MODULE 30
--THE STL--
CONTAINER ADAPTOR

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

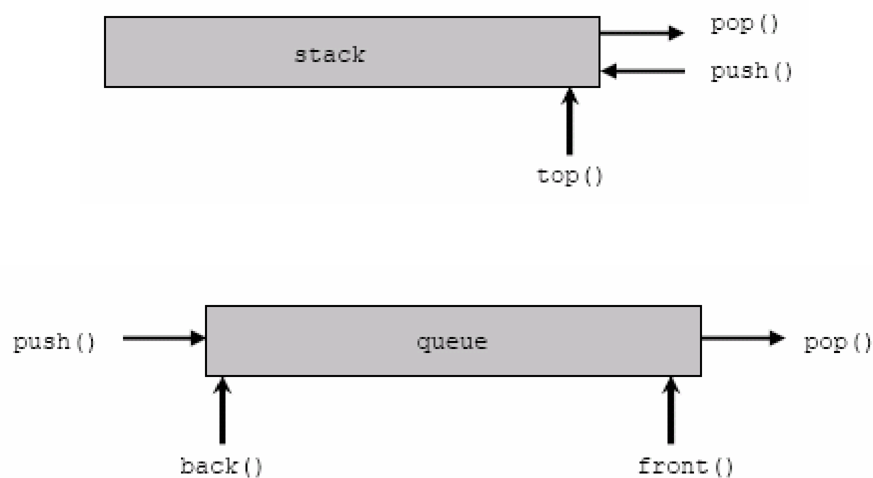
- Able to understand and use container adapters.
- Able to understand and use container adapter `stack`.
- Able to understand and use container adapter `queue`.
- Able to understand and use container adapter priority queue.

30.1 Container Adapters

- In addition to the fundamental container classes, the C++ standard library provides special predefined container adapters that meet special needs. These are implemented by using the fundamental containers classes.
- The predefined container adapters are as follows:

Container Adapter	Description
`Stacks`	Is a container that manages its elements by the LIFO (last-in-first-out) policy.
`Queues`	Is a container that manages its elements by the FIFO (first-in-first-out) policy. That is, it is an ordinary buffer.
`Priority Queues`	Is a container in which the elements may have different priorities. The priority is based on a sorting criterion that the programmer may provide (by default, operator < is used). A priority queue is, in effect, a buffer in which the next element is always the element that has the highest priority inside the queue. If more than one element has the highest priority, the order of these elements is undefined.

Table 30.1



- Container adapters are just special containers that use the general framework of the containers, iterators, and algorithms provided by the STL.

30.2 <stack> Members

Operators

Operator	Description
`operator!=`	Tests if the stack object on the left side of the operator is not equal to the stack object

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

Table 30.2

Classes

Class	Description
`stack Class`	A template container adaptor class that provides a restriction of functionality limiting access to the element most recently added to some underlying container type.

Table 30.3

stack Members

Typedefs

Typedef	Description
`container_type`	A type that provides the base container to be adapted by a stack.
`size_type`	An unsigned integer type that can represent the number of elements in a stack.
`value_type`	A type that represents the type of object stored as an element in a stack.

Table 30.4

Member Functions

Member function	Description
`empty()`	Tests if the stack is empty.
`pop()`	Removes the element from the top of the stack.
`push()`	Adds an element to the top of the stack.
`size()`	Returns the number of elements in the stack.
`stack()`	Constructs a stack that is empty or that is a copy of a base container object.
`top()`	Returns a reference to an element at the top of the stack.

Table 30.5

- The stack must be nonempty to apply the member function. The top of the stack is the position occupied by the most recently added element and is the last element at the end of the container.
- The top of the stack is the position occupied by the most recently added element and is the last element at the end of the container.

```
//stack, pop(), push()
//size() and top()
#include <stack>
#include <iostream>
using namespace std;

int main()
{
stack <int> st1, st2;

//push data element on the stack
st1.push(21);
int j=st1.top();
```

```
cout<<j<<' ';
st1.push(9);
j=st1.top();
cout<<j<<' ';
st1.push(12);
j=st1.top();
cout<<j<<' ';
st1.push(31);
j=st1.top();
cout<<j<<' '<<endl;

stack <int>::size_type i;
i = st1.size();
cout<<"The stack length is "<<i<<endl;

i = st1.top();
cout<<"The element at the top of the stack is "<<i<<endl;

st1.pop();

i = st1.size();
cout<<"After a pop, the stack length is "<<i<<endl;

i = st1.top();
cout<<"After a pop, the element at the top of the stack is "<<i<<endl;
return 0;
}
```

