
# STANDARD TEMPLATE LIBRARY

# Standard Template Library

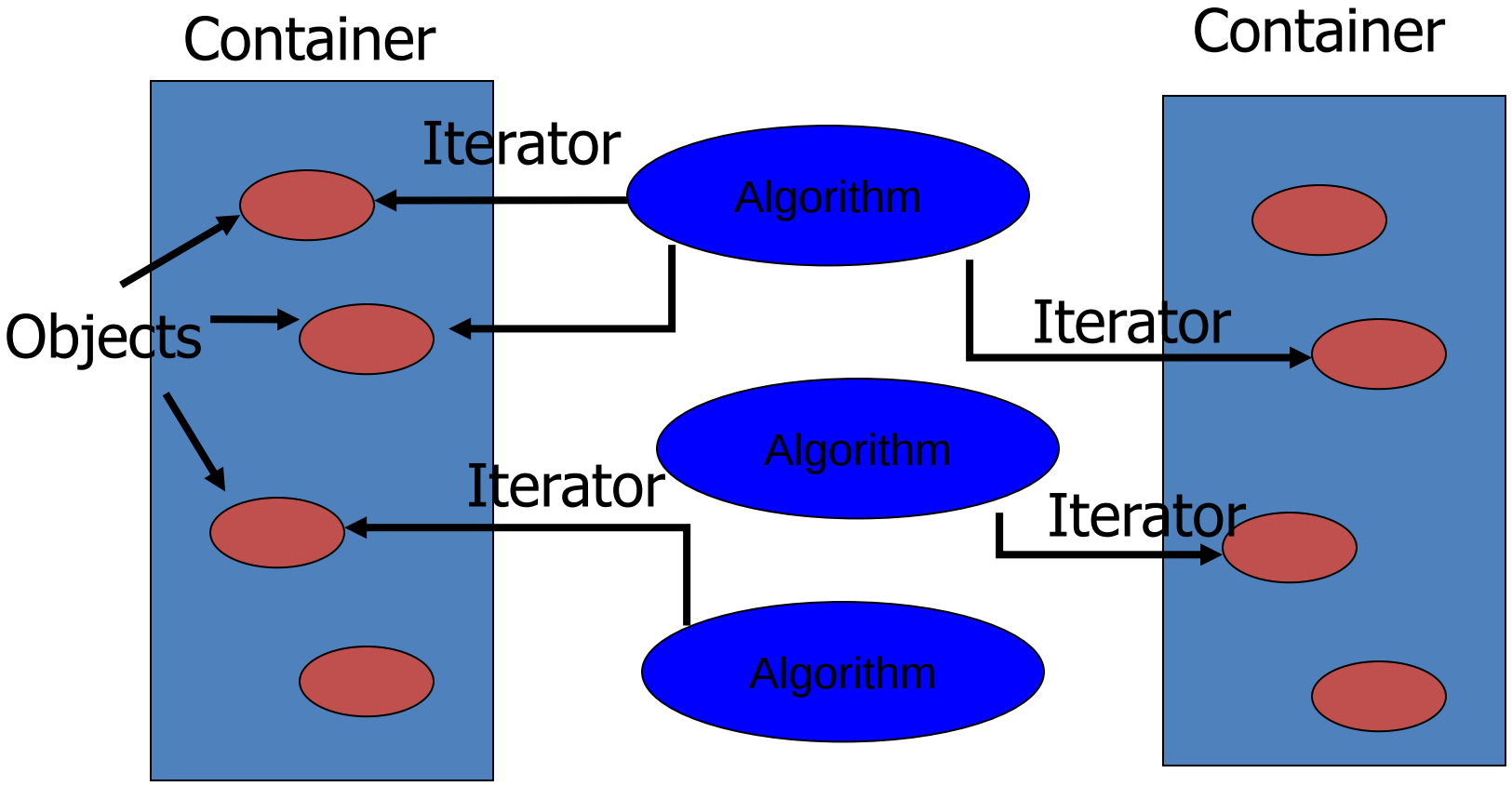
- Templates offers us the feature of generic programming with the help of generic functions and classes( we can create user defined function and class templates).
- C++ also offers us inbuilt template library to provide predefined function and class templates
- The main purpose of this library is to help the user in generic programming, so a set of general purpose templatized classes (Data structures) and functions (algorithms) are there that could be used as a standard approach for storing and processing data.
- The collection of these classes and functions is called Standard Template Library (STL).

