CS100 Recitation 8

teafrogsf

SIST

May 1, 2022

Content

What Is STL

STL, namely **S**tandard **T**emplate **L**ibrary, provides many useful template classes for programmers to use and study.

What Is STL

STL, namely **S**tandard **T**emplate **L**ibrary, provides many useful template classes for programmers to use and study.

“长久以来，软件界一直希望建立一种可重复运用的东西，以及一种得以制造出‘可重复运用的东西’的方法，让工程师/程序员的心血不至于随时间迁移、人事异动、私心欲念、人谋不臧而烟消云散。”

——*The Annotated STL Sources, Ch. 1*

What Is STL

STL, namely **S**tandard **T**emplate **L**ibrary, provides many useful template classes for programmers to use and study.

“长久以来，软件界一直希望建立一种可重复运用的东西，以及一种得以制造出‘可重复运用的东西’的方法，让工程师/程序员的心血不至于随时间迁移、人事异动、私心欲念、人谋不臧而烟消云散。”

——*The Annotated STL Sources, Ch. 1*

STL is contained in the C++ standard library since 1998. There are also many implementations of STL.

Content in STL

There is quite a variety of content in STL, but in general we can divide them into the following categories:
allocators, iterators, sequence containers, associative containers, algorithms, functors and adapters.

Content

What Is Iterators

'Iterator' is in fact an abstract design concept, and there is no entity that *directly* corresponds to this concept.

An iterator behaves like a pointer, in the sense that it is also a smart pointer (similar to `auto_ptr` in the C++ standard library).

Why Abstract

This is because to complete an iterator designed for a particular class would inevitably make extensive use of the implementation details in the class.
Since this is the case, iterators designed directly and exclusively for each class can instead encapsulate the implementation details well.
Therefore, there is a dedicated iterator for each type of STL container.

Iteration

```
vector<int>v;
for(vector<int>::reverse_iterator j = v.rbegin(); j != v.rend
    (); ++j)
    cout << *j << " ";
```

Category of Iterators

Iterators are usually divided into five categories.

Input iterator: the object referred to by this iterator is read-only.

Output iterator: the object referred to by this iterator is write-only.

Forward iterator: allows writable methods (e.g. `replace()`) to be read and written on the interval formed by this iterator. The first three iterators all support `operator++`.

Bidirectional iterator: A bidirectional moveable iterator based on forward iterator. It supports `operator--`.

Random access iterator: It supports all pointer operations, including `p+n`, `p-n`, `p[n]`, `p1-p2`, `p1<p2`.

Content in Iterators

In addition to the previously mentioned operations, iterators usually need to support `operator*`, `operator->`, `begin()`, `end()`, `operator==`, `operator!=` and so on.
Also, the use of iterators may often require knowledge of many associated types, such as the type of the object referred to, the type of the reference, and of course the type of itself.
In order to obtain the exact associated types of an iterator operation, the implementation of iterators depends on *traits*.

Content

What Is Containers

Containers are the first thing most people think of when they think of STL.

As the name implies, a container is a tool for filling things. The STL contains a considerable number of containers, here we will only focus on `vector`, `list` and `map`.

Sequential Containers

Sequential containers, i.e. containers whose elements are *ordered*, but not necessarily *sorted*. C++ itself has a built-in container array, while STL additionally provides `vector`, `list`, `deque`, `stack`, `queue`, `priority_queue`, and so on. Of course, part of this involves algorithm implementations, which we will not discuss in CS100.

`vector`

Unlike traditional arrays, vectors implement automatic space expansion. It also provides a number of useful functions.

Preparations

Here are some code in the source code of `vector`. It has been simplified to some extent, so it may not be exactly the same as what is in the real STL.

Preparations

Using pointers directly as iterators is an easy to understand way of writing, but it is susceptible to security problems (users can access addresses directly through pointers). Today's C++ uses a more safe way.

```
template<class T, class Alloc = alloc> // template class
class vector
{
public:
    typedef T* iterator; // Actually there are many 'typedef'
        s in the original class, for simplification we only
        keep this one
    /*...*/
}
```

Preparations

```
protected:
    iterator start; // Indicates the head of the space
        currently in use
    iterator finish; // Indicates the tail of the space
        currently in use
    iterator end_of_storage; // Indicates the tail of
        currently available space
    /*...*/
```

Preparations

```
protected:
    void insertion_aux(iterator position, const T& x);
    void fill_initialize(size_t n, const T& value)
    {
        start = allocate_and_fill(n, value); // No need to
            master
    }
    /*...*/
```

Properties

```
public:
    iterator begin(){return start;}
    iterator end(){return finish;}
    size_t size() const{return size_t(end() - begin());}
    size_t capacity() const{return size_t(end_of_storage -
        begin());}
    bool empty() const{return begin() == end();}
    T& operator[](size_t n){return *(begin() + n);}
    /*...*/
```

Ctors

```
public:
    vector(): start(0), finish(0), end_of_storage(0){}
    vector(size_t n, const T& value){fill_initialize(n, value
        );} // given initial value
    vector(int n, const T& value){fill_initialize(n, value);}
    vector(long n, const T& value){fill_initialize(n, value)
        ;}
    explicit vector(size_t n){fill_initialize(n, T());}
    /*...*/
```

