UNIT -V

18CSC202J - OBJECT ORIENTED DESIGN AND PROGRAMMING

Standard Template Library

What is stl???

- The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc.
- It is a library of container classes, algorithms, and iterators.
- It is a generalized library and so, its components are parameterized.
- A working knowledge of template classes is a prerequisite for working with STL.

Why use STL???

- STL offers an assortment of containers
- STL publicizes the time and storage complexity of its containers
- STL containers grow and shrink in size automatically
- STL provides built-in algorithms for processing containers
- STL provides iterators that make the containers and algorithms flexible and efficient.
- STL is extensible which means that users can add new containers and new algorithms such that:
  - STL algorithms can process STL containers as well as user defined containers
  - User defined algorithms can process STL containers as well user defined containers

The C++ Standard Template Libraries

- In 1990, Alex Stepanov and Meng Lee of Hewlett Packard Laboratories extended C++ with a library of class and function templates which has come to be known as the STL.
- In 1994, STL was adopted as part of ANSI/ISO Standard C++.

The C++ Standard Template Libraries

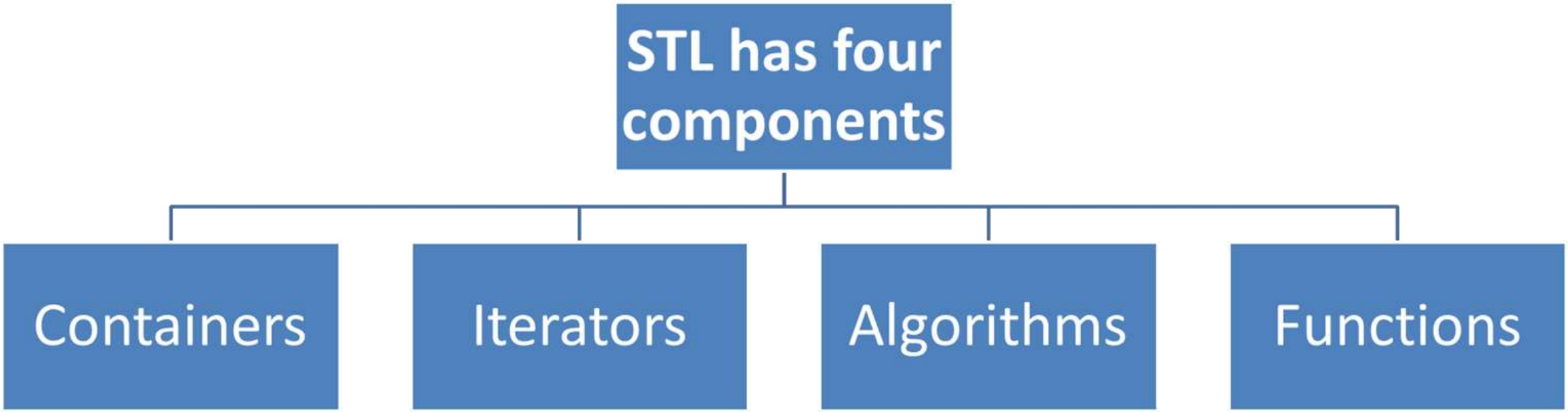
- STL had three basic components:
  - Containers

    Generic class templates for storing collection of data.
  - Algorithms

    Generic function templates for operating on containers.
  - Iterators

    Generalized ‘smart’ pointers that facilitate use of containers.

    They provide an interface that is needed for STL algorithms to operate on STL containers.
- **String abstraction was added during standardization.**


STL has four components
Containers
Iterators
Algorithms
Functions

STL Containers

Sequence and Associative Containers

Containers

- Containers or container classes store objects and data.
- There are in total seven standard “first-class” container classes and three container adaptor classes and only seven header files that provide access to these containers or container adaptors.
  - Sequence Containers
  - Container Adaptors
  - Associative Containers
  - Unordered Associative Containers

Sequence Containers

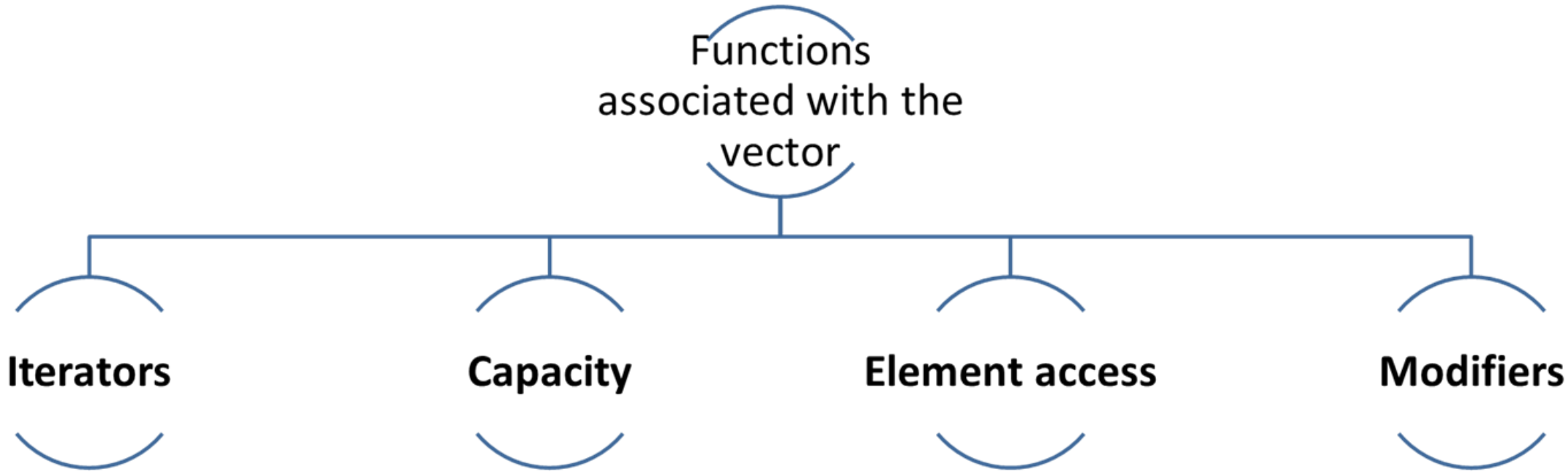
Implement data structures which can be accessed in a sequential manner.

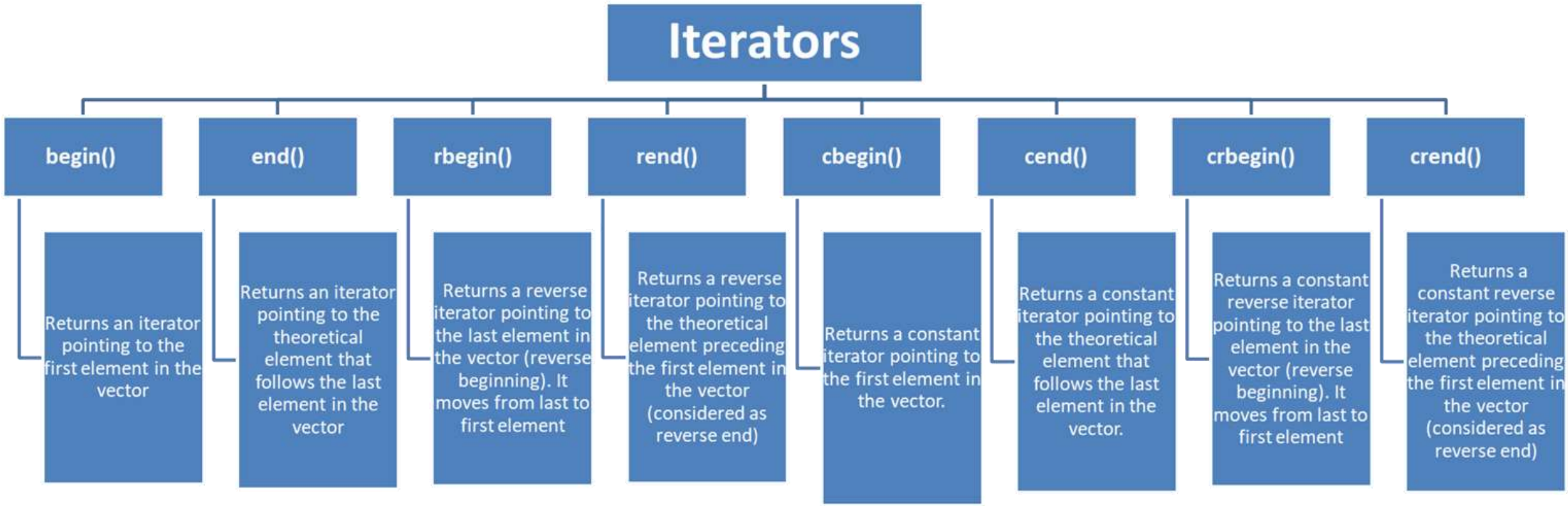
- vector
- list
- deque
- arrays
- forward-list

Topic : Sequence Container: Vector List

Sequence Containers: Vector

- Vectors are same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted, with their storage being handled automatically by the container.
- Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators. In vectors, data is inserted at the end.
- Inserting at the end takes differential time, as sometimes there may be a need of extending the array.
- Removing the last element takes only constant time because no resizing happens. Inserting and erasing at the beginning or in the middle is linear in time.


Functions associated with the vector
Iterators
Capacity
Element access
Modifiers

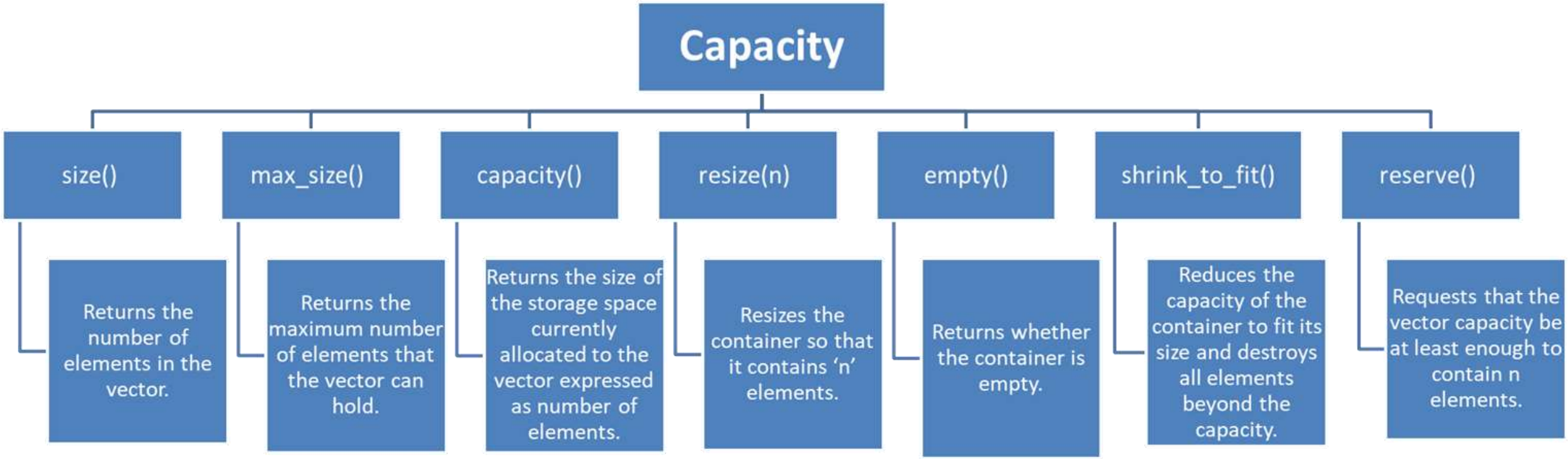

Iterators
begin()
Returns an iterator pointing to the first element in the vector
end()
Returns an iterator pointing to the theoretical element that follows the last element in the vector
rbegin()
Returns a reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element
rend()
Returns a reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)
cbegin()
Returns a constant iterator pointing to the first element in the vector.
cend()
Returns a constant iterator pointing to the theoretical element that follows the last element in the vector.
crbegin()
Returns a constant reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element
crend()
Returns a constant reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)

```cpp
// C++ program to illustrate the  iterators in vector
#include <iostream>
#include <vector>

using namespace std;

int main()
{
        vector<int> g1;

        for (int i = 1; i <= 5; i++)
                g1.push_back(i);

        cout << "Output of begin and end: ";
        for (auto i = g1.begin(); i != g1.end(); ++i)
                cout << *i << " ";

        cout << "\nOutput of cbegin and cend: ";
        for (auto i = g1.cbegin(); i != g1.cend(); ++i)
                cout << *i << " ";

        cout << "\nOutput of rbegin and rend: ";
        for (auto ir = g1.rbegin(); ir != g1.rend(); ++ir)
                cout << *ir << " ";

        cout << "\nOutput of crbegin and crend : ";
        for (auto ir = g1.crbegin(); ir != g1.crend(); ++ir)
                cout << *ir << " ";

        return 0;
}
```

Output:

```
Output of begin and end: 1 2 3 4 5
Output of cbegin and cend: 1 2 3 4 5
Output of rbegin and rend: 5 4 3 2 1
Output of crbegin and crend : 5 4 3 2 1
```

functions associated with the vector

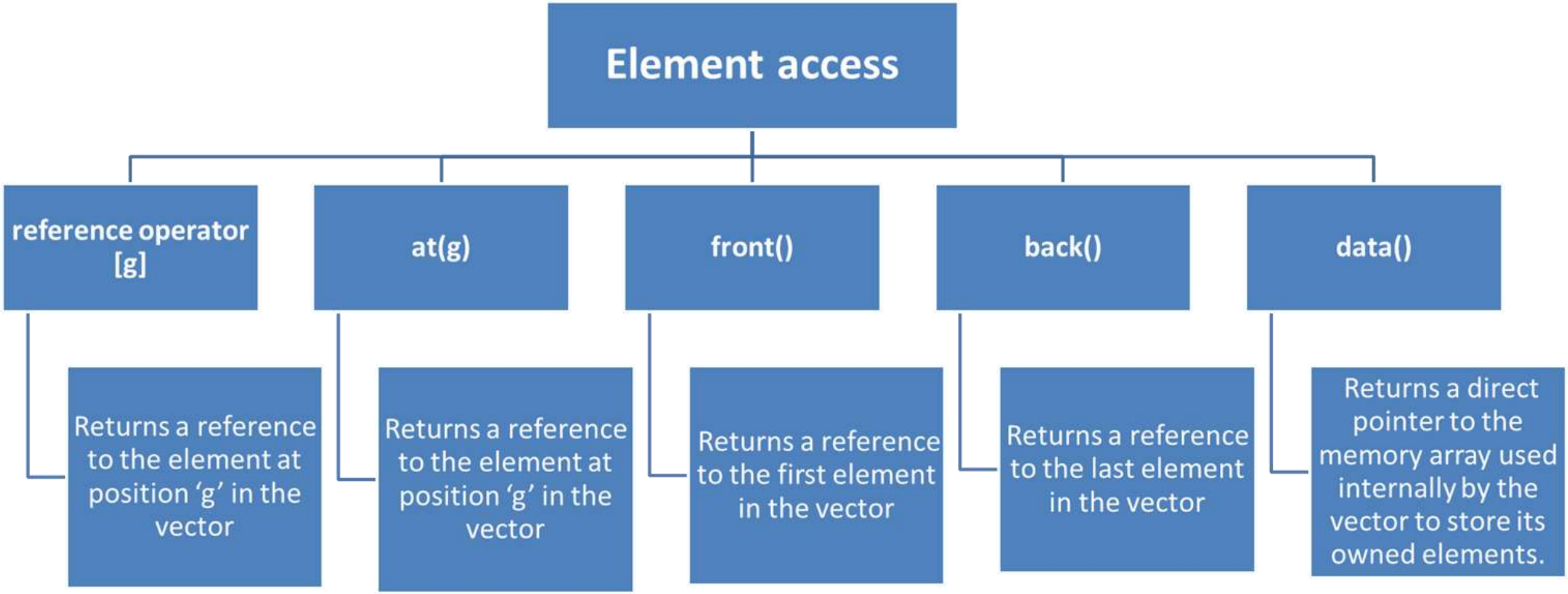


```
// C++ program to illustrate the  capacity function in vector
#include <iostream>
#include <vector>

using namespace std;

int main()
{
		vector<int> g1;

		for (int i = 1; i <= 5; i++)
			g1.push_back(i);

		cout << "Size : " << g1.size();
		cout << "\nCapacity : " << g1.capacity();
		cout << "\nMax_Size : " << g1.max_size();

		// resizes the vector size to 4
		g1.resize(4);

		// prints the vector size after resize()
		cout << "\nSize : " << g1.size();

		// checks if the vector is empty or not
		if (g1.empty() == false)
			cout << "\nVector is not empty";
		else
			cout << "\nVector is empty";

		// Shrinks the vector
		g1.shrink_to_fit();
		cout << "\nVector elements are: ";
		for (auto it = g1.begin(); it != g1.end(); it++)
			cout << *it << " ";

		return 0;
}
```

Output:

```
Size : 5
Capacity : 8
 Max_Size : 4611686018427387903
 Size : 4
Vector is not empty
Vector elements are: 1 2 3 4
```

functions associated with the vector

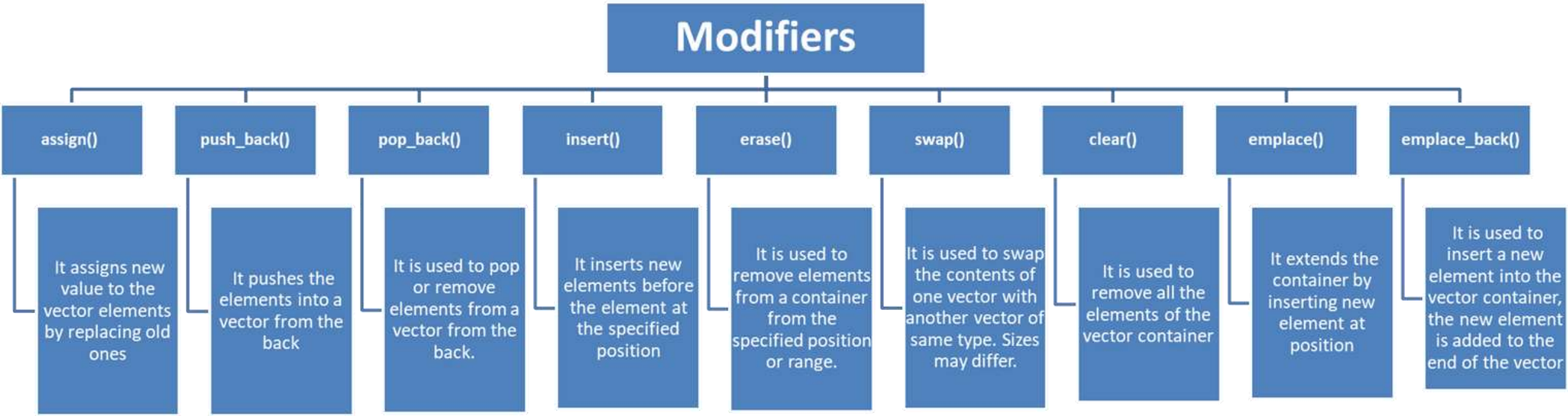


```
// C++ program to illustrate the  element accesser in vector
#include <bits/stdc++.h>
using namespace std;

int main()
{
        vector<int> g1;

        for (int i = 1; i <= 10; i++)
                g1.push_back(i * 10);

        cout << "\nReference operator [g] : g1[2] = " << g1[2];

        cout << "\nat : g1.at(4) = " << g1.at(4);

        cout << "\nfront() : g1.front() = " << g1.front();

        cout << "\nback() : g1.back() = " << g1.back();

        // pointer to the first element
        int* pos = g1.data();

        cout << "\nThe first element is " << *pos;
        return 0;
}
```

