C++ Programming II

Standard Template Library - STL
Introduction to Iterators and Algorithms

C++ Programming II
September 23, 2018

Prof. Dr. P. Arnold
Bern University of Applied Sciences

Agenda

- **Iterators**

Agenda

- **Iterators**
- **Algorithms**

Iterators

STL - Iterators

Operators

- An iterator is an object that can traverse (iterate over) a container class without the user having to know how the container is implemented
- An iterator is best visualized as a pointer to a given element in the container, with a set of overloaded operators to provide a set of well-defined functions
  1. `operator*` - Dereferencing the iterator returns the element that the iterator is currently pointing at
  2. `operator++` - Moves the iterator to the next element in the container. Most iterators also provide `operator--` to move to the previous element
  3. `operator==` and `operator!=` - Basic comparison operators to determine if two iterators point to the same element. To compare the values that two iterators are pointing at, dereference the iterators first, and then use a comparison operator
  4. `operator=` - Assign the iterator to a new position (typically the start or end of the container's elements). To assign the value of the element the iterator points at, dereference the iterator first, then use the assign operator

STL-Iterators

Iterators are the bridge that allow the STL-algorithms to work with STL-containers

STL - Iterators

Common Members

- Each container includes four basic member functions:
  1. `begin()` - returns an iterator representing the beginning of the elements in the container
  2. `end()` - returns an iterator representing the element just past the end of the elements
  3. `cbegin()` - returns a const (read-only) iterator representing the beginning of the elements in the container
  4. `cend()` - returns a const (read-only) iterator representing the element just past the end of the elements
- Note: `end()` points to the element just past the end! This is done primarily to make looping easy

STL - Iterators

Common Members

- Each container includes four basic member functions:
  1. `begin()` - returns an iterator representing the beginning of the elements in the container
  2. `end()` - returns an iterator representing the element just past the end of the elements
  3. `cbegin()` - returns a const (read-only) iterator representing the beginning of the elements in the container
  4. `cend()` - returns a const (read-only) iterator representing the element just past the end of the elements
- Note: `end()` points to the element just past the end! This is done primarily to make looping easy
- All containers provide (at least) two types of iterators:
  1. `container::iterator` provides a read/write iterator
  2. `container::const_iterator` provides a read-only iterator
- See examples ...

STL - Iterators

Example - vector

```cpp
#include <iostream>
#include <vector>
int main()
{
    using namespace std;

    vector<int> vect;
    for (int i=0; i < 6; i++)
    {
        vect.push_back(i);
    }

    vector<int>::const_iterator it; // read-only iterator
    it = vect.cbegin(); // assign it to the start of the vector
    while (it != vect.cend()) // while not at end
    {
        cout << *it << " "; // print value it points to
        ++it; // and iterate to the next element
    }

    cout << endl; // Output: 0 1 2 3 4 5
}
```

STL - Iterators

Example - `list`

```cpp
#include <iostream>
#include <list>
int main()
{
    using namespace std;

    list<int> li;
    for (int i=0; i < 6; i++)
    {
        li.push_back(i);
    }

    list<int>::const_iterator it; // declare an iterator
    it = li.cbegin(); // assign it to the start of the list
    while (it != li.cend()) // while not at end
    {
        cout << *it << " "; // print the value it points to
        ++it; // and iterate to the next element
    }

    cout << endl; // Output: 0 1 2 3 4 5
}
```

Note: The code is almost identical to the vector case, even though vectors and lists have almost completely different internal implementations!

STL - Iterators

Example - set

```cpp
#include <iostream>
#include <set>
int main()
{
    using namespace std;

    set<int> myset;
    myset.insert(7);
    myset.insert(2);
    myset.insert(-6);
    myset.insert(8);
    myset.insert(1);
    myset.insert(-4);

    set<int>::const_iterator it; // declare an iterator
    it = myset.cbegin(); // assign it to the start of the set
    while (it != myset.cend()) // while not at end
    {
        cout << *it << " "; // print the value it points to
        ++it; // and iterate to the next element
    }

    cout << endl; // Output: -6 -4 1 2 7 8
}
```

Note: Besides the creation, the code used to iterate through the elements of the set is essentially identical as before!

STL - Iterators

Example - map

```
#include <iostream>
#include <map>
#include <string>
int main()
{
    using namespace std;

    map<int, string> mymap;
    mymap.insert(make_pair(4, "apple"));
    mymap.insert(make_pair(2, "orange"));
    mymap.insert(make_pair(1, "banana"));
    mymap.insert(make_pair(3, "grapes"));
    mymap.insert(make_pair(6, "mango"));
    mymap.insert(make_pair(5, "peach"));

    map<int, string>::const_iterator it; // declare an iterator
    it = mymap.begin(); // assign it to the start of the vector
    while (it != mymap.end()) // while not at end
    {
        cout << it->first << "=" << it->second << " "; // print
        ++it; // and iterate to the next element
    }
    cout << endl;
    // Output: 1=banana 2=orange 3=grapes 4=apple 5=peach 6=mango
}
```

Note: Iterators make it easy to step through each of the elements of the container. You don't have to care at all how map stores its data!

