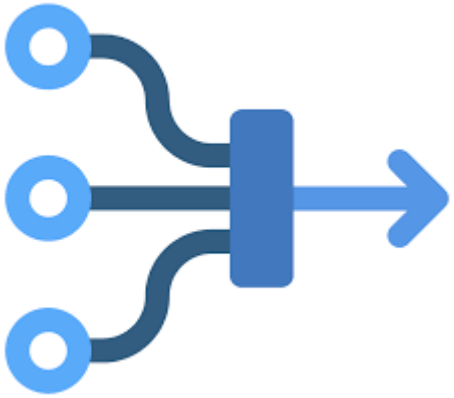
Object Oriented
Programming with C++

Unit-7
I/O and File Management

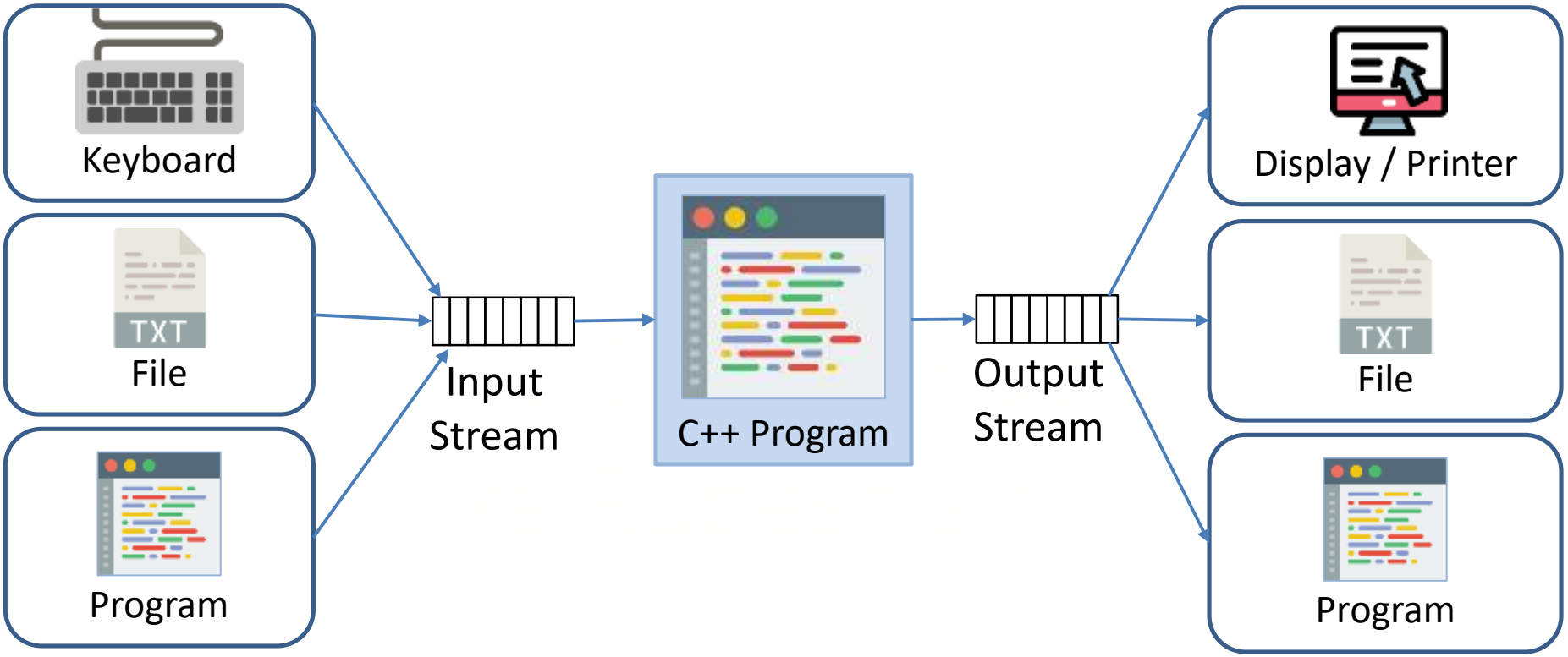
I/O and File Management

- Concept of streams
- cin and cout objects
- C++ stream classes
- Unformatted and formatted I/O
- Manipulators
- File stream
- C++ File stream classes
- File management functions
- File modes
- Binary and random Files

Concepts of Streams



Concept of Streams



Input source to stream

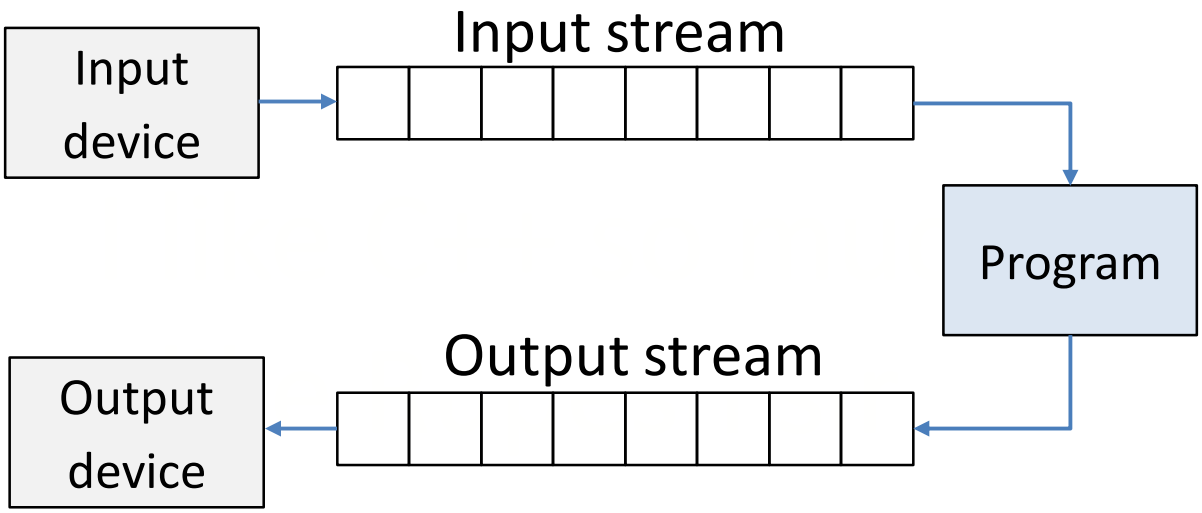
Output target from stream

Concept of streams(Cont…)