Output:

```
Reference operator [g] : g1[2] = 30
at : g1.at(4) = 50
front() : g1.front() = 10
back() : g1.back() = 100
The first element is 10
```

functions associated with the vector



Sequence Container: List

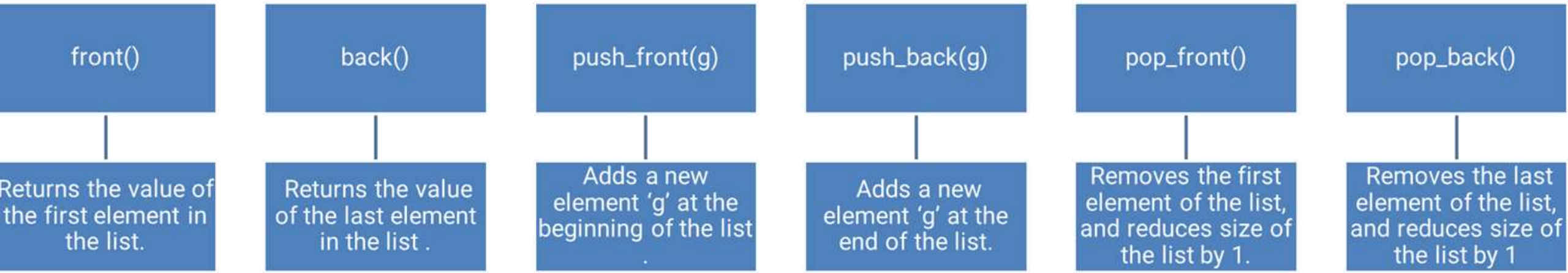
- Lists are sequence containers that allow non-contiguous memory allocation.
- As compared to vector, list has slow traversal, but once a position has been found, insertion and deletion are quick.
- Normally, when we say a List, we talk about doubly linked list. For implementing a singly linked list, we use forward list.

```
#include <iostream>
#include <list>
#include <iterator>
using namespace std;
//function for printing the elements in a list
void showlist(list <int> g)
{
		list <int> :: iterator it;
		for(it = g.begin(); it != g.end(); ++it)
		cout << '\t' << *it;
		cout << '\n';
}
int main()
{
list <int> gqlist1, gqlist2;
for (int i = 0; i < 10; ++i)
{
		gqlist1.push_back(i * 2);
		gqlist2.push_front(i * 3);
}
cout << "\nList 2 (gqlist2) is : ";
cout << "\nList 1 (gqlist1) is : ";
showlist(gqlist1);
showlist(gqlist2);
cout << "\ngqlist1.front() : " << gqlist1.front();
cout << "\ngqlist1.back() : " << gqlist1.back();
cout << "\ngqlist1.pop_front() : ";
gqlist1.pop_front();
showlist(gqlist1);
cout << "\ngqlist2.pop_back() : ";
gqlist2.pop_back();
showlist(gqlist2);
cout << "\ngqlist1.reverse() : ";
gqlist1.reverse();
showlist(gqlist1);
cout << "\ngqlist2.sort(): ";
gqlist2.sort();
showlist(gqlist2);
return 0;
}
```

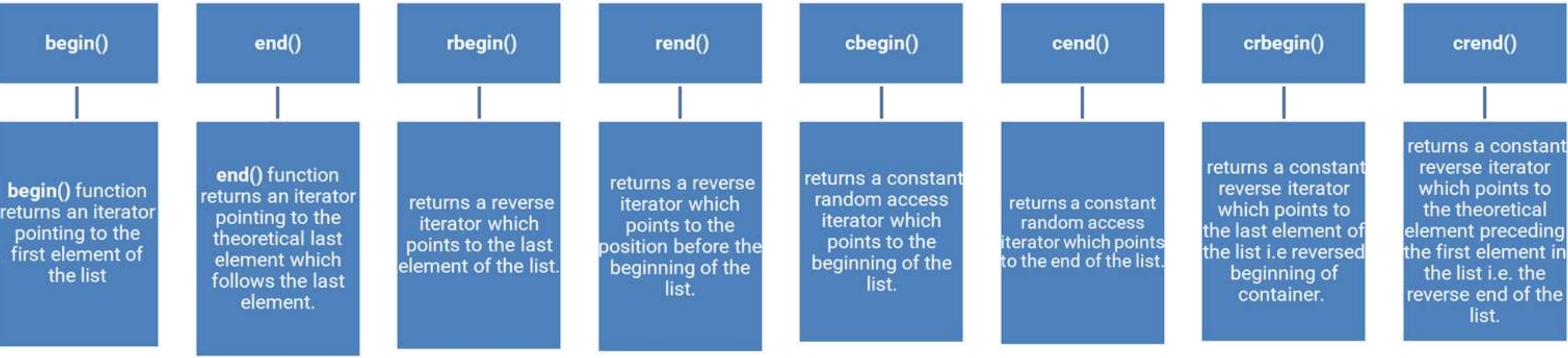
The output of the above program is :

```
List 1 (gqlist1) is : 0 2 4 6 8 10 12 14 16 18
List 2 (gqlist2) is : 27 24 21 18 15 12 9 6 3 0
gqlist1.front() : 0
gqlist1.back() : 18
gqlist1.pop_front() : 2 4 6 8 10 12 14 16 18
gqlist2.pop_back() : 27 24 21 18 15 12 9 6 3
gqlist1.reverse() : 18 16 14 12 10 8 6 4 2
gqlist2.sort(): 3 6 9 12 15 18 21 24 27
```

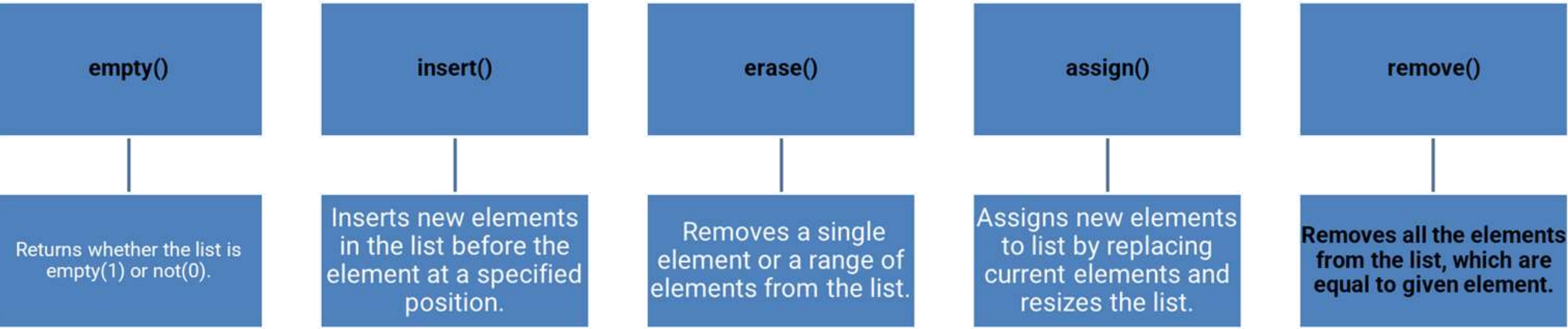
functions associated with the Lists



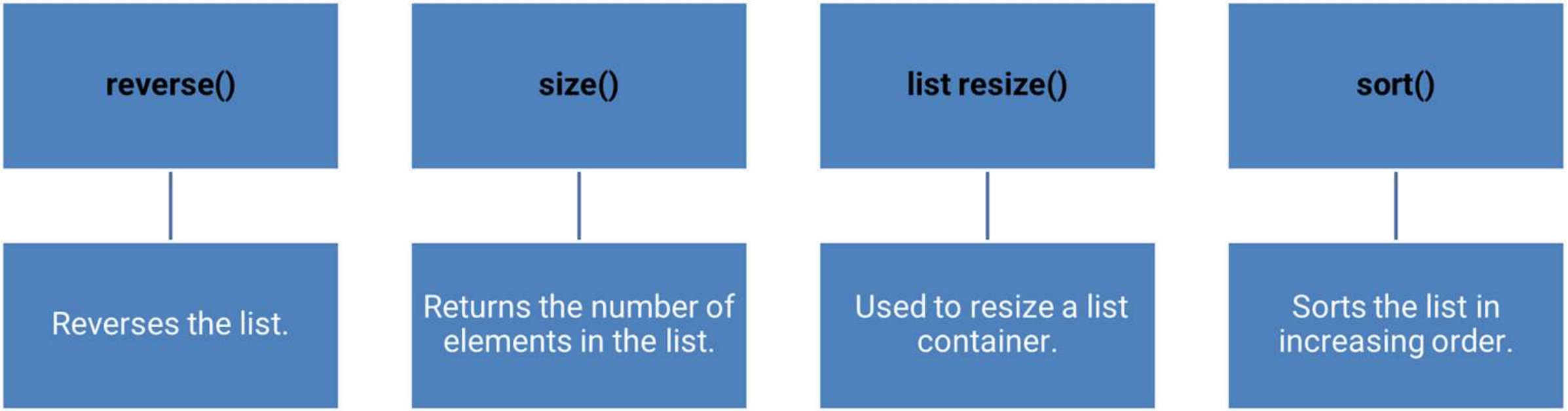
functions associated with the Lists



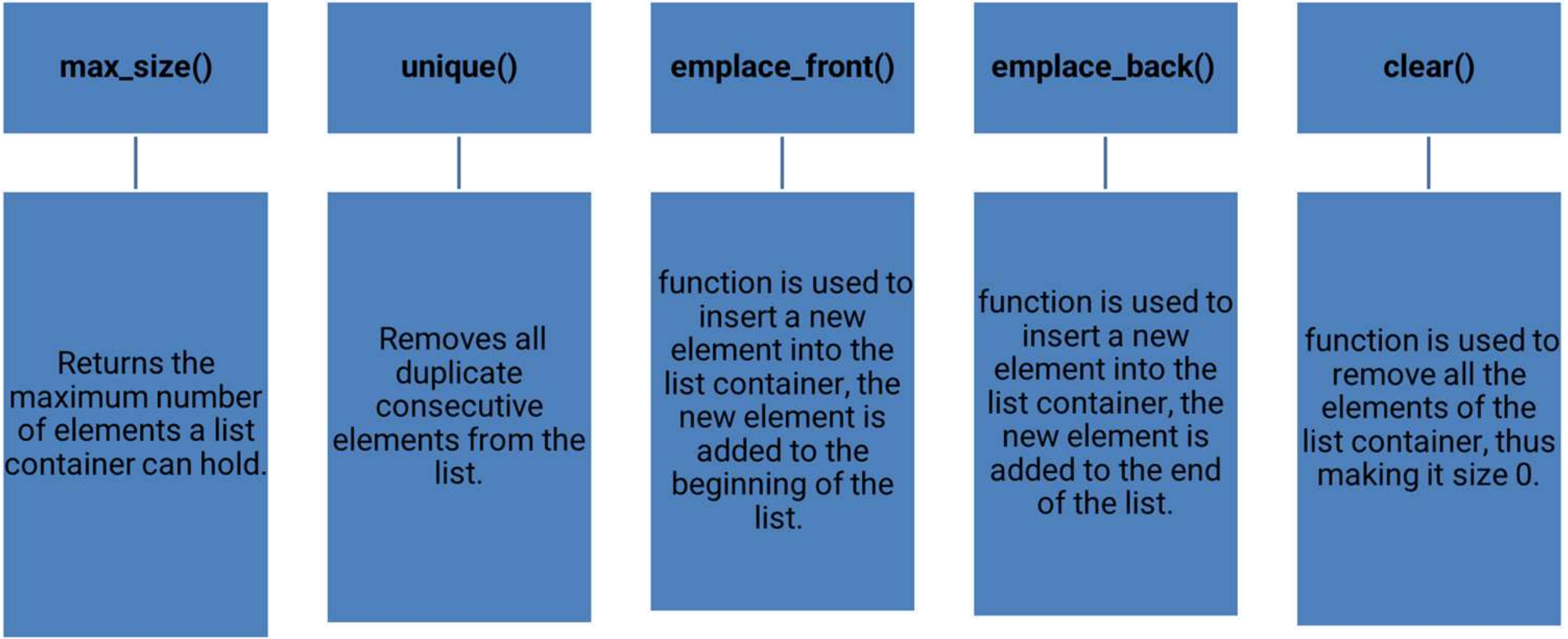
functions associated with the Lists



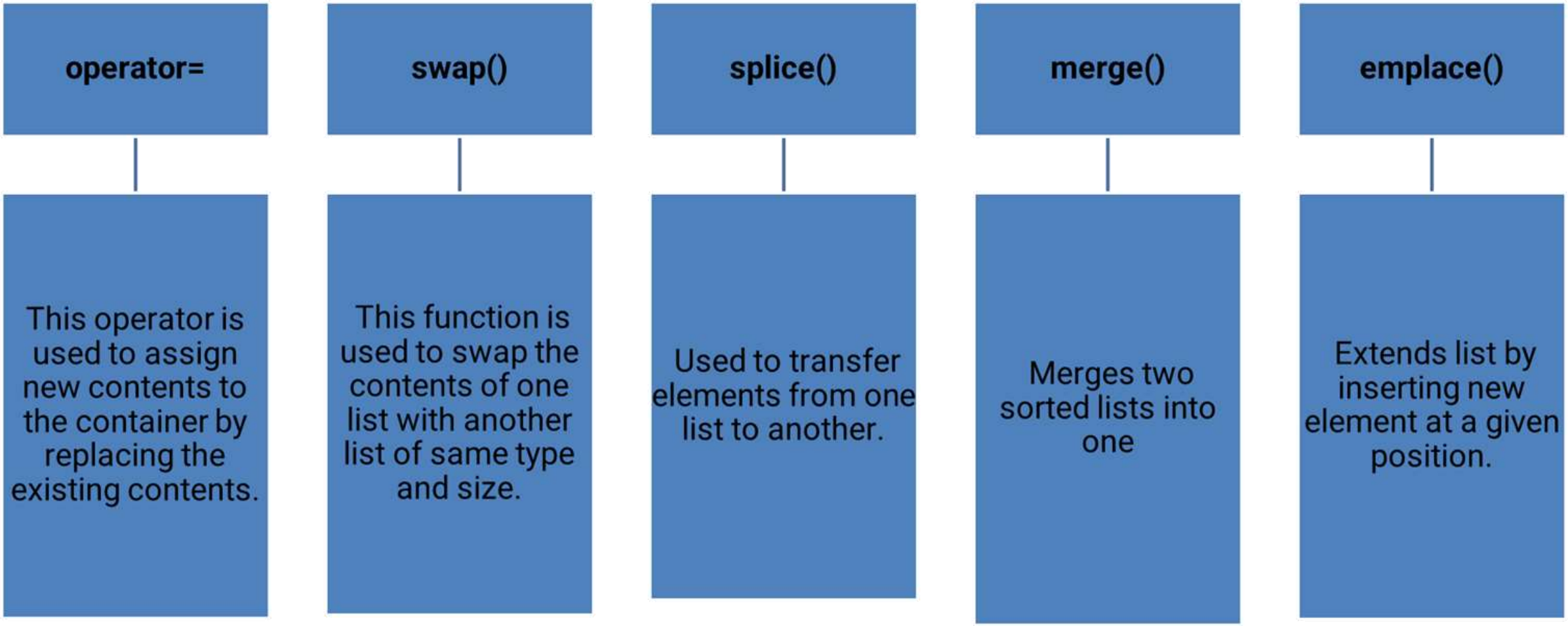
functions associated with the Lists



functions associated with the Lists



functions associated with the Lists



MCQ

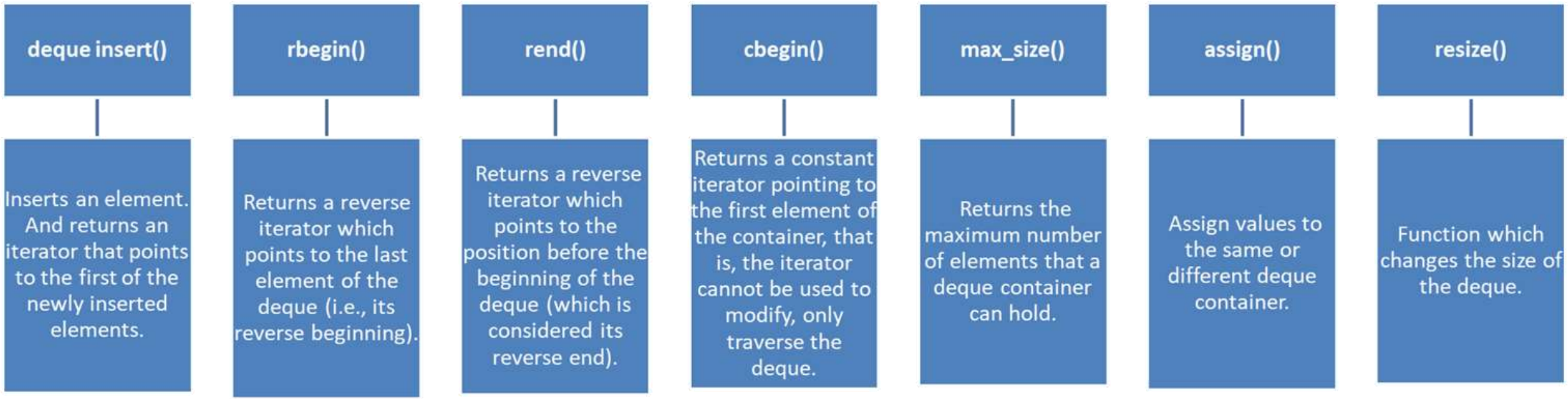
- 1. In which classes we can define the list and vector classes?
  A. Abstract classes
  B. child classes
  C. STL classes
  D. String classes
- 2. Which of the following are the components of STL?
  A. Algorithms
  B. containers
  C. function, iterators
  D. All of these

- 3. Which of the following is to provide a different interface for sequential containers?
  A. container adopters
  B. sequence containers
  C. queue
  D. Associative Containers
- 4. By STL how many components it has been kept?
  A. 3
  B. 4
  C.1
  D. unlimited

Sequence Containers: Deque

- Double ended queues are sequence containers with the feature of expansion and contraction on both the ends.
- They are similar to vectors, but are more efficient in case of insertion and deletion of elements. Unlike vectors, contiguous storage allocation may not be guaranteed.
- Double Ended Queues are basically an implementation of the data structure double ended queue. A queue data structure allows insertion only at the end and deletion from the front.
- This is like a queue in real life, wherein people are removed from the front and added at the back. Double ended queues are a special case of queues where insertion and deletion operations are possible at both the ends.
- The functions for deque are same as vector, with an addition of push and pop operations for both front and back.

