第一章 STL 理论基础

1.1 本章学习重点

- STL 基本概念
- 理解容器的概念
- 理解迭代器的概念
- 理解算法的概念

1.2 STL 基本概念

STL(Standard Template Library,标准模板库)，是惠普实验室开发的一系列软件的统称。现在主要出现在 c++中，但是在引入 c++之前该技术已经存在很长时间了。

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或者模板函数，这相比传统的由函数和类组成的库来说提供了更好的代码重用机会。

STL(Standard Template Library)标准模板库,在我们 c++标准程序库中隶属于 STL 的占到了 80%以上。

在 c++标准中，STL 被组织成以下 13 个头文件：

<algorithm>、<deque>、<functional>、<iterator>、<vector>、<list>、<map>、<memory>、<numeric>、<queue>、<set>、<stack> 和<utility>

那么说了这么多，STL 还有什么优点呢？

- STL 是 C++的一部分，因此不用额外安装什么，它被内建在你的编译器之内。
- STL 的一个重要特点是数据结构和算法的分离。尽管这是个简单的概念，但是这种分离

确实使得 STL 变得非常通用。例如:在 STL 的 vector 容器中，可以放入元素、基础数据类型变量、元素的地址；STL 的 sort() 排序函数可以用来操作 vector,list 等容器。

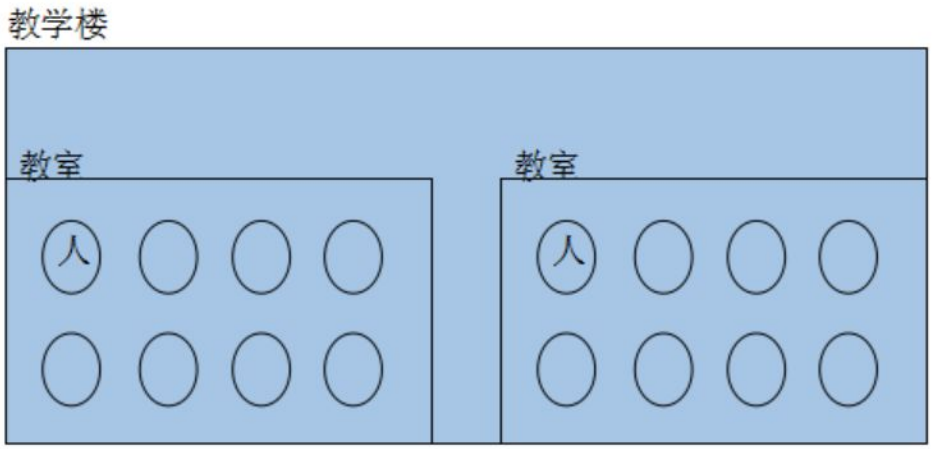
- 程序员可以不用思考 STL 具体的实现过程，只要能够熟练使用 STL 就 OK 了。这样他们就可以把精力放在程序开发的别的方面。
- STL 具有高可重用性，高性能，高移植性，跨平台的优点。
  - 高可重用性：STL 中几乎所有的代码都采用了模板类和模版函数的方式实现，这相比于传统的由函数和类组成的库来说提供了更好的代码重用机会。关于模板的知识，已经给大家介绍了。
  - 高性能：如 map 可以高效地从十万条记录里面查找出指定的记录，因为 map 是采用红黑树的变体实现的。(红黑树是平横二叉树的一种)
  - 高移植性：如在项目 A 上用 STL 编写的模块，可以直接移植到项目 B 上。
  - 跨平台：如用 windows 的 Visual Studio 编写的代码可以在 Mac OS 的 XCode 上直接编译。

Степанов Александр Александрович(STL 创建者)

1.3 STL 三大组件介绍

1.3.1 容器概念介绍

STL 中容器是指存储有限数据元素的一种数据结构。比如栈(stack)，队列(queue)...那么什么是数据结构，我们研究把数据按照特定的方式排列起来，便于我们查找 删除 排序或者其他一些目的，这种不同的排列方式我们就可以叫数据结构。



教室中包含了很多学生，教室就是一个容器，学生就是容器中存储的一个元素，教学楼包含了很多教室，教学楼就是一个容器，教室就是容器中的一个元素。

这里面还包含了一层含义：容器可以包含容器(教学楼和教室都是容器，但是教学楼里可以放教室)

比如拿教室这个容器来举例：我们怎么能快速定位教室中某个人的位置，是不是需要我们教室中的所有人按照一定的规则排序，这样我就能快速定位一个学生的坐在那里，那么依据我们的需求，让学生按照不同的规则排列，这种不同的排列就叫做数据结构。

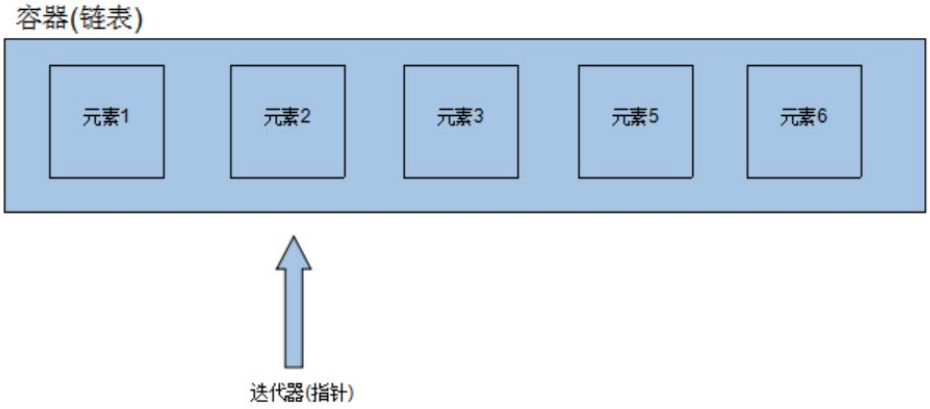
在我们 STL 中容器分为：序列式容器和关联式容器。

- 序列式容器就是根据学生进入教室的时间和地点来决定学生在那个位置，跟学生是谁没关系。
- 关联式容器是指我教室的座位按照一定规则确定好了，每个学生进来，比如根据学生出生年月，从小到大排列，学生坐在那个位置，必须由我的规则来规定。

1.3.2 迭代器介绍

迭代器是一种抽象出来的概念，现实中不容易找出来某项事物与之对应，所以较难理解。

但是在我们程序中，比如我们写的数据，我们通过[]操作符遍历取值，那么[]就是一个迭代器，也必须说我们经常用的指针，他也是一种迭代器。



迭代器（iterator）是一种对象，它能够用来遍历标准模板库容器中的部分或全部元素，每个迭代器对象代表容器中的确定的地址。迭代器修改了常规指针的接口，所谓迭代器是一种概念上的抽象：那些行为上像迭代器的东西都可以叫做迭代器，也就是说迭代器就是对我们普通的指针做了一层封装，其行为也类似指针。我们现在呢？可以单纯得把迭代器理解为，它就是一个指针，用来指向不同的元素,既然是指针，那么指针的一些基本运算操作，比如*、++、==、!=、=,迭代器也可以进行这样的操作。

1.3.3 算法介绍

以有限的步骤，解决逻辑或者数学上的问题，这门学科我们就叫做算法。一般来说，我们每天都在写各种各样的算法，比如我们写的每一个函数，它被用来解决或大或小的问题。

在我们工作中，我们要写一个算法来解决一个问题的时候，那么需要考虑你写的算法需要消耗的计算机资源，包括时间和空间，如果你写一个算法需要消耗 1G 内存来解决这个问题，那么你这个算法也就没有什么价值了。

