

# STL三大组件

## 一. 容器

- 保存数据的数据结构
- 常用的数据结构：数组(array)、链表(list)、tree(树)、栈(stack)、队列(queue)、集合(set)、映射表(map)
- 根据数据在容器中的排列特性,这些数据分为 序列式容器 和 关联式容器
  - 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,除非用删除或插入的操作改变这个位置.
    - Vector容器、Deque容器、List容器等
  - 关联式容器是非线性的树结构,更准确的说是二叉树结构.各元素之间没有严格的物理上的顺序关系,也就是说元素在容器中并没有保存元素置入容器时的逻辑顺序
    - 关联式容器另一个显著的特点是 : 在值中选择一个值作为关键字key,这个关键字对值起到索引的所用,方便查找
    - Set/MultiSet容器、Map/MultiMap容器

## 二. 算法

- 特定的算法往往搭配特定的数据结构,算法与数据结构相辅相成
- 算法分为 : 质变算法 和 非质变算法
  - 质变算法 : 是指运算过程中更改区间内的元素的内容.例如 拷贝、替换、删除 等待
  - 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
- 算法需要包含的头文件 algorithm

## 三. 迭代器

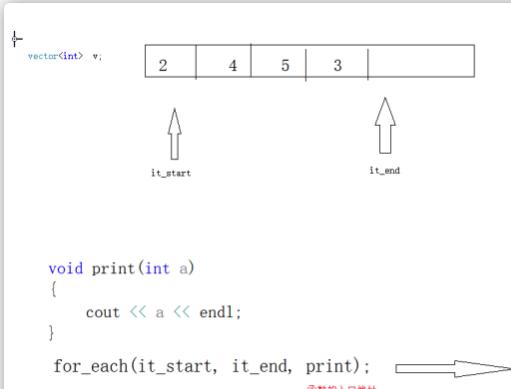
- 迭代器是一种抽象的设计概念 : 提供一种方法,使之能够依序寻访某个容器所含的各个元素,而又无需暴露该容器的内部表示方式

迭代器的种类		
输入迭代器	提供对数据的只读访问	只读,支持 ++、 ==、 !=
输出迭代器	提供对数据的只写访问	只写,支持 ++
前向迭代器	提供读写操作,并能向前推进迭代器	读写,支持 ++、 ==、 !=
双向迭代器	提供读写操作,并能向前和向后操作	读写,支持 ++、 --
随机访问迭代器	提供读写操作,并能以跳跃的方式访问容器的任意数据,是功能最前的迭代器	读写,支持 ++、 --、 [n]、 -n、 <、 <=、 >、 >=

## 1. 普通迭代器

```
1 #include <iostream>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <string>
5 #include <vector>
6 using namespace std;
7
8 void test01() {
9     vector<int> v;
10    v.push_back(2);
11    v.push_back(4);
12    v.push_back(5);
13    v.push_back(3);
14    // 若需要访问容器内的元素，需要拿到容器首元素的迭代器(指针)
15    vector<int>::iterator it_start = v.begin();
16    vector<int>::iterator it_end = v.end();
17    for (; it_start != it_end; it_start++) {
18        cout << *it_start << " ";
19    }
20    cout << endl;
21 }
22 int main() { test01(); }
23
24 // 2 4 5 3
```

## 2. 迭代器通过 for\_each 遍历



```
template<class _InIt, vector<int>::iterator
         class _Fn1> inline void (*_Fn)(int)
         void _For_each_unchecked(_InIt _First, _InIt _Last, _Fn1&
         _Func)
         {
             // perform function for each element
             for (; _First != _Last; ++_First)
                 _Func(*_First);
             print
         }
```

```
template<class _InIt, vector<int>::iterator
         class _Fn1> inline void (*_Fn)(int)
         _Fn for_each(_InIt _First, _InIt _Last, _Fn1 _Func)
         {
             // perform function for each element
             _DEBUG_RANGE_PTR(_First, _Last, _Func);
             _For_each_unchecked(_Unchecked(_First),
             _Unchecked(_Last), _Func);
             return (_Func);
         }
```

```
1 #include <iostream>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <string>
5 #include <vector>
6 using namespace std;
7 void print(int a) { cout << a << endl; }
8 void test01() {
9     vector<int> v;
10    v.push_back(2);
11    v.push_back(4);
12    v.push_back(5);
```

```

13 v.push_back(3);
14 // 若需要访问容器内的元素，需要拿到容器首元素的迭代器(指针)
15 vector<int>::iterator it_start = v.begin();
16 vector<int>::iterator it_end = v.end();
17 // for (; it_start != it_end; it_start++) {
18 //     cout << *it_start << " ";
19 // }
20 // cout << endl;
21 for_each(it_start, it_end, print);
22 }
23 int main() { test01(); }
24
25 // 2
26 // 4
27 // 5
28 // 3

```

### 3. 迭代器指向的元素是自定义的数据类型

```

1 #include <iostream>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <string>
5 #include <vector>
6 using namespace std;
7 class Person {
8 public:
9     Person(int age) { this->age = age; }
10    int age;
11 };
12 void print_person(Person &p) { cout << p.age << endl; }
13 void test() {
14     Person p1(1);
15     Person p2(2);
16     Person p3(3);
17     Person p4(4);
18     vector<Person> v;
19     v.push_back(p1);
20     v.push_back(p2);
21     v.push_back(p3);
22     v.push_back(p4);
23     vector<Person>::iterator it_start = v.begin();
24     vector<Person>::iterator it_end = v.end();
25     for (; it_start != it_end; it_start++) {
26         print_person(*it_start);
27     }
28 }
29 int main() { test(); }
30
31 // 1
32 // 2
33 // 3
34 // 4

```

## 4. 迭代器指向的是一个容器

```
1 #include <iostream>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <string>
5 #include <vector>
6 using namespace std;
7 void test() {
8     vector< vector<int> > v;
9     vector<int> v1;
10    vector<int> v2;
11    vector<int> v3;
12    for (int i = 0; i < 3; i++) {
13        v1.push_back(i);
14        v2.push_back(i + 10);
15        v3.push_back(i + 100);
16    }
17    v.push_back(v1);
18    v.push_back(v2);
19    v.push_back(v3);
20    vector< vector<int> >::iterator it = v.begin();
21    for (; it != v.end(); it++) {
22        // *it 得到的是 vector<int>
23        vector<int>::iterator it_start = (*it).begin();
24        vector<int>::iterator it_end = (*it).end();
25        for (; it_start != it_end; it_start++) {
26            cout << *it_start << endl;
27        }
28    }
29 }
30 int main() { test(); }
31 // 0
32 // 1
33 // 2
34 // 10
35 // 11
36 // 12
37 // 100
38 // 101
39 // 102
```

# 常用容器

## 一. string 容器

### 1. string 容器的基本概念

- string容器是一个类,这个容器中有一个指针,指针维护了一个数组
- string容器提供了 copy、find、insert、replace 等等功能

## 1.2 string 容器的常量操作

### I. string 的构造函数

```
1 /**
2 * @brief string 的构造函数
3 *
4 * string() : 创建一个空的字符串
5 * string(const string& str) : 使用一个 string 对象初始化另一个 string 对象
6 * string(const char *s) : 使用字符串 s 初始化
7 * string(int n,char c) : 使用 n 个字符 c 初始化 v
8 */
9 void test01() {
10     string str;
11     string str1("hello");
12     string str2(str1);
13     string str3(5, 'k');
14     cout << str1 << endl;
15     cout << str2 << endl;
16     cout << str3 << endl;
17 }
18
19 // hello
20 // hello
21 // kkkkk
```

### II. string 容器的基本操作

```
1 /**
2 * @brief string 容器的基本操作
3 *
4 * string &operator=(const char* s) : char * 类型的字符串,赋值给当前的字符串
5 * string &operator=(const string &s) : 把字符串 s 赋给当前的字符串
6 * string &operator=(char c) : 字符赋值给当前的字符串
7 * string &assign(const char *s) : 把字符串 s 赋给当前的字符串
8 * string &assign(const char *s,int n) : 把字符串 s 的前n个字符赋给当前的字符串
9 * string &assign(const string &s) : 把字符串 s 赋给当前字符串
10 * string &assing(int n,char c) : 用 n 个字符 c 赋给当前字符串
11 * string &assign(const string &s,int start,int n) : 将 s 从 start 开始 n
12 * 个字符赋值给字符串
13 *
14 */
15 void test02() {
16     string str("HelloWorld");
17     string str1("Heihei");
18     cout << str << endl;
19     str = str1;
20     cout << str << endl;
21     str = "hehe";
22     cout << str << endl;
23     str = 'c';
24     cout << str << endl;
25     str.assign(str1);
26     cout << str << endl;
27     str.assign("hehe");
28     cout << str << endl;
```

```

29     str.assign("jack", 2);
30     cout << str << endl;
31     str.assign(5, 'c');
32     cout << str << endl;
33     str.assign(str1, 2, 3);
34     cout << str << endl;
35 }
36
37 // HelloWorld
38 // Heihei
39 // hehe
40 // c
41 // Heihei
42 // hehe
43 // ja
44 // ccccc
45 // ihe

```

### III. String 的存取字符操作

```

1 /**
2 * @brief string 存取字符操作
3 *
4 * char &operator[](int n) : 通过[]方式取字符
5 * char & at(int n) : 通过 at 方法获取字符
6 */
7 void test03() {
8     string str("HelloWorld");
9     cout << str[4] << endl;
10    str[4] = 'c';
11    cout << str.at(4) << endl;
12 }
13
14 // o
15 // c

```

### IV. string 的拼接操作

```

1 /**
2 * @brief string 的拼接操作
3 *
4 * string &operator+=(const string& str) : 重载 += 操作符
5 * string &operator+=(const char* str) : 重载 += 操作符
6 * string &operator+=(const char c) : 重载 += 操作符
7 * string &append(const char *s) : 把字符串 s 连接到当前字符串结尾
8 * string &append(const char *s,int n) : 把字符串 s 的前n个字符连接到当前字符串结尾
9 * string &append(const string &s) : 同 operator+=()
10 * string &append(const string &s,int pos,int n) : 把字符串 s 中从 pos 开始的 n
   个字符连接到当前字符串结尾
11 * string &append(int n,char c) : 在当前字符串结尾添加n个字符c
12 */
13 void test04(){
14     string str1("HelloWorld");
15     string str2("123456");
16     cout << str1 << endl;