Dtor

```
public:
    ~vector()
    {
        destroy(start, finish); // No need to master
        deallocate(); // No need to master
    }
    /*...*/
```

Basic Functions

```
public:
    T& front(){return *begin();}
    T& back(){return *(end() - 1);}
    // Note that they also require const overloading. Why?
    /*...*/
```

Basic Functions

```
public:
    void push_back(const T& x)
    {
        if(finish != end_of_storage)
        {
            construct(finish, x); // No need to master
            ++finish;
        }
        else
            insert_aux(end(), x);
    }
    /*...*/
```

Basic Functions

```
public:
    void pop_back()
    {
        --finish;
        destroy(finish);
    }
    /*...*/
```

Basic Functions

Note that there is a return value here, so if you are going to use erase while iterating, be sure to pay attention to whether you have the correct iterator.

```
public:
    iterator erase(iterator position)
    {
        if(position + 1 != end())
            copy(position + 1, finish, position); // std::
                copy, No need to master
        --finish;
        destroy(finish);
        return position;
    }
```

Basic Functions

```
public:
    void resize(size_t newsize, const T &x)
    {
        if(new_size < size())
            erase(begin() + new_size, end());
        else
            insert(end(), new_size - size(), x);
    }
    void resize(size_t newsize){resize(new_size, T());}
    void clear(){erase(begin(), end());}
    void insert(iterator position, size_t n, const T& x);
```

`insert_aux` and `insert`

These two functions are too complex to show.
In general, the logic of both `insert_aux` and `insert` is to determine whether the inserted element exceeds the capacity, and if so, to expand the capacity to max(twice the original length, original length + number of new elements).
They differ slightly in their specific implementation.

Associative Containers

The so-called associative containers, where each element has a key and a value, place the element in the appropriate position (with some specific rules) when it is inserted to them.

Associative Containers

The so-called associative containers, where each element has a key and a value, place the element in the appropriate position (with some specific rules) when it is inserted to them.
The internal structure of associative containers is more complex and we will not go into the details. However, we need to know how to use them correctly.

Practice: LinkedMap

Behavior:

- A generic container just like map (associative)
- Ordered based on insertion order instead of comparison
- Approximate constant time insertion, deletion, look up and iteration

Implementation:

- Composition of a `std::list` and a `std::unordered_map`
- Define an appropriate node structure

Practice: LinkedMap

Interface:

```
iterator begin();
iterator end();
bool empty();
int size();
void erase(const K& k);
//void set(const K& k, const V& v);
V& get(const K& k); // remember const overloading!
void clear();
```

Practice: LinkedMap

Test:

```
LinkedMap<int,double> linkedMap;
linkedMap.get(1) = 1.0;
linkedMap.get(2) = 2.0;
linkedMap.get(0) = 0.0;
LinkedMap<int,double>::iterator_t it = linkedMap.begin();
while(it != linkedMap.end())
{
    std::cout << it->second << " ";
    it++;
}
  std::cout << "\n";
```

Content

Algorithms in STL

STL provides a large number of convenient algorithms for direct use by programmers. Many of the algorithms have relatively clever implementations. But in CS100 we don't need to care how they are implemented, we just need to learn to use them.

For Example

You can use `sort(a,a+n)` to sort $a[1, 2, \ldots, n-1]$, or `sort(a.begin(),a.end())` to sort a vector from the first element to the last one.

You can use `reverse(a,a+n)` to reverse $a[1, 2, \ldots, n-1]$.

Uniformity of interfaces

Why always ‘Left closed, right open’?

Uniformity of interfaces

Why always ‘Left closed, right open’?

Explanation by GKxx: Right minus left is the length, saving the complexity of the operation.

Content

What Is Template

Template supports generic programming.

Template uses generic data type (usually represented by `T`), which is replaced by concrete type at compile type. Template enables “on-the-go”construction of a member of a family of functions and classes that perform the same operation on different data types.

Form

```
template <class T, ··· >
returntype function_name (arguments)
{
    /* Body of function */
}
```

Example

```
template <typename T>
T const& Max (T const& a, T const& b)
{
    return a < b ? b : a;
}
```

Form

```
template <class T1, class T2, .....>
class class_name
{
...
    T1 m_data1; // data items of template type
    // functions of template argument
    void func1 (T1 a, T2& b);
    T1 func2 (T2* x, T2* y);
};
```

Recall STL

All members in STL are constructed using generic programming. Learning generic programming is therefore a very important step in the programming learning path in C++.

Template Metaprogramming

In fact, the use of templates is quite rich and complex, as exemplified by template metaprogramming.

Here is a code example.

Template Metaprogramming

```
template<int N>
struct binary
{static constexpr int value = binary<N / 10>::value << 1 | N
    % 10;};
template<>
struct binary<0>
{static constexpr int value = 0;};
static char array[binary<101>::value]; // Equivalent to:
    static char array[5]
```

Content

More For STL

The Annotated STL Sources, Jie Hou

Note that there may exist some difference between this book and current STL source code.

More For Template

If you are interested in it, maybe you can read these for more information:

https://blog.csdn.net/qq_35637562/article/details/55194097

https://zhuanlan.zhihu.com/p/378356824