MODULE 27
--THE STL--
CONTAINER PART I
`vector`, `deque`

My Training Period: hours

Note: Compiled using VC++7.0/.Net, win32 empty console mode application. **`g++`** examples given at the end of this Module.

Abilities

- Able to understand the containers.
- Able to understand sequence and associative containers.
- Able to understand and use `vector` sequence container.
- Able to understand and use `deque` sequence container.

- The Standard Template Library (STL) is a **generic library** that provides solutions to managing **collections of data** with an efficient algorithm.
- The STL provides a collection of classes that meet different kind of tasks, with algorithms that operate on the classes. STL components are templates, as you have learned in Module 24, it can be used for arbitrary data types.
- Furthermore, STL also provides a framework for other collection of user defined classes or algorithms. The traditional programming such as **dynamic arrays**, **linked lists**, **binary trees**, **search algorithms** and other **data structures routines** can be implemented using STL more efficiently and easily.
- To easily understand this topic, you should have good understanding of the **traditional arrays data type** and **operations** that can be done on arrays elements such as comparison, sorting, deletion, modification, insertion etc. and as well as templates.

27.1 Introduction: STL components

- The STL consist of the containers, iterators, and algorithms.

27.1.1 Containers

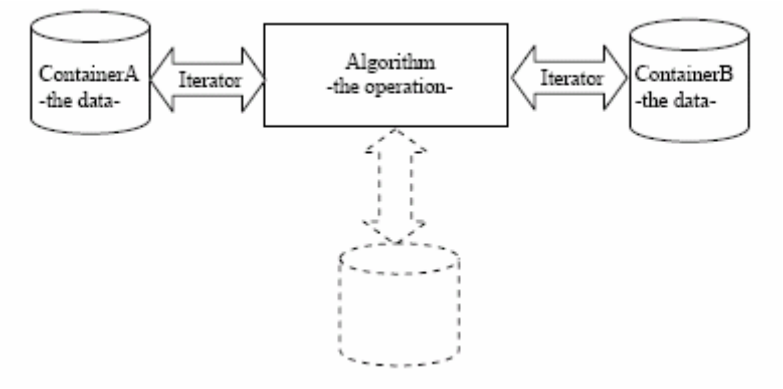
- Container classes' purpose is **to contain other objects**. Each of these classes is a template, and can be instantiated to contain any type of object.
- The STL container classes include `vector`, `list`, `deque`, `set`, `multiset`, `map`, `multimap`, `hash_set`, `hash_multiset`, `hash_map` and `hash_multimap` to suit different kind of tasks. You may find other containers that are implementation dependent/extension.

27.1.2 Iterators

- It is a pointer used to **manipulate the elements** of the objects' collections. These collections may be containers or subsets of containers. Iterators provide common interface for any arbitrary container type.
- Every container class provides its own iterator type but when you try the program examples later, most of the containers have a common iterator types. It is a smart pointer. For simple example, to increment an iterator you call operator ++. To access the value of an iterator you may use operator *.

27.1.3 Algorithms

- Used **to process the elements** of collections. For example, algorithms can **search**, **sort** and **modify**. Algorithms use iterators. Thus, an algorithm has to be written only once to work with arbitrary containers because the iterator interface for iterators is common for all container types.
- We can use a general algorithm to suit our needs even if that need is very special or complex. You will find in the program examples later, most of the member functions for processing the elements or data are common for various containers.
- The data and operations in STL are decoupled. Container classes manage the data, and the operations are defined by the algorithms, used together with the iterators. Conceptually, iterators are the linker between these two components. They let any algorithm interact with any container, graphically shown below.



- Theoretically, you can combine every kind of container with every kind of algorithm. All components work with arbitrary types, a good example of the generic programming concept.
- Containers and algorithms are generic for arbitrary types and classes respectively. The STL provides even more generic components. By using certain **adapters** and **function objects** (or **functors**) you can supplement, constrain, or configure the algorithms and the interfaces for special needs.
- In this module we will discussed in detail about containers and at the same time the iterators and algorithm also will be introduced as well.

27.3 Containers Type

- There are two types of containers.

27.3.1 Sequence containers

- Are **ordered collections** in which every element has a certain position. This '**ordered'** term does not mean ascending or descending, but it refers to a certain position.
- This position depends on the time and place of the insertion, but it is independent of the value of the element. For example, if you put 10 elements into a collection by appending each element at the end of the actual collection, these elements are in the exact order in which you put them.
- The STL contains three predefined sequence container classes: `vector`, `deque`, and `list`.

27.3.2 Associative containers

- Are **sorted collections** in which the **actual position** of an element depends on its **value** due to a **certain sorting criterion**. If you put ten elements into a collection, their **order** depends only on their **value**. The order of insertion doesn't matter.
- The STL contains four predefined associative container classes: set, multiset, map, multimap, hash_map, hash_multimap, hash_set and hash_multiset. Some of these containers are not required by ANSI C++ (ISO/IEC C++).
- An associative container can be considered a special kind of sequence container because sorted collections are ordered according to a sorting criterion. Note that the STL collection types are completely distinct from each other. They have different implementations that are not derived from each other.
- The automatic sorting of elements in associative containers does not mean that those containers are especially designed for sorting elements. You can also sort the elements of a sequence container.
- The key advantage of automatic sorting is better performance when you search elements. In particular, you can always use a binary search, which results in **logarithmic complexity** rather than **linear complexity**.

27.3.2.1 Associative Container Category

- An associative container is a variable-sized container that supports efficient retrieval of elements (values) based on **keys**.
- It supports insertion and removal of elements, but differs from a sequence in that it does not provide a mechanism for inserting an element at a specific position.
- As with all containers, the elements in an associative container are the **type** of `value_type`. Additionally, each element in an associative container has a **key**, of type `key_type`.
- In a **Simple Associative Containers**, the `value_type` and `key_type` are the same that is the elements are their own keys.

- In others, the key is some specific part of the value. Since elements are stored according to their keys, it is essential that the key associated with each element is immutable.
- In simple associative containers this means that the elements themselves are immutable, while in other types of associative containers a **Pair Associative Containers**, the elements themselves are mutable but the part of an element that is its key cannot be modified. This means that an associative container's value type is not a**ssignable**.
- The fact that the value type of an associative container is not assignable has an important consequence: associative containers cannot have mutable iterators. This is simply because a mutable iterator must allow assignment.
- That is, if `i` is a mutable iterator and `t` is an object of `i`'s value type, then `*i = t` must be a valid expression.
- In simple associative containers, where the elements are the keys, the elements are completely immutable; the nested types `iterator` and `const_iterator` are therefore the same. Other types of associative containers, however, do have mutable elements, and do provide iterators through which elements can be modified.
- In pair associative containers, for example, have two different nested types' `iterator` and `const_iterator`.
- Even in this case, `iterator` is not a mutable iterator: as explained above, it does not provide the expression:

```
*i = t.
```

- It is, however, possible to modify an element through such an iterator: if, for example,

`i` is of type `map<int, double>`

- Then:

```
(*i).second = 3
```

- Is a valid expression.
- In some associative containers a **Unique Associative Containers**, it is guaranteed that no two elements have the same key.
- In other associative containers a **Multiple Associative Containers**, multiple elements with the same key are permitted.

27.4 Common Container Operation

- There are operations common to all containers. The following Table is a list of these operations. You will find these operations somewhere in the program examples later.

Sample Code	Operation
`con, con1 and con2 are containers.`	
`ContainerType` `con` `e.g. vector<int> vec0`	Creates an empty container without any element
`ContainerType` `con1(con2)` `e.g. vector<int> vec0(vec1)`	Copies a container of the same type
`ContainerType` `con(begin,end)e.g.` `vector<int> vec0(p.begin(),p.end())`	Creates a container and initializes it with copies of all elements of `[begin, end)`
`con.~`*`ContType`*`()`	Deletes all elements and frees the memory
`con.size()`	Returns the actual number of elements
`con.empty()`	Returns whether the container is empty, equivalent to `size()==0`, but might be faster.
`con.max_size()`	Returns the maximum number of elements possible
`con1 == con2`	Returns whether con1is equal to con2
`con1 != con2`	Returns whether con1is not equal to con2, equivalent to `!(con1==con2)`
`con1 < con2`	Returns whether `con1`is less than `con2`
`con1 > con2`	Returns whether con1is greater than con2, equivalent to `con2 < con1`
`con1 <= con2`	Returns whether `con1`is less than or equal to `con2`, equivalent to `!(con2<con1)`
`con1 >= con2`	Returns whether `con1`is greater than or equal to `con2`,

	equivalent to `!(con1<con2)`
`con1 = con2`	Assignment, assigns all elements of `con1` to `con2`
`con1.swap(con2)`	Swaps the data of `con1`and `con2`
`swap(con1,con2)`	Same but a global function
`con.begin()`	Returns an iterator for the first element
`con.end()`	Returns an iterator for the position after the last element
`con.rbegin()`	Returns a reverse iterator for the first element of a reverse iteration
`con.rend()`	Returns a reverse iterator for the position after the last element of a reverse iteration
`con.insert(position,element)`	Inserts a copy of element.
`con.erase(begin,end)`	Removes all elements of the range `[begin, end)`, some containers return next element not removed
`con.clear()`	Removes all elements, making the container empty
`con.get_allocator()`	Returns the memory model of the container

Table 27.1: Common Operations Examples of Container Classes

- Example:

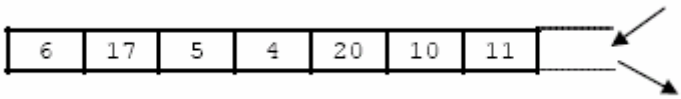
  Initialize with the elements of another container:

```
//ls is a linked list of int
list<int> ls;
...
//copy all elements of the ls list into a con vector
vector<int> con(ls.begin(), ls.end());
```