```

```

17     str1 += str2;
18     cout << str1 << endl;
19     str1 += 'c';
20     cout << str1 << endl;
21     str1.append(str2);
22     cout << str1 << endl;
23     str1.append("hehe");
24     cout << str1 << endl;
25     str1.append("hehe",2);
26     cout << str1 << endl;
27     str1.append(str2,2,3);
28     cout << str1 << endl;
29     str1.append(5,'c');
30     cout << str1 << endl;
31 }
32
33 // HelloWorld
34 // HelloWorld123456
35 // HelloWorld123456c
36 // HelloWorld123456c123456
37 // HelloWorld123456c123456hehe
38 // HelloWorld123456c123456hehehe
39 // HelloWorld123456c123456hehehe345
40 // HelloWorld123456c123456hehehe345cccccc

```

## V. string 容器的查找和替换

```

1 /**
2 * @brief string 容器的查找和替换
3 *
4 * int find(const string& str,int pos = 0) const : 查找 str 第一次出现的位置,从
5 * pos 开始查找
6 * int find(const char* s,int pos = 0) const : 查找 s 第一次出现位置,从 pos 开始
7 * 查找
8 * int find(const char* s,int pos,int n) const : 从 pos 位置查找 s 的前n个字符第
9 * 一次位置
10 * int find(const char c,int pos = 0) const : 查找字符 c 第一次出现的位置
11 * int rfind(const string& str,int pos = npos) const : 查找 str 最后一次位置,从
12 * pos 开始查找
13 * int rfind(const char* s,int pos = npos) const : 查找 s 最后一次出现位置,从
14 * pos 开始查找
15 * int rfind(const char c,int pos = 0) const : 查找字符 c 最后一次出现位置
16 * string &replace(int pos,int n,const string& str) : 替换从 pos 开始n个字符为
17 * 字符串 str
18 * string &replace(int pos,int n,const char* s) : 替换从 pos 开始的n个字符为字符
19 * 串 s
20 */
21 void test05(){
22     string str("helrlowolrd");
23     string str1("wol");
24     cout << str.find(str1) << endl;
25     cout << str.find("wol") << endl;
26     cout << str.find("world",0,2) << endl;
27     cout << str.find('o') << endl;
28     cout << str.rfind("lr") << endl;

```

```

23     str.replace(2,4,str1);
24     cout << str << endl;
25     str.replace(2,2,"123456");
26     cout << str << endl;
27 }
28
29 // 6
30 // 6
31 // 6
32 // 5
33 // 8
34 // hewolwolrd
35 // he123456lwolrd

```

## VI. string 的比较

```

1 /**
2 * @brief string 的比较
3 *
4 * compare函数在 > 时返回1,在 < 时返回 -1,在 == 时返回 0
5 * 比较区分大小写,比较时参考字典顺序,排越前面的越小
6 * 大写的A比小写的a小
7 * int compare(const string &s) const : 与字符串 s 比较
8 * int compare(const char *s) const : 与字符串 s 比较
9 */
10 void test06() {
11     string str1("Hello");
12     string str2("world");
13     // int ret = str1.compare(str2);
14     int ret = str1.compare("hello");
15     if (ret > 0) {
16         cout << "str大" << endl;
17     } else if (ret < 0) {
18         cout << "str小" << endl;
19     } else {
20         cout << "相等" << endl;
21     }
22 }
23
24 // str小

```

## VII. 获取 string 的子串

```

1 /**
2 * @brief 获取 string 的子串
3 *
4 * string substr(int pos = 0,int n = npos) const : 返回由 pos开始的n个字符组成的
5 * 字符串
6 */
7 void test07() {
8     string str1("HelloWorld");
9     string str2 = str1.substr(4, 3);
10    cout << str2 << endl;
11 }
12
13 // owo

```

## VIII. string 插入和删除

```
1 /**
2  * @brief string 插入和删除操作
3  *
4  * string &insert(int pos,const char* s) : 插入字符串
5  * string &insert(int pos,const string& str) : 插入字符串
6  * string &insert(int pos,int n,char c) : 在指定位置插入n个字符c
7  * string &erase(int pos,int n = npos) : 删除从 pos 开始的n个字符
8  */
9 void test08() {
10     string str1("elloworld");
11     string str2("hehe");
12     str1.insert(2, "kkk");
13     cout << str1 << endl;
14     str1.insert(2, str2);
15     cout << str1 << endl;
16     str1.insert(2, 10, 'r');
17     cout << str1 << endl;
18     str1.erase(3, 3);
19     cout << str1 << endl;
20 }
21
22 hekkkelloworld
23 hehehekkkelloworld
24 haaaaaaaaaaaahekkkelloworld
25 haaaaaaaahekkkelloworld
```

## IX. string 对象和 char\* 相互转换

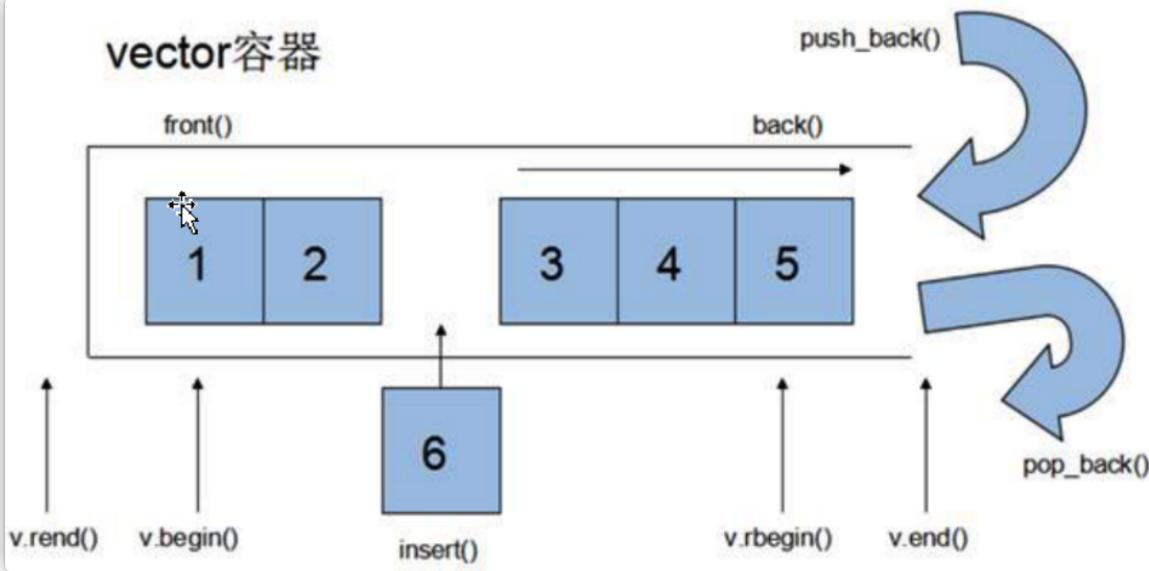
```
1 void test09() {
2     string str("hello");
3     str = "world"; // str.operator=(char *)
4
5     char *s = NULL;
6     // str.c_str(); const char *
7     s = const_cast<char *>(str.c_str());
8     cout << s << endl;
9 }
10
11 // world
```

# 二. vector容器

## 1. vector 容器的基本概念

- vector的数据安排以及操作方式,与array非常相似,两者的唯一差别在于空间的运用的灵活性
  - Array是静态空间,一旦配置了就不能改变
  - Vector是动态空间,随着元素的加入,它的内部机制会自动扩充空间以容纳新元素
- vector的实现技术,关键在于其对大小的控制以及重新配置时的数据移动效率

## vector容器



## 2. vector 容器的迭代器

```
1 #include <iostream>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <string>
5 #include <vector>
6 using namespace std;
7 /**
8 * @brief vector 迭代器
9 *
10 */
11 void test01() {
12     vector<int> v;
13     v.push_back(1);
14     v.push_back(2);
15     v.push_back(3);
16     v.push_back(4);
17     // 从头开始 1 2 3 4
18     // vector<int>::iterator it_start = v.begin();
19     // vector<int>::iterator it_end = v.end();
20     // 从尾开始 4 3 2 1
21     vector<int>::reverse_iterator it_start = v.rbegin();
22     vector<int>::reverse_iterator it_end = v.rend();
23     for (; it_start != it_end; it_start++) {
24         cout << *it_start << " ";
25     }
26     cout << endl;
27 }
28 /**
29 * @brief 空间配置
30 *
31 */
32 void test02() {
33     vector<int> v;
34     for (int i = 0; i < 100; i++) {
35         v.push_back(i);
36         // v.capacity() : 容器的容量
37         cout << v.capacity() << endl;
38 }
```

```
38     }
39 }
40 int main() {
41     test01();
42     test02();
43 }
```

### 3. vector 的数据结构

- vector 所采用的数据结构非常简单,线性连续空间,它以两个迭代器 Myfirst 和 Mylast 分别指向配置得来的连续空间中目前已使用的范围,并以迭代器 Myend 指向整块连续内存空间的尾端
- 为了降低空间配置时的速度成本,vector 实际配置的大小可能比客户端需求大一些,以备将来可能的扩充,这边是容量的概念
- 一个 vector 的容量永远大于或等于其大小,一旦容量等于大小,便是满载,下次再有新增元素,整个 vector 容器就得另觅居所

## 4. vector API 操作

### I. vector 构造函数

```
1 /**
2 * @brief vector 的构造函数
3 *
4 * vector<T> v : 采用模板实现类实现,默认构造函数
5 * vector(v.begin(),v.end()) : 将 v[begin(),end()] 区间中的元素拷贝给本身
6 * vector(n,elem) : 构造函数将n个elem拷贝给本身
7 * vector(const vector &vec) : 拷贝构造函数
8 */
9 void print(int a) { cout << a << endl; }
10 void test03() {
11     vector<int> v1;
12     v1.push_back(1);
13     v1.push_back(2);
14     v1.push_back(3);
15     v1.push_back(4);
16     // vector<int> v2(v1) == vector<int> v2(v1.begin(), v1.end())
17     // vector<int> v2(v1);
18     // vector<int> v2(v1.begin(), v1.end());
19     vector<int> v2(v1.begin() + 1, v1.end());
20     for_each(v2.begin(), v2.end(), print);
21 }
22
23 // 2
24 // 3
25 // 4
```