# Standard Template Library


- The standard template library (STL) contains
  - Containers
  - Algorithms
  - Iterators
- **A *container*** is an object that actually stores data. It is a way data is organized in memory, for example an array of elements.
- ***Algorithms*** in the STL are procedures that are applied to containers to process their data, for example search for an element in an array, or sort an array.
- ***Iterators*** is an object (like a pointer)that points to an element in a container.we can use iterators to move through the contents of containers.For example you can increment an iterator to point to the next element in an array

# Containers, Iterators, Algorithms

Algorithms use iterators to interact with objects stored in containers



# Containers


- A container is a way to store data, either built-in data types like int and float, or class objects
- The STL provides several basic kinds of containers
  - <vector> : one-dimensional array
  - <list> : double linked list
  - <deque> : double-ended queue
  - <queue> : queue
  - <stack> : stack
  - <set> : set
  - <map> : associative array

# Containers


**Types of containers**

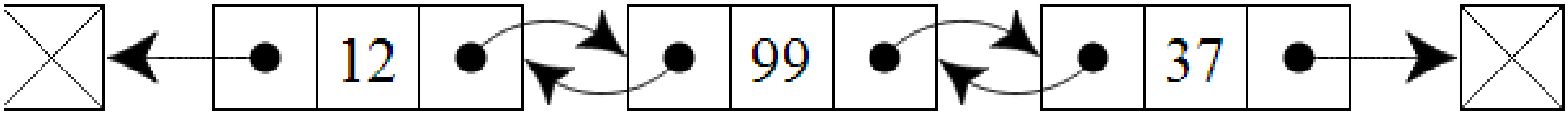
- Sequence container
- Associative container
- Derived container

# Sequence Containers


- A sequence container- stores a set of elements in sequence, in other words each element (except for the first and last one) is preceded by one specific element and followed by another,
  - <vector>
  - <list>
  - <deque>

