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

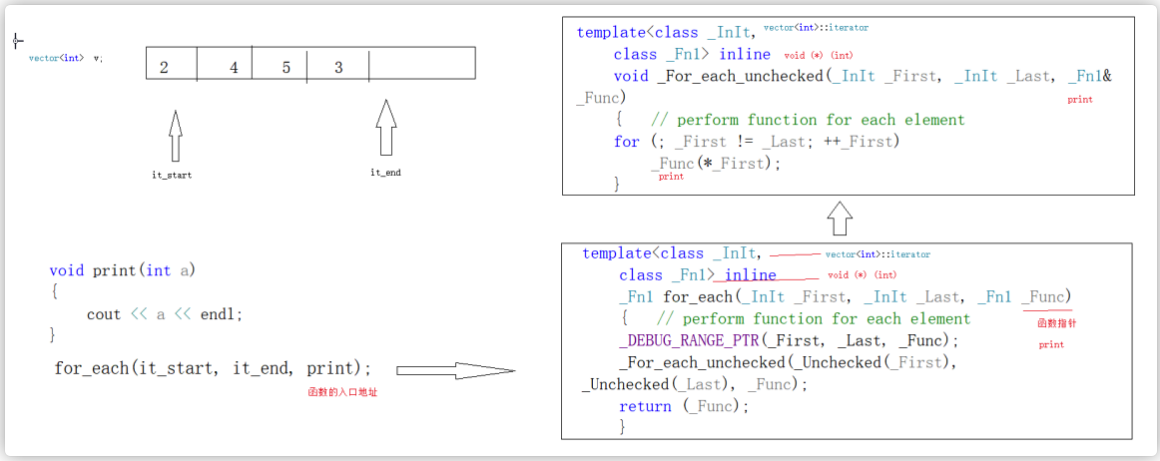
- **迭代器是一种抽象的设计概念 : 提供一种方法,使之能够依序寻访某个容器所含的各个元素,而又无需暴露该容器的内部表示方式**

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

```cpp
#include <iostream>
#include <stdlib.h>
#include <string.h>
#include <string>
#include <vector>
using namespace std;

void test01() {
  vector<int> v;
  v.push_back(2);
  v.push_back(4);
  v.push_back(5);
  v.push_back(3);
  // 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)
  vector<int>::iterator it_start = v.begin();
  vector<int>::iterator it_end = v.end();
  for (; it_start != it_end; it_start++) {
    cout << *it_start << " ";
  }
  cout << endl;
}
int main() { test01(); }

// 2 4 5 3
```

2. 迭代器通过 for_each 遍历




```cpp
#include <iostream>
#include <stdlib.h>
#include <string.h>
#include <string>
#include <vector>
using namespace std;
void print(int a) { cout << a << endl; }
void test01() {
  vector<int> v;
  v.push_back(2);
  v.push_back(4);
  v.push_back(5);
```

```
  v.push_back(3);
  // 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)
  vector<int>::iterator it_start = v.begin();
  vector<int>::iterator it_end = v.end();
  //   for (; it_start != it_end; it_start++) {
  //     cout << *it_start << " ";
  //   }
  //   cout << endl;
  for_each(it_start, it_end, print);
}
int main() { test01(); }

// 2
// 4
// 5
// 3
```

3. 迭代器指向的元素是自定义的数据类型

```
#include <iostream>
#include <stdlib.h>
#include <string.h>
#include <string>
#include <vector>
using namespace std;
class Person {
public:
  Person(int age) { this->age = age; }
  int age;
};
void print_person(Person &p) { cout << p.age << endl; }
void test() {
  Person p1(1);
  Person p2(2);
  Person p3(3);
  Person p4(4);
  vector<Person> v;
  v.push_back(p1);
  v.push_back(p2);
  v.push_back(p3);
  v.push_back(p4);
  vector<Person>::iterator it_start = v.begin();
  vector<Person>::iterator it_end = v.end();
  for (; it_start != it_end; it_start++) {
    print_person(*it_start);
  }
}
int main() { test(); }

// 1
// 2
// 3
// 4
```

4. 迭代器指向的是一个容器

```cpp
#include <iostream>
#include <stdlib.h>
#include <string.h>
#include <string>
#include <vector>
using namespace std;
void test() {
  vector< vector<int> > v;
  vector<int> v1;
  vector<int> v2;
  vector<int> v3;
  for (int i = 0; i < 3; i++) {
    v1.push_back(i);
    v2.push_back(i + 10);
    v3.push_back(i + 100);
  }
  v.push_back(v1);
  v.push_back(v2);
  v.push_back(v3);
  vector< vector<int> >::iterator it = v.begin();
  for (; it != v.end(); it++) {
    // *it 得到的是 vector<int>
    vector<int>::iterator it_start = (*it).begin();
    vector<int>::iterator it_end = (*it).end();
    for (; it_start != it_end; it_start++) {
      cout << *it_start << endl;
    }
  }
}
int main() { test(); }

// 0
// 1
// 2
// 10
// 11
// 12
// 100
// 101
// 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

```
/**
 * @brief string 的构造函数
 *
 * string() : 创建一个空的字符串
 * string(const string& str) : 使用一个 string 对象初始化另一个 string 对象
 * string(const char *s) : 使用字符串 s 初始化
 * string(int n,char c) : 使用 n 个字符 c 初始化 v
 */
void test01() {
  string str;
  string str1("hello");
  string str2(str1);
  string str3(5, 'k');
  cout << str1 << endl;
  cout << str2 << endl;
  cout << str3 << endl;
}

// hello
// hello
// kkkkk
```

II. string 容器的基本操作

```
/**
 * @brief string 容器的基本操作
 *
 * string &operator=(const char* s) : char * 类型的字符串,赋值给当前的字符串
 * string &operator=(const string &s) : 把字符串 s 赋给当前的字符串
 * 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个字符赋给当前的字符串
 * string &assign(const string &s) : 把字符串 s 赋给当前字符串
 * string &assing(int n,char c) : 用 n 个字符 c 赋给当前字符串
 * string &assign(const string &s,int start,int n) : 将 s 从 start 开始 n
 * 个字符赋值给字符串
 *
 */
void test02() {
  string str("HelloWorld");
  string str1("Heihei");
  cout << str << endl;
  str = str1;
  cout << str << endl;
  str = "hehe";
  cout << str << endl;
  str = 'c';
  cout << str << endl;
  str.assign(str1);
  cout << str << endl;
  str.assign("hehe");
  cout << str << endl;
```

```
  str.assign("jack", 2);
  cout << str << endl;
  str.assign(5, 'c');
  cout << str << endl;
  str.assign(str1, 2, 3);
  cout << str << endl;
}

// HelloWorld
// Heihei
// hehe
// c
// Heihei
// hehe
// ja
// ccccc
// ihe
```

III. String 的存取字符操作

```
/**
 * @brief string 存取字符操作
 *
 * char &operator[](int n) : 通过[]方式取字符
 * char & at(int n) : 通过 at 方法获取字符
 */
void test03() {
  string str("HelloWorld");
  cout << str[4] << endl;
  str[4] = 'c';
  cout << str.at(4) << endl;
}

