Chapter 9: Sequential Container[1]

9.1 Overview of the Sequential Containers

Sequential containers (also known as sequence containers) are ordered collections in which every element has 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 six elements into an ordered collection by appending each element at the end of the collection, these elements are in the exact order in which you put them.

The STL contains five predefined sequential container classes: `array`, `vector`, `deque`, `list`, and `forward_list`. Let us start our quick overview of sequential containers with the one we are familiar with: `vector`, followed by `deque`, `array`, `list` (`forward_list`).

(see ppt)

Using `vector`

VecEx.cpp

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

int main()
{
    vector<int> coll;    // vector container for ints

    // insert elements with the values 2 to 7
    for (auto i = 2; i < 8; ++i)
        coll.push_back(i);

    // output all elements followed by a space
    for (auto e : coll)
        cout << e << ' ';
    cout << endl;
    return 0;
}
```

Q: what are the outputs?

A:

(see ppt)

[1] I will use "The C++ Standard Library: A Tutorial and Reference" 2nd Edition by NM Josuttis constantly to reinforce key concepts from the textbook.

Using `deque`

DequeEx.cpp

```
#include <deque>
#include <iostream>
#include <string>
using namespace std;

int main()
{
    deque<string> coll;    // deque container for strings

    // insert elements at the front
    coll.push_front("programming");
    coll.push_front("C++");
    coll.push_front("I love");

    // insert an element at the back
    coll.push_back("language");

    // output all elements followed by a space
    for (auto e : coll)
        cout << e << ' ';
    cout << endl;
    return 0;
}
```

Q: what are the outputs?

A:

(see ppt)

Using `array`

STL container class `array<>` has some unique semantics regarding to initialization and we will cover it later. For now, let us look at its simple usage.

ArrayEx.cpp

```
#include <array>
#include <string>
#include <iostream>
using namespace std;

int main()
{
    array<string, 5> coll ;
    coll[0] = "I";
    coll[1] = "love";
```

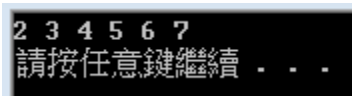
```
    coll[2] = "C++";
    coll[3] = "programming";
    coll[4] = "language";
    for (auto e : coll)
        cout << e << " ";
    cout << endl;
    return 0;
}
```

Q: what are the outputs?

A:

Ex91.cpp

In-class Coding Exercise 9.1: Use `std::array` to add `2 3 4 5 6 7` and output the values, similar to what we have done for `VecEx.cpp`.



(Answer)

Remark: Note that the number of elements is a part of the **type** of an array. Thus, `array<int,2>` and `array<int,10>` are **two different types**.

(see ppt)

Using `list`

ListEx.cpp

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

int main()
{
    list<char> coll;    // list container for chars

    for (auto c = 'a'; c <= 'z'; ++c)
        coll.push_back(c);

    for (auto e : coll)
        cout << e << " ";

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

Q: what are the outputs?

A:

Remark: To print all elements, a **range-based for loop** is used, which is available since C++11 and allows performing statements with each element. **A direct element access by using operator `[]` is not provided for lists** for performance consideration.

In-class Coding Exercise 9.2: Add a range-based for loop so we change the vowels (‘a’, ‘e’, ‘i’, ‘o’, ‘u’) in the container with capital letters (‘A’, ‘E’, ‘I’, ‘O’, ‘U’).

Ex92.cpp

```
#include <list>
#include <iostream>
#include <cctype>
```

```
using namespace std;

int main()
{
    list<char> coll;    // list container for chars

    for (auto c = 'a'; c <= 'z'; ++c)
        coll.push_back(c);

    // your codes




    for (auto e : coll)
        cout << e << " ";

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

A:

(see ppt)