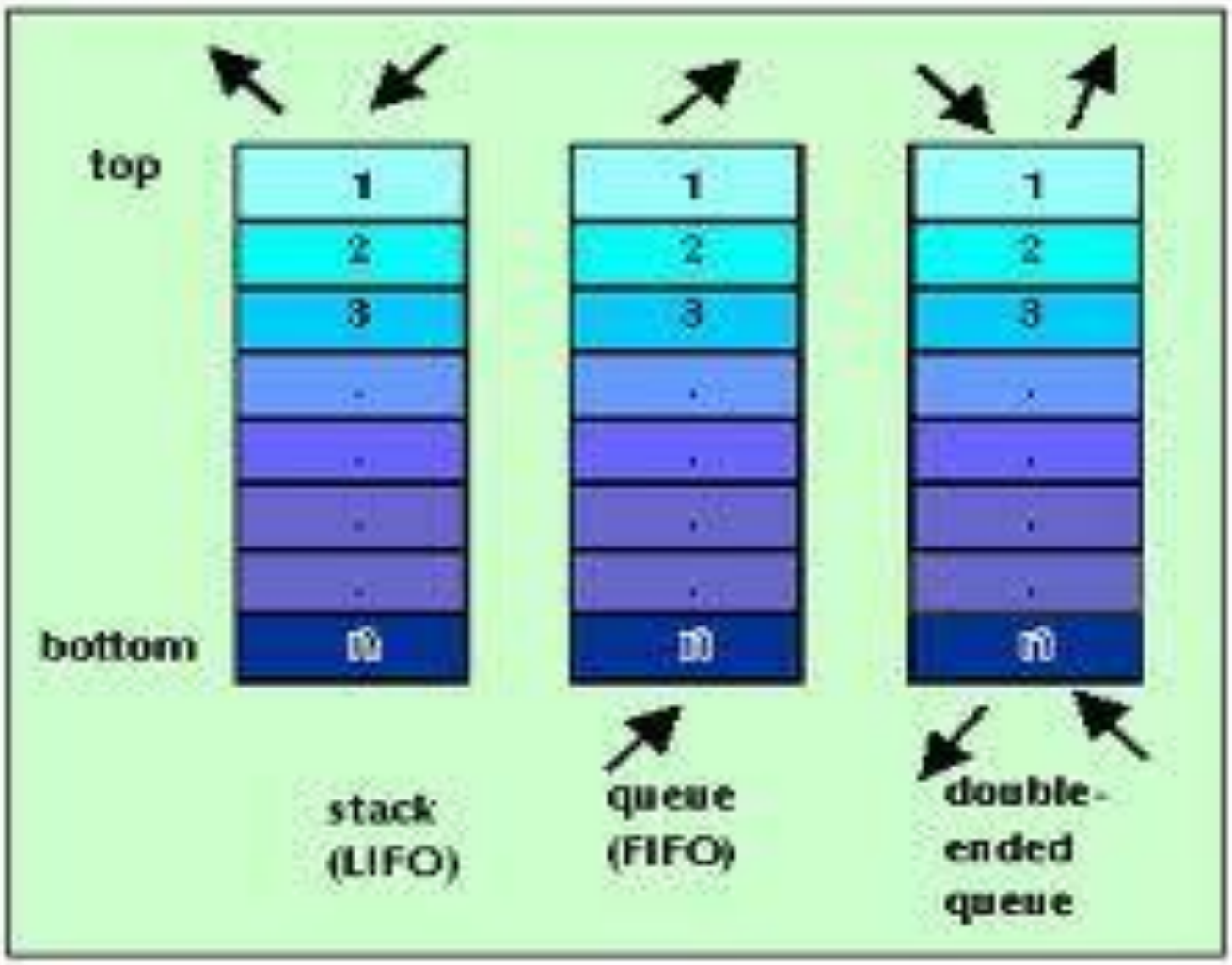
# Sequence Containers

- <vector> is an expandable array (dynamic array) that can shrink or grow in size, but still has the disadvantage of inserting or deleting elements in the middle. Allows insertions and deletions at back.
- <list> is a double linked list (each element has points to its successor and predecessor), it is quick to insert or delete elements but has slow random access.




- <deque> is a double-ended queue, that means one can insert and delete elements from both ends, it is a kind of combination between a stack (last in first out) and a queue (first in first out) and constitutes a compromise between a <vector> and a <list>

# Deque




double-ended queue

# Difference between Array and Vector


- Vector is template class and is C++ only construct whereas arrays are built-in language construct and present in both C and C++.
- Vector are implemented as dynamic arrays with list interface whereas arrays can be implemented as statically or dynamically with primitive data type interface
- Size of arrays are fixed whereas the vectors are resizable i.e. they can grow and shrink as vectors are allocated on heap memory.
- Arrays have to be deallocated explicitly if defined dynamically whereas vectors are automatically de-allocated from heap memory.
- When arrays are passed to a function, a separate parameter for size is also passed whereas in case of passing a vector to a function, there is no such need as vector maintains variables which keeps track of size of container at all times.
- When array becomes full and new elements are inserted; no reallocation is done implicitly whereas When vector becomes larger than its capacity, reallocation is done implicitly.
- Arrays cannot be copied or assigned directly whereas Vectors can be copied or assigned directly.

# Difference between Vector and List

**Insertion and Deletion**

Insertion and Deletion in List is very efficient as compared to vector because to insert an element in list at start, end or middle, internally just a couple of pointers are swapped.

Whereas, in vector insertion and deletion at start or middle will make all elements to shift by one. Also, if there is insufficient contiguous memory in vector at the time of insertion, then a new contiguous memory will be allocated and all elements will be copied there.

So, insertion and deletion in list is much efficient than vector in C++.

**Random Access:**

As List is internally implemented as doubly linked list, therefore no random access is possible in List. It means, to access 15th element in list we need to iterate through first 14 elements in list one by one.

Whereas, vector stores elements at contiguous memory locations like an array. Therefore, in vector random access is possible i.e. we can directly access the 15th element in vector using operator []

# Difference between Vector and Deque


| VECTOR | DEQUE |
| --- | --- |
| Provides insertion and deletion methods at middle and end | Provides insertion and deletion methods at middle, end, beginning |
| Bad performance for insertion and deletion at the front | Good performance for insertion and deletion at the front |
| Stores elements contiguously | It contains lists of memory chunks where elements are stored contiguously |
| Good performance for addition and deletion of elements at the end | Poor performance for addition and deletion of elements at the end |