// o
// c
```

IV. string 的拼接操作

```
/**
 * @brief string 的拼接操作
 *
 * string &operator+=(const string& str) : 重载 += 操作符
 * string &operator+=(const char* str) : 重载 += 操作符
 * string &operator+=(const char c) : 重载 += 操作符
 * string &append(const char *s) : 把字符串 s 连接到当前字符串结尾
 * 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
 * string &append(const string &s) : 同 operator+=()
 * 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
 * string &append(int n,char c) : 在当前字符串结尾添加n个字符c
 */
void test04(){
    string str1("HelloWorld");
    string str2("123456");
    cout << str1 << endl;
```

```
    str1 += str2;
    cout << str1 << endl;
    str1 += 'c';
    cout << str1 << endl;
    str1.append(str2);
    cout << str1 << endl;
    str1.append("hehe");
    cout << str1 << endl;
    str1.append("hehe",2);
    cout << str1 << endl;
    str1.append(str2,2,3);
    cout << str1 << endl;
    str1.append(5,'c');
    cout << str1 << endl;
}

// HelloWorld
// HelloWorld123456
// HelloWorld123456c
// HelloWorld123456c123456
// HelloWorld123456c123456hehe
// HelloWorld123456c123456hehehe
// HelloWorld123456c123456hehehe345
// HelloWorld123456c123456hehehe345ccccc
```

V. string 容器的查找和替换

```
/**
 * @brief string 容器的查找和替换
 *
 * int find(const string& str,int pos = 0) const : 查找 str 第一次出现的位置,从 pos 开始查找
 * int find(const char* s,int pos = 0) const : 查找 s 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 s 的前n个字符第一次位置
 * int find(const char c,int pos = 0) const : 查找字符 c 第一次出现的位置
 * int rfind(const string& str,int pos = npos) const : 查找 str 最后一次位置,从 pos 开始查找
 * int rfind(const char* s,int pos = npos) const : 查找 s 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 s 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n个字符为字符串 str
 * 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
 */
void test05(){
    string str("helrlowolrd");
    string str1("wol");
    cout << str.find(str1) << endl;
    cout << str.find("wol") << endl;
    cout << str.find("world",0,2) << endl;
    cout << str.find('o') << endl;
    cout << str.rfind("lr") << endl;
```

```
    str.replace(2,4,str1);
    cout << str << endl;
    str.replace(2,2,"123456");
    cout << str << endl;
}

// 6
// 6
// 6
// 5
// 8
// hewolwolrd
// he123456lwolrd
```

VI. string 的比较

```
/**
 * @brief string 的比较
 *
 * compare函数在 > 时返回1,在 < 时返回 -1,在 == 时返回 0
 * 比较区分大小写,比较时参考字典顺序,排越前面的越小
 * 大写的A比小写的a小
 * 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
 */
void test06() {
  string str1("Hello");
  string str2("World");
  // int ret = str1.compare(str2);
  int ret = str1.compare("hello");
  if (ret > 0) {
    cout << "str大" << endl;
  } else if (ret < 0) {
    cout << "str小" << endl;
  } else {
    cout << "相等" << endl;
  }
}

// str小
```

VII. 获取 string 的子串

```
/**
 * @brief 获取 string 的子串
 *
 * 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
 */
void test07() {
  string str1("HelloWorld");
  string str2 = str1.substr(4, 3);
  cout << str2 << endl;
}

// oWo
```

VIII. string 插入和删除

```
/**
 * @brief string 插入和删除操作
 *
 * 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
 */
void test08() {
  string str1("helloworld");
  string str2("hehe");
  str1.insert(2, "kkk");
  cout << str1 << endl;
  str1.insert(2, str2);
  cout << str1 << endl;
  str1.insert(2, 10, 'r');
  cout << str1 << endl;
  str1.erase(3, 3);
  cout << str1 << endl;
}

hekkkklloworld
hehehekkkklloworld
herrrrrrrrrrhehekkkklloworld
herrrrrrrhehekkkklloworld
```

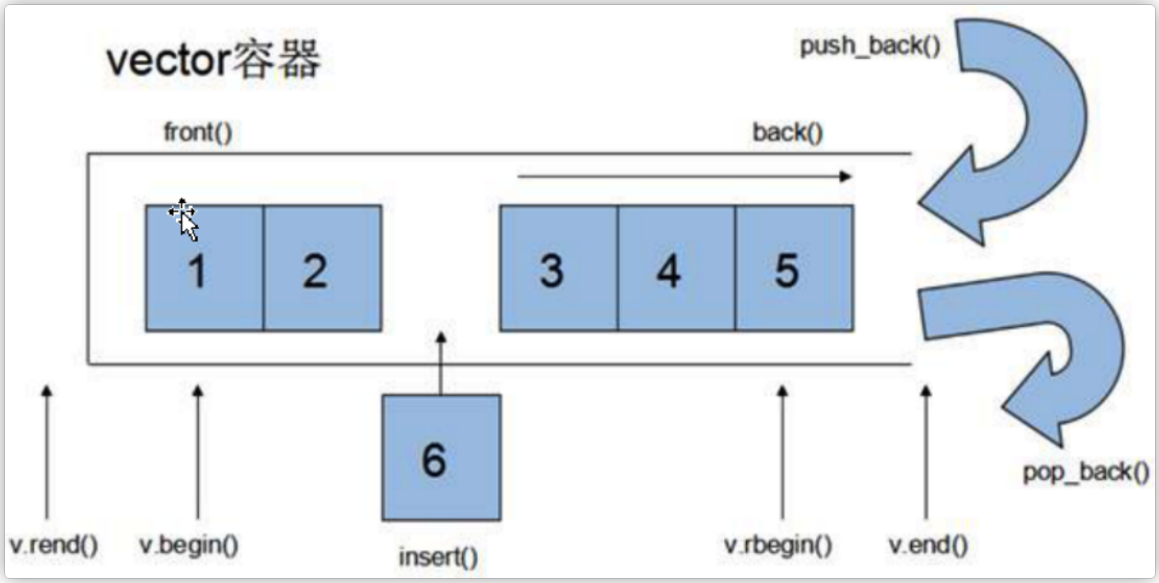
IX. string 对象和 char* 相互转换

```
void test09() {
  string str("hello");
  str = "world"; // str.operator=(char *)

  char *s = NULL;
  // str.c_str(); const char *
  s = const_cast<char *>(str.c_str());
  cout << s << endl;
}

// 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**



2. vector 容器的迭代器

```cpp
#include <iostream>
#include <stdlib.h>
#include <string.h>
#include <string>
#include <vector>
using namespace std;
/**
 * @brief vector 迭代器
 *
 */
void test01() {
  vector<int> v;
  v.push_back(1);
  v.push_back(2);
  v.push_back(3);
  v.push_back(4);
  // 从头开始 1 2 3 4
  // vector<int>::iterator it_start = v.begin();
  // vector<int>::iterator it_end = v.end();
  // 从尾开始 4 3 2 1
  vector<int>::reverse_iterator it_start = v.rbegin();
  vector<int>::reverse_iterator it_end = v.rend();
  for (; it_start != it_end; it_start++) {
    cout << *it_start << " ";
  }
  cout << endl;
}
/**
 * @brief 空间配置
 *
 */
void test02() {
  vector<int> v;
  for (int i = 0; i < 100; i++) {
    v.push_back(i);
    // v.capacity() : 容器的容量
    cout << v.capacity() << endl;
```

```
  }
}
int main() {
  test01();
  test02();
}
```

