ENGG1340 Computer Programming II
Module 10: STL – Iterators

Objective:
- Understand the usage of STL iterators and algorithms

1. Iterators

Items in containers can be accessed through **iterators**. They are classes defined by STL for indexing items. We first introduce the common operations with iterators.

1.1 Declare an iterator

To declare an iterator, we first specify the container type, followed by the scope resolution operator (`::`) and then the word `iterator`. For example, the first line below declares an iterator `itr` for indexing items in a vector of `string`.

```
vector<string>::iterator itr;
list<int>::iterator itr2;
```

The second line declares an iterator `itr2` for indexing a list of `int`. Note that an iterator for a list cannot be used for indexing items in a vector. Furthermore, an iterator for a list of `string` cannot be used for indexing items in a list of `int`.

1.2 Indexing items in a container

Given a container `c`, then `c.begin()` returns an iterator pointing to the first items in `c`. We can assign this value to an iterator, say, `itr`. We can access the item pointed by an iterator using the dereference operator `*`. See below for an example.

```
vector<int> v;
v.push_back(3);
v.push_back(4);
v.push_back(5);  // v = [3, 4, 5] now

vector<int>::iterator itrOfV = v.begin();
*itrOfV = 100;   // change value 3 to 100
cout << *itrOfV << endl;  //print 100
```

Note that in the second last line, we dereference an iterator and change the value of the items.

1.3 Moving the iterator

Given an iterator pointing to an object `obj`, we can move the iterator so that it points to the object after `obj` using the `++` operator. For example, following the previous example, the code below prints the complete content of `v`.

```
vector<int>::iterator itr = v.begin();
for (int i = 0; i < v.size(); i++) {
       cout << *itr << ' ';
       itr++;
}
// print 100 4 5
```

1.4 Testing the end of the container

Given a container `c`, then `c.end()` returns an iterator pointing to the item **after** the last items. Hence, `c.end()` does not point to any real item. Instead, it is used for testing whether an iterator has gone through the container. In particular, if an iterator equals `c.end()`, then the iterator has passed the last items and the processing should finish.

For example, the code below prints the content of a vector `v`.

```
        vector<int>::iterator itr2;
        for (itr2 = v.begin(); itr2 != v.end(); itr2++) {
               cout << *itr2 << ' ';
        }
        // If v = [100,4,5], it prints 100 4 5
```

1.5 Iterator examples

The programs below print the content of a `vector` and a `list`.

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
       vector<int> input;
       for (int i = 0; i < 10; i++) {
              input.push_back(i);
       }

       vector<int>::iterator itr;
       for (itr = input.begin(); itr != input.end(); itr++) {
              cout << *itr << endl;
       }
}
```

```
#include <iostream>
#include <list>
using namespace std;
int main() {
       list<int> input;
       for (int i = 0; i < 10; i++) {
              input.push_back(i);
       }

       list<int>::iterator itr;
       for (itr = input.begin(); itr != input.end(); itr++) {
              cout << *itr << endl;
       }
}
```

Note that the two programs differ only by changing each `vector` into `list`. STL are designed so that the syntax for different containers is almost the same. It eases the usage of the containers.

Printing the content of a `map` is similar. But since items in maps are `<key,value>` pairs, after dereferencing an iterator, we need to specify whether to access the key or the value. `(*itr).first` accesses the key and `(*itr).second` accesses the value.

```
#include <iostream>
#include <map>
using namespace std;
int main() {
	map<int, double> input;
	for (int i = 0; i < 5; i++) {
		input[i * 3 % 5] = i + 0.1;
	}

	map<int, double>::iterator itr;
	for (itr = input.begin();
		itr != input.end(); itr++) {
		cout << (*itr).first << ' ' << (*itr).second << endl;
	}
}
```

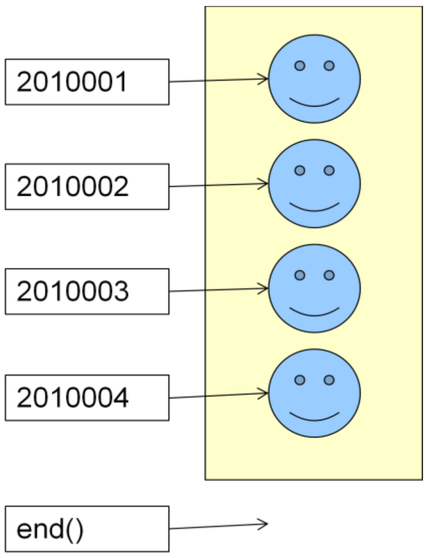
Note that the key value pairs are inserted in the order of `<0,0.1>, <3,1.1>, <4,2.1>, <2,3.1>` and `<1,4.1>`. What is the output of the program? The answer is shown below.