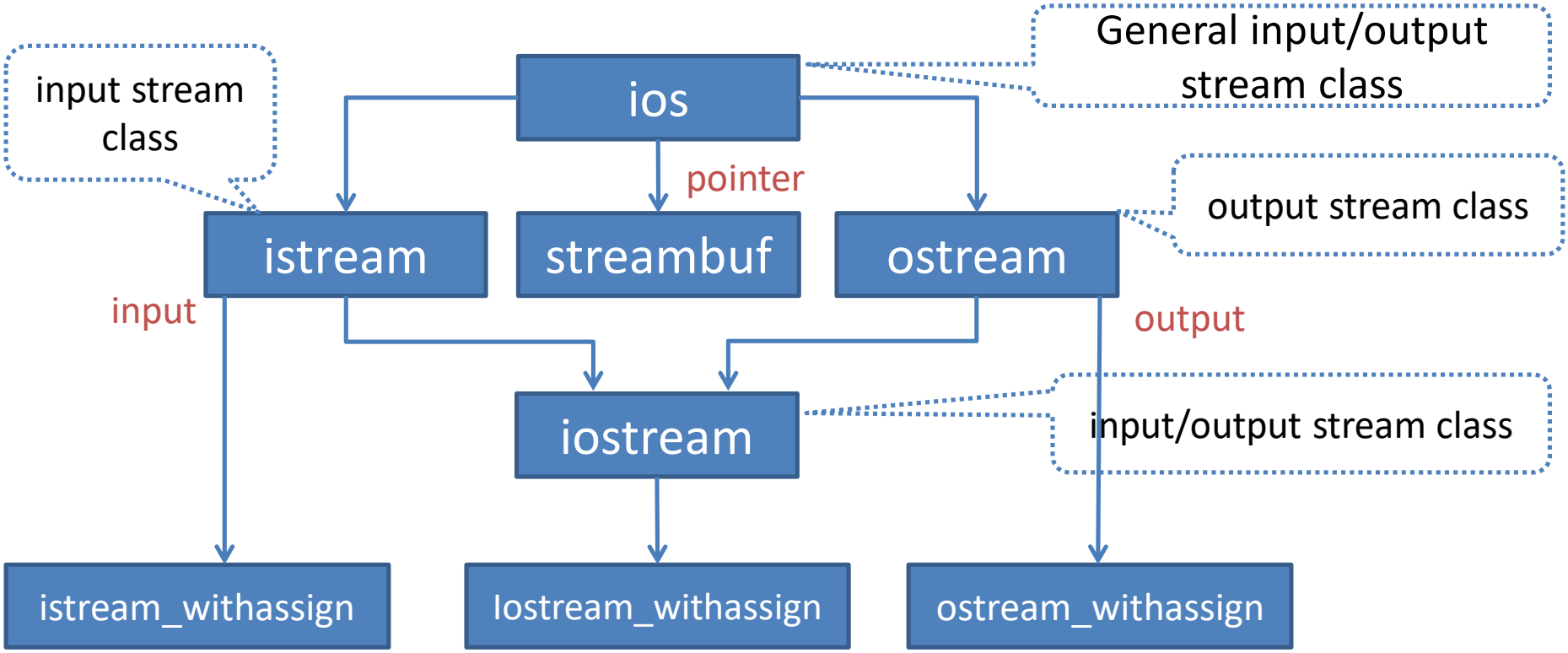
- A **stream** is a general name given to a flow of data.
- A **stream** is a sequence of bytes.
- The source stream that provides data to programs is called **input stream**.
- The destination stream receives output from the program is called **output stream**.

- In header **<iostream>**, a set of class is defined that supports I/O operations.
- The classes used for input/output to the **devices** are declared in the **IOSTREAM** file.
- The classes used for **disk file** are declared in the **FSTREAM** file.

Input/Output streams



Stream class for console I/O operations



- **istream_withassign**, **ostream_withassign** and **iostream_withassign** add assignment operators to its base classes.
- **cout** which is directed to video display, is predefined object of `ostream_withassign`.
- Similarly **cin** is an object of `istream_withassign`.

Unformatted and Formatted I/O

put(), get(), getline(), write() - Unformatted I/O Operations

```
char ch;
cin.get(ch);
ch=cin.get();
cin>>ch;
cout.put(ch);
cout.put('x');

char name[20];
cin.getline(name,10);

cin>>name;

cout.write(name,10);
```

Get a character from keyboard
Similar to cin.get(ch);

The operator >> can also be used to read a character but it will skip the white spaces and newline character.

put() function can be used to display value of variable ch or character.

line size

getline() reads whole line of text that ends with newline character or upto (size-1).

cin can read strings that do not contain white spaces

write() displays string of given size, if the size is greater than the length of line, then it displays the bounds of line.

ios Format Functions

Function	Task
width()	To specify the required field size for displaying an output value
precision()	To specify number of digits to be displayed after the decimal point of a float value.
fill()	To specify a character that is used to fill the unused portion of a field.
setf()	To specify format flags that can control the form of output.
unsetf()	To clear the flags specified

Example:

```
cout.setf(ios::left,ios::adjustfield);
cout.width(6);
cout.fill('#');
cout<<"543";
```

output:

5	4	3	#	#	#

Flags and bit fields

Format required	Flag (arg1)	Bit-field (arg2)
Left justified output	ios::left	ios::adjustfield
Right justified output	ios::right	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

`setf(arg1, arg2)`

arg-1: one of the formatting flags.

arg-2: bit field specifies the group to which the formatting flag belongs.

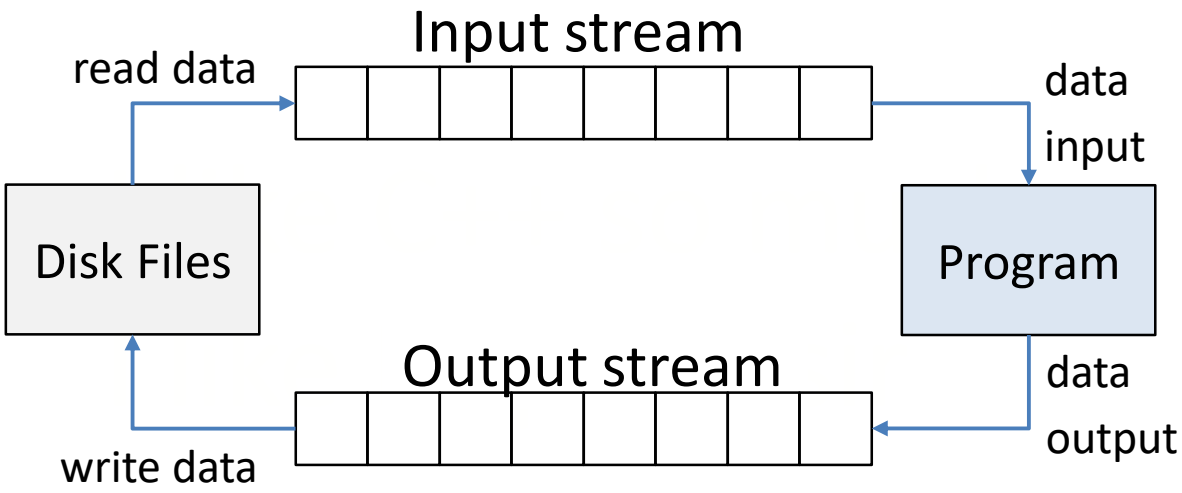
Manipulators for formatted I/O operations

- **Manipulators** are special functions that can be included in the I/O statements to alter the format parameters of a stream.
- To access manipulators, the file `<iomanip>` should be included in the program.

