【C++】Day33

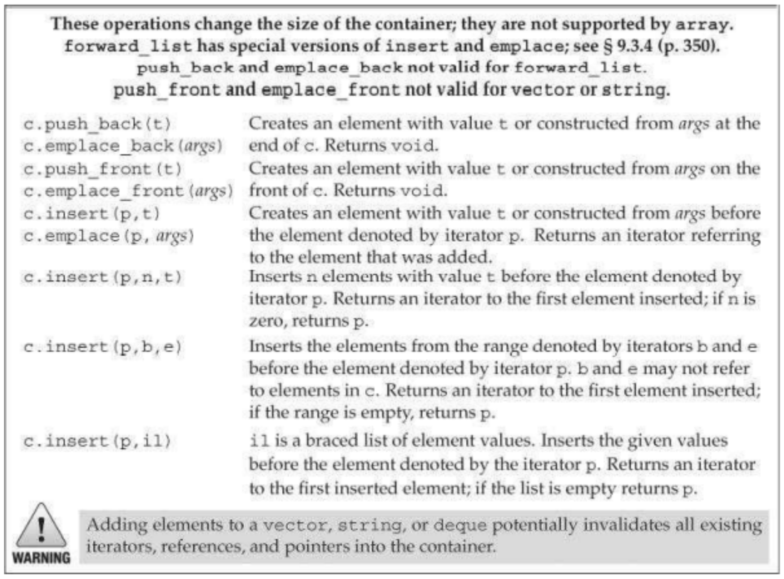
Class	C++
Date	@January 5, 2022
Material	
Series Number	
Summary	

【Ch9】Sequential Container

9.3 Sequential Container Operations

9.3.1 Adding Elements to a Sequential Container

Excepting array, all of the library containers provide flexible memory management. We can add or remove elements dynamically changing the size of the container at run time.



These operations change the size of the container; they are not supported by `array`.
`forward_list` has special versions of `insert` and `emplace`; see § 9.3.4 (p. 350).
`push_back` and `emplace_back` not valid for `forward_list`.
`push_front` and `emplace_front` not valid for `vector` or `string`.

`c.push_back(t)`
`c.emplace_back(`*args*`)`
	Creates an element with value `t` or constructed from *args* at the end of `c`. Returns `void`.

`c.push_front(t)`
`c.emplace_front(`*args*`)`
	Creates an element with value `t` or constructed from *args* on the front of `c`. Returns `void`.

`c.insert(p,t)`
`c.emplace(p, `*args*`)`
	Creates an element with value `t` or constructed from *args* before the element denoted by iterator `p`. Returns an iterator referring to the element that was added.

`c.insert(p,n,t)`	Inserts `n` elements with value `t` before the element denoted by iterator `p`. Returns an iterator to the first element inserted; if `n` is zero, returns `p`.
`c.insert(p,b,e)`	Inserts the elements from the range denoted by iterators `b` and `e` before the element denoted by iterator `p`. `b` and `e` may not refer to elements in `c`. Returns an iterator to the first element inserted; if the range is empty, returns `p`.
`c.insert(p,il)`	`il` is a braced list of element values. Inserts the given values before the element denoted by the iterator `p`. Returns an iterator to the first inserted element; if the list is empty returns `p`.

WARNING Adding elements to a `vector`, `string`, or `deque` potentially invalidates all existing iterators, references, and pointers into the container.

Adding elements to a vector or a string may cause the entire object to be reallocated.

Reallocating an object requires allocating new memory and moving elements from the old space to the new. This is time-consuming.

Key Concept: Container Elements Are Copies

When we use an object to initialize a container, or insert an object into a container, a copy of that object's value is placed in the container, not the object itself.

Key Concept: Passing a Container

When we pass a container into the function, we are passing a copy of the container. Not the container itself

```
//a copy of vec is passed in
void func(vector<int> vec) {
  vec.erase();
  cout << vec.size();
}

int main() {
  vector<int> vec(10, 50);
  //this function will not affect the elements in vec
  func(vec);
  cout << vec.size();
  return 0;
}
```

However, if we pass in a reference or pointer to the container, then the elements inside are changeable:

```
//the reference(address) of the vector is passed and thus make vec changable
void func(vector<int> &vec) {
  //clear() will clear all of the elements in the container
  vec.clear();
  //print 0
  cout << vec.size();
}

int main() {
  vector<int> vec(10, 50);
  func(vec);
  //print 0
  cout << vec.size();
  return 0;
}
```

Using push_front

In addition to `push_back`, the `list`, `forward_list`, and `deque` containers support an analogous operation named `push_front`. This operation inserts a new element at the front of the container:

```
list<int> ilist;
//add elements to the start of ilist
for(size_t ix = 0; ix != 4; ++ix)
  ilist.push_front(ix);
```

The elements in the list will be `{ 3, 2, 1, 0 }`.

Adding Elements at a Specified Point in the Container

The `insert` members let us insert zero or more elements at any point in the container.

Each of the insert functions takes an iterator as its first argument. The iterator indicates where in the container to put the elements. It can refer to any position in the container, including one past the end of the container.

