

1 STL简介

1.1 STL诞生

- 软件界希望建立一种可重复利用的东西
- C++的面向对象（**面对对象的三大特性：封装、继承、多态**）和泛型编程的思想，目的就是复用性提升
- 大多数情况下，数据结构和算法都未能有一套标准，导致被迫从事大量重复工作
- 为了建立数据结构和算法的一套标准，诞生了STL

2. 1.2 STL基本概念

- STL (Standard Template Library, 标准模板库)
- STL从广义上分为：**容器 (container)**、**算法 (algorithm)**、**迭代器 (iterator)**
- **容器和算法之间通过迭代器进行无缝连接**
- STL几乎所有的代码都采用了**模板类和模板函数**

3. 1.3 STL六大组件

- STL大体分为六大组件，分别为**容器、算法、迭代器、仿函数、适配器（配接器）、空间配置器**
 - 1. 容器:各种数据结构，如vector、list、deque、set、map等,用来存放数据。
 - 2. 算法:各种常用的算法，如sort、find、copy、for_each等
 - 3. 迭代器:扮演了容器与算法之间的胶合剂。
 - 4. 仿函数:行为类似函数,可作为算法的某种策略。（eg.重载=、重载()等函数）
 - 5. 适配器:一种用来修饰容器或者仿函数或迭代器接口的东西。
 - 6. 空间配置器:负责空间的配置与管理。（eg. 自动管理堆栈区的内存）

4. 1.4 STL中容器、算法、迭代器

- 容器：置物之所也
- STL容器就是将运用最广泛的一些数据结构实现出来
- 常用的数据结构:数组,链表,树,栈,队列,集合,映射表等
- 这些容器分为**序列式容器**和**关联式容器**两种:
 - 序列式容器:强调值的排序，序列式容器中的每个元素均有固定的位置。如，顺序表，链表，队列，栈（放进去之前什么样，出来什么样）
 - 关联式容器：二叉树结构，各元素之间没有严格的物理上的顺序关系。如，树，图，集合，映射表（会进行排序）
- 算法：问题之解法也

- 有限的步骤,解决逻辑或数学上的问题,这一门学科我们叫做算法(Algorithms) (算法头文件就是 Algorithms)
- 算法分为:**质变算法和非质变算法**:
 - 质变算法:是指运算过程中会更改区间内的元素的内容。例如, 拷贝,替换,删除等等
 - 非质变算法:是指运算过程中不会更改区间内的元素内容,例如, 查找、计数、遍历、寻找极值等等

- 迭代器: 容器和算法之间粘合剂 (通过迭代器来访问容器内容)
- 提供一种方法,使之能够依序**寻访某个容器所含的各个元素**,而又无需暴露该容器的内部表示方式。
- **每个容器都有自己专属的迭代器**
- 迭代器使用非常类似于指针,初学阶段我们可以先理解迭代器为指针

- 迭代器种类:

种类	功能	支持运算	举例
输入迭代器	对数据的只读访问	只读, 支持++, ==, !=	std::istream_iterator
输出迭代器	对数据的只写访问	只写, 支持++	std::ostream_iterator
前向迭代器	读写操作, 并能向前推进迭代器	读写, 支持++, ==, !=	std::forward_list
双向迭代器	读写操作, 并能向前和向后操作	读写, 支持++, --	std::list
随机访问迭代器	读写操作, 可以以跳跃的方式访问任意数据, 功能最强的迭代器	读写, 支持++, --, [n], -n, <, <=, >, <=	std::vector、std::array

Table 1

- 常用的容器中迭代器种类为双向迭代器, 和随机访问迭代器

2 常用容器

5.2.1 vector

- 容器: vector
- 算法: for_each
- 迭代器: vector<数据类型>::iterator
- 容器头文件: #include
- 算法头文件: #include

5.1 (1) 存放内置数据类型

示例:

```
1  #include <iostream>
2  #include <vector>           //vector容器头文件
3  #include <algorithm>        //标准算法头文件
4
5  using namespace std;
6
7  /* vector插入内置数据 */
8
9
10 void myPrint(int val) {
11     cout << val << endl;
12 }
13
14 void test01(void) {
15     /* 创建vector容器 */
16     vector<int> v;           //vector<数据类型>容器变量;
17
18     /* 在容器中放数据 */
19     v.push_back(10);
20     v.push_back(20);
21     v.push_back(30);
22     v.push_back(40);
23     v.push_back(50);
24
25     /* 创建vector迭代器 */
26     /* vector<数据类型>::迭代器类型 迭代器变量名字  iterator: 正向迭代器 */
27     vector<int>::iterator myBegin = v.begin();           //创建起始迭代器 指向v容器中的第一个
28     vector<int>::iterator myEnd = v.end();               //创建终止迭代器 指向v容器中的最后一个元素的下一位
29
30     /* 第一种遍历 */
```

```

31     while (myBegin != myEnd)
32     {
33         cout << *myBegin << endl;
34         myBegin++; //与指针操作一样
35     }
36
37     cout << endl;
38
39     /* 第二种遍历 */
40     for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
41         cout << *it << endl;
42     }
43
44     cout << endl;
45
46     /* 第三种遍历 */
47     /* 参数为（起始位置，终点位置，可调用对象） 可调用对象：这里使用的函数，利用的时回调函数的知识
48     */
49     for_each(v.begin(), v.end(), myPrint);
50
51
52
53     int main(void) {
54
55         test01();
56
57         return 0;
58     }

```

Fence 1

5.2 (2) 存放自定义数据类型

示例:

```

1     #include <iostream>
2     #include <algorithm>
3     #include <vector>
4     #include <string>
5
6     using namespace std;
7
8
9
10
11     class Person {
12     public:
13
14         Person(string name, int age) {

```

```

15         this->m_name = name;
16         this->m_age = age;
17     }
18
19     string m_name;
20     int m_age;
21
22 };
23
24 void myPrintf(Person val) {
25     cout << "姓名: " << val.m_name << " 年龄: " << val.m_age << endl;
26 }
27
28 /* vector容器存放自定义数据类型 */
29
30 void test01(void) {
31     vector<Person> v;
32
33     Person p1("aaa", 10);
34     Person p2("BBB", 20);
35     Person p3("ccc", 30);
36     Person p4("ddd", 40);
37     Person p5("eee", 50);
38     Person p6("fff", 60);
39
40     v.push_back(p1);
41     v.push_back(p2);
42     v.push_back(p3);
43     v.push_back(p4);
44     v.push_back(p5);
45     v.push_back(p6);
46
47     /* for循环遍历 */
48     for (vector<Person>::iterator it = v.begin(); it != v.end(); it++) {
49         cout << "姓名: " << (*it).m_name << "年龄: " << it->m_age << endl;
50     }
51
52     /* for_each()遍历 */
53     for_each(v.begin(), v.end(), myPrintf);
54
55 }
56
57 /* vector容器存放自定义数据类型 指针 */
58 void myPrintf1(Person* val) {
59     cout << "姓名: " << val->m_name << " 年龄: " << (*val).m_age << endl;
60 }
61
62 void test02(void) {
63     vector<Person*> v;
64
65     Person p1("aaa", 10);
66     Person p2("BBB", 20);
67     Person p3("ccc", 30);

```

```

68     Person p4("ddd", 40);
69     Person p5("eee", 50);
70     Person p6("fff", 60);
71
72     v.push_back(&p1);
73     v.push_back(&p2);
74     v.push_back(&p3);
75     v.push_back(&p4);
76     v.push_back(&p5);
77     v.push_back(&p6);
78
79     /* for循环遍历 */
80     for (vector<Person*>::iterator it = v.begin(); it != v.end(); it++) {
81         cout << "姓名: " << (*(it)).m_name << "年龄: " << (*(it))->m_age << endl;
82     }
83
84     /* for_each()遍历 */
85     for_each(v.begin(), v.end(), myPrintf1);
86
87 }
88
89 int main(void) {
90
91     test01();
92
93     cout << endl;
94
95     test02();
96     return 0;
97 }
98

```

Fence 2

5.3 (3) 容器嵌套容器

示例:

```

1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4
5  using namespace std;
6
7
8  /* vector容器嵌套容器 */
9
10 void test01(void) {
11     vector<vector<int>> v;
12
13     /* 创建小容器 */
14     vector<int> v1;
15     vector<int> v2;

```

```

16     vector<int> v3;
17     vector<int> v4;
18
19     /* 在小容器中创建数据 */
20     for (int i = 0; i < 4; i++) {
21         v1.push_back(i * 1);
22         v2.push_back(i * 10);
23         v3.push_back(i * 100);
24         v4.push_back(i * 1000);
25     }
26
27     /* 在大容器中写入数据 */
28     v.push_back(v1);
29     v.push_back(v2);
30     v.push_back(v3);
31     v.push_back(v4);
32
33     /* for循环遍历 */
34     for (vector<vector<int>>::iterator it = v.begin(); it != v.end(); it++) {
35         for (vector<int>::iterator it1 = (*it).begin(); it1 != (*it).end(); it1++) {
36             // *it 可以看<>中的数据是什么，就可以知道类型
37             cout << *it1 << " ";
38         }
39         cout << endl;
40     }
41
42 }
43
44
45
46 int main(void) {
47
48     test01();
49
50
51     return 0;
52 }
53

```

Fence 3

5.4 (4) 基本概念

- 功能：
 - `vector` 数据结构和普通数组非常相似，通常也被称为**单端数组**。
- `vector` 与普通数组的区别：
 - 普通数组在定义时大小是静态的，而 `vector` 则可以**动态扩展**。（`vector`头文件中，会倍数放大你要开辟的空间）
- 动态扩展：

- `vector` 并不是在原空间之上继续扩展，而是寻找更大的内存空间（一般会分配更大的空间，避免频繁开辟空间），然后将原数据移动到新空间，释放原空间。（`vector` != 链表）

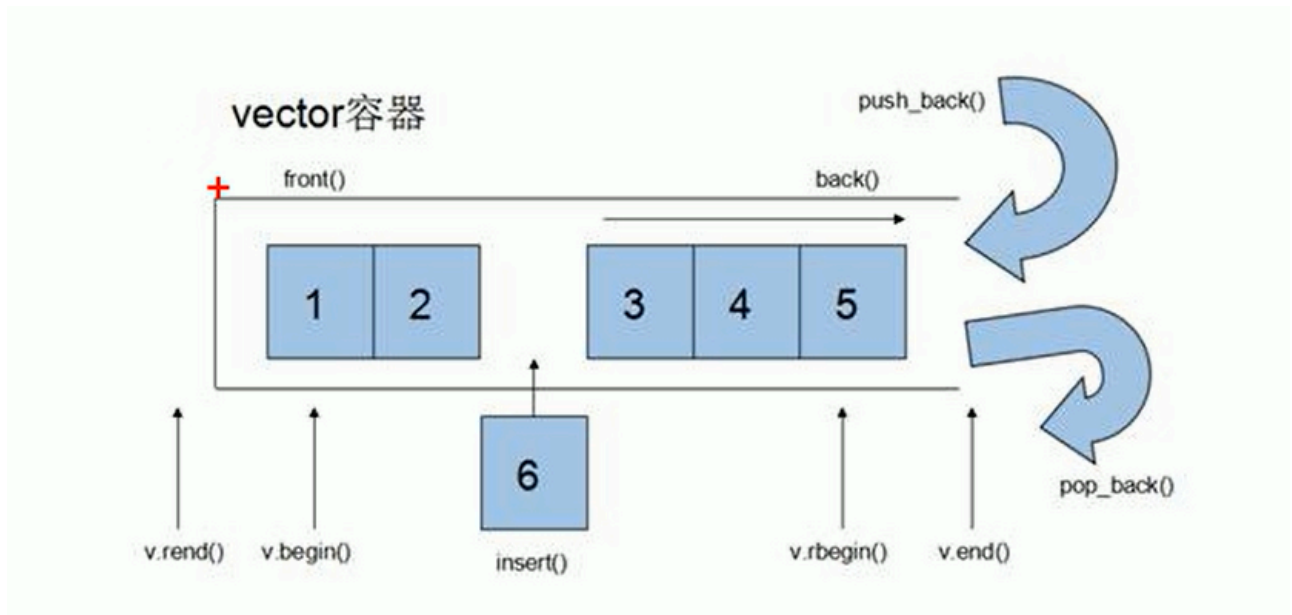


Figure 1

- `vector` 容器的迭代器是支持随机访问的迭代器

5.5 (5) 构造函数

- 功能描述：
 - 创建 `vector` 容器。
- 函数原型：
 - `vector<T> v;`
// 采用模板实现类型实例化，默认为构造函数。
 - `vector(v.begin(), v.end());`
// 将 `v` 中 `[begin(), end())` 区间的元素拷贝给本身。 `end()` 取得是最后一个元素的下一个位置
 - `vector(n, elem);`
// 构造函数将 `n` 个 `elem` 拷贝给本身。
 - `vector(const vector &vec);`
// 拷贝构造函数。

示例：

```

1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4
5  using namespace std;
6
7  /* vector容器的构造函数 */
8
9  void printVector(vector<int> v) {

```



```

10     for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
11         cout << *it << " ";
12     }
13     cout << endl;
14 }
15
16 void test01() {
17     vector<int> v1; // 采用模板实现类型实例化，默认为构造函数。
18     for (int i = 0; i < 10; i++) {
19         v1.push_back(i);
20     }
21     printVector(v1);
22
23     vector<int> v2(v1.begin(), v1.end()); // 将`v`中`[begin(), end())`区间的元素拷
    贝给本身。`end()`取得是最后一个元素的下一个位置
24     printVector(v2);
25
26     vector<int> v3(10,100); // 构造函数将`n`个`elem`拷贝给本身。
27     printVector(v3);
28
29     vector<int> v4(v3); // 拷贝构造函数。
30     printVector(v4);
31 }
32
33 int main() {
34
35     test01();
36
37     getchar();
38     return 0;
39 }
40
41

```

Fence 4

5.6 (6) 赋值操作

- 功能描述：
 - 给 vector 容器进行赋值。
- 函数原型：
 - `vector& operator=(const vector &vec);`
// 重载等号操作符。
 - `assign(beg, end);`
// 将 [beg, end) 区间中的数据拷贝赋值给本身。
 - `assign(n, elem);`
// 将 n 个 elem 拷贝赋值给本身。

示例:

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4
5  using namespace std;
6
7  /* vector容器的赋值操作 */
8
9  void printVector(vector<int> &v) {
10     for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
11         cout << *it << " ";
12     }
13     cout << endl;
14 }
15
16 void test01() {
17     vector<int> v1;
18     for (int i = 0; i < 10; i++) {
19         v1.push_back(i);
20     }
21     printVector(v1);
22
23     vector<int> v2;                                //vector& operator=(const vector & vec);
24     // 重载等号操作符。
25     v2 = v1;
26     printVector(v2);
27
28     vector<int> v3;                                //assign(beg, end); //将[beg, end)区间中的数
29     //拷贝赋值给本身。
30     v3.assign(v1.begin(), v1.end());
31     printVector(v3);
32
33     vector<int> v4;                                //assign(n, elem);将n个elem拷贝赋值给本身。
34     v4.assign(10, 100);
35     printVector(v4);
36 }
37
38 int main() {
39
40     test01();
41
42     getchar();
43     return 0;
44 }
45
46
```

5.7 (7) 容量和大小

- 功能描述：
 - 对 `vector` 容器的容量和大小操作。
- 函数原型：
 - `empty();`
// 判断容器是否为空。
 - `capacity();`
// 容器的容量。
 - `size();`
// 返回容器中元素的个数。
 - `resize(int num);`
// 重新指定容器的长度为 `num`，若容器变长，则用默认值（0）填充新位置。若容器变短，则末尾超出容器长度的元素被删除。（不会改变容量（系统分配的空间），只会改变大小（实际使用的空间））
 - `resize(int num, elem);`
// 重新指定容器的长度为 `num`，若容器变长，则以 `elem` 值填充新位置。若容器变短，则末尾超出容器长度的元素被删除。

示例：

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4
5  using namespace std;
6
7  /* vector容器的容量和大小 */
8
9  void printVector(vector<int>& v) {
10     for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
11         cout << *it << " ";
12     }
13     cout << endl;
14 }
15
16 void test01() {
17     vector<int> v1;
18     for (int i = 0; i < 10; i++) {
19         v1.push_back(i);
20     }
21     printVector(v1);
22
23     if (v1.empty()) { // 判断容器是否为空。
24         cout << "容器为空" << endl;
25     }
26     else
27     {
```

```

28         cout << "容量: " << v1.capacity(); // 容器的容量。
29         cout << endl;
30         cout << "元素个数: " << v1.size(); // 返回容器中元素的个数。
31         cout << endl;
32         cout << "重新指定长度: " << endl; //resize(int num); 重新指定容器的长度为`num`, 若
容器变长, 则默认用`值`填充新位置。若容器变短, 则末尾超出容器长度的元素被删除。
33                                     //resize(int num, elem); 重新指定容器的长度为
`num`, 若容器变长, 则以`elem`值填充新位置。若容器变短, 则末尾超出容器长度的元素被删除。
34         v1.resize(3);
35         printVector(v1);
36         v1.resize(5, 100);
37         printVector(v1);
38     }
39
40
41 }
42
43 int main() {
44
45     test01();
46
47     getchar();
48     return 0;
49 }
50
51

```

Fence 6

5.8 (8) 插入和删除

- 功能描述:
 - 对 `vector` 容器进行插入、删除操作。
- 函数原型:
 - `push_back(ele);`
// 尾部插入元素 `ele`。
 - `pop_back();`
// 删除最后一个元素。
 - `insert(const_iterator pos, ele);`
// 迭代器指向位置 `pos` 插入元素 `ele`。(提供迭代器)
 - `insert(const_iterator pos, int count, ele);`
// 迭代器指向位置 `pos` 插入 `count` 个元素 `ele`。(提供迭代器)
 - `erase(const_iterator pos);`
// 删除迭代器指向的元素。(提供迭代器)
 - `erase(const_iterator start, const_iterator end);`
// 删除迭代器从 `start` 到 `end` 之间的元素。(提供迭代器) (不包括`end`位置上的数)