Function	Manipulator	Meaning
width()	**setw()**	Set the field width.
precision()	**setprecision()**	Set the floating point precision.
fill()	**setfill()**	Set the fill character.
setf()	**setiosflags()**	Set the format flag.
unsetf()	**resetiosflags()**	Clear the flag specified.
"\n"	**endl**	Insert a new line and flush stream.

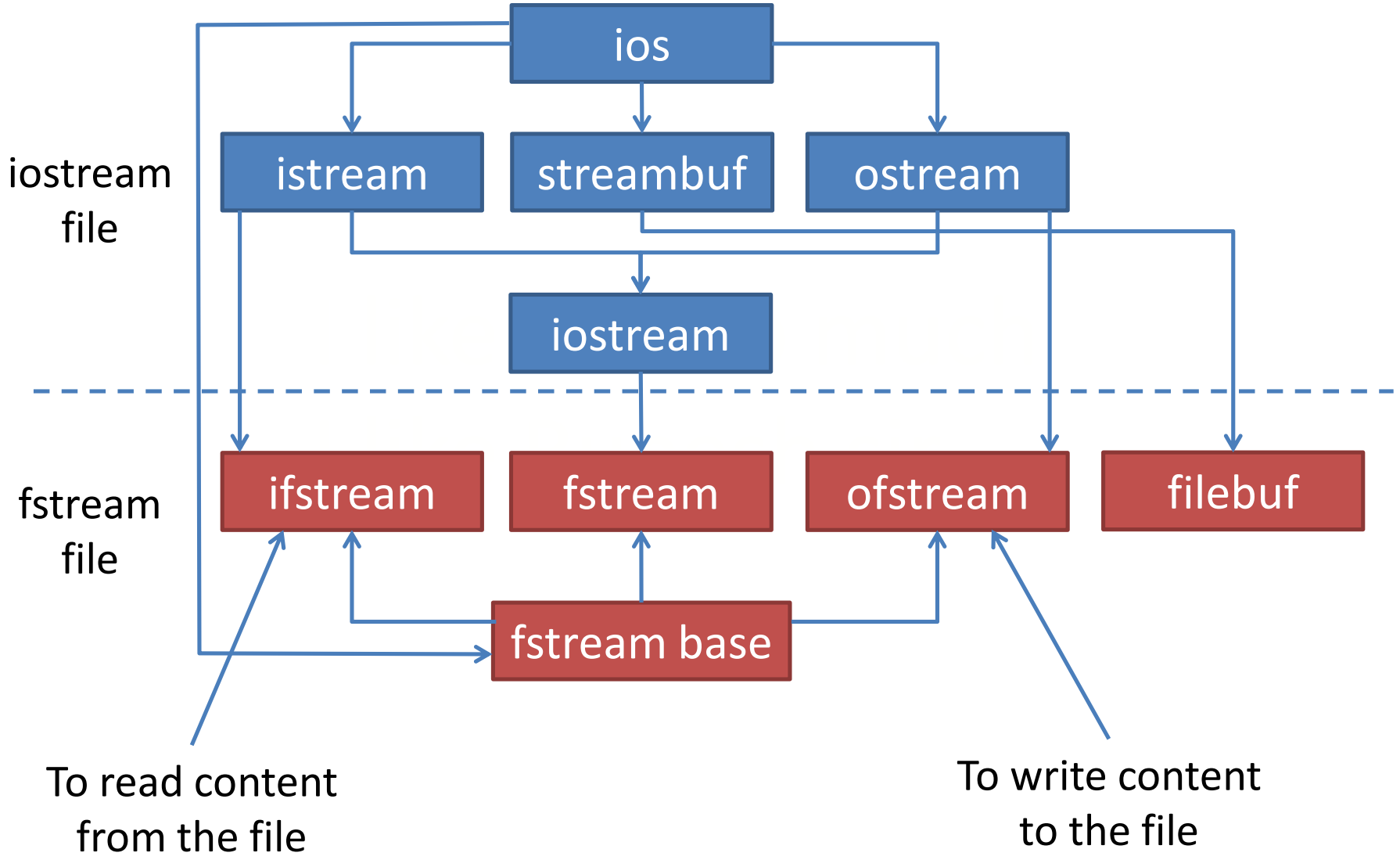
File stream classes

File input output streams



File input output streams

File stream classes for file operations



File stream classes

class	contents
fstreambase	▪ Provides operations common to the file streams. ▪ Contains **open()** and **close()** functions.
ifstream	▪ Provides input operations. ▪ Contains **open()** with default input mode. ▪ Inherits **get(), getline(), read(), seekg()** and **tellg()** functions from istream.
ofstream	▪ Provides output operations. ▪ Contains **open()** with default output mode. ▪ Inherits **put(), seekp(), tellp()** and **write()** functions from **ostream**.
fstream	▪ Provides support for simultaneous input and output operations. ▪ Inherits all the functions from istream and ostream from iostream.
filebuf	▪ Its purpose is to set the file buffers to read and write.

File handling steps

1. Open / Create a file
2. Read / Write a file
3. Close file

Create and Write File (Output)

Create object of **ofstream** class

```
ofstream send;
```

Call **open()** function using **ofstream** object to open a file

```
send.open("abc.txt");
```

This will open existing file, if not exist then it will create file.

Write content in file using **ofstream** object

```
send<<"Hello, this is India";
```

Call **close()** function using **ofstream** object to close file

```
send.close();
```

Open and Read File (Input)

Create object of **ifstream** class

```
ifstream rcv;
```

Call **open()** function using **ifstream** object to open a file

```
rcv.open("abc.txt");
```

Read content of file using **ifstream** object

```
rcv>>name;
```

```
rcv.getline(name);
```

Call **close()** function using **ifstream** object to close file

```
rcv.close();
```

Opening a file

```
ofstream outFile("sample.txt");  //output only
ifstream inFile("sample.txt");  //input only

ofstream outFile;
outFile.open("sample.txt");
ifstream inFile;
inFile.open("sample.txt");
```

This creates **outFile** as an **ofstream** object that manages the output stream.