Elements are inserted before the position denoted by the iterator. For example, this statement:

```
slist.insert(iter, "Hello!"); //insert "Hello!" just before iter
```

inserts a string with value `“Hello!”` just before the element denoted by `iter`.

We can insert elements to the beginning of the container without worrying about whether the container has `push_front`:

```
vector<string> svec;
list<string> slist;
//equivalent to calling slist.push_front("hello");
slist.insert(slist.begin(), "Hello!");
svec.insert(svec.begin(), "Hello!");
```

Warning: It is legal to insert anywhere in a vector, deque or string. However, doing so can be an expensive operation.

Inserting a Range of Elements

The version of `insert` that takes an element count an a value adds the specified number of identical elements before the given position:

```
svec.insert(svec.end(), 10, "hello");
```

This code inserts ten elements at the end of `svec` and initializes each of those elements to the string “Anna”.

The version of `insert` that takes a pair of iterators or an initializer list insert the elements from the given range before the given position:

```
vector<string> v = { "quasi", "simba", "frollo", "scar" };
//insert the last two elements of v at the begining of slist
slist.insert(slist.begin(), v.end() - 2, v.end());
slist.insert(slist.end(), {"these", "word", "will", "go", "at", "the", "end" };
//run-time error: iterators denoting the range to copy from must not refer to the same container as the one we are changing
slist.insert(slist.begin(), slist.begin(), slits.end());
```

Under the new standard, the versions of `insert` that takes a count or a range return an iterator to the first element that was inserted.

Using the Return from insert

We can use the value returned by insert to repeatedly insert elements at a specified position in the container:

```
list<string> lst;
auto iter = list.begin();
string word;
while(cin >> word)
  iter = lst.insert(iter, word); //same as calling push_front
```

Using the Emplace Operations

The new standard introduced three new members-`emplace_front` , `emplace` and `emplace_back` -that construct rather than copy elements.

When we call an `emplace` member, we pass arguments to a constructor for the element type. The `emplace` members use those arguments to construct an element directly in space managed by the container.

For example, assuming `c` holds `Sales_data` elements:

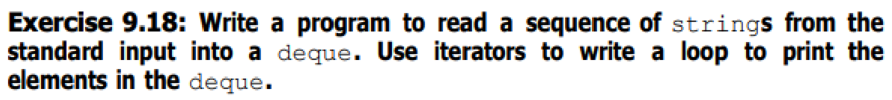
```
//construct a Sales_data object at the end of c
c.emplace_back("ISBN: xxx", 25, 16.0);
//equivalenet
c.push_back(Sales_data("ISBN: xxx", 25, 16.0));
```

The arguments to an `emplace` function vary depending on the element type. The arguments must match a constructor for the element type:

```
/iter refers to an element in c, which holds Sales_data elements
c.emplace_back(); //uses the Sales_data default constructor
c.emplace(iter, "99-999"); //uses Sales_data(string)
//uses the Sales_data constructor that takes an ISBN, a count, and a price
c.emplace_front("ISBN: xxx", 25, 16.00);
```

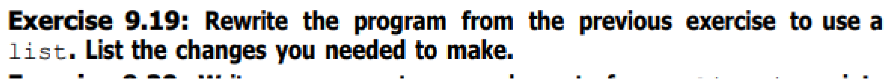
Note: The emplace functions construct elements in the container. The arguments to these functions must match a constructor for the element type.

Exercise

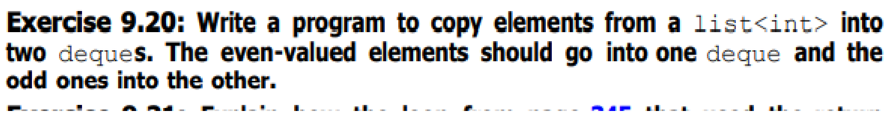


Exercise 9.18: Write a program to read a sequence of `string`s from the standard input into a `deque`. Use iterators to write a loop to print the elements in the `deque`.

```
#include <iostream>
#include <string>
#include <deque>

int main(int argc, char* argv[]) {
    std::deque<std::string> string_deque;
    std::string str;
    while(std::cin >> str) {
        //using push_back
        //string_deque.push_back(str);
        //using insert
        string_deque.insert(string_deque.end(), str);
    }


    auto end = string_deque.cend();
    for(auto begin = string_deque.cbegin(); begin != end; ++begin) {
        std::cout << *begin << std::endl;
    }
    return 0;
}
```


Exercise 9.19: Rewrite the program from the previous exercise to use a `list`. List the changes you needed to make.

```
#include <iostream>
#include <string>
#include <list>

int main(int argc, char* argv[]) {
    std::list<std::string> string_list;
    std::string str;
    while(std::cin >> str) {
        //using push_back
        //string_list.push_back(str);
        //using insert
        string_list.insert(string_list.end(), str);
    }


    auto end = string_list.cend();
    for(auto begin = string_list.cbegin(); begin != end; ++begin) {
        std::cout << *begin << std::endl;
    }
    return 0;
}
```


Exercise 9.20: Write a program to copy elements from a `list<int>` into two `deque`s. The even-valued elements should go into one `deque` and the odd ones into the other.

See 9_20.cpp for code