### II. vector 常用赋值操作

```
1 /**
2 * @brief 常用赋值操作
3 *
4 * assign(beg,end) : 将 [beg,end] 区间中的数据拷贝给本身
5 * assign(n,elem) : 将n个elem拷贝赋值给本身
6 * vector& operator=(const vector &vec) : 重载等号操作符
```

```

7  * swap(vec) : 将vec与本身的元素互换
8  */
9 void test04() {
10    vector<int> v1;
11    v1.push_back(1);
12    v1.push_back(2);
13    v1.push_back(3);
14    v1.push_back(4);
15
16    vector<int> v2;
17    v2.push_back(5);
18    v2.push_back(6);
19    v2.push_back(7);
20    v2.push_back(8);
21
22    // v2.assign(v1.begin(), v1.end()) == v2 = v1
23    // v2.assign(v1.begin(), v1.end());
24    // v2 = v1;
25    // v2.swap(v1);
26    v2.assign(10, 5);
27    for_each(v2.begin(), v2.end(), print);
28 }

```

### III. vector 大小操作

```

1 /**
2  * @brief vector 大小操作
3  *
4  * size() : 返回容器中元素的个数
5  * empty() : 判断容器是否为空
6  * resize(int 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
7  *                   如果容器变短,则末尾超出容器长度的元素被删除
8  * resize(int num, elem) :
9  * 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
10 *                   如果容器变短,则末尾超出容器长度的元素被删除
11 * capacity() : 容器的容量
12 * reserve(int len) : 容器预留len个元素长度,预留位置不初始化,元素不可访问
13 */
14 void test05() {
15    vector<int> v1;
16    v1.push_back(1);
17    v1.push_back(2);
18    v1.push_back(3);
19    v1.push_back(4);
20    v1.push_back(5);
21    cout << v1.size() << endl;
22    cout << v1.capacity() << endl;
23    v1.resize(10, 6);
24    if (!v1.empty()) {
25        cout << "不为空" << endl;
26    }
27    for_each(v1.begin(), v1.end(), print);
28
29    vector<int> v2;
30    // 设置容量
31    v2.reserve(1000);
32    cout << v2.size() << endl;

```

```

33     cout << v2.capacity() << endl;
34 }
35
36 // 5
37 // 8
38 // 不为空
39 // 1
40 // 2
41 // 3
42 // 4
43 // 5
44 // 6
45 // 6
46 // 6
47 // 6
48 // 6
49 // 0
50 // 1000

```

## IV. vector 容器的存取操作

```

1 /**
2 * @brief vector 容器的存取操作
3 *
4 * at(int idx) : 返回索引 idx 所指的数据,如果idx越界,抛出 out_of_range 异常
5 * operator[] : 返回索引 idx 所指的数据,越界时,运行直接报错
6 * front() : 返回容器中第一个数据元素
7 * back() : 返回容器中最后一个数据元素
8 */
9 void test06() {
10     vector<int> v1;
11     v1.push_back(1);
12     v1.push_back(2);
13     v1.push_back(3);
14     v1.push_back(4);
15     v1.push_back(5);
16     cout << v1.at(2) << endl;
17     cout << v1[2] << endl;
18     cout << v1.front() << endl;
19     cout << v1.back() << endl;
20 }
21
22 // 3
23 // 3
24 // 1
25 // 5

```

## V. vector 容器插入和删除操作

```

1 /**
2 * @brief vector 插入和删除操作
3 *
4 * insert(const_iterator pos,int count,ele) :
5 *                                     迭代器指向位置pos插入count
6 * 个元素ele
7 * push_back(ele) : 尾部插入元素ele

```

```

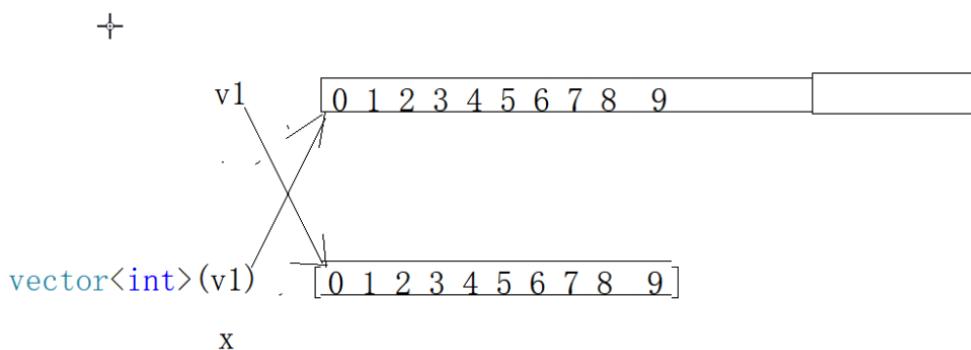
7  * pop_back() : 删除最后一个元素
8  * erase(const_iterator start,const_iterator end) : 删除迭代器从 start 到 end
9  *
10 * erase(const_iterator pos) : 删除迭代器指向的元素
11 * clear() : 删除容器中所有元素
12 */
13 void test07() {
14     vector<int> v1;
15     v1.push_back(1);
16     v1.push_back(2);
17     v1.push_back(3);
18     v1.push_back(4);
19     v1.push_back(5);
20     // v1.insert(v1.begin() + 2, 3, 9);
21     // v1.pop_back();
22     // v1.erase(v1.begin() + 3, v1.end());
23     // v1.erase(v1.begin() + 3);
24     v1.clear();
25     for_each(v1.begin(), v1.end(), print);
26 }

```

## 5. vector 容器的案例

### I. 使用 swap 收缩空间

```
vector<int>(v1). swap(v1);
```



```

1 /**
2  * @brief swap 收缩空间
3  *
4  */
5 void test08() {
6     vector<int> v1;
7     for (int i = 0; i < 10000; i++) {
8         v1.push_back(i);
9     }
10    cout << v1.size() << " " << v1.capacity() << endl;
11    v1.resize(10);
12    cout << v1.size() << " " << v1.capacity() << endl;
13    vector<int>(v1).swap(v1);
14    cout << v1.size() << " " << v1.capacity() << endl;
15 }
16
17 // 10000 16384
18 // 10 16384

```

```
19 // 10 10
```

## II. 计算重新开辟了多少次内存空间

```
1 /**
2 * @brief 计算重新开辟了多少次内存空间
3 *
4 */
5 void test09() {
6     vector<int> v;
7     int *p = NULL;
8     int count = 0;
9     for (int i = 0; i < 10000; i++) {
10         v.push_back(i);
11         if (p != &v[0]) {
12             count++;
13             p = &v[0];
14         }
15     }
16     cout << count << endl;
17 }
18
19 // 15
```

## 6. Vector 容器排序

```
1 /**
2 * @brief vector 容器培训
3 *
4 */
5 bool compare(int a, int b) { return a > b; }
6 void test10() {
7     vector<int> v1;
8     v1.push_back(1);
9     v1.push_back(3);
10    v1.push_back(2);
11    v1.push_back(5);
12    v1.push_back(4);
13    // sort(v1.begin(), v1.end()); 默认是从小到大
14    sort(v1.begin(), v1.end(), compare);
15    for_each(v1.begin(), v1.end(), print);
16 }
17
18 // 5
19 // 4
20 // 3
21 // 2
22 // 1
23
24 class Person {
25 public:
26     Person(int age, string s) {
27         this->age = age;
28         this->name = s;
29     }
30     int age;
```

```

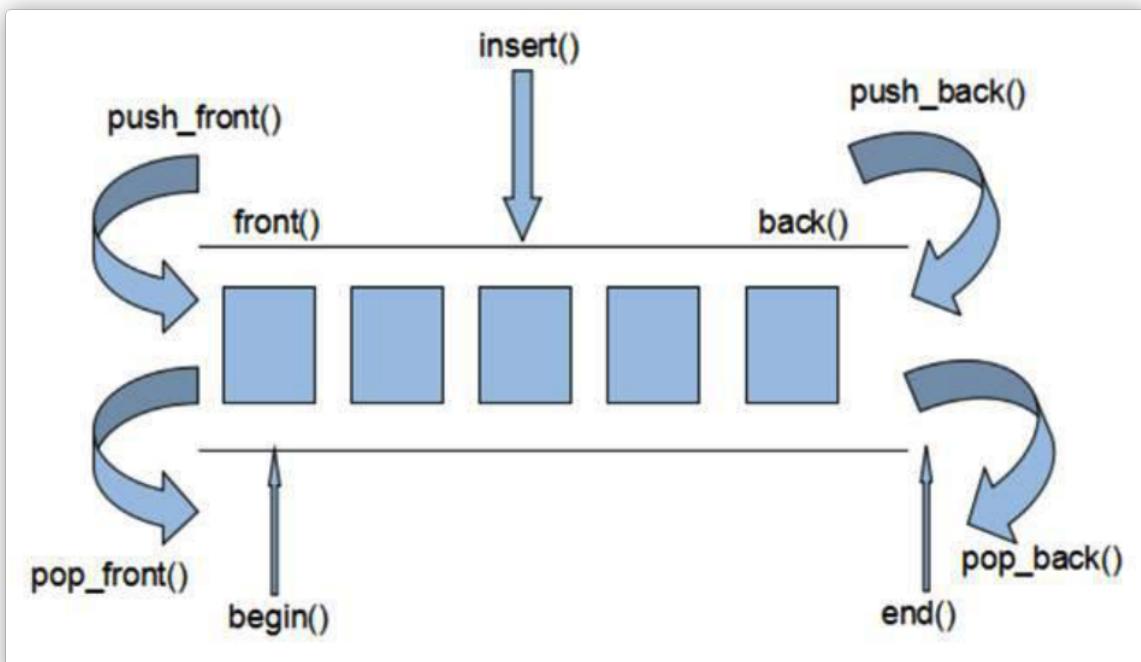
31     string name;
32 }
33 void print_person(Person &p) { cout << p.age << " " << p.name << endl; }
34 bool compare_person(Person &p1, Person &p2) { return p1.age > p2.age; }
35 void test11() {
36     vector<Person> v;
37     v.push_back(Person(10, "安琪拉"));
38     v.push_back(Person(40, "韩信"));
39     v.push_back(Person(50, "盖伦"));
40     v.push_back(Person(20, "猴子"));
41     v.push_back(Person(30, "三眼怪"));
42
43     for_each(v.begin(), v.end(), print_person);
44     sort(v.begin(), v.end(), compare_person);
45     for_each(v.begin(), v.end(), print_person);
46 }
47
48 // 10 安琪拉
49 // 40 韩信
50 // 50 盖伦
51 // 20 猴子
52 // 30 三眼怪
53 // 50 盖伦
54 // 40 韩信
55 // 30 三眼怪
56 // 20 猴子
57 // 10 安琪拉