This object can be any valid C++ name such as myfile, o_file .

- Syntax file **open()** function:

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

- By default **ofstream** opens file for writing only and **ifstream** opens file for reading only.

File open() function

File open() function

Syntax:

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

- By default **ofstream** opens file for writing only
- By default **ifstream** opens file for reading only.

Three ways to create a file

1

```
ofstream send("abc.txt");  //constructor
```

2

```
ofstream send;
send.open("abc.txt");  //open() function
```

3

```
ofstream send;
send.open("abc.txt",ios::out); //open()
function with mode
```

File opening modes

Parameter	Meaning
ios :: in	Open file for reading only
ios :: out	Open file for writing only
ios :: app	Append to end-of-file
ios :: ate	Go to end-of-file on opening
ios :: binary	Binary file
ios :: trunc	Delete content of file if exists
ios :: nocreate	Open fails if the file does not exists
ios :: noreplace	Open fails if the file already exists

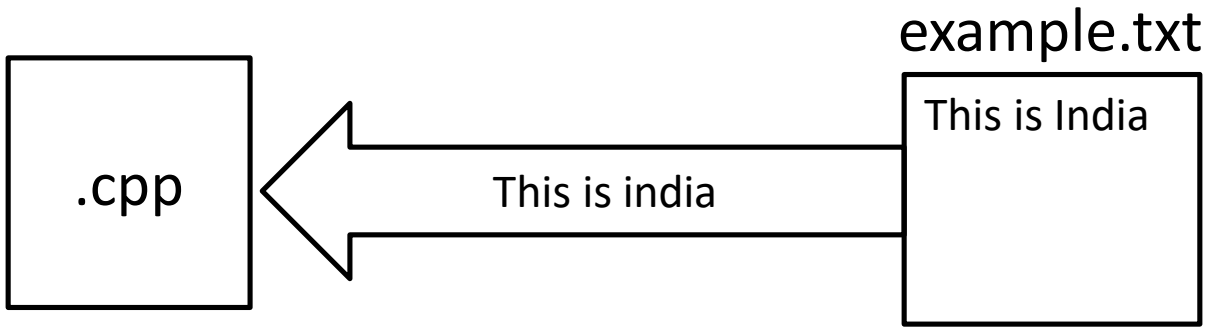
File operations

```
#include <iostream>
#include <fstream>
using namespace std;
int main ()
{
  ofstream myfile;
  myfile.open("example.txt",ios::out);
  myfile << "This is India.\n";
  myfile.close();
}
```

File operations (Cont..)

```
int main ()
{
   char line[50];
   ifstream rfile;
   rfile.open("example.txt",ios::in)
   rfile.getline(line,50);
   // rfile>>line is also valid;

   cout<<line;
   rfile.close();
}
```



File operations program

```
int main()
{
    char product[20];
    int price;
    cout<<"Enter product name=";
    cin>>product;
    cout<<"Enter price=";
    cin>>price;

    ofstream outfile("stock.txt");
    outfile<<product<<endl;
    outfile<<price;

    ifstream infile("stock.txt");
    infile>>product;
    infile>>price;

    cout<<product<<endl;
    cout<<price;
}
```

Opening a file to write data into file

Opening a file to read data from file

File handling Program

- Write a program that opens **two text files** for reading data.
- It creates a **third file** that contains the text of first file and then that of second file

  (text of second file to be appended after text of the first file, to produce the third file).

```
int main() {
   fstream file1,file2,file3;
   file1.open("one.txt",ios::in);
   file2.open("two.txt",ios::in);
   file3.open("three.txt",ios::app);
   char ch1,ch2;
   while(!file1.eof())
   {
      file1.get(ch1); cout<<ch1<<endl;
      file3.put(ch1);
   }
   file1.close();
   while(!file2.eof())
   {
      file2.get(ch2); cout<<ch2<<endl;
      file3.put(ch2);
   }
   file2.close(); file3.close();
}
```

File pointers

- Each file has two associated pointers known as the **file pointers**.
- One of them is called **input pointer (or get pointer)** and the other is called **output pointer (or put pointer)**.
- **Input pointer** is used for reading the content of a given file location.
- **Output pointer** is used for writing to a given file location.

Functions for manipulation of file pointers

Function	Meaning
seekg()	Moves get pointer (input) to specified location
seekp()	Moves put pointer (output) to specified location
tellg()	Gives current position of the get pointer
tellp()	Gives current position of the put pointer

```
ifstream rcv;
ofstream send;

rcv.seekg(30); //move the get pointer to byte number 30 in the file
send.seekp(30);//move the put pointer to byte number 30 in the file
int posn = rcv.tellg();
int posn = send.tellp();
```

Functions for manipulation of file pointers

```
Another prototype
   seekg ( offset, direction );
   seekp ( offset, direction );
```

Function	Meaning
ios::beg	offset counted from the beginning of the stream
ios::cur	offset counted from the current position of the stream pointer
ios::end	offset counted from the end of the stream

write() and read() functions

- The functions **write()** and **read(),** different from the functions **put()** and **get(),** handle the data in binary form.

```
infile.read ((char * ) &V,sizeof(V));
outfile.write ((char *) &V ,sizeof(V));
```

- These functions take two arguments. The first is the address of the variable V, and the second is the length of that variable in bytes.
- The address of the variable must be cast to type **char***(i.e pointer to character type).

Reading & Writing class objects

```
class inventory
{
    char name[10];
    float cost;
    public:
    void readdata()
    {
        cout<<"Enter Name=";
        cin>>name;
        cout<<"Enter cost=";
        cin>>cost;
    }
    void displaydata()
    {
        cout<<"Name="<<name<<endl;
        cout<<"Cost="<<cost;
    }
};
```

Reading & Writing class objects

```
int main()
{
    inventory ob1;
    cout<<"Enter details of product\n";

    fstream file;
    file.open("stock.txt",ios::in | ios::app);

    ob1.readdata();
    file.write((char *)&ob1,sizeof(ob1));

    file.read((char *)&ob1,sizeof(ob1));

    ob1.displaydata();
    file.close();
}
```

Templates, Exceptions and STL

- What is template?
- Function templates and class templates
- Introduction to exception
- Try-catch, throw
- Multiple catch
- Catch all
- Rethrowing exception
- Implementing user defined exceptions
- Overview and use of Standard Template Library(STL)

Exception

1/0

Introduction to Exception

```
int main()
{
   int a,b,c;
   cout<<"Enter value a=";
   cin>>a;
   cout<<"Enter value b=";
   cin>>b;
   c=a/b;
   cout<<"answer="<<c;
}
```

