

# CS-15

# C++ and Object Oriented Programming

## Unit – 5

Presented By : Dhruvita Savaliya

**Working with Files,  
Exception Handling,  
Introduction to Template STL**

# Topics :

- File Stream Classes
- Opening and closing a file
- Error Handling
- File Modes
- File Pointers
- Sequential I/O operations
- Updating a file (Random access)
- Command Line Arguments
- Overview of Exception Handling
  - Need for Exception Handling
  - various components of exception handling
- Introduction to templates
  - Class templates and Function templates
  - Member function templates
  - Overloading of template function
  - Non-type Template argument
- Introduction to STL
  - Overview of iterators, containers

# Introduction :

- As a computer programmer, we have sufficient knowledge of data storage.
- We can store data in computer for future use. We can store and retrieve data as our requirement.
- In C++ files are the best way to read and write data.
- In C++ we have many useful functionalities to handle files.

- File handling is used to store data permanently in a computer.
- Using file handling we can store our data in secondary memory (Hard disk).

How to achieve the File Handling

For achieving file handling we need to follow the following steps:-

STEP 1-Naming a file

STEP 2-Opening a file

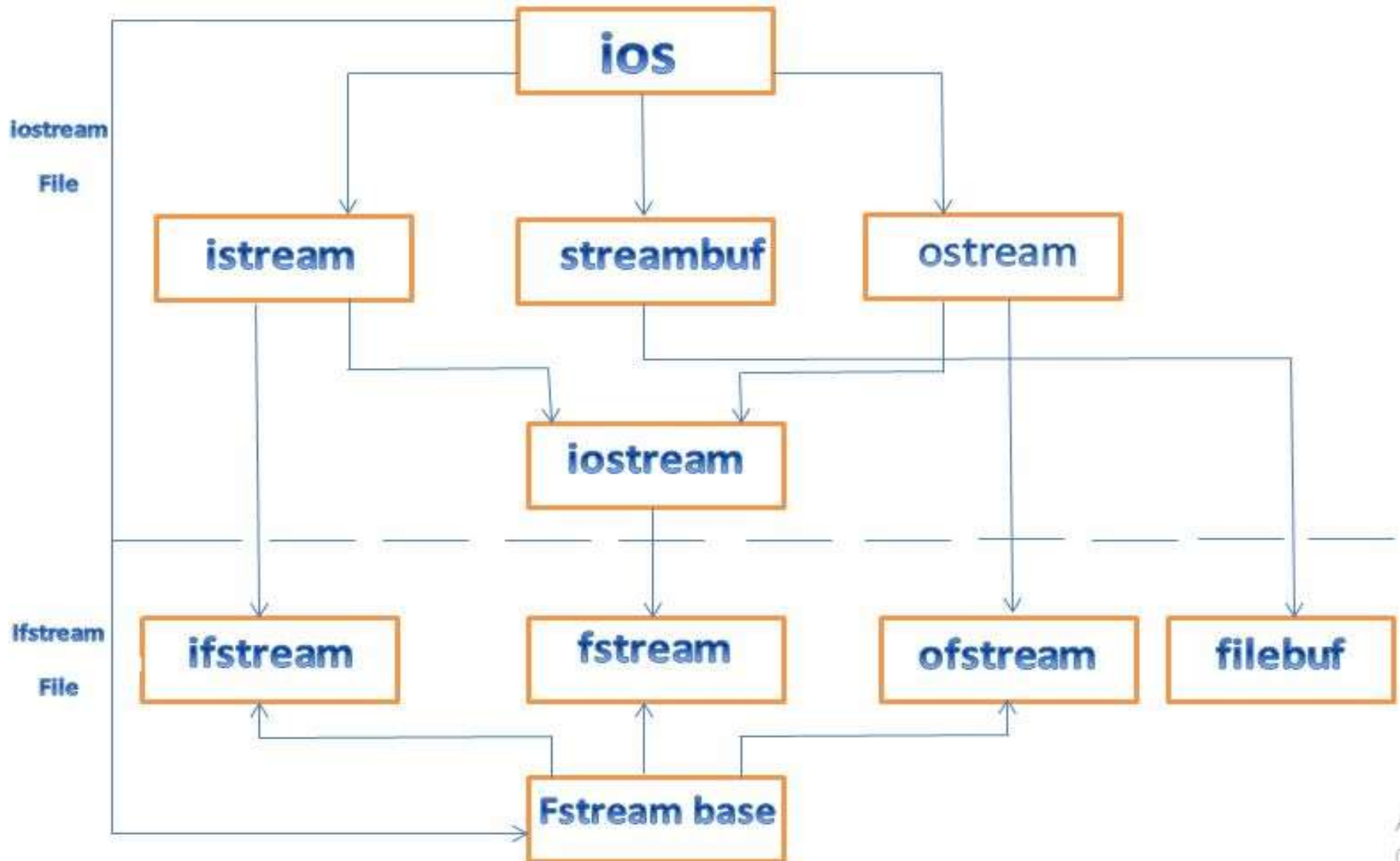
STEP 3-Writing data into the file

STEP 4-Reading data from the file

STEP 5-Closing a file.

# File Stream Class :

- The I/O system of C++ contains a set of classes which define the file handling methods.
- These include ifstream, ofstream and fstream classes.
- These classes are derived from fstream and from the corresponding iostream class.
- These classes, designed to manage the disk files, are declared in fstream and therefore we must include this file in any program that uses files.
- File handling is essential for data storage and retrieval in applications



# Classes of file stream :

## 1. ios:-

- ios stands for input output stream.
- This class is the base class for other classes in this class hierarchy.
- This class contains the necessary facilities that are used by all the other derived classes for input and output operations.

## 2. istream:-

- istream stands for input stream.
- This class is derived from the class 'ios'.
- This class handle input stream.
- The extraction operator(>>) is overloaded in this class to handle input streams from files to the program execution.
- This class declares input functions such as get(), getline() and read().

## 3. ostream:-

- ostream stands for output stream.
- This class is derived from the class 'ios'.
- This class handle output stream.
- The insertion operator(<<) is overloaded in this class to handle output streams to files from the program execution.
- This class declares output functions such as put() and write().

## 4. streambuf:-

- This class contains a pointer which points to the buffer which is used to manage the input and output streams.

**This classes are used with file handling :**

**5. fstream:-**

- This class provides operations common to the file streams. Serves as a base for fstream, ifstream and ofstream class.
- This class contains open() and close() function.

**6. ifstream:-**

- This class provides input operations.
- It contains open() function with default input mode.
- Inherits the functions get(), getline(), read(), seekg() and tellg() functions from the istream.

**7. ofstream:-**

- This class provides output operations.
- It contains open() function with default output mode.
- Inherits the functions put(), write(), seekp() and tellp() functions from the ostream.