Output:

```
Stack st1 data: 21 9 12 31
The stack length is 4
The element at the top of the stack is 31
After a pop, the stack length is 3
After a pop, the element at the top of the stack is 12
Press any key to continue
```

stack Constructor

- Constructs a stack that is empty or that is a copy of a base container class.

```
stack( );
explicit stack(
   const container_type& _Right
);
```

Parameter

Parameter	Description
_Right	The container of which the constructed stack is to be a copy.

Table 30.6

```
//stack, constructor
#include <stack>
#include <vector>
#include <list>
#include <iostream>
using namespace std;

int main()
{
//Declares stack with default deque base container
stack <char> deq1;

//Explicitly declares a stack with deque base container
stack <char, deque<char> > deq2;

//Declares a stack with vector base containers
stack <int, vector<int> > vec;

//Declares a stack with list base container
```

```
stack <int, list<int> > lst;
cout<<endl;

   return 0;
}
//no output
```

stack Class

- A template container adaptor class that provides a restriction of functionality limiting access to the element most recently added to some underlying container type.
- The stack class is used when it is important to be clear that only stack operations are being performed on the container.

```
template <
   class Type,
   class Container = deque<Type>
>
```

Parameters

Parameter	Description
`Type`	The element data type to be stored in the stack.
`Container`	The type of the underlying container used to implement the stack. The default value is the class `deque<Type>`.

Table 30.7

- The elements of class `Type` stipulated in the first template parameter of a stack object are synonymous with value `_type` and must match the type of element in the underlying container class `Container` stipulated by the second template parameter.
- The `Type` must be assignable, so that it is possible to copy objects of that type and to assign values to variables of that type.
- Suitable underlying container classes for stack include deque, list, and vector, or any other sequence container that supports the operations of `back()`, `push_back()`, and `pop_back()`.
- The underlying container class is encapsulated within the container adaptor, which exposes only the limited set of the sequence container member functions as a public interface.
- The stack objects are equality comparable if and only if the elements of class `Type` are equality comparable and are less-than comparable if and only if the elements of class `Type` are less-than comparable.

  - The **stack** class supports a last-in, first-out (LIFO) data structure. A good analogy would be a stack of plates. Elements (plates) may be inserted, inspected, or removed only from the top of the stack, which is the last element at the end of the base container. The restriction to accessing only the top element is the reason for using the stack class.
  - The **queue** class supports a first-in, first-out (FIFO) data structure. A good analogy would be people lining up for a bank teller. Elements (people) may be added to the back of the line and are removed from the front of the line. Both the front and the back of a line may be inspected. The restriction to accessing only the front and back elements in this way is the reason fur using the queue class.
  - The **priority_queue** class orders its elements so that the largest element is always at the top position. It supports insertion of an element and the inspection and removal of the top element. A good analogy would be people lining up where they are arranged by age, height, or some other criterion.

30.3 queue Members

Typedefs

Typedef	Description
`container_type`	A type that provides the base container to be adapted by the queue.
`size_type`	An unsigned integer type that can represent the number of elements in a queue.

`value_type`	A type that represents the type of object stored as an element in a queue.

Table 30.8

Member Functions

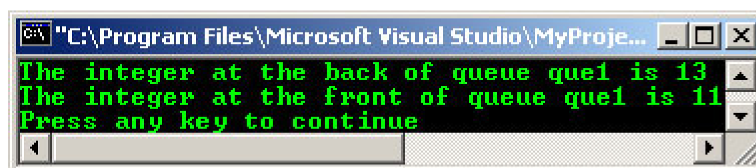
Member function	Description
`Back()`	Returns a reference to the last and most recently added element at the back of the queue.
`empty()`	Tests if the queue is empty.
`front()`	Returns a reference to the first element at the front of the queue.
`pop()`	Removes an element from the front of the queue.
`push()`	Adds an element to the back of the queue.
`queue()`	Constructs a queue that is empty or that is a copy of a base container object.
`size()`	Returns the number of elements in the queue.