STL 为我们的提供的算法，都很高效，而且还有个最大的特点，可复用性。那么我们学习算法，就很简单了，我们只需要去熟悉并且能熟练应用 STL 为我们提供的常用算法就 OK 了。

STL 提供了大约 100 个实现算法的模版函数，比如算法 for_each 将为指定序列中的每一个元素调用指定的函数等。这样一来，只要我们熟悉了 STL 之后，许多代码可以被大大的化简，只需要通过调用一两个算法函数，就可以完成所需要的功能并大大地提升效率

1.3.4 案例

13.4.1 案例一: STL 入门 hello world

```
#define _CRT_SECURE_NO_WARNINGS

#include<iostream>
#include<vector>
#include<algorithm>

using namespace std;



//STL 中的容器 算法 迭代器
void test01(){
```

```
    vector<int> v;  //STL 中的标准容器之一 ： 动态数组
    v.push_back(1); //vector 容器提供的插入数据的方法
    v.push_back(5);
    v.push_back(3);
    v.push_back(7);


    //迭代器
    vector<int>::iterator pStart = v.begin(); //vector 容器提供了 begin()方法 返回指向第一个元素的迭代器
    vector<int>::iterator pEnd = v.end(); //vector 容器提供了 end()方法 返回指向最后一个元素下一个位置的迭代器

    //通过迭代器遍历
    while (pStart != pEnd){
        cout << *pStart << " ";
        pStart++;
    }
    cout << endl;

    //算法 count 算法 用于统计元素的个数
    int n = count(pStart, pEnd, 5);
    cout << "n:" << n << endl;

}

//STL 容器不单单可以存储基础数据类型，也可以存储类对象
class  Teacher
{
public:
    Teacher(int age) :age(age){};
    ~Teacher(){};
public:
    int age;
};

void test02(){

    vector<Teacher> v; //存储 Teacher 类型数据的容器
    Teacher t1(10), t2(20), t3(30);
    v.push_back(t1);
    v.push_back(t2);
    v.push_back(t3);
```

```
    vector<Teacher>::iterator pStart = v.begin();
    vector<Teacher>::iterator pEnd = v.end();

    //通过迭代器遍历
    while (pStart != pEnd){
        cout << pStart->age << " ";
        pStart++;
    }
    cout << endl;

}

//存储 Teacher 类型指针
void test03(){

    vector<Teacher*> v; //存储 Teacher 类型指针
    Teacher* t1 = new Teacher(10);
    Teacher* t2 = new Teacher(20);
    Teacher* t3 = new Teacher(30);

    v.push_back(t1);
    v.push_back(t2);
    v.push_back(t3);

    //拿到容器迭代器
    vector<Teacher*>::iterator pStart = v.begin();
    vector<Teacher*>::iterator pEnd = v.end();

    //通过迭代器遍历
    while (pStart != pEnd){
        cout << (*pStart)->age << " ";
        pStart++;
    }
    cout << endl;

}

//容器嵌套容器  难点(不理解，可以跳过)
void test04(){

    vector<vector<int>> v; //容器中存储容器
    vector<int> v1,v2,v3;
    v1.push_back(1);
```

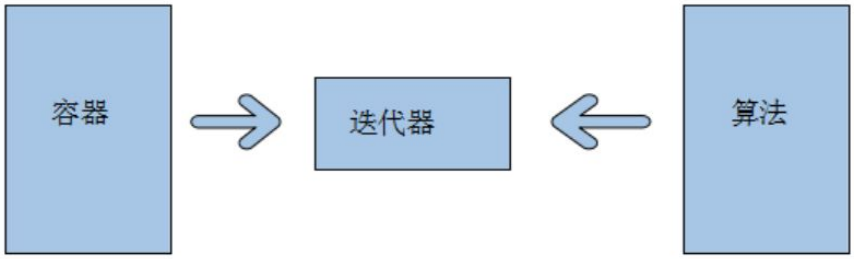
```
    v1.push_back(2);

    v2.push_back(10);

    v3.push_back(100);
    v3.push_back(200);

    v.push_back(v1);
    v.push_back(v2);
    v.push_back(v3);

    //拿到容器迭代器
    vector<vector<int>>::iterator pStart = v.begin();
    vector<vector<int>>::iterator pEnd = v.end();

    //通过迭代器遍历
    while (pStart != pEnd){
        vector<int> vTemp = *pStart; //获得迭代器当前指向的容器

        vector<int>::iterator tmpStart = vTemp.begin();
        vector<int>::iterator tmpEnd = vTemp.end();
        for (; tmpStart != tmpEnd;tmpStart++){
            cout << *tmpStart << " ";
        }
        cout << endl;
        pStart++;
    }
}

int main(){

    //test01();
    //test02();
    //test03();
    test04();

    system("pause");
    return EXIT_SUCCESS;
}
```

13.4.2 案例二：容器算法迭代器实现基本原理

```
#define _CRT_SECURE_NO_WARNINGS

#include<iostream>
#include<string>
using namespace std;
//算法
int mycount(int* start, int* end, int val){
    int n = 0;
    for (int* it = start; it != end;it++){
        if (*it == val){
            n++;
        }
    }
    return n;
}
int main(){

    //容器=>数组
    int arr[] = { 1, 2, 3, 5,4, 5, 6 };
    //迭代器 [] int*p []也是一种迭代器
    int* pStart = arr;  //开始迭代器
    int* pEnd = &(arr[sizeof(arr) / sizeof(int)]);  //结束迭代器
    //p++;
    //cout << *p << endl;
    //p++;
    //cout << *p << endl;
    //遍历容器
    while (pStart != pEnd){
        cout << *pStart << endl;
        pStart++;
    }

    int n = mycount(pStart, pEnd, 5); //算法 通过迭代器对容器中的元素进行统计
    cout << "n:" << n << endl;

    system("pause");
    return EXIT_SUCCESS;
}
```

1.3.5 总结

容器就是数据结构，用来将数据元素按照一定的规则进行排列，不同的容器拥有不同的排列规则，不同的排列规则可以达到不同的数据操作特点，比如数据这种数据结构，我们随机存取就很高效，算法就是提供对容器数据元素的一些操作，比如遍历容器元素，删除容器元素等迭代器就是容器和算法之间的桥梁 ,粘合剂 ,用来将两个相对独立的部件建立起关系。



STL 中容器和算法的设计是彼此分离，这样的好处就是：

1 容器和算法的编写可以分别编写，互补影响

2 容器只需要提供迭代器 算法只需要拿到迭代器就可以完成容器和算法之间的关联和操作

第二章 常用容器

2.1 本章学习重点

- 掌握 string 容器特性,及其相关 API 的使用
- 掌握 vector 容器特性,及其相关 API 的使用
- 掌握 deque 容器特性,及其相关 API 的使用
- 掌握 queue 容器特性,及其相关 API 的使用

- 掌握 stack 容器特性,及其相关 API 的使用
- 掌握 list 容器特性,及其相关 API 的使用
- 掌握 set/multiset 容器特性,及其相关 API 的使用
- 掌握 map/multiset 容器特性,及其相关 API 的使用
- 掌握函数对象的概念
- 理解 STL 容器元素的深拷贝和浅拷贝问题

