

Many data structures

Lecture 14

The Standard Template Library

Apart from list, we have many data-structures that we want. These data structures varies in their efficiencies for different operations. We want templates of them, too.

But it is tedious having to define all of them. Instead, the C++ standard library provides

Reference:

- The C++ Programming Language, Section 17.1–17.4, 18.2.

PMOOP(0396A)

PMOOP(0396A)-14.1

Why don't write them yourselves?

Why we want to stick to a **single implementation**, rather than writing our own container everytime we need it?

Benefit of reuse:

It is less tedious because you don't need to reimplement it everytime.

And the result is more correct: standard library code is probably much better debugged than your implementation.

And the result is faster: It probably implements more performance boosting algorithms.

Benefit of standardization:

Inter-operate with code written by other people better, since they use the same standard library.

Standard C++ template functions works with all standard data structures (containers).

PMOOP(0396A)-14.2

PMOOP(0396A)-14.3

Scanning through a container

We can **walk through an array, vector or deque** like this:

```
vector<int> v;
// fill the vector v
for (int i = 0; i < v.size(); ++i) // Must know size by yourselves for arrays
    cout << v[i]; // Or other processing
```

But we can't do this for other types: the keys are not consecutive integers.

To allow **walking through a container** of any type, we use **iterators**.

It is just the same concept as a node of a linked list, so don't panic.

An iterator **points to an element** of the container. There're 2 operations:

- Retrieving the content.
- Getting an iterator to nearby elements of the same container.

PMOOP(0396A)-14.4

Example use

```
#include <string>
#include <list>
using namespace std;
int main() {
    list<float> lst;
    for (int i = 0; i < 10; ++i)
        lst.push_back(i);
    // Here, begin() returns the start of container,
    // end() returns one-past-end of container
    list<float>::iterator it;
    for (it = lst.begin(); it != lst.end(); ++it)
        cout << *it << endl;
    list<float>::reverse_iterator it2;
    for (it2 = lst.rbegin(); it2 != lst.rend(); ++it2)
        cout << *it2 << endl;
}
```

Note that iterators looks pretty like pointers. In fact, **pointers are conceptually the iterators for arrays**.

PMOOP(0396A)-14.5

Apart from linked lists, we want other **data structures** to **hold our objects** of different types, for efficiency of different operations.

- List:** very fast to add and delete elements, no matter where is the element. But difficult to find the n-th element of a list.
- Vectors:** like arrays. Very efficient on indexing, and fast on adding and deleting elements at the end as well. Slow to add or delete elsewhere.
- Deque:** like vectors, but also allow adding and deleting at beginning. Why we still need vector? Since this makes it a bit slower for indexing.
- Sets:** a container which is **not indexed**, so the structure either have an element or doesn't have it.
- Maps:** a container which is **index by any comparable types**, not just integers. This is actually a binary search tree.

Sometimes we want to make a map that index using integers, just because the index is not consecutive numbers starting from 0.

What nearby element we can find?

Different type of iterators consider different things as “nearby” elements:

- All iterators allow the ++ operator so that you can find the next element. But few provides just that.
An iterator that provide just ++ is said to be forward iterators.
- A **bi-directional** iterator provides — operator so that you can find the previous element. List, set and map iterators are bi-directional.
- A **random-access** iterator provides + and – operators so that you can find an element of a fixed offset from it (e.g., it+10). Vector and deque iterators are random-access.
- So... what iterators is forward? They are less common iterators like the one which iterates through an istream or ostream.
But they are seldom useful, so we will skip it.

PMOOP(0396A)-14.6

Example

```
#include <deque>
#include <iostream>
using namespace std;
int main() {
    deque<int> dq;
    for (int i = 0; i < 10; ++i)
        dq.push_back(i);           // Add at the end
    for (int i = 0; i < 5; ++i)
        dq.push_front(-i);        // Add at the front
    for (int i = 5; i < 10; ++i)
        dq.insert(dq.begin()+10, -i); // Add at the 10-th position
    for (int i = 0; i < dq.size(); ++i)
        cout << dq[i] << endl;    // Print all out
}
```

If we use vector instead: won't compile, since *push_front* is not defined.
The defense is that it is not efficient, so don't allow it to be done carelessly.