- `clear();`
// 删除容器中所有元素。

- 1、正向迭代器： 容器名<类型>::iterator 迭代器名
- 2、常量正向迭代器： 容器名<类型>::const_iterator 迭代器名
- 3、反向迭代器： 容器名<类型>::reverse_iterator 迭代器名
- 4、常量反向迭代器： 容器名<类型>::const_reverse_iterator 迭代器名

示例：

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4
5  using namespace std;
6
7  /* vector容器的插入和删除 */
8
9  void printVector(vector<int>& v) {
10     for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
11         cout << *it << " ";
12     }
13     cout << endl;
14 }
15
16 void test01() {
17     vector<int> v1;
18     for (int i = 0; i < 10; i++) {
19         v1.push_back(i);                // 尾部插入元素`ele`。
20     }
21     printVector(v1);
22
23     cout << "删除最后一个元素" ;
24     v1.pop_back();                      // 删除最后一个元素。
25     printVector(v1);
26
27     vector<int>::iterator it = v1.begin(); //正向迭代器（只能用正向或者常量正向）
28     cout << "指定插入一个元素";          // 迭代器指向位置`pos`插入元素`ele`。（提供迭
    代器）
29     v1.insert(it,100);                  //v.insert(v.begin() + index,
    element);
30     printVector(v1);
31
32     it = v1.begin();                    //重新获取头部位置
33     cout << "指定插入n个相同元素";        // 迭代器指向位置`pos`插入`count`个元素
    `ele`。（提供迭代器）
```

```

34     v1.insert(it, 2, 110);
35     printVector(v1);
36
37     cout << " 删除一个元素 ";
38     v1.erase(v1.begin() + 2);           // 删除迭代器指向的元素。（提供迭代器） 删除
    一个100
39     printVector(v1);
40
41     cout << " 删除区间元素 ";
42     v1.erase(v1.begin(), v1.begin() + 2); // 删除迭代器从`start`到`end`之间的元素。
    删除110、110
43     printVector(v1);                   //清空 v1.erase(v1.begin().v1.end());
44
45     cout << "清空容器";
46     v1.clear();                         // 删除容器中所有元素。
47     printVector(v1);
48 }
49
50 int main() {
51
52     test01();
53
54     getchar();
55     return 0;
56 }
57
58

```

Fence 7

5.9 (9) 数据存取

- 功能描述：
 - 对 `vector` 中的数据进行获取操作。
- 函数原型：
 - `at(int idx);`
// 返回索引 `idx` 所指的数据。
 - `operator[];`
// 返回索引 `idx` 所指的数据。
 - `front();`
// 返回容器中的第一个数据元素。
 - `back();`
// 返回容器中的最后一个数据元素。

示例：

```

1  #include <iostream>

```

```

2  #include <vector>
3  #include <algorithm>
4
5  using namespace std;
6
7  /* vector容器的数据存取*/
8
9  void printVector(vector<int>& v) {
10     for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
11         cout << *it << " ";
12     }
13     cout << endl;
14 }
15
16 void test01() {
17     vector<int> v1;
18     for (int i = 0; i < 10; i++) {
19         v1.push_back(i);
20     }
21     printVector(v1);
22
23     cout << "成员函数at方式: " << endl;
24     for (int i = 0; i < v1.size(); i++) {
25         cout << v1.at(i) << " ";
26     }
27
28     cout << endl << "重载[]方式: " << endl;
29     for (int i = 0; i < v1.size(); i++) {
30         cout << v1[i] << " ";
31     }
32     cout << endl;
33
34     cout << " 返回第一个数据: " << v1.front() << endl; // 返回容器中的第一个数据元素。
35     cout << " 返回最后一个数据: " << v1.back() << endl; // 返回容器中的最后一个数据元素。
36 }
37
38 int main() {
39
40     test01();
41
42     getchar();
43     return 0;
44 }
45
46

```

5.10 (10) 互换容器

- 功能描述：
 - 实现两个容器内元素进行互换。（可以实现内存收缩的效果）
- 函数原型：
 - `swap(vec);`
// 将 `vec` 与自身的元素进行互换。

示例：

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4
5  using namespace std;
6
7  /* vector容器的互换容器*/
8
9  void printVector(vector<int>& v) {
10     for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
11         cout << *it << " ";
12     }
13     cout << endl;
14 }
15
16 void test01(void) {
17     vector<int> v1,v2;
18     for (int i = 0; i < 10; i++) {
19         v1.push_back(i);
20         v2.push_back(i * 10);
21     }
22     cout << "交换前: " << endl;
23     printVector(v1);
24     printVector(v2);
25
26     v1.swap(v2);                // 将 `vec` 与自身的元素进行互换。
27
28     cout << "交换后: " << endl;
29     printVector(v1);
30     printVector(v2);
31
32 }
33
34 void test02(void) {
35     vector<int> v;
36     for (int i = 0; i < 10000; i++) {
37         v.push_back(i);
38     }
39
40     cout << "容量: " << v.capacity() << endl;
```



```

41     cout << "大小: " << v.size() << endl;
42
43     v.resize(3);
44
45     cout << "容量: " << v.capacity() << endl;
46     cout << "大小: " << v.size() << endl;
47
48     /* 使用swap收缩容量 */
49     vector<int>(v).swap(v); //匿名对象(当前行执行完, 系统回收内存), 创建
    了一个新的 vector<int> 对象, 初始化为 v 的内容
50     cout << "容量: " << v.capacity() << endl;
51     cout << "大小: " << v.size() << endl;
52 }
53
54 int main(void) {
55
56     test01();
57     test02();
58
59     return 0;
60 }
61
62

```

Fence 9

5.11 (11) 预留空间

- 功能描述:
 - 减少 `vector` 在动态扩展容量时的扩展次数。
- 函数原型:
 - `reserve(int len);`
// 容器预留 `len` 个元素的长度, 预留位置未初始化, 元素不可访问。

示例:

```

1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4
5  using namespace std;
6
7  /* vector容器的预留空间*/
8
9  void printVector(vector<int>& v) {
10     for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
11         cout << *it << " ";
12     }
13     cout << endl;
14 }
15

```

```

16 void test01(void) {
17     vector<int> v1;
18
19     v1.reserve(10000);           // 容器预留 `len` 个元素的长度，预留位置未初始化，元
                                   // 素不可访问。
20
21     /* 统计开辟空间次数 */
22     int num = 0;
23     int* p = NULL;
24
25     for (int i = 0; i < 10000; i++) {
26         v1.push_back(i);
27
28         if (p != &v1[0]) {       // 如果不等于v1的首地址，就强制等于首地址
29             p = &v1[0];
30             num++;
31         }
32     }
33
34     cout << "num=" << num << endl; // 无预留空间开辟24次
35
36 }
37
38
39 int main(void) {
40
41     test01();
42
43     return 0;
44 }
45
46

```

Fence 10

6. 2.2 String

6.1 (1) 基本概念

- 本质：
 - string是C++风格的字符串，而string**本质上是一个类**
- 与char*的区别：
 - char*是一个指针
 - string是一个类，类内部封装了char*，管理这个字符串，是一个char*型的容器
- 特点
 - string类内部封装了很多成员方法，例如，查找find、拷贝copy、删除delete、替换replace、插入insert
 - string管理char*所分配的内存，不用担心复制越界和取值越界等，由类内部进行负责

6.2 (2) 构造函数

构造函数原型：

- `string();` //创建一个空字符串 例如： `string str;`
- `string(const char* s);` //使用字符串s初始化
- `string(const string& str);` //使用一个string对象初始化另一个string对象
- `string(int n, char c);` //使用n个字符c初始化

示例：

```
1  #include <iostream>
2  #include <string>
3
4  using namespace std;
5
6
7  void test01(void) {
8      string s1;                //使用默认构造函数
9
10     const char* str = "你好";
11     string s2(str);            //使用有参构造    string s5("ab");
12
13     string s3(s2);             //使用拷贝构造
14
15     string s4(10, 'b');        //使用另一种有参构造
16
17     cout << "s2 = " << s2 << endl;
18     cout << "s3 = " << s3 << endl;
19     cout << "s4 = " << s4 << endl;
20
21 }
22
23 int main(void) {
24
25     test01();
26
27     return 0;
28 }
```

Fence 11

6.3 (3) 赋值操作

- 功能描述
 - 给string字符串进行赋值
- 赋值的函数原型
 - `string& operator=(const char *s);` //char*类型字符串 赋值给当前字符串
 - `string& operator=(const string &s);` //字符串s赋值给当前的字符串

- `string& operator=(char c);` //字符赋值给当前的字符串
- `string& assign(const char *s);` //把字符串s赋给当前的字符串
- `string& assign(const char *s, int n);` //把字符串s的前n个字符赋给当前字符串
- `string& assign(const string &s);` //把字符串s赋给当前字符串
- `string& assign(int n, char c);` //用n个字符c赋给当前字符串

示例:

```

1  #include <iostream>
2  #include <string>
3
4  using namespace std;
5
6
7  //string的赋值操作
8  void test01(void) {
9      string s1;                //string& operator=(const char *s);
10     s1 = "你好";
11
12     string s2;                //string& operator=(const string &s);
13     s2 = s1;
14
15     char c = 'A';
16     string s3;                //string& operator=(char c);
17     s3 = c;
18
19     string s4;                //string& assign(const char *s);
20     s4.assign("ABC");
21
22     string s5;                //string& assign(const char *s, int n);
23     s5.assign("ABCD", 2);
24
25     string s6;                //string& assign(const string &s);
26     s6.assign(s5);
27
28     string s7;                //string& assign(int n, char c);
29     s7.assign(7, 'a');
30
31     cout << "s1 = " << s1 << endl;
32     cout << "s2 = " << s2 << endl;
33     cout << "s3 = " << s3 << endl;
34     cout << "s4 = " << s4 << endl;
35     cout << "s5 = " << s5 << endl;
36     cout << "s6 = " << s6 << endl;
37     cout << "s7 = " << s7 << endl;
38
39 }
40
41 int main(void) {
42
43     test01();

```

```

44
45     return 0;
46 }

```

Fence 12

6.4 (4) 字符串拼接

- 功能描述

- 在字符串末尾拼接字符串

- 赋值的函数原型

- `string& operator+=(const char* str);` // 重载 += 操作符
- `string& operator+=(const char c);` // 重载 = 操作符
- `string& operator+=(const string& str);` // 重载 += 操作符
- `string& append(const char* s);` // 将字符数组 s 连接到当前字符串末尾
- `string& append(const char* s, int n);` // 将字符数组 s 的前 n 个字符连接到当前字符串末尾
- `string& append(const string& s);` // 将字符串 s 连接到当前字符串末尾
- `string& append(const string& s, int pos, int n);` // 将字符串 s 从 pos 开始的 n 个字符连接到当前字符串末尾
- 还有其他类型的拼接函数

示例:

```

1  #include <iostream>
2  #include <string>
3
4  using namespace std;
5
6
7  //strin的字符串拼接
8  void test01(void) {
9      string s1 = "我";
10
11      s1 += "爱你"; //string& operator+=(const char* str);
12      cout << "s1 = " << s1 << endl;
13
14      s1 += 's'; //string& operator+=(const char c);
15      cout << "s1 = " << s1 << endl;
16
17      string s2 = "在家"; //string& operator+=(const string& str);
18      s1 += s2;
19      cout << "s1 = " << s1 << endl;
20
21      string s3 = "蜜";
22      s3.append("雪冰城"); //string& append(const char* s);
23      cout << "s3 = " << s3 << endl;

```

```

24
25     s3.append(" nihaoshijie",3);           //string& append(const char* s, int n);
26     cout << "s3 = " << s3 << endl;
27
28     s3.append(s2);                         //string& append(const string& s);
29     cout << "s3 = " << s3 << endl;
30
31     s3.append("ABCDEF",2,3);               //string& append(const string & s, int pos,
int n);
32     cout << "s3 = " << s3 << endl;
33
34
35 }
36
37 int main(void) {
38
39     test01();
40
41     return 0;
42 }

```

Fence 13

6.5 (5) 查找与替换

- 功能描述:

- 查找: 查找指定字符串是否存在
- 替换: 在指定的位置替换字符串

- 函数原型:

- `int find(const string& str, int pos = 0) const;` // 查找 str 第一次出现位置, 从 pos 开始查找, 有返回下标; 没有返回-1, 下面同理。从前往后找
- `int find(const char* s, int pos = 0) const;` // 查找第一次出现位置, 从 pos 开始查找
- `int find(const char* s, int pos, int n) const;` // 从 pos 位置查找前 n 个字符第一次位置
- `int find(const char c, int pos = 0) const;` // 查找字符 c 第一次出现位置
- `int rfind(const string& str, int pos = npos) const;` // 查找 str 最后一次出现位置, 从 pos 开始查找。从后往前找
- `int rfind(const char* s, int pos = npos) const;` // 查找最后一次出现位置, 从 pos 开始查找
- `int rfind(const char* s, int pos, int n) const;` // 从 pos 查找字符的前 n 个字符最后一次位置
- `int rfind(const char c, int pos = 0) const;` // 查找字符 c 最后一次出现位置
- `string& replace(int pos, int n, const string& str);` // 替换从 pos 开始的 n 个字符为字符串 str。如果str的长度大于n, 依然能够将str完整插入到 [pos:pos+n] 中去

- `string& replace(int pos, int n, const char* s);` // 替换从 pos 开始的 n 个字符为字符串 s

示例:

```
1  #include <iostream>
2  #include <string>
3
4  using namespace std;
5
6
7  //strin的查找与替换
8  void test01(void) {
9      string s1 = "ni hao shi jie ni ni hao shi jie";
10     string s2 = "shi jie";
11
12     /* find 从前往后 */
13     cout << s1.find(s2) << endl;
14     cout << s1.find("o s") << endl;
15
16     /* rfind 从后往前 */
17     cout << s1.rfind(s2) << endl;
18     cout << s1.rfind("o s") << endl;
19
20     /* replace */
21     cout << s1.replace(8,3,s2) << endl;
22     cout << s1.replace(8,2, "abc") << endl;
23
24     //指定多少字符，都会替换，
25     string s3 = "abcdefg";
26     cout << s3.replace(1, 3, "1111") << endl;           //替换成a1111efg,不会因为长度不够，而
舍弃
27     cout << s3.replace(1, 3, "22") << endl;
28 }
29
30 int main(void) {
31
32     test01();
33
34     return 0;
35 }
```

Fence 14

6.6 (6) 字符串比较

- 功能描述:

- 字符串之间的比较，判断二者大小的意义不是很大

- 比较方式:

- 字符串比较是按字符的 ASCII 码进行比较

- `==` 返回 0

- > 返回 1
- < 返回 -1

• 函数原型:

- `int compare(const string &s) const;` // 与字符串 s 比较
- `int compare(const char *s) const;` // 与字符串 s 比较

示例:

```

1  #include <iostream>
2  #include <string>
3
4  using namespace std;
5
6
7  //string的字符串比较
8  void test01(void) {
9      string s1 = "abc";
10     string s2 = "abc";
11
12     cout << s1.compare(s2) << endl;           //相等
13     cout << s1.compare("ab") << endl;         //不相等 且第一个不相等的地方的ASCII值s1大 返回1
14     cout << s1.compare("ad") << endl;         //不相等 且第一个不相等的地方的ASCII值s1小 返回-1
15 }
16
17 int main(void) {
18
19     test01();
20
21     return 0;
22 }
```

Fence 15

6.7 (7) 字符串存取

• string 中单个字符存取方式有两种:

- `char& operator[](int n);` // 通过 `[]` 方式获取字符
- `char& at(int n);` // 通过 `at` 方法获取字符

示例:

```

1  #include <iostream>
2  #include <string>
3
4  using namespace std;
5
6
7  //string的字符串存取
```



```

8 void test01(void) {
9     string s1 = "abcef";
10
11     //s1.size() 访问string s1的大小
12     cout << s1[1] << endl;
13     cout << s1.at(2) << endl;
14
15     s1[1] = 'a';
16     cout << s1[1] << endl;
17 }
18
19 int main(void) {
20
21     test01();
22
23     return 0;
24 }

```

Fence 16

6.8 (8) 字符串插入和删除

- 功能描述:

- 对 `string` 字符串进行插入和删除字符操作

- 函数原型:

- `string& insert(int pos, const char* s);` // 插入字符串
- `string& insert(int pos, const string& str);` // 插入字符串
- `string& insert(int pos, int n, char c);` // 在指定位置插入 n 个字符 c
- `string& erase(int pos, int n = npos);` // 删除从 pos 开始的 n 个字符

示例:

```

1 #include <iostream>
2 #include <string>
3
4 using namespace std;
5
6
7 //strin的字符串插入和删除
8 void test01(void) {
9     string s1 = "abcef";
10
11     cout << s1.insert(1, "111") << endl;
12
13     cout << s1.erase(1, 3) << endl;
14 }
15
16 int main(void) {
17

```

```

18     test01();
19
20     return 0;
21 }

```

Fence 17

6.9 (9) 子串的获取

- 功能描述:

- 从字符串中获取想要的子串

- 函数原型:

- `string substr(int pos = 0, int n = npos) const;` // 返回由 pos 开始的 n 个字符组成的字符串

示例:

```

1  #include <iostream>
2  #include <string>
3
4  using namespace std;
5
6
7  //strin的子串获取
8  void test01(void) {
9      string s1 = "abcef";
10     string s2 = "你好世界";
11
12     cout << s1.substr(1,3)<< endl;
13     cout << s2.substr(0, 4) << endl;           //一个汉字站2个字节
14 }
15
16 /* 过滤信息 */
17 void test02(void) {
18     string s1 = "nihaoshijie@qq.com";
19
20     string s2;
21     s2 = s1.substr(0, s1.find("@"));
22
23     cout << s2 << endl;
24 }
25
26 int main(void) {
27
28     test01();
29     test02();
30
31     return 0;
32 }