Table 30.9

- The return value is the last element of the queue. If the queue is empty, the return value is undefined.
- If the return value of `back()` is assigned to a `const_reference`, the queue object cannot be modified. If the return value of `back()` is assigned to a `reference`, the queue object can be modified.

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

int main()
{
queue <int> que1;

que1.push(11);
que1.push(13);

int& x = que1.back();
const int& y = que1.front();

cout<<"The integer at the back of queue que1 is "<<x<<endl;
cout<<"The integer at the front of queue que1 is "<<y<<endl;
return 0;
}
```

Output:

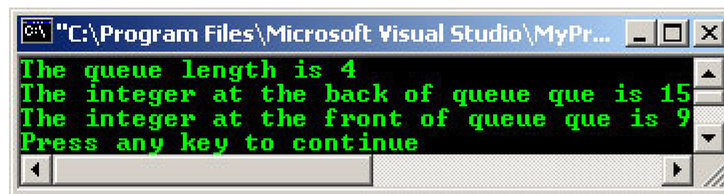


- The return value is the last element of the queue. If the queue is empty, the return value is undefined.
- If the return value of `front()` is assigned to a `const_reference`, the queue object cannot be modified. If the return value of `front()` is assigned to a `reference`, the queue object can be modified.
- The member function returns a `reference` to the first element of the controlled sequence, which must be nonempty.

```
//queue, front()
#include <queue>
#include <iostream>
using namespace std;

int main()
{
queue <int> que;
```

```
que.push(9);
que.push(12);
que.push(20);
que.push(15);

queue <int>::size_type x;
x = que.size();
cout<<"The queue length is "<<x<<endl;

int& y = que.back();
int& z = que.front();
cout<<"The integer at the back of queue que is "<<y<<endl;
cout<<"The integer at the front of queue que is "<<z<<endl;
return 0;
}
```

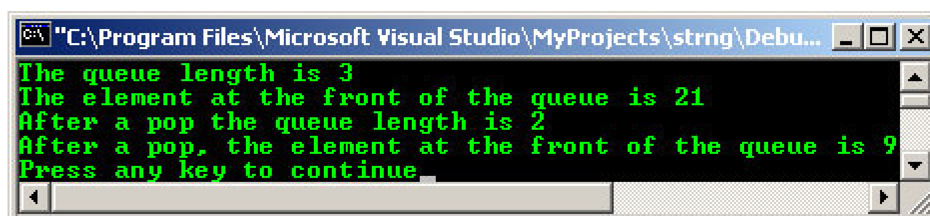
Output:



- The queue must be nonempty to apply the member function. The top of the queue is the position occupied by the most recently added element and is the last element at the end of the container.

```
//queue, pop()
#include <queue>
#include <iostream>
using namespace std;

int main()
{
queue <int> que;

que.push(21);
que.push(9);
que.push(13);

queue <int>::size_type i;
i = que.size();
cout<<"The queue length is "<<i<<endl;

i = que.front();
cout<<"The element at the front of the queue is "<<i<<endl;

que.pop();
i = que.size();
cout<<"After a pop the queue length is "<<i<<endl;

i = que.front();
cout<<"After a pop, the element at the front of the queue is "<<i<<endl;
return 0;
}
```

Output:

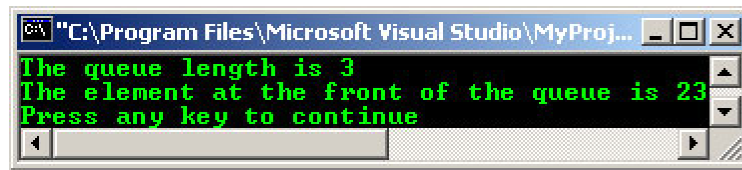


- The top of the queue is the position occupied by the most recently added element and is the last element at the end of the container.

```
//queue, push()
```

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

int main()
{
queue <int> que;

que.push(23);
que.push(15);
que.push(32);

queue <int>::size_type i;
i = que.size();
cout<<"The queue length is "<<i<<endl;
i = que.front();
cout<<"The element at the front of the queue is "<<i<<endl;
return 0;
}
```

Output:



queue Constructor

- Constructs a queue that is empty or that is a copy of a base container object.

```
queue( );
explicit queue(
   const container_type& _Right
);
```

Parameter

Parameter	Description
`_Right`	The `const` container of which the constructed queue is to be a copy.