Methods of Deque

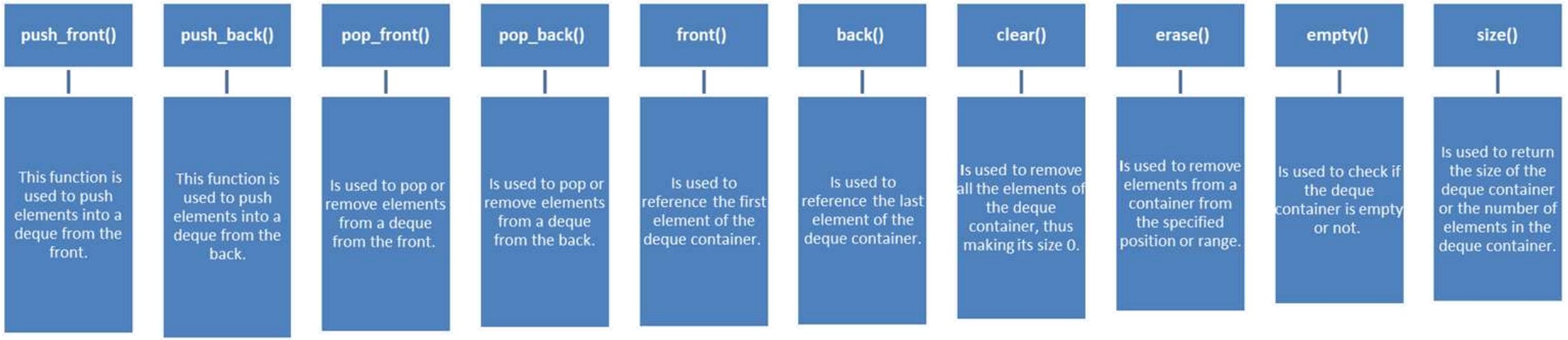


```
#include <iostream>
#include <deque>
using namespace std;

void showdq(deque <int> g)
{
        deque <int> :: iterator it;
        for (it = g.begin(); it != g.end(); ++it)
                cout << '\t' << *it;
        cout << '\n';
}

int main()
{
        deque <int> gquiz;
        gquiz.push_back(10);
        gquiz.push_front(20);
        gquiz.push_back(30);
        gquiz.push_front(15);
        cout << "The deque gquiz is : ";
        showdq(gquiz);

        cout << "\ngquiz.size() : " << gquiz.size();
        cout << "\ngquiz.max_size() : " << gquiz.max_size();

        cout << "\ngquiz.at(2) : " << gquiz.at(2);
        cout << "\ngquiz.front() : " << gquiz.front();
        cout << "\ngquiz.back() : " << gquiz.back();

        cout << "\ngquiz.pop_front() : ";
        gquiz.pop_front();
        showdq(gquiz);

        cout << "\ngquiz.pop_back() : ";
        gquiz.pop_back();
        showdq(gquiz);

        return 0;
}
```

OUTPUT:

```
The deque gquiz is : 15 20 10 30
gquiz.size() : 4
gquiz.max_size() : 4611686018427387903
gquiz.at(2) : 10
gquiz.front() : 15
gquiz.back() : 30
gquiz.pop_front() : 20 10 30
gquiz.pop_back() : 20 10
```

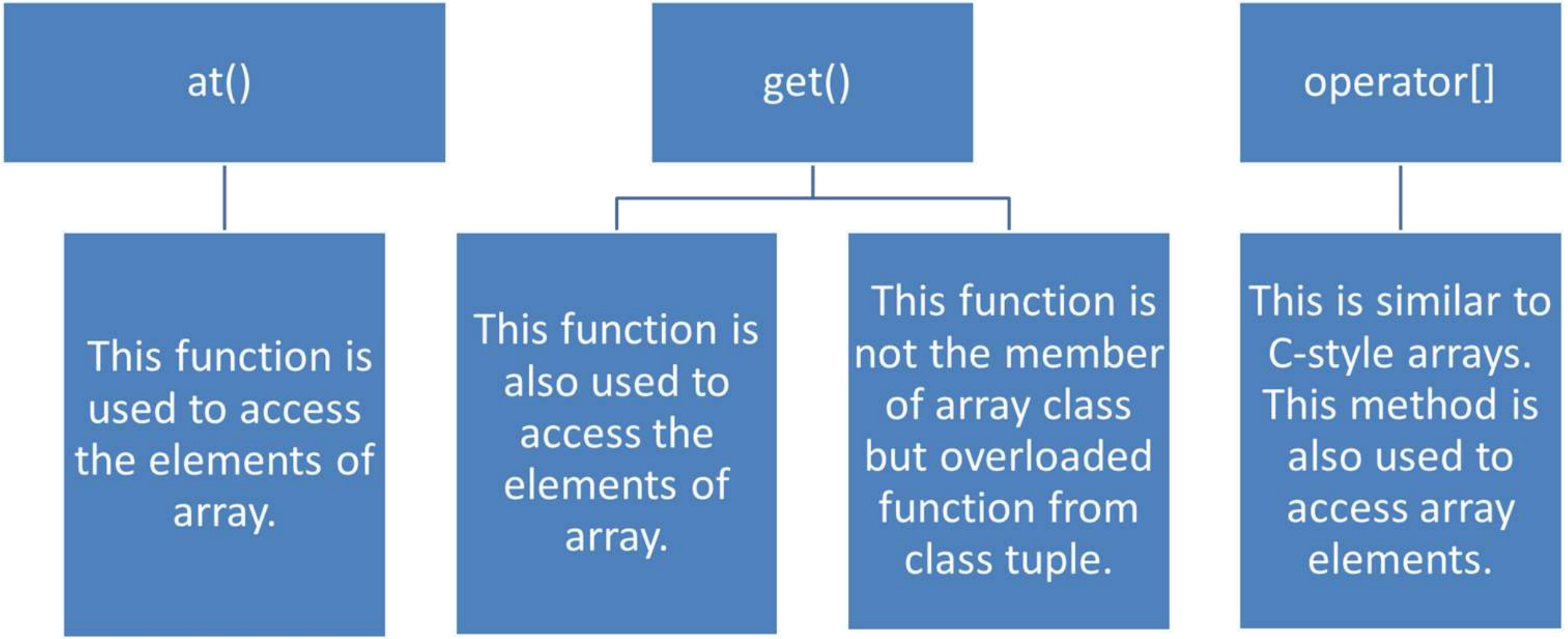
Methods of Deque



Sequence Containers: Array

- The introduction of array class from C++11 has offered a better alternative for C-style arrays. The advantages of array class over C-style array are :-
- Array classes knows its own size, whereas C-style arrays lack this property. So when passing to functions, we don’t need to pass size of Array as a separate parameter.
- With C-style array there is more risk of array being decayed into a pointer. Array classes don’t decay into pointers
- Array classes are generally more efficient, light-weight and reliable than C-style arrays.
-

Operations on array



Array get() function in C++ STL

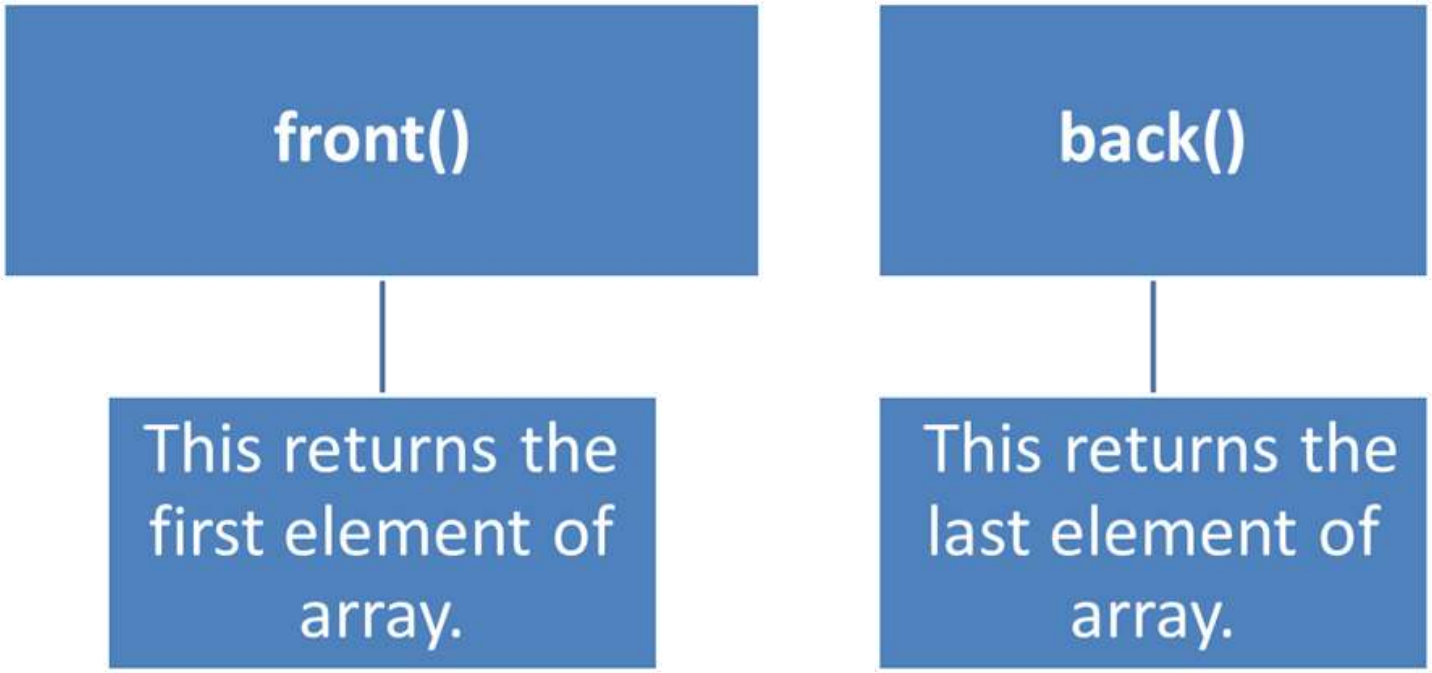
- The **array::get()** is a built-in function in C++ STL which returns a reference to the i-th element of the array container.
- **Syntax:**
- get(array_name) **Parameters:** The function accepts two mandatory parameters which are described below.
- i – position of an element in the array, with 0 as the position of the first element.
- arr_name – an array container.
- **Return Value:** The function returns a reference to the element at the specified position in the array
- **Time complexity:** O(1)

```
// // C++ code to demonstrate working of array,  to() and get()
#include<iostream>
#include<array> // for array, at()
#include<tuple> // for get()
using namespace std;
int main()
{
        // Initializing the array elements
        array<int,6> ar = {1, 2, 3, 4, 5, 6};

        // Printing array elements using at()
        cout << "The array elements are (using at()) : ";
        for ( int i=0; i<6; i++)
        cout << ar.at(i) << " ";
        cout << endl;

        // Printing array elements using get()
        cout << "The array elements are (using get()) : ";
        cout << get<0>(ar) << " " << get<1>(ar) << " ";
        cout << get<2>(ar) << " " << get<3>(ar) << " ";
        cout << get<4>(ar) << " " << get<5>(ar) << " ";
        cout << endl;

        // Printing array elements using operator[]
        cout << "The array elements are (using operator[]) : ";
        for ( int i=0; i<6; i++)
        cout << ar[i] << " ";
        cout << endl;

        return 0; }
```

Output:

```
The array elements are (using at()) : 1 2 3 4 5 6
The array elements are (using get()) : 1 2 3 4 5 6
The array elements are (using operator[]) : 1 2 3 4
5 6
```

Operations on array

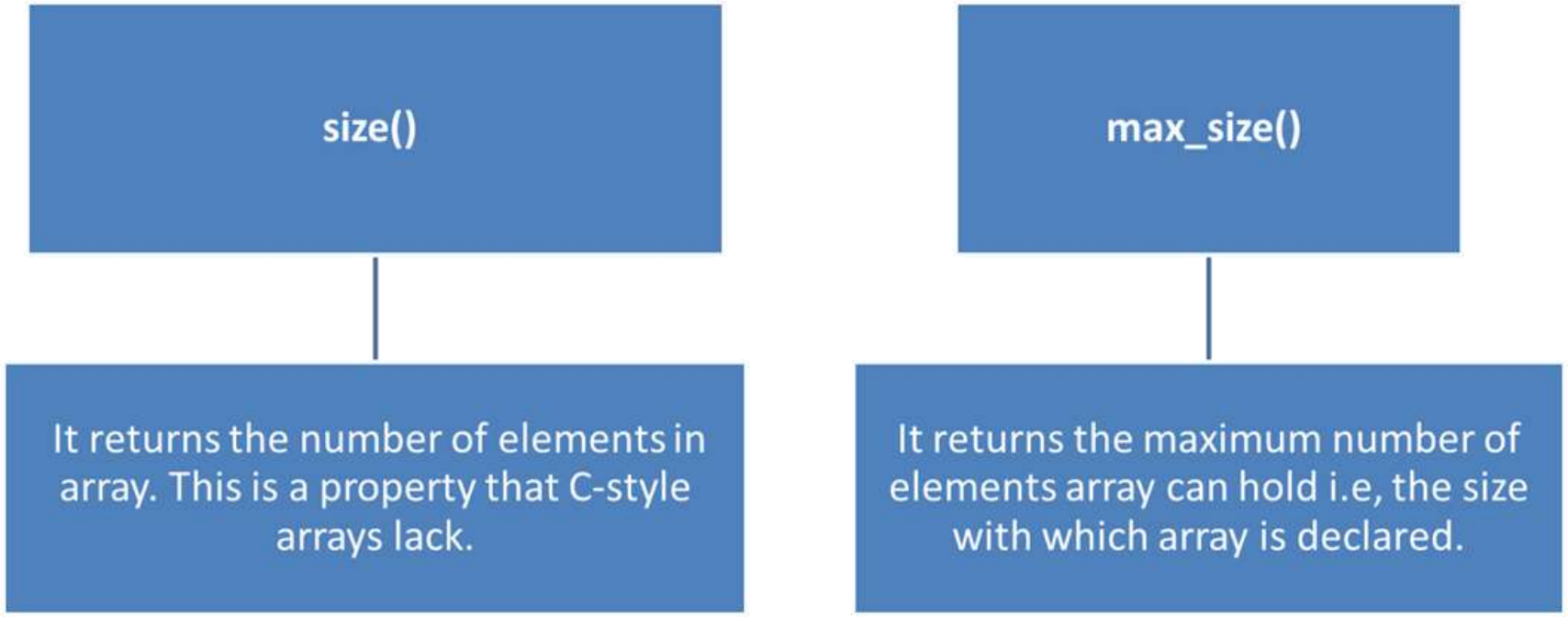


```cpp
// C++ code to demonstrate working of  front() and back()
#include<iostream>
#include<array> // for front() and back()
using namespace std;
int main()
{
        // Initializing the array elements
        array<int,6> ar = {1, 2, 3, 4, 5, 6};

        // Printing first element of array
        cout << "First element of array is : ";
        cout << ar.front() << endl;

        // Printing last element of array
        cout << "Last element of array is : ";
        cout << ar.back() << endl;

        return 0;

}
```

Output:

```
First element of array is : 1
Last element of array is : 6
```

Operations on array

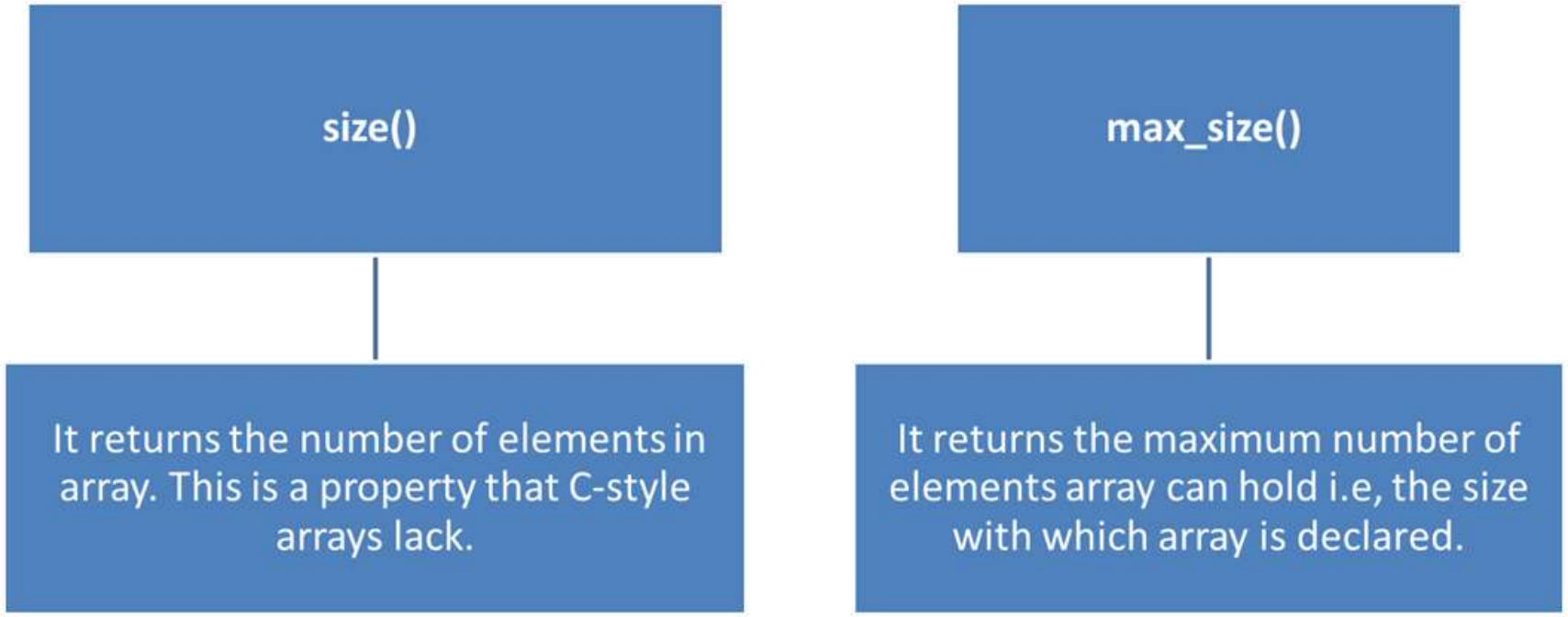


```
// C++ code to demonstrate working of  size() and max_size()
#include<iostream>
#include<array> // for size() and max_size()
using namespace std;
int main()
{
        // Initializing the array elements
        array<int,6> ar = {1, 2, 3, 4, 5, 6};

        // Printing number of array elements
        cout << "The number of array elements is : ";
        cout << ar.size() << endl;

        // Printing maximum elements array can hold
        cout << "Maximum elements array can hold is : ";
        cout << ar.max_size() << endl;

        return 0;

}
```

Output:

```
The number of array elements is : 6
Maximum elements array can hold is : 6
```

Operations on array



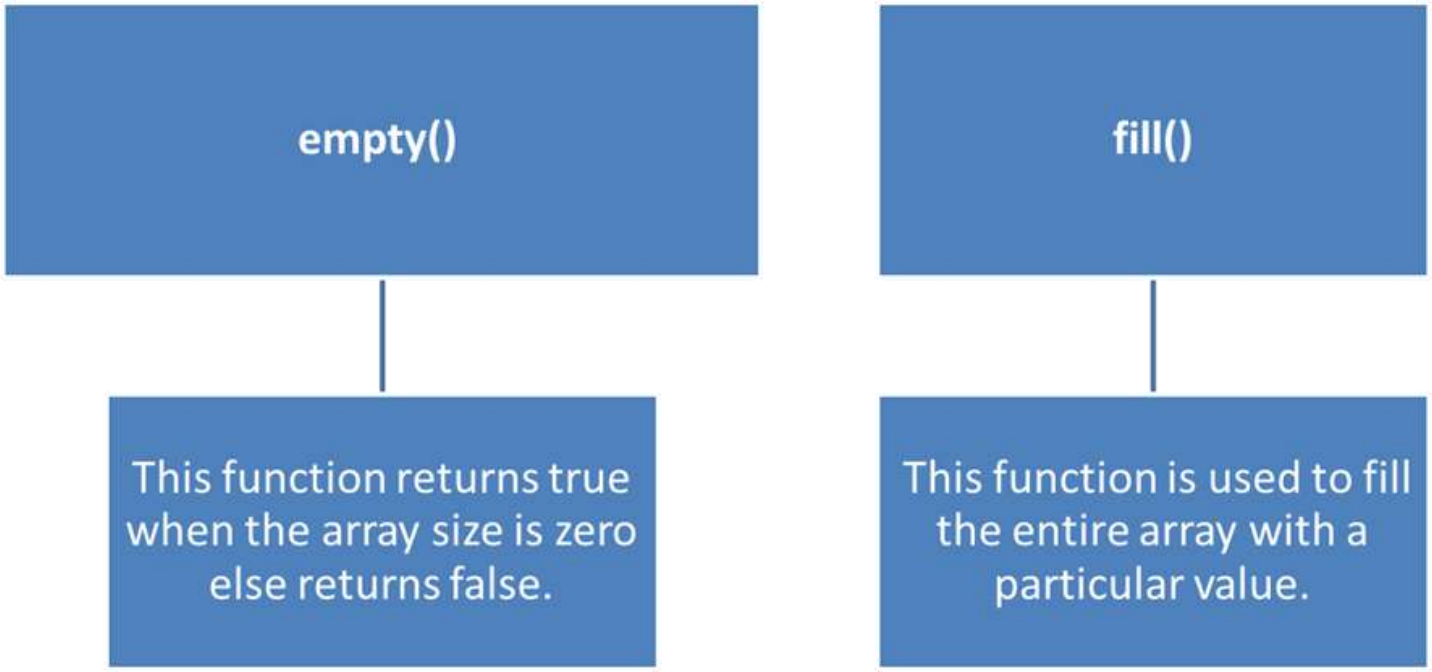
swap() :- The swap() swaps all elements of one array with other.

```
// C++ code to demonstrate working of swap()
#include<iostream>
#include<array> // for swap() and array
using namespace std;
int main()
{

        // Initializing 1st array
        array<int,6> ar = {1, 2, 3, 4, 5, 6};

        // Initializing 2nd array
        array<int,6> ar1 = {7, 8, 9, 10, 11, 12};

        // Printing 1st and 2nd array before swapping
        cout << "The first array elements before swapping are :
";
        for (int i=0; i<6; i++)
        cout << ar[i] << " ";
        cout << endl;
        cout << "The second array elements before swapping
are : ";
for (int i=0; i<6; i++)
cout << ar1[i] << " ";
cout << endl;
// Swapping ar1 values with ar
ar.swap(ar1);

// Printing 1st and 2nd array after swapping
cout << "The first array elements after swapping are : ";
for (int i=0; i<6; i++)
cout << ar[i] << " ";
cout << endl;
cout << "The second array elements after swapping are : ";
for (int i=0; i<6; i++)
cout << ar1[i] << " ";
cout << endl;
return 0;
}
```

Output:

```
The first array elements before swapping are : 1 2 3 4 5 6
 The second array elements before swapping are : 7 8 9 10 11 12
The first array elements after swapping are : 7 8 9 10 11 12
The second array elements after swapping are : 1 2 3 4 5 6
```

Operations on array



```
// C++ code to demonstrate working of empty() and fill()
#include<iostream>
#include<array> // for fill() and empty()
using namespace std;
int main()
{
                array<int,6> ar;      // Declaring 1st array
                array<int,0> ar1;    // Declaring 2nd array
               ar1.empty()? cout << "Array empty":
               cout << "Array not empty";
               cout << endl;        // Checking size of array if it is empty
               // Filling array with 0
               ar.fill(0);
               // Displaying array after filling
               cout << "Array after filling operation is : ";
               for ( int i=0; i<6; i++)
                              cout << ar[i] << " ";
               return 0;
}
```

Output:

```
Array empty
Array after filling operation is : 0 0 0 0 0 0
```

MCQ

1.Which of the following does not support any insertion or deletion?
a) Array
b) Vector
c) Dequeue
d) List

2) Which of the following header file is required to use deque container in C++?
a)<queue>
b)<deque>
c)<dqueue>
d)<cqueue>

3)What is the correct output of the given code snippets?