```

Fence 18

7. 2.3 deque

7.1 (1) 基本概念

- 功能：
 - 双端数组，可以对头尾进行插入删除操作
- deque与vector区别：
 - vector对于头部的插入删除效率低，数据量越大，效率越低
 - deque相对而言，对头部的插入删除速度比vector快
 - vector访问元素时的速度会比deque快，这和两者内部实现有关

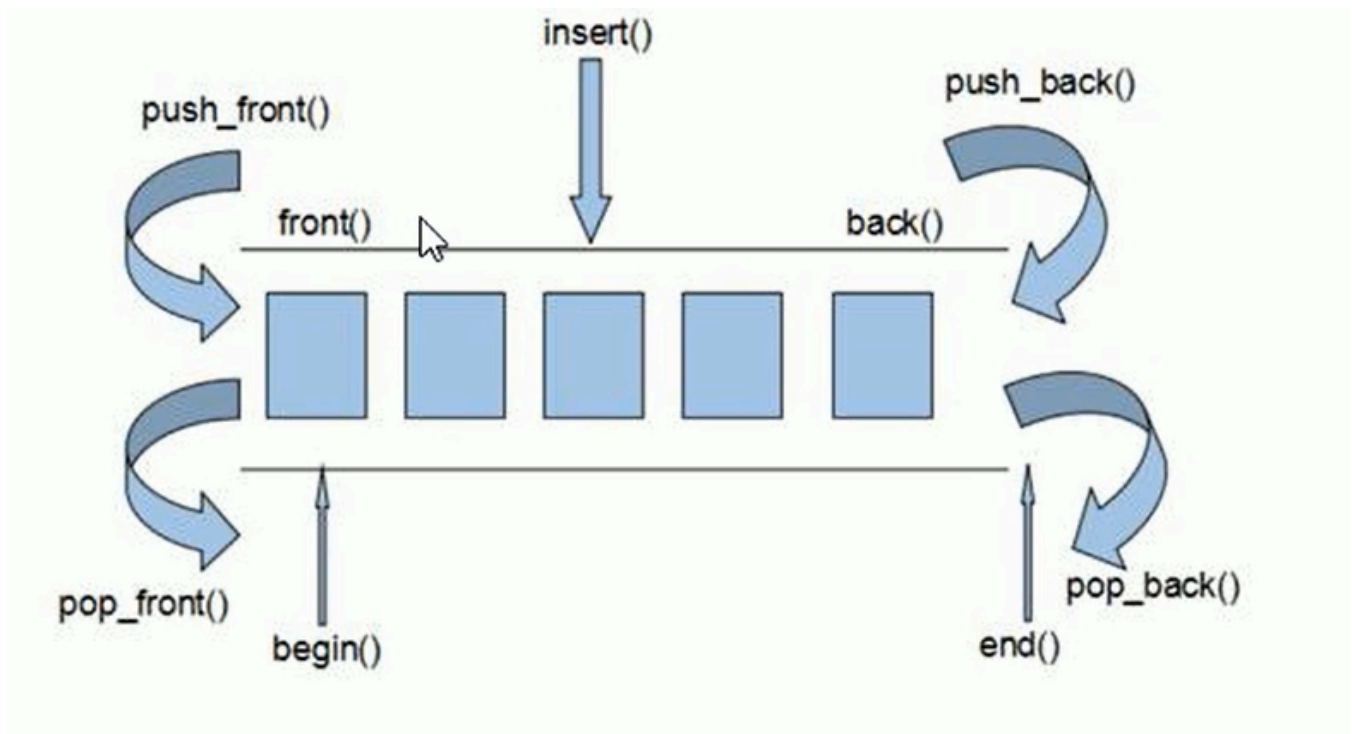


Figure 2

- deque内部工作原理：
 - deque内部有个中控器，维护每段缓冲区中的内容，缓冲区中存放真实数据。
 - 中控器维护的是每个缓冲区的地址，使得使用deque时像是一片连续的内存空间。

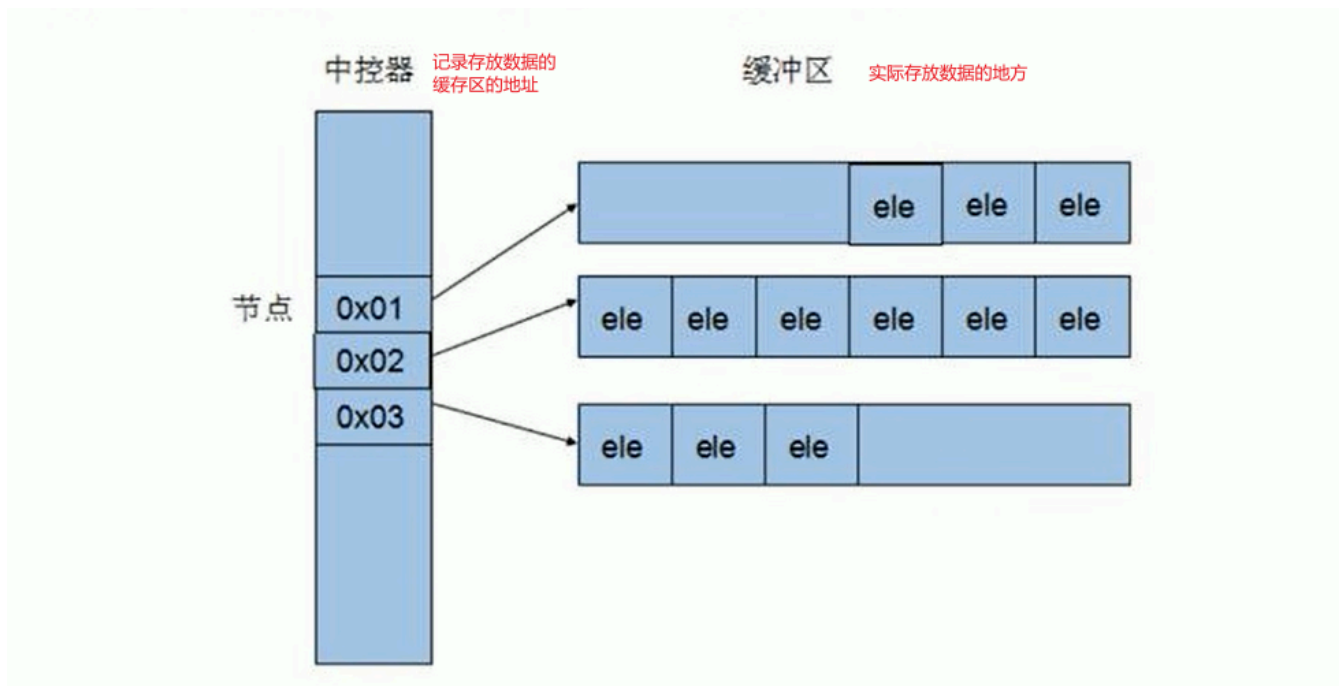


Figure 3

- deque容器的迭代器也是支持随机访问。

7.2 (2) 构造函数

- 功能描述：
 - deque容器构造
- 函数原型：
 - `deque<T> deqT;` // 默认构造形式
 - `deque(beg, end);` // 构造函数将[beg, end)区间中的元素拷贝给本身。
 - `deque(n, elem);` // 构造函数将n个elem拷贝给本身。
 - `deque(const deque &deq);` // 拷贝构造函数

示例：

```

1  #include <iostream>
2  #include <deque>
3  #include <algorithm>
4
5  using namespace std;
6
7  /* deque容器的构造函数*/
8
9  void printDeque(const deque<int>& d) {                                //使用
10     const 让d只读，防止函数中的操作进行误操作
11     for (deque<int>::const_iterator it = d.begin(); it != d.end(); it++) {
12         //const_iterator 常量正向迭代器
13         cout << *it << " ";
14     }
15 }
```

```

12     }
13     cout << endl;
14 }
15
16 void test01(void) {
17     deque<int> d1; //默认构造形式
18     for (int i = 0; i < 20; i++)
19     {
20         d1.push_back(i);
21     }
22     printDeque(d1);
23
24     deque<int> d2(d1.begin(),d1.end()); // 构造函数将[beg, end)区间中的元素拷贝给本身。
25     printDeque(d2);
26
27     deque<int> d3(3,100); // 构造函数将n个elem拷贝给本身。
28     printDeque(d3);
29
30
31     deque<int> d4(d3); // 拷贝构造函数
32     printDeque(d4);
33 }
34
35
36 int main(void) {
37
38     test01();
39
40     return 0;
41 }
42
43

```

Fence 19

7.3 (3) 赋值操作

- 功能描述：
 - 给deque容器进行赋值
- 函数原型：
 - `deque& operator=(const deque &deq);` // 重载赋值操作符
 - `assign(beg, end);` // 将[beg, end)区间中的数据拷贝赋值给本身。
 - `assign(n, elem);` // 将n个elem拷贝赋值给本身。

示例：

```

1  #include <iostream>
2  #include <deque>

```

```

3  #include <algorithm>
4
5  using namespace std;
6
7  /* deque容器的赋值操作 */
8
9  void printDeque(const deque<int>& d) {                                //使用
    const 让d只读，防止函数中的操作进行误操作
10     for (deque<int>::const_iterator it = d.begin(); it != d.end(); it++) {
        //const_iterator 常量正向迭代器
11         cout << *it << " ";
12     }
13     cout << endl;
14 }
15
16 void test01(void) {
17     deque<int> d1;
18     for (int i = 0; i < 20; i++)
19     {
20         d1.push_back(i);
21     }
22     printDeque(d1);
23
24     deque<int> d2;
25     d2 = d1;                                // 重载赋值操作符
26     printDeque(d2);
27
28     deque<int> d3;
29     d3.assign(d1.begin(), d1.end());        // 将[beg, end)区间中的数据拷贝赋值给本身。
30     printDeque(d3);
31
32     deque<int> d4;
33     d4.assign(2, 100);                      // 将n个elem拷贝赋值给本身。
34     printDeque(d4);
35 }
36
37
38 int main(void) {
39
40     test01();
41
42     return 0;
43 }
44
45

```

7.4 (4) 大小操作

- 功能描述:

- 对deque容器的大小进行操作

- 函数原型:

- `deque.empty();` // 判定容器是否为空
- `deque.size();` // 返回容器中元素的个数
- `deque.resize(num);` // 重新指定容器的长度为num, 如果容器变长, 则以默认值 (0) 填充新位置; 若容器变短, 则末尾超出容器长度的元素被删除
- `deque.resize(num, elem);` // 重新指定容器的长度为num, 如果容器变长, 则以elem值填充新位置; 若容器变短, 则末尾超出容器长度的元素被删除

示例:

```
1  #include <iostream>
2  #include <deque>
3  #include <algorithm>
4
5  using namespace std;
6
7  /* deque容器的大小操作 */
8
9  void printDeque(const deque<int>& d) {                                //使用
10     const 让d只读, 防止函数中的操作进行误操作
11     for (deque<int>::const_iterator it = d.begin(); it != d.end(); it++) {
12         //const_iterator 常量正向迭代器
13         cout << *it << " ";
14     }
15     cout << endl;
16 }
17
18 void test01(void) {
19     deque<int> d1;
20     for (int i = 0; i < 20; i++)
21     {
22         d1.push_back(i);
23     }
24     printDeque(d1);
25
26     if (d1.empty()) {
27         cout << "deque容器为空!" << endl;                // 判定容器是否为空
28     }
29     else
30     {
31         cout << "大小: " << d1.size() << endl;            // 返回容器中元素的个数
32         d1.resize(3);                                       // 重新指定容器的长度为num, 如果
33         // 容器变长, 则以默认值 (0) 填充新位置; 若容器变短, 则末尾超出容器长度的元素被删除
34         printDeque(d1);
35     }
```

```

32
33     d1.resize(4, 100);
34     printDeque(d1); // 重新指定容器的长度为num，如果
    容器变长，则以elem值填充新位置；若容器变短，则末尾超出容器长度的元素被删除
35
36
37     }
38 }
39
40
41 int main(void) {
42
43     test01();
44
45     return 0;
46 }
47
48

```

Fence 21

7.5 (5) 插入和删除

- **功能描述：**
 - 向deque容器中插入和删除数据
- **函数原型：**
 - **两端操作：**
 - `push_back(elem);` // 在容器尾部添加一个数据
 - `push_front(elem);` // 在容器头部插入一个数据
 - `pop_back();` // 删除容器最后一个数据
 - `pop_front();` // 删除容器第一个数据
 - **指定位置操作：**
 - `insert(pos, elem);` // 在pos位置(迭代器)插入一个elem元素，返回新数据的位置。
 - `insert(pos, n, elem);` // 在pos位置插入n个elem数据，无返回值。
 - `insert(pos, beg, end);` // 在pos位置插入[beg, end)区间的数据，无返回值。
 - `clear();` // 清空容器的所有数据
 - `erase(beg, end);` // 删除[beg, end)区间的数据，返回下一个数据的位置（返回的是迭代器）。
 - `erase(pos);` // 删除pos位置的数据，返回下一个数据的位置。

示例：

```

1  #include <iostream>
2  #include <deque>
3  #include <algorithm>

```



```

4
5 using namespace std;
6
7 /* deque容器的大小操作 */
8
9 void printDeque(const deque<int>& d) { //使用
const 让d只读，防止函数中的操作进行误操作
10     for (deque<int>::const_iterator it = d.begin(); it != d.end(); it++) {
//const_iterator 常量正向迭代器
11         cout << *it << " ";
12     }
13     cout << endl;
14 }
15
16
17 void test01(void) {
18     /* 两端插入 */
19     deque<int> d1,d2;
20     for (int i = 0; i < 20; i++)
21     {
22         d1.push_back(i); // 在容器尾部添加一个数据
23         d2.push_front(i * 10); // 在容器头部插入一个数据
24     }
25     printDeque(d1);
26     printDeque(d2);
27
28
29     d1.pop_back(); // 删除容器最后一个数据
30     d2.pop_front(); // 删除容器第一个数据
31     printDeque(d1);
32     printDeque(d2);
33
34     /* 指定插入 */
35     d1.insert(d1.begin(),1000); // 在pos位置插入一个elem元素，返回新数据的
位置。
36     printDeque(d1);
37
38     d2.insert(d2.end(), 3, 1000); // 在pos位置插入n个elem数据，无返回值。
39     printDeque(d2);
40
41     d1.insert(d1.begin(), d2.begin(), d2.end()); // 在pos位置插入[beg, end)区间的数据，无返
返回值。
42     printDeque(d1);
43
44     cout << "初始位置" << *(d1.begin()) << endl;
45     deque<int>::iterator a = d1.erase(d1.begin()); // 删除pos位置的数据，返回下一个数据的
位置。 通过偏移删除其他位置
46     printDeque(d1);
47     cout << *a << endl;
48
49     d1.erase(d1.begin(), d1.end()); // 删除[beg, end)区间的数据，返回下一个数据的
位置（返回的是迭代器）。
50

```

```

51     d2.clear(); // 清空容器的所有数据
52     printDeque(d2);
53
54 }
55
56
57
58 int main(void) {
59
60     test01();
61
62     return 0;
63 }
64
65

```

Fence 22

7.6 (6) 数据存取

- 功能描述：
 - 对 `deque` 中的数据的存取操作
- 函数原型：
 - `at(int idx);` // 返回索引 `idx` 所指的数据
 - `operator[];` // 返回索引 `idx` 所指的数据
 - `front();` // 返回容器中第一个数据元素
 - `back();` // 返回容器中最后一个数据元素

示例：

```

1  #include <iostream>
2  #include <deque>
3  #include <algorithm>
4
5  using namespace std;
6
7  /* deque容器的数据存取 */
8
9  void printDeque(const deque<int>& d) { //使用
10     const 让d只读，防止函数中的操作进行误操作
11     for (deque<int>::const_iterator it = d.begin(); it != d.end(); it++) {
12         //const_iterator 常量正向迭代器
13         cout << *it << " ";
14     }
15     cout << endl;
16 }

```

```

17 void test01(void) {
18     deque<int> d1, d2;
19     for (int i = 0; i < 20; i++)
20     {
21         d1.push_back(i);           // 在容器尾部添加一个数据
22         d2.push_front(i * 10);    // 在容器头部插入一个数据
23     }
24     printDeque(d1);
25     printDeque(d2);
26
27
28     /* []索引 */
29     for (int i = 0; i < d1.size(); i++)
30     {
31         cout << d1[i] << " ";
32     }
33     cout << endl;
34
35     /* at()索引 */
36     for (int i = 0; i < d2.size(); i++)
37     {
38         cout << d2.at(i) << " ";
39     }
40     cout << endl;
41
42
43     /* front() 访问返回容器中第一个数据元素 */
44     cout << d1.front() << endl;
45
46     /* back() 访问返回容器中最后一个数据元素 */
47     cout << d2.back() << endl;
48
49 }
50
51
52
53 int main(void) {
54
55     test01();
56
57     return 0;
58 }
59
60

```

- 示例：

38

```

39  int main(void) {
40
41      test01();
42
43      return 0;
44  }
45
46

```

Fence 24

8. 2.4 stack(栈)