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被使用的范围,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

```
/**
 * @brief vector 的构造函数
 *
 * vector<T> v : 采用模板实现类实现,默认构造函数
 * vector(v.begin(),v.end()) : 将 v[begin(),end()] 区间中的元素拷贝给本身
 * 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
 */
void print(int a) { cout << a << endl; }
void test03() {
  vector<int> v1;
  v1.push_back(1);
  v1.push_back(2);
  v1.push_back(3);
  v1.push_back(4);
  // vector<int> v2(v1) == vector<int> v2(v1.begin(), v1.end())
  // vector<int> v2(v1);
  // vector<int> v2(v1.begin(), v1.end());
  vector<int> v2(v1.begin() + 1, v1.end());
  for_each(v2.begin(), v2.end(), print);
}

// 2
// 3
// 4
```

II. vector 常用赋值操作

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

```
 * swap(vec) : 将vec与本身的元素互换
 */
void test04() {
  vector<int> v1;
  v1.push_back(1);
  v1.push_back(2);
  v1.push_back(3);
  v1.push_back(4);

  vector<int> v2;
  v2.push_back(5);
  v2.push_back(6);
  v2.push_back(7);
  v2.push_back(8);

  // v2.assign(v1.begin(), v1.end()) == v2 = v1
  // v2.assign(v1.begin(), v1.end());
  // v2 = v1;
  // v2.swap(v1);
  v2.assign(10, 5);
  for_each(v2.begin(), v2.end(), print);
}
```

III. vector 大小操作

```
/**
 * @brief vector 大小操作
 *
 * size() : 返回容器中元素的个数
 * empty() : 判断容器是否为空
 * 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
 *                   如果容器变短,则末尾超出容器长度的元素被删除
 * 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
 *                         如果容器变短,则末尾超出容器长度的元素被删除
 * capacity() : 容器的容量
 * 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
 */
void test05() {
  vector<int> v1;
  v1.push_back(1);
  v1.push_back(2);
  v1.push_back(3);
  v1.push_back(4);
  v1.push_back(5);
  cout << v1.size() << endl;
  cout << v1.capacity() << endl;
  v1.resize(10, 6);
  if (!v1.empty()) {
    cout << "不为空" << endl;
  }
  for_each(v1.begin(), v1.end(), print);

  vector<int> v2;
  // 设置容量
  v2.reserve(1000);
  cout << v2.size() << endl;
```

```
  cout << v2.capacity() << endl;
}

// 5
// 8
// 不为空
// 1
// 2
// 3
// 4
// 5
// 6
// 6
// 6
// 6
// 6
// 0
// 1000
```

IV. vector 容器的存取操作

```
/**
 * @brief vector 容器的存取操作
 *
 * 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
 * operator[] : 返回索引 idx 所指的数据,越界时,运行直接报错
 * 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
 */
void test06() {
  vector<int> v1;
  v1.push_back(1);
  v1.push_back(2);
  v1.push_back(3);
  v1.push_back(4);
  v1.push_back(5);
  cout << v1.at(2) << endl;
  cout << v1[2] << endl;
  cout << v1.front() << endl;
  cout << v1.back() << endl;
}

// 3
// 3
// 1
// 5
```

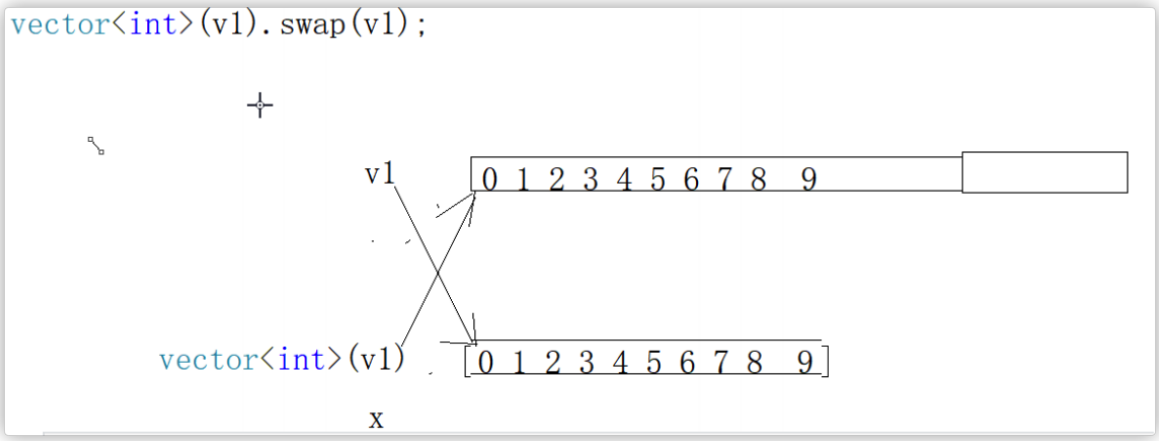
V. vector 容器插入和删除操作

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

```
 * pop_back() : 删除最后一个元素
 * erase(const_iterator start,const_iterator end) : 删除迭代器从 start 到 end
 *                                                  之间的元素
 * erase(const_iterator pos) : 删除迭代器指向的元素
 * clear() : 删除容器中所有元素
 */
void test07() {
  vector<int> v1;
  v1.push_back(1);
  v1.push_back(2);
  v1.push_back(3);
  v1.push_back(4);
  v1.push_back(5);
  // v1.insert(v1.begin() + 2, 3, 9);
  // v1.pop_back();
  // v1.erase(v1.begin()+3,v1.end());
  // v1.erase(v1.begin() + 3);
  v1.clear();
  for_each(v1.begin(), v1.end(), print);
}
```

5. vector 容器的案例

I. 使用 swap 收缩空间




```
/**
 * @brief swap 收缩空间
 *
 */
void test08() {
  vector<int> v1;
  for (int i = 0; i < 10000; i++) {
    v1.push_back(i);
  }
  cout << v1.size() << " " << v1.capacity() << endl;
  v1.resize(10);
  cout << v1.size() << " " << v1.capacity() << endl;
  vector<int>(v1).swap(v1);
  cout << v1.size() << " " << v1.capacity() << endl;
}

// 10000 16384
// 10 16384
```

```
// 10 10
```

II. 计算重新开辟了多少次内存空间

```
/**
 * @brief 计算重新开辟了多少次内存空间
 *
 */
void test09() {
  vector<int> v;
  int *p = NULL;
  int count = 0;
  for (int i = 0; i < 10000; i++) {
    v.push_back(i);
    if (p != &v[0]) {
      count++;
      p = &v[0];
    }
  }
  cout << count << endl;
}

// 15
```