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

int main()
{
    deque<int> d;

    d.add(10);
    d.add(20);
    d.add(30);

    for (int i = 0; i < d.size(); i++) {
        cout << d[i] << " ";
    }

    return 0;
}
```

a)10 20 30
b)Garbage Value
c)Syntax error
d)Runtime error

4.Which of the following class template are based on arrays?
a) vector
b) list
c) dequeue
d) both vector & dequeue

Topic : STL Stack

STL Stack

- Stacks are a type of container adaptors with LIFO(Last In First Out) type of working, where a new element is added at one end and (top) an element is removed from that end only.

- Stack uses an encapsulated object of either vector or deque (by default) or list (sequential container class) as its underlying container, providing a specific set of member functions to access its elements.

Stack Syntax:-

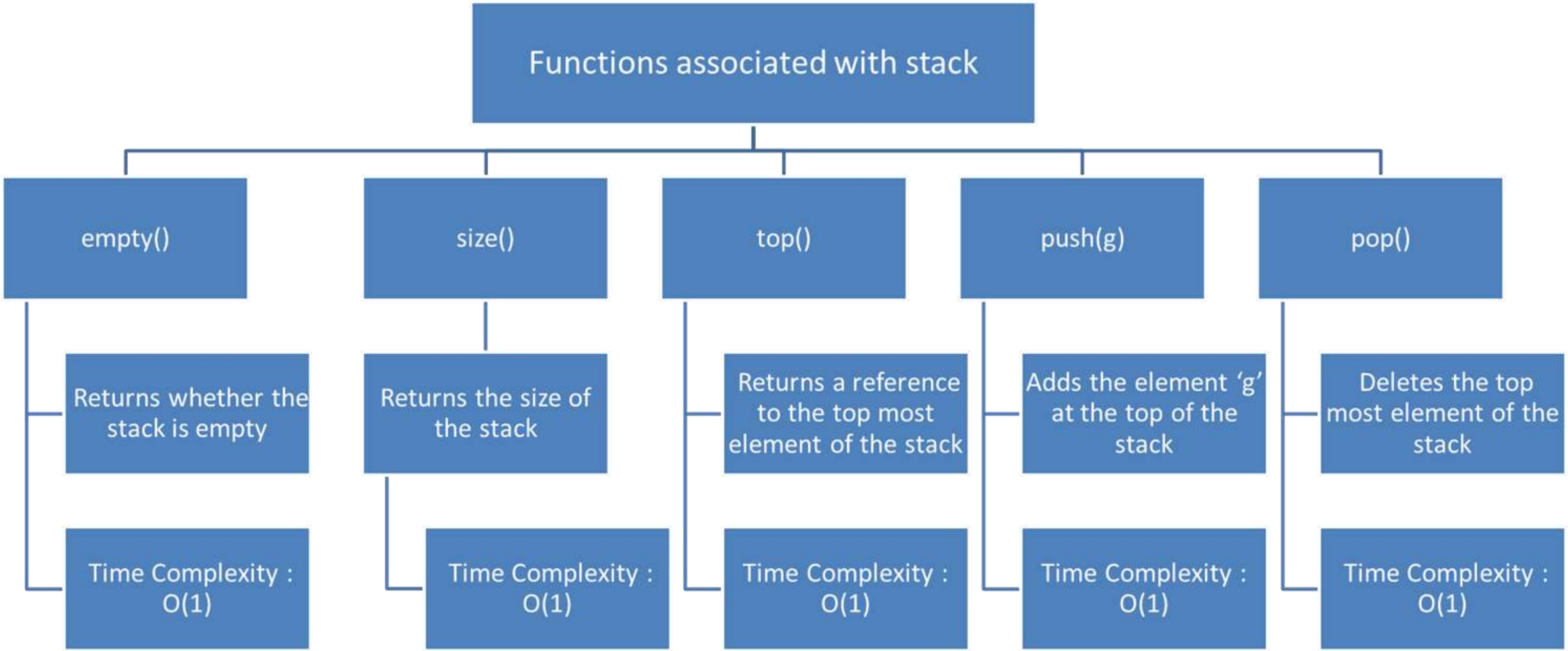
- For creating a stack, we must include the <stack> header file in our code. We then use this syntax to define the std::stack:

- template <class Type, class Container = deque<Type> > class stack;**Type** – is the Type of element contained in the std::stack. It can be any valid C++ type or even a user-defined type.

Example:

```
#include <iostream>
#include <stack>
using namespace std;
int main() {
    stack<int> stack;
    stack.push(21);
    stack.push(22);
   stack.push(24);
    stack.push(25);

stack.pop();
    stack.pop();
    while (!stack.empty()) {
        cout << stack.top() <<" ";
        stack.pop();
    }
}
```

OUTPUT : 22 21


Functions associated with stack
empty()
size()
top()
push(g)
pop()
Returns whether the stack is empty
Time Complexity : O(1)
Returns the size of the stack
Time Complexity : O(1)
Returns a reference to the top most element of the stack
Time Complexity : O(1)
Adds the element ‘g’ at the top of the stack
Time Complexity : O(1)
Deletes the top most element of the stack
Time Complexity : O(1)

```
// CPP program to demonstrate working of STL stack
#include <iostream>
#include <stack>
using namespace std;

void showstack(stack <int> s)
{
        while (!s.empty())
        {
                cout << '\t' << s.top();
                s.pop();
        }
        cout << '\n';
}

int main ()
{
        stack <int> s;
        s.push(10);
        s.push(30);
        s.push(20);
        s.push(5);
        s.push(1);

        cout << "The stack is : ";
        showstack(s);

        cout << "\ns.size() : " << s.size();
        cout << "\ns.top() : " << s.top();


        cout << "\ns.pop() : ";
        s.pop();
        showstack(s);

        return 0;
}
```

Output:

```
The stack is : 1 5 20 30 10
s.size() : 5
s.top() : 1
s.pop() : 5 20 30 10
```

List of functions of Stack

stack::top() top() function is used to reference the top(or the newest) element of the stack.

Syntax :

stackname.top()

Parameters: No value is needed to pass as the parameter.

Return Value: Direct reference to the top element of the stack container.

OUTPUT : 20

Time Complexity: O(n)

Auxiliary Space: O(n)

```
// Application of top() function
#include <iostream>
#include <stack>
using namespace std;

int main()
{
   int sum = 0;
   stack<int> mystack;
   mystack.push(1);
   mystack.push(8);
   mystack.push(3);
   mystack.push(6);
   mystack.push(2);
   // Stack becomes 1, 8, 3, 6, 2
  while (!mystack.empty()
sum = sum + mystack.top();
  mystack.pop();
   }
   cout << sum;
   return 0;
}
```

1. What is the Standard Template Library?
    a) Set of C++ template classes to provide common programming data structures and functions
    b) Set of C++ classes
    c) Set of Template functions used for easy data structures implementation
    d) Set of Template data structures only.

Answer: a
STL expanded as Standard Template Library is set of C++ template classes to provide common programming data structures and functions.

2. How many components STL has?
    a) 1
    b) 2
    c) 3
    d) 4

Answer : d

3. What are Container Adaptors?
    a) Containers that implements data structures which can be accessed sequentially
    b) Containers that implements sorted data structures for fast search in O(logn)
    c) Containers that implements unsorted(hashed) data structures for quick search in O(1)
    d) Containers that provide a different interface for sequential containers.

Answer: d
Container Adaptors is the subset of Containers that provides a different interface for sequential containers.

Topic :Associative Containers: Map, Multimap

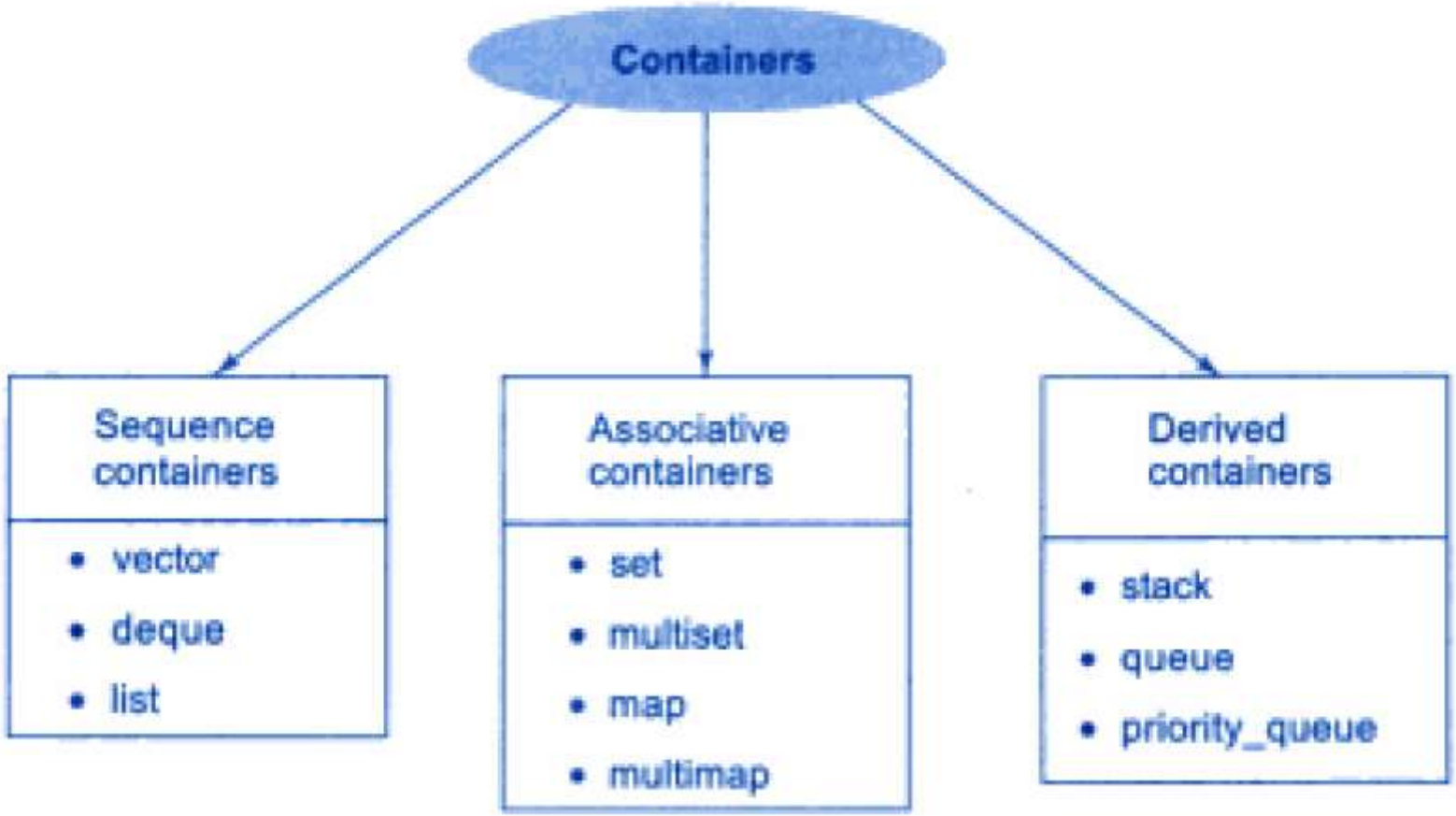
Associative Containers: Map, Multimap

- STL components are now part of standard c++ library defined in namespace std

- The standard template library (STL) contains
  - Containers
  - Algorithms
  - Iterators

Containers supported by STL

Containers are objects that hold data



Associative containers

- They are designed to support direct access to elements using keys
- Not sequential
- There are four types
  - Set
  - Multiset
  - Map
  - Multimap
- Store data in a structure called tree which facilitates fast searching, deletion and insertion
- Slow for random access and inefficient for sorting

Associative Container

- Associative containers implement sorted data structures that can be quickly searched (O(log n) complexity).
- Set collection of unique keys, sorted by keys
- map collection of key-value pairs, sorted by keys, keys are unique

- Multiset collection of keys, sorted by keys
- Multimap collection of key-value pairs, sorted by keys

Associative containers

- Set and Multiset
  - Store number of items and provide operations for manipulating them using the values as keys
  - Difference between set and multiset
    - Multiset allows duplicates , but set does not allow
  - Map and multimap
    - Used to store pairs of items – one called the key and the other called the value
  - Difference between map and multimap
    - Map allows only one key for a given value while multimap permits multiple keys

Member Functions &Element Access

- Here are following points to be noted related to various functions we used in the above example −
- The push_back( ) member function inserts value at the end of the vector, expanding its size as needed.
- The size( ) function displays the size of the vector.
- The function begin( ) returns an iterator to the start of the vector.
- The function end( ) returns an iterator to the end of the vector.
- Accessing elements
- at(g) – Returns a reference to the element at position ‘g’ in the vector
- front() – Returns a reference to the first element in the vector
- back() – Returns a reference to the last element in the vector

Other functions

- **empty :** This method returns true if the list is empty else returns false.
- **size :** This method can be used to find the number of elements present in the list.
- **front and back :** front() is used to get the first element of the list from the start while back() is used to get the first element of the list from the back.
- **swap:** Swaps two list, if there is exception thrown while swapping any element, swap() throws exception. Both lists which are to be swapped must be of the same type, i.e you can't swap list of an integer with list of strings.
- **reverse:** This method can be used to reverse a list completely.
- **sort:** sort method sorts the given list. It does not create new sorted list but changes the position of elements within an existing list to sort it.

Algorithms

- Retrieve or Non-mutating Algorithms
- Mutating Algorithms
- Sorting Algorithms
- Set Algorithms
- Relational Algorithms

Non Mutating Algorithms

- Adjacent_find –adj pairs
- Count-occurrence of a value
- Count_if—no.of elements that matches a predicate
- Equal-if two ranges are equal
- Find-first occurrence of a value
- Find_end
- Find_first_of()
- Find_if()- find the elements that matches a predicate
- For_each()- apply an operation to each element
- Mismatch()
- Search_ch()
- Search_n()

Mutating Algorithms

- Copy()
- Copy_backward()

Algorithms : find()

- InputIterator find (InputIterator first, InputIterator last, const T& val);
- The find() algorithm looks for an element matching *val* between *start* and *end*.
- If an element matching *val* is found, the return value Is an iterator that points to that element. Otherwise, the return value is an iterator that points to *end*.

```
#include <iostream>
 #include <algorithm>
using namespace std;
int main () {
int myints[] = { 10, 20, 30, 40 };
int * p = find (myints, myints+4, 30);
 if (p != myints+4) cout << "Element found in myints: " << *p << '\n';
else   cout << "Element not found in myints\n“;
 return 0; }
```

Find() Algorithm

```
#include <vector>
#include <algorithm>
#include <iostream>
int key;
int arr[] = { 12, 3, 17, 8, 34, 56, 9  };  // standard C array
vector<int> v(arr, arr+7);  // initialize vector with C array
vector<int>::iterator iter;
cout << ”enter value :”;
cin >> key;
iter=find(v.begin(),v.end(),key); // finds integer key in v
if (iter != v.end()) // found the element
   cout << ”Element ” << key << ” found” << endl;
else
  cout << ”Element ” << key << ” not in vector v” << endl;
```

Find_If() Algorithm

```
#include <vector>
#include <algorithm>
#include <iostream>
Bool mytest(int n) { return (n>21) && (n <36); };
int arr[] = { 12, 3, 17, 8, 34, 56, 9  };  // standard C array
vector<int> v(arr, arr+7);  // initialize vector with C array
vector<int>::iterator iter;
iter=find_if(v.begin(),v.end(),mytest);
  // finds element in v  for which mytest is true
if (iter != v.end()) // found the element
   cout << ”found ” << *iter << endl;
else
  cout << ”not found” << endl;
```

Algorithm: count()

- count() returns the number of elements in the given range that are equal to given value.
- Syntax for count is:
- **count(first ,last ,value) : This will return number of the element in range defined**
- by iterators first and last (excluded) which are equal (==) the value

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
int main ()
{
int values[] =
      {5,1,6,9,10,1,12,5,5,5,1,8,9,7,46};
int count_5 = count(values, values+15, 5);
/* now count_5 is equal to 4 */
vector<int> v(values, values+15);
int count_1 = count(v.begin(), v.end(), 1);
/* now count_1 is equal to */
return 0;
}
```

Count_If() Algorithm

```
#include <vector>
#include <algorithm>
#include <iostream>
Bool mytest(int n) { return (n>14) && (n <36); };
int arr[] = { 12, 3, 17, 8, 34, 56, 9  };  // standard C
   array
vector<int> v(arr, arr+7);  // initialize vector with C
   array
int n=count_if(v.begin(),v.end(),mytest);
  // counts element in v  for which mytest is true
cout << ”found ” << n << ” elements” << endl;
```

Algorithms : search

- This function is used to perform searches for a given sequence in a given range. There are two variations of the search():

- **search(first1 ,last1 ,first2 ,last2) :** This function searches for the sequence defined by first2 and last2 in the range first1 and last1(where last1 is excluded). If there is a match an iterator to the first element of the sequence in the range [first1,last1] is returned, else iterator to last1 is returned.
- **search(first1 ,last1 ,first2 ,last2 ,cmp_functions) :** Here cmp_function is used to decide how to check the equality of two elements, it is useful for non-numeric elements like strings and objects.

Algorithms : Search Example 1

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

int main()
{
int inputs1[] = { 1,2,3,4,5,6,7,8};
int inputs2[] = { 2,3,4};
vector<int> v1(inputs1, inputs1+9);
vector<int> v2(inputs2, inputs2+3);

vector<int>::iterator i ,j;

i = search(v1.begin(), v1.end(), v2.begin(), v2.end());

/* now i points to the second element in v1 */

j = search(v1.begin()+2, v1.end(), v2.begin(), v2.end());

/* j now points to the end of v1 as no sequence is equal to 2,3,4 in
[v1.begin()+2 ,v1.end()] */
}
```

Algorithms : sort()

- This function of the STL, sorts the contents of the given range. There are two version of sort() :
- **sort(start_iterator, end_iterator) :** sorts the range defined by iterators start_iterator and end_iterator in ascending order.
- **sort(start_iterator, end_iterator, compare_function) :** this also sorts the given range but you can define how the sorting should be done by compare_function.

Algorithms : sort() Example 1

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

bool compare_function(int i, int j)
{
return i > j;    // return 1 if i>j else 0
}
bool compare_string(string i, string j)
{
return (i.size() < j.size());
}

int main()
{
int arr[5] = {1,5,8,4,2};
sort(arr , arr+5);
// sorts arr[0] to arr[4] in ascending order
/* now the arr is 1,2,4,5,8  */

vector<int> v1;

v1.push_back(8);
v1.push_back(4);
v1.push_back(5);
v1.push_back(1);

/* now the vector v1 is 8,4,5,1 */
   vector<int>::iterator i, j;
i = v1.begin();   // i now points to beginning of the vector v1
   j = v1.end();      // j now points to end of the vector v1
sort(i,j);      //sort(v1.begin() , v1.end() ) can also be used
   /* now the vector v1 is 1,4,5,8 */
/* use of compare_function */
   int a2[] = { 4,3,6,5,6,8,4,3,6 };
sort(a2,a2+9,compare_function); // sorts a2 in descending order
   /* here we have used compare_function which uses      operator(>),
that result into sorting in descending order */

/* compare_function is also used to sort
non-numeric elements such as*/

string s[]={"a" , "abc", "ab" , "abcde"};

sort(s,s+4,compare_string);
   /* now s is "a","ab","abc","abcde" */
}
```

Algorithm: merge()

Combines the elements in the sorted ranges [first1,last1) and [first2,last2), into a new range beginning at *result* with all its elements sorted.
Syntax: OutputIterator merge (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result);

```
#include <iostream>
 #include <algorithm>
#include <vector>
using namespace std;
int main () {
int first[] = {5,10,15,20,25};
 int second[] = {50,40,30,20,10};
vector<int> v(10);
sort (first,first+5);
sort (second,second+5);
merge (first,first+5,second,second+5,v.begin());
cout << "The resulting vector contains:";
for (std::vector<int>::iterator it=v.begin(); it!=v.end(); ++it)
cout << ' ' << *it; std::cout << '\n'; return 0; }
```

for_each

```
#include <iostream>
#include <algorithm>
using namespace std;
void in_to_cm(double); //declaration
int main()
{ //array of inches values
double inches[] = { 3.5, 6.2, 1.0, 12.75, 4.33
};
//output as centimeters
for_each(inches, inches+5, in_to_cm);
cout << endl;
return 0;
}
void in_to_cm(double in) //convert and
display as centimeters
{
cout << (in * 2.54) << ' ';
}
```

The output looks like this:
8.89 15.748 2.54 32.385 10.9982}

Syntax :
Function for_each (InputIterator first, InputIterator last, Function fn);

The for_each() algorithm allows you to do something to every item in a container. You write your own function to determine what that "something" is. Your function can't change the elements in the container, but it can use or display their values.