2.2 string 容器

2.2.1 string 的特性

说到 string 的特性，就不得不和 char*类型的字符串的对比：

- Char*是一个指针，String 是一个类

string 封装了 char*，管理这个字符串，是一个 char*型的容器。

- String 封装了很多实用的成员方法

查找 find，拷贝 copy，删除 delete 替换 replace，插入 insert

- 不用考虑内存释放和越界

string 管理 char*所分配的内存。每一次 string 的复制 ,取值都由 string 类负责维护，不用担心复制越界和取值越界等。

string 和 char*可以互相转换吗？如果能，怎么转换呢？

答案是可以转换。string 转 char*通过 string 提供的 c_str()方法。

```
//string 转 char*
string str = "itcast";
const char* cstr = str.c_str();
//char* 转 string
char* s = "itcast";
string sstr(s);
```

2.2.2 string 常用 API

2.2.2.1 string 构造函数

```
string();//创建一个空的字符串 例如: string str;
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
//例子:
//默认构造函数
string s1;
//拷贝构造函数
string s2(s1);
string s2 = s1;
//带参数构造函数
char* str = "itcast";
string s3(str);
string s4(10, 'a');
```

2.2.2.2 string 基本赋值操作

```
string& operator=(const char* s);//char*类型字符串 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& assign(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
```

2.2.2.3 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
//例子:
string s = "itcast";
```

```
char c = s[1];
c = s.at(1);
```

 多学一招：

问：string 中存取字符[]和 at 的异同?

答: 1 相同,[]和 at 都可以返回第 n 个字符

2 不同，at 访问越界会抛出异常，[]越界会直接程序会挂掉。

2.2.2.4 string 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
```

2.2.2.5 string 查找和替换

```
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
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
```

2.2.2.6 string 比较操作

```
/*
compare 函数在>时返回 1，<时返回 -1，==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
```

2.2.2.7 string 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
```

2.2.2.8 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
```

2.2.3 string 课后练习

```
//用户邮箱地址验证
// 1 判断邮箱有效性 是否包含@和. 并且.在@之后
// 2 判断用户输入的用户名中是否包含除了小写字母之外字符(ASCII 范围 97~122)
// 3 判断用户输入的邮箱地址是否正确(zhaosi@itcast.cn)

#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
#include<string>
using namespace std;

// 1 判断邮箱有效性 是否包含@和.并且.在@之后
bool Check_Valid(string& email){

    int pos1 = email.find("@");
    int pos2 = email.find(".");
```

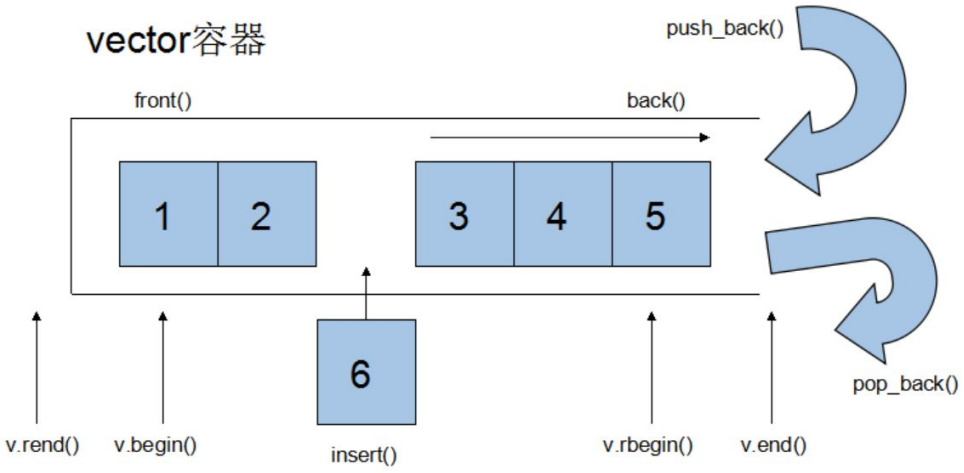
```
    //判断@和.是否存在
    if (pos1  == -1 || pos2 == -1){
        return false;
    }
    //判断@在.之前
    if (pos1 > pos2){
        return false;
    }

    return true;
}

//2 判断用户输入的用户名中是否包含除了小写字母之外字符(ASCII 范围 97~122)
bool Check_Username(string& email){

    int pos = email.find("@");
    string username = email.substr(0,pos-1);
    for (string::iterator it = username.begin(); it != username.end(); it++){
        if (*it < 97 || *it > 122){
            return false;
        }
    }

    return true;

}

// 3 判断用户输入的邮箱地址是否正确(zhaosi@itcast.cn)
bool Check_EqualtTo(string& email){

    string rightEmail = "zhaosi@itcast.cn";
    if (email.compare(rightEmail) != 0){
        return false;
    }
    return true;
}

void testEmail(){

    //用户邮箱地址验证
    // 1 判断邮箱有效性 是否包含@和. 并且.在@之后
    // 2 判断用户输入的用户名中是否包含除了小写字母之外字符(ASCII 范围 97~122)
    // 3 判断用户输入的邮箱地址是否正确(zhaosi@itcast.cn)
```

```
    string email;
    cout << "请输入您的邮箱：" << endl;
    cin >> email;

    bool flag = Check_Valid(email);
    if (!flag){
        cout << "emain 格式不合法!" << endl;
        return;
    }

    flag = Check_Username(email);
    if (!flag){
        cout << "用户名中包含除小写字母之外的字母!" << endl;
        return;
    }

    flag = Check_EqualtTo(email);
    if (!flag){
        cout << "邮箱地址不正确!" << endl;
        return;
    }

    cout << "邮箱输入正确!" << endl;
}

int main(){

    testEmail();

    system("pause");
    return EXIT_SUCCESS;
}
```

2.3 vector 容器

2.3.1 vector 特性

vector 容器是一个长度动态改变的动态数组，既然也是数组，那么其内存是一段连续的内存，具有数组的随机存取的优点。



特性总结:

- **vector 是动态数组，连续内存空间，具有随机存取效率高的优点。**
- **vector 是单口容器，在队尾插入和删除元素效率高，在指定位置插入会导致数据元素移动，效率低。**

多学一招：

问：vector 如何实现动态增长?

答: 当 vector 空间满的时候，再当插入新元素的时候，vector 会重新申请一块更大的内存空间，将原空间数据拷贝到新的内存空间，然后释放旧的内存空间，再将新元素插入到新空间中，以此可以看出 vector 的空间动态增长效率较低。

2.3.2 vector 常用 API

2.3.2.1 vector 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。

//例子 使用第二个构造函数 我们可以...
int arr[] = {2,3,4,1,9};
vector<int> v1(arr, arr + sizeof(arr) / sizeof(int));
```

2.3.2.2 vector 常用赋值操作

```
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。
vector& operator=(const vector  &vec);//重载等号操作符
swap(vec);// 将 vec 与本身的元素互换。

//第一个赋值函数，可以这么写：
int arr[] = { 0, 1, 2, 3, 4 };
assign(arr, arr + 5);//使用数组初始化 vector
```

2.3.2.3 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。
```

注意: resize 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。

 多学一招：

问：reserv 和 resize 的区别?

答： reserve 是容器预留空间，但在空间内不真正创建元素对象，所以在没有添加新的对象之前，不能引用容器内的元素.

resize 是改变容器的大小，且在创建对象，因此，调用这个函数之后，就可以引用容器内的对象了.