Output:
Enter value a=5
Enter value b=2
answer=2

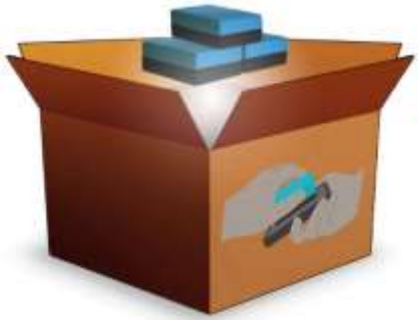
Output:
Enter value a=5
Enter value b=0
Abnormal Termination occur

Introduction to Exception(Cont…)

- Runtime errors are termed as **exception**.
- **Exception handling** is the process to manage the runtime errors by converting the abnormal termination of a program to normal termination of a program.

try, throw and catch

try

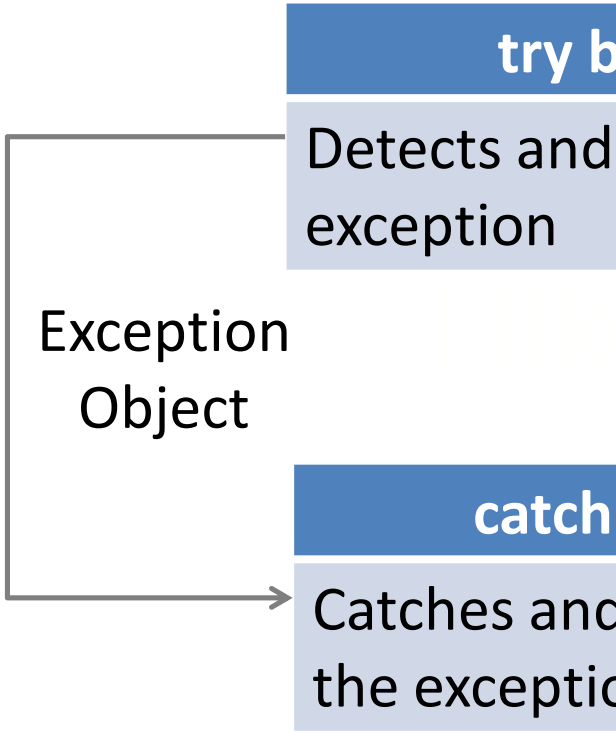


throw

catch

try, throw and catch

- C++ exception handling mechanism is built upon three keywords **try**, **throw** and **catch**.




```
try
{
  .....
  throw exception; //this block
  detects and throws an exception
}
```

```
catch(type arg)
{
  .....
  ... //exception handling block
}
.....
```

try, throw and catch example

```
int main()
{
   int a,b,c;
   cout<<"Enter two values=";
   cin>>a>>b;
   try
   {
      if(b!=0)
            c=a/b;
            cout<<"answer="<<c;
      else
            throw(b);
   }
   catch(int x)
   {
      cout<<"Exception caught: Divide by zero\n";
   }
}
```

Output:

Enter value a=5

Enter value b=0

Exception caught: Divide by zero

Multiple catch example

```
void test(int x){
   try
   {
      if(x==1)
        throw x;
      else if(x==0)
        throw 'x';
      else if(x==-1)
        throw 5.14;
   }
   catch(int i){
     cout<<"\nCaught an integer";
   }
   catch(char ch){
     cout<<"\nCaught a character";
   }
   catch(double i){
     cout<<"\nCaught a double";
   }
}
```

```
int main()
{
   test(1);
   test(0);
   test(-1);
}
```

Output:

Caught an integer

Caught a character

Caught a double

Catch all Exception

Catch all exception

- In some situations, we may not predict all possible types of exceptions and therefore may not be able to design independent catch handlers to catch them.

```
Syntax:
catch(...)
{
      //statements for processing all exceptions
}
```

Catch all exception example

```
#include<iostream>
using namespace std;
void test(int x)
{
   try
   {
      if(x==0) throw x;
      if(x==-1) throw 'a';
      if(x==1) throw 5.15;
   }
   catch(...)
   {
   cout<<"Caught an exception\n";
   }
}
```

```
int main()
{
   test(-1);
   test(0);
   test(1);
}
```

Output:

Caught an exception

Caught an exception

Caught an exception

Re-Throwing exception

- An exception is thrown from the catch block is known as the **re-throwing** exception.
- It can be simply invoked by **throw** without arguments.
- Rethrown exception will be caught by **newly defined catch statement**.

```
void divide(double x, double y){
  try
  {
   if(y==0)
     throw y;
   else
     cout<<"Division="<<x/y;
  }
  catch(double)
  {
   cout<<"Exception inside
function\n";
   throw;
  }
}
```

```
int main()
{
  try
  {
   divide(10.5,2.0);
   divide(20.0,0.0);
  }
  catch(double)
  {
   cout<<"Exception inside
main function";
  }
}
```

Output:

Division=5.25

Exception inside function

Exception inside main function

Exceptions thrown from functions

```
#include <iostream>
using namespace std;
void test(int x)
{
cout<<"Inside function:"<<x<<endl;
if(x) throw x;
}
int main()
{
cout<<"Start"<<endl;
try
{
test(0);
test(1);
test(2);
}
catch(int x)
{
cout<<"Caught an int exception:"<< x<<endl;
}

}
```

User defined Exception

- There maybe situations where you want to generate some **user specific exceptions** which are not pre-defined in C++.
- In such cases C++ provided the mechanism to create our own exceptions by inheriting the exception class in C++.

User defined Exception

```
#include <iostream>
#include <exception>
class myexception: public exception
{
  virtual const char* what() const throw()
  {
    return "My exception happened";
  }
} myex;
int main (){
  try
  {
    throw myex;
  }
  catch (exception& e)
  {
    cout << e.what() << '\n';
  }
}
```

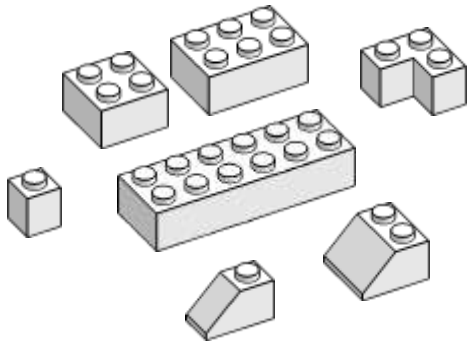
User defined Exception(Cont…)

```
double myfunction (char arg) throw (int);
```

- This declares a function called **myfunction**, which takes one argument of type **char** and returns a value of type **double**.
- If this function throws an exception of some type other than int, the function calls **std::unexpected** instead of looking for a handler or calling **std::terminate.**
- If this throw specifier is left empty with no type, this means that **std::unexpected** is called for any exception.
- Functions with no throw specifier (regular functions) never call **std::unexpected**, but follow the normal path of looking for their exception handler.

```
int myfunction (int param) throw(); //all exceptions call unexpected
int myfunction (int param);         //normal exception handling
```

