STL 标准模板库

标准模板库（英文：Standard Template Library，缩写：STL），是一个 C++软件库，也是 C++标准程序库的一部分。其中包含 5 个组件，分别为算法、容器、迭代器、函数、适配器。

模板是 C++程序设计语言中的一个重要特征，而标准模板库正是基于此特征。标准模板库使得 C++编程语言在有了同 Java 一样强大的类库的同时，保有了更大的可扩展性。

目录：

Vector 和 list 的区别

String 类型

Stack 和 queue

一 顺序容器的初始化

顺序容器主要是 vector 和 list

他们的初始化方式有五种

1.直接初始化一个空的容器

2.用一个容器去初始化另一个容器

3.指定容器的初始大小

4.指定容器的初始大小和初始值

5.用一对迭代器范围去初始化容器

例子如下：

```
#include <iostream>
#include <string>
#include <vector>
#include <list>

using namespace std;

int main(int argc, char **argv) {
    vector<string> vec;  //构造一个空数组
    vec.push_back("hello");
    vec.push_back("world");

    vector<string> vec2(vec); //用一个容器去初始化另一个容器

    vector<string> vec3(vec.begin(), vec.end()); //用一对迭代器去遍历另一个容器

    vector<string> vec4(10, "test"); //指定容器的大小和初始值
```

```
    list<string> lst(vec.begin(), vec.end());

    return 0;
}
```

第二种和第五种初始化方式的区别在于：

第二种不仅要求容器类型相同，还要求容器元素类型完全一致，而第五种不要求容器相同，对于容器元素，要求能相互兼容即可

指针可以当做迭代器，所以可以这样做：

```
#include <iostream>
#include <string>
#include <vector>
#include <list>

using namespace std;

int main(int argc, char **argv) {
    const size_t MAX_SIZE = 3;
    string arr[MAX_SIZE] = { "hello", "world", "foobar" };

    vector<string> vec(arr, arr + MAX_SIZE);

    return 0;
}
```

二 容器元素的类型约束

1.容器元素必须支持赋值运算

2.元素类型的对象必须可以复制

例如：

```
#include <iostream>
#include <string>
#include <vector>
#include <list>

using namespace std;

class Student{
private:
    //编译错误的根源所在
    Student(const Student &);
    Student &operator=(const Student &);
};

int main(int argc, char **argv) {
    vector<Student> vec(10);

    return 0;
}
```

这个例子编译时错误的，因为 Student 是不可复制的，不满足 STL 中容器元素的基本要求。

三 迭代器

迭代器支持以下操作

```
*iter
Iter->
++iter
--iter
Iter1 == iter2
Iter1 !=iter2
```

Vector 还提供了以下操作

```
Iter+n
Iter1 += iter2
> >= < <=
```

迭代器范围

C++ 语言使用一对迭代器标记迭代器范围(iterator range),这两个迭代器分别指向同一个容器中的两个元素或超出末端的下一位置,通常将它们命名为 first 和 last,或 beg 和 end,用于标记容器中的一段元素范围。

尽管 last 和 end 这两个名字很常见,但是它们却容易引起误解。其实第二个迭代器从来都不是指向元素范围的最后一个元素,而是指向最后一个元素的下一位置。该范围内的元素包括迭代器 first 指向的元素,以及从 first 开始一直到迭代器 last 指向的位置之前的所有元素。如果两个迭代器相等,则迭代器范围为空。

此类元素范围称为左闭合区间(left-inclusive interval),其标准表示

方式为:

[first, last)

表示范围从 first 开始,到 last 结束,但不包括 last。迭代器 last 可以等于 first,或者指向 first 标记的元素后面的某个元素,但绝对不能指 first 标记的元素前面的元素。

使用左闭合区间的编程意义

因为左闭合区间有两个方便使用的性质,所以标准库使用此烦区间。

假设 first 和 last 标记了一个有效的迭代器范围,于是:

1. 当 first 与 last 相等时,迭代器范围为空;

2. 当 first 与不相等时,迭代器范围内至少有一个元素,而且 first 指向该区间中的第一元素。此外,通过若干次自增运算可以使 first 的值不断增大,直到 first == last 为止。