# Program example 1-vector

```
#include<iostream>
#include<vector>
using namespace std;
int main()
{
   vector<int> v1(10);
   //vector<int>v1; //Zero size vector
   cout<<"size is\t"<<v1.size()<<"\n";
   for(int i=0;i<=9;i++)
   {

         v1[i]=i;
   }
for(int i=10;i<=19;i++)
   {
       v1.push_back(i);
   }
cout<<"size is\t"<<v1.size()<<"\n";
 for(int i=0;i<=19;i++)
   {
        cout<<v1[i]<<"\t";
   }
v1.pop_back();
v1.pop_back();
cout<<"\n New size of the vector:"<<v1.size();
// use iterator to access the values
   vector<int>::iterator v = v1.begin();
   while( v != v1.end())
   {
     cout << "\nvalue of v = " << *v;
     v++;
   }
   return 0;
}
```

# Member functions of vector


- size(): It will give total number of elements in the vector
- erase(): It is used to delete some specific element/ or range of elements
- begin(): It gives the reference of beginning of vector
- end(): It gives the reference after the last element of vector
- clear():It is used to delete all elements of vector at once
- pop_back(): It is used to delete element from the last only
- push_back():It is used to insert element at the last only
- insert(): It is used to insert element at any given position in the vector

# Program example 2-vector


```
#include<iostream>
#include<vector>
using namespace std;
int main()
{
   vector<int> v1(10);
   cout<<"size is\t"<<v1.size()<<"\n";
   for(int i=0;i<=9;i++)
   {

         v1[i]=i;
   }
vector<int>:: iterator i=v1.begin();
i=i+3;
v1.insert(i,100);
//v1.erase(v1.begin());
v1.erase(v1.begin()+1,v1.begin()+5);
 for(int i=0;i<v1.size();i++)
   {
         cout<<v1[i]<<"\t";
   }
   cout<<"\n"<<v1.size();
   v1.clear();
   cout<<"\n"<<v1.size();
}
```

# Program example 1-list

```
#include<iostream>
#include<list>
using namespace std;
void display(list<int> &l)
{
list<int>::iterator p;
for(p=l.begin();p!=l.end();p++)
cout<<"\n"<<*p;
}
int main()
{
list<int> list1;
list<int> list2(5);
for(int i=0;i<3;i++)
list1.push_back(i);
cout<<"\n Displaying first list:";
display(list1);
list<int>::iterator p;
for(p=list2.begin();p!=list2.end();p++)
*p=1;
cout<<"\n Displaying Second list:";
display(list2);
cout<<"\n Pushing element at front:";
list1.push_front(100);
display(list1);
cout<<"\n Poping element from front:";
list2.pop_front();
display(list2);
cout<<"\nSorting first list:";
list1.sort();
display(list1);
cout<<"\n Sorting second list:";
list2.sort();
display(list2);
cout<<"\n Merging list:";
list1.merge(list2);
display(list1);
list1.reverse();
display(list1);
cout<<"\n Reversed merged list";
}
```

# Member functions of List


- size(): It will give total number of elements in the list
- begin():It gives the reference of beginning of list
- end(): It gives the reference after the last element of list
- push_front():It is used to insert element at front
- pop_front():It is used to delete element from front
- push_back(): It is used to insert element at last
- sort():It is used to arrange the elements in ascending order(by default)
- reverse(): It is used to arrange the elements in reverse order
- merge(): It is used to combine the elements of two lists[After combining the elements will be sorted in ascending order automatically]
- clear(): It is used to delete all elements of list at once

# Associative Containers


- An associative container are designed to support direct access to elements using keys and they are non-sequential.
  - The keys, typically a number or a string, are used by the container to arrange the stored elements in a specific order, for example in a dictionary the entries are ordered alphabetically.
  - Elements are usually arranged in sorted fashion automatically
- There are 4 types of associative containers:
  - Set
  - Multiset
  - Map
  - Multimap
- All these store data in a structure called ***tree*** which facilitates fast searching, deletion and insertion but slow for random access

# Associative Containers

- A <set> stores a number of items which contain keys.The keys are the attributes used to order the items, for example a set might store objects of the class.

  Person which are ordered alphabetically using their name

- A <map> stores pairs of objects: a key object and

  an associated value object. A <map> is somehow similar to an array except instead of accessing its elements with index numbers, you access them with indices of an arbitrary type.