```

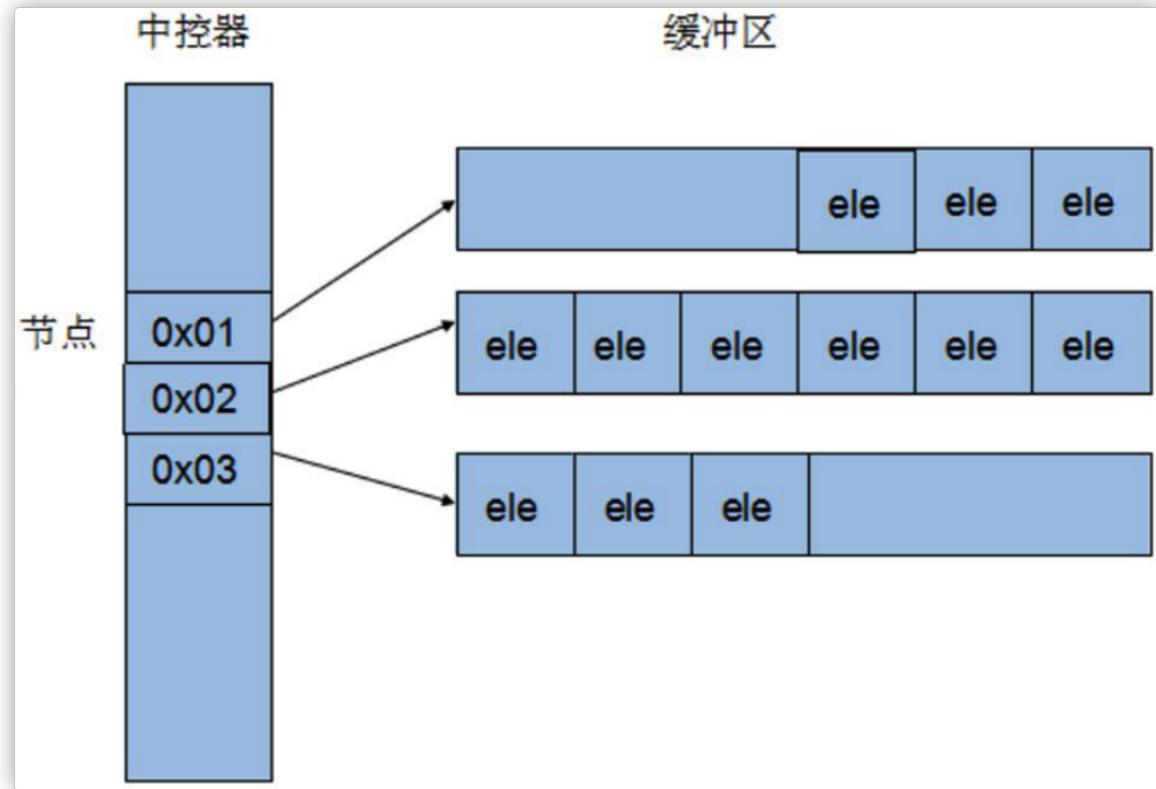
## 三. deque 容器

### 1. deque 容器的基本概念

- vector 容器是单向开口的连续内存空间,deque 是一种双向开口的连续线性空间
- 双向开口,可以在头尾两端分别做元素的插入和删除操作
- vector 容器也可以在头尾两端插入元素,但是在其头部操作效率奇差



## 2. deque 容器的实现原理



## 3. deque 常用API

### I. deque 容器的构造函数

```
1 #include <deque>
2 #include <iostream>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <string>
6 using namespace std;
7
8 /**
9 * @brief deque 的构造函数
10 */
11 deque<T> deque : 默认构造函数
12 deque(beg,end) : 构造函数将 [beg,end) 区间中的元素拷贝给本身
13 deque(n,elem) : 构造函数将n个elem拷贝给本身
14 deque(const deque &deq) : 拷贝构造函数
15 */
16 void print(int a) { cout << a << endl; }
17 void test01() {
18     deque<int> d1;
19     d1.push_back(1);
20     d1.push_back(2);
21     d1.push_back(3);
22     deque<int> d2(d1);
23     for_each(d2.begin(), d2.end(), print);
24     deque<int> d3(d1.begin(), d1.end());
25     for_each(d3.begin(), d3.end(), print);
26     deque<int> d4(5, 8);
27     for_each(d4.begin(), d4.end(), print);
```

```
28 }
29 int main() { test01(); }
30
31 1
32 2
33 3
34 1
35 2
36 3
37 8
38 8
39 8
40 8
41 8
```

## II. Deque 容器的赋值操作

```
1 /**
2 * @brief deque 赋值操作
3 *
4 * assign(beg,end) : 将[beg,end)区间中的数据拷贝赋值给本身
5 * assign(n,elem) : 将n个elem拷贝赋值给本身
6 * deque& operator=(const deque &deq) : 重载等号操作符
7 * swap(deq) : 将deq与本身的元素互换
8 */
9 void test02() {
10     deque<int> d1;
11     d1.push_back(1);
12     d1.push_back(2);
13     d1.push_back(3);
14     deque<int> d2;
15     d2.assign(d1.begin(), d1.end());
16     for_each(d2.begin(), d2.end(), print);
17     deque<int> d3;
18     d3.assign(3, 9);
19     for_each(d3.begin(), d3.end(), print);
20     deque<int> d4;
21     d4 = d3;
22     for_each(d4.begin(), d4.end(), print);
23     d3.swap(d2);
24     for_each(d3.begin(), d3.end(), print);
25 }
26
27 1
28 2
29 3
30 9
31 9
32 9
33 9
34 9
35 9
36 1
37 2
38 3
```

### III. Deque 容器的大小操作

```
1 /**
2 * @brief deque 大小操作
3 *
4 * deque.size() : 返回容器中元素的个数
5 * deque.empty() : 判断容器是否为空
6 * deque.resize(num) : 重新指定容器的长度,若容器变长,则以默认值填充新位置
7 *                      如果容器变短,则末尾超出容器长度的元素被删除
8 * deque.resize(num, elem) :
9 * 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
10 *                      如果容器变短,则末尾超出容器长度的元素被删除
11 */
12 void test03() {
13     deque<int> d1;
14     d1.push_back(1);
15     d1.push_back(2);
16     d1.push_back(3);
17     cout << d1.size() << endl;
18     if (!d1.empty()) {
19         cout << "不为空!" << endl;
20     }
21     d1.resize(2);
22     for_each(d1.begin(), d1.end(), print);
23     d1.resize(3, 5);
24     for_each(d1.begin(), d1.end(), print);
25 }
26
27 3
28 不为空!
29 1
30 2
31 1
32 2
33 5
```

### IV. 双端的删除操作

```
1 /**
2 * @brief deque 双端插入和删除操作
3 *
4 * push_back(elem) : 在容器尾部添加一个数据
5 * push_front(elem) : 在容器头部插入一个数据
6 * pop_back() : 删除容器最后一个数据
7 * pop_front() : 删除容器第一个数据
8 */
9 void test04() {
10     deque<int> d1;
11     d1.push_back(1);
12     d1.push_back(2);
13     d1.push_back(3);
14     d1.push_back(4);
15     d1.push_back(5);
16     d1.push_back(6);
17     for_each(d1.begin(), d1.end(), print);
18     d1.pop_back();
```

```
19     d1.pop_front();
20     for_each(d1.begin(), d1.end(), print);
21 }
22
23 1
24 2
25 3
26 4
27 5
28 6
29 2
30 3
31 4
32 5
```

## V. deque 容器的存取操作

```
1 /**
2 * @brief deque 容器的存取操作
3 *
4 * at(idx) : 返回索引idx所指的数据,如果idx越界,抛出 out_of_range
5 * operator[idx] : 返回索引idx所指的数据,如果idx越界,不抛出异常,直接出错
6 * front() : 返回第一个数据
7 * back() : 返回最后一个数据
8 */
9 void test05() {
10     deque<int> d1;
11     d1.push_back(1);
12     d1.push_back(2);
13     d1.push_back(3);
14     cout << d1.at(2) << endl;
15     d1[2] = 10;
16     cout << d1.front() << endl;
17     cout << d1.back() << endl;
18 }
19
20 3
21 1
22 10
```

## VI. deque 容器的插入操作

```
1 /**
2 * @brief deque插入操作
3 *
4 * insert(pos,elem) : 在pos位置插入一个elem元素的拷贝,返回新数据的位置
5 * insert(pos,n,elem) : 在pos位置插入n个elem数据,无返回值
6 * insert(pos,beg,end) : 在pos位置插入[beg,end)区间的数据,无返回值
7 */
8 void test06() {
9     deque<int> d1;
10    d1.push_back(1);
11    d1.push_back(2);
12    d1.push_back(3);
13    d1.insert(d1.begin(), 10);
14    for_each(d1.begin(), d1.end(), print);
```

```

15 d1.insert(d1.begin(), 3, 2);
16 for_each(d1.begin(), d1.end(), print);
17 deque<int> d2;
18 d2.insert(d2.begin(), d1.begin(), d1.end());
19 }
20
21 10
22 1
23 2
24 3
25 2
26 2
27 2
28 10
29 1
30 2
31 3
32