6. Vector 容器排序

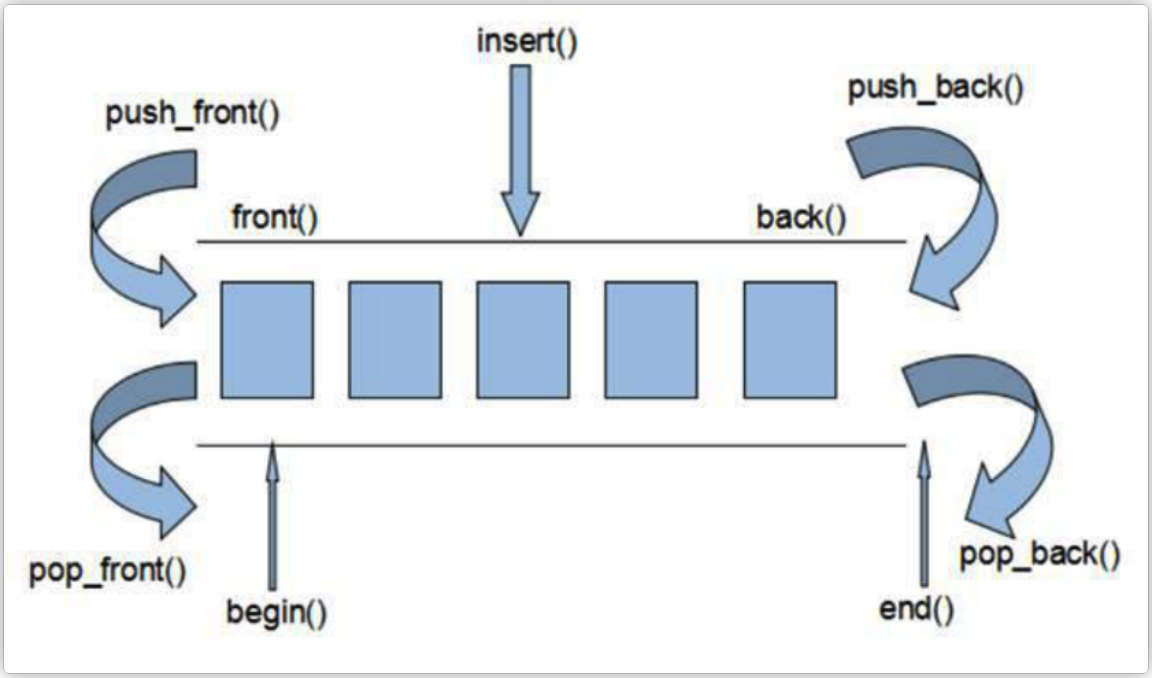
```
/**
 * @brief vector 容器培训
 *
 */
bool compare(int a, int b) { return a > b; }
void test10() {
  vector<int> v1;
  v1.push_back(1);
  v1.push_back(3);
  v1.push_back(2);
  v1.push_back(5);
  v1.push_back(4);
  // sort(v1.begin(), v1.end()); 默认是从小到大
  sort(v1.begin(), v1.end(), compare);
  for_each(v1.begin(), v1.end(), print);
}

// 5
// 4
// 3
// 2
// 1

class Person {
public:
  Person(int age, string s) {
    this->age = age;
    this->name = s;
  }
  int age;
```

```
  string name;
};
void print_person(Person &p) { cout << p.age << " " << p.name << endl; }
bool compare_person(Person &p1, Person &p2) { return p1.age > p2.age; }
void test11() {
  vector<Person> v;
  v.push_back(Person(10, "安琪拉"));
  v.push_back(Person(40, "韩信"));
  v.push_back(Person(50, "盖伦"));
  v.push_back(Person(20, "猴子"));
  v.push_back(Person(30, "三眼怪"));

  for_each(v.begin(), v.end(), print_person);
  sort(v.begin(), v.end(), compare_person);
  for_each(v.begin(), v.end(), print_person);
}

// 10 安琪拉
// 40 韩信
// 50 盖伦
// 20 猴子
// 30 三眼怪
// 50 盖伦
// 40 韩信
// 30 三眼怪
// 20 猴子
// 10 安琪拉
```

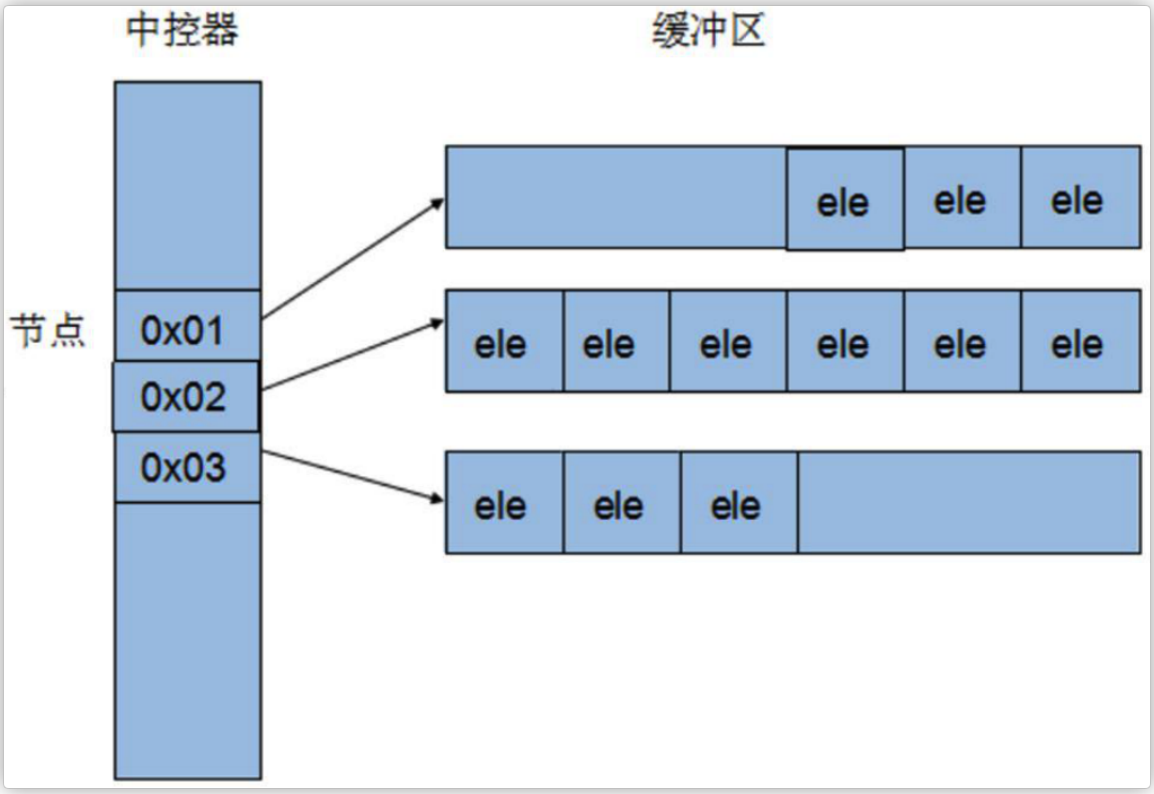
三. deque 容器

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

```cpp
#include <deque>
#include <iostream>
#include <stdlib.h>
#include <string.h>
#include <string>
using namespace std;

/**
 * @brief deque 的构造函数
 *
 * deque<T> deqT : 默认构造函数
 * 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
 */
void print(int a) { cout << a << endl; }
void test01() {
  deque<int> d1;
  d1.push_back(1);
  d1.push_back(2);
  d1.push_back(3);
  deque<int> d2(d1);
  for_each(d2.begin(), d2.end(), print);
  deque<int> d3(d1.begin(), d1.end());
  for_each(d3.begin(), d3.end(), print);
  deque<int> d4(5, 8);
  for_each(d4.begin(), d4.end(), print);
```

```
}
int main() { test01(); }

1
2
3
1
2
3
8
8
8
8
8
```