PMOOP(0396A)-14.8

Example

```
#include <list>
#include <iostream>
using namespace std;
int main() {
    list<float> list1, list2;
    list<float>::iterator it1, it2;
    for (int i = 0; i < 10; ++i) {
        list1.push_back(i);
        list2.push_back(-i);
    }
    it1 = list1.begin();
    ++it1; ++it1; // now at 2nd place of list1
    it2 = list2.begin();
    for (int i = 0; i < 5; ++i) ++it2; // now at 5th place of list2
    list1.splice(it1, list2, it2, list2.end()); // steal 5 elements from list2
    list1.reverse();
    for (it1 = list1.begin(); it1 != list1.end(); it1++)
        cout << *it1 << endl;
}
```

PMOOP(0396A)-14.10

Vector, deque

We start the tour to containers by two simple ones: **vector** and **deque**.

- Vector and deque has one important template parameter, which is the type of each element. Need just a **copy-constructor**.
This is the case for any container element-type except set.
- They share the property that they may be **indexed** using the [] operator, with integers from 0 to *size()*-1.
For any container, *size()* returns the number of elements stored.
- The [] operator won't do **range checking**. The *at* function (e.g., *v.at(5)*) does that. You can make a derived class which check [] as well.
- The *push_back()* and *pop_back()* function add and delete at the end.
- Deques also has *push_back()* and *pop_front()* for changing the front. And in doing so, changing the indexing of the elements as well.
- One can also use *insert()* and *erase()* to change other places.
But much slower than functions like *push_back()*.

PMOOP(0396A)-14.7

List

- The list container is a **doubly-linked** list.
- No [] operator is defined by default.
Again, less easy to do inefficient thing.
- It has *push_front*, *push_back*, *pop_front*, *pop_back*, *insert* and *erase*.
- It also has *sort* to sort lists, and *merge* to combine 2 sorted lists...
This is specific to list: to sort containers which support random access, we can use standard template functions.
- ... and also has *reverse()* to reverse the list...
- ... and *splice()*, which allow moving elements from another list into a specific position of it (specified by an iterator).
- And, all these are done **without copying** list elements.
Possible only for lists. Memory of other containers can't be adopted by another.

PMOOP(0396A)-14.9

Sets

Next we will learn STL **set**.

- Rather than saying that it has no **index**, we should really think that it has no **data** associated, so the only thing stored are the indices.
- The template parameter of a set **must be comparable**.
E.g., you can't have set of ostreams. But you can have a set of vectors, sets or maps: lexicographical compare is used.
- The ordering of a set is **always from small to large**. We can't insert at a particular location, so insertion syntax is a bit different.
- There is an *insert()* function, but it doesn't take an iterator argument. No *push_back*, *pop_back*, etc.
- The *erase()* function can take either an element or an iterator.
- The *find()* function locate an argument and give an iterator. It returns *end()* if not found.

PMOOP(0396A)-14.11

Example

```
#include <set>
#include <iostream>
using namespace std;
int main() {
    set<float> s;
    for (int i = 0; i < 5; ++i)
        s.insert(i*i);           // Insert elements
    for (int i = 0; i < 5; ++i)
        if (s.find(i) != s.end()) // Find an element
            cout << "Found " << i << endl;
    s.erase(4);                 // Erase an element by its value
    set<float>::iterator it;
    for (it = s.begin(); it != s.end(); ++it)
        cout << "Containing " << *it << endl;
}
```

PMOOP(0396A)-14.12

Maps

- If we **stick a value into each element** of a set, we get a map.
- The original index is usually called a **key**. So the map stores **key-value pairs**. The type is `pair<KeyType, ValueType>`, like this:

```
template<class _T1, class _T2>
struct pair {
    typedef _T1 first_type; // define pair::first_type
    typedef _T2 second_type;
    _T1 first;
    _T2 second;
    // and constructor
};
```

- The key must be a comparable type, while the value just needs to be able to copy-construct.
- We can use the convenient [] notation to access a map. Apart from that, all the operations of `set` still applies.

PMOOP(0396A)-14.13

Example

```
#include <map>
#include <string>
#include <iostream>
using namespace std;
int main() {
    map<string, float> m;
    for (int i = 0; i < 10; ++i) {
        string s(i, 'a'); // i copies of 'a'
        m[s] = i;         // the easy way to access the map
    }
    m.insert(pair<string, float>("aaaB", 10.5)); // insert one more
    m.erase(m.find("aaa"));
    map<string, float>::iterator mit;
    for (mit = m.begin(); mit != m.end(); ++mit)
        cout << mit->first << " ==> " << mit->second << endl;
}
```

PMOOP(0396A)-14.14

Before we move on...