```
0 0.1
1 4.1
2 3.1
3 1.1
4 2.1
```

Notice that the pairs are printed in ascending order of the key. It is expected because a `map` stores the pairs in sorted order of the keys. Hence, you may use a `map` to sort a set of items indirectly.

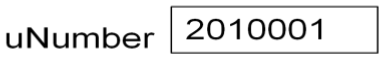
1.6 Real world example

Some people have difficulties understanding iterators. We try to give a real-world analogy here. Assume we have a group of students. An iterator is like a student number.

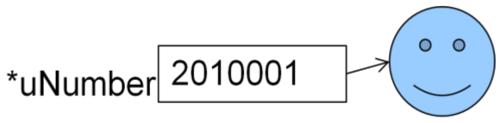


When we declare an iterator `vector<string>::iterator itr`, it is like creating a student number variable.

We use `itr = v.begin()` to obtain an iterator to the first item. It is just like obtaining the student number of the first student. In this example, it is 2010001.



Dereferencing an iterator by `*itr` is like calling out the student using the student number.



A change in the iterator is like a change in the student number, so that it points to another student. On the other hand, if we dereference an iterator first and then performance changes on the item, it is like calling the student out and the changes the attribute of the student (e.g., giving him to a prize, etc). The change is on the item (i.e., the student), but the iterator (i.e., student number) is unchanged and it points to the same item as before.

Finally, as shown in the figure, `end()` does not point to any item. It points to the item after the last item, which is used to test whether all items have been passed.

1.7 Types of iterators

There are different types of iterators. Forward iterators are simple iterators that support all the operations we introduced in Section 1.5.

Category	Functions supported
Forward iterator	Assignment: `a = b;` Increment: `a++;` Dereference: `*a;` Equality test: `a == b;`
Bidirectional iterator	Bidirectoional iterators are also forward iterators Decrement: `a--;`
Random access iterator	Random access iterators are also bidirectional Arithmetic +/-: `a+5; a-5; a-b;` Inequality test: `a<b; a>b; a<=b; a>=b;` Compound: `a += 5; a -= 5;` Offset dereference: `a[5]; //i.e., *(a+5)`

Bidirectional iterators are iterators that support one more operation, which is the `--` decrement operator. It moves the iterator to the items one position earlier. Note that bidirectional iterators are also forward iterators. They support all operations of forward iterators and can be used wherever forward iterators can be used. The reverse is not true.

Random access iterators are more general that they allow 4 more operations.

- Arithmetics: If `a` is an iterator, `a+i` is an iterator pointing to the item that is `i` positions after the items pointed by `a`. For example, if `a` points to `v[4]` of a vector `v`, then `a+5` points to `v[9]` of the vector. `a-i` is an iterator pointing to the item at `i` positions before that pointed by `a`. If `a` and `b` are iterators, `a-b` returns how many position is `a` after `b`. E.g., for any vector `v`, `v.end() - v.begin()` equals `v.size()`.
- Inequality test: `a<b` is defined and is true if the item pointed by `a` is at an earlier position than `b`. `a<=b, a>b, a>=b` are defined similarly.
- Compound: Operators like `+=, -=` are defined
- Offset dereference: The operator `[ ]` is defined. For any integer `i`, `a[i] = *(a+i)`.

All iterator operations specified above take only O(1) time.

The type of an iterator is determined by the container which the iterator is pointing to. Iterators to vectors are random access iterators. Iterators to list or map are bidirectional iterators.

1.8 Iterators and pointers

We notice that iterators and pointers are similar and they are both used for pointing to items. Furthermore, to access the items that are pointed by an iterator or a pointer, we use the dereference operator `*` in both cases.

However, iterators and pointers are substantially different.

- A pointer is the **memory address** of the item it points to. An iterator is not an address. It is an object that stores information about the item it points to. What information an iterator store is usually unknown to the programmers.
- If we perform an increment operator `++` to a pointer, it increases the memory address stored in the pointer by the size of one item. When we perform an increment operator `++` to an iterator, it moves the iterator to the next item in the container. This item can be stored anywhere in the memory, especially for the case of `list` or `map`.

Generally speaking, iterators are designed to avoid low-level memory manipulation. It is usually easier to work with iterators.

2 Algorithms

Recall that STL provides containers for storing items and iterators for accessing items. It also provides algorithms to manipulate items. There are about 70 algorithms totally. The algorithms are generic that they can be applied to items of different types. These functions are called **template functions**. Below we introduce a few commonly used algorithms. To use those algorithms, include the <algorithm> library.