Template



Need of Templates

```
int add(int x, int y)
{
  return x+y;
}
```

```
float add(float x, float y)
{
   return x+y;
}
```

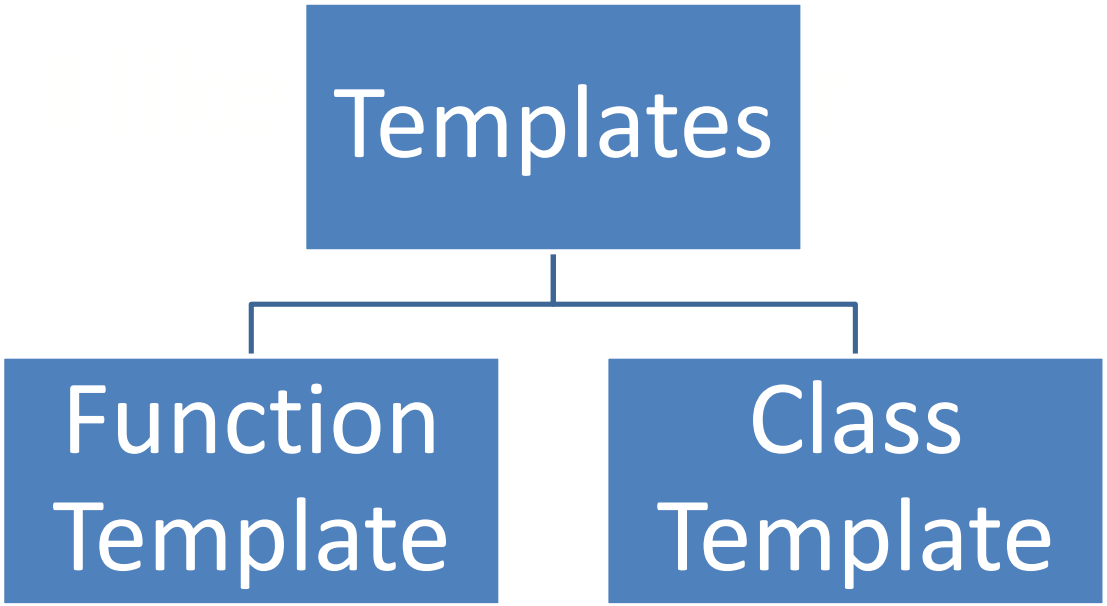
```
char add(char x,char y)
{
      return x+y;
}
```

```
double add(double x, double y)
{
  return x+y;
}
```

We need a single function that will work for int, float, double etc…

Templates

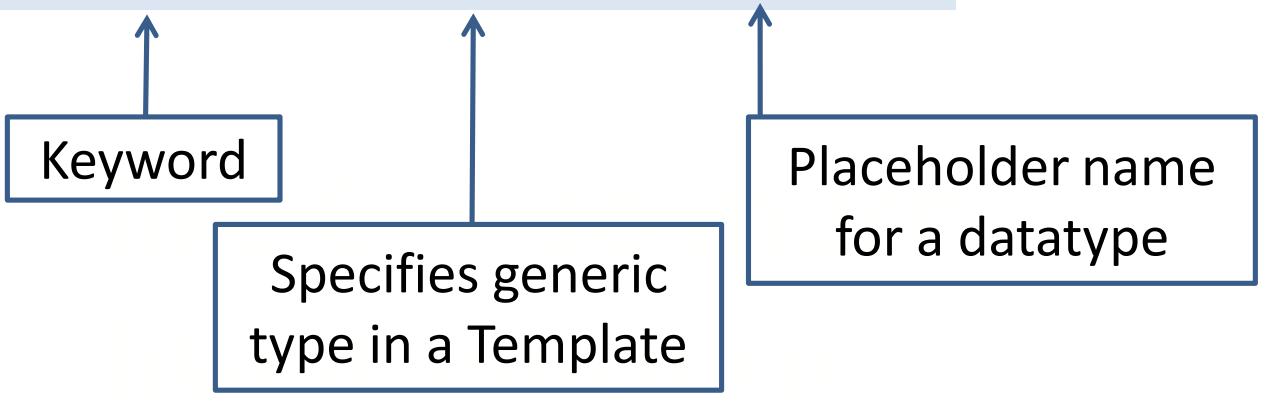
- Templates concept enables us to define **generic classes** and **functions**.
- This allows a function or class to work on many different data types without being rewritten for each one.



Function Template

Syntax:

```
template<class Ttype>
```




```
template<typename Ttype>
```

Templates

- C++ **templates** are a powerful mechanism for code reuse, as they enable the programmer to write code that behaves the same for any data type.
- By **template** we can define generic classes and functions.
- In simple terms, you can create a single function or a class to work with different data types using **templates**.
- It can be 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.

Function Template

- Suppose you write a function printData:

```
void printData(int value){
      cout<<"The value is "<<value;
}
```

- Now if you want to print double values or string values, then you have to overload the function:

```
void printData(float value){
      cout<<"The value is "<<value;
}
void printData(char *value) {
      cout<<"The value is "<<*value;
}
```

- To perform same operation with different data type, we have to write same code multiple time.

Function Template (Cont…)

- C++ provides templates to reduce this type of duplication of code.

```
template<typename T>
void printData(T value){
      cout<<"The value is "<<value;
}
```

- We can now use `printData` for any data type. Here **T** is a template parameter that identifies a type.
- Then, anywhere in the function where **T** appears, it is replaced with whatever type the function is instantiated.

```
int i=3;
float d=4.75;
char *s="hello";
printData(i); // T is int
printData(d); // T is float
printData(s); // T is string
```

```
#include <iostream>
using namespace std;
template <typename T>
T Large(T n1, T n2)
{
 return (n1 > n2) ? n1 : n2;
}
int main(){
   int i1, i2; float f1, f2; char c1, c2;
   cout << "Enter two integers:\n";
   cin >> i1 >> i2;
   cout << Large(i1, i2) <<" is larger." << endl;
   cout << "\nEnter two floating-point numbers:\n";
   cin >> f1 >> f2;
   cout << Large(f1, f2) <<" is larger." << endl;
   cout << "\nEnter two characters:\n";
   cin >> c1 >> c2;
   cout << Large(c1, c2) << " has larger ASCII value.";
}
```