Transform()

```
#include <iostream>
#include <algorithm>
using namespace std;
int main()
{ //array of inches values
double inches[] = { 3.5, 6.2, 1.0, 12.75, 4.33 };
double centi[5];
double in_to_cm(double); //prototype
//transform into array centi[]
transform(inches, inches+5, centi, in_to_cm);
for(int j=0; j<5; j++) //display array centi[]
cout << centi[j] << ' ';
cout << endl;
return 0;
}
double in_to_cm(double in) //convert inches to
      centimeters
{
return (in * 2.54); //return result
}
```

The transform() algorithm does something to every item in a container, and places the resulting values in a different container (or the same one).
Again, a user-written function determines what will be done to each item. The return type of this function must be the same as that of the destination container.

Syntax:
OutputIterator transform (InputIterator first1, InputIterator last1, OutputIterator result, UnaryOperation op);

Maps

- Associative container that associates objects of type *Key* with objects of type *Data*
  - Sorted according to keys
- Map
  - Stores (key, object) pairs
  - Unimodal: duplicate keys not allowed
  - AKA: table, associative array

The STL Map Template

- map()
- map(const key_compare& comp)

- pair<iterator, bool> insert(const value_type& x)
  - Inserts x into the map
- iterator insert(iterator pos, const value_type& x)
  - Inserts x into the map, using pos as a hint to where it will be inserted
- void insert(iterator, iterator)
  - Inserts a range into the map

STL Map Template

- void erase(iterator pos)
  - Erases the element pointed to by pos
- size_type erase(const key_type& k)
  - Erases the element whose key is k
- void erase(iterator first, iterator last)
  - Erases all elements in a range

- iterator find(const key_type& k)
  - Finds an element whose key is k.
- data_type& operator[](const key_type& k)
  - Returns a reference to the object that is associated with a particular key.
  - If the map does not already contain such an object, operator[] inserts the default object data_type()

Map Usage Example

```
#include <iostream>
#include<iterator>
#include <map>
#include <algorithm>
#include<cstring>

using namespace std;

map<const char*, int> months;
map<const char*, int>::iterator cur;

int main() {
   months["january"] = 31;
   months["february"] = 28;
   months["march"] = 31;
   months["april"] = 30;
   months["may"] = 31;
   months["june"] = 30;
   months["july"] = 31;
   months["august"] = 31;
   months["september"] = 30;
   months["october"] = 31;
```

Example Maps

```
#include <iostream>
#include <map>

using namespace std;

int main ()
{
  map<int,string> m;
m.insert(pair<int,string>(5,"ABCD"));
   m.insert(pair<int,string>(6,"EFGH"));
    m.insert(pair<int,string>(7,"IJKL"));
  cout << m.at(5)<<endl ;
  cout << m.at(6) <<endl;
// prints value associated with key
5,6


  /* note that the parameters in the above at() are the keys not the index */

  cout << m[7]<<endl ; // prints value associated with key 7
```

Multi Set Example

```
#include<iostream>
#include <set>
using namespace std;
int main()
{
   // multiset declare
   multiset<int> s;
    // Elements added to set
   s.insert(12);
   s.insert(10);
   s.insert(2);
   s.insert(10); // duplicate added
   s.insert(90);
   s.insert(85);
   s.insert(75);
   s.insert(90);
   s.insert(95);
   s.insert(45);
   s.insert(80);
   s.insert(45);
    // Iterator declared to traverse
   // set elements
```

Function Objects

- Functors (Function Objects or Functionals) are simply put **object + ()**.
- In other words, a **functor** is any object that can be used with **()** in the manner of a function.
- This includes normal functions, pointers to functions, and class objects for which the **() operator** (function call operator) is overloaded, i.e., classes for which the function **operator()()** is defined.
- Sometimes we can use a function object when an ordinary function won't work. The STL often uses function objects and provides several function objects that are very helpful.
- Function objects are another example of the power of generic programming and the concept of pure abstraction. We could say that anything that behaves like a function is a function. So, if we define an object that behaves as a function, it can be used as a function.

Function Objects

```
#include <iostream>
#include<vector>>
#include <algorithm>
using namespace std;
class InCm {
public:
    void operator()(double in)
 {
          cout << (in * 2.54) << " ";
    }
};
int main()
{
vector<double> inches;
inches.push_back(3.5);
inches.push_back(7);


InCm in_to_cm;
```

Advantages of function object

- Function object are "smart functions."
- Each function object has its own type.
- Function objects are usually faster than ordinary functions.

MCQ

- Which container can have the same keys?
  a) map
  b) multimap
  c) unordered map
  d) set

Answer: b

- Which container is used to keep priority based elements?
  a) queue
  b) stack
  c) set
  d) priority queue

Answer: d

MCQ

- How many components STL has?

a) 1
b) 2
c) 3
d) 4

Answer: d

Explanation: STL has four components namely Algorithms, Containers, Functors and Iterators.

- Which of the following is correct about map and multimap?
  a) Map can have same keys whereas multimap cannot
  b) Implementation of maps and multimap are different
  c) Multimap can have same keys whereas the map cannot
  d) Average search time of map is greater than multimap

Answer: c

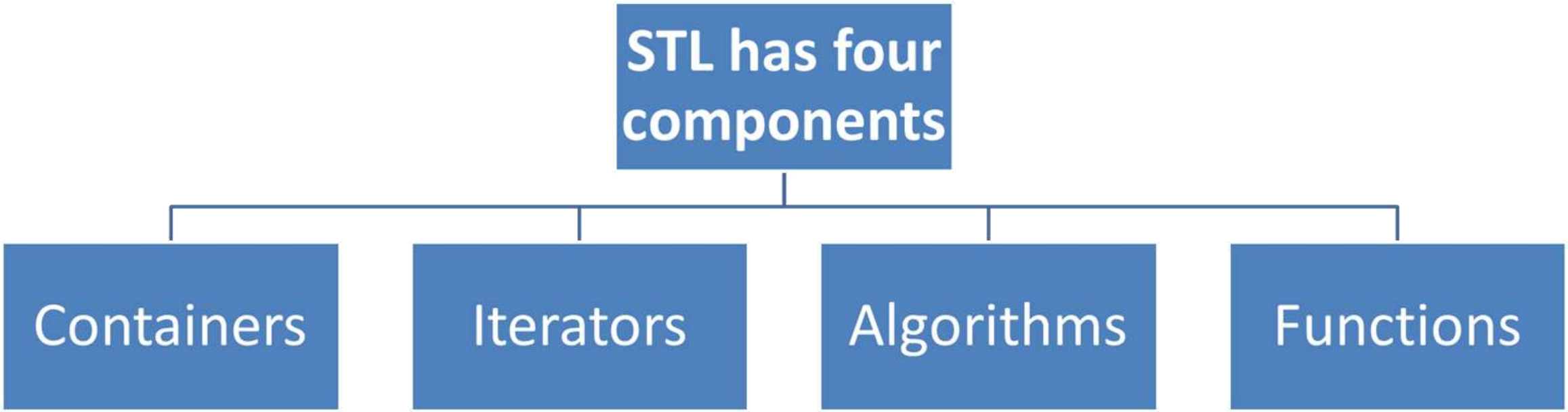
MCQ

- Which header is need to be used with function objects?
  a) <function>
  b) <functional>
  c) <funct>
  d) <functionstream>

Answer: b

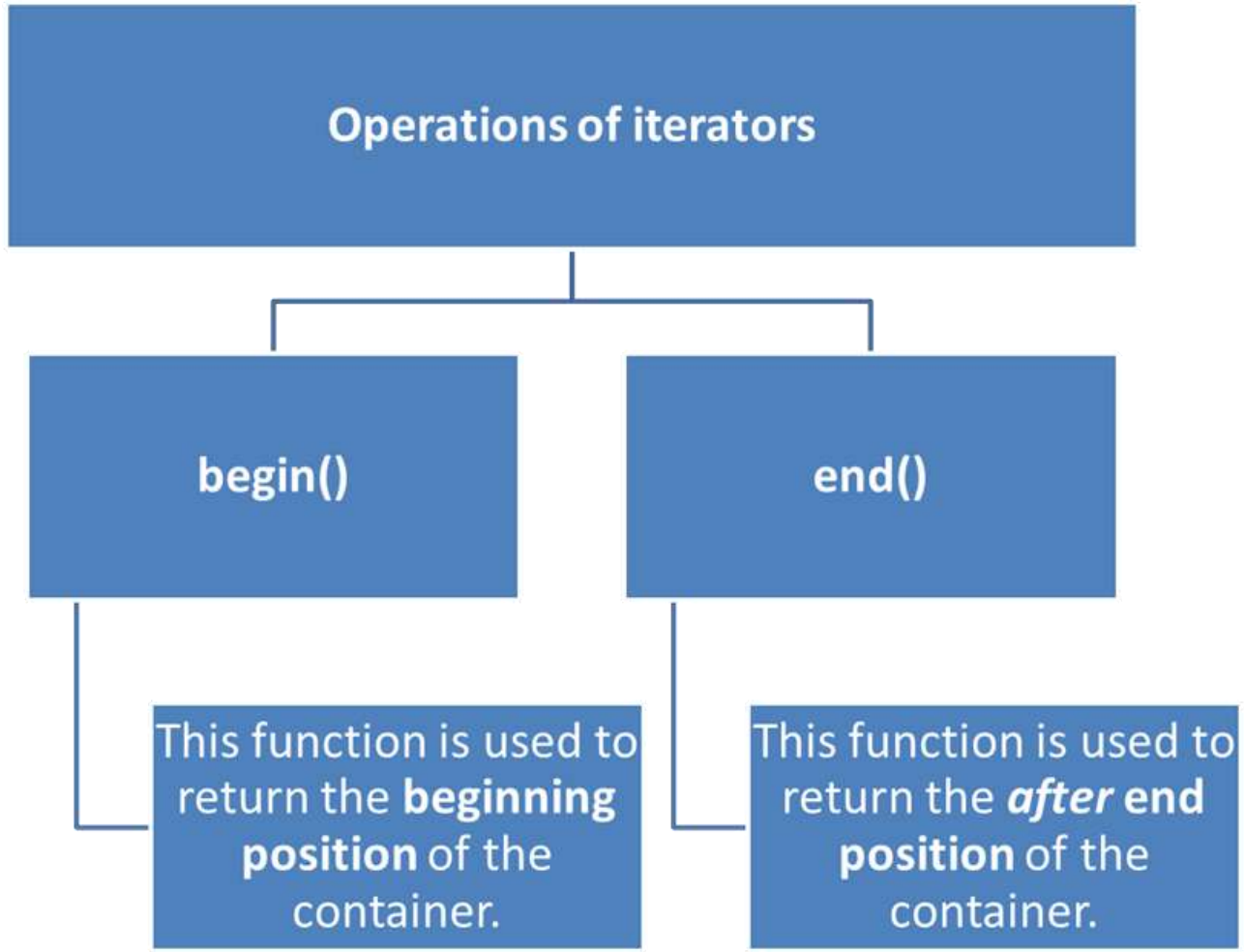
Explanation: <functional> header is need to be used with function objects.

Topic :STL Iterators


STL has four components
Containers
Iterators
Algorithms
Functions

STL Iterators

- Are used to point at the memory addresses of STL containers.
- They are primarily used in sequence of numbers, characters etc.
- They reduce the complexity and execution time of program.


Operations of iterators
begin()
end()
This function is used to return the **beginning position** of the container.
This function is used to return the ***after*** **end position** of the container.

```
// C++ code to demonstrate the working of  iterator, begin() and end()

#include<iostream>
#include<iterator> // for iterators
#include<vector> // for vectors
using namespace std;

int main()
{
        vector<int> ar = { 1, 2, 3, 4, 5 };

        // Declaring iterator to a vector
        vector<int>::iterator ptr;

        // Displaying vector elements using begin() and end()
        cout << "The vector elements are : ";

        for (ptr = ar.begin(); ptr < ar.end(); ptr++)
                cout << *ptr << " ";

        return 0;
}
```

Output:

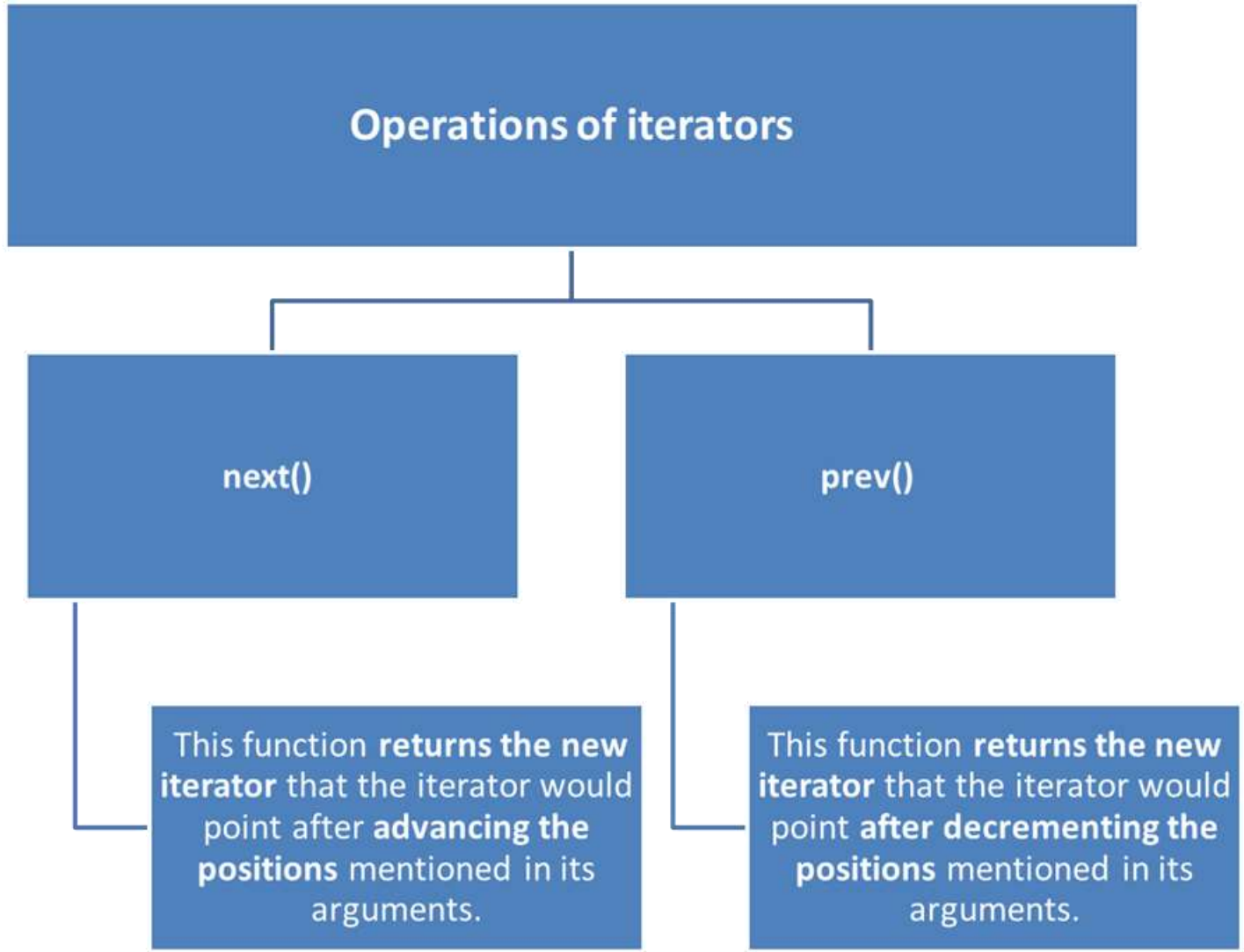
```
The vector elements are : 1 2 3 4 5
```

advance() :- This function is used to **increment the iterator position** till the specified number mentioned in its arguments.

```
// C++ code to demonstrate the working of  advance()
#include<iostream>
#include<iterator> // for iterators
#include<vector> // for vectors
using namespace std;
int main()
{
        vector<int> ar = { 1, 2, 3, 4, 5 };

        // Declaring iterator to a vector
        vector<int>::iterator ptr = ar.begin();

        // Using advance() to increment iterator position
        // points to 4
        advance(ptr, 3);

        // Displaying iterator position
        cout << "The position of iterator after advancing is : ";
        cout << *ptr << " ";

        return 0;

}
```

Output:

```
The position of iterator after advancing is : 4
```


Operations of iterators
next()
prev()
This function returns the new iterator that the iterator would point after advancing the positions mentioned in its arguments.
This function returns the new iterator that the iterator would point after decrementing the positions mentioned in its arguments.

```cpp
// C++ code to demonstrate the working of   next() and prev()
#include<iostream>
#include<iterator> // for iterators
#include<vector> // for vectors
using namespace std;
int main()
{
        vector<int> ar = { 1, 2, 3, 4, 5 };

        // Declaring iterators to a vector
        vector<int>::iterator ptr = ar.begin();
        vector<int>::iterator ftr = ar.end();


        // Using next() to return new iterator
        // points to 4
        auto it = next(ptr, 3);

        // Using prev() to return new iterator
        // points to 3
        auto it1 = prev(ftr, 3);

        // Displaying iterator position
        cout << "The position of new iterator using next() is : ";
        cout << *it << " ";
        cout << endl;

        // Displaying iterator position
        cout << "The position of new iterator using prev() is : ";
        cout << *it1 << " ";
        cout << endl;

        return 0;
}
```

Output:

```
The position of new iterator using next() is : 4
The position of new iterator using prev() is : 3
```

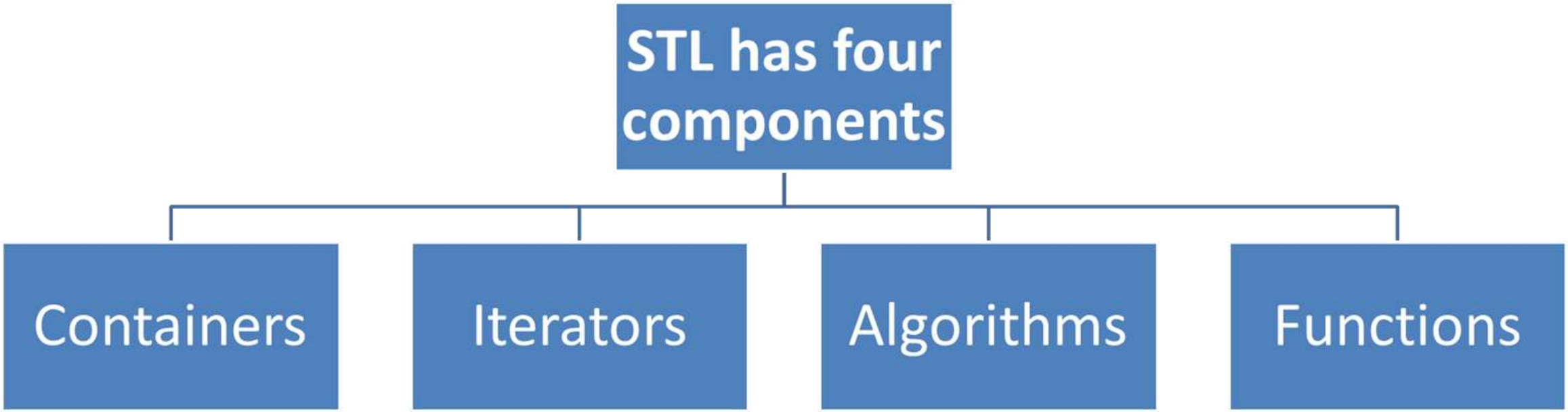
inserter() :- This function is used to **insert the elements at any position** in the container. It accepts **2 arguments, the container and iterator to position where the elements have to be inserted**.

```
// C++ code to demonstrate the working of  inserter()
#include<iostream>
#include<iterator> // for iterators
#include<vector> // for vectors
using namespace std;
int main()
{
        vector<int> ar = { 1, 2, 3, 4, 5 };
        vector<int> ar1 = {10, 20, 30};

        // Declaring iterator to a vector
        vector<int>::iterator ptr = ar.begin();

        // Using advance to set position
        advance(ptr, 3);

        // copying 1 vector elements in other using inserter()
        // inserts ar1 after 3rd position in ar
        copy(ar1.begin(), ar1.end(), inserter(ar,ptr));

        // Displaying new vector elements
        cout << "The new vector after inserting elements is : ";
        for (int &x : ar)
                cout << x << " ";

        return 0;
}
```

Output:

```
The new vector after inserting elements is : 1 2 3 10 20 30 4 5
```

Topic : STL Algorithm
Function Objects


STL has four components
Containers
Iterators
Algorithms
Functions

STL Algorithms

- Are used to point at the memory addresses of STL containers.
- They are primarily used in sequence of numbers, characters etc.
- They reduce the complexity and execution time of program.