- <set> and <map> only allow one key of each value[unique], whereas <multiset> and <multimap> allow multiple identical key values

# Derived containers


- The containers derived from different sequence containers.
- Derived containers
  - Stacks [LIFO]
  - Queues [FIFO]
  - Priority Queues
- Also known as **container adaptors**
- These do not support iterators so can not be used for data manipulation.
- However , they support two member functions
  - pop()
  - push()

# Algorithms


- Algorithms are the functions that can be used generally across a variety of containers for processing of their contents.
- Each containers provides functions for basic operations.
- STL provides various standard algorithms to support more extended or complex operations.
- STL algorithms are not member functions or friends of containers
- These Standard algorithms are standalone templates.
- STL algorithms reinforce the philosophy of reusability.
- include<algorithm>-This header file should be included to use predefined algorithms

# Algorithms


- STL algorithms, based on the nature of operations they perform, may be categorized as under:
  - Non mutating algorithms.
  - Mutating algorithms
  - Sorting/Searching algorithms
  - Set algorithms

# Algorithms


- **Non-mutating algorithms(Non-modifying sequence algorithms)**
- **for_each()** Do specified operation for each element in a sequence
- **find()** Find the first occurence of a specified value in a sequence
- **find_if()** Find the first match of a predicate in a sequence
- **find_first_of()** Find the first occurence of a value from one sequence in another
- **adjacent_find()** Find the first occurence of an adjacent pair of values
- **count()** Count occurences of a value in a sequence
- **count_if()** Count matches of a predicate in a sequence
- **accumulate()** Accumulate (i.e., obtain the sum of) the elements of a sequence
- **equal()** Compare two ranges
- **max_element()** Find the highest element in a sequence
- **min_element()** Find the lowest element in a sequence

# Mutating algorithms(Modifying sequence operations)

- **copy()** Copy range of elements
- **copy_n()** Copy elements (function template )
- **copy_if()** Copy certain elements of range (function template )
- **copy_backward()** Copy range of elements backward (function template )
- **move()** Move range of elements (function template )
- **move_backward()** Move range of elements backward (function template )
- **swap()** Exchange values of two objects (function template
- **swap_ranges()** Exchange values of two ranges (function template )
- **iter_swap()** Exchange values of objects pointed by two iterators (function template
- **transform()** Transform range (function template )
- **replace()** Replace value in range (function template )
- **replace_if()** Replace values in range (function template )
- **replace_copy()** Copy range replacing value (function template )
- **replace_copy_if()** Copy range replacing value (function template )

# Mutating algorithms(Modifying sequence operations

- **fill()** Fill range with value (function template )
- **fill_n()** Fill sequence with value (function template )
- **generate()** Generate values for range with function (function template )
- **generate_n()** Generate values for sequence with function (function template )
- **remove()** Remove value from range (function template )
- **remove_if()** Remove elements from range (function template)
- **remove_copy()** Copy range removing value (function template)
- **remove_copy_if()** Copy range removing values (function template )
- **unique()** Remove consecutive duplicates in range (function template)
- **unique_copy()** Copy range removing duplicates (function template)
- **reverse()** Reverserange (function template )
- **reverse_copy()** Copy range reversed (function template )
- **rotate()** Rotate left the elements in range (function template )
- **rotate_copy()** Copy range rotated left (function template )
- **random_shuffle()** Randomly rearrange elements in range (function template)
- **shuffle ()** Randomly rearrange elements in range using generator (function template )

# Sorting Algorithms


- **sort()** Sort elements in range (function template )
- **stable_sort** Sort elements preserving order of equivalents (function template )
- **partial_sort** Partially sort elements in range (function template )
- **partial_sort_copy** Copy and partially sort range (function template )
- **is_sorted** Check whether range is sorted (function template )
- **is_sorted_until** Find first unsorted element in range (function template )
- **nth_element**Sort element in range (function template )

# **Binary search** (operating on partitioned/sorted ranges)

- **lower_bound** Return iterator to lower bound (function template )
- **upper_bound** Return iterator to upper bound (function template )
- **equal_range** Get subrange of equal elements (function template )
- **binary_search** Test if value exists in sorted sequence (function template )

# Set algorithms


- **set_union** : Union of two sorted ranges
- **set_intersection :** Intersection of two sorted ranges
- **set_difference :** Difference of two sorted ranges
- **set_symmetric_difference** : Symmetric difference of two sorted ranges