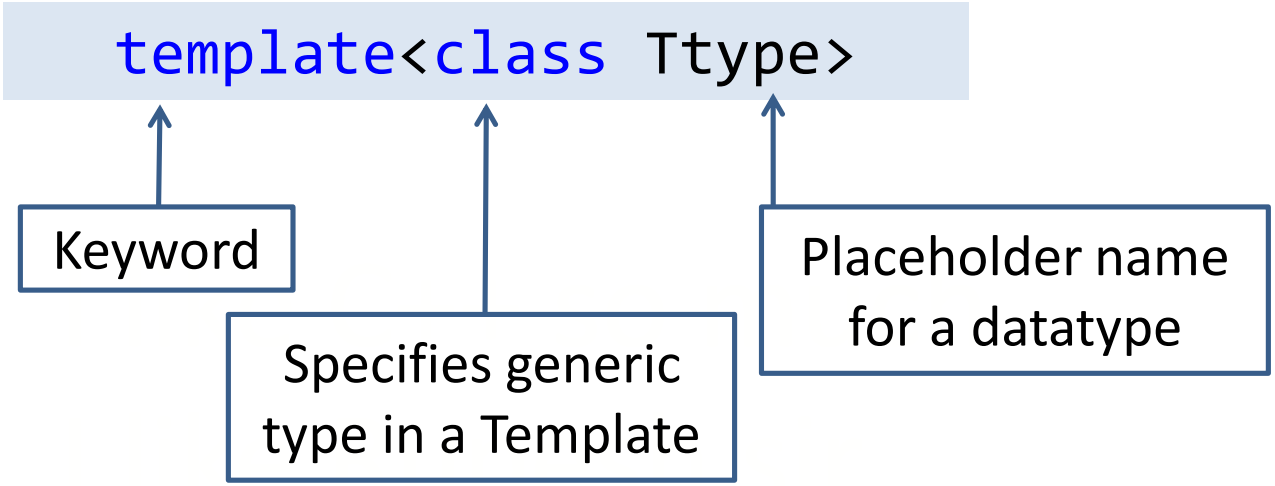
- T is a **template** argument that accepts different data types
- **typename** is a keyword
- You can also use keyword **class** instead of **typename**

Class Template

- Sometimes, you need a class implementation that is same for all classes, only the data types used are different.
- Normally, you would need to create a different class for each data type OR create different member variables and functions within a single class.

Class Template

Syntax:



Object of template class

The object of template class are created as follows

```
class name <data type> object name;
```

```
template<class Ttype>
class sample
{
   Ttype a,b;
   public:
      void getdata()
      {
        cin>>a>>b;
      }
      void sum();
};
```

```
int main()
{
  sample <int>s1;
  sample <float>s2;
  s1.getdata();
  s1.sum();
  s2.getdata();
  s2.sum();
}
```

Class Template Example

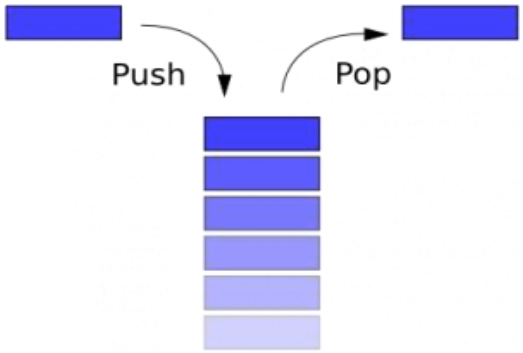
```
template<class T1, class T2>
class Sample
{
   T1 a; T2 b;
   public:
   Sample(T1 x,T2 y){
      a=x;
      b=y;
   }
   void disp(){
      cout<<"\na="<<a<<"\tb="<<b;
   }
};
int main(){
 Sample <int,float> S1(12,23.3);
 Sample <char,int> S2('N',12);
 S1.disp();
 S2.disp();
}
```

- To create a class template object, define the data type inside a **< >** at the time of object creation.
- className<int> classObj;
  className<float> classObj;

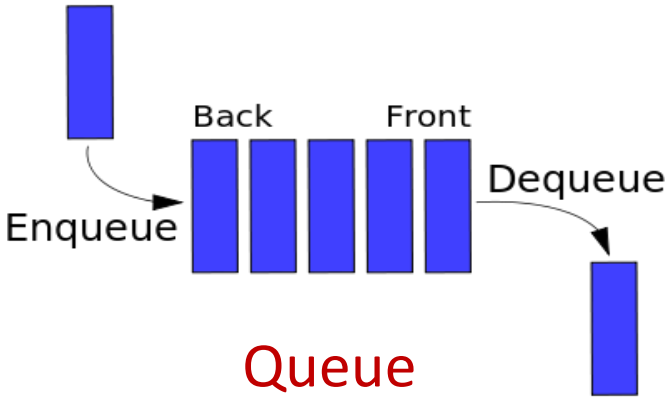
GTU Programs

1. Write a function template for finding the minimum value contained in an array.
2. Create a generic class stack using template and implement common Push and Pop operations for different data types.
3. Write program to swap Number using Function Template.