27.5 Sequence Containers

27.5.1 Vectors

- A vector manages its elements in a dynamic array. It enables random access. Appending and removing elements at the end of the array is very fast.
- However, inserting an element in the middle or at the beginning of the array takes time because all the following elements have to be moved to make room for it while maintaining the order.
- It allows **constant time** insertions and deletions at the end of the sequence. Inserting or deleting elements in the middle of a vector requires **linear time**. The structure of vector can be depicted as follow:

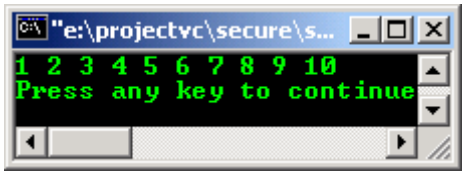


- Vector reallocation occurs when a member function must increase the sequence contained in the vector object beyond its current storage capacity. Other insertions and deletions may alter various storage addresses within the sequence.
- In all such cases, iterators or references that point at altered portions of the sequence become invalid. If no reallocation happens, only iterators and references before the insertion/deletion point remain valid.
- The `vector<bool>` class is a full specialization of the template class vector for elements of type bool with an allocator for the underlying type used by the specialization.
- The `vector<bool>` reference class is a nested class whose objects are able to provide references to elements (single bits) within a `vector<bool>` object.
- The list class container is superior with respect to insertions and deletions at any location within a sequence. The performance of the deque class container is superior with respect to insertions and deletions at the beginning and end of a sequence compared to vector.
- The following general example defines a vector for integer values, inserts ten elements, and prints the elements of the vector:

```
//vector example
#include <iostream>
//vector header file
#include <vector>
using namespace std;
```

```
int main()
{
//vector container for integer elements
//declaration
vector<int> coll;

//append elements with values 1 to 10
for(int i=1; i<=10; ++i)
     coll.push_back(i);

//print all elements separated by a space
for(i=0; i<coll.size(); ++i)
     cout<<coll[i]<<' ';

cout<<endl;
return 0;
}
```

Output:



- Let us dig more detail about the vector. A lot of stuff has been provided by the C++ STL, our task is to learn how to use them properly, before you create or refine your own containers. Do not reinvent the wheel.

<vector> Header Members

- The following section is a <vector> header member.

Operators

Operator	Brief Description
`operator!=`	Tests if the vector object on the left side of the operator is not equal to the vector object on the right side. The return value is **true** if the vectors are not equal; **false** if the vectors are equal.
`operator<`	Tests if the vector object on the left side of the operator is less than the vector object on the right side. The return value is **true** if the vector on the left side of the operator is less than the vector on the right side of the operator; otherwise **false**.
`operator<=`	Tests if the vector object on the left side of the operator is less than or equal to the vector object on the right side. The return value is **true** if the vector on the left side of the operator is less than or equal to the vector on the right side of the operator; otherwise **false**.
`operator==`	Tests if the vector object on the left side of the operator is equal to the vector object on the right side. The return value is **true** if the vector on the left side of the operator is equal to the vector on the right side of the operator; otherwise **false**.
`operator>`	Tests if the vector object on the left side of the operator is greater than the vector object on the right side. The return value is **true** if the vector on the left side of the operator is greater than the vector on the right side of the operator; otherwise **false**.
`operator>=`	Tests if the vector object on the left side of the operator is greater than or equal to the vector object on the right side. The return value is **true** if the vector on the left side of the operator is greater than or equal to the vector on the right side of the vector; otherwise **false**.

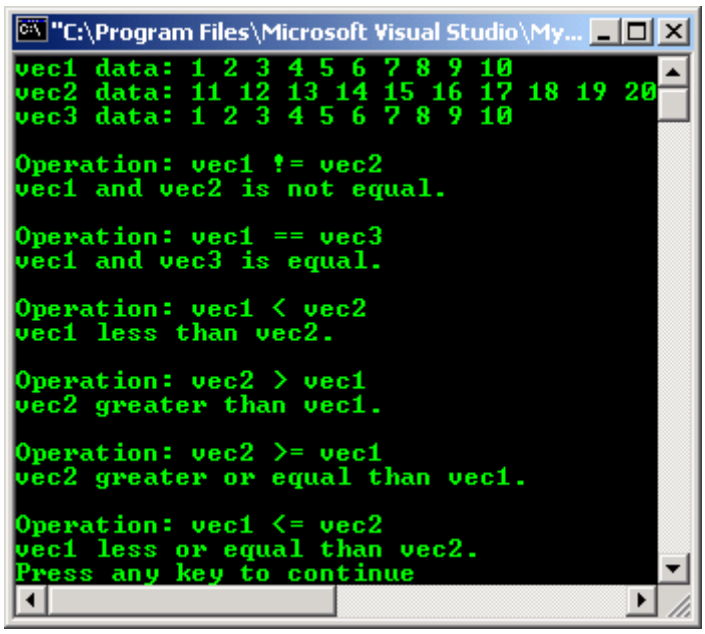
Table 27.2

- Program examples:

```
//vector, operators
#include <vector>
#include <iostream>
```

```
using namespace std;

int main()
{

//vector container for integer elements
//declaration
vector<int> vec1, vec2, vec3;

cout<<"vec1 data: ";
//append elements with values 1 to 10
for(int i=1; i<=10; ++i)
       vec1.push_back(i);
//print all elements separated by a space
for(i=0; i<vec1.size(); ++i)
       cout<<vec1[i]<<' ';
cout<<endl;

cout<<"vec2 data: ";
//append elements with values 1 to 10
for(i=11; i<=20; ++i)
       vec2.push_back(i);
//print all elements separated by a space
for(i=0; i<vec2.size(); ++i)
       cout<<vec2[i]<<' ';
cout<<endl;

cout<<"vec3 data: ";
//append elements with values 1 to 10
for(i=1; i<=10; ++i)
       vec3.push_back(i);
//print all elements separated by a space
for(i=0; i<vec3.size(); ++i)
       cout<<vec3[i]<<' ';
cout<<"\n\n";

cout<<"Operation: vec1 != vec2"<<endl;
if(vec1 != vec2)
      cout<<"vec1 and vec2 is not equal."<<endl;
else
      cout<<"vec1 and vec2 is equal."<<endl;

cout<<"\nOperation: vec1 == vec3"<<endl;
if(vec1 == vec3)
      cout<<"vec1 and vec3 is equal."<<endl;
else
      cout<<"vec1 and vec3 is not equal."<<endl;

cout<<"\nOperation: vec1 < vec2"<<endl;
if(vec1 < vec2)
      cout<<"vec1 less than vec2."<<endl;
else
      cout<<"vec1 is not less than vec2."<<endl;

cout<<"\nOperation: vec2 > vec1"<<endl;
if(vec2 > vec1)
      cout<<"vec2 greater than vec1."<<endl;
else
      cout<<"vec2 is not greater than vec1."<<endl;

cout<<"\nOperation: vec2 >= vec1"<<endl;
if(vec2 >= vec1)
      cout<<"vec2 greater or equal than vec1."<<endl;
else
      cout<<"vec2 is not greater or equal than vec1."<<endl;

cout<<"\nOperation: vec1 <= vec2"<<endl;
if(vec1 <= vec2)
      cout<<"vec1 less or equal than vec2."<<endl;
else
      cout<<"vec1 is not less or equal than vec2."<<endl;
   return 0;
}
```

Output:



vector Class Template

Class	Description
`vector Class`	A template class of sequence containers that arrange elements of a given type in a linear arrangement and allow fast random access to any element.

Table 27.3

- The STL vector class is a template class of sequence containers that arrange elements of a given type in a linear arrangement and allow fast random access to any element.
- They should be the preferred container for a sequence when random-access performance is concerned.

vector Class Template Members

vector Class template Typedefs

Typedef	Description
`allocator_type`	A type that represents the allocator class for the vector object.
`const_iterator`	A type that provides a random-access iterator that can read a `const` element in a vector.
`const_pointer`	A type that provides a pointer to a `const` element in a vector.
`const_reference`	A type that provides a reference to a `const` element stored in a vector for reading and performing `const` operations.
`const_reverse_iterator`	A type that provides a random-access iterator that can read any `const` element in the vector.
`difference_type`	A type that provides the difference between the addresses of two elements in a vector.
`iterator`	A type that provides a random-access iterator that can read or modify any element in a vector.
`pointer`	A type that provides a pointer to an element in a vector.
`reference`	A type that provides a reference to an element stored in a vector.
`reverse_iterator`	A type that provides a random-access iterator that can read or modify any element in a reversed vector.
`size_type`	A type that counts the number of elements in a vector.
`value_type`	A type that represents the data type stored in a vector.

Table 27.4

vector Class Template Member Functions

Member function	Description
`assign()`	Erases a vector and copies the specified elements to the empty vector.
`at()`	Returns a reference to the element at a specified location in the vector.

back()	Returns a reference to the last element of the vector.
begin()	Returns a random-access iterator to the first element in the container.
capacity()	Returns the number of elements that the vector could contain without allocating more storage.
clear()	Erases the elements of the vector.
empty()	Tests if the vector container is empty.
end()	Returns a random-access iterator that point just beyond the end of the vector.
erase()	Removes an element or a range of elements in a vector from specified positions.
front()	Returns a reference to the first element in a vector.
get_allocator()	Returns an object to the allocator class used by a vector.
insert()	Inserts an element or a number of elements into the vector at a specified position.
max_size()	Returns the maximum length of the vector.
pop_back()	Deletes the element at the end of the vector.
push_back()	Add an element to the end of the vector.
rbegin()	Returns an iterator to the first element in a reversed vector.
rend()	Returns an iterator to the end of a reversed vector.
resize()	Specifies a new size for a vector.
reserve()	Reserves a minimum length of storage for a vector object.
size()	Returns the number of elements in the vector.
swap()	Exchanges the elements of two vectors.
vector()	Vector constructor, constructs a vector of a specific size or with elements of a specific value or with a specific allocator or as a copy of some other vector.

