【C++】Day37

Class	C++
Date	@January 12, 2022
Material	
Series Number	
Summary	

【Ch9】Sequential Container

9.6 Container Adaptors

An adaptor is a mechanism for making one thing act like another.

A container adaptor takes an existing container type and makes it act like a different type. For example, the stack adaptor takes a sequential contaienr(other than `array` or `froward_list`) and makes it operates as if it were a stack.

Table 9.17. Operations and Types Common to the Container Adaptors

`size_type`	Type large enough to hold the size of the largest object of this type.
`value_type`	Element type.
`container_type`	Type of the underlying container on which the adaptor is implemented.
`A a;`	Create a new empty adaptor named `a`.
`A a(c);`	Create a new adaptor named `a` with a copy of the container `c`.
relational operators	Each adaptor supports all the relational operators: `==`, `!=`, `<`, `<=`, `>`, `>=`. These operators return the result of comparing the underlying containers.
`a.empty()`	`false` if `a` has any elements, `true` otherwise.
`a.size()`	Number of elements in `a`.
`swap(a, b)` `a.swap(b)`	Swaps the contents of `a` and `b`; `a` and `b` must have the same type, including the type of the container on which they are implemented.

Defining an Adaptor

Each adaptor defines two constructors: the default constructor that creates an empty object, and a constructor that takes a container and initializes the adaptor by coping the given container.

For example, assuming that dep is a `deque<int>`, we can use `deq` to initialize a new `stack` as follows:

```
stack<int> stk(deq); //copies elements from deq into stk
```

We can also override the default container type by naming a sequential container as a second type argument when we create the adaptor:

```
//empty stack implemented on top of vector
stack<string, vector<string>> str_stk;
//str_stk2 is implemented on top of vector and initially holds a copy of svec
stack<string, vector<string>> str_stk2(svec);
```

There are constraints on which containers can be used for a given adaptor:

- All of the adaptors require the ability to add and remove elements. As a result, they cannot be built on an array. Similarly, we cannot use `forward_list` because all of the adaptors requrie operations that add, remove, or access the last element in the container
- A `stack` requries only `push_back`, `pop_back`, and `back` operations, so we can use any of the remaining container types for a `stack`.
- The `queue` adaptor requires `back`, `push_back`, `front`, and `push_front`, so it can be built on a `list` or `deque` but not on a `vector`.
- A `priority_queue` requires random access in addition to the `front`, `push_back`, and `pop_back` operations; it can be built on a `vector` or a `deque` but not on a `list`.

Stack Adaptor

The stack type is defined in the stack header. The operations provided by a stack are listed in the following table.

Table 9.18. Stack Operations in Addition to Those in Table 9.17

Uses `deque` by default; can be implemented on a `list` or `vector` as well.	
`s.pop()`	Removes, but does not return, the top element from the `stack`.
`s.push(item)`	Creates a new top element on the `stack` by copying or moving `item`, or
`s.emplace(args)`	by constructing the element from *args*.
`s.top()`	Returns, but does not remove, the top element on the `stack`.

The following program illustrates the use of stack:

```
stack<int> intStack; //empty stack
for(size_t ix = 0; ix != 10; ++ix) {
  intStack.push(ix);
}
while(!intStack.empty()) {
  int value = intStack.top();
  cout << value << endl;
  intStack.pop(); //pop the top element, and repeat
}
```

The Queue Adaptors

The queue and priority_queue adaptors are defined in the queue header.

By default `queue` uses `deque` and `priority_queue` uses `vector`; queue can use a `list` or `vector` as well, `priority_queue` can use a `deque`.	
`q.pop()`	Removes, but does not return, the front element or highest-priority element from the queue or `priority_queue`, respectively.
`q.front()`	Returns, but does not remove, the front or back element of `q`.
`q.back()`	**Valid only for `queue`**
`q.top()`	Returns, but does not remove, the highest-priority element. **Valid only for `priority_queue`.**
`q.push(item)`	Create an element with value `item` or constructed from *args* at the end
`q.emplace(args)`	of the queue or in its appropriate position in `priority_queue`.