```

## VII. deque 容器的删除操作

```

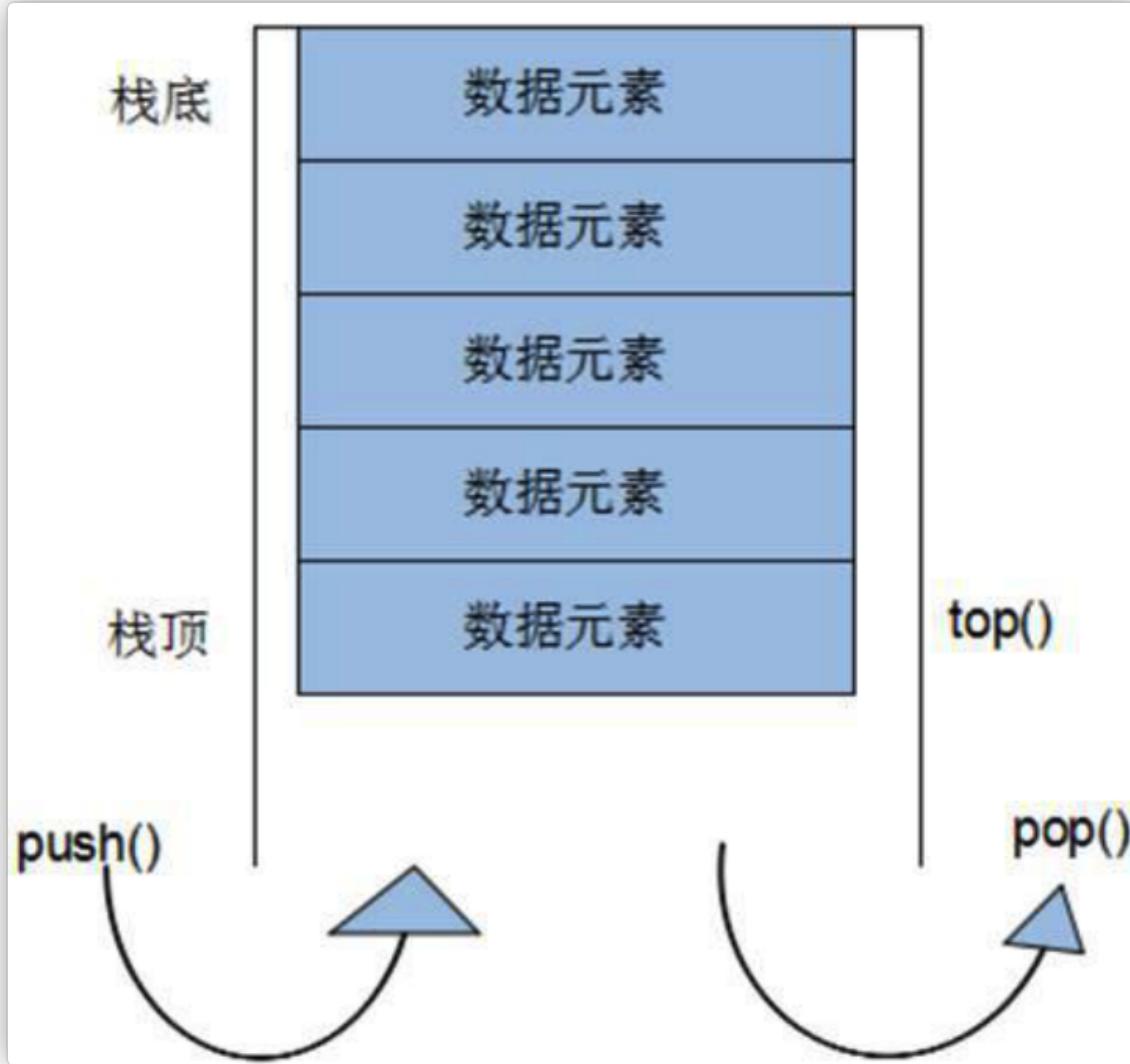
1 /**
2 * @brief deque 删除操作
3 *
4 * clear() : 移除容器的所有数据
5 * erase(beg,end) : 删除[beg,end]区间的数据,返回下一个数据的位置
6 * erase(pos) : 删除pos位置的数据,返回下一个数据的位置
7 */
8 void test07() {
9     deque<int> d1;
10    d1.push_back(1);
11    d1.push_back(2);
12    d1.push_back(3);
13    // d1.clear();
14    // for_each(d1.begin(), d1.end(), print);
15
16    // d1.erase(d1.begin()+1,d1.end());
17    d1.erase(d1.begin() + 1);
18    for_each(d1.begin(), d1.end(), print);
19 }
20
21 1
22 3

```

# 四. stack 容器

## 1. stack 容器基本概念

- stack 是一种先进后出(First In Last Out,FILO)的数据结构,它只有一个出口
- stack 容器允许新增元素,移除元素,取得栈顶元素,但是除了最顶端外,没有任何其他方法可以存取 stack 的其他元素
- stack 不允许有遍历行为
- 有元素推入栈的操作成为 : push, 将元素推出 stack 的操作成为 : pop



## 2. stack 没有迭代器

- stack 所有元素的进出都必须符合“先进后出”的条件,只有 stack 顶端的元素,才有机会被外界取用
- stack 不提供遍历功能,也不提供迭代器

## 3. stack 常用API

### I. stack 构造函数

```

1 | stack<T> stkT : stack 采用模板类实现,stack对象的默认构造形式
2 | stack(const stack &stkT) : 拷贝构造函数

```

### II. stack 赋值操作

```

1 | stack& operator=(const stack &stk) : 重载等号操作符

```

### III. stack 数据存取操作

```

1 | push(elem) : 向栈顶添加元素
2 | pop() : 从栈顶移除第一个元素
3 | top() : 返回栈顶元素

```

## IV. stack 大小操作

- 1 `empty()` : 判断堆栈是否为空
- 2 `size()` : 返回堆栈的大小

## 五. queue 容器

### 1. queue 容器的基本概念

Queue 是一种先进先出(First In First Out)的数据结构,它有两个出口,queue 容器允许从一端新增元素,从另一端移除元素



### 2. queue 没有迭代器

- Queue 所有元素的进出都必须符合“先进先出”的条件,只有queue的顶端元素,才有机会被外界取用
- Queue 不提供遍历功能,也不提供迭代器

### 3. queue 常用API

#### I. queue 的构造函数

- 1 `queue<T> queT` : queue 采用模板类实现,queue 对象的默认构造形式
- 2 `queue(const queue &que)` : 拷贝构造函数

#### II. queue 存取、插入和删除操作

- 1 `push(elem)` : 往队尾添加元素
- 2 `pop()` : 从队头移除第一个元素
- 3 `back()` : 返回最后一个元素
- 4 `front()` : 返回第一个元素

### III. queue 赋值操作

1 | `queue& operator=(const queue &que)` : 重载等号操作符

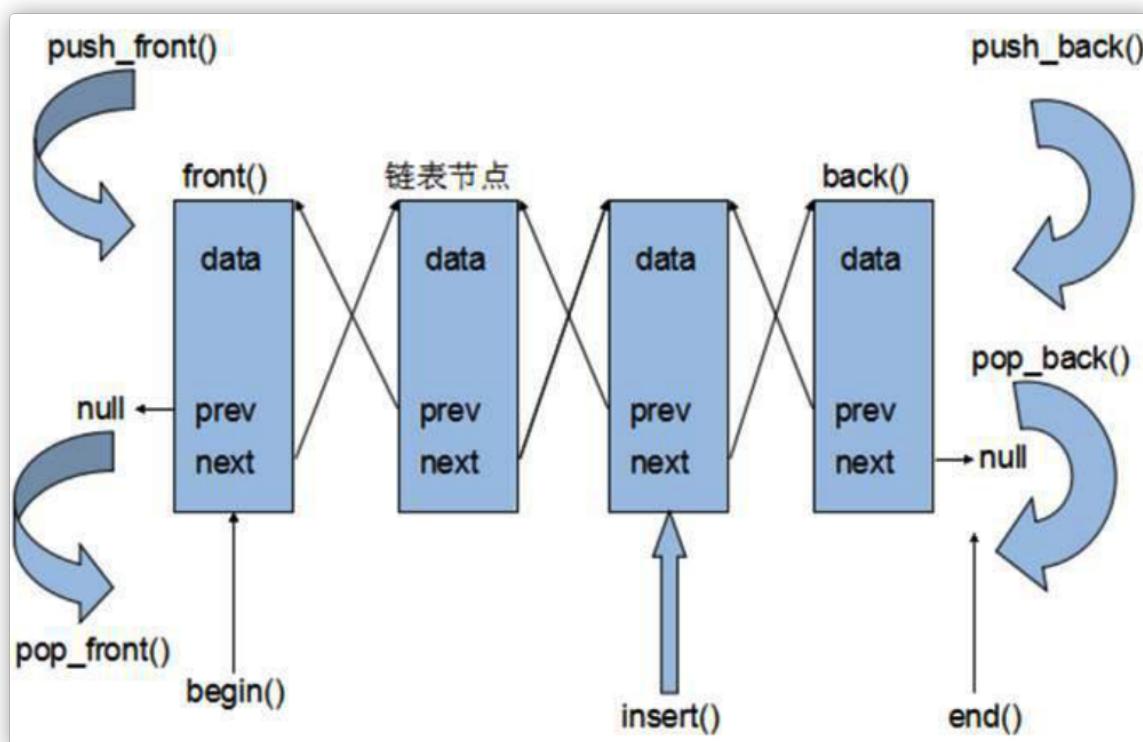
### IV. queue 大小操作

1 | `empty()` : 判断队列是否为空  
2 | `size()` : 返回队列的大小

## 六. list 容器

### 1. list 容器基本概念

- 链表是一种物理存储单元上非连续、非顺序的存储结构,数据元素的逻辑顺序是通过链表中的指针链接次序实现的
- 链表由一系列结点(链表中每一个元素称为结点)组成,结点可以在运行时动态生成
- 每个结点包括两个部分 : 一个是存储数据元素的数据域,另一个是存储下一个结点地址的指针域
- 相较于 `vector` 的连续线性空间, `list` 就显得负责许多,它的好处是每次插入或者删除一个元素,就是配置或者释放一个元素的空间
- `list` 对于空间的运用有绝对的精准,一点也不浪费
- 对于任何位置的元素插入或者元素的移除, `list` 永远是常数时间
- `list` 和 `vector` 是两个最常被使用的容器
- `list` 容器是一个双向链表
- 采用动态存储分配,不会造成内存浪费和溢出
- 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
- 链表灵活,但是空间和时间额外耗费较大



## 2. list 容器的迭代器

- List 迭代器必须有能力执行list的节点，并有能力进行正确的递增、递减、取值、成员存取操作
- 由于 List 是一个双向链表，迭代器必须能够具备前移、后移的能力，所以 List 容器提供的是 **Bidirectional Iterators**
- 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，因为 vector 的插入操作可能造成记忆体重新配置，导致原有的迭代器全部失效，甚至 List 元素的删除，也只有被删除的那个元素的迭代器失效，其他迭代器不受影响

## 3. list 容器的数据结构

list 容器不仅是一个双向链表，而且还是一个循环的双向链表

## 4. list 常用API

### I. list 构造函数

```
1 list<T> lstT : list 采用模板类实现,对象的默认构造形式  
2 list(beg,end) : 构造函数将[beg,end)区间中的元素拷贝给本身  
3 list(n,elem) : 构造函数将n个elem拷贝给本身  
4 list(const list &lst) : 拷贝构造函数
```

### II. list 数据元素插入和删除操作

```
1 push_back(elem) : 在容器尾部加入一个元素  
2 pop_back() : 删除容器中最后一个元素  
3 push_front(elem) : 在容器开头插入一个元素  
4 pop_front() : 从容器开头移除第一个元素  
5 insert(pos,elem) : 在pos位置插入elem元素的拷贝,返回新数据的位置  
6 insert(pos,n,elem) : 在pos位置插入n个elem数据,无返回值  
7 insert(pos,beg,end) : 在pos位置插入[beg,end)区间的数据,无返回值  
8 clear() : 移除容器的所有数据  
9 erase(beg,end) : 删除[beg,end)区间的数据,返回下一个数据的位置  
10 erase(pos) : 删除pos位置的数据,返回下一个数据的位置  
11 remove(elem) : 删除容器中所有与elem值匹配的元素
```