巧用 reserve 增加程序效率？

```
vector<int> v;
int* p = NULL;
int count = 0;// 统计 vector 容量增长几次?
for (int i = 0; i < 100000;i++){
     v.push_back(i);
     if (p != &v[0]){
          p = &v[0];
          count++;
     }
}
cout << "count:" << count << endl; //打印出 30
```

我向 vector 插入了 10 万个元素，vector 一共重新分配内存 30 次.

```
vector<int> v;
v.reserve(100000);
int* p = NULL;
int count = 0;// 统计 vector 容量增长几次?
for (int i = 0; i < 100000;i++){
     v.push_back(i);
     if (p != &v[0]){
          p = &v[0];
          count++;
     }
}
cout << "count:" << count << endl; //打印出 30
```

再次向 vector 插入了 10 万个元素，vector 一共重新分配内存 1 次.

当我们知道我们存储的元素大概有多少的时候,我们就可以使用 reserve 方法，来减少

vector 重新申请内存-拷贝数据-释放旧空间的次数。

2.3.2.4 vector 数据存取操作

```
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
```

2.3.2.5 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
```

总结: vector 是个动态数组，当空间不足的时候插入新元素，vector 会重新申请一块更大的内存空间，将旧空间数据拷贝到新空间，然后释放旧空间。vector 是单口容器，所以在尾端插入和删除元素效率较高，在指定位置插入，势必会引起数据元素移动，效率较低。

2.3 deque 容器

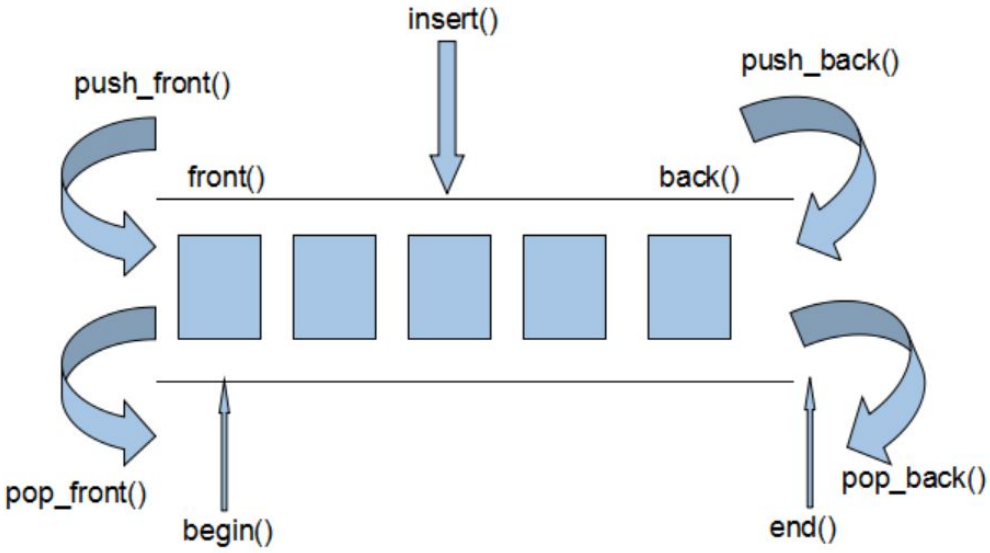
2.3.1 deque 特性

deque 是“double-ended queue”的缩写,和 vector 一样，deque 也支持随机存取。vector 是单向开口的连续性空间，deque 则是一种双向开口的连续性空间，所谓双向开口，意思是可以在头尾两端分别做元素的插入和删除操作，vector 当然也可以在头尾两端进行插入和删除操作，但是头部插入和删除操作效率奇差，无法被接受。

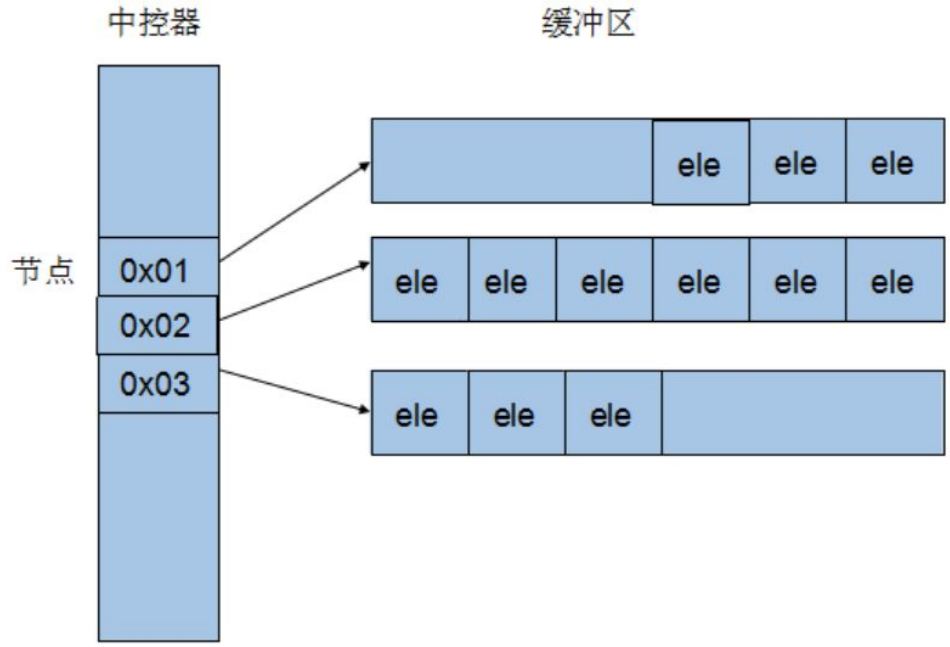
deque 和 vector 的最大差异？

一在于 deque 允许常数时间内对头端进行元素插入和删除操作。

二在于 deque 没有容量的概念，因为它是动态的以分段的连续空间组合而成，随时可以增加一段新的空间并链接起来，换句话说，像 vector 那样“因旧空间不足而重新分配一块更大的空间，然后再复制元素，释放空间”这样的操作不会发生在 deque 身上，也因此 deque 没有必要提供所谓的空间保留功能。



deque 操作示意图



deque 原理示意图

特性总结:

- **双端插入和删除元素效率较高.**
- **指定位置插入也会导致数据元素移动,降低效率.**
- **可随机存取,效率高.**

2.3.2 deque 常用 API

2.3.2.1 deque 构造函数

```
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。
```

2.3.2.2 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
```

2.3.2.3 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。
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。
```

2.3.2.3 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
```

2.3.2.4 deque 数据存取

```
at(idx);//返回索引 idx 所指的数据，如果 idx 越界，抛出 out_of_range。
operator[];//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
```

2.3.2.5 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。
```

经验之谈：deque 是分段连续的内存空间，通过中控器维持一种连续内存空间的状态，其实现复杂性要大于 vector queue stack 等容器，其迭代器的实现也更加复杂，在需要对

deque 容器元素进行排序的时候，建议先将 deque 容器中数据数据元素拷贝到 vector 容器中，对 vector 进行排序，然后再将排序完成的数据拷贝回 deque 容器。

2.3.2.5 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。
```

2.3.3 deque 课后练习

```
//评委打分案例(sort 算法排序)
//创建 5 个选手(姓名，得分)，10 个评委对 5 个选手进行打分
//得分规则：去除最高分，去除最低分，取出平均分
//按得分对 5 名选手进行排名

#define _CRT_SECURE_NO_WARNINGS

#include<iostream>
#include<deque>
#include<algorithm>
#include<vector>
#include<string>
using namespace std;

//选手类
class Player{
public:
    string name;
    int score;
};

//创建选手
void Create_Player(vector<Player>& plist){

    string randseed = "ABCDE";
    for (int i = 0; i < 5;i++){
        Player player;
        player.name = "选手";
        player.name += randseed[i];
```