- It is possible to insert and erase many elements at a time. The “many elements” is specified by a pair of iterators. E.g.,

```
void f(vector<int>& v, list<int>& lst) {
    v.insert(v.end(), lst.begin(), lst.end()); // insert all elements of lst
    v.erase(v.begin()+5, v.begin()+9); // remove v[5] to v[8]
}
```

- Multiset and multimap are similar to set and map, but **can hold many keys that compares the same**. The `upper_bound` and `lower_bound` functions tells you the start and end of all elements of a key. E.g.,

```
void erase_all(multimap<string, int>& mm, string key) {
    mm.erase(mm.lower_bound(key), mm.upper_bound(key));
}
```

- Before you think that you don't need to take the DSA course, you should realize that everything make sense because you already know how to implement them.

PMOOP(0396A)-14.15

The unifying concept: iterator

- Different containers have different interface, because they **accommodate different needs**.
- But **iterators syntax is the same** for all containers. This allows us to write very generic code.
- E.g., to copy some elements of a container to the end of a vector vec:

```
template<class ElementType, class ForwardIterator>
copy_to_end(vector<ElementType>& vec,
            ForwardIterator i1, ForwardIterator i2)
{
    for (ForwardIterator t = i1; t != i2; ++t)
        vec.push_back(*t);
}
```

E.g., you can copy all elements of a list `lst` to a vector `vec` by `copy_to_end(vec, lst.begin(), lst.end())`. This works **for any container**, as long as the elements are **assignment-compatible**.

PMOOP(0396A)-14.16

Standard algorithms

The generality comes from the use of **templates**: E.g., we can have a template that require the argument to be any random-access iterator.

However, just like other **useful** and **general** things, they are **done by some library function**. E.g.,

- `copy(beginIter, endIter, outIter)`: copy everything between `fromIter` and `endIter` to another container, starting at `outIter` towards the end.
- `copy_backward(beginIter, endIter, outIter)`: similar, but copy things to `outIter` towards the beginning.
- `sort(beginIter, endIter)`: sort elements between `beginIter` and `endIter`.
- `find(beginIter, endIter, value)`: find a specific value in the elements between `beginIter` and `endIter`.

Unlike the `find()` member function which finds keys, this finds values. This don't work for map, though.

PMOOP(0396A)-14.17

Example

```
#include <iostream>
#include <vector>
#include <algorithm> // Use standard algorithms
using namespace std;
int main() {
    vector<int> v, v2;
    for (int i = 0; i < 10; ++i)
        v.push_back(i); // 0 1 2 ... 9
    for (int i = 0; i < 10; ++i)
        v2.push_back(i * 3); // 0 3 6 ... 27
    copy_backward(v2.begin()+2, v2.end()-2, v.end()-2); // result?
    sort(v.begin(), v.end()); // result?
    for (int i = 0; i < 10; ++i)
        if (find(v.begin(), v.end(), i) != v.end())
            cout << i << " present.\n";
}
```

PMOOP(0396A)-14.18

Modifying iterators

- But... that doesn't solve our problem, because none of the algorithm can **add elements to the container**.
- **Except** that the iterator might add or remove elements anyway.
There is no requirement that an iterator cannot add things to a container. We just haven't seen one.
- There are 3 such iterators, called inserters, which adds elements to a container c:
 - `back_inserter(c)`: add a new element at the end whenever written.
 - `front_inserter(c)`: add a new element at the front whenever written. (Only if `push_front` defined)
 - `back_inserter(c, iter)`: add a new element just after `iter` whenever written.

PMOOP(0396A)-14.19

Example

```
#include <iostream>
#include <vector>
#include <set>
#include <algorithm>
#include <iterator> // Defines the inserters
using namespace std;
int main() {
    vector<int> v;
    set<int> s;
    for (int i = 0; i < 10; ++i) {
        v.push_back(i);
        s.insert(i * 5);
    }
    copy(s.begin(), s.end(), back_inserter(v)); // Add to the end of list
    for (int i = 0; i < v.size(); ++i)
        cout << v[i] << endl;
}
```

PMOOP(0396A)-14.20