STL ALgorithms

- STL has an ocean of algorithms, for all < algorithm > library functions
- Some of the most used algorithms on vectors and most useful one's in Competitive Programming are mentioned as follows :
  - **sort(first_iterator, last_iterator)** – To sort the given vector.
  - **reverse(first_iterator, last_iterator)** – To reverse a vector.
  - ***max_element (first_iterator, last_iterator)** – To find the maximum element of a vector.
  - ***min_element (first_iterator, last_iterator)** – To find the minimum element of a vector.
  - **accumulate(first_iterator, last_iterator, initial value of sum)** – Does the summation of vector elements

```
// A C++ program to demonstrate working of sort(), reverse()
#include <algorithm>
#include <iostream>
#include <vector>
#include <numeric> //For accumulate operation
using namespace std;

int main()
{
        // Initializing vector with array values
        int arr[] = {10, 20, 5, 23 ,42 , 15};
        int n = sizeof(arr)/sizeof(arr[0]);
        vector<int> vect(arr, arr+n);

        cout << "Vector is: ";
        for (int i=0; i<n; i++)
                cout << vect[i] << " ";

        // Sorting the Vector in Ascending order
        sort(vect.begin(), vect.end());

        cout << "\nVector after sorting is: ";
        for (int i=0; i<n; i++)
        cout << vect[i] << " ";

        // Reversing the Vector
        reverse(vect.begin(), vect.end());

        cout << "\nVector after reversing is: ";
        for (int i=0; i<6; i++)
                cout << vect[i] << " ";

        cout << "\nMaximum element of vector is: ";
        cout << *max_element(vect.begin(), vect.end());

        cout << "\nMinimum element of vector is: ";
        cout << *min_element(vect.begin(), vect.end());

        // Starting the summation from 0
        cout << "\nThe summation of vector elements is:
";
        cout << accumulate(vect.begin(), vect.end(), 0);

        return 0;
}
```

Output:
Vector before sorting is: 10 20 5 23 42 15 Vector after sorting is: 5 10 15 20 23 42 Vector before reversing is: 5 10 15 20 23 42 Vector after reversing is: 42 23 20 15 10 5 Maximum element of vector is: 42
Minimum element of vector is: 5
The summation of vector elements is: 115

count(first_iterator, last_iterator,x) – To count the occurrences of x in vector.
find(first_iterator, last_iterator, x) – Points to last address of vector ((name_of_vector).end()) if element is not present in vector.

```
// C++ program to demonstrate working of count()  and find()
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main()
{
        // Initializing vector with array values
        int arr[] = {10, 20, 5, 23 ,42, 20, 15};
        int n = sizeof(arr)/sizeof(arr[0]);
        vector<int> vect(arr, arr+n);

        cout << "Occurrences of 20 in vector : ";

        // Counts the occurrences of 20 from 1st to
        // last element
        cout << count(vect.begin(), vect.end(), 20);

        // find() returns iterator to last address if
        // element not present
        find(vect.begin(), vect.end(),5) != vect.end()?

<< "\nElement found":

                                                cout

                                        cout <<

"\nElement not found";

        return 0;
}
```

Output:

```
Occurrences of 20 in vector: 2
Element found
```

merge() in C++ STL

- C++ offers in its STL library a merge() which is quite useful to **merge sort two containers** into a **single** container.
  It is defined in header “**algorithm**“. It is implemented in two ways.
- **Syntax 1 : Using operator “<"**

```
// C++ code to demonstrate the working of  merge() implementation 1

#include <bits/stdc++.h>
using namespace std;

int main()
{
        // initializing 1st container
        vector<int> arr1 = { 1, 4, 6, 3, 2 };

        // initializing 2nd container
        vector<int> arr2 = { 6, 2, 5, 7, 1 };

        // declaring resultant container
        vector<int> arr3(10);

        // sorting initial containers
        sort(arr1.begin(), arr1.end());
        sort(arr2.begin(), arr2.end());

        // using merge() to merge the initial containers
        merge(arr1.begin(), arr1.end(), arr2.begin(), arr2.end(),
arr3.begin());

        // printing the resultant merged container
        cout << "The container after merging initial containers is : ";

        for (int i = 0; i < arr3.size(); i++)
                cout << arr3[i] << " ";
        return 0;
}
```

Output:

```
The container after merging initial containers
is : 1 1 2 2 3 4 5 6 6 7
```

search() in c++ STL

- **std::search** is defined in the header file <algorithm> and used to find out the presence of a subsequence satisfying a condition (equality if no such predicate is defined) with respect to another sequence.
- It searches the sequence [first1, last1) for the first occurrence of the subsequence defined by [first2, last2), and returns an iterator to its first element of the occurrence, or last1 if no occurrences are found.
- It compares the elements in both ranges sequentially using operator== (version 1) or based on any given predicate (version 2). A subsequence of [first1, last1) is considered a match only when this is true for all the elements of [first2, last2). Finally, std::search returns the first of such occurrences.
- It can be used in either of the two versions, as depicted below :
  1. **For comparing elements using ==**
  2. **For comparison based on a predicate (or condition)**

1. **For comparing elements using ==**

ForwardIterator1 search (ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2);

- first1: Forward iterator to beginning of first container to be searched into.

last1: Forward iterator to end of first container to be searched into.
first2: Forward iterator to the beginning of the subsequence of second container to be searched for.
last2: Forward iterator to the ending of the subsequence of second container to be searched for.

- Returns: an iterator to the first element of the first occurrence of [first2, last2) in [first1, last1), or last1
if no occurrences are found.

```
// C++ program to demonstrate the use of std::search
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main()
{
        int i, j;
        // Declaring the sequence to be searched into
        vector<int> v1 = { 1, 2, 3, 4, 5, 6, 7 };
        // Declaring the subsequence to be searched for
        vector<int> v2 = { 3, 4, 5 };
        // Declaring an iterator for storing the returning pointer
        vector<int>::iterator i1;
        // Using std::search and storing the result in
        // iterator i1
        i1 = std::search(v1.begin(), v1.end(), v2.begin(), v2.end());
        // checking if iterator i1 contains end pointer of v1 or not
        if (i1 != v1.end()) {
                cout << "vector2 is present at index " << (i1 - v1.begin());
        } else {
                cout << "vector2 is not present in vector1";
        }
        return 0;
}
```

Output:

vector2 is present at index 2

For comparison based on a predicate (or condition) :

- ForwardIterator1 search (ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate pred);
- All the arguments are same as previous template, just one more argument is added
- pred: Binary function that accepts two elements as arguments (one of each of the two containers, in the same order), and returns a value convertible to bool. The returned value indicates whether the elements are considered to match in the context of this function. The function shall not modify any of its arguments. This can either be a function pointer or a function object.
- Returns: an iterator, to the first element of the first occurrence of [first2, last2) satisfying a predicate, in [first1, last1), or last1 if no occurrences are found.

```
// C++ program to demonstrate the use of std::search
// with binary predicate
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
// Defining the BinaryPredicate function
bool pred(int i, int j)
{
 if (i > j)
 {return 1;}
else
{return 0;}
}
int main()
{
        int i, j;
        // Declaring the sequence to be searched into
        vector<int> v1 = { 1, 2, 3, 4, 5, 6, 7 };
        // Declaring the subsequence to be compared to based
        // on predicate
        vector<int> v2 = { 3, 4, 5 };
        // Declaring an iterator for storing the returning pointer
        vector<int>::iterator i1;
        // Using std::search and storing the result in
        // iterator i1 based on predicate pred
        i1 = std::search(v1.begin(), v1.end(), v2.begin(), v2.end(), pred);
        // checking if iterator i1 contains end pointer of v1 or not
        if (i1 != v1.end()) {
                cout << "vector1 elements are greater than vector2 starting "<< "from position " << (i1 - v1.begin());
        } else {
                cout << "vector1 elements are not greater than vector2 "<< "elements consecutively.";
        }
        return 0;
}
```

Output:
vector1 elements are greater than vector2 starting from position 3

for_each() in STL

- **Apply function to range**
- Applies function *fn* to each of the elements in the range `[first,last)`.
- The behavior of this template function is equivalent to:

```
template<class InputIterator, class Function>
  Function for_each(InputIterator first, InputIterator last, Function fn)
{
  while (first!=last) {
    fn (*first);
    ++first;
  }
  return fn;      // or, since C++11: return move(fn);
}
```

- **Parameters**

first, last

✓ Input iterators to the initial and final positions in a sequence. The range used is [first,last), which contains all the elements between first and last, including the element pointed by first but not the element pointed by last.

Fn

✓ Unary function that accepts an element in the range as argument.
This can either be a function pointer or a move constructible function object.
Its return value, if any, is ignored.

```
// for_each example
#include <iostream>     // std::cout
#include <algorithm>    // std::for_each
#include <vector>       // std::vector
void myfunction (int i) {  // function:
  std::cout << ' ' << i;
}
struct myclass {          // function object type:
  void operator() (int i) {std::cout << ' ' << i;}
} myobject;
int main ()
 {
  std::vector<int> myvector;
  myvector.push_back(10);
  myvector.push_back(20);
  myvector.push_back(30);
  std::cout << "myvector contains:";

for_each (myvector.begin(), myvector.end(), myfunction);
  std::cout << '\n';
  // or:
  std::cout << "myvector contains:";
  for_each (myvector.begin(), myvector.end(), myobject);
  std::cout << '\n';
  return 0;
}
```

Output:

```
myvector contains: 10 20 30
myvector contains: 10 20 30
```

Functors in C++

Function objects

- Consider a function that takes only one argument.
- However, while calling this function we have a lot more information that we would like to pass to this function, but we cannot as it accepts only one parameter. What can be done?
- One obvious answer might be global variables.
- However, good coding practices do not advocate the use of global variables and say they must be used only when there is no other alternative.
- **Functors** are objects that can be treated as though they are a function or function pointer.
- Functors are most commonly used along with STLs.

The functor allows an instance object of some class to be called as if it were an ordinary function.

- Let us consider a function that takes one argument. We can use this function as function object to do some task on a set of data

```
#include <iostream>
#include <algorithm>
using namespace std;
int square(int x)
{

  return x*x; //return square of x
}
int main()
{
  int data[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

transform(data, data+10, data, square);// array name, elements,
name, square and store

  for (int i = 0; i<10; i++)
    cout << data[i] << endl;
}
```

Output

```
0
1
4
9
16
25
36
49
64
81
```

```
// A C++ program uses transform() in STL to add  1 to all elements of arr[]
#include <bits/stdc++.h>
using namespace std;

int increment(int x) { return (x+1); }

int main()
{
        int arr[] = {1, 2, 3, 4, 5};
        int n = sizeof(arr)/sizeof(arr[0]);

        // Apply increment to all elements of
        // arr[] and store the modified elements
        // back in arr[]
        transform(arr, arr+n, arr, increment);

        for (int i=0; i<n; i++)
                cout << arr[i] << S" ";

        return 0;
}
```

Output:

```
2  3  4  5  6
```

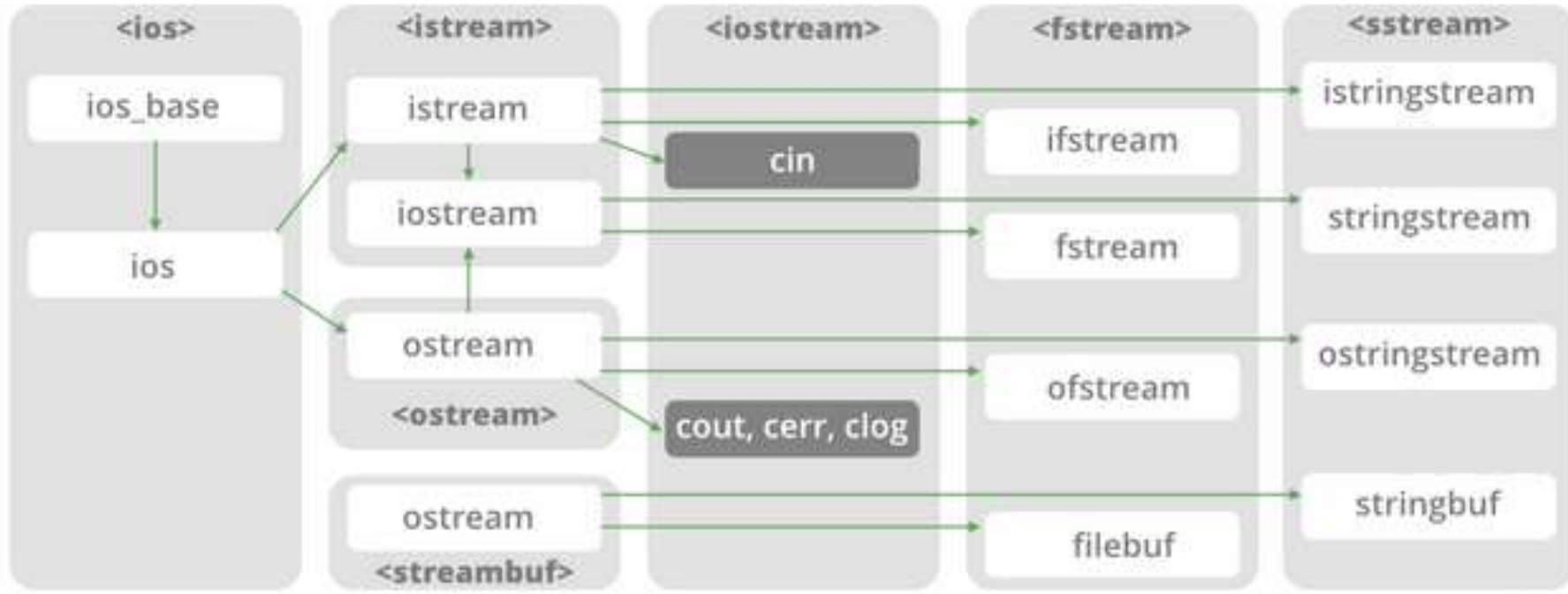
Topic : Streams and Files

Learning Objectives

- C++ I/O streams.
- Reading and writing sequential files.
- Reading and writing random access files.

C++ Files and Streams

- C++ views each files as a sequence of bytes.
- Each file ends with an ***end-of-file*** marker.
- When a file is ***opened***, an object is created and a stream is associated with the object.
- To perform file processing in C++, the header files **<iostream.h>** and **<fstream.h>** must be included.
- <fstream.> includes <ifstream> and <ofstream>


<ios>
ios_base
ios
<istream>
istream
iostream
ostream
<ostream>
ostream
<streambuf>
<iostream>
cin
cout, cerr, clog
<fstream>
ifstream
fstream
ofstream
filebuf
<sstream>
istringstream
stringstream
ostringstream
stringbuf

MEMBER CONSTANT	STANDS FOR	ACCESS
in *	input	File open for reading: the internal stream buffer supports input operations.
out	output	File open for writing: the internal stream buffer supports output operations.
binary	binary	Operations are performed in binary mode rather than text.
ate	at end	The output position starts at the end of the file.
app	append	All output operations happen at the end of the file, appending to its existing contents.
trunc	truncate	Any contents that existed in the file before it is open are discarded.

Default Open Modes :	
ifstream	ios::in
ofstream	ios::out
fstream	ios::in | ios::out

Creating a sequential file

```
// Fig. 14.4: fig14_04.cpp  D&D p.708
// Create a sequential file
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
int main()
{
  // ofstream constructor opens file
  ofstream outClientFile( "clients.dat", ios::out );

  if ( !outClientFile ) {  // overloaded ! operator
    cerr << "File could not be opened" << endl;
    exit( 1 );    // prototype in stdlib.h
  }
```

Sequential file

```
cout << "Enter the account, name, and balance.\n"
      << "Enter end-of-file to end input.\n? ";
   int account;
  char name[ 30 ];
  float balance;

  while ( cin >> account >> name >> balance ) {
    outClientFile << account << ' ' << name
                  << ' ' << balance << '\n';
    cout << "? ";
  }

  return 0;  // ofstream destructor closes file
}
```

How to open a file in C++ ?

Ofstream outClientFile(“clients.dat”, ios:out)

OR

Ofstream outClientFile;

outClientFile.open(“clients.dat”, ios:out)

File Open Modes

ios:: app - (append) write all output to the end of file
ios:: ate - data can be written anywhere in the file
ios:: binary - read/write data in binary format
ios:: in - (input) open a file for input
ios::out - (output) open afile for output
ios: trunc -(truncate) discard the files' contents if
it exists

File Open Modes contd…..

ios:nocreate - if the file does **NOT** exists, the open operation fails

ios:noreplace - if the file exists, the open operation fails

How to close a file in C++?

The file is closed implicitly when a destructor for the corresponding object is called

OR

by using member function *close:*

outClientFile.close();

Reading and printing a sequential file

```
// Reading and printing a sequential file
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>
void outputLine( int, const char *, double );
int main()
{
  // ifstream constructor opens the file
  ifstream inClientFile( "clients.dat", ios::in );

  if ( !inClientFile ) {
    cerr << "File could not be opened\n";
    exit( 1 );
  }
```

```
int account;
  char name[ 30 ];
  double balance;

  cout << setiosflags( ios::left ) << setw( 10 ) << "Account"
       << setw( 13 ) << "Name" << "Balance\n";

  while ( inClientFile >> account >> name >> balance )
    outputLine( account, name, balance );

  return 0;  // ifstream destructor closes the file
}

void outputLine( int acct, const char *name, double bal )
{
  cout << setiosflags( ios::left ) << setw( 10 ) << acct
       << setw( 13 ) << name << setw( 7 ) << setprecision( 2 )
       << resetiosflags( ios::left )
       << setiosflags( ios::fixed | ios::showpoint )
       << bal << '\n';
}
```

File position pointer

<istream> and <ostream> classes provide member functions for repositioning the *file pointer* (the byte number of the next byte in the file to be read or to be written.)

These member functions are:

seekg (seek get) for istream class

seekp (seek put) for ostream class

Examples of moving a file pointer

inClientFile.seekg(0) - repositions the file get pointer to the beginning of the file

inClientFile.seekg(n, ios:beg) - repositions the file get pointer to the n-th byte of the file

inClientFile.seekg(m, ios:end) -repositions the file get pointer to the m-th byte from the end of file

nClientFile.seekg(0, ios:end) - repositions the file get pointer to the end of the file

The same operations can be performed with <ostream> function member *seekp.*

Updating a sequential file

Data that is formatted and written to a sequential file **cannot be modified easily** without the risk of destroying other data in the file.

If we want to modify a record of data, the new data may be longer than the old one and it could overwrite parts of the record following it.