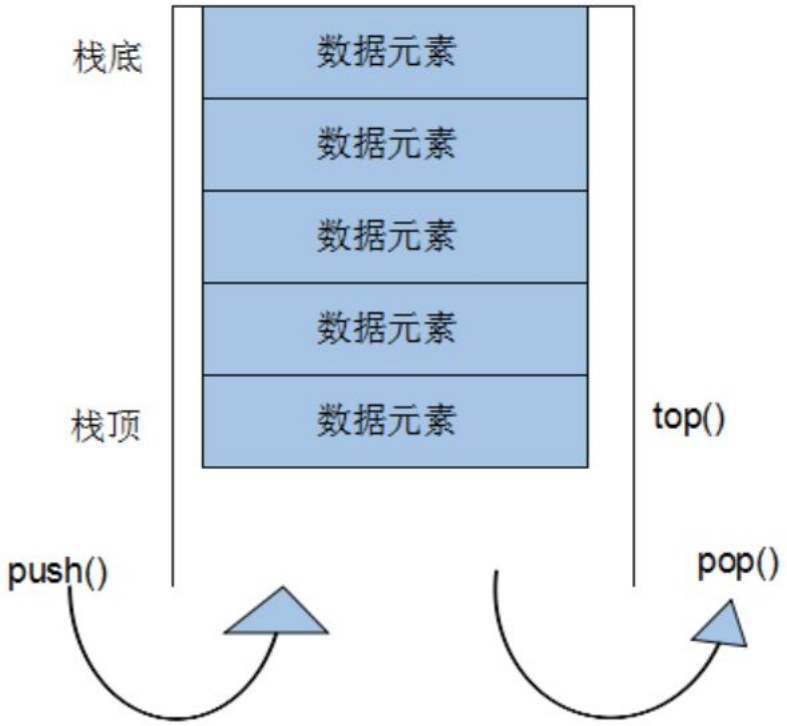
```
            player.score = 0;

            plist.push_back(player);
        }
}



bool myconpare(int v1,int v2){
    return v1 < v2;
}
//对选手进行打分
//打分规则： 去除最高分 去除最低分 取平均分
void Set_Player_Score(vector<Player>& plist){

    for (vector<Player>::iterator it = plist.begin(); it != plist.end();it++){

        deque<int> dscore; //保存评委的 10 次打分
        for (int i = 0; i < 10;i++){
            int score = 50 + rand() % 50;
            dscore.push_back(score);
        }
        //排序从大到小或者从小到大 sort 算法
        sort(dscore.begin(), dscore.end(), myconpare);
        //去除最低分 去除最高分
        dscore.pop_front();
        dscore.pop_back();
        //求平均分
        int totalscore = 0;
        for (deque<int>::iterator dit = dscore.begin(); dit != dscore.end();dit++){
            totalscore += *dit;
        }
        int scoreavg = totalscore / dscore.size();
        (*it).score = scoreavg;
    }

}

bool conparePlayer(Player player1, Player player2){
    return player1.score > player2.score;
}
//按照得分排名
void Show_Player_List(vector<Player>& plist){

    sort(plist.begin(), plist.end(), conparePlayer);
```

```
    cout << "选手得分排名:" << endl;
    for (vector<Player>::iterator it = plist.begin(); it != plist.end();it++){
        cout << "姓名:" << it->name << " 得分:" << it->score << endl;
    }

}

void test(){

    vector<Player> plist;
    //创建 5 名选手
    Create_Player(plist);
    //对 5 名选手进行打分
    Set_Player_Score(plist);
    //按照得分排名
    Show_Player_List(plist);

}
int main(){

    test();

    system("pause");
    return EXIT_SUCCESS;
}
```

2.4 stack 容器

2.4.1 stack 特性

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 只允许在栈顶新增元素，移除元素，获得顶端元素，但是除了顶端之外，其他地方不允许存取元素，只有栈顶元素可以被外界使用，也就是说 stack 不具有遍历行为，没有迭代器。



特性总结:

- **栈不能遍历,不支持随机存取，只能通过 top 从栈顶获取和删除元素.**

2.4.2 stack 常用 API

2.4.2.1 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 &stk);//拷贝构造函数
```

2.4.2.2 stack 赋值操作

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

2.4.2.3 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
```

2.4.2.3 stack 大小操作

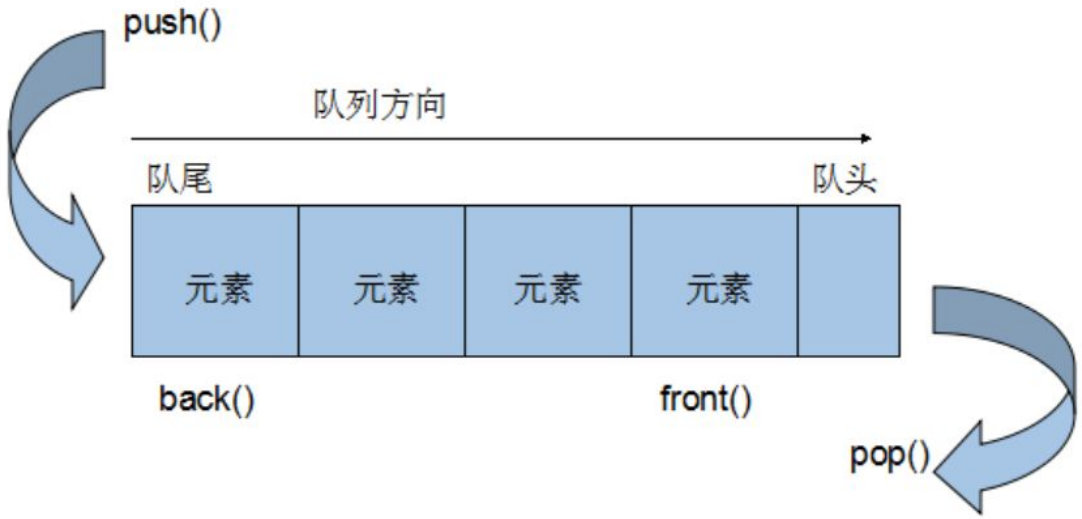
```
empty();//判断堆栈是否为空
size();//返回堆栈的大小
```

思考：一个栈的入栈顺序是: a b c d e, 则栈的输出顺序应该为?