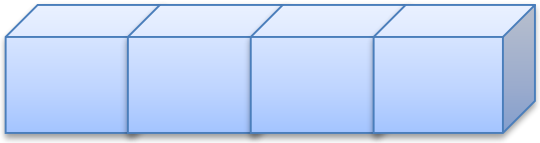
STL – Standard Template Library



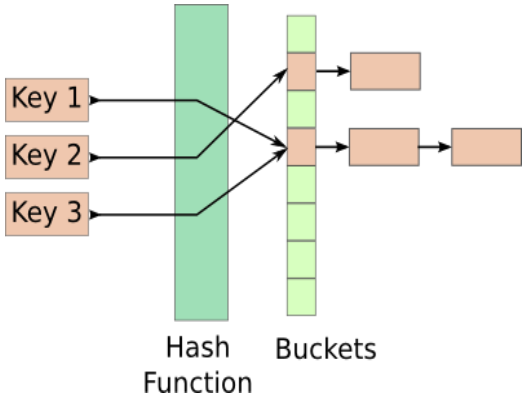
Stack



Queue



Vector



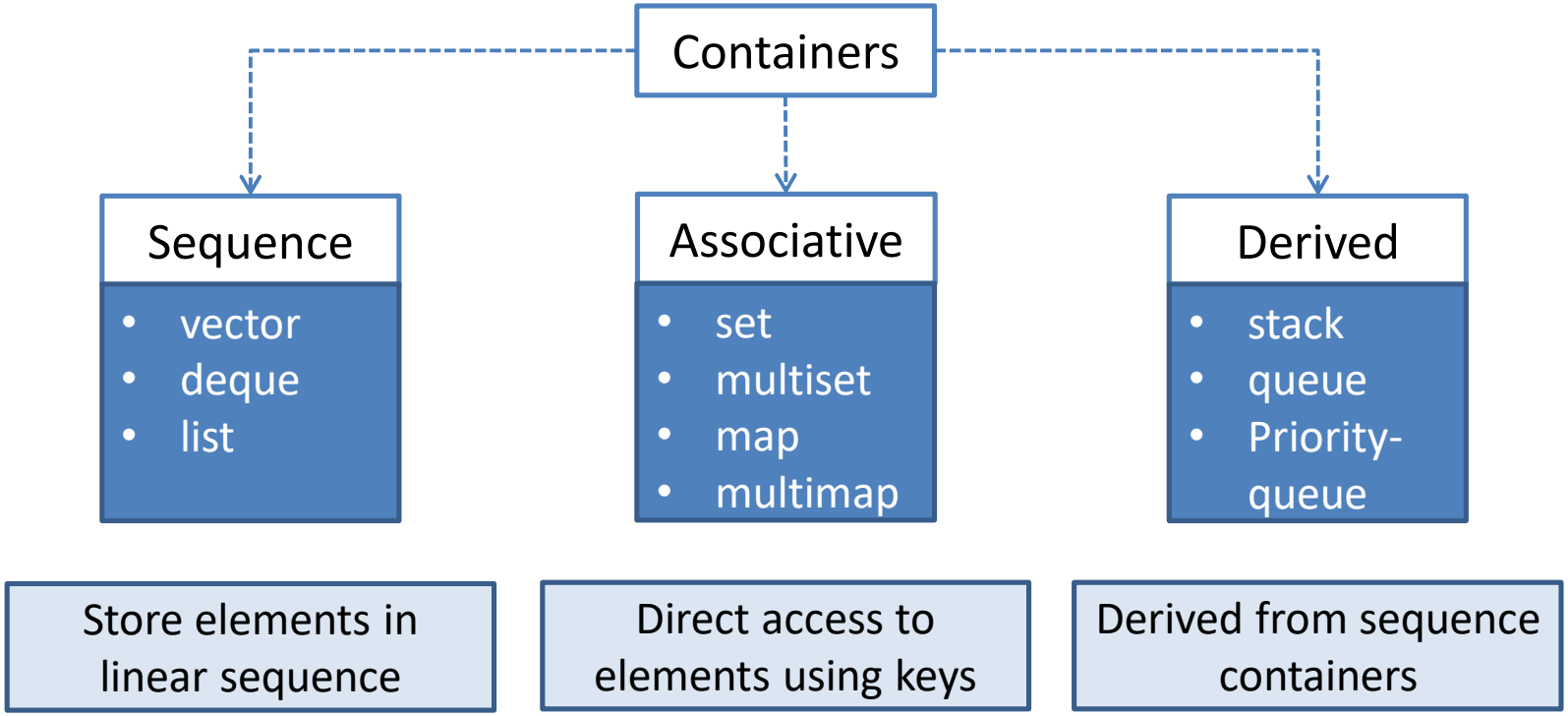
map

STL- Standard Template Library

- The C++ **STL** (Standard Template Library) is a powerful set of C++ template classes to provides general-purpose templatized classes and functions that implement many popular and commonly used algorithms and data structures like vectors, lists, queues, and stacks.
- There are three core components of STL as follows:
  1. Containers (an object to store data)
  2. Algorithms (procedure to process data)
  3. Iterators (pointer object to point elements in container)

STL- Containers

- A **container** is an object the actually stores data.
- The STL containers can be implemented by **class templates** to hold different data types.



STL Algorithms

- It is a procedure that is **used to process data** contained in containers.
- It includes algorithms that are used for initializing, searching, copying, sorting and merging.

- Mutating Sequence Algorithms

  like copy(), remove(), replace(), fill(), swap(), etc.,
- Non Modifying sequence Algorithms

  like find(), count(),search(), mismatch(), and equal()
- Numerical Algorithms

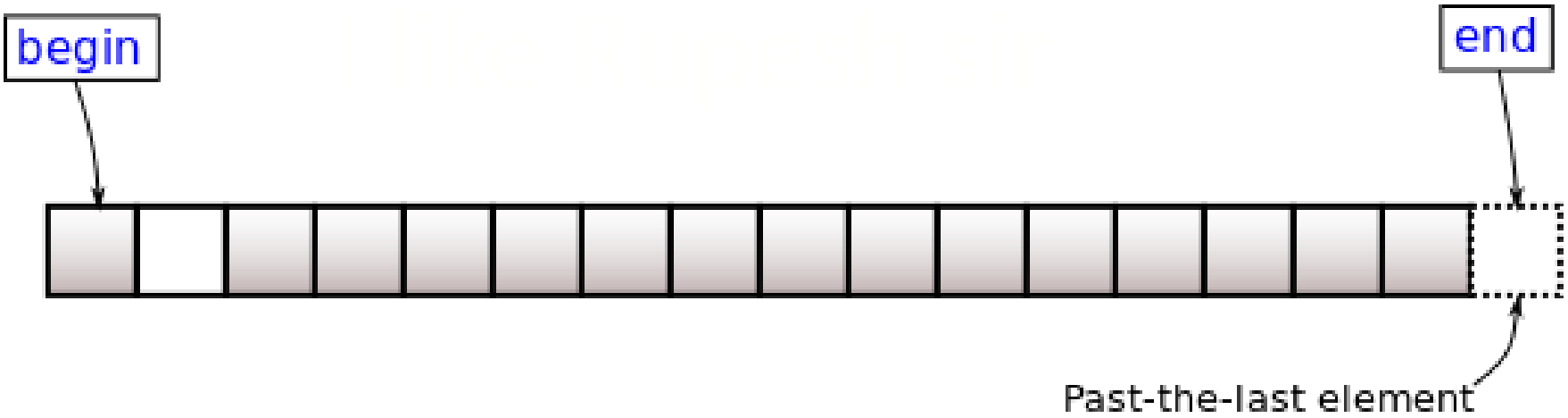
  accumulate(), partial_sum(), inner_product(), and adjacent_difference()

STL- Algorithms

- STL provide number of algorithms that can be used of any container, irrespective of their type. Algorithms library contains built in functions that performs complex algorithms on the data structures.
- For example: one can reverse a range with reverse() function, sort a range with sort() function, search in a range with binary_search() and so on.
- Algorithm library provides abstraction, i.e you don't necessarily need to know how the the algorithm works.

STL- Iterations

- Iterators behave like **pointers**.
- Iterators are used to **access container elements**.
- They are used to traverse from one element to another.



STL components

- STL provides numerous **containers** and **algorithms** which are very useful in completive programming , for example you can very easily define a linked list in a single statement by using list container of container library in STL , saving your time and effort.
- STL is a generic library , i.e a same **container** or **algorithm** can be operated on any data types , you don’t have to define the same algorithm for different type of elements.
- For example , sort algorithm will sort the elements in the given range irrespective of their data type , we don’t have to implement different sort algorithm for different datatypes.

Thank You