**8. fstream:-**

- This class provides support for simultaneous input and output operations.
- Inherits all the functions from istream and ostream classes through iostream.

**9. filebuf:-**

- Its purpose is to set the file buffers to read and write.
- We can also use file buffer member function to determine the length of the file.



# Opening and Closing Files :

- A file must be opened before you can read from it or write to it.
- Either **ofstream** or **fstream** object may be used to open a file for writing.
- **ifstream** object is used to open a file for reading purpose only.
- **Opening a File.**

1. Using the open( ) function.
2. Using the constructor.

## 1. Using the open( ) function

- **Syntax :**

**fileObject.open(“*File Name*”,*mode*);**

- To open a file using object we can use open( ) function.

- **Example :**

```
ofstream file;    //here file is object of ofstream class  
file.open(“MyFirstFile.txt”);
```

## 2. Using constructor :

- ofstream file(“abc.txt” ); or ifstream file(“abc.txt”);

- **Closing a file :**

- We have to close the object before open another file object.

- **Syntax :**

```
fileObj.close( );
```

- **Example :**

```
ofstream file;
```

```
file.open("MyFirstFile.txt");
```

```
file.close();
```

- **Opening multiple files :**

- We can open more than one data file in a single file.

- But we have to close opened file fist.

- **Example :**

```
ofstream File;
```

```
File.open("File1.txt");
```

```
...
```

```
File.close();
```

```
File.open("File2.txt");
```

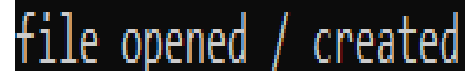
```
...
```

```
File.close();
```

## Example of opening and closing file :

```
#include<iostream.h>
#include<conio.h>
#include<fstream.h>
using namespace std;

void main( )
{
    ofstream file;
    file.open("abc.txt");
    //file.open("E:\\sybca\\file_open_close\\abc.txt");
    if (file)
    {
        cout << "file opened / created..";
    }
    else
    {
        cout << "Something wrong to open file";
    }
    file.close( );
    getch( );
    return 0;
}
```



## Reading from txt file :

```
#include<iostream>
#include<conio.h>
#include<fstream>
#include<string>
using namespace std;

int main( )
{
    string data;
    ifstream file;
    file.open("abc.txt");
    if (file)
    {
        cout << "file opened";
    }
}
```

```
else
{
    cout << "Something
    wrong to open file";
}
while (!file.eof( ))
{
    getline(file,data);
    cout <<endl<< data;
}
file.close( );
getch( );
return 0;
}
```

```
file opened
hello
welcome to the bca
```

**Writing to a file :**

```
#include<iostream>
#include<conio.h>
#include<fstream>
using namespace std;

int main()
{
    ofstream file;
    file.open("abc.txt");
    if (file)
    {
        cout << "file opened";
    }
    else
    {
        cout << "Something wrong to open file";
    }
    file << "hello";
    file << endl << "sya";
    file << endl << "syb";
    file.close();
    getch();
    return 0;
}
```

```
char c = 'A';
for (int x = 1; x <= 26; x++)
{
    file.put(c);
    c++;
}
```

It will write A to Z in abc.txt file

file opened

abc - Notepad

File Edit Format View Help

hello  
sya  
syb

## Write into a file from get values from user :

```
#include<iostream>
#include<conio.h>
#include<fstream>
#include<string>
using namespace std;

int main()
{
    string data;
    fstream file;
    file.open("abc.txt");
    if (file)
    {
        cout << "file opened";
    }
    else
    {
        cout << "Something wrong to open file";
    }
    getline(cin, data, '$');
    cout << data;

    file << data;

    file.close();
    getch();
    return 0;
}
```

```
file opened
hello
welcome

to sybca
$

hello
welcome

to sybca
```

abc - Notepad

File Edit Format View Help

```
hello
welcome

to sybca
```

# File Mode :

Member Constant	Stands For	Access
ios::in	input	File open for reading: the internal stream buffer supports input operations.
ios::out	output	File open for writing: the internal stream buffer supports output operations.
ios::binary	binary	Operations are performed in binary mode rather than text.
ios::ate	at end	The output position starts at the end of the file.
ios::app	append	All output operations happen at the end of the file, appending to its existing contents.
ios::trunc	truncate	Any contents that existed in the file before it is open are discarded.
ios::nocreate	Do not create	Does not allow to create new file if it does not exist.
ios::noreplace	Do not replace	Does not replace old file with new file.

- Both `ios::app` and `ios::ate` take us to the end of the file when it is opened.
- The difference between the two modes is that `ios::app` allow us to add data to the end of the file only, while `ios::ate` mode permits us add data or to modify the existing data anywhere in the file.
- We can combine more than one file modes using pipe symbol or bitwise OR operator (`|`).
- **Fstream File;**
- **`File.open("Res.txt",ios::out|ios::nocreate);`**
- **Default Open Modes :**
- this modes are optional with this this classes : →

ifstream	ios::in
ofstream	ios::out
fstream	ios::in   ios::out



## Example of append mode :

```
#include <iostream>
#include <fstream>
```

```
using namespace std;
```

```
int main() {
```

```
// open a text file for appending
ofstream my_file("abc.txt", ios::app);
```


```
// if the file doesn't open successfully, print an error message
if (!my_file) {
    cout << "Failed to open the file for appending." << endl;
    return 1;
```

```
}
// append multiple lines to the file
my_file << endl << "Line 4" << endl;
my_file << "Line 5" << endl;
my_file << "Line 6" << endl;
```

```
// close the file
my_file.close();
```

```
return 0;
}
```


## Before execution :

 abc - Notepad

File Edit Format View Help

hello  
welcome|

## After execution :

 abc - Notepad

File Edit Format View Help

hello  
welcome  
Line 4  
Line 5  
Line 6

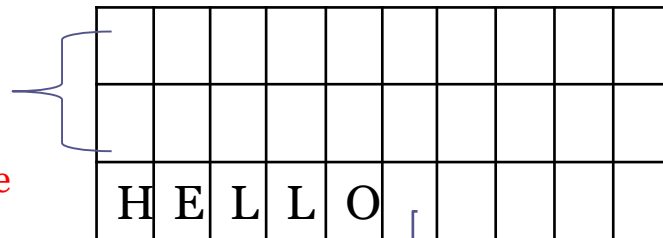
# File Pointers :

- The file pointers are the pointers that are used to set or get the position of the cursor in the file while reading or writing a file.
- There are two file pointers associated with each file known as input pointer and output pointer which is also known as `get_pointer` and `put_pointer`.
- The input pointer is used to get or set the location of the pointer while reading a file and the output pointer is used to get or set the location of the pointer while writing a file.
- The position of file pointers is set automatically when a file is opened and gets read or written.
- A pointer is used to handle and keep track of the files being accessed.