2.5 queue 容器

2.5.1 queue 特性

queue 是一种先进先出(first in first out, FIFO)的数据类型,他有两个口，数据元素只能从一个口进，从另一个口出.队列只允许从队尾加入元素，队头删除元素，必须符合先进先出的原则，queue 和 stack 一样不具有遍历行为。



特性总结：

- 必须从一个口数据元素入队，另一个口数据元素出队。
- 不能随机存取，不支持遍历

2.5.2 queue 常用 API

2.5.2.1 queue 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
```

2.5.2.2 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
```

2.5.2.3 queue 赋值操作

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

2.5.2.3 queue 大小操作

```
empty();//判断队列是否为空
size();//返回队列的大小
```

课后思考：

队列和栈的共同点是什么？

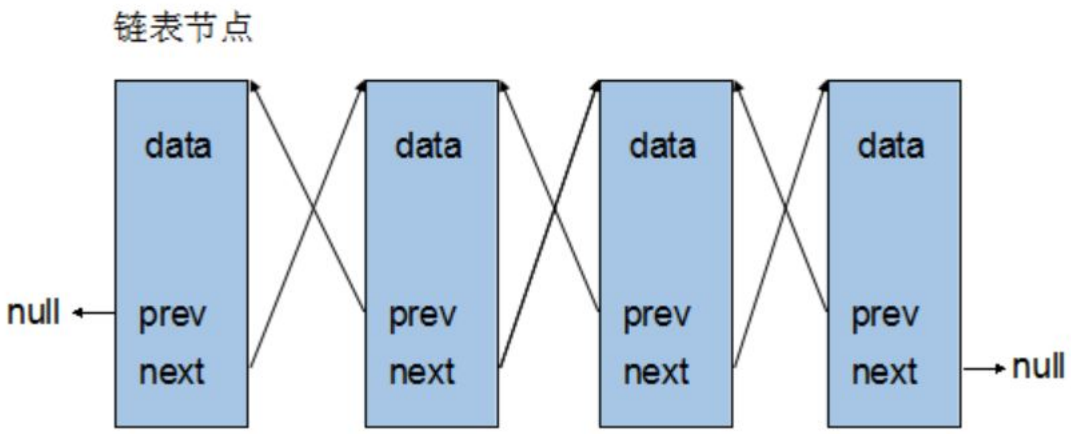
A. 都是先进先出 B.都是先进后出 C.只允许在端点出删除和插入操作 D.没有共同点

2.6 list 容器

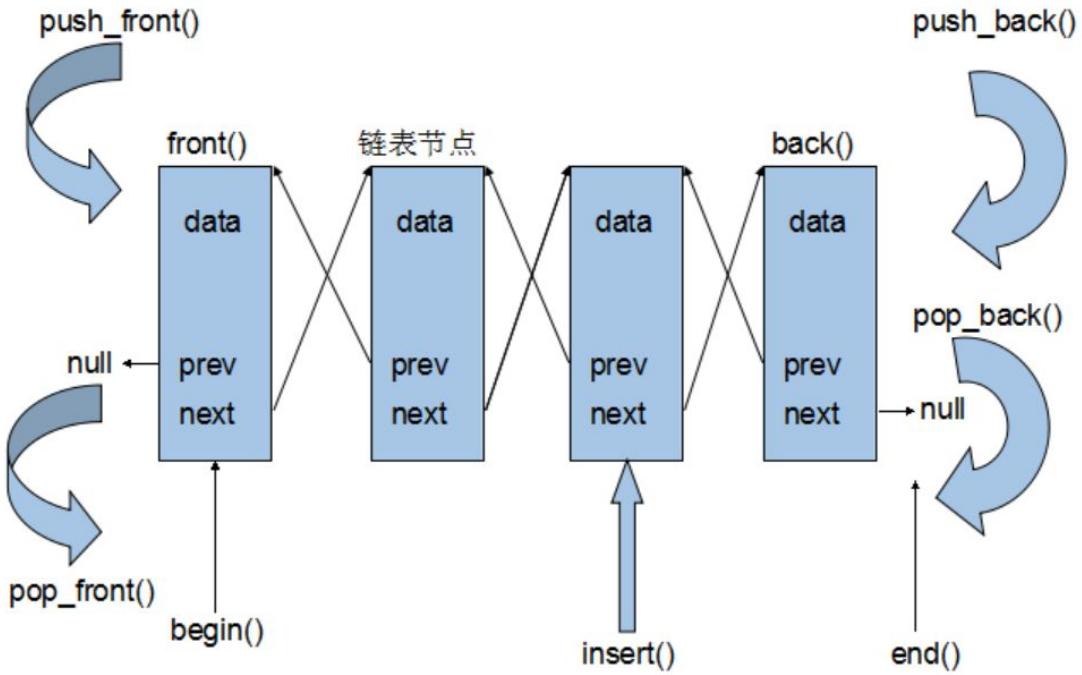
2.6.1 list 特性

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。



list 示意图



list 操作示意图

特性总结:

- 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

2.6.2 list 常用 API

2.6.2.1 list 构造函数

```
list<T> lstT;//list 采用采用模板类实现,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。
```

2.6.2.2 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 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。
```

2.6.2.3 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。
```

2.6.2.4 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。
```

2.6.2.5 list 数据的存取