```
#include <algorithm>
```

2.1 `sort`

One of the most commonly used algorithms is `sort()`, which sorts an array of items.

```
void sort(RandomAccessIterator first,
          RandomAccessIterator last);
```

It sorts the items in the range `[first, last)` into ascending order. It is important to notice that the item pointed by `last` is not included in the sorting. See below for an example.

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

int main() {
	vector<string> input;
	for (int i = 0; i < 5; i++) {
		string s; cin >> s;
		input.push_back(s);
	}
	sort(input.begin(), input.end());

	vector<string>::iterator itr;
	for (itr = input.begin(); itr != input.end(); itr++)
		cout << *itr << endl;
}
```

If the input are "cat", "boy", "apple", "egg" and "dog", then the output would be "apple", "boy", "cat", "dog" and "egg". Notice that the range given is input.begin() and input.end(). The items pointed by input.end() is not included for sorting and it is correct.

Sorting a vector of int uses exactly the same function call to sort(), as shown below.

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

int main() {
	vector<int> input;
	for (int i = 0; i < 5; i++) {
		int s; cin >> s;
		input.push_back(s);
	}
	sort(input.begin(), input.end());

	vector<int>::iterator itr;
	for (itr = input.begin(); itr != input.end(); itr++)
		cout << *itr << endl;
}
```

We can choose not to sort the complete vector. Instead, we can sort only a certain range of the vector. The function call below sorts the items from the second position up to the second last position (inclusive)

```
    sort(input.begin() + 1, input.end() - 1);
```

Besides containers, we can also sort an array, as follows.

```
#include <iostream>
#include <algorithm>
using namespace std;
int main() {
	int input[10];
	for (int i = 0; i < 10; i++) {
		input[i] = i * 3 % 10;
	}
	sort(input, input + 10);

	for (int i = 0; i < 10; i++) {
		cout << input[i] << ' ';
	}
}
```

Note that STL would convert an integer pointer into a random access iterator. Note that the array name input is a constant pointer pointing to the first item and input+10 points to input[10], which is one position after the last item. Hence all items in input are sorted.

We cannot sort a list or a map using their iterators, because those containers provide only bidirectional iterators instead of random access iterators.

sort() takes O(n log n) time on average, where n is the number of items in the range.

2.2 Sorting objects of user-defined classes

Assume we have defined a new class `Email` and have a vector of `Email` objects. The following program tries to sort the objects.

```
class Email {
public:
	string sender;
	string subject;
};

int main() {
	vector<Email> v;
	for (int i = 0; i < 10; i++) {
		Email s;
		cin >> s.sender >> s.subject;
		v.push_back(s);
	}
	sort(v.begin(), v.end()); //Error!
	vector<Email>::iterator itr;
	for (itr = v.begin(); itr != v.end(); itr++)
		cout << (*itr).sender << '\t' << (*itr).subject << endl;
}
```

The program is very similar to that of sorting integers or strings. However, a compilation error occurs. The main problem is that the compiler does not know what the meaning of "ascending order" is for user-defined classes. We may want to sort by sender name first, or we may want to sort by subject first. The compiler does not know it and it requires us to define a **less-than relationship** between two `Email` objects.

Hence, we need to overload the less-than operator < for `sort()` to work with user-defined classes. We can add the following operator overloading before `main()`. Note that it will instruct `sort()` to order the emails by sender name first. For emails from the same sender, we will sort emails with shorter subjects to earlier positions.

```
bool operator <(const Email& a, const Email& b) {
	if (a.sender < b.sender) return true;
	if (a.sender > b.sender) return false;
	return a.subject.size() < b.subject.size();
}
```

You can define the less-than operator < in other way. The `sort()` function will arrange the objects in ascending order according to the operator defined.

3 A brief summary

We have introduced the three components of STL.
- Containers are classes for storing items of different types.
- Iterators are classes for indexing items in containers.
- Algorithms are common operations used to manipulate items in containers.

STL is **efficient** and **robust**. In most cases, they provide better implementations for algorithms and data structures than most programmers can achieve. They also allow programmers to focus on issues specific to their applications. Hence, it is recommended to use STL instead of re-implementing the functionalities whenever possible.

In many cases, the same functionalities can be obtained using different containers or algorithms. Programmers should understand the performance guarantees of different containers and choose the best one for their applications.

4 Further readings

STL is covered in Chapter 15.3 of "C++ How to Program, Ninth Edition".

More information about STL is available at:

http://www.cplusplus.com/reference/stl/
http://www.cplusplus.com/reference/algorithm/