion & deletion anywhere	<list>	bidirectional

5) Set algorithms

6) Relational "

Adjacent find()

It finds the adjacent pair of objects that are equal

Count()

It counts the occurance of a value in a sequence.

Count-if()

Count the number of elements that matches a predicate

equal()

It returns true if two ranges are same.

find()

Find the 1st occurance of a value in a sequence.

find_end()

Find last occurance of a value in a sequence.

find_first_of()

Finds a value in from one sequence in another

find_if()

It finds the 1st match of a predicate in a sequence.

for_each()

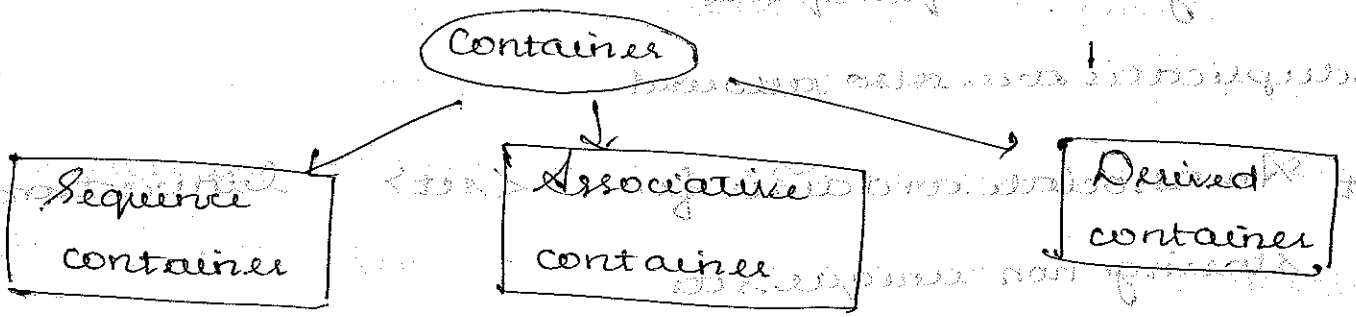
Apply an operation to each element

mismatch()

finds first element for which two sequences differ

queue	First in first out	<queue>	no iterator
priority queue	The first element is always is highest priority element.	<queue>	no iterator.

The components of containers.



vector
dequeue
list

set
multiset
map
multimap

stack
queue
priority queue.

Algorithms:-

Algorithms are fns that can be used generally across a variety of containers for processing the contents, although each container provides functions for its basic operations.

Based on the nature of operations, the algorithms are classified as follows.

i) Retrieve / Non-mutating

ii) Mutating algorithms

iii) Sorting "

Sorting algorithm:

binary_search()

equal_range

inplace_merge()

lower_bound()

make_heap()

merge()

nth_element()

partial_sort()

partial_sort_copy()

partition()

pop_heap()

push_heap()

sort()

sort_heap()

stable_partition()

stable_sort()

upper_bound()

Set algorithms:-

includes()

set_difference()

set_intersection()

set_symmetric_difference()

set_union()

Numeric algorithm:-

accumulate()

adjacent_difference

inner_product()

partial_sum()

Relational Algo:

equal()

lexicographical_compare()

max()

max_element()

min()

min_element()

mismatch()

search()

finds the 1st a subsequence with a sequence.

search_n()

find a sequence of a specified no of elements.

Mutating algorithm:-

copy()

copy_backward()

fill()

fill_n()

generate()

generate_n()

iter_swap()

random_shuffle()

remove()

remove_copy()

remove_copy_if()

remove_if()

replace()

replace_copy()

replace_copy_if()

replace_if()

reverse()

reverse_copy()

rotate()

rotate_copy()

swap()

swap_ranges()

transform()

unique()

unique_copy()

It behaves like pointer, used to access the container elements, they are often used to traverse from one element to another. This process is called iterating to the container.

iterator	Access method	Direction of movement	I/O capability	Remarks
input ptr	linear	forward only	read only	cannot be saved
output ptr	linear	forward only	write only	cannot be saved
Forward	linear	forward only	read / write	can be saved.
bidirectional	linear	forward & backw	read / write	can be saved
random	random	forw & backward	read / write	can be saved.

Operations supported by the operator:-

Iterator	element access	Read	Write	Increment operation	Comparison
I/p	⟷	V=*p		++	!=, ==
O/p			*P=V	++	
forward	⟶	V=*p	*P=V	++	==, !=
bidirectional	⟶	V=*p	*P=V	++, --	==, !=
random	⟶, []	V=*P	*P=V	++, --, +, -, +=, -=, ==, !=, <, >	
				+=, -+	<=, >=