- **We can open files in 3 modes :**

1. Reading mode
2. Writing mode
3. Append mode

Take 1<sup>st</sup>  
Position of  
reading –  
writing mode



Take 6<sup>th</sup> position of  
append mode

<b>Function Nam</b>	<b>Syntax</b>	<b>Use</b>
seekg()	void seekg(offset, from)	Seek get pointer, Moves the input pointer to the specified location.
seekp()	Void seekp(offset,from)	Seek put pointer, Moves the output pointer to the specified location.
tellg()	int tellg()	Tell get pointer, Returns the current position of the input pointer.
tellp()	int tellp()	Tell put pointer, Returns the current position of the output pointer.

# seekg() & seekp()

- We can reposition the file-position pointer in istream and ostream using its special member functions.
- These member functions are 'seekg' and 'seekp'.
- 'seekg' or 'seek get' is used for istream and 'seekp' or 'seek put' is used for ostream.
- Both these member functions take long integer as arguments.
- A second argument is used to specify the direction of seek.
- **Navigate Pointer / seek directions / reference point :**
  1. ios::beg // refers to the beginning of the file
  2. ios::cur // refers to the current position in the file
  3. ios::end // refers to the end of the file

- **seekg()** is used to move the get pointer to the desired location concerning a reference point.
- **Syntax:** `file_pointer.seekg(number of bytes ,Reference point);`
- **Example:**  
`fin.seekg(10,ios::beg);`  
*File.seekg(0); // move to first character*
- **Exampe :**  

```
#include<iostream>
#include<conio.h>
#include<fstream>
#include<string>
using namespace std;
```

```
int main()
{
    string data;
    ifstream file;
    file.open("abc.txt");
    file.seekg(4); //start read from 4th position.
    while (!file.eof())
    {
        getline(file, data);
        cout << endl << data;
    }
    file.close();
    getch();
    return 0;
}
```



abc - Notepad

File Edit Format View Help

|ABCDEFGHIJKLMNPOQRSTUVWXYZ

```
file opened
EFGHIJKLMNPOQRSTUVWXYZ
```

- **seekp()** is used to move the put pointer to the desired location concerning a reference point.

- **Syntax:**

```
file_pointer.seekp(number of
bytes ,Reference point);
```

- **Example:**

```
fout.seekp(10,ios::beg);
```

### Example of seekp( ) :

```
#include <iostream>
```

```
#include <fstream>
```

```
#include <string>
```

```
#include <conio.h>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    ofstream file; // To WRITE data to file...
```

```
file.open("example.txt");
```

```
// Write some initial data
```

```
file << "Hello, World!";
```

```
// Use seekp to set the position to overwrite the 7th
character
```

```
file.seekp(7);
```

```
// Move to the 7th character position
```

```
// Write new data at 7th position
```

```
file << "C++";
```


```
file.close();
```

```
getch();
```

```
return 0;
```

```
}
```

It will store hello, c++ld!  
Instead of hello, world!

 example - Notepad

File Edit Format View Help  
Hello, C++ld!

- **tellg():**
- tellg() is used to realize which they get pointer is in a file.
- **Syntax:** file\_pointer.tellg();
- **Example:** int posn = fin.tellg();
- **Example :**

```
#include <iostream>
#include <fstream>
#include <conio.h>
using namespace std;
int main()
{
    ifstream file("example.txt");

    if (!file) {
        cout << "Unable to open file" <<
endl;
        getch();
        return 1;
    }

    // Read first character
    char ch;
```


```
file.get(ch);
cout << "First character: " << ch <<
endl;

// Get current position
//streampos position = file.tellg();
int position = file.tellg();
cout << "Current position: " <<
position << endl;

// Read next character
file.get(ch);
cout << "Second character: " << ch <<
endl;

// Get new position
position = file.tellg();
cout << "New position: " << position
<< endl;

file.close();
getch();
return 0;
}
```

 example - Notepad

File Edit Format View Help

Hello, world!

```
First character: H
Current position: 1
Second character: e
New position: 2
```

- **tellp():**

tellp() is used to realize which the pointer is in a file.

- **Syntax:** file\_pointer.tellp();

- **Example:** int posn=fout.tellp();

```
#include <iostream>
#include <fstream>
#include <conio.h>
using namespace std;
int main()
{
    ofstream outFile("example.txt");
    //fstream outFile("example.txt");
    if (!outFile) {
        cout << "Error opening file!";
        return 1;
    }
```

```
outFile << "Hello, World!" << endl;
```

```
//outFile.seekg(5);//it not use in ofstream
```

```
// Get the current position of the put pointer
```

```
streampos position = outFile.tellp();
```

```
cout << "Current position in the file:
" << position << endl;
```

```
outFile << "This is a test." << endl;
```

```
// Check position again
```

```
position = outFile.tellp();
```

```
cout << "Current position after
writing more: " << position <<
endl;
```

```
outFile.close();
```

```
getch();
```

```
return 0;
```

```
}
```

```
Current position in the file: 15
Current position after writing more: 32
```

```
Hello, World!
This is a test.
```



## ❖ End of File :

- While reading a file we need to detect the end of file to prevent reading after the last byte of file. We can detect the end of file by two ways...
- Using while loop and Using eof( ) function

### 1. Using while loop :

- We can use object of input file stream in while loop.
- **Example :**  

```
ifstream File;
while (File) {           //Work until end of file position      }
```
- Here, the object File returns 0 if any error occurs during the file reading such as end of file otherwise it returns non-zero value.

### 2. Using eof() function :

- We can use eof() function to detect **end of file which is a member function of ios class.**
- **It returns a non-zero value if the file pointer is reached at the end of file during read operation otherwise it returns zero.**
- ```
ifstream File;
```
- ```
if(File.eof()!=0){           //Work until end of file position      }
```

# Sequential I/O Operations :

- In sequential input/output operations, that data is read or written in sequential manner, one character after another.
- To do so we have a pair of functions `get()` and `put()` to read and write characters (one by one) sequentially from and to a file respectively.
- C++ supports `read()` and `write()` function to read and `write()` more bytes of binary data at a time.

```

#include<iostream>
#include<conio.h>
#include<fstream>
#include<string>
using namespace std;

int main()
{
    char data[30];
    fstream file;
    file.open("abc.txt",ios::in|ios::out);
    if (file)
    {
        cout << "file opened";
    }
    else
    {
        cout << "Something wrong to open
            file";
    }
}

```

```

}
cin.getline(data,30);
for (int x = 0; x <strlen(data); x++)
{
    file.put(data[x]);
}
file.seekg(0);//to move 1st position

char a;

while (file){
    file.get(a);
    cout << a;
}
file.close();
getch();
return 0;
}

```

- **The read() and write() functions**
- The read() and write() functions are used to read and write blocks of data of a binary file.
- **Syntax :**  
inputFileObj.read((char \*)&arr, sizeof(arr));  
outputFileObj.write((char \*)&arr, sizeof(arr));
- The read() and write() functions can also be used to read and write class objects.
- So this can be helpful to write the object's properties on the disk file which is stored permanently.
- But remember that only the member variables of the class are written to the file and not the member functions.