```
front();//返回第一个元素。
back();//返回最后一个元素。
```

2.6.2.5 list 反转排列排序

```
reverse();//反转链表，比如 lst 包含 1,3,5 元素，运行此方法后，lst 就包含 5,3,1 元素。
sort(); //list 排序
```

课后思考:

- 链表和数组有什么区别？

1) 数组必须事先定义固定的长度（元素个数），不能适应数据动态地增减的情况。当数据增加时，可能超出原先定义的元素个数；当数据减少时，造成内存浪费。

2) 链表动态地进行存储分配，可以适应数据动态地增减的情况，且可以方便地插入、删除数据元素。（数组中插入、删除数据项时，需要移动其它数据项）

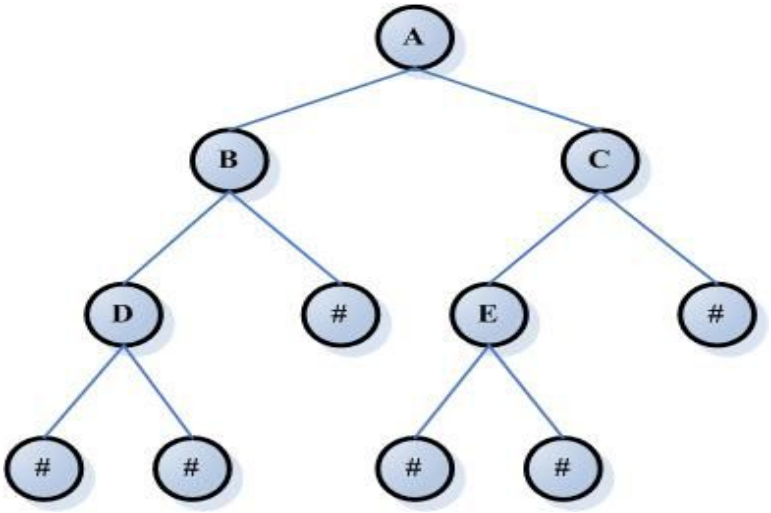
2.7 set/multiset 容器

2.7.1 set/multiset 特性

set/multiset 的特性是所有元素会根据元素的值自动进行排序。set 是以 RB-tree（红

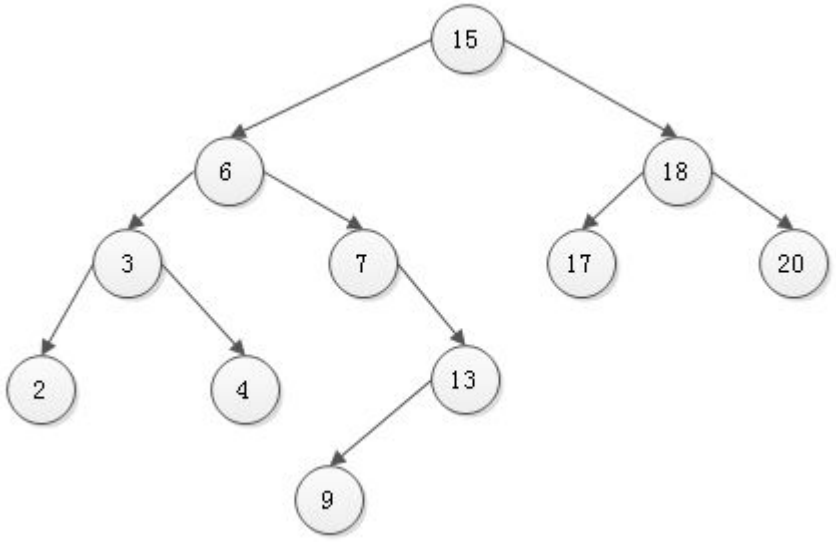
黑树，平衡二叉树的一种）为底层机制，其查找效率非常好。set 容器中不允许重复元素,multiset 允许重复元素。

二叉树就是任何节点最多只允许有两个字节点。分别是左子结点和右子节点。



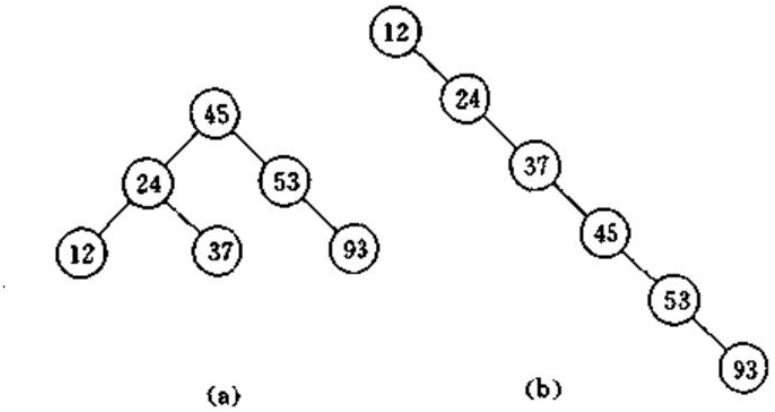
二叉树示意图

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。因此从根节点一直向左走，一直到无路可走，即得到最小值，一直向右走，直至无路可走，可得到最大值。那么在儿茶搜索树中找到最大元素和最小元素是非常简单的事情。下图为二叉搜索树：

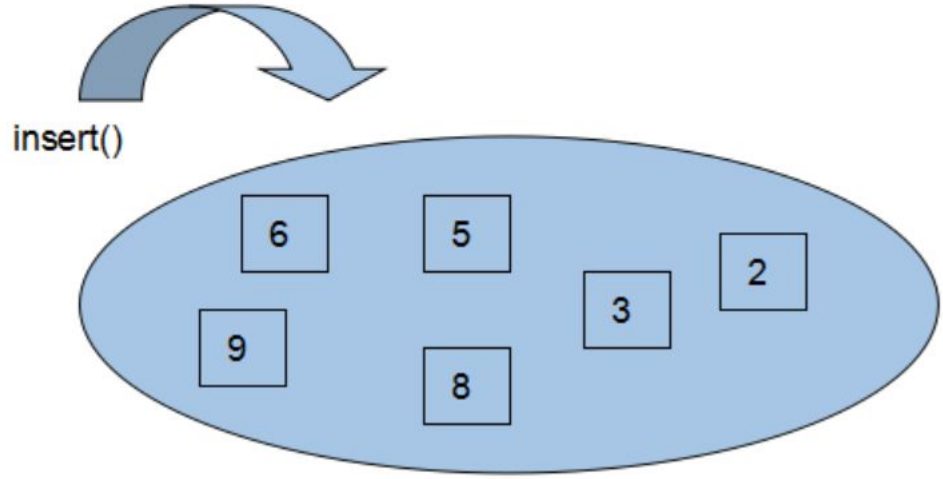


上面我们介绍了二叉搜索树，那么当一个二叉搜索树的左子树和右子树不平衡的时候，那么搜索依据上图表示，搜索 9 所花费的时间要比搜索 17 所花费的时间要多，由于我们的输入或者经过我们插入或者删除操作，二叉树失去平衡，造成搜索效率降低。

所以我们有了一个平衡二叉树的概念，所谓的平衡不是指的完全平衡。



RB-tree(红黑树)为二叉树的一种。



问：我们可以通过 set 的迭代器改变元素的值吗？

答: 不行，因为 set 集合是根据元素值进行排序，关系到 set 的排序规则，如果任意改变 set 的元素值，会严重破坏 set 组织。

2.7.2set 常用 API

2.7.2.1 set 构造函数

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

2.7.2.2 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
```

2.7.2.2 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
```

2.7.2.2 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。
```

2.7.2.3 set 查找操作

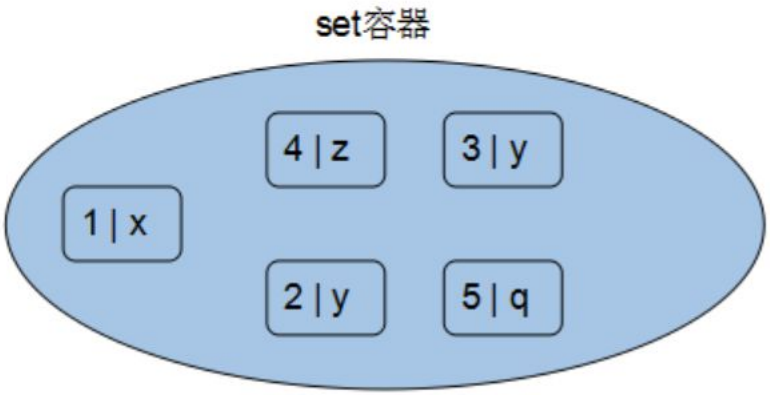
```
find(key);//查找键 key 是否存在,若存在，返回该键的元素的迭代器；若不存在，返回 map.end();
lower_bound(keyElem);//返回第一个 key>=keyElem 元素的迭代器。
upper_bound(keyElem);//返回第一个 key>keyElem 元素的迭代器。
equal_range(keyElem);//返回容器中 key 与 keyElem 相等的上下限的两个迭代器。
```

问题: 我们发现打印出来 set 集合中的元素是从小到大的升序排列，那么我们如何指定排序为降序呢？这个问题呢？我们需要了解函数对象的概念。

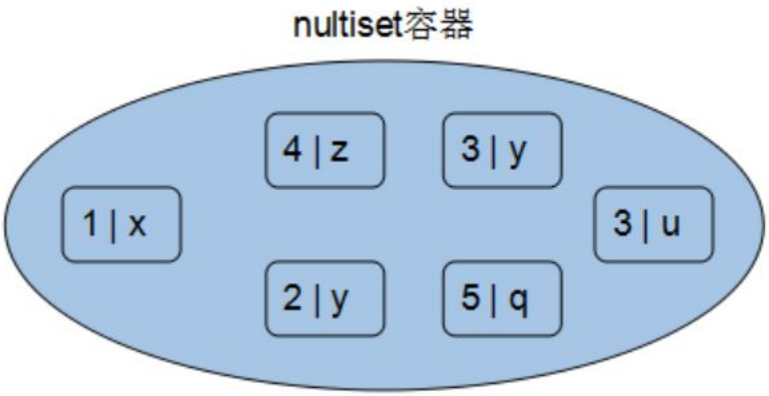
2.8 map/multimap 容器

2.8.1 map/multimap 特性

map 相对于 set 区别，map 具有键值和实值，所有元素根据键值自动排序。pair 的第一元素被称为键值，第二元素被称为实值。map 也是以红黑树为底层实现机制。



set 容器示意图



multiset 容器示意图

问题 ：我们通过 map 的迭代器可以修改 map 的键值吗？

答案是否定的，键值关系到容器内元素的排列规则，任意改变键值会破坏容器的排列规则，但是你可以改变实值。

map 和 multimap 区别在于，map 不允许相同 key 值存在，multimap 则允许相同 key 值存在。

2.8.2 对组

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 的两个公有函数 first 和 second 访问。

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

如何创建对组?

```
//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", 30);
cout << pair2.first << endl;
cout << pair2.second << endl;
//pair=赋值
pair<string, int> pair3 = pair2;
cout << pair3.first << endl;
cout << pair3.second << endl;
```