这两个性质意味着程序员可以安全地编写如下的循环,通过测试迭代器处理

一段元素:

```
while (first != last) {
   ++first;
}
```

特殊的迭代器成员 begin 和 end

有四个特殊的迭代器:

c.begin() //指向容器 C 的第一个元素

C.end() //指向最后一个元素的下一个位置

C.rbegin() //返回一个逆序迭代器，指向容器 c 的最后一个元素

C.rend() //返回一个逆序迭代器，指向容器 c 的第一个元素的前面的位置

分别去顺序迭代和逆序迭代容器，例如:

```
#include <iostream>
#include <string>
#include <vector>
#include <list>

using namespace std;

int main(int argc, char **argv) {
```

```
        vector<string> vec;
        vec.push_back("beijing");
        vec.push_back("shanghai");
        vec.push_back("guangzhou");
        vec.push_back("shenzhen");

        for (vector<string>::iterator iter = vec.begin(); iter !=
    vec.end();
                ++iter) {
            cout << *iter << endl;
        }

        for (vector<string>::reverse_iterator iter = vec.rbegin();
                iter != vec.rend(); ++iter) {
            cout << *iter << endl;
        }

        return 0;
    }
```

顺序容器的操作

每个顺序容器都提供了以下操作

添加元素

在容器中删除元素

设置容器大小

容器内置了一些类型，例如 size_type 和 iterator

Size_type：无符号整数，当我们采用下标的方式遍历容器，尽可能采用这种类型

Iterator：此容器类型的迭代器类型，每个容器都内置了自己的迭代

器类型

往容器中添加元素

1.在容器的指定位置添加元素

在容器中使用 push_back 或者 push_front 可以在容器的尾部和首部添加元素

还可以使用 insert(p, t) 在指定位置元素之前添加元素 p 是迭代器，t 时元素的值

```
#include <iostream>
#include <string>
#include <vector>
#include <list>

using namespace std;

int main(int argc, char **argv) {

    vector<string> vec;
    vec.push_back("beijing");
    vec.push_back("shanghai");
    vec.push_back("guangzhou");
    vec.push_back("shenzhen");

    list<string> lst;

    lst.push_front("test"); //在容器的开头添加元素
    lst.push_back("foo");   //在容器后面追加元素

    lst.insert(lst.begin(), "shenzhen");  //在指定迭代器位置添加元素

    return 0;
}
```

2.插入一段元素

插入一段元素采用的是 insert 的另外两个版本

Insert(p, n, t) //在迭代器 p 指向的位置插入 n 个元素，初始值为 t

Insert(p, b, e) //在迭代器 p 指向的位置插入 b 和 e 之间的元素

例如：

```
#include <iostream>
#include <string>
#include <vector>
#include <list>

using namespace std;

int main(int argc, char **argv) {

    vector<string> vec;
    vec.push_back("beijing");
    vec.push_back("shanghai");
    vec.push_back("guangzhou");
    vec.push_back("shenzhen");

    list<string> lst;

    lst.insert(lst.begin(), 5, "hangzhou");   //在开头处插入5个“杭州”
    lst.insert(lst.end(), vec.begin(), vec.end()); //在后面添加 vec 的所有元素

    return 0;
}
```

3.迭代器的失效问题

任何 insert 或者 push 操作都可能导致迭代器失效。当编写循环将元素插入到 vector 或 list 容器中时，程序必须确保迭代器在每次循环后都得到更新

4.元素的访问

遍历容器的方式有两种，一种是采用下标，一种是迭代器。

但是 list 不支持下标访问，所以迭代器是更通用的操作

例如：

```
#include <iostream>
#include <string>
#include <vector>
#include <list>
using namespace std;
int main(int argc, char **argv) {

    vector<string> vec;
    vec.push_back("beijing");
    vec.push_back("shanghai");
    vec.push_back("guangzhou");
    vec.push_back("shenzhen");

    for(vector<string>::size_type ix = 0; ix != vec.size(); ++ix){
        //
    }

    for(vector<string>::iterator iter = vec.begin(); iter != vec.end();
++iter){
        //
    }

    return 0;
}
```