### III. list 大小操作

```
1 size() : 返回容器中元素的个数  
2 empty() : 判断容器是否为空  
3 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  
4 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
```

### IV. list 赋值操作

```
1 assign(beg,end) : 将[beg,end)区间中的数据拷贝赋值给本身  
2 assign(n,elem) : 将n个elem拷贝赋值给本身  
3 list& operator=(const list &lst) : 重载等号操作符  
4 swap(lst) : 将lst与本身的元素互换
```

## V. list 数据的存取

```
1 | front() : 返回第一个元素  
2 | back() : 返回最后一个元素
```

## VI. list 反转排序

```
1 | reverse() : 反转链表  
2 | sort() : list排序
```

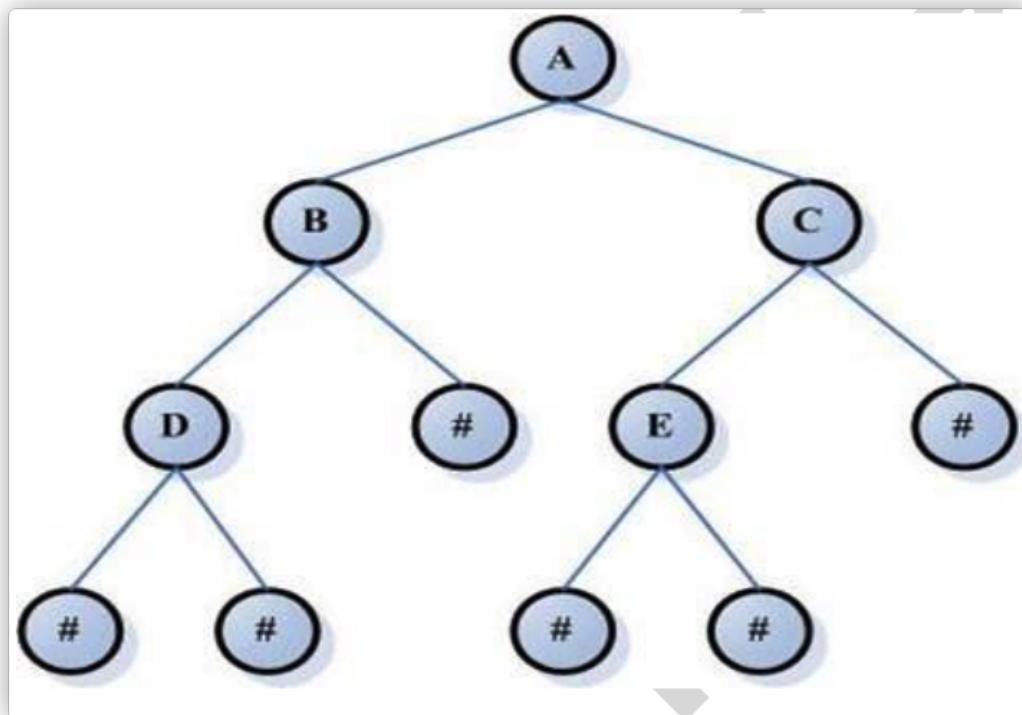
# 七. set/multiset 容器

## 1. set/multiset 容器基本概念

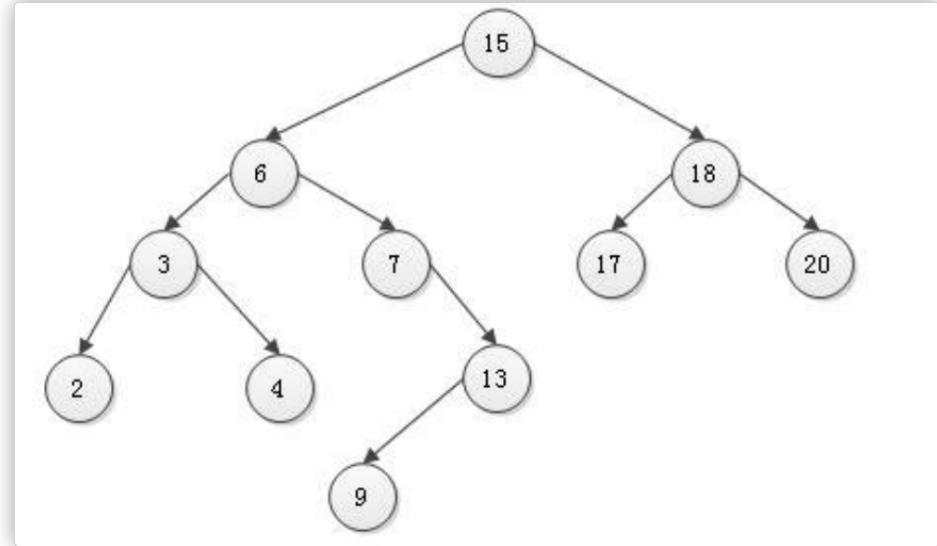
- Set 的特性是：所有元素都会根据元素的键值自动被排序
- Set 的元素不像 map 那样可以同时拥有实值和键值,Set 的元素即使键值又是实值
- Set 不允许两个元素有相同的键值
- 不能通过 Set 的迭代器改变 Set 元素的值,因为 Set 元素值就是其键值,关系到 Set 元素的排序规则.Set 的 iterator 是一种 const\_iterator
- Set 拥有和 list 某些相同的性质,当对容器中的元素进行插入操作或者删除操作的时候,操作之前所有的迭代器,在操作完成之后依然有效,被删除的那个元素的迭代器必然是一个例外
- multiset 特性及用法和 set 完全相同,唯一的差别在于它允许键值重复
- set 和 multiset 的底层实现是红黑树,红黑树为平衡二叉树的一种

树的简单知识：

- 二叉树就是任何节点最多只允许有两个子节点,分别是左子节点和右子节点



- 二叉搜索树,是指二叉树中的节点按照一定的规则进行排序,使得对二叉树中元素访问更加高效
  - 二叉搜索树的放置规则是：任何节点的元素值一定大于其左子树中的每一个节点的元素值,并且小于其右子树的值



- 如果由于我们的输入或者经过我们插入或者删除操作,二叉树失去平衡,造成搜索效率降低
- 平衡二叉树中的平衡,不是指的完全平衡

## 2. set 常用API

### I. set 构造函数

```

1 | set<T> st : 默认构造函数
2 | multiset<T> mst : multiset 默认构造函数
3 | set(const set &st) : 拷贝构造函数

```

### II. set 赋值操作

```

1 | set& operator=(const set &st) : 重载等号操作符
2 | swap(st) : 交换两个集合容器

```

### III. set 大小操作

```

1 | size() : 返回容器中元素的数目
2 | empty() : 判断容器是否为空

```

### IV. set 插入和删除操作

```

1 | insert(elem) : 在容器中插入元素
2 | clear() : 清除所有元素
3 | erase(pos) : 删除pos迭代器所指的元素,返回下一个元素的迭代器
4 | 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
5 | erase(elem) : 删除容器中值为elem的元素

```

### V. set 查找操作

```
1 | find(key) : 查找键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()
2 | count(key) : 查找键key的元素个数
3 | lower_bound(keyElem) : 返回第一个key>=keyElem元素的迭代器
4 | upper_bound(keyElem) : 返回第一个key>keyElem元素的迭代器
5 | equal_range(keyElem) : 返回容器中key与keyElem相等的上下限的两个迭代器
```

### 3. 对组 pair

对组(pair)将一对值组合成一个值,这一对值可以具有不同的数据类型,两个值可以分别用 pair 的两个公有属性 first 和 second 访问

```
1 | 类模板 : template <class T1,class T2> struct pair
2 |
3 | 第一种方法创建一个对组
4 | pair<string,int> pair1(string("name"),20);
5 | cout << pair1.first << endl; 访问pair第一个值
6 | cout << pair1.second << endl; 访问pair第二个值
7 |
8 | 第二种
9 | pair<string,int> pair2 = make_pair("name",20);
10 | cout << pair2.first << endl;
11 | cout << pair2.second << endl;
12 |
13 | 第三种=赋值
14 | pair<string,int> pair3 = pair2;
15 | cout << pair3.first << endl;
16 | cout << pair3.second << endl;
```

## 八. map/multimap 容器

### 1. map/multimap 基本概念

- Map 的特性,所有元素都会根据元素的键值自动排序,Map 所有的元素都是 pair,同时拥有实值和键值,pair的第一元素被视为键值.第二元素被视为实值,map不允许两个元素有相同的键值
- 不能通过 Map 的迭代器改变 Map 的键值,因为 Map 的键值关系到 Map 元素的排列规则,任意改变 Map 键值将会严重破坏 Map 组织.但是如果想要修改元素的实值,那么是可以的
- Map 和 List 拥有相同的某些性质,当对它的容器元素进行新增操作或者删除操作时,操作之前的所有迭代器,在操作完成之后依然有效,当然被删除的那个元素的迭代器必然是个例外
- Multimap 个 Map 的操作类型,唯一的区别 Multimap 键值可以重复
- Map 和 Multimap 都是以红黑树为底层实现机制

### 2. map/multimap 常用API

#### I. map 构造函数

```
1 | map<T1,T2> mapTT : map 默认构造函数
2 | map(const map &map) : 拷贝构造函数
```

## II. map 赋值操作

```
1 | map& operator=(const map &map) : 重载等号操作符  
2 | swap(mp) : 交换两个集合容器
```

## III. map 大小操作

```
1 | size() : 返回容器中元素的数据  
2 | empty() : 判断容器是否为空
```

## IV. map 插入数据元素操作

```
1 | map.insert(...) : 往容器中插入元素,返回 pair<iterator,bool>  
2 |  
3 | map<int,string> mapStu;  
4 | // 第一种 通过pair的方式插入对象  
5 | mapStu.insert(pair<int,string>(3,"小张"));  
6 | // 第二种 通过pair的方式插入对象  
7 | mapStu.insert(make_pair(-1,"校长"));  
8 | // 第三种 通过value_type的方式插入对象  
9 | mapStu.insert(map<int,string>::value_type(1,"小李"));  
10 | // 第四种 通过数组的方式插入值  
11 | mapStu[3] = "小刘";  
12 | mapStu[5] = "小王";
```