II. Deque 容器的赋值操作

```
/**
 * @brief deque 赋值操作
 *
 * 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
 * deque& operator=(const deque &deq) : 重载等号操作符
 * swap(deq) : 将deq与本身的元素互换
 */
void test02() {
  deque<int> d1;
  d1.push_back(1);
  d1.push_back(2);
  d1.push_back(3);
  deque<int> d2;
  d2.assign(d1.begin(), d1.end());
  for_each(d2.begin(), d2.end(), print);
  deque<int> d3;
  d3.assign(3, 9);
  for_each(d3.begin(), d3.end(), print);
  deque<int> d4;
  d4 = d3;
  for_each(d4.begin(), d4.end(), print);
  d3.swap(d2);
  for_each(d3.begin(), d3.end(), print);
}

1
2
3
9
9
9
9
9
9
1
2
3
```

III. Deque 容器的大小操作

```
/**
 * @brief deque 大小操作
 *
 * deque.size() : 返回容器中元素的个数
 * deque.empty() : 判断容器是否为空
 * 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
 *                     如果容器变短,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,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
 */
void test03() {
  deque<int> d1;
  d1.push_back(1);
  d1.push_back(2);
  d1.push_back(3);
  cout << d1.size() << endl;
  if (!d1.empty()) {
    cout << "不为空!" << endl;
  }
  d1.resize(2);
  for_each(d1.begin(), d1.end(), print);
  d1.resize(3,5);
  for_each(d1.begin(), d1.end(), print);
}

3
不为空!
1
2
1
2
5
```

IV. 双端的删除操作

```
/**
 * @brief deque 双端插入和删除操作
 *
 * 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
 */
void test04() {
  deque<int> d1;
  d1.push_back(1);
  d1.push_back(2);
  d1.push_back(3);
  d1.push_back(4);
  d1.push_back(5);
  d1.push_back(6);
  for_each(d1.begin(), d1.end(), print);
  d1.pop_back();
```

```
  d1.pop_front();
  for_each(d1.begin(), d1.end(), print);
}

1
2
3
4
5
6
2
3
4
5
```

V. deque 容器的存取操作

```
/**
 * @brief deque 容器的存取操作
 *
 * at(idx) : 返回索引idx所指的数据,如果idx越界,抛出 out_of_range
 * 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
 * front() : 返回第一个数据
 * back() : 返回最后一个数据
 */
void test05() {
  deque<int> d1;
  d1.push_back(1);
  d1.push_back(2);
  d1.push_back(3);
  cout << d1.at(2) << endl;
  d1[2] = 10;
  cout << d1.front() << endl;
  cout << d1.back() << endl;
}

3
1
10
```

VI. deque 容器的插入操作

```
/**
 * @brief deque插入操作
 *
 * 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
 * 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
 */
void test06() {
  deque<int> d1;
  d1.push_back(1);
  d1.push_back(2);
  d1.push_back(3);
  d1.insert(d1.begin(), 10);
  for_each(d1.begin(), d1.end(), print);
```

```
  d1.insert(d1.begin(), 3, 2);
  for_each(d1.begin(), d1.end(), print);
  deque<int> d2;
  d2.insert(d2.begin(), d1.begin(), d1.end());
}

10
1
2
3
2
2
2
10
1
2
3

```

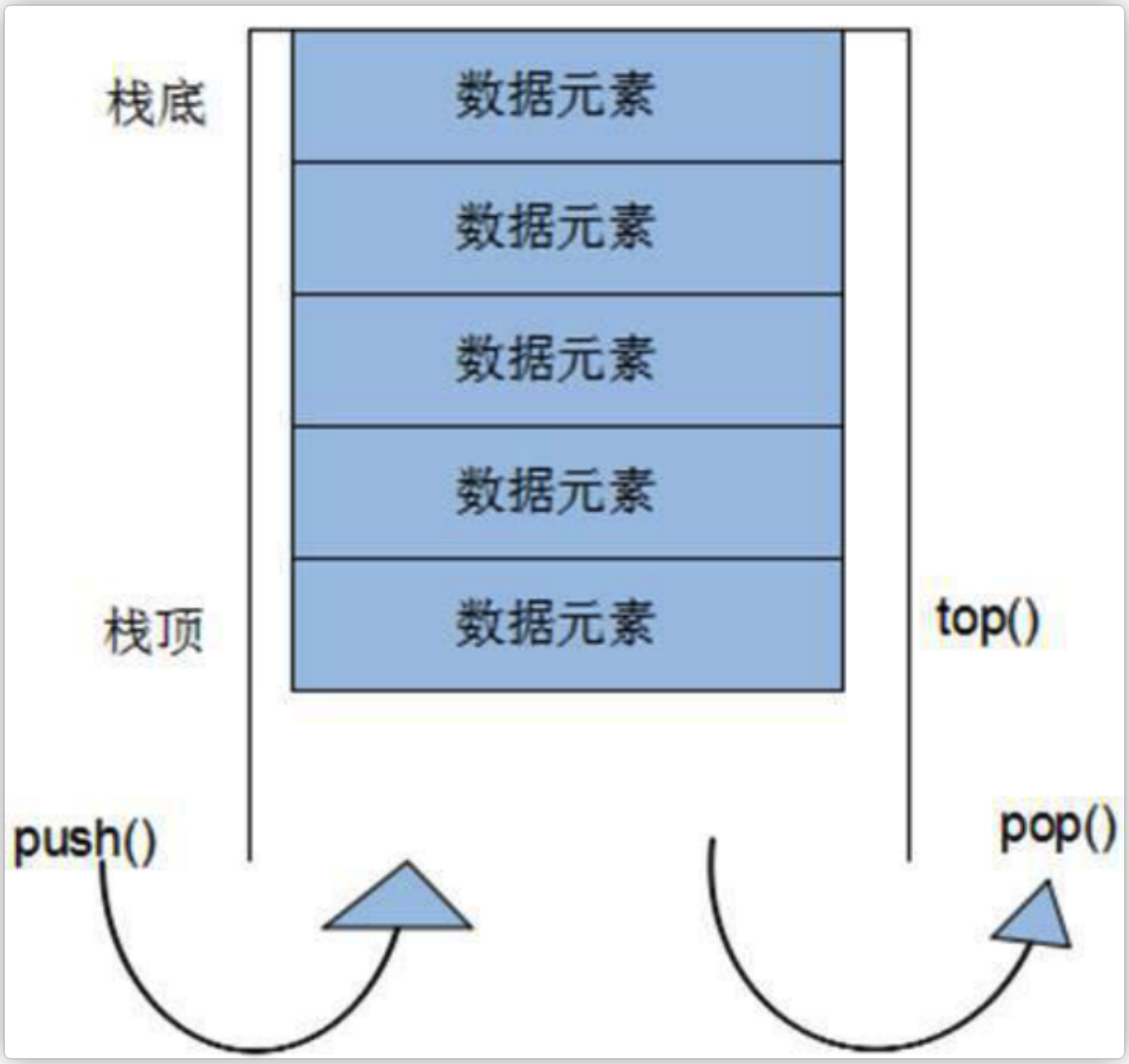
VII. deque 容器的删除操作

```
/**
 * @brief deque 删除操作
 *
 * 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
 */
void test07() {
  deque<int> d1;
  d1.push_back(1);
  d1.push_back(2);
  d1.push_back(3);
  // d1.clear();
  // for_each(d1.begin(), d1.end(), print);

  // d1.erase(d1.begin()+1,d1.end());
  d1.erase(d1.begin()+1);
  for_each(d1.begin(), d1.end(), print);
}

1
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

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

II. stack 赋值操作

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

III. stack 数据存取操作

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

IV. stack 大小操作