STL - Iterators

Containers and Iterator Types

Containers can also be classified into the following two categories:

1. Array based containers: `vector`, `deque`
2. Node base containers: `list`, associative containers and unordered containers

Depending on the container categories, different iterators are available:

- **Random Access Iterator**: `vector`, `deque`, `array`

```
1 vector<int> vec;
2 auto itr = begin(vec);
3 itr = itr + 5;  // advance itr by 5
4 itr = itr - 4;
5 if (itr2 > itr1) // compare position
6 ++itr;   // pre inc. faster than itr++
7 --itr;
```

STL - Iterators

Containers and Iterator Types

Containers can also be classified into the following two categories:

1. Array based containers: `vector`, `deque`
2. Node base containers: `list`, associative containers and unordered containers

Depending on the container categories, different iterators are available:

- **Bidirectional Iterator**: `list`, `set/multiset`, `map/multimap`
- No random access, no advance by value, no compare!

```
1 list<int> li;
2 auto itr = begin(li);
3 ++itr;
4 --itr;
```

STL - Iterators

Containers and Iterator Types

Containers can also be classified into the following two categories:

1. Array based containers: `vector`, `deque`
2. Node base containers: `list`, associative containers and unordered containers

Depending on the container categories, different iterators are available:

- **Forward Iterator**: `forward_list`
- Can only be incremented

```
1 forward_list<int> fList;
2 auto itr = begin(fList);
3 ++itr;
```

- Unordered containers at least provide forward iterators

STL - Iterators

Containers and Iterator Types

Containers can also be classified into the following two categories:

1. Array based containers: `vector`, `deque`
2. Node base containers: `list`, associative containers and unordered containers

Depending on the container categories, different iterators are available:

- **Input Iterator**: read and process values while iterating forward (read-only)
- **Output Iterator**: output values while iterating forward (write-only)
- These two iterators provide a subset of forward iterator

```
1 // Input
2 int x = *itr;
3 // Output
4 *itr = 100;
```

STL - Iterators

Predefined Iterators - Iterator Adaptors

- Iterator Adaptors do more than just iterating:
  1. Insert Iterator
  2. Stream Iterator
  3. Reverse Iterator

STL - Iterators

Predefined Iterators - Iterator Adaptors

- **Insert Iterator**

```
1  // 1. Insert Iterator:
2  vector<int> vec1 = {4,5};
3  vector<int> vec2 = {12, 14, 16, 18};
4  vector<int>::iterator it = find(vec2.begin(), vec2.end(), 16);
5
6  insert_iterator<vector<int>> i_itr(vec2,it);
7  copy(vec1.begin(),vec1.end(),   // source
8       i_itr);                    // destination
9
10 //vec2: {12, 14, 4, 5, 16, 18}
11 // Other insert iterators:
12 // back_insert_iterator,
13 // front_insert_iterator
```

- front and back inserter iterators insert at front and back, respectively

STL - Iterators

Predefined Iterators - Iterator Adaptors

- **Stream Iterator**

```
1  // 2. Stream Iterator: ()
2  vector<string> vec4;
3  copy(istream_iterator<string>(cin), istream_iterator<string>(),
4       back_inserter(vec4));
5
6  copy(vec4.begin(), vec4.end(),                  // source
7       ostream_iterator<string>(cout, " "));      // dest
8
9  // Combine both
10 copy(istream_iterator<string>(cin), istream_iterator<string>(),
11      ostream_iterator<string>(cout, " "));
```

- Stream iterators are used to iterate through the data to and from a stream
- The default constructor `istream_iterator<string>()` represents the end of a stream
- `back_inserter(vec4)` is a function which returns a back inserter iterator from vector `vec4`

STL - Iterators

Predefined Iterators - Iterator Adaptors

- **Reverse Iterator**

```
1  // 3. Reverse Iterator:
2  vector<int> vec = {4,5,6,7};
3  reverse_iterator<vector<int>::iterator> ritr;
4
5  for(ritr = vec.rbegin(); ritr != vec.rend(); ritr++)
6      cout << *ritr << endl;    // prints: 7 6 5 4
```

- Traverse container in reverse order
- STL provides `rbegin()` and `rend()`

Algorithms

STL - Algorithms

Iterators and Algorithms

- Algorithms are mostly loops!
- Whenever you see a while or a for loop in your code, you should consider to replace by a STL-Algorithm
- STL makes your code:
  1. **more efficient**
  2. **less buggy**
  3. **more readable**
  4. **more clean**

STL - Algorithms

Notes about Algorithms

```
1  vector<int> vec = { 4, 2, 5, 1, 3, 9};
2  vector<int>::iterator itr = min_element(vec.begin(), vec.end());
3  // itr -> 1
4
5  // 1) Algorithm process ranges in a half-open way: [begin, end)
6  sort(vec.begin(), itr);  // vec: { 2, 4, 5, 1, 3, 9}
7
8  reverse(itr, vec.end());  // vec: { 2, 4, 5, 9, 3, 1}
9  // itr => 9 , points to old location!
10
11 // 2) With copy you have to provide enough space for destination
12 vector<int> vec2(3);
13 copy(itr, vec.end(),  // Source
14      vec2.begin());   // Destination
15 //vec2 needs to have at least space for 3 elements.
16
17 // 3) Use insert to overcome this safety issue!
18 vector<int> vec3;
19 copy(itr, vec.end(), back_inserter(vec3));  // Inserting
20 // back_insert_iterator      Not efficient, since element-wise
21
22 vec3.insert(vec3.end(), itr, vec.end());  // Efficient and safe
```

- Algorithms mostly work on ranges, represented by a pair or more iterators
- STL provides many ways of doing the same thing

STL - Algorithms

Notes about Algorithms

```
1  // 4) Algorithm with function
2  bool isOdd(int i)
3  {
4      return i%2;
5  }
6
7  int main()
8  {
9      vector<int> vec = {2, 4, 5, 9, 2};
10     auto itr = find_if(vec.begin(), vec.end(), isOdd);
11     // itr -> 5
12 }
13
14
15 // 5) Algorithm with native C++ array
16 int arr[4] = {6,3,7,4};
17 sort(arr, arr+4);
```

- Iterators are a pure abstract concept
- A raw pointer can be think of an iterator!

Thank You

Questions

???