## V. map 删除操作

```
1 | clear() : 删除所有元素  
2 | erase(pos) : 删除pos迭代器所指的元素,返回下一个元素的迭代器  
3 | 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  
4 | erase(keyElem) : 删除容器中key为keyElem的对组
```

## VI. map 查找操作

```
1 | find(key) : 查找键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()  
2 | count(keyElem) : 返回容器中key为keyElem的对组个数,对map来说,要么是0,要么是1;对multimap来说,值可能大于1  
3 | lower_bound(keyElem) : 返回第一个key>=keyElem元素的迭代器  
4 | upper_bound(keyElem) : 返回第一个key>keyElem元素的迭代器  
5 | equal_range(keyElem) : 返回容器中key与keyElem相等的上下限的两个迭代器
```

# 九. STL 容器使用时机

	vector	deque	list	set	multiset	map	multimap
典型内存结构	单端数组	双端数组	双向链表	二叉树	二叉树	二叉树	二叉树
可随机存取	是	是	否	否	否	对 key 而言: 不是	否

	vector	deque	list	set	multiset	map	multimap
元素搜寻速度	慢	慢	非常慢	快	快	对 key 而言：快	对 key 而言：快
元素安插移除	尾端	头尾两端	任何位置	-	-	-	-

- **vector** 的使用场景：比如软件历史操作记录的存储,经常查看历史记录但却不会删除记录
- **deque** 的使用场景：比如排队购票系统,对排队者的存储可以采用queue,支持头端的快速移除,尾端的快速添加.如果采用vector,则头端移除时,会移动大量的数据,速度慢

### vector 与 deque 的比较

- **vector.at()** 比 **deque.at()** 效率高,比如 **vector.at(0)** 是固定的,**deque** 的开始位置却是不固定的
- 如果有大量释放操作的话,**vector** 花的时间更少,这跟两者的内部实现有关
- **deque** 支持头部的快速插入与快速移除,这是**deque**的优点
- **list** 的使用场景：比如公交车乘客的存储,随时可能有乘客上车,支持频繁的不确定位置元素的移除插入
- **set** 的使用场景：比如对手机游戏的个人得分记录的存储,存储要求从高分到低分的顺序排列
- **map** 的使用场景：比如按ID号存储十万个用户,想要快速通过ID查找对应的用户,二叉树的查找效率,这是就体现出来了.如果是 **vector** 容器,最坏的情况下可能要遍历完整个容器才能找到该用户

# 算法

## 一. 函数对象

- 函数对象就是类中重载了(),operator()函数,可以像函数一样调用,所以我们也把函数对象叫做仿函数
- 函数对象(仿函数)是一个类,不是一个函数
- 函数对象(仿函数)重载了“()”操作符使得它可以像函数一样调用
- 假如某个类有一个重载的operator(),而且重载的operator()要求获取一个参数,我们就将这个类称为“一元仿函数”(unary functor);相反,如果重载的operator()要求获取两个参数,就将这个类成为“二元仿函数”(binary functor)

```

1 #include <algorithm>
2 #include <iostream>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <string>
6 #include <vector>
7 using namespace std;
8
9 class MyAdd {
10 public:
11     int operator()(int a, int b) { return a + b; }
12 };
13 void test01() {
14     MyAdd p;
15     cout << p(2, 3) << endl;

```

```

16     cout << MyAdd()(4, 5) << endl;
17 }
18
19 void print(int a) { cout << a << endl; }
20 class Print {
21 public:
22     void operator()(int a) { cout << a << endl; }
23 };
24
25 bool compare(int a, int b) { return a > b; }
26 class Compare {
27 public:
28     bool operator()(int a, int b) { return a > b; }
29 };
30 void test02() {
31     vector<int> v;
32     v.push_back(1);
33     v.push_back(2);
34     v.push_back(4);
35     v.push_back(5);
36     sort(v.begin(), v.end(), Compare());
37     for_each(v.begin(), v.end(), Print());
38 }
39 int main() {
40     test01();
41     test02();
42 }
43
44 5
45 9
46 5
47 4
48 2
49 1

```

- 函数对象通常不定义构造函数和析构函数,所以在构造和析构时不会发生任何问题,避免了函数调用的运行时问题
- 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
- 函数对象可内联编译,性能好.用函数指针几乎不可能
- 模板函数对象使函数具有通用性,这也是它的优势之一

## 二. 谓词

- 谓词是指普通函数或重载的operator()返回值是bool类型的函数对象(仿函数)
- 如果operator接收一个参数,那么叫做一元谓词;如果接收两个参数,那么叫做二元谓词
- 谓词可作为一个判断式

```

1 #include <algorithm>
2 #include <iostream>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <string>
6 #include <vector>
7 using namespace std;
8
9 class Print {

```

```

10 public:
11     void operator()(int a) { cout << a << endl; }
12 };
13 bool greater2(int a) { return a > 2; }
14 void test01() {
15     vector<int> v;
16     v.push_back(1);
17     v.push_back(3);
18     v.push_back(2);
19     v.push_back(5);
20     vector<int>::iterator it = find_if(v.begin(), v.end(), greater2);
21     if (it != v.end()) {
22         cout << *it << endl;
23     }
24     for_each(v.begin(), v.end(), Print());
25 }
26 int main() { test01(); }
27
28 3
29 1
30 3
31 2
32 5

```

### 三. 内建函数对象

- STL内建了一些函数对象,分为 : 算数类函数对象,关系运算类函数对象,逻辑运算类仿函数.这些仿函数所产生的对象,用法和一般函数完全相同
- 当然我们还可以产生无名的临时对象来履行函数功能
- 使用内建函数对象,需要引入头文件 `#include <functional>`

```

1 #include <algorithm>
2 #include <functional>
3 #include <iostream>
4 #include <stdlib.h>
5 #include <string.h>
6 #include <string>
7 #include <vector>
8 using namespace std;
9
10 /**
11 * @brief
12 *
13 * 6个算数类函数对象,除了negate是一元运算,其他都是二元运算
14 * template<class T> T plus<T> : 加法仿函数
15 * template<class T> T minus<T> : 减法仿函数
16 * template<class T> T multiplies<T> : 乘法仿函数
17 * template<class T> T divides<T> : 除法仿函数
18 * template<class T> T modulus<T> : 取模仿函数
19 * template<class T> T negate<T> : 取反仿函数
20 *
21 * 6个关系运算类函数对象,每一种都是二元运算
22 * template<class T> bool equal_to<T> : 等于
23 * template<class T> bool not_equal_to<T> : 不等于
24 * template<class T> bool greater<T> : 大于
25 * template<class T> bool greater_equal<T> : 大于等于

```

```

26 * template<class T> bool less<T> : 小于
27 * template<class T> bool less_equal<T> : 小于等于
28 *
29 * 逻辑运算类运算函数,not为一元运算,其余为二元运算
30 * template<class T> bool logical_and<T> : 逻辑与
31 * template<class T> bool logical_or<T> : 逻辑或
32 * template<class T> bool logical_not<T> : 逻辑非
33 */
34 void test01() {
35     negate<int> p;
36     cout << p(5) << endl;
37     cout << negate<int>()(2) << endl;
38 }
39 void test02() { cout << plus<int>()(2, 3) << endl; }
40 class Print {
41 public:
42     void operator()(int a) { cout << a << endl; }
43 };
44 void test03() {
45     vector<int> v;
46     v.push_back(1);
47     v.push_back(3);
48     v.push_back(2);
49     v.push_back(5);
50     sort(v.begin(), v.end(), greater<int>());
51     for_each(v.begin(), v.end(), Print());
52     sort(v.begin(), v.end(), less<int>());
53     for_each(v.begin(), v.end(), Print());
54 }
55 int main() {
56     test01();
57     test02();
58     test03();
59 }
60
61 -5
62 -2
63 5
64 5
65 3
66 2
67 1
68 1
69 2
70 3
71 5

```

## 四. 适配器

用来适配参数,扩展参数接口,一般结合仿函数一起使用

# 1. 函数对象适配器

```
1 #include <algorithm>
2 #include <functional>
3 #include <iostream>
4 #include <stdlib.h>
5 #include <string.h>
6 #include <string>
7 #include <vector>
8 using namespace std;
9
10 /**
11 * @brief
12 * 一元继承 : public unary_function<参数1, 返回值类型>
13 * 二元继承 : public binary_function<int, int, void>
14 */
15 class Print : public binary_function<int, int, void> {
16 public:
17     void operator()(int a, int num) const {
18         cout << a << " " << num << endl;
19         cout << a + num << endl;
20     }
21 };
22 /**
23 * @brief 绑定参数
24 * bind1st bind2nd
25 *
26 * bind1st : 将参数绑定为函数对象的第一个参数
27 * bind2nd : 将参数绑定为函数对象的第二个参数
28 */
29 void test01() {
30     vector<int> v;
31     v.push_back(1);
32     v.push_back(3);
33     v.push_back(2);
34     v.push_back(5);
35     for_each(v.begin(), v.end(), bind2nd(Print(), 200));
36 }
37
38 class GreaterThanFive : public unary_function<int, bool> {
39 public:
40     bool operator()(int v) const { return v > 5; }
41 };
42
43 /**
44 * @brief 取反适配器
45 *
46 * not1 : 对一元函数对象取反
47 * not2 : 对二元函数对象取反
48 */
49 void test02() {
50     vector<int> v;
51     for (int i = 0; i < 10; i++) {
52         v.push_back(i);
53     }
54     // 返回第一个大于5的迭代器
```

```

55 // vector<int>::iterator it = find_if(v.begin(), v.end(),
56 // GreaterThenFive());
57 // 返回第一个小于5的迭代器
58 // vector<int>::iterator it = find_if(v.begin(), v.end(),
59 // not1(GreaterThenFive())); 自定义输入
60 vector<int>::iterator it =
61     find_if(v.begin(), v.end(), not1(bind2nd(greater<int>(), 5)));
62 if (it == v.end()) {
63     cout << "没找到" << endl;
64 } else {
65     cout << "找到 " << *it << endl;
66 }
67 // 排序 二元函数对象
68 sort(v.begin(), v.end(), not2(less<int>()));
69 }

70 void MyPrint03(int v, int v2) { cout << v + v2 << " " ; }
71 /**
72 * @brief 函数指针适配器 prt_fun
73 *
74 * ptr_fun() : 把一个普通的函数指针适配成函数对象
75 */
76 void test03() {
77     vector<int> v;
78     for (int i = 0; i < 10; i++) {
79         v.push_back(i);
80     }
81     for_each(v.begin(), v.end(), bind2nd(ptr_fun(MyPrint03), 100));
82 }
83

84 class Person {
85 public:
86     Person(string name, int age) {
87         m_Name = name;
88         m_Age = age;
89     }
90     void ShowPerson() {
91         cout << "成员函数: "
92             << "Name: " << m_Name << " Age: " << m_Age << endl;
93     }
94     void Plus100() { m_Age += 100; }

95

96 public:
97     string m_Name;
98     int m_Age;
99 };
100 void MyPrint04(Person &p) {
101     cout << "姓名: " << p.m_Name << " 年龄: " << p.m_Age << endl;
102 }
103 /**
104 * @brief 成员函数适配器
105 *
106 * 如果容器中存放的是对象指针,那么用 mem_fun
107 * 如果容器中存放的是对象实体,那么用 mem_fun_ref
108 */
109 void test04() {
110     vector<Person> v;
111     Person p1("aaa", 10);