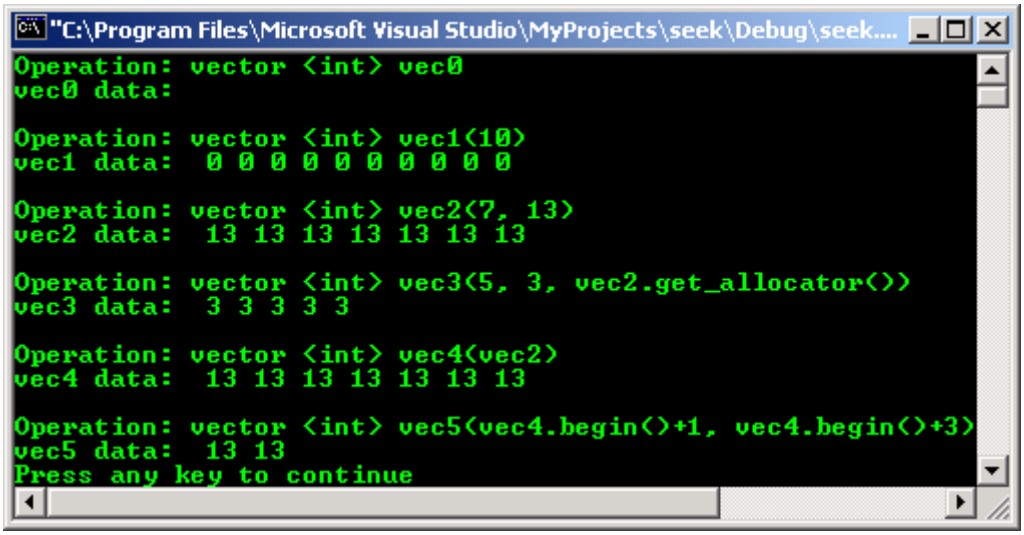
Table 27.5

- The following section demonstrate the program examples using the member functions and the typedefs
- Vector constructor, constructs a vector of a specific size or with elements of a specific value or with a specific allocator or as a copy of all or part of some other vector.
- All constructors store an allocator object and initialize the vector.

```
//vector constructors
#include <vector>
#include <iostream>
using namespace std;

int main()
{
vector <int>::iterator vec0Iter, vec1Iter, vec2Iter, vec3Iter, vec4Iter, vec5Iter;

//Create an empty vector vec0
vector <int> vec0;

//Create a vector vec1 with 10 elements of default value 0
vector <int> vec1(10);

//Create a vector vec2 with 7 elements of value 13
vector <int> vec2(7, 13);

//Create a vector vec3 with 5 elements of value 3 and with the allocator
//of vector vec2
vector <int> vec3(5, 3, vec2.get_allocator());

//vector vec4, a copy of vector vec2
vector <int> vec4(vec2);

//Create a vector vec5 by copying the range of vec4[_First, _Last)
vector <int> vec5(vec4.begin() + 1, vec4.begin() + 3);

cout<<"Operation: vector <int> vec0\n";
cout<<"vec0 data: ";
for(vec0Iter = vec0.begin(); vec0Iter != vec0.end(); vec0Iter++)
cout<<" "<<*vec0Iter;
cout<<endl;

cout<<"\nOperation: vector <int> vec1(10)\n";
cout<<"vec1 data: ";
for(vec1Iter = vec1.begin(); vec1Iter != vec1.end(); vec1Iter++)
cout<<" "<<*vec1Iter;
```

```
cout<<endl;

cout<<"\nOperation: vector <int> vec2(7, 13)\n";
cout<<"vec2 data: ";
for(vec2Iter = vec2.begin(); vec2Iter != vec2.end(); vec2Iter++)
cout<<" "<<*vec2Iter;
cout<<endl;

cout<<"\nOperation: vector <int> vec3(5, 3, vec2.get_allocator())\n";
cout<<"vec3 data: ";
for(vec3Iter = vec3.begin(); vec3Iter != vec3.end(); vec3Iter++)
cout<<" "<<*vec3Iter;
cout<<endl;

cout<<"\nOperation: vector <int> vec4(vec2)\n";
cout<<"vec4 data: ";
for(vec4Iter = vec4.begin(); vec4Iter != vec4.end(); vec4Iter++)
cout<<" "<<*vec4Iter;
cout<<endl;

cout<<"\nOperation: vector <int> vec5(vec4.begin()+1, vec4.begin()+3)\n";
cout<<"vec5 data: ";
for(vec5Iter = vec5.begin(); vec5Iter != vec5.end(); vec5Iter++)
cout<<" "<<*vec5Iter;
cout<<endl;
return 0;
}
```

Output:

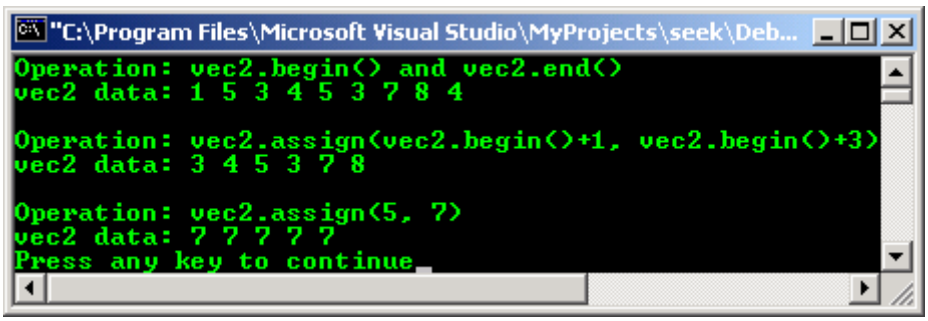


- After erasing any existing elements in a vector, `assign()` either inserts a specified range of elements from the original vector into a vector or inserts copies of a new element of a specified value into a vector.

```
//vector, assign()
#include <vector>
#include <iostream>
using namespace std;

int main()
{
vector <int> vec2;
vector <int>::iterator Iter;

vec2.push_back(1);
vec2.push_back(5);
vec2.push_back(3);
vec2.push_back(4);
vec2.push_back(5);
vec2.push_back(3);
vec2.push_back(7);
vec2.push_back(8);
vec2.push_back(4);

cout<<"Operation: vec2.begin() and vec2.end()"<<endl;
```

```
cout<<"vec2 data: ";
for(Iter = vec2.begin(); Iter != vec2.end(); Iter++)
cout<<*Iter<<" ";
cout<<"\n\n";

cout<<"Operation: vec2.assign(vec2.begin()+1, vec2.begin()+3)"<<endl;
vec2.assign(vec2.begin()+2, vec2.begin()+8);
cout<<"vec2 data: ";
for(Iter = vec2.begin(); Iter != vec2.end(); Iter++)
cout<<*Iter<<" ";
cout<<"\n\n";

cout<<"Operation: vec2.assign(5, 7)"<<endl;
vec2.assign(5, 7);
cout<<"vec2 data: ";
for(Iter = vec2.begin(); Iter != vec2.end(); Iter++)
cout<<*Iter<<" ";
cout<<endl;
return 0;
}
```

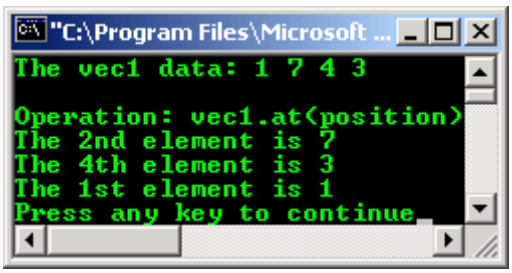
Output:



- The return value is a reference to the element subscripted in the argument. If `_Off` is greater than the size of the vector, `at()` throws an exception.
- If the return value of `at()` is assigned to a `const_reference`, the vector object cannot be modified. If the return value of `at()` is assigned to a `reference`, the vector object can be modified.

```
//vector, at()
#include <vector>
#include <iostream>
using namespace std;

int main()
{
vector <int> vec1;

vec1.push_back(1);
vec1.push_back(7);
vec1.push_back(4);
vec1.push_back(3);

//print all elements separated by a space
cout<<"The vec1 data: ";
for(int i=0; i<vec1.size(); ++i)
     cout<<vec1[i]<<' ';

cout<<"\n\nOperation: vec1.at(position)";
const int &x = vec1.at(1);
int &y = vec1.at(3);
int &z = vec1.at(0);
cout<<"\nThe 2nd element is "<<x<<endl;
cout<<"The 4th element is "<<y<<endl;
cout<<"The 1st element is "<<z<<endl;
return 0;
}
```

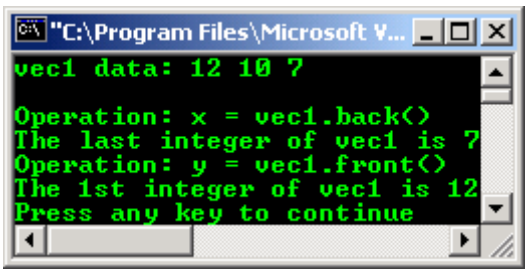
Output:



- For `back()`, the return value is the last element of the vector. If the vector is empty, the return value is undefined.
- If the return value of `back()` is assigned to a `const_reference`, the vector object cannot be modified. If the return value of `back()` is assigned to a `reference`, the vector object can be modified.
- For `front()`, the return value is a reference to the first element in the vector object. If the vector is empty, the return is undefined.
- If the return value of `front()` is assigned to a `const_reference`, the vector object cannot be modified. If the return value of `front()` is assigned to a `reference`, the vector object can be modified.

```
//vector, back() and front()
#include <vector>
#include <iostream>
using namespace std;

int main()
{
vector <int> vec1, vec2;
vec1.push_back(12);
vec1.push_back(10);
vec1.push_back(7);

//print all elements separated by a space
cout<<"vec1 data: ";
for(int i=0; i<vec1.size(); ++i)
         cout<<vec1[i]<<' ';
cout<<endl;


int& x = vec1.back();
const int& y = vec1.front();

cout<<"\nOperation: x = vec1.back()\n";
cout<<"The last integer of vec1 is "<<x<<endl;

cout<<"Operation: y = vec1.front()\n";
cout<<"The 1st integer of vec1 is "<<y<<endl;

return 0;
}
```