删除元素

1.删第一个或最后一个元素

类似与插入元素，pop_front 或者 pop_back 可以删除第一个或者最后一个元素

```
#include <iostream>
#include <string>
#include <vector>
#include <list>
using namespace std;
int main(int argc, char **argv) {

    vector<string> vec;
    vec.push_back("beijing");
    vec.push_back("shanghai");
    vec.push_back("guangzhou");
    vec.push_back("shenzhen");

    list<string> lst(vec.begin(), vec.end());

    lst.pop_back();  //删除最后一个元素
    lst.pop_front(); //删除第一个元素

    return 0;
}
```

2.删除容器的一个元素

与 insert 对应，删除采用的是 erase 操作，该操作有两个版本：删除由一个迭代器指向的元素，或者删除由一对迭代器标记的一段元素。

例如：

```
#include <iostream>
#include <string>
#include <vector>
#include <list>
#include <algorithm>
using namespace std;
int main(int argc, char **argv) {

    vector<string> vec;
    vec.push_back("beijing");
    vec.push_back("shanghai");
    vec.push_back("guangzhou");
    vec.push_back("shenzhen");
    vec.push_back("jinan");

    list<string> lst(vec.begin(), vec.end());


    //查找“shanghai”元素
    list<string>::iterator  iter1  =  find(lst.begin(),  lst.end(),
"shanghai");
    lst.erase(iter1);//删除这一个元素

    //查找“guanzghou”元素的迭代器和“shenzhen”的迭代器
    list<string>::iterator  first  =  find(lst.begin(),  lst.end(),
"guanzghou"),
            last = find(lst.begin(), lst.end(), "shenzhen");

    lst.erase(first, last); //删除这一段元素

}
```

这里采用了标准库的 find 算法，下文将会提到

删除元素时可能的陷阱：

3.删除容器的所有元素

删除所有元素有两种办法，一种是 clear，一种是采用 erase 删除所有的范围

```
#include <iostream>
#include <string>
#include <vector>
#include <list>
#include <algorithm>
using namespace std;
int main(int argc, char **argv) {

    vector<string> vec;
    vec.push_back("beijing");
    vec.push_back("shanghai");
    vec.push_back("guangzhou");
    vec.push_back("shenzhen");
    vec.push_back("jinan");

    list<string> lst(vec.begin(), vec.end());

    lst.clear();
    lst.erase(lst.begin(), lst.end());
}
```

查找元素的操作

因为顺序容器内部没有内置 find 操作，所以我们采用标准库提供的一个泛型算法：

例子如下：

```
#include <iostream>
#include <string>
#include <vector>
#include <list>
#include <algorithm>
using namespace std;
int main(int argc, char **argv) {

    vector<string> vec;
    vec.push_back("beijing");
    vec.push_back("shanghai");
    vec.push_back("guangzhou");
    vec.push_back("shenzhen");
    vec.push_back("jinan");

    std::string query;  //要查询的单词
    vector<string>::iterator iter = std::find(vec.begin(), vec.end(),
           "beijing");
    if (iter == vec.end()) {//表示没有找到
        cout << "not found!" << endl;
    } else {
        cout << *iter << endl;
    }

}
```

需要注意的是，必须考虑到查找不到的情况

容器大小的操作

size() //返回容器的大小

empty() //返回容器是否为空

resize(n) //重置容器的大小

例如：

```
#include <iostream>
#include <string>
#include <vector>
#include <list>
#include <algorithm>
using namespace std;
int main(int argc, char **argv) {

    list<string> lst(10);
    cout << lst.size() << endl;

    cout << lst.empty() << endl;

    lst.resize(100); //重置容器元素个数为100
    cout << lst.size() << endl;

}
```