# for_each()- Program example 1-Algorithm

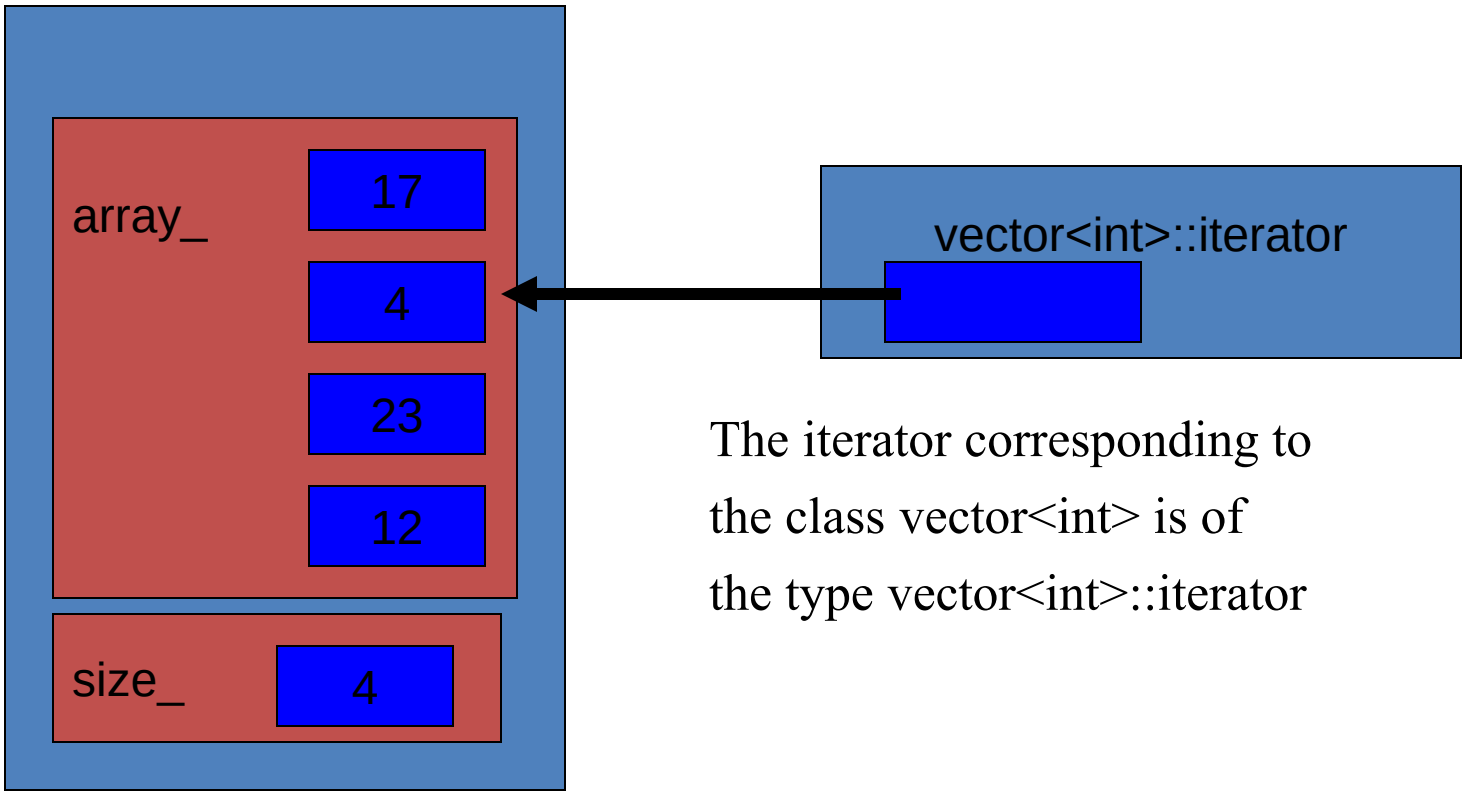
```
#include <vector>
#include <algorithm>
#include <iostream>
void show(int n)
{
  cout << n << ” ”;
}
main()
{
 int arr[] = { 12, 3, 17, 8 };  // standard C array
vector<int> v(arr, arr+4);  // initialize vector with C array
for_each (v.begin(), v.end(), show); // apply function show
                // to each element of vector v
}
```

# find ()- Program example 2-Algorithm

```
#include <vector>
#include <algorithm>
#include <iostream>
main()  {
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;   }
```