8.1 (1) 基本概念

- 概念：stack是一种**先入后出**（First In Last Out, FILO）的数据结构，它只有一个出口。

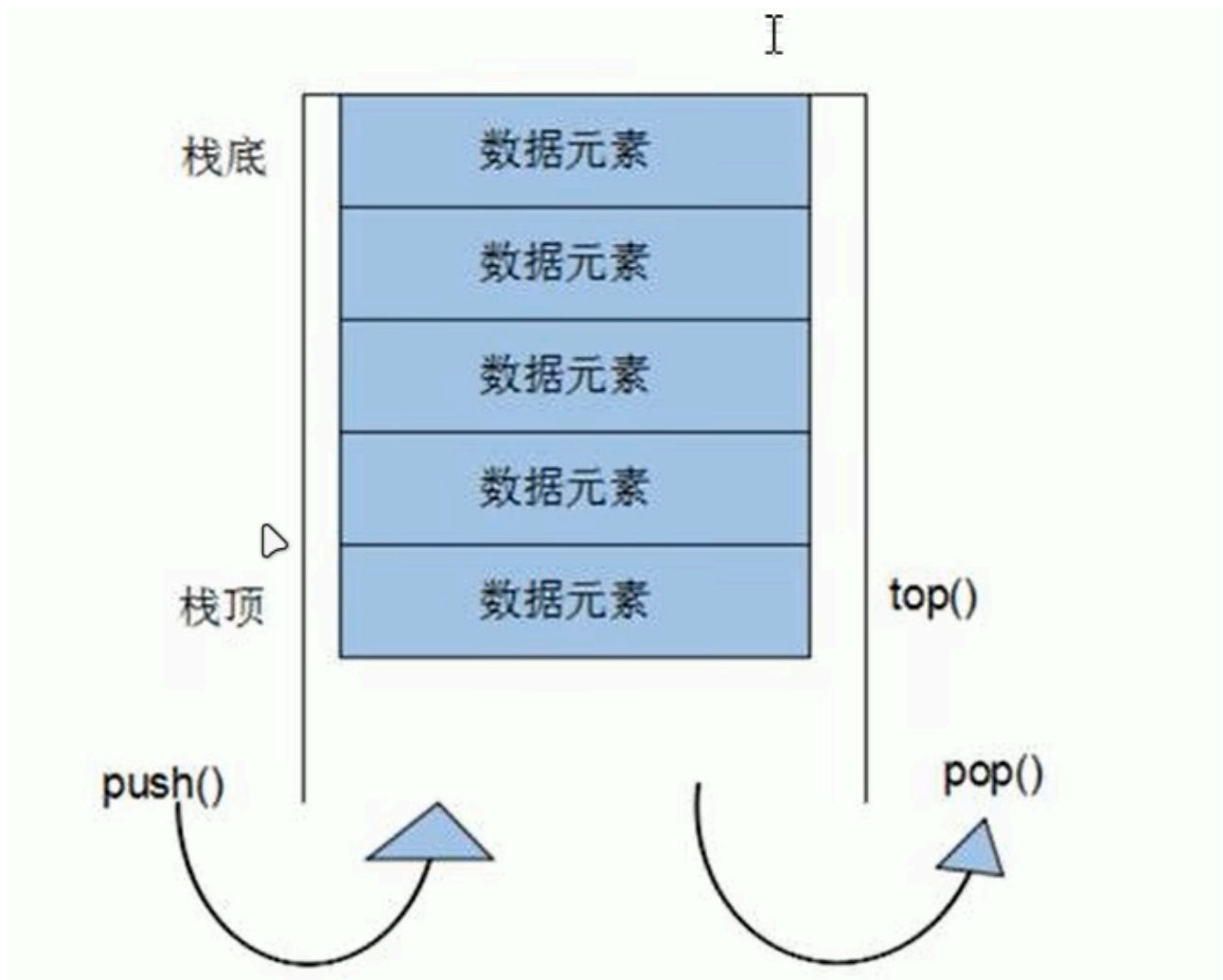


Figure 4

- 栈中只有顶端的元素才可以被外界使用，因此**栈不允许有遍历行为**。
- 向栈写入数据——入栈（push（））
- 从栈取出数据——出栈（pop（））

8.2 (2) 常用接口

- 功能描述：栈常用的对外接口
- 构造函数：
 - `stack<T> stk;` //默认构造
 - `stack(const stack &stk);` //拷贝构造
- 赋值操作：
 - `stack& operator=(const stack &stk);` //重载等号操作符
- 数据存取：
 - `push(elem);` //向栈顶添加元素
 - `pop();` //移除栈顶第一个元素
 - `top();` //返回栈顶元素
- 大小操作：
 - `empty();` //判断栈是否为空
 - `size();` //返回栈大小

示例：

```
1  #include <iostream>
2  #include <stack>
3
4  using namespace std;
5
6  /* statc的常用接口（先进后出的特点） */
7
8  void test01() {
9      stack<int> s;
10
11      //入栈
12      s.push(10);
13      s.push(20);
14      s.push(30);
15      s.push(40);
16
17      //查看栈中所有数据
18      while (!s.empty())          //不为空，显示
19      {
20          cout << "查看栈顶元素: " << s.top() << endl;
21
22          cout << "出栈前的大小: " << s.size() << endl;
23
24          //出栈
25          s.pop();
26      }
```

```

27         cout << "出栈后的大小: " << s.size() << endl;
28     }
29
30
31 }
32
33 int main() {
34     test01();
35
36     return 0;
37 }

```

Fence 25

9.2.5 queue (队列)

9.1 (1) 基本概念

- 概念：Queue是一种**先进先出**（First In First Out, FIFO）的数据结构，它有两个出口。

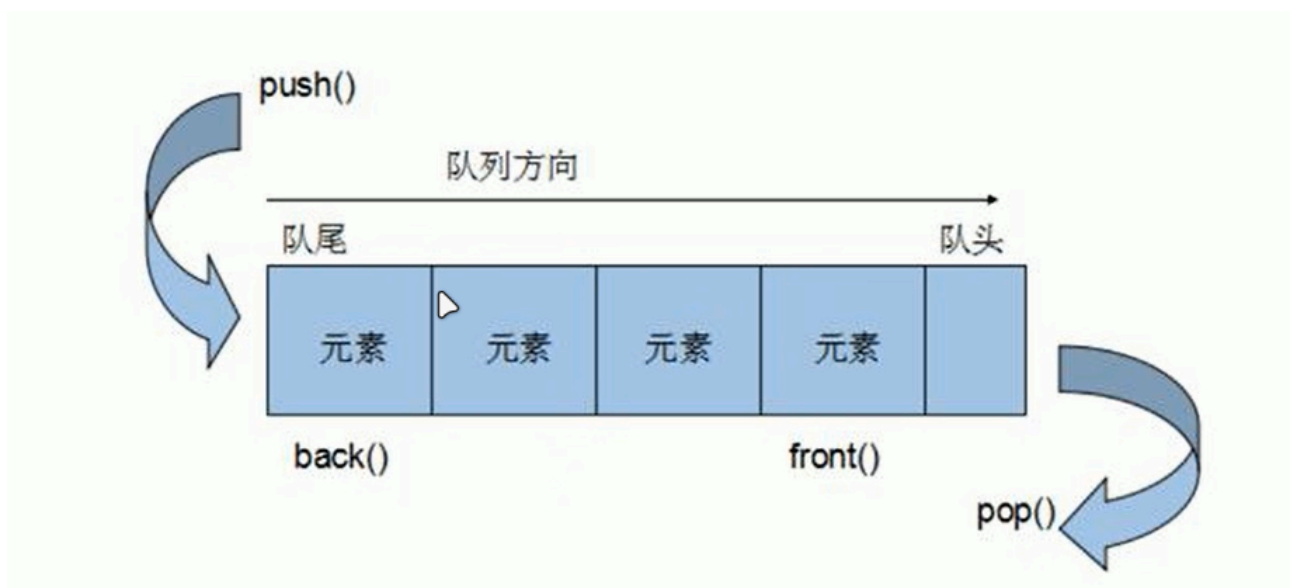


Figure 5

- 队列容器允许从一端新增元素，从另一端移除元素。
- 队列中只有队头和队尾才可以被外界使用，因此**队列不允许有遍历行为**。
- 队列中进数据——入队（push），只能队尾进数据。
- 队列中出数据——出队（pop）。只能队头出数据。

9.2 (2) 常用接口

- 功能描述：栈容器常用的对外接口
- 构造函数：
 - `queue<T> que;` // queue采用模板类实现，queue对象的默认构造形式
 - `queue(const queue &que);` // 拷贝构造函数

- 赋值操作：
 - `queue& operator=(const queue &q);` // 重载等号操作符
- 数据存取：
 - `push(elem);` // 往队尾添加元素
 - `pop();` // 从队头移除第一个元素
 - `back();` // 返回最后一个元素
 - `front();` // 返回第一个元素
- 大小操作：
 - `empty();` // 判断堆栈是否为空
 - `size();` // 返回栈的大小

示例:

```
1  #include <iostream>
2  #include <queue>
3
4  using namespace std;
5
6  /* queue的常用接口（先进先出） */
7
8  void test01() {
9      queue<int> q;
10
11     //入队
12     q.push(10);
13     q.push(20);
14     q.push(30);
15     q.push(40);
16
17     cout << "队列大小" << q.size() << endl;
18
19     //队列不为空，查看队尾、对头，出队
20     while (!q.empty())
21     {
22
23         cout << "对头: " << q.front() << endl;
24         cout << "对尾: " << q.back() << endl;
25
26         //出队
27         q.pop();
28
29         cout << endl;
30     }
31
32     cout << "队列大小" << q.size() << endl;
33
34 }
```



```

35
36 int main() {
37     test01();
38
39     return 0;
40 }

```

Fence 26

10.2.6 list

10.1 (1) 基本概念

- 功能：将数据进行链式存储
- 优点：
 - 对比数组，对于任意的位置进行快速的插入或者删除元素（链表执行插入和删除操作十分方便，修改指针即可，不需要移动大量元素。）
 - 能分散存储，采用动态存储分配，不会造成内存浪费和溢出。
- 缺点：
 - 容器遍历速度，没有数组快；链表灵活，但是空间（指针域）和时间（遍历）额外耗费较大。
 - 占用空间比数组大。
- 链表（list）是一种物理存储单元上**非连续的存储结构**，数据元素的逻辑顺序是通过链表中的指针链接实现的。
- 链表的组成：由一系列**结点**组成。
- 结点的组成：一个存储数据元素的**数据域**，另一个存储下一个节点地址的**指针域**。
- STL中的链表是一个**双向循环链表**（一个结点中指针域，维护两个指针，存储上一个结点的地址——prev；另外一个存储下一个结点的地址——next）。

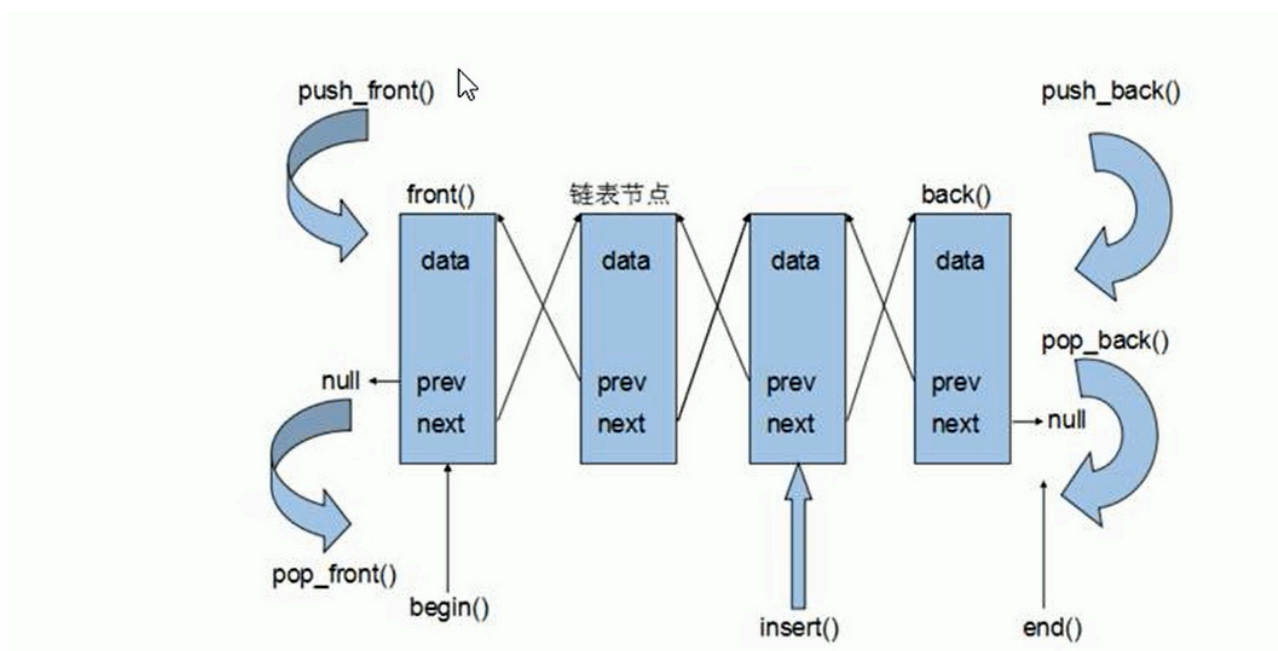


Figure 6

- 由于链表的存储方式并不是连续的内存空间，因此链表list中的迭代器只支持前移和后移，属于双向迭代器。
- List有一个重要的性质，插入操作和删除操作都不会造成原有list迭代器的失效，这在vector是不成立的。

总结：STL中List和Vector是两个最常被使用的容器，各有优缺点。

10.2 (2) 构造函数

- 功能描述：
 - 创建list容器
- 函数原型：
 - `list<T> lst;` //list采用模板类实现对象的默认构造形式
 - `list(beg, end);` //构造函数将[beg, end)区间中的元素拷贝给本身
 - `list(n, elem);` //构造函数将n个elem拷贝给本身
 - `list(const list &lst);` //拷贝构造函数

示例：

```

1  #include <iostream>
2  #include <list>
3
4  using namespace std;
5
6  /* list的构造函数 */
7
8  void printList(const list<int> &L) {
9      for(list<int>::const_iterator it = L.begin(); it != L.end(); it++) {
10         cout << *it << " ";
11     }
12     cout << endl;
13 }
14
15 void test01() {
16     list<int> l1;                                //默认构造
17
18     l1.push_back(10);
19     l1.push_back(20);
20     l1.push_back(30);
21     l1.push_back(40);
22
23     printList(l1);
24
25     list<int> l2(l1.begin(), l1.end());           //构造函数(区间)
26     printList(l2);
27
28     list<int> l3(l2);                             //拷贝构造

```

```

29     printList(l3);
30
31     list<int> l4(10,100);           //构造函数(n个m的形式)
32     printList(l4);
33 }
34
35 int main() {
36     test01();
37
38     return 0;
39 }

```

Fence 27

10.3 (3) 赋值和交换

- 功能描述：
 - 给list容器进行赋值，以及交换list容器
- 函数原型：
 - `assign(beg, end)`；将[beg, end)区间中的数据拷贝赋值给本身。
 - `assign(n, elem)`；将n个elem拷贝赋值给本身。
 - `list& operator=(const list &lst)`；重载等号操作符
 - `swap(lst)`；将lst与本身的元素互换。

示例：

```

1  #include <iostream>
2  #include <list>
3
4  using namespace std;
5
6  /* list的赋值和交换 */
7
8  void printList(const list<int>& L) {
9      for (list<int>::const_iterator it = L.begin(); it != L.end(); it++) {
10         cout << *it << " ";
11     }
12     cout << endl;
13 }
14
15 void test01() {
16     list<int> l1;           //默认构造
17
18     l1.push_back(10);
19     l1.push_back(20);
20     l1.push_back(30);

```

```

21     l1.push_back(40);
22
23     printList(l1);
24
25     list<int> l2;                                //将[beg, end)区间中的数据拷贝赋值给本身。
26     l2.assign(l1.begin(), l1.end());
27     printList(l2);
28
29     list<int> l3;                                // 将n个elem拷贝赋值给本身。
30     l3.assign(10, 100);
31     printList(l3);
32
33     list<int> l4;                                //重载等号操作符
34     l4 = l3;
35     printList(l4);
36
37     l1.swap(l4);                                //将l1与本身的元素互换。
38     printList(l1);
39     printList(l4);
40 }
41
42 int main() {
43     test01();
44
45     return 0;
46 }

```

Fence 28

10.4 (4) 大小操作

- 功能描述：
 - 对list容器的大小进行操作
- 函数原型：
 - `size()`; //返回容器中元素的个数
 - `empty()`; //判断容器是否为空
 - `resize(num)`; //重新指定容器的长度为num，若容器变长，则以默认值填充新位置。
//如果容器变短，则末尾超出容器长度的元素被删除。
 - `resize(num, elem)`; //重新指定容器的长度为num，若容器变长，则以elem值填充新位置。
//如果容器变短，则末尾超出容器长度的元素被删除。

示例：

```

1  #include <iostream>
2  #include <list>
3

```

```

4   using namespace std;
5
6   /* list的大小操作 */
7
8   void printList(const list<int>& L) {
9       for (list<int>::const_iterator it = L.begin(); it != L.end(); it++) {
10          cout << *it << " ";
11      }
12      cout << endl;
13  }
14
15  void test01() {
16      list<int> l1;                                //默认构造
17
18      l1.push_back(10);
19      l1.push_back(20);
20      l1.push_back(30);
21      l1.push_back(40);
22
23      printList(l1);
24
25      if(!l1.empty())                               //判断容器是否为空
26      {
27          cout << "大小: " << l1.size() << endl; //返回容器中元素的个数
28      }
29
30      //重新指定容器的长度为num，若容器变长，则以默认值填充新位置。如果容器变短，则末尾超出容器长度的元
      素被删除。
31      l1.resize(8);
32      printList(l1);
33
34      cout << endl;
35
36      l1.resize(3);
37
38      printList(l1);
39
40      cout << endl;
41
42      //重新指定容器的长度为num，若容器变长，则以elem值填充新位置。如果容器变短，则末尾超出容器长度的元
      素被删除。
43      l1.resize(8,80);
44      printList(l1);
45
46      cout << endl;
47
48      l1.resize(3,80);
49
50      printList(l1);
51
52      cout << endl;
53
54  }

```

```

55
56     int main() {
57         test01();
58
59         return 0;
60     }