```
empty() : 判断堆栈是否为空
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

```
queue<T> queT : 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
```

II. queue 存取、插入和删除操作

```
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
```

III. queue 赋值操作

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

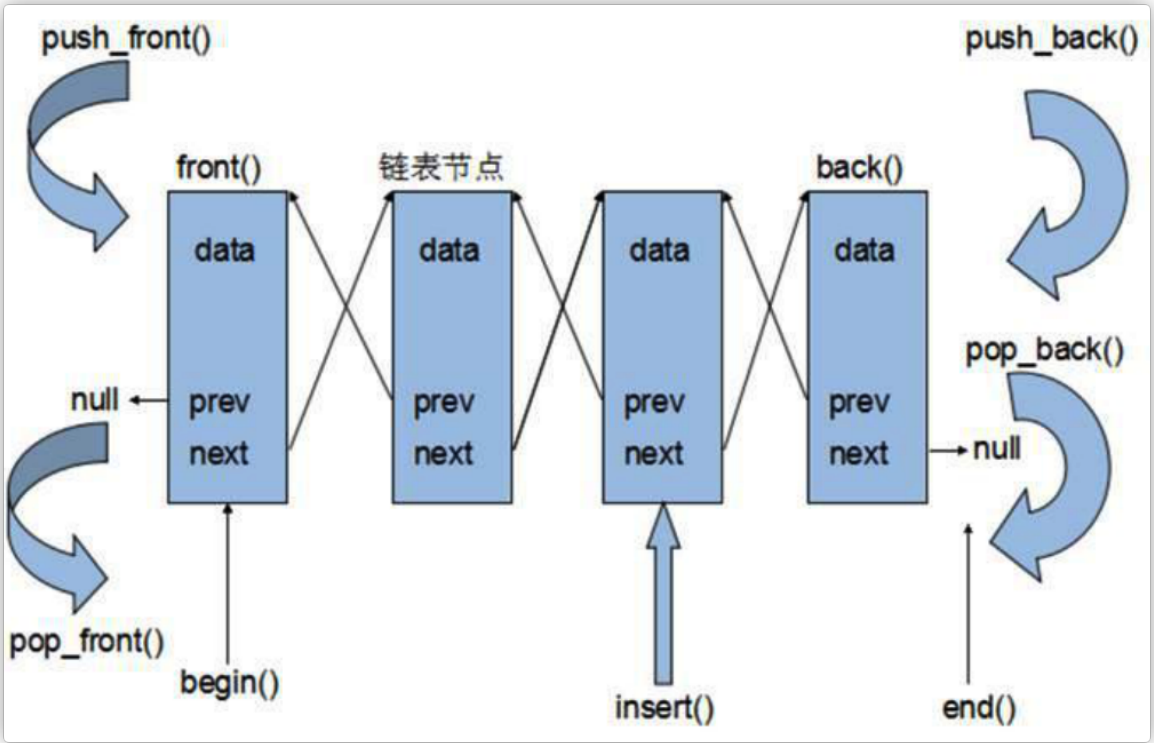
IV. queue 大小操作

```
empty() ： 判断队列是否为空
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

```
list<T> lstT : 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
```

II. list 数据元素插入和删除操作

```
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
```

III. list 大小操作

```
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
```

IV. list 赋值操作

```
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
```

V. list 数据的存取

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

VI. list 反转排序

```
reverse() : 反转链表
sort() : list排序
```

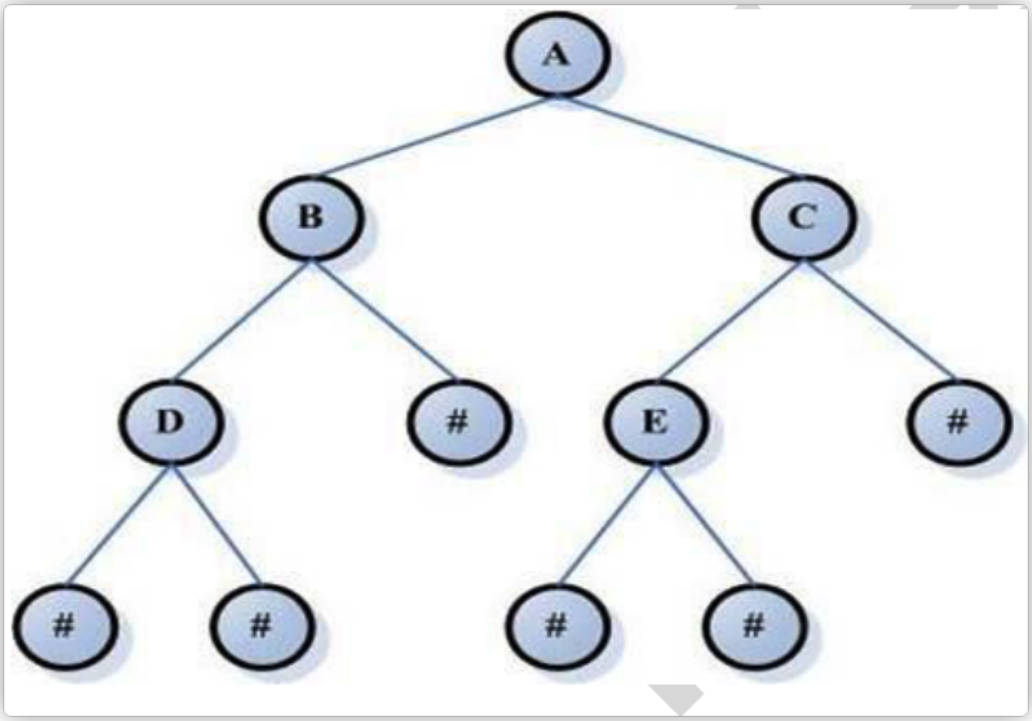
七. set/multiset 容器

1. set/multiset 容器基本概念

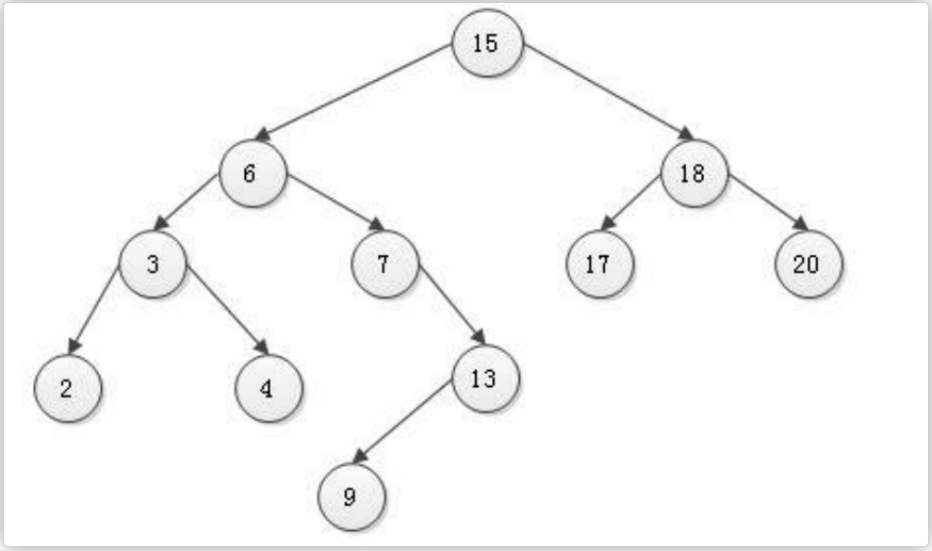
- **Set 的特性是 : 所有元素都会根据元素的键值自动被排序**
- **Set 的元素不像 map 那样可以同时拥有实值和键值,Set 的元素即使键值又是实值**
- **Set 不允许两个元素有相同的键值**
- **不能通过 Set 的迭代器改变 Set 元素的值,因为 Set 元素值就是其键值,关系到 Set 元素的排序规则.Set 的iterator是一种 const_iterator**
- **Set 拥有和 list 某些相同的性质,当对容器中的元素进行插入操作或者删除操作的时候,操作之前所有的迭代器,在操作完成之后依然有效,被删除的那个元素的迭代器必然是一个例外**
- **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