# Iterators

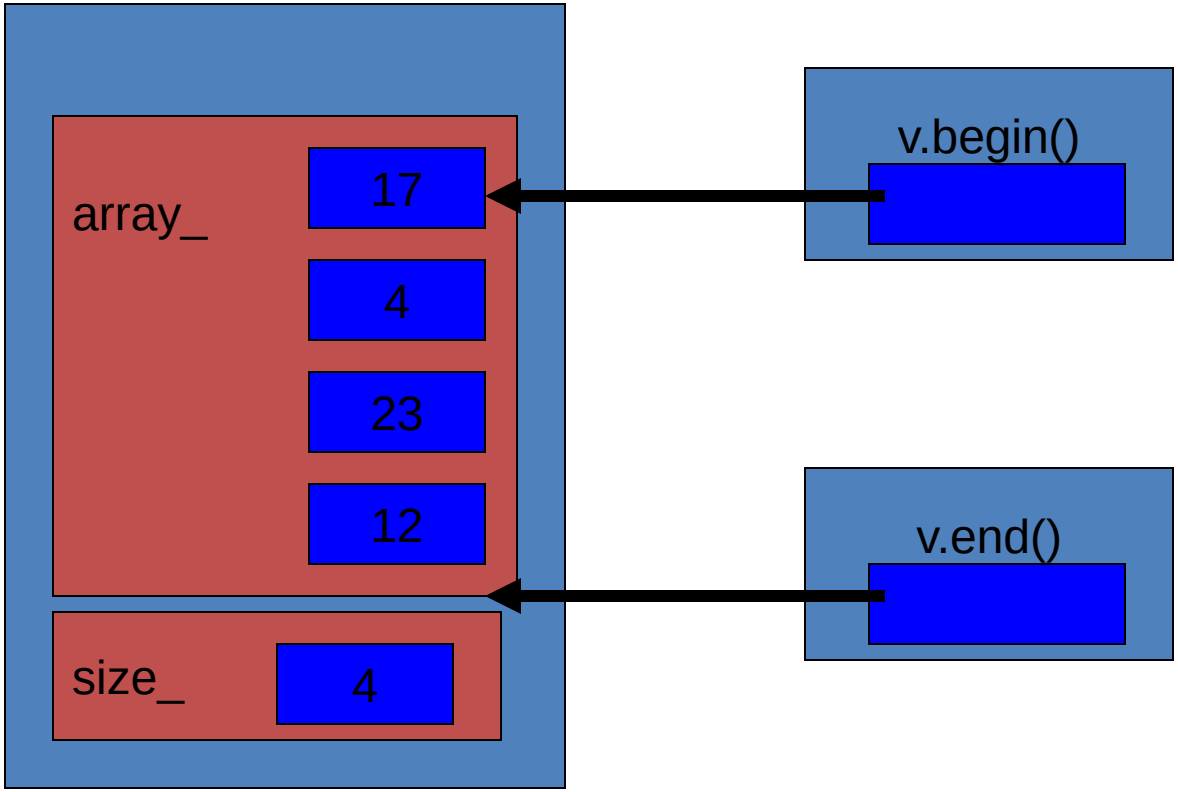
- Iterators are pointer-like entities that are used to access individual elements in a container.
- Often they are used to move sequentially from element to element, a process called *iterating* through a container.




The iterator corresponding to the class vector<int> is of the type vector<int>::iterator