```

Fence 29

10.5 (5) 插入和删除

- 功能描述：
 - 对list容器进行数据的插入和删除
- 函数原型：
 - `push_back(elem);` //在容器尾部加入一个元素
 - `pop_back();` //删除容器中最后一个元素
 - `push_front(elem);` //在容器开头插入一个元素
 - `pop_front();` //从容器开头移除第一个元素
 - `insert(pos, elem);` //在pos位置插入elem元素的拷贝，返回新数据的位置。
 - `insert(pos, n, elem);` //在pos位置插入n个elem数据，无返回值。
 - `insert(pos, beg, end);` //在pos位置插入[beg, end)区间的数据，无返回值。
 - `clear();` //移除容器的所有数据
 - `erase(beg, end);` //删除[beg, end)区间的数据，返回下一个数据的位置。
 - `erase(pos);` //删除pos位置的数据，返回下一个数据的位置。
 - `remove(elem);` //删除容器中所有与elem值匹配的元素。

示例：

```

1     #include <iostream>
2     #include <list>
3
4     using namespace std;
5
6     /* list的插入和删除 */
7
8     void printList(const list<int>& L) {
9         for (list<int>::const_iterator it = L.begin(); it != L.end(); it++) {
10             cout << *it << " ";
11         }
12         cout << endl;
13     }
14

```

```

15
16 void test01() {
17     list<int> l1; //默认构造
18
19     l1.push_back(10); //在容器尾部加入一个元素
20     l1.push_back(20);
21     l1.push_back(30);
22     l1.push_back(40);
23
24     printList(l1);
25
26     /* 插入 */
27     l1.push_front(90); //在容器开头插入一个元素
28     printList(l1);
29
30     list<int>::iterator it = l1.begin(); //在pos位置插入elem元素的拷贝，返回新数据的
位置。
31     it = l1.insert(++it, 80); //不能it+1，没有对应的int重载
32     printList(l1);
33
34     l1.insert(it, 4, 70); //在pos位置插入n个elem数据，无返回值。
35     printList(l1);
36
37     l1.insert(it, l1.begin(), l1.end()); //在pos位置插入[beg, end)区间的数据，无返
返回值。
38     printList(l1);
39
40     cout << endl;
41
42     /* 删除 */
43     l1.pop_back(); //删除容器中最后一个元素
44     printList(l1);
45
46     l1.pop_front(); //从容器开头移除第一个元素
47     printList(l1);
48
49     if (!l1.empty()) {
50         l1.erase(it); //删除pos位置的数据，返回下一个数据的位置。
it存的是80的位置
51         printList(l1);
52
53         l1.remove(90); //删除容器中所有与elem值匹配的元素。
54         printList(l1);
55
56         it = l1.begin();
57
58         l1.erase(l1.begin(), ++it); //删除[beg, end)区间的数据，返回下一个数据
的位置。l1.erase(it, ++it)会出现，第二个it未定义行为
59         printList(l1);
60
61         l1.clear();
62         if (l1.empty())
63             cout << "清空" << endl;

```

```

64     }
65 }
66
67 int main() {
68     test01();
69
70     return 0;
71 }

```

Fence 30

10.6 (6) 数据存取

- 功能描述
 - 对list容器中的数据进行存取（非连续空间——不能随机访问）
- 函数原型
 - `front()`; //返回第一个元素。
 - `back()`; //返回最后一个元素。

示例:

```

1  #include <iostream>
2  #include <list>
3
4  using namespace std;
5
6  /* list的数据存取 */
7
8  void printList(const list<int>& L) {
9      for (list<int>::const_iterator it = L.begin(); it != L.end(); it++) {
10         cout << *it << " ";
11     }
12     cout << endl;
13 }
14
15
16 void test01() {
17     list<int> l1; //默认构造
18
19     l1.push_back(10); //在容器尾部加入一个元素
20     l1.push_back(20);
21     l1.push_back(30);
22     l1.push_back(40);
23
24     printList(l1);
25
26     //不能使用[]访问元素，且没有at接口 迭代器也不支持随机访问 (it = it + 1)
27     cout << "第一个元素: " << l1.front() << endl;
28     cout << "第二个元素: " << l1.back() << endl;

```



```

29     }
30
31     int main() {
32         test01();
33
34         return 0;
35     }

```

Fence 31

10.7 (7) 反转和排序

- 功能描述：
 - 将容器中的元素反转，以及将容器中的数据进行排序
- 函数原型：
 - `reverse();` // 反转链表
 - `sort();` // 链表排序

示例：

```

1  #include <iostream>
2  #include <list>
3
4  using namespace std;
5
6  /* list的反转和排序 */
7
8  void printList(const list<int>& L) {
9      for (list<int>::const_iterator it = L.begin(); it != L.end(); it++) {
10         cout << *it << " ";
11     }
12     cout << endl;
13 }
14
15
16 bool myCompare(int v1,int v2) {
17     //更换成降序
18     return v1 > v2;
19 }
20
21 void test01() {
22     list<int> l1; //默认构造
23
24     l1.push_back(10); //在容器尾部加入一个元素
25     l1.push_back(40);
26     l1.push_back(20);
27     l1.push_back(30);
28
29     printList(l1);

```

```

30
31     l1.reverse();                // 反转链表
32     printList(l1);
33
34     l1.sort();                    // 链表排序（从小到大） 不能使用算法中的
    sort(l1,begin(),l1.end()); 所有不支持随机访问迭代器的容器，不能使用标准算法
35     printList(l1);                //不支持随机访问迭代器的容器，内部会提供对应
    的算法
36
37     l1.sort(myCompare);           //排序（从大到小） 提供仿函数或者函数
38     printList(l1);
39 }
40
41 int main() {
42     test01();
43
44     return 0;
45 }

```

Fence 32

11. 2.7 set/multiset（集合容器）

11.1 (1) 基本概念

- 简介：
 - 所有元素都会在插入时自动被排序。
- 本质：
 - set/multiset属于**关联式容器**，底层结构是用**二叉树**实现。
- set和multiset区别：
 - set不允许容器中有重复的元素。
 - multiset允许容器中有重复的元素。

11.2 (2) 构造函数和赋值

- 功能描述：创建set容器以及赋值
- 构造：
 - `set<T> st;` //默认构造函数
 - `set(const set &st);` //拷贝构造函数
- 赋值：
 - `set& operator=(const set &st);` //重载等号操作符

示例:

```
1  #include <iostream>
2  #include <set>
3
4  using namespace std;
5
6  /* set/multiset的构造函数和赋值 */
7
8  void printSet(const set<int>& L) {
9      for (set<int>::const_iterator it = L.begin(); it != L.end(); it++) {
10         cout << *it << " ";
11     }
12     cout << endl;
13 }
14
15 void printMultiset(const multiset<int>& L) {
16     for (multiset<int>::const_iterator it = L.begin(); it != L.end(); it++) {
17         cout << *it << " ";
18     }
19     cout << endl;
20 }
21
22
23 void test01() {
24     set<int> s1;                //默认构造函数
25     multiset<int> m1;
26
27     //set没有push_back, 只能通过insert()添加数据
28     s1.insert(10);
29     s1.insert(40);
30     s1.insert(30);
31     s1.insert(20);
32     s1.insert(30);
33
34     m1.insert(10);
35     m1.insert(10);
36     m1.insert(5);
37
38     printSet(s1);              //set不允许容器中有重复的元素, 且会自动排序
39     printMultiset(m1);        //multiset允许容器中有重复的元素, 且会自动排序
40
41     set<int> s2(s1);          //拷贝构造函数
42     printSet(s2);
43
44     set<int> s3 = s1;         //重载等号操作符
45     printSet(s3);
46
47 }
48
49 int main() {
50     test01();
51 }
```

```
52     return 0;
53 }
```

Fence 33

11.3 (3) 大小和交换

- 功能描述：
 - 统计set容器大小以及交换set容器
- 函数原型：
 - `size()`; // 返回容器中元素的数量
 - `empty()`; // 判断容器是否为空
 - `swap(st)`; // 交换两个集合容器

示例:

```
1  #include <iostream>
2  #include <set>
3
4  using namespace std;
5
6  /* set/multiset的大小和交换 */
7
8  void printSet(const set<int>& L) {
9      for (set<int>::const_iterator it = L.begin(); it != L.end(); it++) {
10         cout << *it << " ";
11     }
12     cout << endl;
13 }
14
15
16
17 void test01() {
18     set<int> s1;                //默认构造函数
19     set<int> s2;
20
21     //set没有push_back,只能通过insert()添加数据
22     s1.insert(10);
23     s1.insert(40);
24     s1.insert(30);
25     s1.insert(20);
26     s1.insert(30);
27
28     s2.insert(10);
29     s2.insert(10);
30     s2.insert(5);
31
32     printSet(s1);
```

```

33     printSet(s2);
34
35     if (!s1.empty() && !s2.empty()) {
36         cout << "s1的大小: " << s1.size() << endl;
37         cout << "s2的大小: " << s2.size() << endl;
38
39         s1.swap(s2);
40
41         cout << "交换后: " << endl;
42
43         printSet(s1);
44         printSet(s2);
45
46     }
47 }
48
49 int main() {
50     test01();
51
52     return 0;
53 }

```

Fence 34

11.4 (4) 插入和删除

- 功能描述:

- set容器进行插入数据和删除数据

- 函数原型:

- `insert(elem);` // 在容器中插入元素。
- `clear();` // 清除所有元素。
- `erase(pos);` // 删除pos指代的元素，返回下一个元素的迭代器。
- `erase(beg, end);` // 删除区间[beg, end)的所有元素，返回下一个元素的迭代器。
- `erase(elem);` // 删除容器中值为elem的元素。

示例:

```

1  #include <iostream>
2  #include <set>
3
4  using namespace std;
5
6  /* set/multiset的插入和删除 */
7
8  void printSet(const set<int>& L) {

```

```

9      for (set<int>::const_iterator it = L.begin(); it != L.end(); it++) {
10          cout << *it << " ";
11      }
12      cout << endl;
13  }
14
15
16
17  void test01() {
18      set<int> s1;                //默认构造函数
19      set<int> s2;
20
21      //set没有push_back,只能通过insert()添加数据
22      s1.insert(10);              // 在容器中插入元素。
23      s1.insert(40);
24      s1.insert(30);
25      s1.insert(20);
26      s1.insert(30);
27
28      s2.insert(10);
29      s2.insert(10);
30      s2.insert(5);
31
32      printSet(s1);
33      printSet(s2);
34
35      set<int>::iterator it = s1.begin();
36      s1.erase(it);              // 删除pos指代的元素, 返回下一个元素的迭代器。
37      printSet(s1);
38
39      s1.erase(20);
40      printSet(s1);              // 删除容器中值为elem的元素。
41
42      s1.erase(s1.begin(), it);  // 删除区间[beg, end)的所有元素, 返回下一个元素的迭代器。
43      printSet(s1);
44
45      s2.clear();                // 清除所有元素。
46      printSet(s2);
47  }
48
49  int main() {
50      test01();
51
52      return 0;
53  }

```

11.5 (5) 查找和统计

- 功能描述：
 - 对set容器进行查找数据以及统计数据
- 函数原型：
 - `find(key);` // 查找key是否存在，若存在，返回该键的元素的迭代器；若不存在，返回`set.end()`
 - `count(key);` // 统计key的元素个数，对于set而言，只可能为1或者0；对于multiset来说，可能会出现大于1

示例：

```
1  #include <iostream>
2  #include <set>
3
4  using namespace std;
5
6  /* set/multiset的查找和统计 */
7
8  void printSet(const set<int>& L) {
9      for (set<int>::const_iterator it = L.begin(); it != L.end(); it++) {
10         cout << *it << " ";
11     }
12     cout << endl;
13 }
14
15
16
17 void test01() {
18     set<int> s1;                //默认构造函数
19     set<int> s2;
20
21     //set没有push_back, 只能通过insert()添加数据
22     s1.insert(10);              // 在容器中插入元素。
23     s1.insert(40);
24     s1.insert(30);
25     s1.insert(20);
26     s1.insert(30);
27
28     s2.insert(10);
29     s2.insert(10);
30     s2.insert(5);
31
32     printSet(s1);
33     printSet(s2);
34
35     // 查找key是否存在，若存在，返回该键的元素的迭代器；若不存在，返回set.end()
36     set<int>::iterator it = s1.find(10);
37     if (it != s1.end()) {
38         cout << "找到" << endl;
```

```

39     }
40     else
41     {
42         cout << "未找到" << endl;
43     }
44
45     // 统计key的元素个数，对于set而言，只可能为1或者0；对于multiset来说，可能会出现大于1
46     int num = s1.count(10);
47     cout << num << endl;
48 }
49
50 int main() {
51     test01();
52
53     return 0;
54 }

```

Fence 36

11.6 (6) set与multiset的区别

- 区别：
 - set不可以插入重复数据，而multiset可以
 - set插入数据的同时会返回插入结果，表示插入是否成功
 - multiset不会检测数据，因此可以插入重复数据

示例：

```

1  #include <iostream>
2  #include <set>
3
4  using namespace std;
5
6  /* set/multiset的区别*/
7
8  void printSet(const set<int>& L) {
9      for (set<int>::const_iterator it = L.begin(); it != L.end(); it++) {
10         cout << *it << " ";
11     }
12     cout << endl;
13 }
14
15 void printSet(const multiset<int>& L) {
16     for (multiset<int>::const_iterator it = L.begin(); it != L.end(); it++) {
17         cout << *it << " ";
18     }
19     cout << endl;
20 }
21

```



```

22 void test01() {
23     set<int> s1; //默认构造函数
24     multiset<int> s2;
25
26     //set没有push_back,只能通过insert()添加数据
27     s1.insert(10); // 在容器中插入元素。
28     s1.insert(40);
29     s1.insert(30);
30     s1.insert(20);
31     s1.insert(30);
32
33     s2.insert(10); //直接返回的是迭代器
34     s2.insert(10);
35     s2.insert(5);
36
37     printSet(s1);
38     printSet(s2);
39
40     //insert里面使用了对组, pair<迭代器, 插入成功的布尔值>
41     pair<set<int>::iterator, bool> re = s1.insert(70);
42
43     if (re.second) { //通过second属性查看
44         cout << "插入成功" << endl;
45     }
46     else
47     {
48         cout << "插入失败" << endl;
49     }
50
51
52
53 }
54
55 int main() {
56     test01();
57
58     return 0;
59 }

```

Fence 37

11.7 (7) 容器排序

- 主要技术点：
 - 利用仿函数，改变排序规则

自定义数据类型，必须指定排序规则。

示例1——set存放内置的数据类型：

```

1  #include <iostream>
2  #include <set>
3
4  using namespace std;
5
6  /* set存放内置的数据类型 */
7
8  //仿函数 本质就是一个类
9  class MyCompareF
10 {
11 public:
12     //添加const原因 explicit set(const Compare& comp, const Allocator& alloc =
Allocator());
13     bool operator()(int v1, int v2) const{
14         return v1 > v2;
15     }
16
17 };
18
19 void printSet(const set<int>& L) {
20     for (set<int>::const_iterator it = L.begin(); it != L.end(); it++) {
21         cout << *it << " ";
22     }
23     cout << endl;
24 }
25
26 void printSet(const set<int, MyCompareF>& L) {
27     for (set<int, MyCompareF>::const_iterator it = L.begin(); it != L.end(); it++) {
28         cout << *it << " ";
29     }
30     cout << endl;
31 }
32
33
34
35
36
37
38 void test01() {
39     set<int> s1;
40
41     s1.insert(10);
42     s1.insert(20);
43     s1.insert(30);
44     s1.insert(40);
45     printSet(s1);                //默认从小到大
46
47     /* 更改排序规则--使用仿函数 */
48     set<int, MyCompareF> s2;
49
50     s2.insert(10);
51     s2.insert(20);
52     s2.insert(30);

```

```

53     s2.insert(40);
54     printSet(s2);           //从大到小
55
56 }
57
58 int main() {
59     test01();
60
61     return 0;
62 }

```

Fence 38

示例2——自定义数据类型指定排序规则：

```

1  #include <iostream>
2  #include <set>
3
4  using namespace std;
5
6  /* set自定义数据类型指定排序规则 */
7
8  class Person
9  {
10 public:
11     Person(char a, int age) {
12         this->m_a = a;
13         this->m_age = age;
14     }
15
16     char m_a;
17     int m_age;
18 private:
19
20 };
21
22
23
24 //仿函数 本质就是一个类
25 class MyCompareF
26 {
27 public:
28
29     bool operator()(const Person &v1, const Person &v2) const {
30         return v1.m_age > v2.m_age;
31     }
32
33 };
34
35 void printSet(const set<Person, MyCompareF>& L) {
36     for (set<Person, MyCompareF>::const_iterator it = L.begin(); it != L.end(); it++)
37     {

```

```

37         cout << "姓名: " << it->m_a << " " << "年龄: " << it->m_age << endl;;
38     }
39     cout << endl;
40 }
41
42
43 void test01() {
44     set<Person, MyCompareF> s1;
45     Person p1('A', 10);
46     Person p2('B', 20);
47     Person p3('C', 30);
48     Person p4('D', 50);
49     s1.insert(p1);
50     s1.insert(p2);
51     s1.insert(p3);
52     s1.insert(p4);
53
54     printSet(s1);
55
56
57
58 }
59
60 int main() {
61     test01();
62
63     return 0;
64 }

```

Fence 39

12. 2.8 pair(队组)

12.1 (1) 创建

- 功能描述:
 - 成对出现的数据，利用对组可以返回两个数据
- 两种创建方式:
 - `pair<type, type> p (value1, value2);`
 - `pair<type, type> p = make_pair(value1, value2);`

示例:

```

1  #include <iostream>
2  #include <set>
3
4  using namespace std;
5

```

```

6  /* pair 队组*/
7
8  void printSet(const set<int>& L) {
9      for (set<int>::const_iterator it = L.begin(); it != L.end(); it++) {
10         cout << *it << " ";
11     }
12     cout << endl;
13 }
14
15
16
17 void test01() {
18     //默认构造
19     pair<char, int> p('A',20);
20     cout << "第一个数据: " << p.first << endl;
21     cout << "第二个数据: " << p.second << endl;
22
23     //make_pair()
24     pair<char, int> p1 = make_pair('B', 10);
25     cout << "第一个数据: " << p1.first << endl;
26     cout << "第二个数据: " << p1.second << endl;
27 }
28
29 int main() {
30     test01();
31
32     return 0;
33 }