`forward list`

Conceptually, a forward list is a restricted list such that **it is not able to iterate backward**. As benefits, it uses less memory and provides slightly better runtime performance than list.

```
#include <forward_list>
#include <iostream>

using namespace std;

int main()
{
    forward_list<char> coll = {'o', 'o', 'p'};

    for (auto e : coll)
        cout << e ;

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

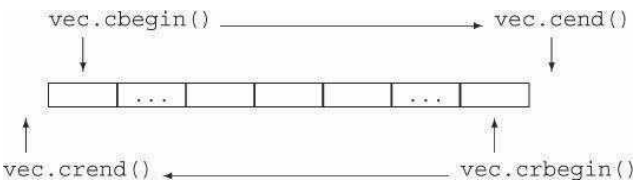
A:

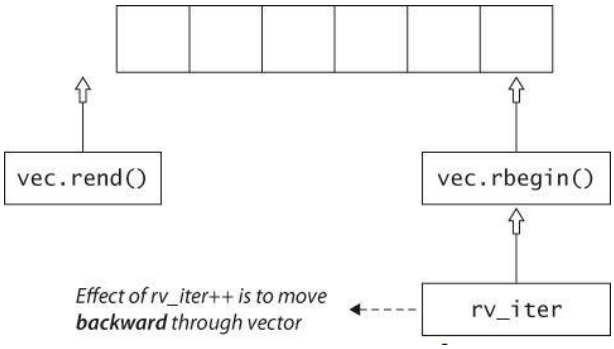
(see ppt)

9.2.1 Iterator

`begin` **and** `end` **Members**

The `begin` and `end` operations yield iterators that refer to the **first** and **one past the last element** in the container. The `rbegin` and `rend` operations yield iterators that refer to the **last** and **one past the first element** in the container. `cbegin`, `cend`, `crbegin` and `crend` yield constant iterators.





IterEx.cpp

```
#include <list>
#include <string>
#include <iostream>
using namespace std;

int main(){
    list<string> a = { "SW Chang", "CC Chou", "CS Chen" };
    cout << "The first author is: " << *(a.begin()) << endl;
    cout << "The last author is: " << *(a.rbegin()) << endl;
    cout << "The authors in order are: ";
    for (auto it = a.cbegin(); it != a.cend(); ++it)
        cout << *it << " ";
    cout << endl;
    cout << "The authors in reverse order are: ";
    for (auto it = a.crbegin(); it != a.crend(); ++it)
        cout << *it << " ";
    cout << endl;
    return 0;
}
```

```
The first author is: SW Chang
The last author is: CS Chen
The authors in order are: SW Chang CC Chou CS Chen
The authors in reverse order are: CS Chen CC Chou SW Chang
請按任意鍵繼續 . . .
```

Range-Based `for` Loops versus Iterators

Having introduced iterators, we can explain the exact behavior of range-based `for` loops. **For containers, in fact, a range-based `for` loop is nothing but a convenience interface**, which is defined to iterate over all elements of the passed range/collection. Within each loop body, the actual element is initialized by the value the current iterator refers to. Thus,

```
for (type elem : coll) {
...
```

```
}
```

is interpreted as

```
for (auto pos=coll.begin(); pos!=coll.end(); ++pos) {
    type elem = *pos;
}
```

For example

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

int main()
{
    vector<int> coll;    // vector container for ints

    // insert elements with the values 0 to 5
    for (int i = 0; i < 6; ++i)
        coll.push_back(i);

    // output all elements followed by a space
    for (auto e: coll)
        cout << e << ' ';
    cout << endl;
    return 0;
}
```

The range `for` highlighted in red is interpreted as:

```
for (auto pos=coll.begin(); pos!=coll.end(); ++pos)
    { int elem = *pos;
    cout << elem << ' ';
}
```

In-class Coding Exercise 9.3: rewrite those codes highlighted in red using iterator operation.