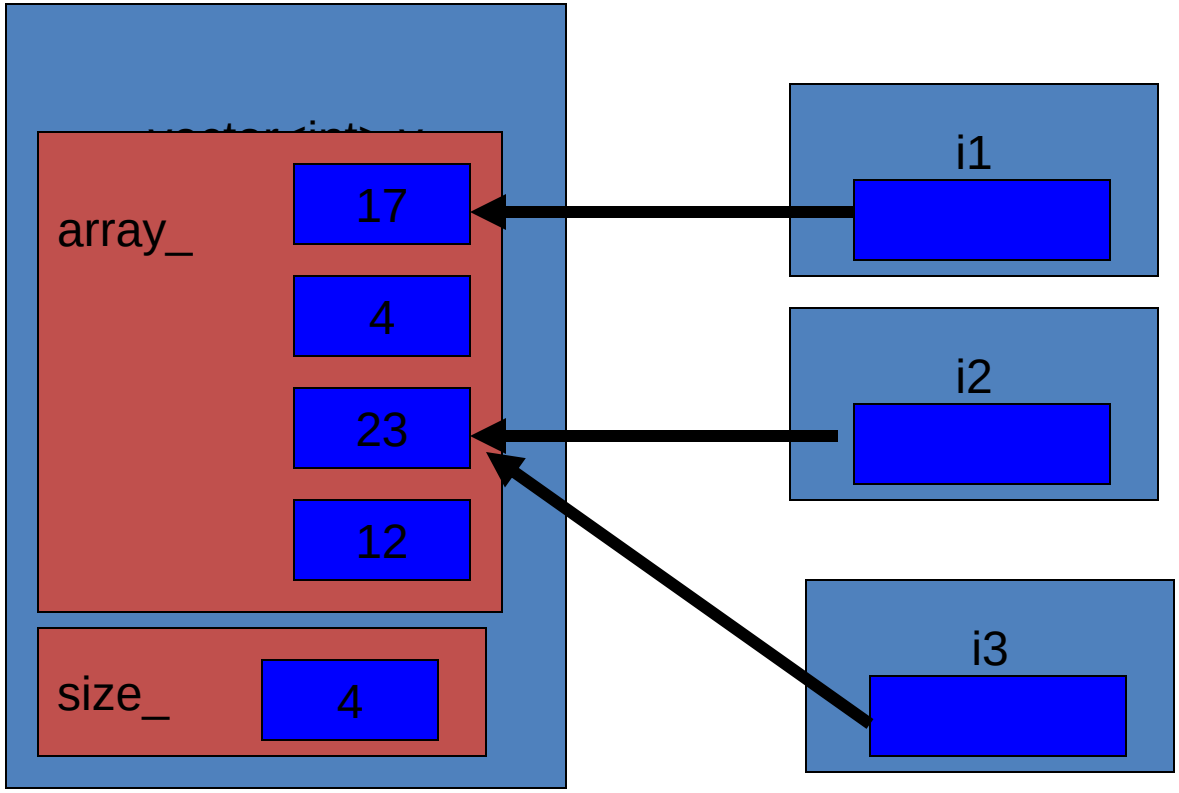
# Iterators

- The member functions begin() and end() return an iterator to the first and past the last element of a container



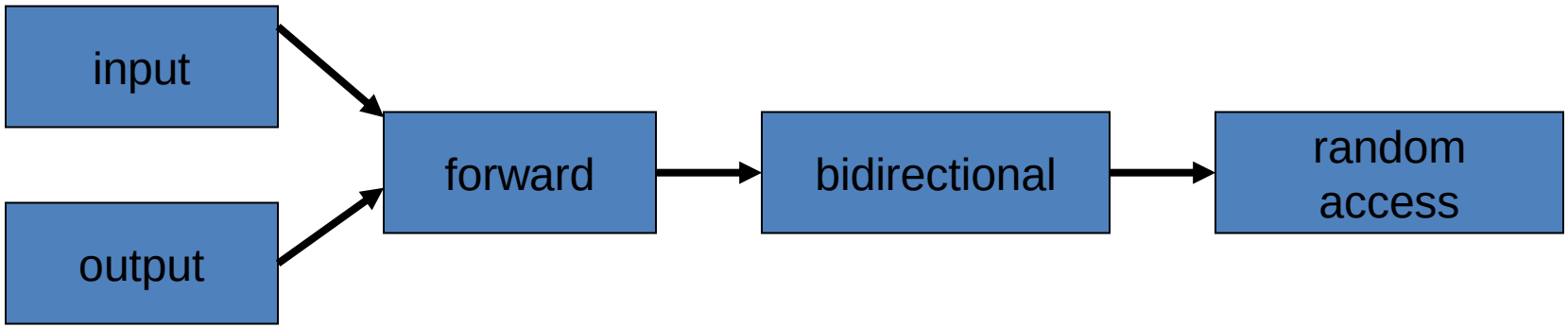
# Iterators

- One can have multiple iterators pointing to different or identical elements in the container



# Iterator Categories

- Not every iterator can be used with every container for example the list class provides no random access iterator
- Every algorithm requires an iterator with a certain level of capability for example to use the [] operator you need a random access iterator
- Iterators are divided into five categories in which a higher (more specific) category always subsumes a lower (more general) category, e.g. An algorithm that accepts a forward iterator will also work with a bidirectional iterator and a random access iterator



# Iterators—Program example

```
#include <vector>
#include <iostream>
main()
{
int arr[] = { 12, 3, 17, 8 };  // standard C array
vector<int> v(arr, arr+4);  // initialize vector with C array
for (vector<int>::iterator i=v.begin(); i!=v.end(); i++)
// initialize i with pointer to first element of v
// i++ increment iterator, move iterator to next element
{
    cout << *i << " ";   // de-referencing iterator returns the
                         // value of the element the iterator points at
}
cout << endl;
}
```