Problems with sequential files

Sequential files are inappropriate for so-called “instant access” applications in which a particular record of information must be located immediately.

These applications include banking systems, point-of-sale systems, airline reservation systems, (or any data-base system.)

Random access files

Instant access is possible with random access files.

Individual records of a **random access file** can be accessed directly (and quickly) without searching many other records.

Example of a Program that Creates a Random Access File

```
// Fig. 14.11: clntdata.h
// Definition of struct clientData used in
// Figs. 14.11, 14.12, 14.14 and 14.15.
#ifndef CLNTDATA_H
#define CLNTDATA_H
struct clientData {
   int accountNumber;
   char lastName[ 15 ];
   char firstName[ 10 ];
   float balance;
};
#endif
```

Creating a random access file

```
// Creating a randomly accessed file sequentially
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include "clntdata.h"
int main()
{
        ofstream outCredit( "credit1.dat", ios::out);
if ( !outCredit ) {
   cerr << "File could not be opened." << endl;
   exit( 1 );
  }
```

```
clientData blankClient = { 0, "", "", 0.0 };

for ( int i = 0; i < 100; i++ )
outCredit.write
(reinterpret_cast<const char *>( &blankClient ),
sizeof( clientData ) );
return 0;
}
```

<ostream> membEr function *write*

The <ostream> member function *write* outputs a fixed number of bytes beginning at a specific location in memory to the specific stream. When the stream is associated with a file, the data is written beginning at the location in the file specified by the “put” file pointer.

Writing data randomly to a random file

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include "clntdata.h"
int main()
{
   ofstream outCredit( "credit.dat", ios::ate );

   if ( !outCredit ) {
      cerr << "File could not be opened." << endl;
      exit( 1 );
   }
```

```
cout << "Enter account number "<< "(1 to 100, 0 to end
input)\n? ";
clientData client;
   cin >> client.accountNumber;

   while ( client.accountNumber > 0 &&
client.accountNumber <= 100 )
 {
      cout << "Enter lastname, firstname, balance\n? ";
      cin >> client.lastName >> client.firstName>>
client.balance;
```

```
outCredit.seekp( ( client.accountNumber - 1 ) *sizeof(
clientData ) );
      outCredit.write(
         reinterpret_cast<const char *>( &client ),
         sizeof( clientData ) );

      cout << "Enter account number\n? ";
      cin >> client.accountNumber;
   }

   return 0;
}
```

Reading data from a random file

```
#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include <stdlib.h>
#include "clntdata.h"
void outputLine( ostream&, const clientData & );
int main()
{
   ifstream inCredit( "credit.dat", ios::in );
   if ( !inCredit ) {
      cerr << "File could not be opened." << endl;
      exit( 1 );
   }
```

```
cout << setiosflags( ios::left ) << setw( 10 ) << "Account"<< setw( 16 ) << "Last Name"
<< setw( 11 )
      << "First Name" << resetiosflags( ios::left )<< setw( 10 ) << "Balance" << endl;

   clientData client;

   inCredit.read( reinterpret_cast<char *>( &client ),
                  sizeof( clientData ) );
```

```
while ( inCredit && !inCredit.eof() ) {

    if ( client.accountNumber != 0 )
       outputLine( cout, client );

    inCredit.read( reinterpret_cast<char *>( &client ),
                   sizeof( clientData ) );
  }

  return 0;
}
```

```
void outputLine( ostream &output, const clientData &c )
{
   output << setiosflags( ios::left ) << setw( 10 )<< c.accountNumber <<
setw( 16 ) << c.lastName
        << setw( 11 ) << c.firstName << setw( 10 )<< setprecision( 2 ) <<
resetiosflags( ios::left )
        << setiosflags( ios::fixed | ios::showpoint )<< c.balance << '\n';
}
```

The <istream> function *read*

```
inCredit.read (reinterpret_cast<char *>(&client),sizeof(clientData));
```

The <istream> function inputs a specified (by sizeof(clientData)) number of bytes from the current position of the specified stream into an object.

Topic : Files

Learning Objectives

- Reading and writing sequential files.
- Reading and writing random access files.

files

- A file is a collection of related data stored in a particular area on the disk.
- A program typically involves either

i. data transfer between the console unit and the program (or)

Ii. Data transfer between the program and a disk file.

The stream that supplies data to the program is known as input stream.

The stream that receives data from the program is known as output stream.

Disk files

- read data (input stream)
  - data input
  - program
  - data output
- write data (output stream)
- Disk files

Classes for the file stream operations

A set of classes that define the file handling methods in c++ includes (defined in iostream.h and fstream.h) derived from fstreambase

ifstream input stream

ofstream output stream

fstream file stream

Opening and closing a file :

Before using a file, we have to,

- Give a suitable name for the file
- Think about data type and structure
- Purpose of the file
- Opening method:

A file can be opened in two ways,

i. Using the constructor function of the class. It is useful when we use only one file in the stream.

ii. Using the member function open() of the class, when we want to manage multiple files using one stream.

Opening files using constructor:

Step1: create the input stream using ifstream and output stream using ofstream.

Step 2: initialize the file object with the desired filename.

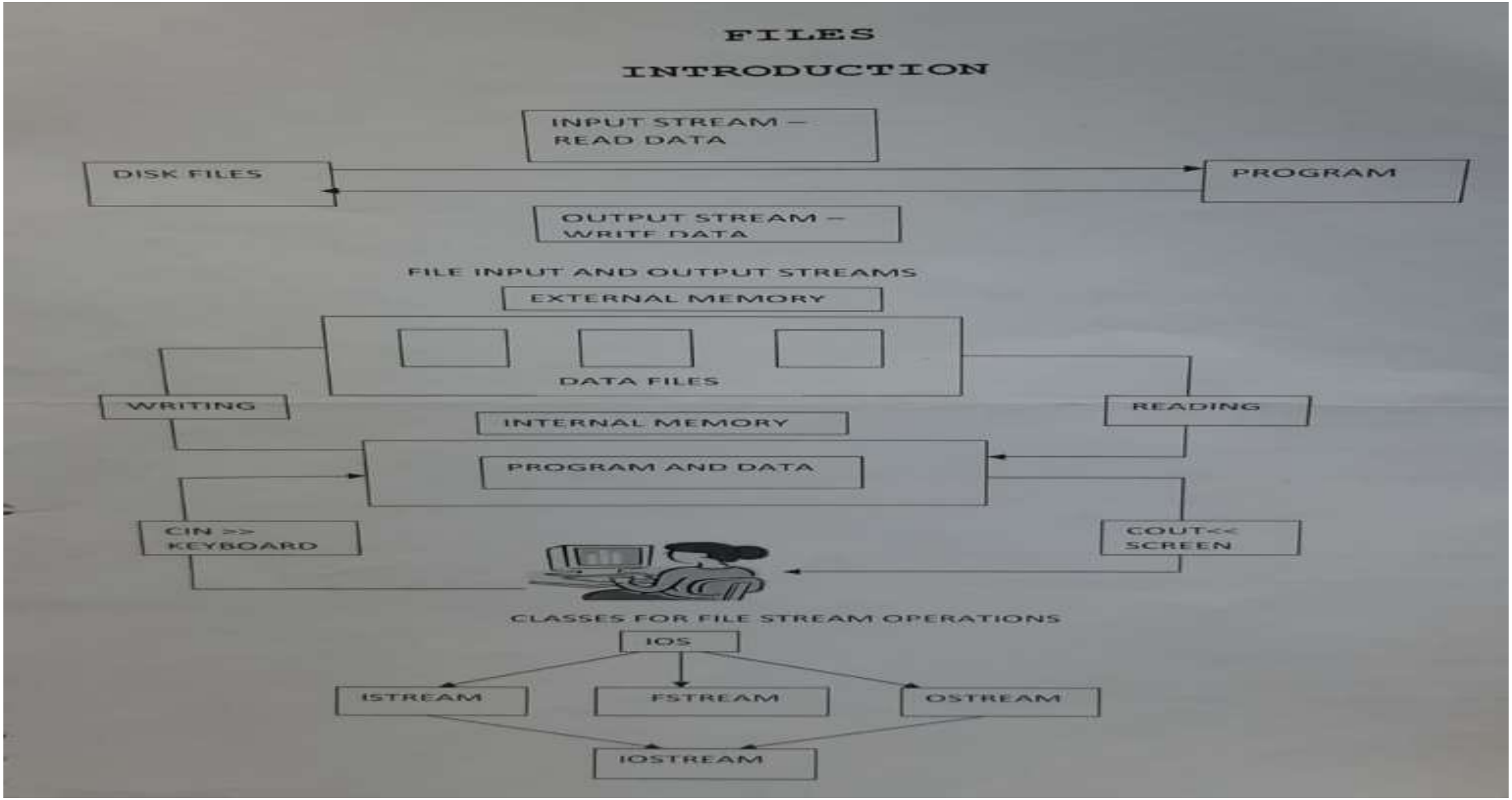
Eg: ofstream outfile(“result”); this stmt opens the file “results” for output and attaches at to the output stream outfile.

Eg : ifstream infile(“data”); this stmt declares infile as an ifstream object and attaches it to the file data for input.

Eg: outfile.close(); disconnects file “result” from outfile.

Eg: infile.close(); disconnects file “data” from infile.

Streams and Files



Working with single file- creating file with constructor functions

```
#include <fstream.h>
void main()
{
ofstream outf("ITEM");
cout <<"enter item name:";
char name[30];
cin>>name;
outf<<name<<"\n";
cout<<enter item cost";
float cost;
cin >>cost;
outf<<cost<<"\n";
outf.close();
ifstream inf("ITEM");
inf>>name;
inf>>cost;
cout<<"\n";
cout<<"iem name:"<<name<<endl;
cout<<"item cost:"<<cost<<endl;
inf.close();
}
```

Output:

Enter item name : disk

Enter item cost : 500

Item name : disk

Item cost : 500

Opening files using open()

g.f : file_stream_class stream object

fstream fin;

stream_object.open("file_name");

fin.open("DATA")

```
ofstream outfile;
outfile.open("DATA1");
.
outfile.close();
outfile.open("DATA2");
.
.
outfile.close();
```

if two files are opened in sequence for writing the data. The first file should be closed before opening the second file.

Stream_object.open(“filename”,mode);

The mode specifies the purpose of the file.

The file can be opened in the absence of the mode by using the default values in the absence of actual values.

File Open Modes

ios:: app - (append) write all output to the end of file

ios:: ate - data can be written anywhere in the file

ios:: binary - read/write data in binary format

ios:: in - (input) open a file for input

ios::out - (output) open a file for output

ios: trunc -(truncate) discard the files contents if it exists

ios::nocreate – open files if the file does not exists

ios :: noreplace – opens the file is the file already exists

Eg : fout.open(“DATA”, ios::app|ios :: nocreate) the mode can combine two or more parameters using the bitwise OR operator.

C++ Files and Streams

- C++ views each files as a sequence of bytes.
- Each file ends with an ***end-of-file*** marker.
- When a file is ***opened***, an object is created and a stream is associated with the object.
- To perform file processing in C++, the header files **<iostream.h>** and **<fstream.h>** must be included.
- <fstream> includes <ifstream> and <ofstream>

How to open and close a file in C++ ?

ofstream outClientFile("clients.dat", ios:out)

OR

ofstream outClientFile;

outClientFile.open("clients.dat", ios:out);

The file is closed implicitly when a destructor for the corresponding object is called

OR

by using member function *close;*

outClientFile.close();

```
#include<fstream.h>
void main()
{
ofstream fout;
fout.open("country");i
fout<<"united states of
america"<<endl;
fout <<"united
kingdom"<<endl;
fout<<"south korea"<<endl;
fout.close();
fout.open("capital");
fout<<"washington"<<endl;
fout<<"london"<<endl;
fout<<"seoul"<<endl;
fout.close();
//reading the file
const int n=80;
char line[n];
ifstream fin;
fin.open("country");
cout<<"contents of country
file"<<endl;
while(fin)
{
fin.getline(line,n);
cout<<line;
}
fin.close();
fin.open("capital");
cout<<endl<<contents of
capital file"<<"\n";
while(fin)
{
fin.getline(line,n);
cout<<line;
}
fin.close();
```

Output:

contents of the country file

united states of america

united kingdom

south korea

contents of capital file

Washington

London

seoul

Detecting end_of_file:

It prevents further attempt to read data from the file.

while(fin) □ when fin is a ifstream object which returns o if any error occurs in the file operation including the end of file condition.

Here the while loop is terminated when fin returns the value o on reaching the file end_of_file condition.

```
if(fin1.eof()!=0)
{
exit(1)
}
```

eof () is a member function of ios class . Here it returns a non-zero value if the end-of-file condition is encountererd and zero otherwise

Reading from two file simultaneously

```
#include <iostream.h>
#include<fstream.h>
#include<stdlib.h>
void main()
{
const int SIZE= 80;
fstream fin1,fin2;
fin1.open("country");
fin2.open("capital");
for(int i =1;i<=10;i++)
{
if(fin1.eof()!=0)
{
cout<<"exit from coutry"<<endl;
exit(1);
}
fin1.getline(line,SIZE);
cout<<"capital of "<<line;
if(fin2.eof()!=0)
{
cout<<"exit from capital"<<endl;
exit(1);
}
fin2.getline(line,size);
cout<<line<<endl;
}
}
```

output:

capital of united states of America

washington

capital of united kingdom

london

capital of south korea

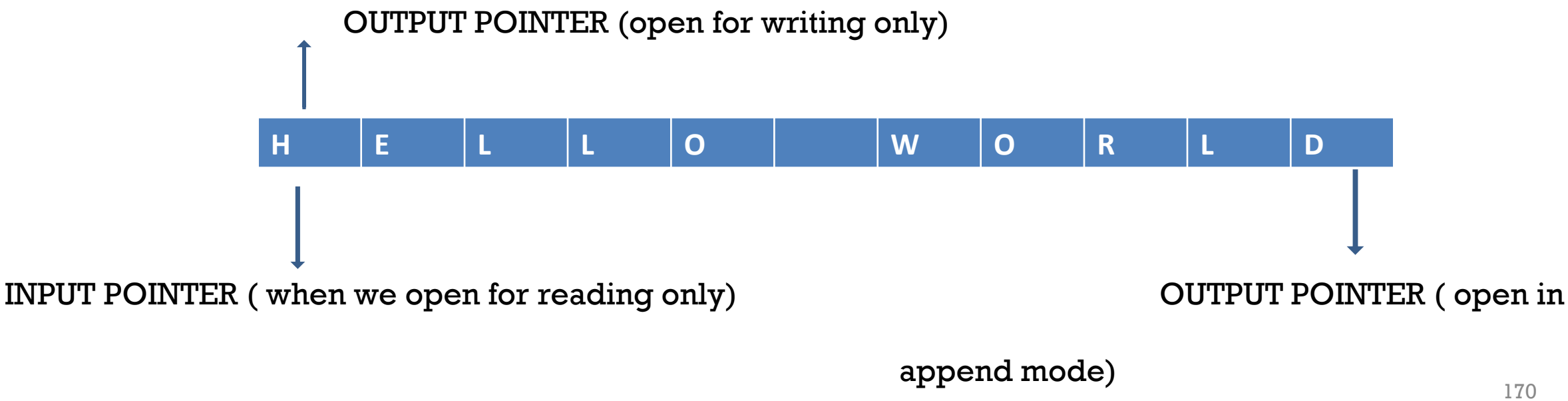
seoul

Functions for manipulation of file pointers

Each file has two associated pointers known as file pointers.

i.input pointer or get pointer used for reading from the file

ii.output pointer or put pointer used for writing into the file



File position pointer

<istream> and <ostream> classes provide member functions for repositioning the *file pointer* (the byte number of the next byte in the file to be read or to be written.)

These member functions are:

seekg (seek get) for istream class moves input pointer to a specified location

seekp (seek put) for ostream class moves output pointer to a specified location

tellg (tell get**)** gives the current position of the input pointer

tellp(tell put) gives the current position of the output pointer

seekg(offset,reposition)

seekp(offset,refposition)

The parameter offset represents the no.of.bytes the file pointer is to be moved from the location specified by the parameter of the reposition

The ref position takes one of the following three constants defined in the ios class

ios::beg start of the file

ios ::cur current position of the pointer

ios::end end of the file

fout.seekg(0,ios::beg) – go to start or repositions the file get pointer to the beginning of the file

fout.seekg(0,ios::cur) – stay at the current position

fout.seekg(0, ios::end) - repositions the file get pointer to the end of the file

fout.seekg(n, ios::beg) – move to the n+1-th byte in the file

fout.seekg(m, ios::cur) – go forward by m-th byte from the current position

fout.seekg(0, ios::end) - repositions the file get pointer to the end of the file

fout.seekg(-m,ios::cur)-go backward by m bytes from the current position

fout.seekg(-m,ios::end) – go backward by m bytes from the end

The same operations can be performed with **<ostream> function member *seekp***

Creating a sequential file

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
int main()
{

  ofstream outClientFile( "clients.dat", ios::out );
  if ( !outClientFile )
{
cerr << "File could not be opened" << endl;
     exit( 1 );
}
cout << "Enter the account, name, and balance.\n"
       << "Enter end-of-file to end input.\n? ";
  int account;
  char name[ 30 ];
  float balance;

  while ( cin >> account >> name >> balance )
{
   outClientFile << account << ' ' << name
                 << ' ' << balance << '\n';
     cout << "? ";
  }
  return 0;
}
```

Reading and printing a sequential file

```
// Reading and printing a sequential file
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>
void outputLine( int, const char *, double );
int main()
{
   // ifstream constructor opens the file
   ifstream inClientFile( "clients.dat", ios::in );

   if ( !inClientFile ) {
      cerr << "File could not be opened\n";
      exit( 1 );
   }
```

```
int account;
   char name[ 30 ];
   double balance;

   cout << setiosflags( ios::left ) << setw( 10 ) << "Account"
        << setw( 13 ) << "Name" << "Balance\n";

   while ( inClientFile >> account >> name >> balance )
     outputLine( account, name, balance );

   return 0;  // ifstream destructor closes the file
}

void outputLine( int acct, const char *name, double bal )
{
   cout << setiosflags( ios::left ) << setw( 10 ) << acct
        << setw( 13 ) << name << setw( 7 ) << setprecision( 2 )
        << resetiosflags( ios::left )
        << setiosflags( ios::fixed | ios::showpoint )
        << bal << '\n';
}
```

Updating a sequential file

Data that is formatted and written to a sequential file **cannot be modified easily** without the risk of destroying other data in the file.

If we want to modify a record of data, the new data may be longer than the old one and it could overwrite parts of the record following it.

Sequential files are inappropriate for so-called “instant access” applications in which a particular record of information must be located immediately.

These applications include banking systems, point-of-sale systems, airline reservation systems, (or any data-base system.)

Instant access is possible with random access files.

Individual records of a **random access file** can be accessed directly (and quickly) without searching many other records.

Example of a Program that Creates a Random Access File

```
// Fig. 14.11: clntdata.h
// Definition of struct clientData used in
// Figs. 14.11, 14.12, 14.14 and 14.15.
#ifndef CLNTDATA_H
#define CLNTDATA_H
struct clientData {
   int accountNumber;
   char lastName[ 15 ];
   char firstName[ 10 ];
   float balance;
};
#endif
// Creating a randomly accessed file sequentially
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include "clntdata.h"
int main()
{
            ofstream outCredit( "credit1.dat",
ios::out);
if ( !outCredit ) {
      cerr << "File could not be opened." << endl;
      exit( 1 );
   }
```

```
clientData blankClient = { 0, "", "", 0.0 };

for ( int i = 0; i < 100; i++ )
  outCredit.write
        (reinterpret_cast<const char *>( &blankClient ),
        sizeof( clientData ) );
  return 0;
}
```

<ostream> member function *write*

The <ostream> member function *write* outputs a fixed number of bytes beginning at a specific location in memory to the specific stream. When the stream is associated with a file, the data is written beginning at the location in the file specified by the “put” file pointer.