```

```
112 Person p2("bbb", 20);
113 Person p3("ccc", 30);
114 Person p4("ddd", 40);
115 v.push_back(p1);
116 v.push_back(p2);
117 v.push_back(p3);
118 v.push_back(p4);
119 // 利用 mem_fun_ref 将 Person 内部成员函数适配
120 for_each(v.begin(), v.end(), mem_fun_ref(&Person::ShowPerson));
121 }
122 void test05() {
123     vector<Person *> v;
124     Person p1("aaa", 10);
125     Person p2("bbb", 20);
126     Person p3("ccc", 30);
127     Person p4("ddd", 40);
128     v.push_back(&p1);
129     v.push_back(&p2);
130     v.push_back(&p3);
131     v.push_back(&p4);
132     for_each(v.begin(), v.end(), mem_fun(&Person::ShowPerson));
133 }
134
135 int main() {
136     test01();
137     cout << "======" << endl;
138     test02();
139     cout << "======" << endl;
140     test03();
141     cout << "======" << endl;
142     test04();
143     cout << "======" << endl;
144     test05();
145 }
146
147 1 200
148 201
149 3 200
150 203
151 2 200
152 202
153 5 200
154 205
155 =====
156 找到 0
157 =====
158 100 101 102 103 104 105 106 107 108 109 =====
159 成员函数: Name: aaa Age: 10
160 成员函数: Name: bbb Age: 20
161 成员函数: Name: ccc Age: 30
162 成员函数: Name: ddd Age: 40
163 =====
164 成员函数: Name: aaa Age: 10
165 成员函数: Name: bbb Age: 20
166 成员函数: Name: ccc Age: 30
167 成员函数: Name: ddd Age: 40
```

## 五. 算法概述

- 算法主要是由头文件“algorithm”组成的
- 是所有STL头文件中最大的一个,其中常用的功能涉及到比较、交换、查找、遍历、复制、修改、反转、排序、合并等
- 体积很小,只包括在几个序列容器上进行的简单运算的模板函数
- 定义了一些模板类,用以声明函数对象

### 1. 常用遍历算法

#### I. for\_each 遍历算法

```
1 /**
2 * @brief 遍历算法 遍历容器元素
3 *
4 * @param beg 开始迭代器
5 * @param end 结束迭代器
6 * @param _callback 函数回调或者函数对象
7 * @return 函数对象
8 */
9 for_each(iterator beg, iterator end, _callback);
```

#### II. transform 算法

```
1 /**
2 * @brief 将指定容器区间元素搬运到另一容器中
3 *        transform 不会给目标容器分配内存,所以需要我们提前分配好内存
4 * @param beg1 源容器开始迭代器
5 * @param end1 源容器结束迭代器
6 * @param beg2 目标容器开始迭代器
7 * @param _callback 回到函数或者函数对象
8 * @return 返回目标容器迭代器
9 */
10 transform(iterator beg1, iterator end1, iterator beg2, _callback);
```

### 2. 常用查找算法

```
1 /**
2 * @brief find 算法 查找元素
3 *
4 * @param beg 容器开始迭代器
5 * @param end 容器结束迭代器
6 * @param value 查找的元素
7 * @return 返回查找元素的位置
8 */
9 find(iterator beg, iterator end, value);
10
11 /**
12 * @brief find_if 算法 条件查找
13 *
14 * @param beg 容器开始迭代器
15 * @param end 容器结束迭代器
16 * @param callback 回调函数或者谓词(返回bool类型的函数对象)
17 * @return bool 查找返回true,否则false
```

```

18 */
19 find_if(iterator beg, iterator end, _callback);
20
21 /**
22 * @brief adjacent_find 算法 查找相邻重复元素
23 *
24 * @param beg 容器开始迭代器
25 * @param end 容器结束迭代器
26 * @param _callback 回调函数或者谓词(返回bool类型的函数对象)
27 * @return 返回相邻元素的第一个位置的迭代器
28 */
29 adjacent_find(iterator beg, iterator end, _callback);
30
31 /**
32 * @brief binary_search 算法 二分查找法
33 * 在无序序列中不可用
34 * @param beg 容器开始迭代器
35 * @param end 容器结束迭代器
36 * @param value 查找的元素
37 * @return bool 查找返回true,否则返回false
38 */
39 bool binary_search(iterator beg, iterator end, value);
40
41 /**
42 * @brief count 算法 统计元素出现次数
43 *
44 * @param beg 容器开始迭代器
45 * @param end 容器结束迭代器
46 * @param value 查找的元素
47 * @return int 返回元素个数
48 */
49 count(iterator beg, iterator end, value);
50
51 /**
52 * @brief count_if 算法 统计元素出现次数
53 *
54 * @param beg 容器开始迭代器
55 * @param end 容器结束迭代器
56 * @param callback 回调函数或者谓词(返回bool类型的函数对象)
57 * @return int 返回元素个数
58 */
59 count_if(iterator beg, iterator end, _callback);

```

### 3. 常用排序算法

```

1 /**
2 * @brief merge 算法 容器元素合并,并存储到另一容器
3 * 两个容器必须是有序的
4 * @param beg1 容器1 开始迭代器
5 * @param end1 容器1 结束迭代器
6 * @param beg2 容器2 开始迭代器
7 * @param end2 容器2 结束迭代器
8 * @param dest 目标容器开始迭代器
9 */
10 merge(iterator beg1,iterator end1,iterator beg2,iterator end2,iterator
dest);
11

```

```

12 /**
13 * @brief sort 算法 容器元素排序
14 *
15 * @param beg 容器1 开始迭代器
16 * @param end 容器1 结束迭代器
17 * @param _callback 回调函数或者谓词(返回bool类型的函数对象)
18 */
19 sort(iterator beg,iterator end,_callback);

20 /**
21 * @brief random_shuffle 算法 对指定范围内的元素随机调整次序
22 *
23 * @param beg 容器开始迭代器
24 * @param end 容器结束迭代器
25 */
26 random_shuffle(iterator beg,iterator end);

27 /**
28 * @brief reverse 算法 反转指定范围的元素
29 *
30 * @param beg 容器开始迭代器
31 * @param end 容器结束迭代器
32 */
33 reverse(iterator beg,iterator end);

```

## 4. 常用拷贝和替换算法

```

1 /**
2 * @brief copy 算法 将容器内指定范围的元素拷贝到另一容器中
3 *
4 * @param beg 容器开始迭代器
5 * @param end 容器结束迭代器
6 * @param dest 目标起始迭代器
7 */
8 copy(iterator beg,iterator end,iterator dest);

9 /**
10 * @brief replace 算法 将容器内指定范围的旧元素修改为新元素
11 *
12 * @param beg 容器开始迭代器
13 * @param end 容器结束迭代器
14 * @param oldValue 旧元素
15 * @param newValue 新元素
16 */
17 replace(iterator beg,iterator end,oldValue,newValue);

18 /**
19 * @brief replace_if 算法 将容器内指定范围满足条件的元素替换为新元素
20 *
21 * @param beg 容器开始迭代器
22 * @param end 容器结束迭代器
23 * @param _callback 函数回调或者谓词(返回bool类型的函数对象)
24 * @param newValue 新元素
25 */
26 replace_if(iterator beg,iterator end,_callback,newValue);

27 /**
28 * @brief
29 */
30 /**

```

```
31 * @brief swap 算法 互换两个容器的元素
32 *
33 * @param c1 容器1
34 * @param c2 容器2
35 */
36 swap(container c1,container c2);
```

## 5. 常用算数生成算法

```
1 /**
2  * @brief accumulate 算法 计算容器元素累积总和
3 *
4 * @param beg 容器开始迭代器
5 * @param end 容器结束迭代器
6 * @param value 累加值
7 */
8 accumulate(iterator beg,iterator end,value);
9
10 /**
11 * @brief fill 算法 向容器中添加元素
12 *
13 * @param beg 容器开始迭代器
14 * @param end 容器结束迭代器
15 * @param value 填充元素
16 */
17 fill(iterator beg,iterator end,value);
```

## 6. 常用集合算法

```
1 /**
2  * @brief set_intersection 算法 求两个 set 集合的交集
3 * 两个集合必须是有序序列
4 * @param beg1 容器1 开始迭代器
5 * @param end1 容器1 结束迭代器
6 * @param beg2 容器2 开始迭代器
7 * @param end2 容器2 结束迭代器
8 * @param dest 目标容器开始迭代器
9 * @return 目标容器的最后一个元素的迭代器地址
10 */
11 set_intersection(iterator beg1, iterator end1, iterator beg2, iterator end2,
12                  iterator dest);
13
14 /**
15 * @brief set_union 算法 求两个 set 集合的并集
16 * 两个集合必须是有序序列
17 * @param beg1 容器1 开始迭代器
18 * @param end1 容器1 结束迭代器
19 * @param beg2 容器2 开始迭代器
20 * @param end2 容器2 结束迭代器
21 * @param dest 目标容器开始迭代器
22 * @return 目标容器的最后一个元素的迭代器地址
23 */
24 set_union(iterator beg1, iterator end1, iterator beg2, iterator end2,
25           iterator dest);
26
27 /**
```

```
28 * @brief set_difference 算法 求两个 set 集合的差集
29 *
30 * @param beg1 容器1 开始迭代器
31 * @param end1 容器1 结束迭代器
32 * @param beg2 容器2 开始迭代器
33 * @param end2 容器2 结束迭代器
34 * @param dest 目标容器开始迭代器
35 * @return 目标容器的最后一个元素的迭代器地址
36 */
37 set_difference(iterator beg1, iterator end1, iterator beg2, iterator end2,
38                 iterator dest);
```