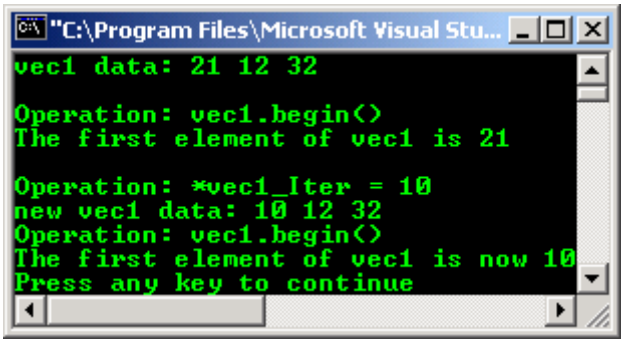
Output:



- The return value is a random-access iterator addressing the first element in the vector or to the location succeeding an empty vector.
- If the return value of `begin()` is assigned to a `const_iterator`, the vector object cannot be modified. If the return value of `begin()` is assigned to an `iterator`, the vector object can be modified.

```
//vector, begin()
#include <vector>
#include <iostream>
using namespace std;

int main()
{
vector <int> vec1;
vector <int>::iterator vec1_Iter;

vec1.push_back(21);
vec1.push_back(12);
vec1.push_back(32);

cout<<"vec1 data: ";
for(int i=0; i<vec1.size(); ++i)
cout<<vec1[i]<<' ';
cout<<endl;

cout<<"\nOperation: vec1.begin()\n";
vec1_Iter = vec1.begin();
cout<<"The first element of vec1 is "<<*vec1_Iter<<endl;

cout<<"\nOperation: *vec1_Iter = 10\n";
*vec1_Iter = 10;
cout<<"new vec1 data: ";
for(i=0; i<vec1.size(); ++i)
cout<<vec1[i]<<' ';
cout<<endl;

cout<<"Operation: vec1.begin()\n";
vec1_Iter = vec1.begin();
cout<<"The first element of vec1 is now "<<*vec1_Iter<<endl;
return 0;
}
```

Output:

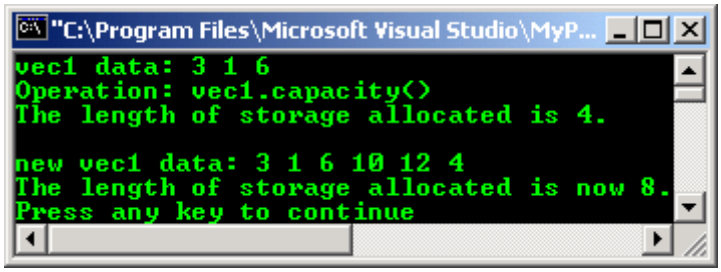


- The return value is the current length of storage allocated for the vector.
- The member function `resize()` will be more efficient if sufficient memory is allocated to accommodate it. Use the member function `reserve()` to specify the amount of memory allocated.

```
//vector, capacity()
//and size()
#include <vector>
#include <iostream>
using namespace std;

int main()
{
vector <int> vec1;

vec1.push_back(3);
vec1.push_back(1);
vec1.push_back(6);

cout<<"vec1 data: ";
for(int i=0; i<vec1.size(); ++i)
cout<<vec1[i]<<' ';
cout<<endl;

cout<<"Operation: vec1.capacity()\n";
cout<<"The length of storage allocated is "<<vec1.capacity()<<"."<<endl;
```

```
vec1.push_back(10);
vec1.push_back(12);
vec1.push_back(4);

cout<<"\nnew vec1 data: ";
for(i=0; i<vec1.size(); ++i)
cout<<vec1[i]<<' ';
cout<<endl;

cout<<"The length of storage allocated is now "<<vec1.capacity()<<"."<<endl;
return 0;
}
```

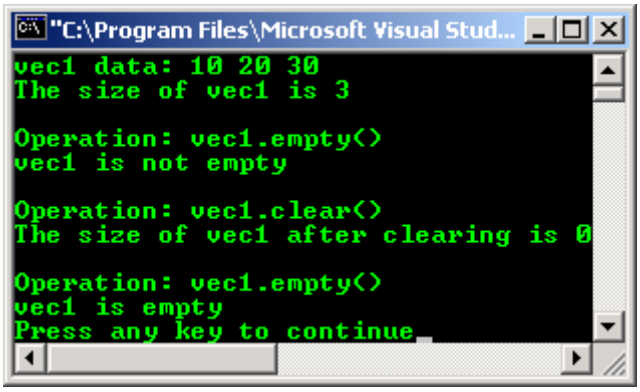
Output:



- For `empty()`, the return value is true if the vector is empty; false if the vector is not empty.

```
//vector, clear(), empty()
//and size()
#include <vector>
#include <iostream>
using namespace std;

int main()
{
vector <int> vec1;

vec1.push_back(10);
vec1.push_back(20);
vec1.push_back(30);

cout<<"vec1 data: ";
for(int i=0; i<vec1.size(); ++i)
cout<<vec1[i]<<' ';
cout<<endl;

cout<<"The size of vec1 is "<<vec1.size()<<endl;

cout<<"\nOperation: vec1.empty()"<<endl;
if(vec1.empty())
cout<<"vec1 is empty"<<endl;
else
cout<<"vec1 is not empty"<<endl;

cout<<"\nOperation: vec1.clear()"<<endl;
vec1.clear();
cout<<"The size of vec1 after clearing is "<<vec1.size()<<endl;

cout<<"\nOperation: vec1.empty()"<<endl;
if(vec1.empty())
cout<<"vec1 is empty"<<endl;
else
cout<<"vec1 is not empty"<<endl;
return 0;
}
```

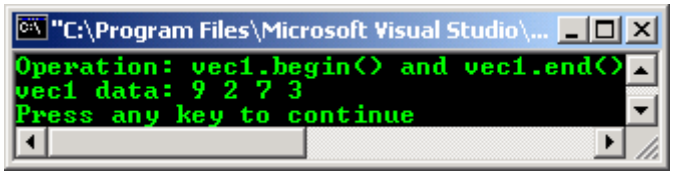
Output:



- For `end()`, the return value is a pointer to the end of the vector object. If the vector is empty, the result is undefined.
- If the return value of `end()` is assigned to a variable of type `const_iterator`, the vector object cannot be modified. If the return value of `end()` is assigned to a variable of type `iterator`, the vector object can be modified.

```
//vector, begin(), end()
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> vec1;
   vector <int>::iterator vec1_Iter;

   vec1.push_back(9);
   vec1.push_back(2);
   vec1.push_back(7);
   vec1.push_back(3);

   cout<<"Operation: vec1.begin() and vec1.end()"<<endl;
   cout<<"vec1 data: ";
   for(vec1_Iter = vec1.begin(); vec1_Iter != vec1.end(); vec1_Iter++)
      cout<<*vec1_Iter<<' ';
   cout<<endl;
   return 0;
}
```

Output:

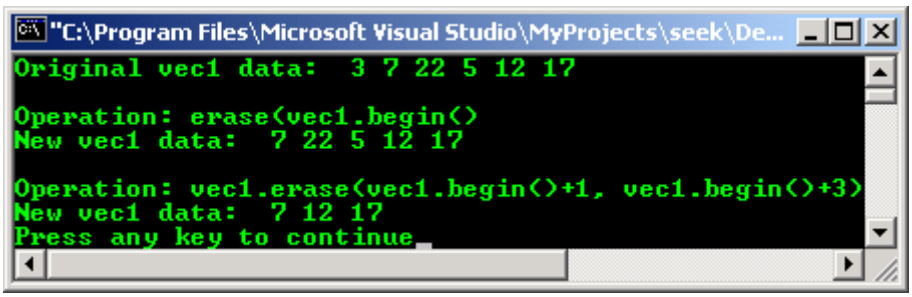


- The return value for `erase()` is an iterator that designates the first element remaining beyond any elements removed, or a pointer to the end of the vector if no such element exists.

```
//vector, erase(), begin()
//and end()
#include <vector>
#include <iostream>
using namespace std;

int main()
{
vector <int> vec1;
vector <int>::iterator Iter;
vec1.push_back(3);
vec1.push_back(7);
vec1.push_back(22);
vec1.push_back(5);
vec1.push_back(12);
vec1.push_back(17);
```

```
cout<<"Original vec1 data: ";
for(Iter = vec1.begin(); Iter != vec1.end(); Iter++)
cout<<" "<<*Iter;
cout<<endl;

cout<<"\nOperation: erase(vec1.begin()"<<endl;
vec1.erase(vec1.begin());
cout<<"New vec1 data: ";
for(Iter = vec1.begin(); Iter != vec1.end(); Iter++)
  cout<<" "<<*Iter;
cout<<endl;

cout<<"\nOperation: vec1.erase(vec1.begin()+1, vec1.begin()+3)"<<endl;
vec1.erase(vec1.begin() + 1, vec1.begin() + 3);
cout<<"New vec1 data: ";
for(Iter = vec1.begin(); Iter != vec1.end(); Iter++)
   cout<<" "<<*Iter;
cout<<endl;
return 0;
}
```

Output:

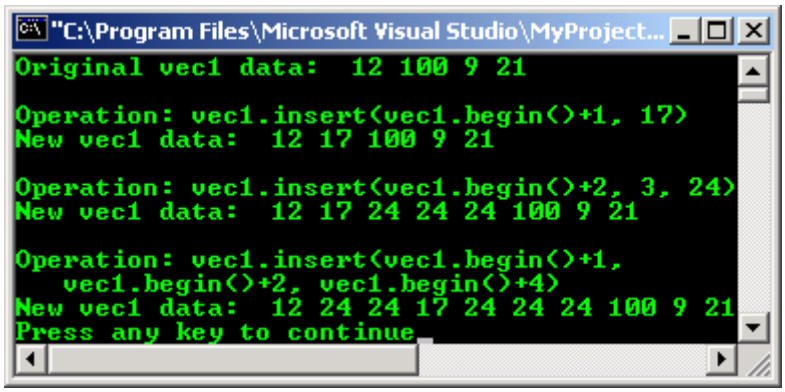


- The return value is the first `insert()` function returns an iterator that point to the position where the new element was inserted into the vector.

```
//vector, insert()
//begin(), end()
#include <vector>
#include <iostream>
using namespace std;

int main()
{
vector <int> vec1;
vector <int>::iterator Iter;
vec1.push_back(12);
vec1.push_back(100);
vec1.push_back(9);
vec1.push_back(21);

cout<<"Original vec1 data: ";
for(Iter = vec1.begin(); Iter != vec1.end(); Iter++)
cout<<" "<<*Iter;
cout<<endl;

cout<<"\nOperation: vec1.insert(vec1.begin()+1, 17)"<<endl;
vec1.insert(vec1.begin()+1, 17);
cout<<"New vec1 data: ";
for(Iter = vec1.begin(); Iter != vec1.end(); Iter++)
cout<<" "<<*Iter;
cout<<endl;

cout<<"\nOperation: vec1.insert(vec1.begin()+2, 3, 24)"<<endl;
vec1.insert(vec1.begin()+2, 3, 24);
cout<<"New vec1 data: ";
for(Iter = vec1.begin(); Iter != vec1.end(); Iter++)
cout<<" "<<*Iter;
cout<<endl;

cout<<"\nOperation: vec1.insert(vec1.begin()+1, \n"
"   vec1.begin()+2, vec1.begin()+4)"<<endl;
vec1.insert(vec1.begin()+1, vec1.begin()+2, vec1.begin()+4);
cout<<"New vec1 data: ";
for(Iter = vec1.begin(); Iter != vec1.end(); Iter++)
```

```
cout<<" "<<*Iter;
cout<<endl;
return 0;
}
```

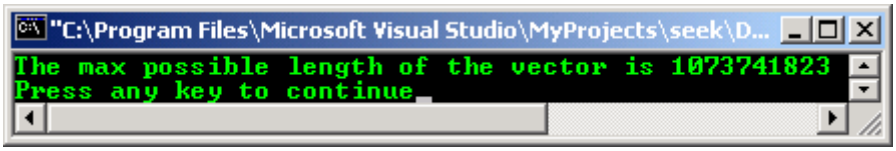
Output:



- The return value is the maximum possible length of the vector.

```
//vector, max_size()
#include <vector>
#include <iostream>
using namespace std;

int main()
{
vector <int> vec1;
vector <int>::size_type i;

i = vec1.max_size();
cout<<"The max possible length of the vector is "<<i<<endl;
return 0;
}
```

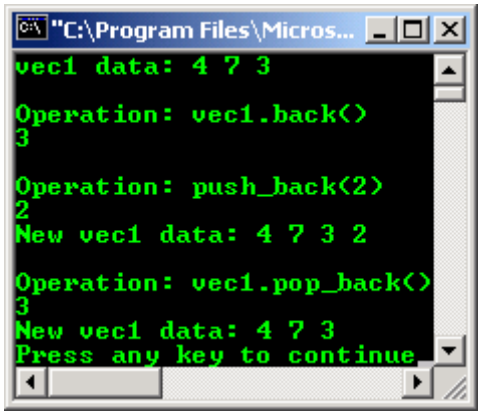
Output:




```
//vector, pop_back(), back()
//and push_back()
#include <vector>
#include <iostream>
using namespace std;

int main()
{
vector <int> vec1;

vec1.push_back(4);
vec1.push_back(7);
vec1.push_back(3);

cout<<"vec1 data: ";
for(int i=0; i<vec1.size(); ++i)
cout<<vec1[i]<<' ';
cout<<endl;

cout<<"\nOperation: vec1.back()\n";
cout<<vec1.back()<<endl;

cout<<"\nOperation: push_back(2)\n";
vec1.push_back(2);
cout<<vec1.back()<<endl;

cout<<"New vec1 data: ";
for(i=0; i<vec1.size(); ++i)
```

```
cout<<vec1[i]<<' ';
cout<<endl;

cout<<"\nOperation: vec1.pop_back()\n";
vec1.pop_back();
cout<<vec1.back()<<endl;

cout<<"New vec1 data: ";
for(i=0; i<vec1.size(); ++i)
cout<<vec1[i]<<' ';
cout<<endl;

return 0;
}
```

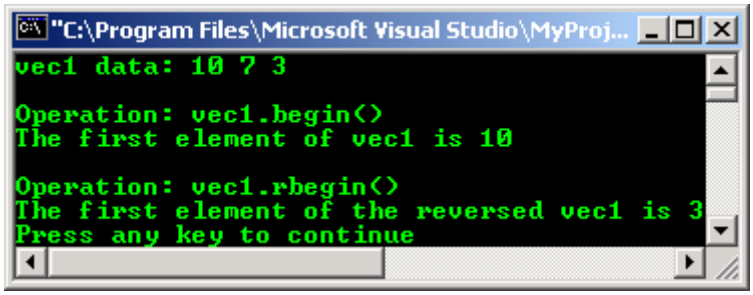
Output:



- The return value is a reverse random-access iterator addressing the first element in a reversed vector or addressing what had been the last element in the un reversed vector.
- If the return value of `rbegin()` is assigned to a `const_reverse_iterator`, the vector object cannot be modified. If the return value of `rbegin()` is assigned to a `reverse_iterator`, the vector object can be modified.

```
//vector, rbegin()
#include <vector>
#include <iostream>
using namespace std;

int main()
{
vector <int> vec1;
vector <int>::iterator vec1_Iter;
vector <int>::reverse_iterator vec1_rIter;

vec1.push_back(10);
vec1.push_back(7);
vec1.push_back(3);

cout<<"vec1 data: ";
for(int i=0; i<vec1.size(); ++i)
cout<<vec1[i]<<' ';
cout<<endl;

cout<<"\nOperation: vec1.begin()\n";
vec1_Iter = vec1.begin();
cout<<"The first element of vec1 is "<<*vec1_Iter<<endl;

cout<<"\nOperation: vec1.rbegin()\n";
vec1_rIter = vec1.rbegin();
cout<<"The first element of the reversed vec1 is "<<*vec1_rIter<<endl;
return 0;
}
```

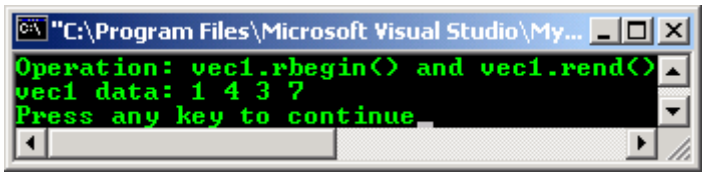
Output:



- The return value is a reverse random-access iterator that addresses the location succeeding the last element in a reversed vector (the location that had preceded the first element in the unreversed vector).
- `rend()` is used with a reversed vector just as `end()` is used with a vector.
- If the return value of `rend()` is assigned to a `const_reverse_iterator`, then the vector object cannot be modified. If the return value of `rend()` is assigned to a `reverse_iterator`, then the vector object can be modified.
- `rend()` can be used to test to whether a reverse iterator has reached the end of its vector.
- The value returned by `rend()` should not be dereferenced.

```
//vector, rend()
//and rbegin()
#include <vector>
#include <iostream>
using namespace std;

int main()
{
vector <int> vec1;
vector <int>::reverse_iterator vec1_rIter;

vec1.push_back(7);
vec1.push_back(3);
vec1.push_back(4);
vec1.push_back(1);

cout<<"Operation: vec1.rbegin() and vec1.rend()\n";
cout<<"vec1 data: ";
for(vec1_rIter = vec1.rbegin(); vec1_rIter != vec1.rend(); vec1_rIter++)
      cout<<*vec1_rIter<<' ';
cout<<endl;
return 0;
}
```

Output:

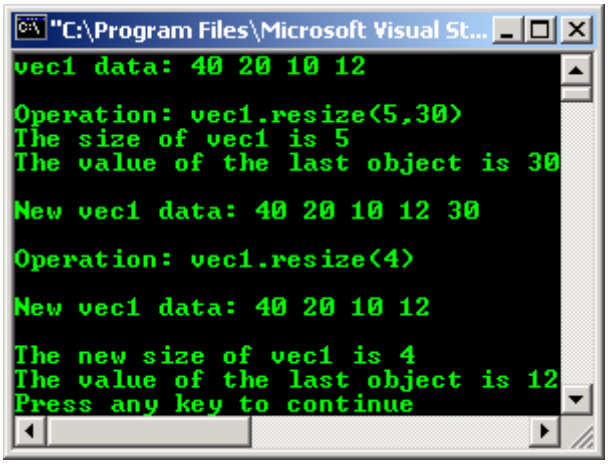


- If the container's size is less than the requested size, `_Newsize`, elements are added to the vector until it reaches the requested size.
- If the container's size is larger than the requested size, the elements closest to the end of the container are deleted until the container reaches the size `_Newsize`. If the present size of the container is the same as the requested size, no action is taken.
- `size()` reflects the current size of the vector.

```
//vector, resize()
//and size()
#include <vector>
#include <iostream>
using namespace std;

int main()
{
vector <int> vec1;
vec1.push_back(40);
vec1.push_back(20);
```

```
vec1.push_back(10);
vec1.push_back(12);

cout<<"vec1 data: ";
for(int i=0; i<vec1.size(); ++i)
cout<<vec1[i]<<' ';
cout<<endl;

//resize to 5 and add data at the end...
cout<<"\nOperation: vec1.resize(5,30)\n";
vec1.resize(5,30);
cout<<"The size of vec1 is "<<vec1.size()<<endl;
cout<<"The value of the last object is "<<vec1.back()<<endl;
cout<<"\nNew vec1 data: ";
for(i=0; i<vec1.size(); ++i)
cout<<vec1[i]<<' ';
cout<<endl;

cout<<"\nOperation: vec1.resize(4)\n";
vec1.resize(4);

cout<<"\nNew vec1 data: ";
for(i=0; i<vec1.size(); ++i)
cout<<vec1[i]<<' ';
cout<<endl;

cout<<"\nThe new size of vec1 is "<<vec1.size()<<endl;
cout<<"The value of the last object is "<<vec1.back()<<endl;
return 0;
}
```