Ex93.cpp

```
#include <list>
#include <iostream>
#include <cctype>
using namespace std;

int main()
{
    list<char> coll;    // list container for chars

    for (auto c = 'a'; c <= 'z'; ++c)
```

```
        coll.push_back(c);

    for (auto& e : coll) {
        if (e == 'a' || e == 'e' || e == 'i' || e == 'o' || e == 'u')
            e = toupper(e);
    }

    for (auto e : coll)
        cout << e << " ";

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

A:

(see ppt)

9.2.4 Defining and Initializing a Container

```
list<string> authors = {"Milton", "Shakespeare", "Austen"};
list<string> list2(authors); // ok: types match
deque<string> authList1(authors); // error: container types don't match
deque<string> authList2(authors.begin(),authors.end());
```

```
vector<const char*> articles = {"a", "an", "the"};
vector<string> words = articles; // error: element types must match
// ok: converts const char* elements to string
forward_list<string> words(articles.begin(), articles.end());
```

Special issues on STL `array<>`

The fixed-size nature of `array`s affects the behavior of the constructors that `array` can define.

(Rule 1) Unlike the other containers, a default-constructed array is not empty: it has as many elements as its size. These elements are default initialized just as are elements in a built-in array.

(Rule 2) If we list initialize the `array`, the number of the initializers must be equal to or less than the size of the array. If there are fewer initializers than the size of the array, the initializers are used for the first elements and any remaining elements are value initialized (e.g. `int` will be initialized to `0` and class must have a default constructor)

```
array<int, 10> ia1; // ten default-initialized ints (奇怪的數字)
array<int, 10> ia2 = {0,1,2,3,4,5,6,7,8,9}; // list initialization
array<int, 10> ia3 = {42}; // ia3[0] is 42, remaining elements are 0
```

Remark: STL `array<>` interface support is better than built-in array. For example, we cannot copy or assign objects of built-in array types but we can do it on STL `array<>`.

```
int digs[10] = {0,1,2,3,4,5,6,7,8,9};
int cpy[10] = digs; // error: no copy or assignment for built-in arrays

array<int, 10> digits = {0,1,2,3,4,5,6,7,8,9};
array<int, 10> copy = digits; // ok: so long as array types match
```

9.3.1 Adding Elements to a Sequential Container

(see ppt)

`push_back`: `vector`, `deque`, `list`, `string`

`push_front`: `deque`, `list`, `forward_list`

Add at a specific position in a container (`insert`)

The `push_back` and `push_front` operations provide convenient ways to insert a single element at the end or beginning of a sequential container. The `insert` members let us insert zero or more elements at any point in the container. The `insert` members are supported for `vector`, `deque`, `list`, and `string`.

Each of the `insert` functions takes an **iterator** as its first argument. The iterator indicates where in the container to put the element(s). Element(s) are inserted **before** the position denoted by the iterator.

Q: (Food for thought) why before, why not after? Think about the left inclusive rule [begin, end) in C++.

A:

Insert a single element

InsertExample1.cpp

```
list<string> slist;
// equivalent to calling slist.push_front("Hello");
slist.insert(slist.begin(), "Hello!");
// equivalent to calling slist.push_back("world");
slist.insert(slist.end(), "world");

// no push_front on vector but we can insert before begin()
// warning: inserting anywhere but at the end of a vector might be slow
vector<string> svec;
svec.insert(svec.begin(), "Hello!");
```

Q: what are the output?

A:

Insert a range of elements

InsertExample2.cpp

```
vector<string> v = {"quasi", "simba", "frollo", "scar"};
// insert the last two elements of v at the beginning of slist
slist.insert(slist.begin(), v.end() - 2, v.end());
slist.insert(slist.end(), {"these", "words", "will",
"go", "at", "the", "end"});
```

Q: what are the output?

A:

In addition to the versions of `insert` that take iterators, `string` provides versions that **take an index**. The index indicates the position before which to `insert` the given values:

`s.insert(pos, args)`	Insert characters specified by *args* before *pos*. *pos* can be an index or an iterator. Versions taking an index return a reference to `s`; those taking an iterator return an iterator denoting the first inserted character.