# Error Handling during File Operations :

- It's quite common that errors may occur during file operations. There may have different reasons for arising errors while working with files. The following are the common problems that lead to errors during file operations.
  1. When trying to open a file for reading might not exist.
  2. When trying to read from a file beyond its total number of characters.
  3. When trying to perform a read operation from a file that has opened in write mode.
  4. When trying to perform a write operation on a file that has opened in reading mode.
  5. When trying to operate on a file that has not been open.

<b>Function</b>	<b>Syntax</b>	<b>Return Value</b>
bad()	int bad()	It returns a non-zero (true) value if an invalid operation is attempted or an unrecoverable error has occurred. Returns zero if it may be possible to recover from any other error reported and continue operations.
fail()	int fail( )	It returns a non-zero (true) value when an input or output operation has failed.
good()	int good()	It returns a non-zero (true) value when no error has occurred; otherwise returns zero (false).
eof()	int eof( )	It returns a non-zero (true) value when end-of-file is encountered while reading; otherwise returns zero (false).

**File.bad() :**

```
#include<iostream>
#include<conio.h>
#include<fstream>
#include<string>
using namespace std;

int main()
{
    fstream file;
    file.open("my_file.txt", ios::out);

    string data;
    file >> data;
    //we are try to read from a file
    //The file mode is written
```

```
if (!file.bad()){
    cout << "Operation not success!!!"
        << endl;
}
else {
    cout << "Data read from file - " <<
        data << endl;
}
getch();
return 0;
}
```

```
Operation not success!!!
```

**File.fail() :**

```
#include<iostream>
#include<conio.h>
#include<fstream>
#include<string>
using namespace std;
```

```
int main()
{
    fstream file;
    file.open("my_file.txt", ios::out);
```

```
string data;
```

```
file >> data;
```

```
//we are try to read from a file
```

```
//The file mode is written
```

```
if (file.fail()){
    cout << "Operation not success!!!"
        << endl;
}
else {
    cout << "Data read from file - " <<
        data << endl;
}
getch();
return 0;
}
```

```
Operation not success!!!
```



## Reading from txt file :

```
#include<iostream>
#include<conio.h>
#include<fstream>
#include<string>
using namespace std;

int main( )
{
    string data;
    ifstream file;
    file.open("abc.txt");
    if (file)
    {
        cout << "file opened";
```

```
    }
    else
    {
        cout << "Something wrong
        to open file";
    }
    while (!file.eof( ))
    {
        getline(file,data);
        cout <<endl<< data;
    }
    file.close( );
    getch( );
    return 0;
}
```

```
file opened
hello
welcome to the bca
```

```
#include <iostream>
#include <fstream>
#include <conio.h>
using namespace std;
int main()
{
    fstream file;
    file.open("my_file.txt", ios::in);
    cout << "goodbit: " << file.good() << endl;
    getch();
    return 0;
}
```

```
goodbit: 1
```

# Update a file (Random Access) :

- In sequential file, it becomes difficult to update some portion of the file.
- In sequential access method, you have to read the entire file from the first byte to the last byte of the file.
- While in case of random access method you can move to any location and update that portion of the file.
- The best example of sequential access method is old type cassettes. In the tape cassette you have to fast forward or rewind the cassette to go to a particular location, a particular track number or at some location in the song.
- But the CD/DVD players or the songs played in some media player in computers are the example of random access method.
- We can seek to particular song number or at exact location in a song directly without fast forwarding or rewinding it.

- We may need to access some portion of the file to read, write, update or delete data from it.
- To do update a file we have to use following methods.
- `seekg()`
- `seekp()`
- `tellg()`
- `tellp()`

# Command-line argument :

- Command line arguments are the arguments that are passed to the main function.
- These arguments are passed by the command at DOS prompt. So they are known as the command line arguments.
- As normal function we cannot pass arguments to the main function as we don't call the main function explicitly but it is called implicitly when we run the program.
- That's why pass the command line arguments, we have to run the program by the command and the arguments to the main function are passed as the supplementary commands.

**Example :**

```

#include<iostream.h>
#include<conio.h>
int main(int argc, char *argv[])
{
    clrscr();
    int i;
    cout<<endl<<"Total arguments="<<argc;
    cout<<endl<<"Program name is="<<argv[0];
    cout<<endl<<"Other Arguments are\n\n";
    for(i=1;i<argc;i++)
    {
        cout<<endl<<argv[i];
    }
    cout<<endl<<"Total Number of Argument are : "<<argc;
    getch();
    return(0);
}

```

run the above program from **dos shell** and enter following arguments:

C:\TC\BIN\SOURCE> prog\_name.exe hello

# Exception Handling :

- exceptions are runtime anomalies or abnormal conditions that a program encounters during its execution.
- The process of handling these exceptions is called exception handling.
- Using the exception handling mechanism, the control from one part of the program where the exception occurred can be transferred to another part of the code.
- So basically using exception handling in C++, we can handle the exceptions so that our program keeps running.
- **What is a C++ Exception?**
- An exception is an unexpected problem that arises during the execution of a program our program terminates suddenly with some errors/issues. Exception occurs during the running of the program (runtime).

- Generally exceptions are generated at runtime during program execution. So at the time of compilation or say before running the program, the exception cannot be tracked like other type of errors such as syntax errors or logical errors.
- To handle the exceptions in our program we have to monitor the code that can cause an exception such as,
  - Divide-by ZERO exception
  - Array index out of range or any other exception.
- In such block of code we can apply the exception handling mechanism that identifies the exception and informs about the exception.
- **Overview of Exception Handling :**
- The exception handling mechanism includes following simple steps:
  1. Identify the code that can cause exception.  
→ (Use **try** block for this.)
  2. Report the exception.  
→ (Use **throw** keyword)
  3. Get the information about the exception and handle it.  
→ (Use **catch** block)



# Need for Exception Handling :

- To understand need of Exception Handling we have to see an example of 0 divided by some value.
- Example :

```
void main()
{
    int a=10,b=0,c;
    c=a/b;
    cout<<"Value of C ="<<c;
}
```

## Output:

It will throw runtime error →



Unhandled exception at 0x002E5DA0 in exception\_handling.exe:  
0xC0000094: Integer division by zero.

So as we seen in above example.

When we got any error like this we have to use Exception Handling.

- **Why Exception Handling?**
- Exception handling in C++ separates error handling code from normal code to ensure that the normal execution of the program is not interrupted by errors or exceptions.
- Unlike using multiple if-else conditions to handle errors, exception handling **improves code readability and maintainability** by allowing developers to easily manage exceptional situations.
- In C++, exception handling enables the creation of a hierarchy of exception objects, grouping exceptions in namespaces or classes, and categorizing exceptions according to their types.
- Using exception handling in C++, we can throw any number of exceptions from a function but we can choose to handle some of the thrown exceptions.