Output:

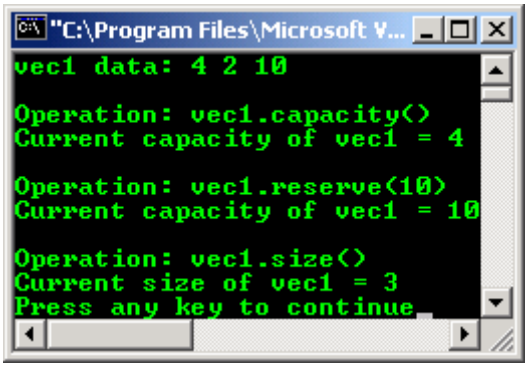


- The return value is the current length of the vector.

```
//vector, reserve()
//capacity() and size()
#include <vector>
#include <iostream>
using namespace std;

int main()
{
vector <int> vec1;

vec1.push_back(4);
vec1.push_back(2);
vec1.push_back(10);

cout<<"vec1 data: ";
for(int i=0; i<vec1.size(); ++i)
cout<<vec1[i]<<' ';
cout<<endl;

cout<<"\nOperation: vec1.capacity()"<<endl;
cout<<"Current capacity of vec1 = "<<vec1.capacity()<<endl;
cout<<"\nOperation: vec1.reserve(10)"<<endl;
vec1.reserve(10);
```

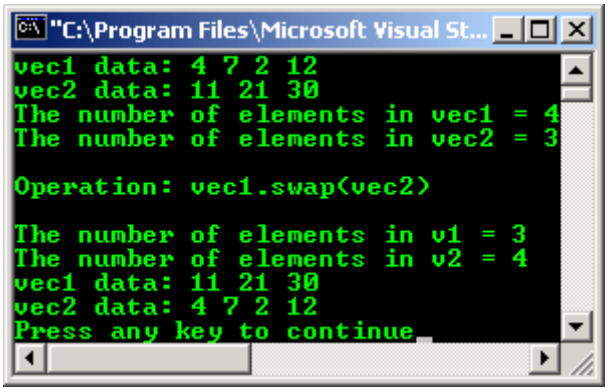
```
cout<<"Current capacity of vec1 = "<<vec1.capacity()<<endl;
cout<<"\nOperation: vec1.size()"<<endl;
cout<<"Current size of vec1 = "<<vec1.size()<<endl;
return 0;
}
```

Output:




```
//vector, swap()
#include <vector>
#include <iostream>
using namespace std;

int main()
{
vector <int> vec1, vec2;

vec1.push_back(4);
vec1.push_back(7);
vec1.push_back(2);
vec1.push_back(12);

cout<<"vec1 data: ";
for(int i=0; i<vec1.size(); ++i)
cout<<vec1[i]<<' ';
cout<<endl;

vec2.push_back(11);
vec2.push_back(21);
vec2.push_back(30);

cout<<"vec2 data: ";
for(i=0; i<vec2.size(); ++i)
cout<<vec2[i]<<' ';
cout<<endl;

cout<<"The number of elements in vec1 = "<<vec1.size()<<endl;
cout<<"The number of elements in vec2 = "<<vec2.size()<<endl;
cout<<endl;

cout<<"Operation: vec1.swap(vec2)\n"<<endl;
vec1.swap(vec2);
cout<<"The number of elements in v1 = "<<vec1.size()<<endl;
cout<<"The number of elements in v2 = "<<vec2.size()<<endl;

cout<<"vec1 data: ";
for(i=0; i<vec1.size(); ++i)
cout<<vec1[i]<<' ';
cout<<endl;

cout<<"vec2 data: ";
for(i=0; i<vec2.size(); ++i)
cout<<vec2[i]<<' ';
cout<<endl;
return 0;
}
```

Output:



vector Class Template Operator

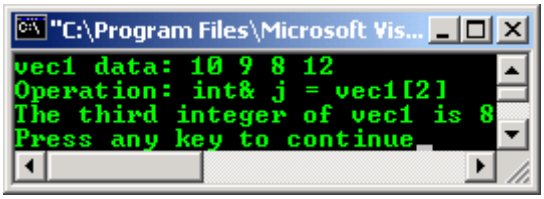
Operator	Description
`operator[]`	Returns a reference to the vector element at a specified position.

Table 27.6

- The return value is, if the position specified is greater than the size of the container, the result is undefined.
- If the return value of `operator[]` is assigned to a `const_reference`, the vector object cannot be modified. If the return value of `operator[]` is assigned to a `reference`, the vector object can be modified.

```
//vector operator[]
#include <vector>
#include <iostream>
using namespace std;

int main()
{
vector <int> vec1;

vec1.push_back(10);
vec1.push_back(9);
vec1.push_back(8);
vec1.push_back(12);

cout<<"vec1 data: ";
for(int i=0; i<vec1.size(); ++i)
cout<<vec1[i]<<' ';
cout<<endl;

cout<<"Operation: int& j = vec1[2]\n";
int& j = vec1[2];
cout<<"The third integer of vec1 is "<<j<<endl;
return 0;
}
```

Output:



vector Class Template Specializations

Specialization	Description
`vector<bool> Class`	A full specialization of the template class vector for elements of type `bool` with an allocator for the underlying type used by the specialization.

Table 27.7

vector<bool> Class

vector<bool> Class Members

vector<bool> Typedefs

Typedef	Description
`const_iterator`	A type that describes an object that can serve as a constant random-access iterator for the sequence of Boolean elements contained by the vector.
`const_pointer`	A type that describes an object that can serve as a constant pointer to a Boolean element of the sequence contained by the vector.
`const_reference`	A type that describes an object that can serve as a constant reference to a Boolean element of the sequence contained by the vector.
`iterator`	A type that describes an object that can serve as a random-access iterator for a sequence of Boolean elements contained by a vector.
`pointer`	A type that describes an object that can serve as a constant pointer to a Boolean element of the sequence contained by the vector.

Table 27.8

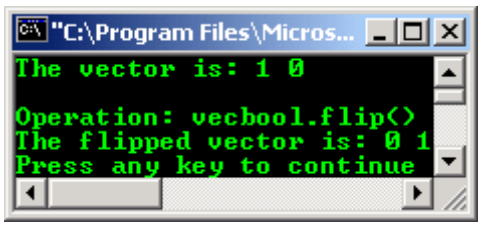
vector<bool> Member Functions

Member function	Description
`flip()`	Reverses all bits in the vector.
`swap()`	Exchanges the elements of two vectors with Boolean elements.

Table 27.9

```
//vector_bool, flip()
#include <vector>
#include <iostream>
using namespace std;

int main()
{
_Bvector vecbool;

vecbool.push_back(1);
vecbool.push_back(0);

cout<<"The vector is: "<<vecbool.front()<<" "<<vecbool.back()<<endl;
cout<<"\nOperation: vecbool.flip()\n";
vecbool.flip();
cout<<"The flipped vector is: "<<vecbool.front()<<" "<<vecbool.back()<<endl;
return 0;
}
```

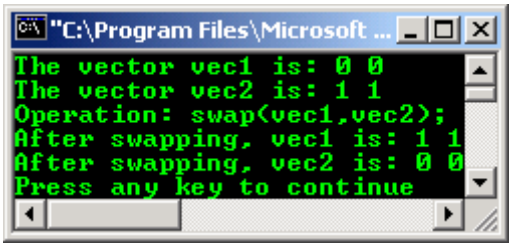
Output:




```
//vector_bool, swap()
#include <vector>
#include <iostream>
using namespace std;

int main()
{
_Bvector vec1, vec2;
```

```
vec1.push_back(0);
vec1.push_back(0);
vec2.push_back(1);
vec2.push_back(1);

cout<<"The vector vec1 is: "<<vec1.front()<<" "<<vec1.back()<<endl;
cout<<"The vector vec2 is: "<<vec2.front()<<" "<<vec2.back()<<endl;

cout<<"Operation: swap(vec1, vec2);\n";
swap(vec1, vec2);
cout<<"After swapping, vec1 is: "<<vec1.front()<<" "<<vec1.back()<<endl;
cout<<"After swapping, vec2 is: "<<vec2.front()<<" "<<vec2.back()<<endl;
return 0;
}
```

Output:



Nested Classes

Nested class	Description
`vector<bool> reference Class`	A nested class whose objects are able to provide references to elements (single bits) within a `vector<bool>` object.