```
string s1 = "I love C++";
s1.insert(s1.size(), 3, '!'); // s1 == "I love C++!!!
```

```
string s2("C++ Primer");
s2.insert(s2.size(), " 4th Ed."); // s == "C++ Primer 4th Ed."
```

Inserting a Range of Elements

`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`.

Example: Consider two sorted containers: one is a `list` with three elements `{"Austen", "Milton", "Shakespeare"}`. The other is a `vector` with another three elements `{ "Dickens", "Freud", "Hemingway" }`. Insert all the elements in vector into the list while maintain their relative alphabetic order between `list` and `vector`.

```
The list after insertion is:
Austen Dickens Freud Hemingway Milton Shakespeare
請按任意鍵繼續 . . .
```

insertRange.cpp

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

using namespace std;

int main()
{
    list<string> aList = { "Austen", "Milton", "Shakespeare" };
    vector<string> v = { "Dickens", "Freud", "Hemingway" };
    auto iter = aList.begin();

    while (iter != aList.end()) {
        if (*v.rbegin() < *iter)
            { aList.insert(iter, v.begin(),
            v.end()); break;
        }
        ++iter;
    }

    cout << "The list after insertion is: " << endl;
    for (auto e : aList)
        cout << e << " ";
    cout << endl;
    return 0;
}
```

Using the Return from `insert`

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

The iterator returned by `insert` is sometimes very useful. For example, we can use the value to repeatedly insert elements at a specific position in the container.

Example: Consider again a list with three elements `{"Austen", "Milton", "Shakespeare"}`. Repeatedly insert elements given from input in front of `"Milton"`.

```
Insert a few words in front of Milton:
Freud Alice Nilson
^Z
The list after insertion is:
Austen Nilson Alice Freud Milton Shakespeare
請按任意鍵繼續 . . .
```

insertReturn.cpp

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

using namespace std;

int main()
{
    list<string> aList = { "Austen", "Milton", "Shakespeare" };

    auto iter = aList.begin();
    while (iter != aList.end())
        { if (*iter == "Milton")
            break;
        ++iter;
    }
    string word;

    cout << "Insert a few words in front of Milton: " << endl;
    while (cin >> word)
        iter = aList.insert(iter, word);

    cout << "The list after insertion is: " << endl;
    for (auto e : aList)
        cout << e << " ";
    cout << endl;
    return 0;
}
```

In class Exercise 9.4: The above codes insert elements given from input in front of "Milton". Nevertheless, the insertion elements are in an order of FILO (first in last out). Now rewrite the codes so the insertion elements are in an order of FIFO (first in first out).

```
Insert a few words in front of Milton:
Freud Alice Nilson
^Z
The list after insertion is:
Austen Freud Alice Nilson Milton Shakespeare
請按任意鍵繼續 . . .
```

Ex94.cpp

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

using namespace std;

int main()
{
    list<string> aList = { "Austen", "Milton", "Shakespeare" };

    auto iter = aList.begin();
```

```
    while (iter != aList.end())
        { if (*iter == "Milton")
            break;
        ++iter;
    }
    string word;

    cout << "Insert a few words in front of Milton: " << endl;

    // your codes



    cout << "The list after insertion is: " << endl;
    for (auto e : aList)
        cout << e << " ";
    cout << endl;
    return 0;
}
```

Answer:

Using the Emplace Operations (C++11)

The new standard introduced three new members—`emplace_front`, `emplace`, and `emplace_back`—that **construct** rather than **copy** elements. These operations correspond to the copy version of `push_front`, `insert`, and `push_back` operations.

```
vector<Sales_item> c;
// construct a Sales_data object at the end of c
// uses the three-argument Sales_data constructor
c.emplace_back("978-0590353403", 25, 15.99);
// error: there is no version of push_back that takes three arguments
c.push_back("978-0590353403", 25, 15.99);
// ok: we create a temporary Sales_data object to pass to push_back
c.push_back(Sales_data("978-0590353403", 25, 15.99));
```