```

Fence 40

13. 2.9 map/multimap

13.1 (1) 基本概念

- 简介：
 - map 中所有元素都是 pair。
 - pair 中第一个元素是 key（键值），起到索引作用，第二个元素是 value（实值）。
 - 所有元素会根据键元素自动排序。
- 本质：
 - map/multimap 属于关联式容器，底层结构是用二叉树实现。
- 优点：
 - 可以根据 key 值快速找到 value 值。
- map 和 multimap 区别：
 - map 不允许容器中有重复键值元素。

- `multimap` 允许容器中有重复键值元素。

13.2 (2) 构造和赋值

- 功能描述：
 - 对 `map` 容器进行构造和赋值操作。
- 函数原型：
 - 构造：
 - `map<T1, T2> mp;` // `map` 默认构造函数
 - `map(const map &mp);` // 拷贝构造函数
 - 赋值：
 - `map& operator=(const map &mp);` // 重载等号操作符

示例:

```
1  #include <iostream>
2  #include <map>
3
4  using namespace std;
5
6  /* map的构造和赋值 */
7
8
9  void printMap(const map<int,int> &m) {
10     for (map<int, int>::const_iterator it = m.begin(); it != m.end();it++) {
11         cout << it->first << " " << it->second << endl;
12     }
13     cout << endl;
14 }
15
16 void test01() {
17     map<int, int> m1;
18
19     //map::insert 方法期待一个 std::pair 类型来插入键值对。
20     m1.insert(pair<int,int> (1,1));
21     m1.insert(pair<int, int>(3, 1));
22     m1.insert(pair<int, int>(2, 3));
23     m1.insert(pair<int, int>(3, 4));
24     m1.insert({ 4,5 });
25     printMap(m1);
26
27     map<int, int> m2(m1);          // 拷贝构造函数
28     printMap(m2);
29
30     map<int, int> m3;
```

```

31     m3 = m2;                                // 重载等号操作符
32     printMap(m3);
33 }
34
35 int main() {
36     test01();
37
38     return 0;
39 }

```

Fence 41

13.3 (3) 大小和交换

- 功能描述：
 - 用于统计 `map` 容器大小以及交换 `map` 容器。
- 函数原型：
 - `size()` //返回容器中元素的数目。
 - `empty()` // 判断容器是否为空。
 - `swap(st)` // 交换两个集合容器。

示例：

```

1  #include <iostream>
2  #include <map>
3
4  using namespace std;
5
6  /* map的大小和交换 */
7
8
9  void printMap(const map<int, int>& m) {
10     for (map<int, int>::const_iterator it = m.begin(); it != m.end(); it++) {
11         cout << it->first << " " << it->second << endl;
12     }
13     cout << endl;
14 }
15
16 void test01() {
17     map<int, int> m1;
18
19     //map::insert 方法期待一个 std::pair 类型来插入键值对。
20     m1.insert(pair<int, int>(1, 1));
21     m1.insert(pair<int, int>(3, 1));
22     m1.insert(pair<int, int>(2, 3));
23     m1.insert(pair<int, int>(3, 4));
24     m1.insert({ 4,5 });
25     printMap(m1);

```

```

26
27     if (!m1.empty()) {                                // 判断容器是否为空。
28         cout << "大小: " << m1.size() << endl;      //返回容器中元素的数目。
29     }
30
31     map<int, int> m2;
32     m2.insert(pair<int, int>(10, 1));
33     m2.insert(pair<int, int>(30, 1));
34     m2.insert(pair<int, int>(20, 3));
35     printMap(m2);
36
37     cout << "交换: " << endl;
38     m1.swap(m2);
39     printMap(m1);
40     printMap(m2);
41
42
43 }
44
45 int main() {
46     test01();
47
48     return 0;
49 }

```

Fence 42

13.4 (4) 插入和删除

- 功能描述：
 - map容器进行插入数据和删除数据
- 函数原型：
 - `insert(elem);` // 在容器中插入元素。
 - `clear();` // 清除所有元素。
 - `erase(pos);` // 删除pos迭代器所指的元素，返回下一个元素的迭代器。
 - `erase(beg, end);` // 删除区间[beg,end)的所有元素，返回下一个元素的迭代器。
 - `erase(key);` // 删除容器中值为key的元素。

示例：

```

1  #include <iostream>
2  #include <map>
3
4  using namespace std;
5
6  /* map的插入和删除 */
7

```



```

8
9 void printMap(const map<int, int>& m) {
10     for (map<int, int>::const_iterator it = m.begin(); it != m.end(); it++) {
11         cout << it->first << " " << it->second << endl;
12     }
13     cout << endl;
14 }
15
16 void test01() {
17     map<int, int> m1;
18
19     //map::insert 方法期待一个 std::pair 类型来插入键值对。
20     m1.insert(pair<int, int>(1, 1)); // 在容器中插入元素。第一种
21     m1.insert(pair<int, int>(3, 1));
22     m1.insert(pair<int, int>(2, 3));
23     m1.insert(pair<int, int>(3, 4));
24     m1.insert(make_pair(5, 5)); //第二种
25     m1.insert({ 4, 5 }); //第三种
26     m1.insert(map<int, int>::value_type(6, 6)); //第四种
27     m1[7] = 8; //第五种 存在风险，使用时如果没有创建键值，会
默认创建一个0；如果有会覆盖原本数据
28     cout << m1[8] << "《-值" << endl; //[]多用于访问map的value值
29     printMap(m1);
30
31     m1.erase(8); // 删除容器中值为key的元素。
32     printMap(m1);
33
34     m1.erase(m1.begin()); // 删除pos迭代器所指的元素，返回下一个元素的
迭代器。
35     printMap(m1);
36
37     map<int, int>::iterator it = m1.begin();
38     m1.erase(m1.begin(), ++it); // 删除区间[beg, end)的所有元素，返回下一个
元素的迭代器。
39     printMap(m1);
40
41     m1.clear(); // 清除所有元素。
42     printMap(m1);
43
44
45
46
47
48 }
49
50 int main() {
51     test01();
52
53     return 0;
54 }

```

13.5 (5) 查找和统计

- 功能描述：
 - 对 map 容器进行查找数据以及统计数据
- 函数原型：
 - `find(key)`; //查找 key 是否存在，若存在，返回该键的元素的迭代器；若不存在，返回 `map.end()`。
 - `count(key)`; //统计 key 的元素个数。map 只为0或者1；multimap会出现大于1的情况。

示例：

```
1  #include <iostream>
2  #include <map>
3
4  using namespace std;
5
6  /* map的查找和统计 */
7
8
9  void printMap(const map<int, int>& m) {
10     for (map<int, int>::const_iterator it = m.begin(); it != m.end(); it++) {
11         cout << it->first << " " << it->second << endl;
12     }
13     cout << endl;
14 }
15
16 void test01() {
17     map<int, int> m1;
18
19     //map::insert 方法期待一个 std::pair 类型来插入键值对。
20     m1.insert(pair<int, int>(1, 1));           // 在容器中插入元素。第一种
21     m1.insert(pair<int, int>(3, 1));
22     m1.insert(pair<int, int>(2, 3));
23     m1.insert(pair<int, int>(3, 4));
24     m1.insert(make_pair(5, 5));               //第二种
25     m1.insert({ 4,5 });                       //第三种
26     m1.insert(map<int, int>::value_type(6, 6)); //第四种
27     m1[7] = 8;                                //第五种 存在风险，使用时如果没有创建键值，会
28     //默认创建一个0；如果有会覆盖原本数据
29     cout << m1[8] << " 《-值" << endl;        //[]多用于访问map的value值
30     printMap(m1);
31
32     int i = 5;
33     if (m1.find(i) != m1.end()) {
34         cout << m1[i] << endl;
35         cout << "数量: " << m1.count(i) << endl;
36     }
37     else {
38         cout << "未找到" << endl;
39         cout << "数量: " << m1.count(i) << endl;
```

```
39     }
40
41
42
43
44
45 }
46
47 int main() {
48     test01();
49
50     return 0;
51 }
```

Fence 44

13.6 (6) 排序

- 主要技术点：
 - 利用仿函数，改变排序规则

自定义数据类型，必须指定排序规则。

示例：

```
1  #include <iostream>
2  #include <map>
3
4  using namespace std;
5
6  /* map的排序 */
7
8  class Person
9  {
10 public:
11     Person(char name,int age) {
12         this->m_name = name;
13         this->m_age = age;
14     }
15
16     char m_name;
17     int m_age;
18 };
19
20
21
22 //仿函数
23 class MyCompareMap
24 {
```

```

25 public:
26     //降序 不能通过value来排序
27     bool operator()(int v1, int v2) const {
28         return v1 > v2;
29     }
30
31     bool operator()(const Person p1, const Person p2) const {
32         return p1.m_age > p2.m_age;
33     }
34
35 };
36
37
38
39 void printMap(const map<int, int, MyCompareMap>& m) {
40     for (map<int, int, MyCompareMap>::const_iterator it = m.begin(); it != m.end();
41         it++) {
42         cout << it->first << " " << it->second << endl;
43     }
44     cout << endl;
45 }
46
47 void printMap(const map<Person, int, MyCompareMap>& m) {
48     for (map<Person, int, MyCompareMap>::const_iterator it = m.begin(); it != m.end();
49         it++) {
50         cout << "姓名: " << it->first.m_name
51             << "年龄: " << it->first.m_age << " " << it->second << endl;
52     }
53     cout << endl;
54 }
55
56 void test01() {
57     map<int, int, MyCompareMap> m1;
58
59     //map::insert 方法期待一个 std::pair 类型来插入键值对。
60     m1.insert(pair<int, int>(1, 1)); // 在容器中插入元素。第一种
61     m1.insert(pair<int, int>(3, 1));
62     m1.insert(pair<int, int>(2, 3));
63     m1.insert(pair<int, int>(3, 4));
64     m1.insert(make_pair(5, 5)); //第二种
65     m1.insert({ 4, 5 }); //第三种
66     m1.insert(map<int, int>::value_type(6, 6)); //第四种
67     m1[7] = 8; //第五种 存在风险, 使用时如果没有创建键值, 会
68     //默认创建一个0; 如果有会覆盖原本数据
69     cout << m1[8] << " 《-值" << endl; //[]多用于访问map的value值
70     printMap(m1);
71
72     map<Person, int, MyCompareMap> m2;
73
74     Person p1('A', 10);
75     Person p2('B', 40);
76     Person p3('C', 20);

```

```
75     m2.insert(make_pair(p1, 100));
76     m2.insert(make_pair(p2, 200));
77     m2.insert(make_pair(p3, 300));
78
79     printMap(m2);
80 }
81
82 int main() {
83     test01();
84
85     return 0;
86 }
```

3 函数对象

14.3.1 函数对象（仿函数）

14.1 (1) 函数对象概念

- 概念：
 - 重载函数调用操作符的类，其对象常称为函数对象
 - 函数对象使用重载的()时，行为类似函数调用，也叫仿函数
- 本质：函数对象(仿函数)是一个类，不是一个函数
- 4种实现方式：函数指针（function pointer）、lambda表达式、“带有成员函数 operator ()”的class建立的object、“带有转换函数可以将自己转换为 pointer to function”的class所建立的object。

14.2 (2) 函数对象使用

- 特点：
 - 函数对象在使用时，可以像普通函数那样调用，可以有参数，可以有返回值
 - 函数对象超出普通函数的概念，函数对象可以有自己的状态
 - 函数对象可以作为参数传递

示例：

```
1  #include <iostream>
2
3
4  using namespace std;
5
6  class MyAdd
7  {
8  public:
9      int operator()(int v1, int v2) {
10         return v1 + v2;
11     }
12 };
13
14 class MyChar
15 {
16 public:
17     MyChar() {
18         this->count = 0;
19     }
```

```

20     void operator()(char a) {
21         cout << a << endl;
22         this->count++;
23     }
24
25     int count;           //记录仿函数使用状态
26 };
27
28
29
30 //函数对象在使用时，可以像普通函数那样调用，可以有参数，可以有返回值
31 void test01() {
32     MyAdd myAdd;           //myAdd叫做函数对象
33     cout << myAdd(10, 10) << endl; //本质上是myAdd()(10,10)
34 }
35
36
37 // 函数对象超出普通函数的概念，函数对象可以有自己的状态
38 void test02() {
39     MyChar myChar;
40     myChar('A');
41     myChar('A');
42
43     cout << myChar.count << endl; //打印状态
44 }
45
46 //函数对象可以作为参数传递
47 void doChar(MyChar &mC, char a) {
48     mC(a);
49 }
50
51 void test03() {
52     MyChar myChar;
53     doChar(myChar, 'B');
54 }
55
56 int main() {
57     test01();
58     test02();
59     test03();
60
61     return 0;
62 }

```

15.3.2 谓词

15.1 (1) 概念

- 概念：
 - 返回 `bool` 类型的仿函数称为谓词。
 - 如果 `operator()` 接受一个参数，那么称作一元谓词。
 - 如果 `operator()` 接受两个参数，那么称作二元谓词。

示例1——一元谓词：

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4
5
6  using namespace std;
7
8  /* 一元谓词 */
9  class MyC{
10 public:
11     bool operator()(int var) {
12         return var > 5;
13     }
14 };
15
16 void test01() {
17     vector<int> v;
18
19     for (int i = 0; i < 10; i++) {
20         v.push_back(i);
21     }
22
23     //寻找大于5的数
24     //MyC() 匿名函数对象
25     //find_if() 按照条件查找元素的算法
26     vector<int>::iterator it = find_if(v.begin(), v.end(), MyC());
27
28     if (it != v.end()) {
29         cout << *it << endl;
30     }
31     else {
32         cout << "未找到" << endl;
33     }
34 }
35
36
37
38
```



```

39  int main() {
40      test01();
41
42      return 0;
43  }

```

Fence 47

示例2——二元谓词：

```

1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4
5
6  using namespace std;
7
8  /* 二元谓词 */
9  class MyC {
10 public:
11     bool operator()(int v1,int v2) {
12         return v1 > v2;
13     }
14 };
15
16 void test01() {
17     vector<int> v;
18
19     for (int i = 0; i < 10; i++) {
20         v.push_back(i);
21     }
22
23     //改变排序规则从大到小
24     sort(v.begin(),v.end(),MyC());
25     for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
26         cout << *it << " ";
27     }
28     cout << endl;
29 }
30
31
32
33
34 int main() {
35     test01();
36
37     return 0;
38 }

```

Fence 48

16.3.3 内建函数对象

16.1 (1) 意义

- 概念：
 - STL内建了一些函数对象。
- 分类：
 - 算术仿函数
 - 关系仿函数
 - 逻辑仿函数
- 用法：
 - 这些仿函数所产生的对象，用法和一般函数完全相同。
 - 使用内建函数对象，需要引入头文件 `#include <functional>`。

16.2 (2) 算术仿函数

- 功能描述：
 - 实现四则运算
 - 其中 `negate` 是一元运算，其他都是二元运算
- 仿函数原型：
 - `template<class T> T plus<T>` // 加法仿函数
 - `template<class T> T minus<T>` // 减法仿函数
 - `template<class T> T multiplies<T>` // 乘法仿函数
 - `template<class T> T divides<T>` // 除法仿函数
 - `template<class T> T modulus<T>` // 取模仿函数
 - `template<class T> T negate<T>` // 取反仿函数

template为声明模板T

T plus第一个为返回值类型，第二个为参数列表类型

示例：

```
1  #include <iostream>
2  #include <vector>
3  #include <functional>
4
5
6  using namespace std;
7
```

```

8  /* 算术仿函数 */
9
10 void test01() {
11     // 取反仿函数
12     negate<int> n;
13
14     cout << n(5) << endl;
15
16     // 加法仿函数 二元仿函数 但是两个参数只能是同一类型
17     plus<int> p;
18     cout << p(5,6) << endl;
19
20     // 减法仿函数
21     minus<int> mi;
22     cout << mi(5, 6) << endl;
23
24     // 乘法仿函数
25     multiplies<int> mu;
26     cout << mu(5, 6) << endl;
27
28     // 除法仿函数
29     divides<int> d;
30     cout << d(10, 5) << endl;
31
32     // 取模仿函数
33     modulus<int> mo;
34     cout << mo(10, 5) << endl;
35 }
36
37
38
39
40 int main() {
41     test01();
42
43     return 0;
44 }

```

Fence 49

16.3 (3) 关系仿函数

- 功能描述：
 - 实现关系对比
- 仿函数原型：
 - `template<class T> bool equal_to<T>` // 等于
 - `template<class T> bool not_equal_to<T>` // 不等于
 - `template<class T> bool greater<T>` // 大于
 - `template<class T> bool greater_equal<T>` // 大于等于

- `template<class T> bool less<T> // 小于`
- `template<class T> bool less_equal<T> // 小于等于`

示例:

```
1  #include <iostream>
2  #include <vector>
3  #include <functional>
4  #include <algorithm>
5
6
7  using namespace std;
8
9  /* 关系仿函数 */
10
11 void test01() {
12     vector<int> v;
13
14     v.push_back(10);
15     v.push_back(30);
16     v.push_back(40);
17     v.push_back(20);
18
19     sort(v.begin(), v.end(), greater<int>());
20
21     // 大于
22     for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
23         cout << *it << " ";
24     }
25     cout << endl;
26 }
27
28
29
30
31 int main() {
32     test01();
33
34     return 0;
35 }
```

Fence 50

16.4 (4) 逻辑仿函数

- 功能描述:
 - 实现逻辑运算
- 函数原型:

- `template<class T> bool logical_and<T> //逻辑与`
- `template<class T> bool logical_or<T> //逻辑或`
- `template<class T> bool logical_not<T> //逻辑非`

示例:

```
1  #include <iostream>
2  #include <vector>
3  #include <functional>
4  #include <algorithm>
5
6
7  using namespace std;
8
9  /* 逻辑仿函数 */
10
11 void test01() {
12     vector<bool> v;
13
14     v.push_back(true);
15     v.push_back(false);
16     v.push_back(true);
17     v.push_back(true);
18
19
20     for (vector<bool>::iterator it = v.begin(); it != v.end(); it++) {
21         cout << *it << " ";
22     }
23     cout << endl;
24
25     //利用逻辑非 将容器v搬运到容器v1中, 并且取反
26     vector<bool> v1;
27     v1.resize(v.size());
28
29     //搬运算法
30     transform(v.begin(), v.end(), v1.begin(), logical_not<bool>());
31
32     for (vector<bool>::iterator it = v1.begin(); it != v1.end(); it++) {
33         cout << *it << " ";
34     }
35     cout << endl;
36 }
37
38
39
40
41 int main() {
42     test01();
43
44     return 0;
```


4 算法

- 概述：
 - 算法主要是由头文件 `<algorithm>`、`<functional>`、`<numeric>` 组成。
 - `<algorithm>` 是所有STL头文件中最大的一个，范围涉及到比较、交换、查找、遍历操作、复制、修改等等。
 - `<numeric>` 体积很小，只包括几个在序列上面进行简单数学运算的模板函数。
 - `<functional>` 定义了一些模板类，用以声明函数对象。

17. 4.1 遍历算法

- 算法简介：
 - `for_each` //遍历容器
 - `transform` //搬运容器到另一个容器中

17.1 (1) for_each()

- 功能描述：
 - 实现遍历容器
- 函数原型：
 - `for_each(iterator beg, iterator end, _func);`
 - 遍历算法 遍历容器元素
 - `beg` 开始迭代器
 - `end` 结束迭代器
 - `_func` 函数或者函数对象

示例：

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4
5
6  using namespace std;
7
8  /* for_each */
9
10 // 普通函数
11 void print01(int v) {
12     cout << v << " ";
13 }
14
```

```

15 //仿函数
16 class Print02
17 {
18 public:
19     void operator()(int v) {
20         cout << v << " ";
21     }
22
23 };
24
25
26
27 void test01() {
28     vector<int> v;
29     v.push_back(10);
30     v.push_back(20);
31     v.push_back(30);
32     v.push_back(40);
33     v.push_back(50);
34     v.push_back(60);
35
36     for_each(v.begin(),v.end(), print01);
37     cout << endl;
38
39     for_each(v.begin(), v.end(), Print02());
40     cout << endl;
41
42 }
43
44
45 int main() {
46     test01();
47
48     return 0;
49 }

```

Fence 52

17.2 (2) transform()

- 功能描述：
 - 搬运容器到另一个容器中
- 函数原型：
 - `transform(iterator beg1, iterator end1, iterator beg2, _func);`
 - `beg1`：源容器开始迭代器
 - `end1`：源容器结束迭代器
 - `beg2`：目标容器开始迭代器
 - `_func`：函数或者函数对象

- 目标容器必须提前开辟空间，否则会报错

示例:

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4
5
6  using namespace std;
7
8  /* transform */
9
10 //仿函数
11 class Transform
12 {
13 public:
14     int operator()(int v) {
15         cout << v << " ";
16         return v;
17     }
18
19 };
20
21 class Print
22 {
23 public:
24     void operator()(int v) {
25         cout << v << " ";
26     }
27
28 };
29
30 void test01() {
31     vector<int> v;
32     v.push_back(10);
33     v.push_back(20);
34     v.push_back(30);
35     v.push_back(40);
36     v.push_back(50);
37     v.push_back(60);
38
39     vector<int> v1;
40     v1.resize(6); //目标容器需要提前开辟空间，否则会报错
41
42
43     transform(v.begin(), v.end(), v1.begin(), Transform()); //可在搬运时，进行数据处理
44     cout << endl;
45     for_each(v1.begin(), v1.end(), Print());
46
47 }
```

```
48
49
50 int main() {
51     test01();
52
53     return 0;
54 }
```

Fence 53

18.4.2 查找算法

- 算法简介：
 - `find` // 查找元素
 - `find_if` // 按条件查找元素
 - `adjacent_find` // 查找相邻重复元素
 - `binary_search` // 二分查找法
 - `count` // 统计元素个数
 - `count_if` // 按条件统计元素个数

18.1 (1) `find()`

- 功能描述：
 - 查找指定元素，找到返回指定元素的迭代器，找不到返回结束迭代器`end()`
- 函数原型：
 - `find(iterator beg, iterator end, value);`
 - // 按值查找元素，找到返回指定位置迭代器，找不到返回结束迭代器位置
 - // `beg` 开始迭代器
 - // `end` 结束迭代器
 - // `value` 查找的元素
- 查找自定义类型，需要重载`==`
- 返回值为迭代器

示例：

```
1 #include <iostream>
2 #include <vector>
3 #include <algorithm>
4
5
6 using namespace std;
7
```

```

8  /* find */
9
10 class Person{
11 public:
12     Person(char name,int age) {
13         this->m_name = name;
14         this->m_age = age;
15     }
16
17     //重载== 让find底层知道如何对比自定义数据类型
18     bool operator==(const Person &p1) {
19         return this->m_name == p1.m_name;
20     }
21
22     char m_name;
23     int m_age;
24 };
25
26 //仿函数
27 class Transform
28 {
29 public:
30     int operator()(int v) {
31         cout << v << " ";
32         return v;
33     }
34
35 };
36
37 class Print
38 {
39 public:
40     void operator()(int v) {
41         cout << v << " ";
42     }
43
44 };
45
46 void test01() {
47     //查找内置数据类型
48     vector<int> v;
49     v.push_back(10);
50     v.push_back(20);
51     v.push_back(30);
52     v.push_back(40);
53     v.push_back(50);
54     v.push_back(60);
55
56     vector<int>::iterator it;
57     it = find(v.begin(),v.end(),5);
58     if (it != v.end()) {
59         cout << *it << endl;
60     }

```

```

61     else {
62         cout << "没有" << endl;
63     }
64
65     //查找自定义数据类型
66     vector<Person> v1;
67     char c[4] = {'A','B','C','D'};
68     for (int i = 0; i < 4; i++) {
69         int t = rand() % 100 + 1;
70         Person p(c[i],t);
71
72         v1.push_back(p);
73     }
74
75     vector<Person>::iterator itP;
76     Person q('A',20);
77     itP = find(v1.begin(), v1.end(), q);
78     if (itP != v1.end()) {
79         cout << itP->m_name << " " << itP->m_age << endl;
80     }
81     else {
82         cout << "没有" << endl;
83     }
84
85 }
86
87
88 int main() {
89     test01();
90
91     return 0;
92 }

```

Fence 54

18.2 (2) find_if()

- 功能描述：
 - 按条件查找元素
- 函数原型：
 - find_if(iterator beg, iterator end, _Pred);
 - // 按值查找元素，找到返回指定位置迭代器，找不到返回结束迭代器位置
 - // beg 开始迭代器
 - // end 结束迭代器
 - // _Pred 函数或谓词（返回 bool 类型的仿函数）

示例：

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4
5
6  using namespace std;
7
8  /* find_if */
9
10 class Person {
11 public:
12     Person(char name, int age) {
13         this->m_name = name;
14         this->m_age = age;
15     }
16
17     //重载== 让find底层知道如何对比自定义数据类型
18     bool operator==(const Person& p1) {
19         return this->m_name == p1.m_name;
20     }
21
22     char m_name;
23     int m_age;
24 };
25
26 //仿函数
27 class GreaterFive
28 {
29 public:
30     bool operator()(int v) {
31         return v > 20;
32     }
33
34
35 };
36
37 class Print
38 {
39 public:
40     bool operator()(Person& p1) {
41         return p1.m_name == 'A';
42     }
43
44 };
45
46 void test01() {
47     //查找内置数据类型
48     vector<int> v;
49     v.push_back(10);
50     v.push_back(20);
51     v.push_back(30);
52     v.push_back(40);
53     v.push_back(50);
```

```

54     v.push_back(60);
55
56     vector<int>::iterator it;
57     it = find_if(v.begin(), v.end(), GreaterFive());
58     if (it != v.end()) {
59         cout << *it << endl;
60     }
61     else {
62         cout << "没有" << endl;
63     }
64
65     //查找自定义数据类型
66     vector<Person> v1;
67     char c[4] = { 'A', 'B', 'C', 'D' };
68     for (int i = 0; i < 4; i++) {
69         int t = rand() % 100 + 1;
70         Person p(c[i], t);
71
72         v1.push_back(p);
73     }
74
75     vector<Person>::iterator itP;
76     Person q('A', 20);
77     itP = find_if(v1.begin(), v1.end(), Print());
78     if (itP != v1.end()) {
79         cout << itP->m_name << " " << itP->m_age << endl;
80     }
81     else {
82         cout << "没有" << endl;
83     }
84
85 }
86
87
88 int main() {
89     test01();
90
91     return 0;
92 }

```

Fence 55

18.3 (3) adjacent_find()

- 功能描述：
 - 查找相邻重复元素
- 函数原型：
 - `adjacent_find(iterator beg, iterator end);`
 - // 查找相邻重复元素, 返回相邻元素的第一个位置的迭代器
 - // beg 开始迭代器

- // end 结束迭代器

示例:

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4
5
6  using namespace std;
7
8  /* adjacent_find */
9
10 class Person {
11 public:
12     Person(char name, int age) {
13         this->m_name = name;
14         this->m_age = age;
15     }
16
17     //重载== 让find底层知道如何对比自定义数据类型
18     bool operator==(const Person& p1) {
19         return this->m_name == p1.m_name;
20     }
21
22     char m_name;
23     int m_age;
24 };
25
26 //仿函数
27 class GreaterFive
28 {
29 public:
30     bool operator()(int v) {
31         return v > 20;
32     }
33
34
35 };
36
37 class Print
38 {
39 public:
40     bool operator()(Person& p1) {
41         return p1.m_name == 'A';
42     }
43
44 };
45
46 void test01() {
47     //查找内置数据类型
```

```

48     vector<int> v;
49     v.push_back(10);
50     v.push_back(20);
51     v.push_back(20);
52     v.push_back(40);
53     v.push_back(50);
54     v.push_back(60);
55
56     vector<int>::iterator it;
57     it = adjacent_find(v.begin(),v.end());
58     if (it != v.end()) {
59         cout << *it << endl;
60     }
61     else {
62         cout << "没有" << endl;
63     }
64
65     //查找自定义数据类型
66     vector<Person> v1;
67     char c[4] = { 'A','B','C','D' };
68     for (int i = 0; i < 4; i++) {
69         Person p(c[i], 1);
70
71         v1.push_back(p);
72     }
73
74     vector<Person>::iterator itP;
75     Person q('A', 20);
76     itP = adjacent_find(v1.begin(), v1.end());
77     if (itP != v1.end()) {
78         cout << itP->m_name << " " << itP->m_age << endl;
79     }
80     else {
81         cout << "没有" << endl;
82     }
83
84 }
85
86
87 int main() {
88     test01();
89
90     return 0;
91 }

```


18.4 (4) binary_search()

- 功能描述：
 - 查找指定元素是否存在
- 函数原型：
 - `bool binary_search(iterator beg, iterator end, value);`
 - // 查找指定的元素，查到返回true 否则false
 - // 注意：在**无序序列中不可用**(必须为排好序的序列,且必须为升序，降序也找不到)
 - // beg 开始迭代器
 - // end 结束迭代器
 - // value 查找的元素（实际值）

示例：

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4
5
6  using namespace std;
7
8  /* binary_search */
9
10
11
12 void test01() {
13     //查找内置数据类型
14     vector<int> v;
15     v.push_back(60);
16     v.push_back(50);
17     v.push_back(40);
18     v.push_back(30);
19     v.push_back(20);
20     v.push_back(10);
21
22     bool it;
23     it = binary_search(v.begin(), v.end(), 10);           //降序无法寻找
24     if (it) {
25         cout << "有" << endl;
26     }
27     else {
28         cout << "没有" << endl;
29     }
30
31
32
33 }
34
```

```

35
36     int main() {
37         test01();
38
39         return 0;
40     }

```

Fence 57

18.5 (5) count()

- 功能描述：
 - 统计元素个数
- 函数原型：
 - count(iterator beg, iterator end, value);
 - // 统计元素出现次数
 - // beg 开始迭代器
 - // end 结束迭代器
 - // value 统计的元素
- 自定义数据类型，需要重载==

示例：

```

1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4
5
6  using namespace std;
7
8  /* count */
9  class Person
10 {
11 public:
12     Person(char name,int age) {
13         this->m_name = name;
14         this->m_age = age;
15     }
16
17     //需要重载==
18     bool operator==(const Person &p) {
19         return ((this->m_name == p.m_name) && (this->m_age == p.m_age));
20     }
21
22     char m_name;
23     int m_age;
24 };

```

```

25
26
27 void test01() {
28     //统计内置数据类型
29     vector<int> v;
30     v.push_back(60);
31     v.push_back(50);
32     v.push_back(20);
33     v.push_back(20);
34     v.push_back(20);
35     v.push_back(10);
36
37     int it;
38     it = count(v.begin(), v.end(), 20);
39     cout << it << endl;
40
41     //统计自定义类型
42     vector<Person> v1;
43     Person p1('A', 20);
44     Person p2('A', 20);
45     Person p3('B', 20);
46     Person p4('A', 30);
47
48     v1.push_back(p1);
49     v1.push_back(p2);
50     v1.push_back(p3);
51     v1.push_back(p4);
52
53     Person p5('A', 20);
54
55     int num = count(v1.begin(), v1.end(), p5);
56     cout << num << endl;
57 }
58
59
60 int main() {
61     test01();
62
63     return 0;
64 }

```

Fence 58

18.6 (6) count_if()

- 功能描述：
 - 按条件统计元素个数
- 函数原型：
 - `count_if(iterator beg, iterator end, _Pred);`
 - 按条件统计元素出现次数

- `beg` 开始迭代器
- `end` 结束迭代器
- `_Pred` 谓词

示例:

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4
5
6  using namespace std;
7
8  /* count_if */
9  class Person
10 {
11 public:
12     Person(char name, int age) {
13         this->m_name = name;
14         this->m_age = age;
15     }
16
17     //需要重载==
18     bool operator==(const Person& p) {
19         return ((this->m_name == p.m_name) && (this->m_age == p.m_age));
20     }
21
22     char m_name;
23     int m_age;
24 };
25
26 class Pr
27 {
28 public:
29     bool operator()(const int& v) {
30         return v > 20;
31     }
32 };
33
34 class Pr1
35 {
36 public:
37     bool operator()(const Person& v) {
38         return v.m_age > 20;
39     }
40 };
41
42
43 void test01() {
44     //统计内置数据类型
```

```

45     vector<int> v;
46     v.push_back(60);
47     v.push_back(50);
48     v.push_back(20);
49     v.push_back(20);
50     v.push_back(20);
51     v.push_back(10);
52
53     int it;
54     it = count_if(v.begin(), v.end(), Pr());
55     cout << it << endl;
56
57     //统计自定义类型
58     vector<Person> v1;
59     Person p1('A', 20);
60     Person p2('A', 20);
61     Person p3('B', 20);
62     Person p4('A', 30);
63
64     v1.push_back(p1);
65     v1.push_back(p2);
66     v1.push_back(p3);
67     v1.push_back(p4);
68
69
70     int num = count_if(v1.begin(), v1.end(), Pr1());
71     cout << num << endl;
72 }
73
74
75 int main() {
76     test01();
77
78     return 0;
79 }

```

Fence 59

19.4.3 排序算法

- 算法简介:

- `sort` // 对容器内元素进行排序
- `random_shuffle` // 洗牌, 指定范围内的元素随机调整次序
- `merge` // 容器元素合并, 并存储到另一容器中
- `reverse` // 反转指定范围的元素

19.1 (1) sort()

- 功能描述：
 - 对容器内元素进行排序
- 函数原型：
 - `sort(iterator beg, iterator end, _Pred);`
 - // 按值查找元素，找到返回指定位置迭代器，找不到返回结束迭代器位置
 - // `beg` 开始迭代器
 - // `end` 结束迭代器
 - // `_Pred` 谓词（可选，默认从小到大的排序）

示例：

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  #include <functional>
5
6  using namespace std;
7
8  /* sort */
9
10 void Print(int v) {
11     cout << v << " ";
12 }
13
14 void test01() {
15     //内置数据类型
16     vector<int> v;
17     v.push_back(60);
18     v.push_back(50);
19     v.push_back(20);
20     v.push_back(20);
21     v.push_back(20);
22     v.push_back(10);
23
24     sort(v.begin(),v.end());
25     for_each(v.begin(),v.end(), Print);
26     cout << endl;
27
28     //该降序
29     sort(v.begin(), v.end(), greater<int>());
30     for_each(v.begin(), v.end(), Print);
31     cout << endl;
32
33 }
34
35
```

```
36 int main() {
37     test01();
38
39     return 0;
40 }
```

Fence 60

19.2 (2) random_shuffle()

- 功能描述：
 - 洗牌：指定范围内的元素随机调整次序
- 函数原型：
 - `random_shuffle(iterator beg, iterator end);`
 - 指定范围内的元素随机调整次序
 - `beg` 开始迭代器
 - `end` 结束迭代器
- 需要加随机数种子

示例：

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  #include <functional>
5
6  using namespace std;
7
8  /* random_shuffle */
9
10
11 void Print(int v) {
12     cout << v << " ";
13 }
14
15 void test01() {
16
17     vector<int> v;
18     v.push_back(60);
19     v.push_back(50);
20     v.push_back(20);
21     v.push_back(20);
22     v.push_back(20);
23     v.push_back(10);
```