- A program can throw both pre-defined as well as a custom exception(s) as per the need of the program. We will be taking the example of both the pre-defined as well as custom exceptions later in this article.
- Errors and Exceptions can be of two types in C++.
- **Compile Time Errors** - those errors that are caught during compilation time.
- **Run Time Errors** - those errors that cannot be caught during compile time. As we cannot check these errors during compile time, we name them Exceptions.
- **Note:** Exception handling in C++ can throw both the basic data type as well as user-defined objects as an exception. For throwing an exception in C++, we use the throw keyword.

# Various Components of Exception Handling :

- The exception handling mechanism has 3 components as shown below.

**1. Try    2. Catch    3. Throw**

## **1. Try :**

- The try is a block in which the statements are placed which may generate an exception.

So the statements to monitor for exception are inserted in the try block.

## **2. Catch :**

- There must be a matching catch block after a try block which receives the exception information and handles the exception.

## **3. Throw :**

- The throw keyword is used to report an exception when it is generated. **Note:** Multiple catch statements can be used to catch different type of exceptions thrown by try block.

**Syntax :**

```
try {
    // Block of code to try
    throw exception;
}
catch () {
    // Block of code to handle errors
}
```

**Example :**

```
#include <iostream>
#include <conio.h>
using namespace std;
```

```
int main()
{
    int x = 50;
    int y = 0;
    int answer = 0;
    try
    {
        if (y == 0)
```

```
{
    throw "Division by zero!";
}
answer = x / y;
cout << " Output: " << answer <<
    endl;
}
catch (const char *errorMessage)
{
    cout << errorMessage << endl;
}
getch();
return 0;
}
```

**Output :**

If value of y is 0 then → Division by zero!

If value of y is 2 then → Output: 25

**Example :**

```
#include <iostream>
#include<conio.h>
using namespace std;
```

```
int main( )
{
    int age = 115;
    try{
        if (age >= 18) {
            cout << "Access granted - you are old enough.";
        }
        else {
            throw (age);
        }
    }
    catch (int myNum) {
        cout << "Access denied - You must be at least 18 years old.\n";
        cout << "Age is: " << myNum;
    }
    getch();
    return 0;
}
```

```
Access denied - You must be at least 18 years old.
Age is: 15
```

```
#include <iostream>
#include<conio.h>
using namespace std;
int main()
{
    int amt ,bal;
    try{
        cout << "enter balance";
        cin >> bal;
        if (bal <= 0)    {
            throw invalid_argument
            ("Enter proper balance");
        }
        cout << "Enter Amount";
        cin >> amt;
        if (amt <= 0) {
            throw invalid_argument
            ("Please enter Proper amount");
        }
        if (amt > bal)
```

```
{
    throw runtime_error("amout
is > than available balance");
}
    bal = bal - amt;
    cout <<"Avail balance is :" <<bal ;
}
catch (exception &e) {
    cout << e.what() ;
}
    getch();
    return 0;
}
```

## Output :

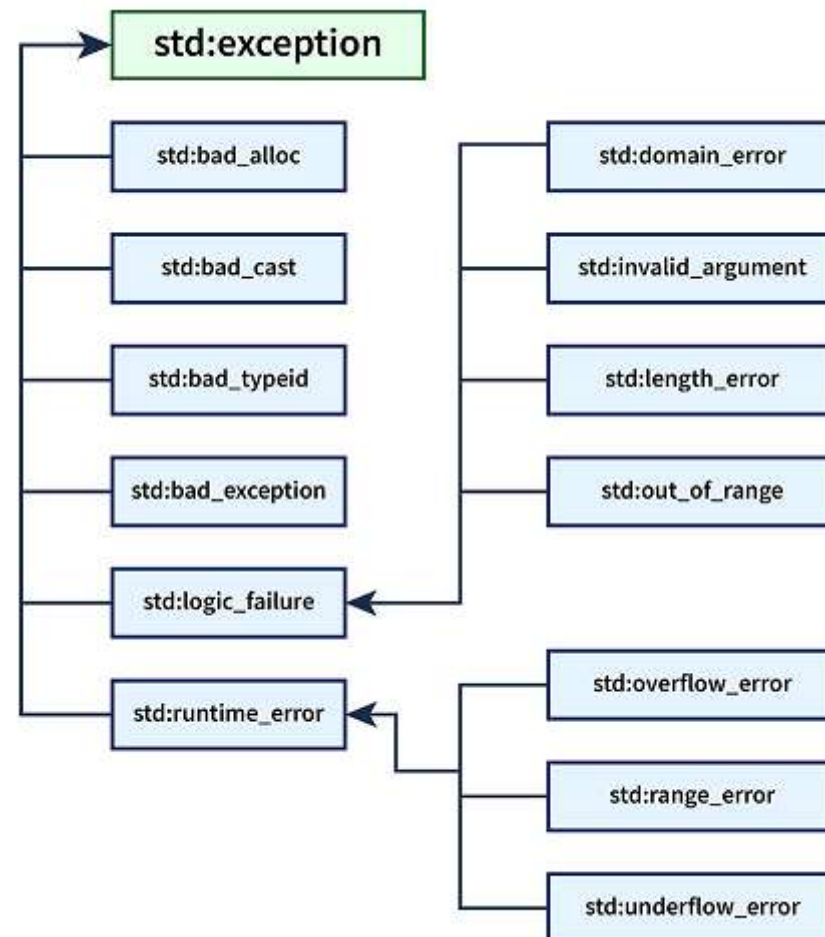
1. enter balance0  
Enter proper balance

2. enter balance1000  
Enter Amount0  
Please enter Proper amount

3. enter balance1000  
Enter Amount1200  
amout is > than available balance

4. enter balance1000  
Enter Amount400  
Avail balance is :600

- The C++ library has some built-in standard exceptions under the `<exception>` header which can be used in our program. The standard exceptions are arranged in a parent-child class hierarchy. Refer to the image shown below for better visualization.