Writing data randomly to a random file

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include "clntdata.h"
int main()
{
   ofstream outCredit( "credit.dat", ios::ate );

   if ( !outCredit ) {
      cerr << "File could not be opened." << endl;
      exit( 1 );
   }
```

```
cout << "Enter account number "<< "(1 to 100, 0 to end
input)\n? ";
clientData client;
   cin >> client.accountNumber;

   while ( client.accountNumber > 0 &&
client.accountNumber <= 100 )
 {
      cout << "Enter lastname, firstname, balance\n? ";
      cin >> client.lastName >> client.firstName>>
client.balance;
```

```
outCredit.seekp( ( client.accountNumber - 1 ) *sizeof(
clientData ) );
      outCredit.write(
         reinterpret_cast<const char *>( &client ),
         sizeof( clientData ) );

      cout << "Enter account number\n? ";
      cin >> client.accountNumber;
   }

   return 0;
}
```

Reading data from a random file

```
#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include <stdlib.h>
#include "clntdata.h"
void outputLine( ostream&, const clientData & );
int main()
{
   ifstream inCredit( "credit.dat", ios::in );
   if ( !inCredit ) {
      cerr << "File could not be opened." << endl;
      exit( 1 );
   }
```

```
cout << setiosflags( ios::left ) << setw( 10 ) << "Account"<< setw( 16 ) << "Last Name"
<< setw( 11 )
      << "First Name" << resetiosflags( ios::left )<< setw( 10 ) << "Balance" << endl;

   clientData client;

   inCredit.read( reinterpret_cast<char *>( &client ),
                  sizeof( clientData ) );
```

```
while ( inCredit && !inCredit.eof() ) {

    if ( client.accountNumber != 0 )
       outputLine( cout, client );

    inCredit.read( reinterpret_cast<char *>( &client ),
                   sizeof( clientData ) );
  }

  return 0;
}
```

```
void outputLine( ostream &output, const clientData &c )
{
   output << setiosflags( ios::left ) << setw( 10 )<< c.accountNumber <<
setw( 16 ) << c.lastName
        << setw( 11 ) << c.firstName << setw( 10 )<< setprecision( 2 ) <<
resetiosflags( ios::left )
        << setiosflags( ios::fixed | ios::showpoint )<< c.balance << '\n';
}
```

The <istream> function *read*

```
inCredit.read (reinterpret_cast<char *>(&client),sizeof(clientData));
```

The <istream> function inputs a specified (by sizeof(clientData)) number of bytes from the current position of the specified stream into an object.

STREAMS

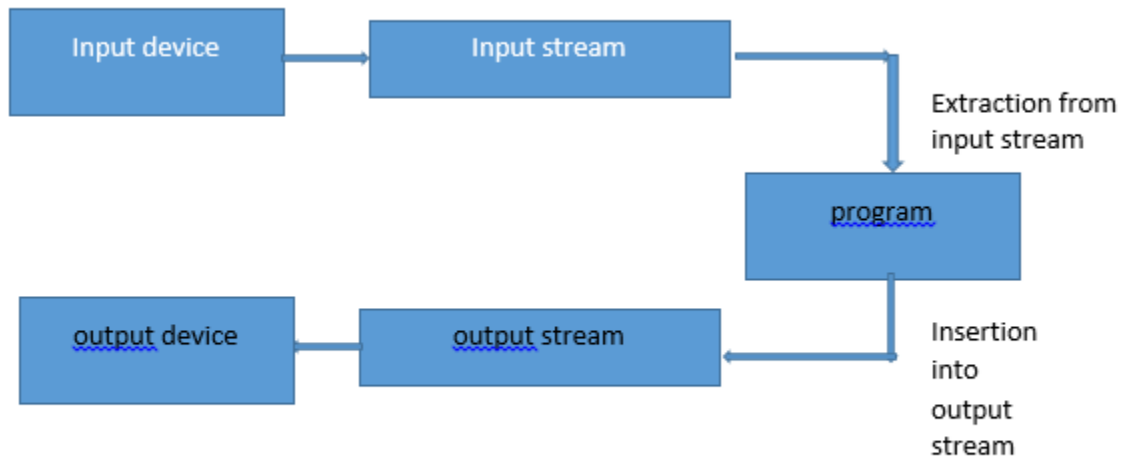
C++ uses the concept of istream and ostream classes to implement its I/O operations with the console and disk files.

C++ streams : - stream acts as an interface between the program and the input-output device.

A stream is a sequence of bytes.

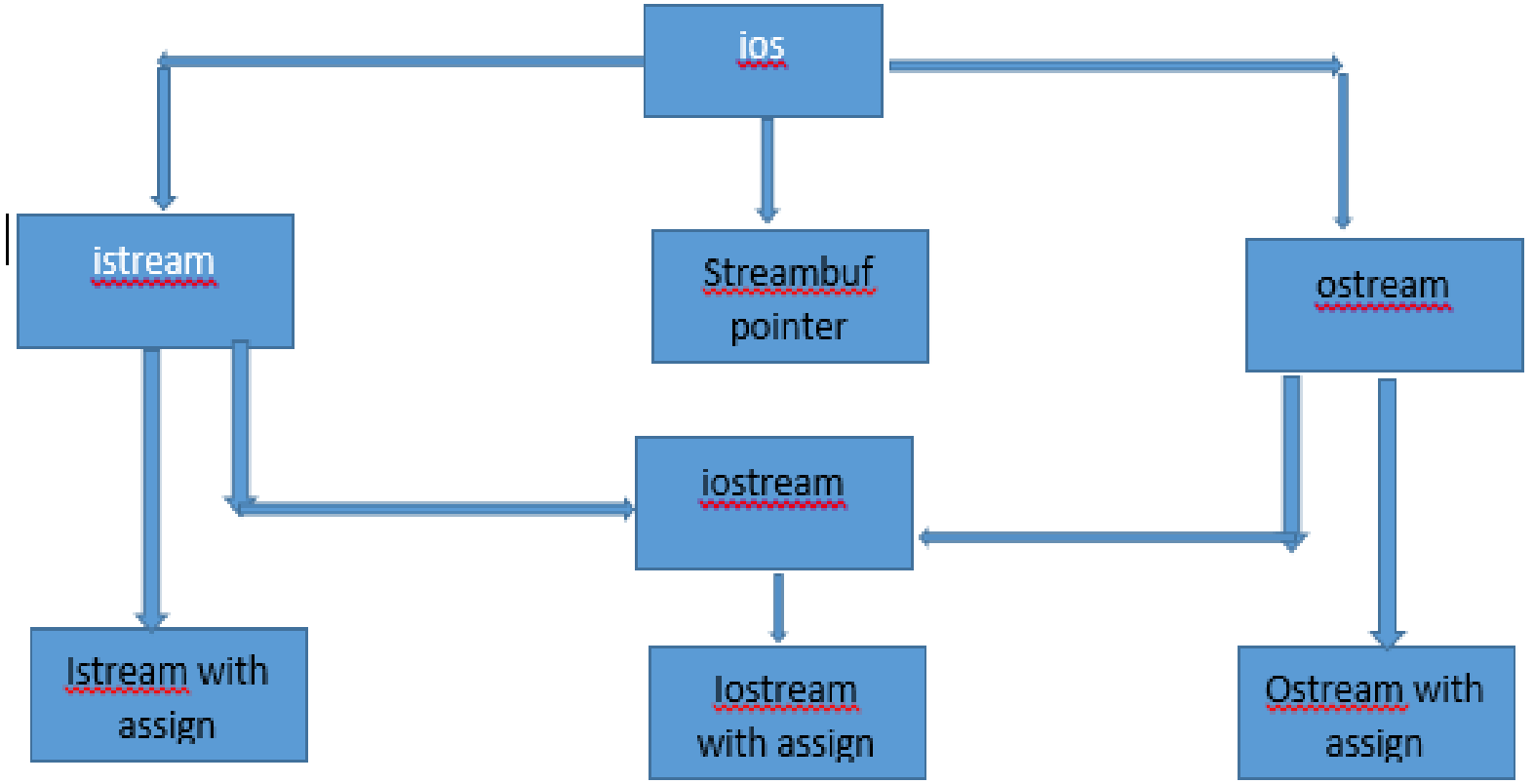
It acts as a source from which the data can be obtained or as a destination to which output data can be sent .

- The source stream that provides data to the program is called the input stream.
- The destination stream that receives output from the program is called the output stream.
- A program extracts the bytes from an input stream and inserts bytes into an output stream
- Input stream connected to the standard input device(keyboard)
- Output stream connected to the standard output device(screen)


Input device
Input stream
Extraction from input stream
program
output device
output stream
Insertion into output stream

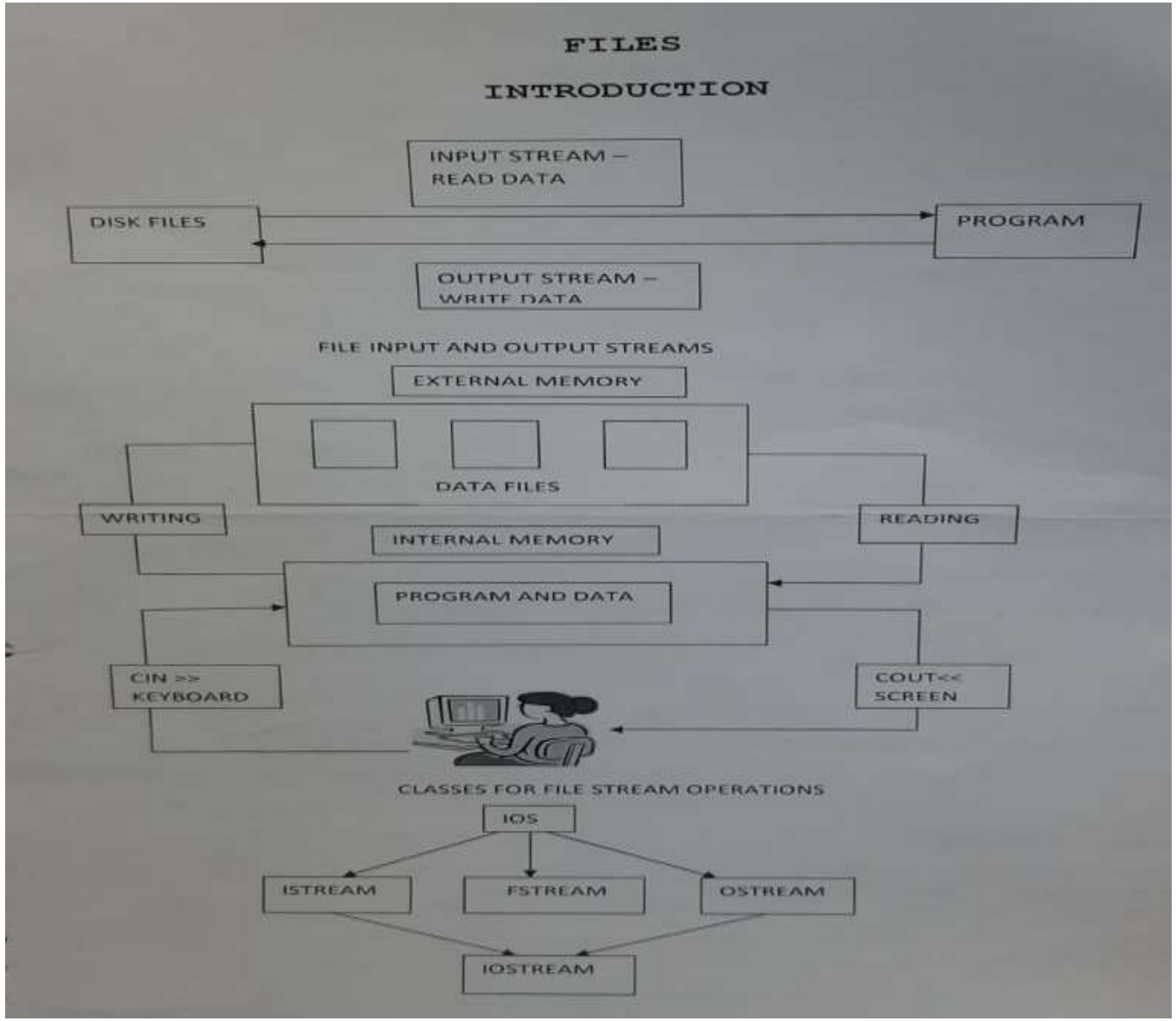
- ios is the baseclass for the istream(input stream) and ostream (output stream) is the base class for iostream(input/output stream).
- The class ios is declared as the virtual base class so that only one copy of its members are inherited by the iostream.

C++ stream classes



Class name	contents
ios(general input/output stream classes)	• Contains basic facilities that are used by all other input and output classes • It also contains a pointer to buffer object(stream buf object) • Declares constants and functions for formatted I/O operations
istream(Input stream)	• Inherits the properties of ios • Declares input functions such as get(),getline() & read() • Contains overloaded extraction operator >>
ostream(output stream)	• Inherits the properties of ios • Declares output functions such as put(),write() • Contains overloaded insertion operator <<
iostream(input/output stream)	• Inherits the properties of istream and ostream through multiple inheritance and contains all the input and output functions
Stream, buf	• Provides an interface to physical devices through buffers • It acts as a base for filebuf class used in files

Streams and Files



- **FSTREAM:**

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

File Operations

- **Opening and Closing a file**
- A file can be opened in two ways
    - Using the constructor of the class
    - Using the member function open() of the class

Opening file using constructor

- File name is used to initialize the file stream object.
  1. Create a file stream object to manage the stream using the appropriate class. The class ofstream is used to create the output stream and class ifstream to create the input stream.
  2. Initialize the file object with the desired filename.

General format

- Streamclass_name file_objectname (“filename”);
- Ofstream outfile (“results”);
- ifstream infile (“results”);

Read and write operations on files

Operation	Console	File stream
READ	Cin>>name;	File_objectname>>name;
WRITE	Cout<<salary;	File_objectname<<salary;

```
#include<iostream.h>
#include<fstream.h>
int main()
{
char name[10];
float sal;
ofstream outfile(“employee”);
for(int i=0;i<3;i++)
{
cout<<“Enter name and salary of employee”<<i+1;
cin>>name>>sal;
outfile<<name<<sal;
}
outfile.close();
ifstream inpfile(“employee”);
for(i=0;i<3;i++)
{
inpfile>>name;
inpfile>>sal;
cout<<“employee”<<i+1;
cout<<name<<Sal;
}
inpfile.close();
}
```

output

File object name

Write Mode

input

Read Mode

Output

```
Enter name and salary of employee 1: Dev 25000
Enter name and salary of employee 2: Aditya 45000
Enter name and salary of employee 3: Sarthak  60000
Employee 1: Dev 25000
Employee 2: Aditya 45000
Employee 3: Sarthak  60000
```

Opening file using member function

File position pointer

- Both istream and ostream provide member functions for repositioning the file-position pointer.

Member Functions	Explanation
tellg()	Current position of get pointer
tellp()	Current position of put pointer
seekg()	Moves get pointer to a specified location (input)
seekp()	Moves put pointer to a specified location (output)

- **Argument seek direction**

Seek Directions	Explanation
ios::beg	Offset counted from the beginning of the stream
ios::cur	Offset counted from the current stream
ios::end	Offset counted from the end of the stream

Examples of positioning the “get” file-position pointer

- **//position to the nth byte of fileobject (assumes ios::beg) – default**
- fileobject.seekg(n);

- **//position n bytes forward in fileobject**
- fileobject.seekg(n, ios::cur);

- **//position n bytes back from end of fileobject**
- fileobject.seekg(n, ios::end);

- **//position at end of fileobject**
- fileobject.seekg(0, ios::end);

```
#include<iostream.h>
#include<fstream.h>
int main()
{
streampos begin,end;
ifstream myfile("example.bin", ios::binary);
begin=myfile.tellg(); //current pos of the file
myfile.seekg(0, ios::end); // moves ptr to the end of file
end=myfile.tellg();
myfile.close();
cout<<"Size is:"<<(end-begin)<<"bytes";
return 0;
}
```

Reads in binary

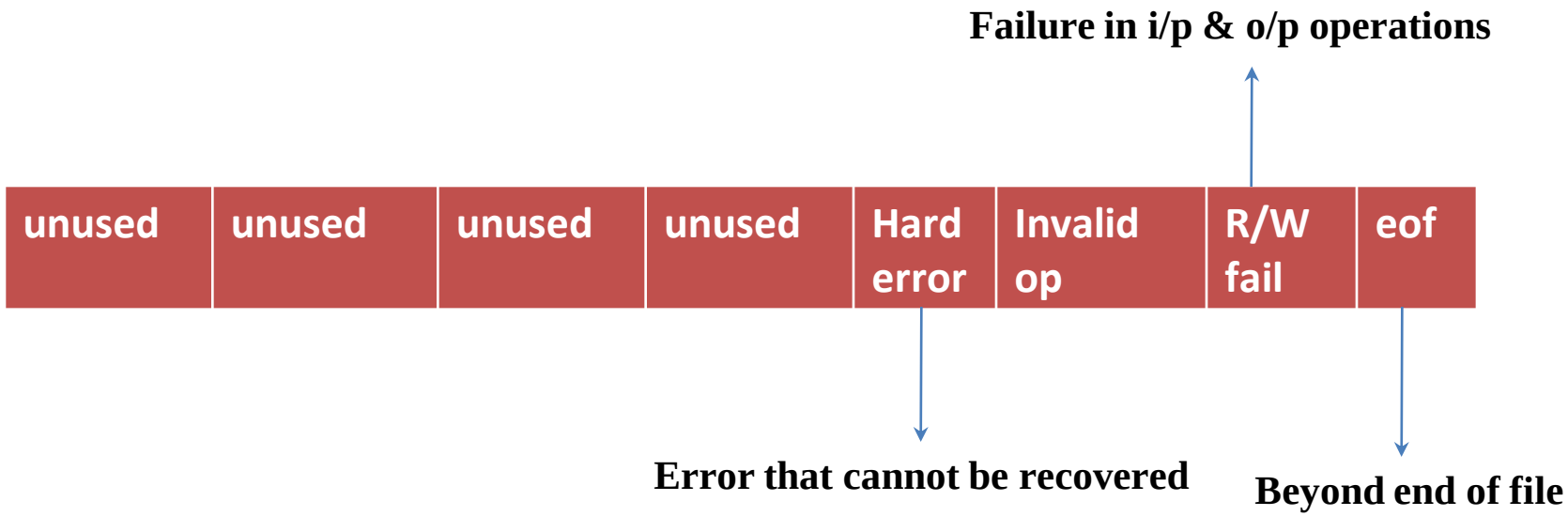
Output

Size is: 40 bytes

Error Handling in file I/O

- **Stream State Flag**

- To check errors and to ensure smooth processing C++ file stream inherit ‘stream-state’ members from the ios class that store the information on the status of a file that is being currently used.



Name	Meaning
eofbit	1 when end-of-file is encountered, 0 otherwise
failbit	1 when a non-fatal I/O error has occurred, 0 otherwise
Badbit	1 when a fatal I/O error has occurred, 0 otherwise
goodbit	0 value

- **Error Handling Functions**

Function	Meaning
Int bad()	Returns non-zero value if an invalid operation, 0 otherwise
Int eof()	Returns non-zero(true) value if end-of-file is encountered while reading, 0 otherwise
Int fail()	Returns non-zero(true) when an input or output operation has failed, 0 otherwise
Int good()	Returns non-zero (true)if no error has occurred, 0 otherwise
Clear()	Resets the error state so further operations can be attempted

Error handling situations in files

1. Open a non existent file in read mode.
2. Open a read only file in ios::out mode.
3. Open a file with invalid name.
4. Read beyond eof.
5. Write more data In a file stored on disk when no sufficient disk space is available.
6. Manipulate data stored in an unopened file.
7. No file is converted with the stream object.
8. Media errors which may occur while reading/writing data.

```
#include<iostream.h>
#include<fstream.h>
#include<process.h>
#include<conio.h>
int main()
{
char fname[20];
cout<<"Enter file name";
cin.getline(fname, 20);
ifstream fin(fname, ios::in);
{
cout<<"Error in opening the file\n";
cout<<"press a key to exit…\n";
return  0;
exit(1);
}
int val1,val2;
int res=0;
char op;
fin>>val1>>val2>>op;
switch(op)
{
case '+':
     res=val+val2;
cout<<"\n"<<val1<<"+"<<val2<<"="<<res;
     break;
case '-':
     res=val-val2;
cout<<"\n"<<val1<<"-"<<val2<<"="<<res;
     break;
case '*':
      res=val*val2;
cout<<"\n"<<val1<<"*"<<val2<<"="<<res;
     break;
case '/':
      if(val2==0)
{
cout<<"\n Divide by Zero Error..!!";
cout<<"\n Press any key to exit…";
     return 0;
     exit(2);
}
     res=val1/val2;
     cout<<val1<<"/"<<val2<<"="<<res;
     break;
}
fin.close();
cout<<"\n press any key to exit…";
return 0;
}
```