```

24
25     for_each(v.begin(), v.end(), Print);
26     cout << endl;
27
28     //洗牌
29     random_shuffle(v.begin(),v.end());
30     for_each(v.begin(), v.end(), Print);
31     cout << endl;
32
33 }
34
35
36 int main() {
37     srand((unsigned int)time(NULL));           //需要添加随机数种子 这样random_shuffle才
能每次打乱结果不一样
38
39     test01();
40
41     return 0;
42 }

```

Fence 61

19.3 (3) merge()

- 功能描述：
 - 两个容器元素合并，并存储到另一个容器中(合并完，依然为有序序列)
- 函数原型：
 - `merge(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest);`
 - 容器元素合并，并存储到另一个容器中
 - **注意：两个容器必须是有序的**(必须为升序)
 - `beg1` 容器1开始迭代器
 - `end1` 容器1结束迭代器
 - `beg2` 容器2开始迭代器
 - `end2` 容器2结束迭代器
 - `dest` 目标容器开始迭代器
- 提前给容器开辟空间

示例：

```

1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  #include <functional>
5

```



```

6   using namespace std;
7
8   /* merge */
9
10
11  void Print(int v) {
12      cout << v << " ";
13  }
14
15  void test01() {
16
17      vector<int> v,v1,v2;
18      v.push_back(60);
19      v.push_back(50);
20      v.push_back(20);
21      v.push_back(20);
22      v.push_back(20);
23      v.push_back(10);
24
25      v1.push_back(6);
26      v1.push_back(5);
27      v1.push_back(2);
28      v1.push_back(2);
29      v1.push_back(2);
30      v1.push_back(1);
31
32      // 对v和v1进行排序 merge必须为升序
33      sort(v.begin(), v.end());
34      sort(v1.begin(), v1.end());
35
36      int size1 = v.size() + v1.size();
37      v2.resize(size1);           //提前开辟空间
38
39      for_each(v.begin(), v.end(), Print);
40      cout << endl;
41      for_each(v1.begin(), v1.end(), Print);
42      cout << endl;
43
44      merge(v.begin(),v.end(),v1.begin(),v1.end(),v2.begin());
45      for_each(v2.begin(), v2.end(), Print);
46      cout << endl;
47
48  }
49
50
51  int main() {
52      test01();
53
54      return 0;
55  }

```

19.4 (4) reverse()

- 功能描述：
 - 将容器内元素进行反转
- 函数原型：
 - `reverse(iterator beg, iterator end);`
 - `// 反转指定范围的元素`
 - `// beg 开始迭代器`
 - `// end 结束迭代器`

示例:

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  #include <functional>
5
6  using namespace std;
7
8  /* reverse */
9
10
11 void Print(int v) {
12     cout << v << " ";
13 }
14
15 void test01() {
16
17     vector<int> v, v1, v2;
18     v.push_back(60);
19     v.push_back(50);
20     v.push_back(20);
21     v.push_back(20);
22     v.push_back(20);
23     v.push_back(10);
24
25
26     for_each(v.begin(), v.end(), Print);
27     cout << endl << "反转后" << endl;
28     reverse(v.begin(), v.end());
29     for_each(v.begin(), v.end(), Print);
30     cout << endl;
31 }
32
33
34 int main() {
35     test01();
36 }
```

```
37     return 0;
38 }
```

Fence 63

20.4.4 拷贝和替换算法

- 算法简介：
 - `copy`：将容器内指定范围的元素拷贝到另一容器中
 - `replace`：将容器内指定范围的旧元素修改为新元素
 - `replace_if`：容器内指定范围满足条件的元素替换为新元素
 - `swap`：互换两个容器的元素

20.1 (1) `copy()`

- 功能描述：
 - 容器内指定范围的元素拷贝到另一容器中
- 函数原型：
 - `copy(iterator beg, iterator end, iterator dest);`
 - 说明：
 - 按值查找元素，找到返回指定位置迭代器，找不到返回结束迭代器位置
 - `// beg` 开始迭代器
 - `// end` 结束迭代器
 - `// dest` 目标迭代器开始迭代器
- 必须先开辟空间

示例：

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  #include <functional>
5
6  using namespace std;
7
8  /* copy */
9
10
11 void Print(int v) {
12     cout << v << " ";
13 }
14
15 void test01() {
```

```

16
17     vector<int> v, v1;
18     v.push_back(60);
19     v.push_back(50);
20     v.push_back(20);
21     v.push_back(20);
22     v.push_back(20);
23     v.push_back(10);
24
25     //开辟空间
26     v1.resize(v.size());
27
28
29     for_each(v.begin(), v.end(), Print);
30     cout << endl << "复制后" << endl;
31     copy(v.begin(), v.end(), v1.begin());
32     for_each(v1.begin(), v1.end(), Print);
33     cout << endl;
34 }
35
36
37 int main() {
38     test01();
39
40     return 0;
41 }

```

Fence 64

20.2 (2) replace()

- 功能描述：
 - 将容器内指定范围的旧元素修改为新元素。
- 函数原型：
 - `replace(iterator beg, iterator end, oldvalue, newvalue);`
 - 将区间内旧元素替换成新元素
 - `beg`：开始迭代器
 - `end`：结束迭代器
 - `oldvalue`：旧元素
 - `newvalue`：新元素

示例：

```

1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  #include <functional>

```

```

5
6 using namespace std;
7
8 /* replace */
9
10
11 void Print(int v) {
12     cout << v << " ";
13 }
14
15 void test01() {
16
17     vector<int> v, v1;
18     v.push_back(60);
19     v.push_back(50);
20     v.push_back(20);
21     v.push_back(20);
22     v.push_back(20);
23     v.push_back(10);
24
25     for_each(v.begin(), v.end(), Print);
26     cout << endl << " 替换后" << endl;
27
28     replace(v.begin(), v.end(), 20, 200);
29     for_each(v.begin(), v.end(), Print);
30 }
31
32
33 int main() {
34     test01();
35
36     return 0;
37 }

```

Fence 65

20.3 (3) replace_if()

- 功能描述：
 - 将区间内满足条件的元素，替换成指定元素。
- 函数原型：
 - `replace_if(iterator beg, iterator end, _pred, newvalue);`
 - 按条件替换元素，满足条件的替换成指定元素
 - `beg`：开始迭代器
 - `end`：结束迭代器
 - `_pred`：谓词
 - `newvalue`：替换的新元素

示例:

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  #include <functional>
5
6  using namespace std;
7
8  /* replace_if */
9
10
11 void Print(int v) {
12     cout << v << " ";
13 }
14
15 class MyPr
16 {
17 public:
18     bool operator()(int v) {
19         return v > 20;
20     }
21 };
22
23
24
25 void test01() {
26
27     vector<int> v, v1;
28     v.push_back(60);
29     v.push_back(50);
30     v.push_back(20);
31     v.push_back(20);
32     v.push_back(20);
33     v.push_back(10);
34
35     for_each(v.begin(), v.end(), Print);
36     cout << endl << " 替换后" << endl;
37
38     replace_if(v.begin(), v.end(), MyPr(), 200);
39     for_each(v.begin(), v.end(), Print);
40 }
41
42
43 int main() {
44     test01();
45
46     return 0;
47 }
```

20.4 (4) swap()

- 功能描述：
 - 互换两个同种类型容器的元素
- 函数原型：
 - `swap(container c1, container c2);`
 - 互换两个容器的元素 同种类型的容器的交换
 - `c1`: 容器1
 - `c2`: 容器2

示例:

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  #include <functional>
5
6  using namespace std;
7
8  /* swap */
9
10
11 void Print(int v) {
12     cout << v << " ";
13 }
14
15 class MyPr
16 {
17 public:
18     bool operator()(int v) {
19         return v > 20;
20     }
21 };
22
23
24
25 void test01() {
26
27     vector<int> v, v1;
28     v.push_back(60);
29     v.push_back(50);
30     v.push_back(20);
31     v.push_back(20);
32     v.push_back(20);
33     v.push_back(10);
34
35     v1.push_back(6);
36     v1.push_back(5);
```

```

37     v1.push_back(2);
38     v1.push_back(2);
39     v1.push_back(2);
40
41     for_each(v.begin(), v.end(), Print);
42     cout << endl;
43     for_each(v1.begin(), v1.end(), Print);
44     cout << endl << " 交换后" << endl;
45
46     swap(v,v1);
47     for_each(v.begin(), v.end(), Print);
48     cout << endl;
49     for_each(v1.begin(), v1.end(), Print);
50 }
51
52
53 int main() {
54     test01();
55
56     return 0;
57 }

```

Fence 67

21.4.5 算术生成算法

- 注意：
 - 算术生成算法属于小型算法，使用时包含的头文件为 `#include <numeric>`
- 算法简介：
 - `accumulate` // 计算容器元素累计总和
 - `fill` // 向容器中添加元素

21.1 (1) accumulate()

- 功能描述：
 - 计算区间内容容器元素累计总和
- 函数原型：
 - `accumulate(iterator beg, iterator end, value);`
 - 计算容器元素累计总和
 - `beg`：开始迭代器
 - `end`：结束迭代器
 - `value`：起始的累加值，该值不在容器中

示例：


```

1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  #include <numeric>
5
6  using namespace std;
7
8  /* accumulate */
9
10
11 void Print(int v) {
12     cout << v << " ";
13 }
14
15 class MyPr
16 {
17 public:
18     bool operator()(int v) {
19         return v > 20;
20     }
21 };
22
23
24
25 void test01() {
26
27     vector<int> v, v1;
28     v.push_back(60);
29     v.push_back(50);
30     v.push_back(20);
31     v.push_back(20);
32     v.push_back(20);
33     v.push_back(10);
34
35
36     for_each(v.begin(), v.end(), Print);
37     cout << endl;
38
39
40     int sum = accumulate(v.begin(), v.end(), 0); //如果不为0, sum的值是在原容器中的
    总和加上该值
41     cout << "总和: " << sum << endl;
42 }
43
44
45 int main() {
46     test01();
47
48     return 0;
49 }

```

21.2 (2) fill()

- 功能描述：
 - 向容器中填充指定的元素(会覆盖原有的值)
- 函数原型：
 - `fill(iterator beg, iterator end, value);`
 - 解释：
 - `beg`: 开始迭代器
 - `end`: 结束迭代器
 - `value`: 填充的值

示例:

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  #include <numeric>
5
6  using namespace std;
7
8  /* fill */
9
10
11 void Print(int v) {
12     cout << v << " ";
13 }
14
15 class MyPr
16 {
17 public:
18     bool operator()(int v) {
19         return v > 20;
20     }
21 };
22
23
24
25 void test01() {
26
27     vector<int> v, v1;
28     v.push_back(60);
29     v.push_back(50);
30     v.push_back(20);
31     v.push_back(20);
32     v.push_back(20);
33     v.push_back(10);
34
35 }
```

```

36     for_each(v.begin(), v.end(), Print);
37     cout << endl;
38
39     fill(v.begin(), v.end(), 100);
40
41     for_each(v.begin(), v.end(), Print);
42     cout << endl;
43
44     v.resize(10);
45     fill(v.begin(), v.end(), 100);
46     for_each(v.begin(), v.end(), Print);
47     cout << endl;
48 }
49
50
51 int main() {
52     test01();
53
54     return 0;
55 }

```

Fence 69

22. 4.6 集合算法

- 算法简介：
 - `set_intersection` // 求两个容器的交集
 - `set_union` // 求两个容器的并集
 - `set_difference` // 求两个容器的差集

22.1 (1) set_intersection()

- 功能描述：
 - 求两个容器的交集
- 函数原型：
 - `set_intersection(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest);`
 - // 求两个集合的交集
 - // 注意： **两个集合必须是有序序列**
 - // beg1 容器1开始迭代器
 - // end1 容器1结束迭代器
 - // beg2 容器2开始迭代器
 - // end2 容器2结束迭代器
 - // dest 目标容器开始迭代器

- 目标容器需要提取开辟空间
- 遍历时，使用 `set_intersection` 返回的迭代器，否则会将没有使用的地方遍历出来。

示例:

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  #include <numeric>
5
6  using namespace std;
7
8  /* set_intersection */
9
10
11 void Print(int v) {
12     cout << v << " ";
13 }
14
15 class MyPr
16 {
17 public:
18     bool operator()(int v) {
19         return v > 20;
20     }
21 };
22
23
24
25 void test01() {
26
27     vector<int> v, v1,v2;
28     v.push_back(1);
29     v.push_back(2);
30     v.push_back(3);
31     v.push_back(4);
32     v.push_back(5);
33     v.push_back(6);
34
35     v1.push_back(2);
36     v1.push_back(12);
37     v1.push_back(22);
38     v1.push_back(32);
39     v1.push_back(42);
40     v1.push_back(52);
41
42     //开辟空间 空间大小取值，最小容器的大小
43
44     v2.resize(min(v1.size(),v.size()));
45
46 }
```

```

47     for_each(v.begin(), v.end(), Print);
48     cout << endl;
49     for_each(v1.begin(), v1.end(), Print);
50     cout << endl;
51
52     //记录结束位置
53     vector<int>::iterator itEnd =
set_intersection(v.begin(),v.end(),v1.begin(),v1.end(),v2.begin());
54
55     //使用itEnd作为结束迭代器
56     for_each(v2.begin(), itEnd, Print);
57     cout << endl;
58 }
59
60
61 int main() {
62     test01();
63
64     return 0;
65 }

```

Fence 70

22.2 (2) set_union()

功能描述:

- 求两个集合的并集。

函数原型:

- `set_union(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest);`
 - // 求两个集合的并集
 - // 注意: 两个集合必须是**有序序列**
 - // beg1 容器1开始迭代器
 - // end1 容器1结束迭代器
 - // beg2 容器2开始迭代器
 - // end2 容器2结束迭代器
 - // dest 目标容器开始迭代器
- 目标容器需要提取开辟空间
- 遍历时, 使用 `set_union` 返回的迭代器, 否则会将没有使用的地方遍历出来。

示例:

```

1  #include <iostream>

```

```

2  #include <vector>
3  #include <algorithm>
4  #include <numeric>
5
6  using namespace std;
7
8  /* set_union */
9
10
11 void Print(int v) {
12     cout << v << " ";
13 }
14
15 class MyPr
16 {
17 public:
18     bool operator()(int v) {
19         return v > 20;
20     }
21 };
22
23
24
25 void test01() {
26
27     vector<int> v, v1, v2;
28     v.push_back(1);
29     v.push_back(2);
30     v.push_back(3);
31     v.push_back(4);
32     v.push_back(5);
33     v.push_back(6);
34
35     v1.push_back(2);
36     v1.push_back(12);
37     v1.push_back(22);
38     v1.push_back(32);
39     v1.push_back(42);
40     v1.push_back(52);
41
42     //开辟空间 空间大小取值，两个容器大小的和
43
44     v2.resize(v1.size() + v.size());
45
46
47     for_each(v.begin(), v.end(), Print);
48     cout << endl;
49     for_each(v1.begin(), v1.end(), Print);
50     cout << endl;
51
52     //记录结束位置
53     vector<int>::iterator itEnd = set_union(v.begin(), v.end(), v1.begin(), v1.end(),
v2.begin());

```

```

54
55     //使用itEnd作为结束迭代器
56     for_each(v2.begin(), itEnd, Print);
57     cout << endl;
58 }
59
60
61 int main() {
62     test01();
63
64     return 0;
65 }

```

Fence 71

22.3 (3) set_difference()

- 功能描述：
 - 求两个集合的差集
- 函数原型：
 - `set_difference(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest);`
 - // 求两个集合的差集(上面为v1对于v2的差集, 差集 = $v1 - (v2 \text{ 与 } v1 \text{ 求交集})$)
 - // 注意:两个集合必须是有序列
 - // beg1 容器1开始迭代器
 - // end1 容器1结束迭代器
 - // beg2 容器2开始迭代器
 - // end2 容器2结束迭代器
 - // dest 目标容器开始迭代器

示例:

```

1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  #include <numeric>
5
6  using namespace std;
7
8  /* set_difference */
9
10
11 void Print(int v) {
12     cout << v << " ";

```

```

13     }
14
15     class MyPr
16     {
17     public:
18         bool operator()(int v) {
19             return v > 20;
20         }
21     };
22
23
24
25     void test01() {
26
27         vector<int> v, v1, v2, v3;
28         v.push_back(1);
29         v.push_back(2);
30         v.push_back(3);
31         v.push_back(4);
32         v.push_back(5);
33         v.push_back(6);
34
35         v1.push_back(2);
36         v1.push_back(12);
37         v1.push_back(22);
38         v1.push_back(32);
39         v1.push_back(42);
40
41
42         //开辟空间 空间大小取值, 求谁的差集, 用谁的容器大小 或者 直接用大的, 如: A和B的差集 A的大小
43
44         //v和v1的差集 容器大小
45         v2.resize(v.size());
46
47         //v1和v的差集 容器大小
48         v3.resize(v1.size());
49
50         //v2.resize(max(v.size(),v1.size()));
51
52
53
54         for_each(v.begin(), v.end(), Print);
55         cout << endl;
56         for_each(v1.begin(), v1.end(), Print);
57         cout << endl;
58
59         //记录结束位置
60         //v和v1的差集
61         vector<int>::iterator itEnd1 = set_difference(v.begin(), v.end(), v1.begin(),
62             v1.end(), v2.begin());
63
64         //v1和v的差集

```



```

64     vector<int>::iterator itEnd2 = set_difference(v1.begin(), v1.end(), v.begin(),
65     v.end(), v3.begin());
66     //使用itEnd作为结束迭代器
67     for_each(v2.begin(), itEnd1, Print);
68     cout << endl;
69     for_each(v3.begin(), itEnd2, Print);
70     cout << endl;
71 }
72
73
74 int main() {
75     test01();
76
77     return 0;
78 }

```

Fence 72

写程序：理清关系-》搭建最基础-》抽象出各个类的父类-》搭建子类接口 -》编写接口

选中+atl + 回车 直接在cpp中创建

/stirng 转 int

//c_str() 转 char* (C语言风格)

//atoi() 转 int

atoi(o.m_order[i]["stuId"].c_str())