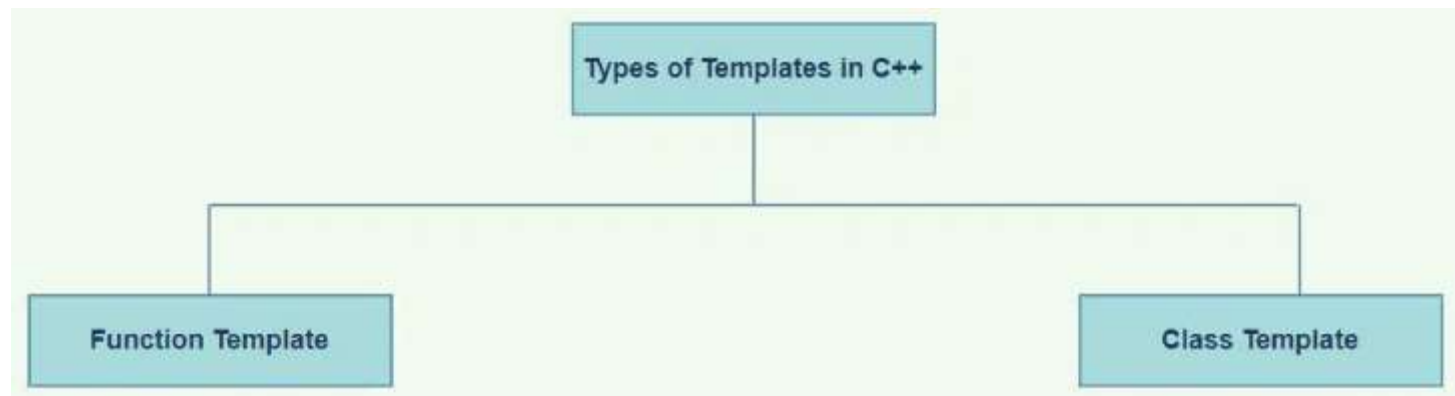
# Introduction to template :

- Templates in c++ is defined as a blueprint or formula for creating a generic class or a function.
- Generic Programming is an approach to programming where generic types are used as parameters in algorithms to work for a variety of data types.
- In C++, a template is a straightforward yet effective tool.
- To **avoid** having to write the **same code** for many data types, the simple concept is to pass the data type as a parameter.
- Special functions that can interact with generic types are known as function templates.
- As a result, we can develop a function template whose functionality can be applied to multiple types or classes without having to duplicate the full code for each kind.

# What is Template :

- Templates in C++ is an interesting feature that is used for generic programming and templates in c++ is defined as a blueprint or formula for creating a generic class or a function.
- Simply put, you can create a single function or single class to work with different data types using templates.
- C++ template is also known as generic functions or classes which is a very powerful feature in C++.
- Template is independent of data type
- A keyword “template” in c++ is used for the template’s syntax and angled bracket in a parameter (t), which defines the data type variable.

- **How do templates work in C++?**
- Templates in c++ works in such a way that it gets expanded at compiler time, just like macros and allows a function or class to work on different data types without being rewritten.
- **Types of Templates in C++**



# What is the function template in C++?

- A function template in c++ is a single function template that works with multiple data types simultaneously, but a standard function works only with one set of data types.
- Examples of function templates are sort(), max(), min(), printArray().

- **Syntax :**

```
template<class type>
```

```
return_type func_name(parameter list)
```

```
{  
    //body of the function  
}
```

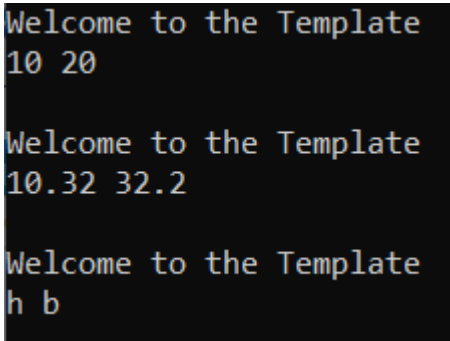
- Here, type is a placeholder name for a data type used by the function. It is used within the function definition.
- The class keyword is used to specify a generic type in a template declaration.

```
#include<iostream>
#include<conio.h>
using namespace std;
template<class T>
void fun(T a,T b)
```

```
{
    cout << "Welcome to the Template"<<endl;
    cout << a << " " << b <<endl <<endl;
}
```

```
void main()
{
    fun(10,20);
    fun(10.32, 32.20);
    fun('h', 'b');
    getch();
}
```

← **Example of function template**



```
Welcome to the Template
10 20

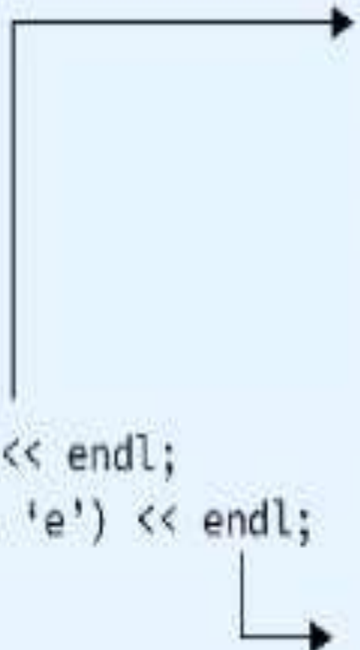
Welcome to the Template
10.32 32.2

Welcome to the Template
h b
```

```
template <typename T>
T myMax (T x, T y)
{
    return (x > y) ? x: y;
}
```

```
int main ()
{
    cout << myMax<int> (3, 7) << endl;
    cout << myMax<char> ('g' , 'e') << endl;
    return 0;
}
```

*Compiler internally generates  
and adds below code*



```
int myMax (int x, int y)
{
    return (x > y) ? x: y;
}
```

*Compiler internally generates  
and adds below code*

```
char myMax (char x , char y)
{
    return (x > y) ? x: y;
}
```

# Multiple Template Parameters :

- we can use multiple template parameters and even use default arguments for those parameters.

- **Example :**

```
#include<iostream>
```

```
#include<conio.h>
```

```
using namespace std;
```

```
template<class T1,class T2,class T3=int>
```

```
void fun(T1 a, T2 b,T3 c)
```

```
{
```

```
    cout << "Welcome to the Template" << endl;
```

```
    cout << a << " " << b << " "c << endl << endl;
```

```
}
```

```
void main()
```

```
{
```

```
    fun(10, 20.56,false);
```

```
    getch();
```

```
}
```

```
Welcome to the Template
10 20.56 0
```

- **Overloading of Template Function in C++**
- As the normal functions, a template function can also be overloaded.
- You can have same name of your function for template function and an UDF function. This will result in function overloading.
- When a template function is overloaded call to the function with same name will be the template function is called.
- If no function definition match with function called the compiler will generate an error.
- So, by overloading template functions in C++, we can define function templates in C++ having the same name but called with different arguments.



## ← Example of overloading of function template

```
#include <iostream>
#include<conio.h>
using namespace std;
template <class T>
void display(T t1)
{
    cout << "Inside the display template function: " << t1 << endl;
}
void display(int t1)
{
    cout << "Inside the overloaded integer-display template function: " << t1 << endl;
}
int main()
{
    display(20);
    //it will call a display function with int parameter
    //it will not call template function with type of T parameter
    display(20.55);
    display('G');
    getch();
    return 0;
}
```

```
Inside the overloaded integer-display template function: 20
Inside the display template function: 20.55
Inside the display template function: G
```

# Class Templates :

- Class templates like function templates, class templates are useful when a class defines something that is independent of the data type.
- Can be useful for classes / Example like LinkedList, BinaryTree, Stack, Queue, Array, etc.