Table 27.10

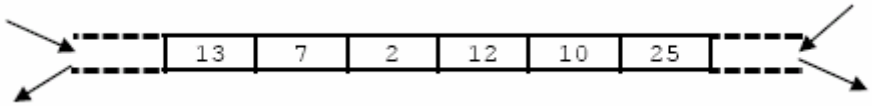
----------End of vector----------
---www.tenouk.com---

Further reading and digging:

1. Check the best selling C / C++ and STL books at Amazon.com.

27.5.2 deque

- The term **deque** (pronounced “deck”) is an abbreviation for ‘double-ended queue’. It is a dynamic array that is implemented so that it can grow in both directions.
- So, inserting elements at the end and at the beginning is fast. However, inserting elements in the middle takes time because elements must be moved. Deque structure can be depicted as follow:



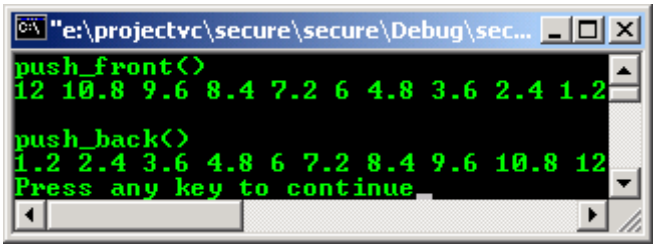
- Deque reallocation occurs when a member function must insert or erase elements of the sequence:
  - If an element is inserted into an empty sequence, or if an element is erased to leave an empty sequence, then iterators earlier returned by `begin()` and `end()` become invalid.
  - If an element is inserted at the first position of the deque, then all iterators, but no references, that designate existing elements become invalid.
  - If an element is inserted at the end of the deque, then `end()` and all iterators, but no references, that designate existing elements become invalid.

- If an element is erased at the front of the deque, only that iterator and references to the erased element become invalid.
- If the last element is erased from the end of the deque, only that iterator to the final element and references to the erased element become invalid.

- Otherwise, inserting or erasing an element invalidates all iterators and references.
- The following general deque example declares a deque for floating-point values, inserts elements from 1.2 to 12 at the front of the container, and prints all elements of the deque:

```
//simple deque example
#include <iostream>
#include <deque>
using namespace std;

int main()
{
//deque container for floating-point elements
//declaration
deque<float> elem, elem1;

//insert the elements each at the front
cout<<"push_front()\n";
for(int i=1; i<=10; ++i)
//insert at the front
elem.push_front(i*(1.2));

//print all elements separated by a space
for(i=0; i<elem.size(); ++i)
cout<<elem[i]<<' ';
cout<<endl;

//insert the elements each at the back
cout<<"\npush_back()\n";
//insert at the back
for(i=1; i<=10; ++i)
elem1.push_back(i*(1.2));

//print all elements separated by a space
for(i=0; i<elem1.size(); ++i)
cout<<elem1[i]<<' ';
cout<<endl;

return 0;
}
```

Output:



<deque> Header Members

Operators

Operator	Description
operator!=	Tests if the deque object on the left side of the operator is not equal to the deque object on the right side.
operator<	Tests if the deque object on the left side of the operator is less than the deque object on the right side.
operator<=	Tests if the deque object on the left side of the operator is less than or equal to the deque object on the right side.
operator==	Tests if the deque object on the left side of the operator is equal to the deque object on the right side.
operator>	Tests if the deque object on the left side of the operator is greater than the deque object on the right side.

`operator>=`	Tests if the deque object on the left side of the operator is greater than or equal to the deque object on the right side.

Table 27.11

deque Template Class

Class	Description
`deque Class`	A template class of sequence containers that arrange elements of a given type in a linear arrangement and, like vectors, allow fast random access to any element and efficient insertion and deletion at the back of the container.

Table 27.12

- The STL sequence container deque arranges elements of a given type in a linear arrangement and, like vectors, allow fast random access to any element and efficient insertion and deletion at the back of the container.
- However, unlike a vector, the deque class also supports efficient insertion and deletion at the front of the container.

deque Template Class Members

Typedefs

Typedef	Description
`allocator_type`	A type that represents the allocator class for the deque object.
`const_iterator`	A type that provides a random-access iterator that can access and read a `const` element in the deque.
`const_pointer`	A type that provides a pointer to a `const` element in a deque.
`const_reference`	A type that provides a reference to a `const` element stored in a deque for reading and performing `const` operations.
`const_reverse_iterator`	A type that provides a random-access iterator that can read any `const` element in the deque.
`difference_type`	A type that provides the difference between two iterators that refer to elements within the same deque.
`iterator`	A type that provides a random-access iterator that can read or modify any element in a deque.
`pointer`	A type that provides a pointer to an element in a deque.
`reference`	A type that provides a reference to an element stored in a deque.
`reverse_iterator`	A type that provides a random-access iterator that can read or modify an element in a reversed deque.
`size_type`	A type that counts the number of elements in a deque.
`value_type`	A type that represents the data type stored in a deque.

Table 27.13

deque Template Class Member Functions

Member function	Description
`assign()`	Erases elements from a deque and copies a new set of elements to the target deque.
`at()`	Returns a reference to the element at a specified location in the deque.
`back()`	Returns a reference to the last element of the deque.
`begin()`	Returns an iterator addressing the first element in the deque.
`clear()`	Erases all the elements of a deque.
`deque()`	deque constructor, constructs a deque of a specific size or with elements of a specific value or with a specific allocator or as a copy of all or part of some other deque.
`empty()`	Tests if a deque is empty.
`end()`	Returns an iterator that addresses the location succeeding the last element in a deque.
`erase()`	Removes an element or a range of elements in a deque from specified positions.
`front()`	Returns a reference to the first element in a deque.
`get_allocator()`	Returns a copy of the allocator object used to construct the deque.

`insert()`	Inserts an element or a number of elements or a range of elements into the deque at a specified position.
`max_size()`	Returns the maximum length of the deque.
`pop_back()`	Deletes the element at the end of the deque.
`pop_front()`	Deletes the element at the beginning of the deque.
`push_back()`	Adds an element to the end of the deque.
`push_front()`	Adds an element to the beginning of the deque.
`rbegin()`	Returns an iterator to the first element in a reversed deque.
`rend()`	Returns an iterator that point just beyond the last element in a reversed deque.
`resize()`	Specifies a new size for a deque.
`size()`	Returns the number of elements in the deque.
`swap()`	Exchanges the elements of two deques.

Table 27.14

deque Template Class Operator

Operator	Description
`operator[]`	Returns a reference to the deque element at a specified position.