Table 30.10

- The default base container for queue is deque. You can also specify list as a base container, but you cannot specify vector, because it lacks the required `pop_front()` member function.

```
//queue, constructor
#include <queue>
#include <vector>
#include <list>
#include <iostream>
using namespace std;

int main()
{
//Declares queue with default deque base container
queue <char> que;

//Explicitly declares a queue with deque base container
queue <char, deque<char> > que1;

//These lines don't cause an error, even though they
//declares a queue with a vector base container
queue <int, vector<int> > que2;
que2.push(12);
//but the following would cause an error because vector has
//no pop_front() member function
//que2.pop();

//Declares a queue with list base container
queue <int, list<int> > que3;
cout<<endl;
```

```
return 0;
}
//no output
```

queue Class

- A template container adaptor class that provides a restriction of functionality for some underlying container type, limiting access to the front and back elements.
- Elements can be added at the back or removed from the front, and elements can be inspected at either end of the queue.

```
template <
   class Type,
   class Container = deque<Type>
>
```

Parameters

Parameter	Description
`Type`	The element data type to be stored in the queue.
`Container`	The type of the underlying container used to implement the queue.

Table 30.11

- The elements of class `Type` stipulated in the first template parameter of a queue object are synonymous with value `_type` and must match the type of element in the underlying container class `Container` stipulated by the second template parameter.
- The `Type` must be assignable, so that it is possible to copy objects of that type and to assign values to variables of that type.
- Suitable underlying container classes for queue include deque and list, or any other sequence container that supports the operations of `front()`, `back()`, `push_back()`, and `pop_front()`.
- The underlying container class is encapsulated within the container adaptor, which exposes only the limited set of the sequence container member functions as a public interface.
- The queue objects are equality comparable if and only if the elements of class `Type` are equality comparable, and are less-than comparable if and only if the elements of class `Type` are less-than comparable.
- These three types of container adaptors defined by the STL, each restricts the functionality of some underlying container class to provide a precisely controlled interface to a standard data structure.
- The following is a program example compiled using **g++**.

```
//*******stackpopush.cpp*******
//stack, pop(), push()
//size() and top()
#include <stack>
#include <iostream>
using namespace std;

int main()
{
stack <int> st1, st2;

//push data element on the stack
st1.push(21);
int j=st1.top();
cout<<j<<' ';
st1.push(9);
j=st1.top();
cout<<j<<' ';
st1.push(12);
j=st1.top();
cout<<j<<' ';
st1.push(31);
j=st1.top();
cout<<j<<' '<<endl;

stack <int>::size_type i;
i = st1.size();
cout<<"The stack length is "<<i<<endl;

i = st1.top();
```

```
cout<<"The element at the top of the stack is "<<i<<endl;

st1.pop();

i = st1.size();
cout<<"After a pop, the stack length is "<<i<<endl;

i = st1.top();
cout<<"After a pop, the element at the top of the stack is "<<i<<endl;
return 0;
}
```

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

```
21 9 12 31
The stack length is 4
The element at the top of the stack is 31
After a pop, the stack length is 3
After a pop, the element at the top of the stack is 12
```

```
//*****queuepop.cpp*******
//queue, pop()
#include <queue>
#include <iostream>
using namespace std;

int main()
{
queue <int> que;

que.push(21);
que.push(9);
que.push(13);

queue <int>::size_type i;
i = que.size();
cout<<"The queue length is "<<i<<endl;

i = que.front();
cout<<"The element at the front of the queue is "<<i<<endl;

que.pop();
i = que.size();
cout<<"After a pop the queue length is "<<i<<endl;

i = que.front();
cout<<"After a pop, the element at the front of the queue is "<<i<<endl;
return 0;
}
```

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

```
The queue length is 3
The element at the front of the queue is 21
After a pop the queue length is 2
After a pop, the element at the front of the queue is 9
```

--End of Container Adaptor----------------------------------

---www.tenouk.com---

Further reading and digging:

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