- **Syntax :**

```
template<class Ttype>
class class_name
{
    . . . .
}
class_name<type> obj_name;
```

Here,

**Ttype** is a placeholder name which will be determined when the class is instantiated. We can define more than one generic data type using a comma-separated list. The Ttype can be used inside the class body.

**For an object : class\_name:** It is the name of the class.

**type:** It is the type of the data that the class is operating on.

**obj:** It is the name of the object.

```
#include <iostream>
#include<conio.h>
using namespace std;
template<class T>
class A
{
public:
    T num1 = 5;
    T num2 = 6;
    void add()
    {
        cout << "Addition of num1 and num2 : " << num1 + num2 << endl;
    }

};

int main()
{
    A<int> d;
    d.add();
    getch();
    return 0;
}
```

# Member function template(outside) :

- If we want to use a function template as a member function of any class.
- Let's see how to create template member function outside of the class.

## Syntax :

```
template <class T>
class ClassName {
    ... ..
    // Function prototype
    returnType functionName();
};
// Function definition
```

```
template <class T>
returnType
ClassName<T>::function_Name()
{
    .....
}
```

## How to create class object :

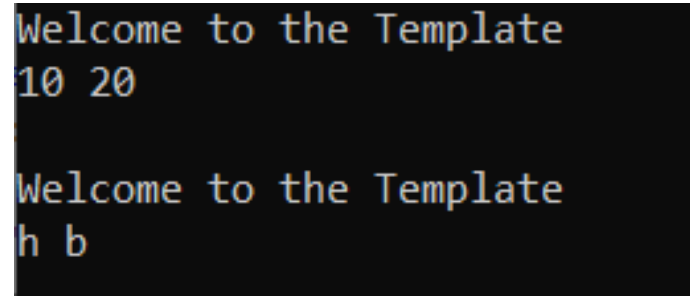
```
Class_name <Data
    type>object_nm;
```

## Example of Member function template :

```
#include<iostream>
#include<conio.h>
using namespace std;
template<class T>

class demo{
public:
    void fun(T,T);
};
template<class T>
void demo<T>::fun(T a, T b)
{
    cout << "Welcome to the Template" << endl;
    cout << a << " " << b << endl << endl;
}

void main()
{
    demo <int> obj1;
    demo <char> obj2;
    obj1.fun(10, 20);
    obj2.fun('h', 'b');
    getch();
}
```



```
Welcome to the Template
10 20

Welcome to the Template
h b
```

## Class template with multiple parameters :

- We can use more than one generic data type in a class template, and each generic data type is separated by the comma.

- **Syntax**

```
template<class T1, class T2, .....>
```

```
class class_name
```

```
{
```

```
    // Body of the class.
```

```
}
```

```
Class_nm <type,type>obj_nm;
```

```
#include <iostream>
#include<conio.h>
using namespace std;
template<class T1, class T2>
class A
{
    T1 a;
    T2 b;
public:
    A(T1 x, T2 y)
    {
        a = x;
        b = y;
    }
    void display()
    {
        std::cout << "Values of a and b are : " << a << ", " << b << std::endl;
    }
};

int main()
{
    A<int, float> d(5, 6.5);
    d.display();
    getch();
    return 0;
}
```

Values of a and b are : 5 ,6.5

# Introduction to STL :

- **Alexander Stepanov** invented the Standard Template Library (STL) in 1994.
- The STL is a set of algorithms and data structures that help with storing and manipulating objects, making programs more reusable and robust.



# STL :

- The C++ **Standard Template Library** (STL) is a set of template classes and functions that provides the implementation of common data structures and algorithms such as lists, stacks, arrays, sorting, searching, etc.
- It also provides the iterators and functions which makes it easier to work with algorithms and containers.
- It is a generalized library so we can use it with almost every data type without repeating the implementation code.

- **Components of STL**
- STL is a set of classes that provide generic classes and functions that can be used to implement's data structures and algorithm.
- The components of STL are the features provided by Standard Template Library (STL) in C++ that can be classified into 4 types:
  1. **Containers**
  2. **Algorithms**
  3. **Iterators**
  4. **Functors**

# Container :

- Containers are the data structures used to store objects and data according to the requirement.
- It is way that stored data is organized in memory.
- Ex: storing array of elements in memory.
- Each container is implemented as a template class that also contains the methods to perform basic operations on it.
- Every STL container is defined inside its own header file.
- STL containers store data and organize them in a specific manner as required.
- For example, **vectors** store data of the same type in a sequential order. Whereas, **maps** store data in key-value pairs.
- Containers can be further classified into 3 types:
  1. **Sequence Containers**
  2. **Associative Containers**
  3. **Unordered Associated Containers**

## 1. Sequence containers:

1. Array
2. Vector
3. Queue
4. Deque
5. Forward\_list
6. List

## 2. Associative containers:

1. Set
2. Multiset
3. Map
4. Multimap

## 3. Unordered associative containers:

1. Unordered\_set
2. Unordered\_multiset
3. Unordered\_map
4. Unordered\_multimap

- vectors are like resizable arrays; they store data of the same type in a sequence and their size can be changed during runtime as needed.
- We need to import the <vector> header file to use a vector.

- **Example of vector :**

```
#include <iostream>
```

```
#include <vector>
```

```
#include <conio.h>
```

```
using namespace std;
```

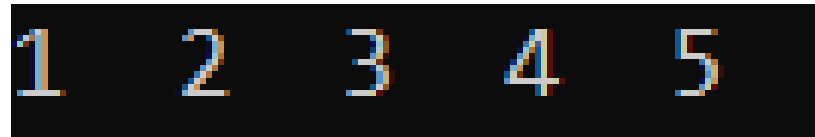
```
void main() {
```

```
// create vector of int type
```

```
vector<int> numbers{ 1, 2, 3, 4, 5 };
```

```
// print vector elements using ranged loop
```

```
    for (int number : numbers) {  
        cout << number << " ";
```



```
    }
```