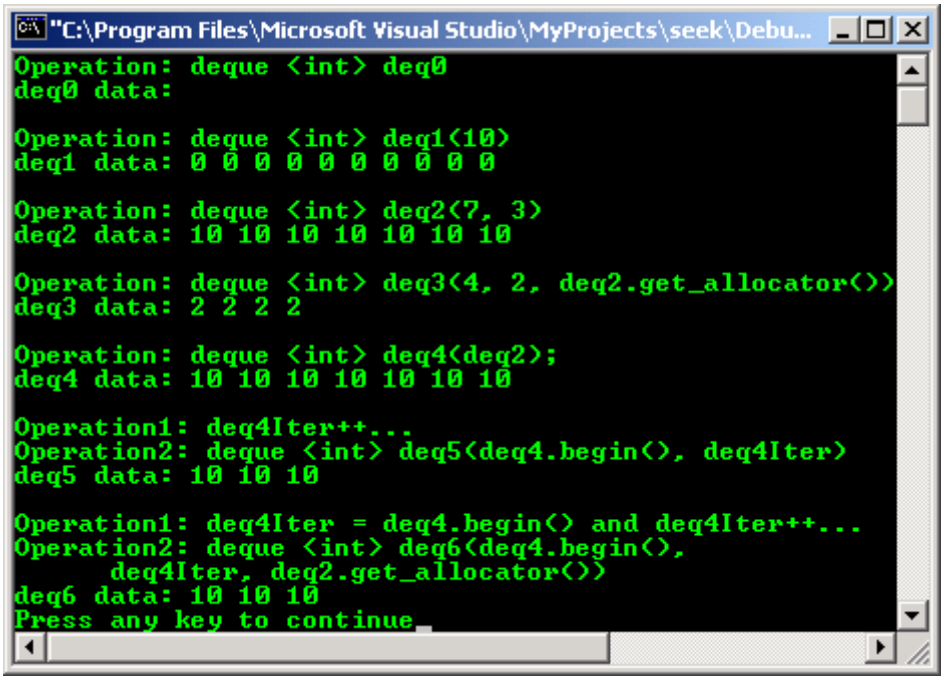
Table 27.15

- deque constructor, constructs a deque of a specific size or with elements of a specific value or with a specific allocator or as a copy of all or part of some other deque.
- All constructors store an allocator object and initialize the deque.
- None of the constructors perform any interim reallocations.

```
//deque, constructors
#include <deque>
#include <iostream>
using namespace std;

int main()
{
deque <int>::iterator deq0Iter, deq1Iter, deq2Iter, deq3Iter, deq4Iter, deq5Iter,
deq6Iter;

//Create an empty deque deq0
deque <int> deq0;

//Create a deque deq1 with 10 elements of default value 0
deque <int> deq1(10);

//Create a deque deq2 with 7 elements of value 10
deque <int> deq2(7, 10);

//Create a deque deq3 with 4 elements of value 2 and with the
//allocator of deque deq2
deque <int> deq3(4, 2, deq2.get_allocator());

//Create a copy, deque deq4, of deque deq2
deque <int> deq4(deq2);

//deque deq5 a copy of the deq4[_First, _Last) range
deq4Iter = deq4.begin();
deq4Iter++;
deq4Iter++;
deq4Iter++;
deque <int> deq5(deq4.begin(), deq4Iter);

//Create a deque deq6 by copying the range deq4[_First, _Last) and
//the allocator of deque deq2
deq4Iter = deq4.begin();
deq4Iter++;
deq4Iter++;
deq4Iter++;
deque <int> deq6(deq4.begin(), deq4Iter, deq2.get_allocator());

//------------------------------------
cout<<"Operation: deque <int> deq0\n";
cout<<"deq0 data: ";
for(deq0Iter = deq0.begin(); deq0Iter != deq0.end(); deq0Iter++)
cout<<*deq0Iter<<" ";
```

```
cout<<endl;

cout<<"\nOperation: deque <int> deq1(10)\n";
cout<<"deq1 data: ";
for(deq1Iter = deq1.begin(); deq1Iter != deq1.end(); deq1Iter++)
cout<<*deq1Iter<<" ";
cout<<endl;

cout<<"\nOperation: deque <int> deq2(7, 3)\n";
cout<<"deq2 data: ";
for(deq2Iter = deq2.begin(); deq2Iter != deq2.end(); deq2Iter++)
cout<<*deq2Iter<<" ";
cout<<endl;

cout<<"\nOperation: deque <int> deq3(4, 2, deq2.get_allocator())\n";
cout<<"deq3 data: ";
for(deq3Iter = deq3.begin(); deq3Iter != deq3.end(); deq3Iter++)
cout<<*deq3Iter<<" ";
cout<<endl;

cout<<"\nOperation: deque <int> deq4(deq2);\n";
cout<<"deq4 data: ";
for(deq4Iter = deq4.begin(); deq4Iter != deq4.end(); deq4Iter++)
cout<<*deq4Iter<<" ";
cout<<endl;

cout<<"\nOperation1: deq4Iter++...\n";
cout<<"Operation2: deque <int> deq5(deq4.begin(), deq4Iter)\n";
cout<<"deq5 data: ";
for(deq5Iter = deq5.begin(); deq5Iter != deq5.end(); deq5Iter++)
cout << *deq5Iter<<" ";
cout << endl;

cout<<"\nOperation1: deq4Iter = deq4.begin() and deq4Iter++...\n";
cout<<"Operation2: deque <int> deq6(deq4.begin(), \n"
"      deq4Iter, deq2.get_allocator())\n";
cout<<"deq6 data: ";
for(deq6Iter = deq6.begin(); deq6Iter != deq6.end(); deq6Iter++)
cout<<*deq6Iter<<" ";
cout<<endl;
return 0;
}
```

Output:



- The following are program examples compiled using **g++**. Well, it seems that compiling STL programs using g++ is smoother because if you use an old constructs, that is not based on the standard, in your program, g++ will prompt you!

```
//*******vector.cp*********
//vector constructors
#include <vector>
#include <iostream>
using namespace std;

int main()
{
vector <int>::iterator vec0Iter, vec1Iter, vec2Iter, vec3Iter, vec4Iter, vec5Iter;

//Create an empty vector vec0
vector <int> vec0;

//Create a vector vec1 with 10 elements of default value 0
vector <int> vec1(10);

//Create a vector vec2 with 7 elements of value 13
vector <int> vec2(7, 13);

//Create a vector vec3 with 5 elements of value 3 and with the allocator
//of vector vec2
vector <int> vec3(5, 3, vec2.get_allocator());

//vector vec4, a copy of vector vec2
vector <int> vec4(vec2);

//Create a vector vec5 by copying the range of vec4[_First, _Last)
vector <int> vec5(vec4.begin() + 1, vec4.begin() + 3);

cout<<"Operation: vector <int> vec0\n";
cout<<"vec0 data: ";
for(vec0Iter = vec0.begin(); vec0Iter != vec0.end(); vec0Iter++)
cout<<" "<<*vec0Iter;
cout<<endl;

cout<<"\nOperation: vector <int> vec1(10)\n";
cout<<"vec1 data: ";
for(vec1Iter = vec1.begin(); vec1Iter != vec1.end(); vec1Iter++)
cout<<" "<<*vec1Iter;
cout<<endl;

cout<<"\nOperation: vector <int> vec2(7, 13)\n";
cout<<"vec2 data: ";
for(vec2Iter = vec2.begin(); vec2Iter != vec2.end(); vec2Iter++)
cout<<" "<<*vec2Iter;
cout<<endl;

cout<<"\nOperation: vector <int> vec3(5, 3, vec2.get_allocator())\n";
cout<<"vec3 data: ";
for(vec3Iter = vec3.begin(); vec3Iter != vec3.end(); vec3Iter++)
cout<<" "<<*vec3Iter;
cout<<endl;

cout<<"\nOperation: vector <int> vec4(vec2)\n";
cout<<"vec4 data: ";
for(vec4Iter = vec4.begin(); vec4Iter != vec4.end(); vec4Iter++)
cout<<" "<<*vec4Iter;
cout<<endl;

cout<<"\nOperation: vector <int> vec5(vec4.begin()+1, vec4.begin()+3)\n";
cout<<"vec5 data: ";
for(vec5Iter = vec5.begin(); vec5Iter != vec5.end(); vec5Iter++)
cout<<" "<<*vec5Iter;
cout<<endl;
return 0;
}
```

[bodo@bakawali ~]$ g++ vector.cpp -o vector
[bodo@bakawali ~]$ ./vector

```
Operation: vector <int> vec0
vec0 data:

Operation: vector <int> vec1(10)
vec1 data:  0 0 0 0 0 0 0 0 0 0

Operation: vector <int> vec2(7, 13)
vec2 data:  13 13 13 13 13 13 13
```

```
Operation: vector <int> vec3(5, 3, vec2.get_allocator())
vec3 data:  3 3 3 3 3

Operation: vector <int> vec4(vec2)
vec4 data:  13 13 13 13 13 13 13

Operation: vector <int> vec5(vec4.begin()+1, vec4.begin()+3)
vec5 data:  13 13
```

```
//********deque.cpp*********
//deque, constructors
#include <deque>
#include <iostream>
using namespace std;

int main()
{
deque <int>::iterator deq0Iter, deq1Iter, deq2Iter, deq3Iter, deq4Iter, deq5Iter,
deq6Iter;

//Create an empty deque deq0
deque <int> deq0;

//Create a deque deq1 with 10 elements of default value 0
deque <int> deq1(10);

//Create a deque deq2 with 7 elements of value 10
deque <int> deq2(7, 10);

//Create a deque deq3 with 4 elements of value 2 and with the
//allocator of deque deq2
deque <int> deq3(4, 2, deq2.get_allocator());

//Create a copy, deque deq4, of deque deq2
deque <int> deq4(deq2);

//deque deq5 a copy of the deq4[_First, _Last) range
deq4Iter = deq4.begin();
deq4Iter++;
deq4Iter++;
deq4Iter++;
deque <int> deq5(deq4.begin(), deq4Iter);

//Create a deque deq6 by copying the range deq4[_First, _Last) and
//the allocator of deque deq2
deq4Iter = deq4.begin();
deq4Iter++;
deq4Iter++;
deq4Iter++;
deque <int> deq6(deq4.begin(), deq4Iter, deq2.get_allocator());

//------------------------------------
cout<<"Operation: deque <int> deq0\n";
cout<<"deq0 data: ";
for(deq0Iter = deq0.begin(); deq0Iter != deq0.end(); deq0Iter++)
cout<<*deq0Iter<<" ";
cout<<endl;

cout<<"\nOperation: deque <int> deq1(10)\n";
cout<<"deq1 data: ";
for(deq1Iter = deq1.begin(); deq1Iter != deq1.end(); deq1Iter++)
cout<<*deq1Iter<<" ";
cout<<endl;

cout<<"\nOperation: deque <int> deq2(7, 3)\n";
cout<<"deq2 data: ";
for(deq2Iter = deq2.begin(); deq2Iter != deq2.end(); deq2Iter++)
cout<<*deq2Iter<<" ";
cout<<endl;

cout<<"\nOperation: deque <int> deq3(4, 2, deq2.get_allocator())\n";
cout<<"deq3 data: ";
for(deq3Iter = deq3.begin(); deq3Iter != deq3.end(); deq3Iter++)
cout<<*deq3Iter<<" ";
cout<<endl;

cout<<"\nOperation: deque <int> deq4(deq2);\n";
cout<<"deq4 data: ";
for(deq4Iter = deq4.begin(); deq4Iter != deq4.end(); deq4Iter++)
cout<<*deq4Iter<<" ";
```

```
cout<<endl;

cout<<"\nOperation1: deq4Iter++...\n";
cout<<"Operation2: deque <int> deq5(deq4.begin(), deq4Iter)\n";
cout<<"deq5 data: ";
for(deq5Iter = deq5.begin(); deq5Iter != deq5.end(); deq5Iter++)
cout << *deq5Iter<<" ";
cout << endl;

cout<<"\nOperation1: deq4Iter = deq4.begin() and deq4Iter++...\n";
cout<<"Operation2: deque <int> deq6(deq4.begin(), \n"
"      deq4Iter, deq2.get_allocator())\n";
cout<<"deq6 data: ";
for(deq6Iter = deq6.begin(); deq6Iter != deq6.end(); deq6Iter++)
cout<<*deq6Iter<<" ";
cout<<endl;
return 0;
}
```

[bodo@bakawali ~]$ g++ deque.cpp -o deque
[bodo@bakawali ~]$ ./deque

```
Operation: deque <int> deq0
deq0 data:

Operation: deque <int> deq1(10)
deq1 data: 0 0 0 0 0 0 0 0 0 0

Operation: deque <int> deq2(7, 3)
deq2 data: 10 10 10 10 10 10 10

Operation: deque <int> deq3(4, 2, deq2.get_allocator())
deq3 data: 2 2 2 2

Operation: deque <int> deq4(deq2);
deq4 data: 10 10 10 10 10 10 10

Operation1: deq4Iter++...
Operation2: deque <int> deq5(deq4.begin(), deq4Iter)
deq5 data: 10 10 10

Operation1: deq4Iter = deq4.begin() and deq4Iter++...
Operation2: deque <int> deq6(deq4.begin(),
      deq4Iter, deq2.get_allocator())
deq6 data: 10 10 10
```

----------End of deque----------
---www.tenouk.com---

Further reading and digging:

1. Check the best selling C / C++ and STL books at Amazon.com.