```
set<T> st : 默认构造函数
multiset<T> mst : multiset 默认构造函数
set(const set &st) : 拷贝构造函数
```

II. set 赋值操作

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

III. set 大小操作

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

IV. set 插入和删除操作

```
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(elem) : 删除容器中值为elem的元素
```

V. set 查找操作

```
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()
count(key) : 查找建key的元素个数
lower_bound(keyElem) : 返回第一个key>=keyElem元素的迭代器
upper_bound(keyElem) : 返回第一个key>keyElem元素的迭代器
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 secong 访问

```
类模板 : template <class T1,class T2> struct pair

第一种方法创建一个对组
pair<string,int> pair1(string("name"),20);
cout << pair1.first << endl; 访问pair第一个值
cout << pair1.second << endl; 访问pair第二个值

第二种
pair<string,int> pair2 = make_pair("name",20);
cout << pair2.first << endl;
cout << pari2.second << endl;

第三种=赋值
pari<string,int> pair3 = pair2;
cout << pair3.first << endl;
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

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

II. map 赋值操作

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

III. map 大小操作

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

IV. map 插入数据元素操作

```
map.insert(...) : 往容器中插入元素,返回 pair<iterator,bool>

map<int,string> mapStu;
// 第一种 通过pair的方式插入对象
mapStu.insert(pair<int,string>(3,"小张"));
// 第二种 通过pair的方式插入对象
mapStu.insert(make_pair(-1,"校长"));
// 第三种 通过value_type的方式插入对象
mapStu.insert(map<int,string>::value_type(1,"小李"));
// 第四种 通过数组的方式插入值
mapStu[3] = "小刘";
mapStu[5] = "小王";
```

V. map 删除操作

```
clear() : 删除所有元素
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(keyElem) : 删除容器中key为keyElem的对组
```

VI. map 查找操作

```
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()
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
lower_bound(keyElem) : 返回第一个key>=keyElem元素的迭代器
upper_bound(keyElem) : 返回第一个key>keyElem元素的迭代器
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)**

```
#include <algorithm>
#include <iostream>
#include <stdlib.h>
#include <string.h>
#include <string>
#include <vector>
using namespace std;

class MyAdd {
public:
  int operator()(int a, int b) { return a + b; }
};
void test01() {
  MyAdd p;
  cout << p(2, 3) << endl;
```

```
  cout << MyAdd()(4, 5) << endl;
}

void print(int a) { cout << a << endl; }
class Print {
public:
  void operator()(int a) { cout << a << endl; }
};

bool compare(int a, int b) { return a > b; }
class Compare {
public:
  bool operator()(int a, int b) { return a > b; }
};
void test02() {
  vector<int> v;
  v.push_back(1);
  v.push_back(2);
  v.push_back(4);
  v.push_back(5);
  sort(v.begin(), v.end(), Compare());
  for_each(v.begin(), v.end(), Print());
}
int main() {
  test01();
  test02();
}

5
9
5
4
2
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**

```
#include <algorithm>
#include <iostream>
#include <stdlib.h>
#include <string.h>
#include <string>
#include <vector>
using namespace std;

class Print {
```

```
public:
  void operator()(int a) { cout << a << endl; }
};
bool greater2(int a) { return a > 2; }
void test01() {
  vector<int> v;
  v.push_back(1);
  v.push_back(3);
  v.push_back(2);
  v.push_back(5);
  vector<int>::iterator it = find_if(v.begin(), v.end(), greater2);
  if (it != v.end()) {
    cout << *it << endl;
  }
  for_each(v.begin(), v.end(), Print());
}
int main() { test01(); }

3
1
3
2
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>`

```
#include <algorithm>
#include <functional>
#include <iostream>
#include <stdlib.h>
#include <string.h>
#include <string>
#include <vector>
using namespace std;

/**
 * @brief
 *
 * 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
 * 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
 *
 * 6个关系运算类函数对象,每一种都是二元运算
 * 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
 *
 * 逻辑运算类运算函数,not为一元运算,其余为二元运算
 * 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
 */
void test01() {
  negate<int> p;
  cout << p(5) << endl;
  cout << negate<int>()(2) << endl;
}
void test02() { cout << plus<int>()(2, 3) << endl; }
class Print {
public:
  void operator()(int a) { cout << a << endl; }
};
void test03() {
  vector<int> v;
  v.push_back(1);
  v.push_back(3);
  v.push_back(2);
  v.push_back(5);
  sort(v.begin(), v.end(), greater<int>());
  for_each(v.begin(), v.end(), Print());
  sort(v.begin(), v.end(), less<int>());
  for_each(v.begin(), v.end(), Print());
}
int main() {
  test01();
  test02();
  test03();
}

-5
-2
5
5
3
2
1
1
2
3
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

```cpp
#include <algorithm>
#include <functional>
#include <iostream>
#include <stdlib.h>
#include <string.h>
#include <string>
#include <vector>
using namespace std;

/**
 * @brief
 * 一元继承 : public unary_function<参数1,返回值类型>
 * 二元继承 : public binary_function<int,int,void>
 */
class Print : public binary_function<int, int, void> {
public:
  void operator()(int a, int num) const {
    cout << a << " " << num << endl;
    cout << a + num << endl;
  }
};
/**
 * @brief 绑定参数
 * bind1st bind2nd
 *
 * bind1st : 将参数绑定为函数对象的第一个参数
 * bind2nd : 将参数绑定为函数对象的第二个参数
 */
void test01() {
  vector<int> v;
  v.push_back(1);
  v.push_back(3);
  v.push_back(2);
  v.push_back(5);
  for_each(v.begin(), v.end(), bind2nd(Print(), 200));
}

class GreaterThenFive : public unary_function<int, bool> {
public:
  bool operator()(int v) const { return v > 5; }
};

/**
 * @brief 取反适配器
 *
 * not1 : 对一元函数对象取反
 * not2 : 对二元函数对象取反
 */
void test02() {
  vector<int> v;
  for (int i = 0; i < 10; i++) {
    v.push_back(i);
  }
  // 返回第一个大于5的迭代器
```

