C++ Standard Template Library (STL)

CSCI 1061U — Programming Workshop 2

faisal.qureshi@uoit.ca
Faculty of Science
University of Ontario Institute of Technology
STL

• Containers
  • Sequential containers
  • Associative containers
  • Container adapters

• Generic algorithms

• Iterators
  • Reverse iterators
  • Const and non-const (mutable) iterators
Sequential Containers

• Sequential containers are class that can be used for storing other items
  • DynamicArray — `std::vector<T>`
  • LinkedList — `std::list<T>`
  • Deques — `std::deque<T>`
Sequential Containers

Allocating a vector of int
```
#include <vector>
std::vector<int>
```

Allocating a list of strings
```
#include <list>
#include <string>
std::vector<std::string>
```

Allocating a deque of vectors of strings
```
#include <deque>
#include <vector>
#include <string>
std::deque<std::vector<std::string> >
```
Iterators

- C++ STL containers (both sequential and associative) define “helper classes,” called *iterators*, to help iterate over each item in the container.
Iterator Example

```cpp
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v;
    v.push_back(1);
    v.push_back(2);
    v.push_back(3);

    std::vector<int>::iterator i;
    for (i = v.begin(); i != v.end(); ++i) {
        std::cout << *i << std::endl;
    }

    return 0;
}
```

iterator i is a pointer to an element stored in the container.
Array Traversal

Case 1

```cpp
#include <iostream>

int main()
{
    int a[] = {1, 3, 5, 7, 9};

    for (int i=0; i<5; ++i) {
        std::cout << a[i] << std::endl;
    }

    return 0;
}
```

Case 2

```cpp
#include <iostream>

int main()
{
    int a[] = {1, 3, 5, 7, 9};

    for (int* i = &a[0]; i != &a[4]; ++i) {
        std::cout << *i << std::endl;
    }

    return 0;
}
```
std::set<T>

- Implements sets, each value can only occur once
- Efficient at testing membership O(log n)
  - Use `find()` method
- The value type must have `<` operator
- Iterator is available
- Addition, deletion is supported
Iterators

• Iterator variable has the same semantics as a pointer to the stored element

• Deference (use operator *) to get the actual value
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v;
    v.push_back(1);
    v.push_back(2);
    v.push_back(3);

    std::vector<int>::iterator i;
    for (i = v.begin(); i != v.end(); ++i)
    {
        std::cout << *i << std::endl;
    }

    return 0;
}
Reverse Iterator

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

int main()
{
    list<string> names;
    names.push_back("john");
    names.push_back("amanda");

    list<string>::reverse_iterator i;
    for (i = names.rbegin(); i != names.rend(); ++i) {
        cout << *i << endl;
    }

    return 0;
}
```
Iterators

Different kinds of iterator

`std::vector<int>::iterator`
`std::vector<int>::reverse_iterator`
`std::vector<int>::const_iterator`
`std::vector<int>::const_reverse_iterator`

Methods for initializing and checking iterators

`begin()`, `end()`
`cbegin()`, `cend()`
`rbegin()`, `rend()`
`crbegin()`, `crend()`
Containers

• Sequential containers are class that can be used for storing other items
  
  • DynamicArray — `std::vector<T>` \(O(1)\)
  
  • LinkedList — `std::list<T>` \(O(n)\)
  
  • Deques — `std::deque<T>` \(O(1)\)
Containers

• Sequential containers are class that can be used for storing other items

  • DynamicArray — `std::vector<T>`
  • LinkedList — `std::list<T>`
  • Deques — `std::deque<T>`

Adding an element

End

Anywhere

Both ends
Containers

- Sequential containers are a class that can be used for storing other items
  - DynamicArray — `std::vector<T>` \( \mathcal{O}(n) \)
  - LinkedList — `std::list<T>` \( \mathcal{O}(1) \)
  - Deques — `std::deque<T>` \( \mathcal{O}(n) \)

Deleting an element
Associative Containers

• Sequential arrays provide no meaningful way to index the stored data

• Example
  • std::vector index elements with integers 0, 1, 2, ... that may have no relationship to the stored data

• It would be nice if we can do the following
  • ages["earth"] = 4530000000;
  • cout << lastnames["John"] << endl;
Associative Container

**std::map<K,V>**

- Store (key, value) **std::pair<K,V>**
- Unique key
- Supported operations
  - insertion
  - removal
  - lookup

No specific order (no element 0)
#include <map>
#include <iostream>
#include <string>
using namespace std;

int main()
{
    map<string, int> super_heros;

    super_heros["batman"] = 32;
    super_heros["wolverine"] = 137;
    super_heros["jean gray"] = 25;
    super_heros["superman"] = 35;

    map<string, int>::iterator i;
    for (i = super_heros.begin(); i != super_heros.end(); ++i)
    {
        cout << "Age of " << i->first << " is " << i->second << endl;
    }

    return 0;
}
#include <map>
#include <iostream>
#include <string>
using namespace std;

int main()
{
    map<string, int> super_heros;

    super_heros["batman"] = 32;
    super_heros["wolverine"] = 137;
    super_heros["jean gray"] = 25;
    super_heros["superman"] = 35;

    map<string, int>::iterator i = super_heros.find("batman");
    if (i != super_heros.end()) {
        cout << "Batman is " << i->second << endl;
    }

    return 0;
}
Associative Container

std::map<K,V>

- Maps are sometimes referred to as *hashes* or *dictionaries*
- **Key** type should have `<` operator
  - Use std::string and not char[]
- **Value** type should have default constructor
- Provides [] operator for both insertion and retrieval
Associative Container

`std::map<K,V>`

- Efficient key lookup $O(\log n)$
- Use `find()` function for key lookup
  - Returns an iterative to the (key,value) pair if the key is found
  - Otherwise returns an iterator equal to `end()`
Associative Container

`std::multimap<K,V>`

- Store (key, value) `std::pair<K,V>`
- Unique key
- Supported operations
  - insertion
  - removal
  - lookup

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>batman</td>
<td>32</td>
</tr>
<tr>
<td>wolverine</td>
<td>137</td>
</tr>
<tr>
<td>jean gray</td>
<td>25</td>
</tr>
<tr>
<td>superman</td>
<td>35</td>
</tr>
</tbody>
</table>
```

No specific order (no element 0)
Container Adapters

- Container adapters are template classes that are implemented “on top of” other classes
  - `std::stack`
  - `std::queue`
  - `std::priority_queue`
Container Adapters

- Adapter template classes have “default” underlying containers
  - `std::stack` is implemented using `std::deque`
- It is possible to specify a different underlying container
  - `std::stack<int, std::vector<int>>`
Algorithms

- Strives to be optimally efficient
- Non-modifying sequence algorithms
- Modifying algorithms
Non-modifying Sequence Algorithms

• Template functions operate on containers
• Does not modify the content of that container

• Example
  • `std::find()`
Modifying Sequence Algorithms

• STL functions that can change the content of a container

• Adding or removing elements from a container may invalidate an iterator

• `std::list` guarantees that no iterator will be changed

• `std::vector` and `std::deque` do not provide any guarantees
STL Algorithms

```
#include <iostream>
#include <string>
#include <map>

using namespace std;

int main()
{
    map<int, string> persons = {{10, "Walker"}, {43, "Judy"}};
    for (auto i : persons) {
        cout << "id = " << i.first << " name = " << i.second << endl;
    }

    return 0;
}

Compile as follows

g++ -std=c++11 ranged-for.cpp
```