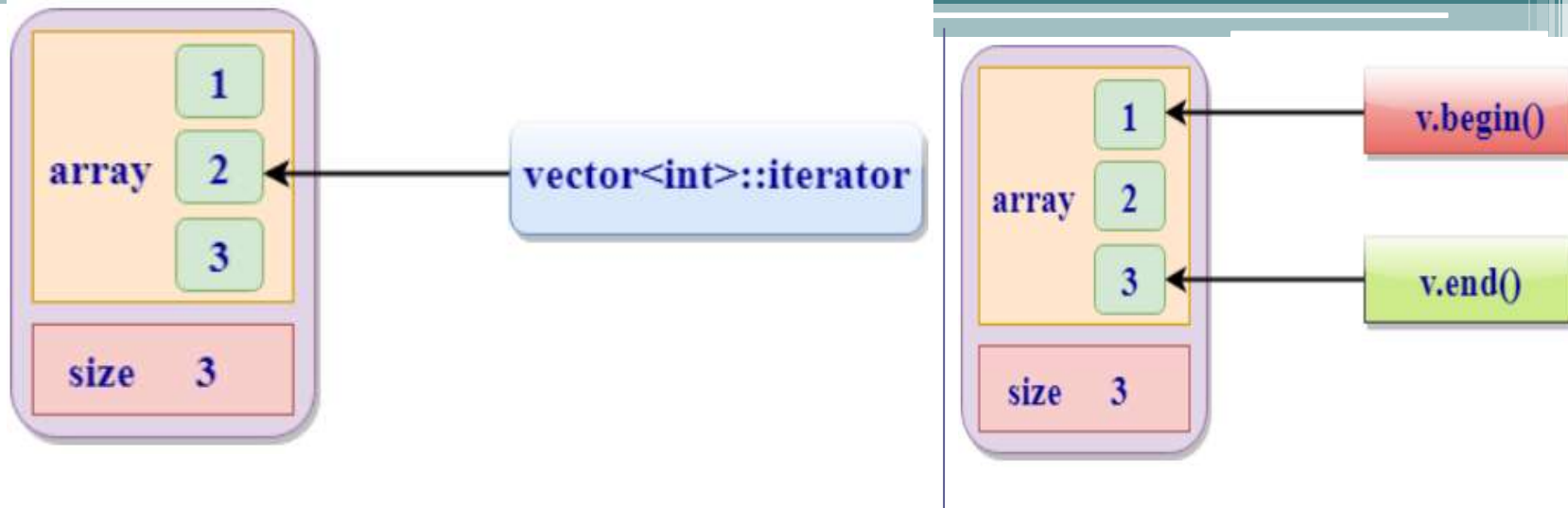
```
    getch();
```

```
}
```

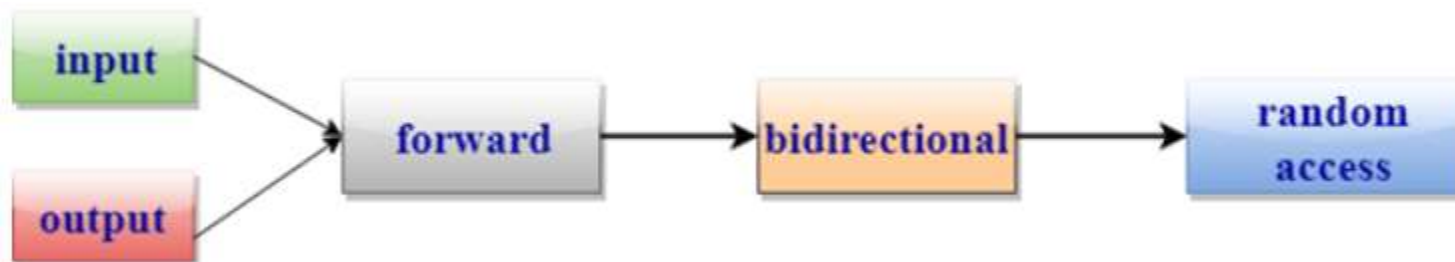
# Iterator :

- Iterators are pointer-like entities used to access the individual elements in a container. Iterators are moved sequentially from one element to another element. This process is known as iterating through a container.
- Iterators are defined inside the **<iterator>** header file.
- Iterators are pointer type of objects, they point to the element of container.
- Ex : it is used to point out element to an array.

Ex :



- Iterator contains mainly two functions:
- **begin()**: The member function `begin()` returns an iterator to the first element of the vector.
- **end()**: The member function `end()` returns an iterator to the past-the-last element of a container.
- Iterators are mainly divided into five categories:



- In C++ STL, iterators are of 5 types:
  1. **Input Iterators** : Input Iterators can be used to read values from a sequence once and only move forward.
  2. **Output Iterators** : Output Iterators can be used to write values into a sequence once and only move forward.
  3. **Forward Iterators** : Forward Iterators combine the features of both input and output iterators.
  4. **Bidirectional Iterators** : Bidirectional Iterators support all operations of forward iterators and additionally can move backward.
  5. **Random Access Iterators** : Random Access Iterators support all operations of bidirectional iterators and additionally provide efficient random access to elements.



- **Example of iterator :**

```
#include <iostream>
#include <vector>
#include <conio.h>
using namespace std;
void main() {
    vector<int> numbers{ 1, 2, 3, 4, 5 };
    // initialize vector iterator to point to the first element
    vector<int>::iterator itr = numbers.begin();
    cout << "First Element: " << *itr << " " << endl;
    // change iterator to point to the last element
    itr = numbers.end() - 1;
    cout << "Last Element: " << *itr;
    getch();
}
```

```
First Element: 1
Last Element: 5
```

# Algorithm :

- STL algorithms offer a wide range of functions to perform common operations on data (mainly containers).
- These functions implement the most efficient version of the algorithm for tasks such as sorting, searching, modifying and manipulating data in containers, etc. All STL algorithms are defined inside the **<algorithm>** and **<numeric>** header file.
- Types of algorithm :
  1. Sorting algorithms
  2. Searching algorithms
  3. Copying algorithms
  4. Counting algorithms

- **Example of sorting data of vector**

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <conio.h>
using namespace std;

int main() {

    // initialize vector of int type
    vector<int> numbers = { 3, 2, 5, 1, 4 };

    // sort vector elements in ascending order
    sort(numbers.begin(), numbers.end());

    // print the sorted vector
    for (int number : numbers) {
        cout << number << " ";
    }
    getch();
    return 0;
}
```



1 2 3 4 5

- **Benefits of C++ Standard Template Library (STL) :**

- The key benefit of the STL is that it provides a way to **write generic, reusable code and tested code** that can be applied to different data types. This means you can write an algorithm once, and then use it with other types of data without having to write separate code for each type.

Other benefits include:

1. STL provides flexibility through customizable templates, functors, and lambdas.
2. Pre-implemented tools let you focus on problem-solving rather than low-level coding.
3. STL handles memory management which reduces common errors like memory leaks.

- **Limitations of C++ Standard Template Library (STL)**
- The major limitation of the C++ Standard Template Library (STL) is **Performance Overheads**. While STL is highly optimized for general use cases, its generic nature can lead to less efficient memory usage and execution time compared to custom and specialized solutions.  
Other limitations can be:
  1. Complexity of debugging.
  2. Lack of control over memory management.
  3. Limited support for concurrent programming.
  4. Limited flexibility with custom data structures integrating.

# More about STL

- <https://www.geeksforgeeks.org/the-c-standard-template-library-stl/>
- <https://www.javatpoint.com/cpp-stl-components>