Vector 的内存增加策略

Vector 内部的元素时连续存储的，类似于数据结构中的顺序表

Vector 内置了两个函数：

Capacity 操作获取容器需要分配更多的存储空间之前能够存储的元素个数总数

Reserve 操作告诉 vector 容器应该预留多少个元素的存储空间

```
#include <iostream>
#include <string>
#include <vector>
#include <list>
#include <algorithm>
```

```
using namespace std;
int main(int argc, char **argv) {

    vector<int> vec; //空数组
    cout <<"size:" <<vec.size() <<" capacity:" <<vec.capacity()
<< endl;

    //放入 24 个元素
    for(vector<int>::size_type ix = 0; ix != 24; ++ix){
        vec.push_back(24);
    }
    cout <<"size:" <<vec.size() <<" capacity:" <<vec.capacity()
<< endl;

    //将容器的预留空间设为 50
    vec.reserve(50);
    cout <<"size:" <<vec.size() <<" capacity:" <<vec.capacity()
<< endl;

    //将容器的现有空间填充完毕
    while(vec.size() != vec.capacity()){
        vec.push_back(0);
    }
    cout <<"size:" <<vec.size() <<" capacity:" <<vec.capacity()
<< endl;

    //此时容器满，再放入一个元素
    vec.push_back(42);
    cout <<"size:" <<vec.size() <<" capacity:" <<vec.capacity()
<< endl;

}
```

Vector 和 list 的区别

Vector 内部 采用顺序表实现，而 list 采用链表实现，所以二者的差别很大程度是顺序表和链表的差别

如果不知道选择哪种容器，通常 vector 是最佳选择

String 类型

构造 string 类型的方法

可以使用字符串字面值直接初始化字符串

插入元素

提供了 insert 方法，有大量的重载版本

删除元素

Erase 方法

```
int main(int argc, char **argv) {
    string s = "helloworld";

    s.insert(2, 5, 's');   // pos n  value
    cout << s << endl;

    s = "helloworld";

    s.insert(4, "test");   // pos  C-style string
    cout << s << endl;

    s = "helloworld";
    s.insert(2, "test", 3);  // pos char* len (0 -len-1)
    cout << s << endl;

    s = "helloworld";
    string _tmp = "foobar";
    s.insert(5, _tmp, 3, 2); //pos string pos2 len
    cout << s << endl;  // hellobaworld

    s = "helloworld";
    string _tmp2 = "foobar";
    s.insert(0, _tmp2);   // pos  string
    cout << s << endl;   // foobarhelloworld
```

```
    string s2 = "helloworld";

    s2.erase(4, 5);   //pos n

    cout << s2 << endl;  //helld

}
```

截取字串 substr

Append

在 string 后面串接新的字符串

```
#include "test_include.h"
using namespace std;
int main() {
    string s = "helloworld";

    cout << s.substr(5, 3) << endl;  // pos n

    s.append("hello");

    cout << s << endl;

    s.append("hello", 3); //hel

    cout << s << endl;

    s.append(string("world"));
    cout << s << endl;

    s.append(string("foobar"), 3, 3);  // bar

    cout << s << endl;

    s.append(8, 's');  // 8 s
    cout << s << endl;

    string tmp = "zhangsan";
```

```
    string::iterator it1 = find(tmp.begin(), tmp.end(), 'h'),
           it2 = find(tmp.begin(), tmp.end(), 'g');
    //查找 h 和 g 所在位置的迭代器

    s.append(it1, it2);   //han

    cout << s << endl;


}
```

Replace

将字符插入到某些位置，并替换已经存在的字符

```
#include "test_include.h"
using namespace std;
int main() {
    string s = "helloworld";


    s.replace(4, 3, "test");  // owo -> test

    cout << s << endl;  //helltestrld


    s.replace(4, 3, string("foobar"), 3, 3);   // tes -> bar

    cout << s << endl;  // hellbartrld

}
```

String 内部的查找操作

```
#include "test_include.h"
using namespace std;
int main() {
    string s = "helloworld";

    string::size_type pos1 = s.find(string("test"), 2); //

    if (pos1 != string::npos) {   //如果查找不到
        cout << pos1 << endl;
    } else {
        cout << "not found" << endl;
    }
}
```

String 的比较操作

采用的是字典排序，规则与 C 风格字符串相同

Stack 和 queue

Stack 和 queue 是标准库对栈和队列两种数据结构的封装，他们的使用方法和以前封装的 stack 和 queue 类似