```
  // vector<int>::iterator it = find_if(v.begin(), v.end(),
GreaterThenFive());
  // 返回第一个小于5的迭代器
  // vector<int>::iterator it = find_if(v.begin(), v.end(),
  // not1(GreaterThenFive())); 自定义输入
  vector<int>::iterator it =
      find_if(v.begin(), v.end(), not1(bind2nd(greater<int>(), 5)));
  if (it == v.end()) {
    cout << "没找到" << endl;
  } else {
    cout << "找到 " << *it << endl;
  }
  // 排序 二元函数对象
  sort(v.begin(), v.end(), not2(less<int>()));
}

void MyPrint03(int v, int v2) { cout << v + v2 << " "; }
/**
 * @brief 函数指针适配器 prt_fun
 *
 * ptr_fun() : 把一个普通的函数指针适配成函数对象
 */
void test03() {
  vector<int> v;
  for (int i = 0; i < 10; i++) {
    v.push_back(i);
  }
  for_each(v.begin(), v.end(), bind2nd(ptr_fun(MyPrint03), 100));
}

class Person {
public:
  Person(string name, int age) {
    m_Name = name;
    m_Age = age;
  }
  void ShowPerson() {
    cout << "成员函数: "
         << "Name: " << m_Name << " Age: " << m_Age << endl;
  }
  void Plus100() { m_Age += 100; }

public:
  string m_Name;
  int m_Age;
};
void MyPrint04(Person &p) {
  cout << "姓名: " << p.m_Name << " 年龄: " << p.m_Age << endl;
}
/**
 * @brief 成员函数适配器
 *
 * 如果容器中存放的是对象指针,那么用 mem_fun
 * 如果容器中存放的是对象实体,那么用 men_fun_ref
 */
void test04() {
  vector<Person> v;
  Person p1("aaa", 10);
```

```
  Person p2("bbb", 20);
  Person p3("ccc", 30);
  Person p4("ddd", 40);
  v.push_back(p1);
  v.push_back(p2);
  v.push_back(p3);
  v.push_back(p4);
  // 利用 mem_fun_ref 将 Person 内部成员函数适配
  for_each(v.begin(), v.end(), mem_fun_ref(&Person::ShowPerson));
}
void test05() {
  vector<Person *> v;
  Person p1("aaa", 10);
  Person p2("bbb", 20);
  Person p3("ccc", 30);
  Person p4("ddd", 40);
  v.push_back(&p1);
  v.push_back(&p2);
  v.push_back(&p3);
  v.push_back(&p4);
  for_each(v.begin(), v.end(), mem_fun(&Person::ShowPerson));
}

int main() {
  test01();
  cout << "===========" << endl;
  test02();
  cout << "===========" << endl;
  test03();
  cout << "===========" << endl;
  test04();
  cout << "===========" << endl;
  test05();
}

1 200
201
3 200
203
2 200
202
5 200
205
===========
找到 0
===========
100 101 102 103 104 105 106 107 108 109 ===========
成员函数: Name: aaa Age: 10
成员函数: Name: bbb Age: 20
成员函数: Name: ccc Age: 30
成员函数: Name: ddd Age: 40
===========
成员函数: Name: aaa Age: 10
成员函数: Name: bbb Age: 20
成员函数: Name: ccc Age: 30
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

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

II. transform 算法

```
/**
 * @brief 将指定容器区间元素搬运到另一容器中
 *          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
 * @param beg1 源容器开始迭代器
 * @param end1 源容器结束迭代器
 * @param beg2 目标容器开始迭代器
 * @param _callback 回到函数或者函数对象
 * @return 返回目标容器迭代器
 */
transform(iterator beg1,iterator end1,iterator beg2,_callback);
```

2. 常用查找算法

```
/**
 * @brief find 算法 查找元素
 *
 * @param beg 容器开始迭代器
 * @param end 容器结束迭代器
 * @param value 查找的元素
 * @return 返回查找元素的位置
 */
find(iterator beg, iterator end, value);

/**
 * @brief find_if 算法 条件查找
 *
 * @param beg 容器开始迭代器
 * @param end 容器结束迭代器
 * @param callback 回调函数或者谓词(返回bool类型的函数对象)
 * @return bool 查找返回true,否则false
```

```
 */
find_if(iterator beg, iterator end, _callback);

/**
 * @brief adjacent_find 算法 查找相邻重复元素
 *
 * @param beg 容器开始迭代器
 * @param end 容器结束迭代器
 * @param _callback 回调函数或者谓词(返回bool类型的函数对象)
 * @return 返回相邻元素的第一个位置的迭代器
 */
adjacent_find(iterator beg, iterator end, _callback);

/**
 * @brief binary_search 算法 二分查找法
 *  在无序序列中不可用
 * @param beg 容器开始迭代器
 * @param end 容器结束迭代器
 * @param value 查找的元素
 * @return bool 查找返回true,否则返回false
 */
bool binary_search(iterator beg, iterator end, value);

/**
 * @brief count 算法 统计元素出现次数
 *
 * @param beg 容器开始迭代器
 * @param end 容器结束迭代器
 * @param value 查找的元素
 * @return int 返回元素个数
 */
count(iterator beg, iterator end, value);

/**
 * @brief count_if 算法 统计元素出现次数
 *
 * @param beg 容器开始迭代器
 * @param end 容器结束迭代器
 * @param callback 回调函数或者谓词(返回bool类型的函数对象)
 * @return int 返回元素个数
 */
count_if(iterator beg, iterator end, _callback);
```

3. 常用排序算法

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

```

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

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

/**
 * @brief reverse 算法 反转指定范围的元素
 *
 * @param beg 容器开始迭代器
 * @param end 容器结束迭代器
 */
reverse(iterator beg,iterator end);
```

4. 常用拷贝和替换算法

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

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

/**
 * @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
 *
 * @param beg 容器开始迭代器
 * @param end 容器结束迭代器
 * @param _callback 函数回调或者谓词(返回bool类型的函数对象)
 * @param newValue 新元素
 */
replace_if(iterator beg,iterator end,_callback,newValue);

/**
```

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

5. 常用算数生成算法

```
/**
 * @brief accumulate 算法 计算容器元素累积总和
 *
 * @param beg 容器开始迭代器
 * @param end 容器结束迭代器
 * @param value 累加值
 */
accumulate(iterator beg,iterator end,value);

/**
 * @brief fill 算法 向容器中添加元素
 *
 * @param beg 容器开始迭代器
 * @param end 容器结束迭代器
 * @param value 填充元素
 */
fill(iterator beg,iterator end,value);
```

6. 常用集合算法

```
/**
 * @brief set_intersection 算法 求两个 set 集合的交集
 *  两个集合必须是有序序列
 * @param beg1 容器1 开始迭代器
 * @param end1 容器1 结束迭代器
 * @param beg2 容器2 开始迭代器
 * @param end2 容器2 结束迭代器
 * @param dest 目标容器开始迭代器
 * @return 目标容器的最后一个元素的迭代器地址
 */
set_intersection(iterator beg1, iterator end1, iterator beg2, iterator end2,
                 iterator dest);

/**
 * @brief set_union 算法 求两个 set 集合的并集
 *  两个集合必须是有序序列
 * @param beg1 容器1 开始迭代器
 * @param end1 容器1 结束迭代器
 * @param beg2 容器2 开始迭代器
 * @param end2 容器2 结束迭代器
 * @param dest 目标容器开始迭代器
 * @return 目标容器的最后一个元素的迭代器地址
 */
set_unnion(iterator beg1, iterator end1, iterator beg2, iterator end2,
           iterator dest);

/**
```

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