2.8.3 map 常用 API

2.8.3.1 map 构造函数

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

2.8.3.2 map 赋值操作

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

2.8.3.3 map 大小操作

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

2.8.3.4 map 插入数据元素操作

```
map.insert(...); //往容器插入元素，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.inset(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";
```

注意:

- 前三种方法，采用的是 insert()方法，该方法返回值为 pair<iterator,bool>
- 第四种方法非常直观，但存在一个性能的问题。插入 3 时，先在 mapStu 中查找主键为 3 的项，若没发现，则将一个键为 3，值为初始化值的对组插入到 mapStu 中，然后再将值修改成“小刘”。若发现已存在 3 这个键，则修改这个键对应的 value。
- string strName = mapStu[2]; //取操作或插入操作

只有当 mapStu 存在 2 这个键时才是正确的取操作，否则会自动插入一个实例， 键为 2，值为初始化值。

2.8.3.5 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。
```

2.8.3.5 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。
```

2.8.4 multimap 案例操练

```
    //multimap 案例
    //公司今天招聘了 5 个员工，5 名员工进入公司之后，需要指派员工在那个部门工作
    //人员信息有: 姓名 年龄 电话 工资等组成
    //通过 Multimap 进行信息的插入 保存 显示
    //分部门显示员工信息 显示全部员工信息
```

```cpp
#define _CRT_SECURE_NO_WARNINGS

#include<iostream>
#include<map>
#include<string>
#include<vector>
using namespace std;

//multimap 案例
//公司今天招聘了 5 个员工，5 名员工进入公司之后，需要指派员工在那个部门工作
//人员信息有: 姓名 年龄 电话 工资等组成
//通过 Multimap 进行信息的插入 保存 显示
//分部门显示员工信息 显示全部员工信息



#define SALE_DEPATMENT 1 //销售部门
#define DEVELOP_DEPATMENT 2 //研发部门
#define FINACIAL_DEPATMENT 3 //财务部门
#define ALL_DEPATMENT 4 //所有部门

//员工类
class person{
public:
    string name; //员工姓名
    int age; //员工年龄
    double salary; //员工工资
    string tele; //员工电话
};

//创建 5 个员工
void CreatePerson(vector<person>& vlist){

    string seed = "ABCDE";
    for (int i = 0; i < 5;i++){
        person p;
        p.name = "员工";
        p.name += seed[i];
        p.age = rand() % 30 + 20;
        p.salary = rand() % 20000 + 10000;
        p.tele = "010-8888888";
        vlist.push_back(p);
    }

}
```

```
//5 名员工分配到不同的部门
void PersonByGroup(vector<person>& vlist, multimap<int, person>& plist){


    int operate = -1; //用户的操作

    for (vector<person>::iterator it = vlist.begin(); it != vlist.end();it++){

        cout << "当前员工信息:" << endl;
        cout << "姓名: " << it->name << " 年龄:" << it->age << " 工资:" << it->salary <<
" 电话: " << it->tele << endl;
        cout << "请对该员工进行部门分配(1 销售部门, 2 研发部门, 3 财务部门):" << endl;
        scanf("%d", &operate);

        while (true){

            if (operate == SALE_DEPATMENT){  //将该员工加入到销售部门
                plist.insert(make_pair(SALE_DEPATMENT, *it));
                break;
            }
            else if (operate == DEVELOP_DEPATMENT){
                plist.insert(make_pair(DEVELOP_DEPATMENT, *it));
                break;
            }
            else if (operate == FINACIAL_DEPATMENT){
                plist.insert(make_pair(FINACIAL_DEPATMENT, *it));
                break;
            }else{
                cout <<"您的输入有误，请重新输入(1 销售部门, 2 研发部门, 3 财务部门):"
<< endl;
                scanf("%d", &operate);
            }

        }

    }
    cout << "员工部门分配完毕!" << endl;
    cout << "********************************************************" << endl;

}

//打印员工信息
void printList(multimap<int, person>& plist, int myoperate){
```

```
    if (myoperate == ALL_DEPATMENT){
        for (multimap<int, person>::iterator it = plist.begin(); it != plist.end(); it++){
            cout <<"姓名：" << it->second.name <<" 年龄:" << it->second.age <<" 工资:"
<< it->second.salary <<" 电话：" << it->second.tele << endl;
        }
        return;
    }

    multimap<int, person>::iterator it = plist.find(myoperate);
    int depatCount = plist.count(myoperate);
    int num = 0;
    if (it != plist.end()){
        while (it != plist.end() && num < depatCount){
            cout <<"姓名：" << it->second.name <<" 年龄:" << it->second.age <<" 工资:"
<< it->second.salary <<" 电话：" << it->second.tele << endl;
            it++;
            num++;
        }
    }
}

//根据用户操作显示不同部门的人员列表
void ShowPersonList(multimap<int, person>& plist, int myoperate){

    switch (myoperate)
    {
    case SALE_DEPATMENT:
        printList(plist, SALE_DEPATMENT);
        break;
    case DEVELOP_DEPATMENT:
        printList(plist, DEVELOP_DEPATMENT);
        break;
    case FINACIAL_DEPATMENT:
        printList(plist, FINACIAL_DEPATMENT);
        break;
    case ALL_DEPATMENT:
        printList(plist, ALL_DEPATMENT);
        break;
    }
}

//用户操作菜单
void PersonMenue(multimap<int, person>& plist){
```

```
    int flag = -1;
    int isexit = 0;
    while (true){
        cout <<"请输入您的操作((1 销售部门, 2 研发部门, 3 财务部门, 4 所有部门, 0 退出):
" << endl;
        scanf("%d",&flag);

        switch (flag)
        {
            case SALE_DEPATMENT:
                ShowPersonList(plist, SALE_DEPATMENT);
                break;
            case DEVELOP_DEPATMENT:
                ShowPersonList(plist, DEVELOP_DEPATMENT);
                break;
            case FINACIAL_DEPATMENT:
                ShowPersonList(plist, FINACIAL_DEPATMENT);
                break;
            case ALL_DEPATMENT:
                ShowPersonList(plist, ALL_DEPATMENT);
                break;
            case 0:
                isexit = 1;
                break;
            default:
                cout << "您的输入有误，请重新输入!" << endl;
                break;
        }

        if (isexit == 1){
            break;
        }
    }

}



int main(){

    vector<person>  vlist; //创建的 5 个员工 未分组
    multimap<int, person> plist; //保存分组后员工信息
```

```
    //创建 5 个员工
    CreatePerson(vlist);
    //5 名员工分配到不同的部门
    PersonByGroup(vlist, plist);
    //根据用户输入显示不同部门员工信息列表 或者 显示全部员工的信息列表
    PersonMenue(plist);

    system("pause");
    return EXIT_SUCCESS;
}
```

2.9 STL 容器共性机制

STL 容器所提供的都是值(value)寓意，而非引用(reference)寓意，也就是说当我们给容器中插入元素的时候，容器内部实施了拷贝动作，将我们要插入的元素再另行拷贝一份放入到容器中，而不是将原数据元素直接放进容器中，也就是说我们提供的元素必须能够被拷贝。

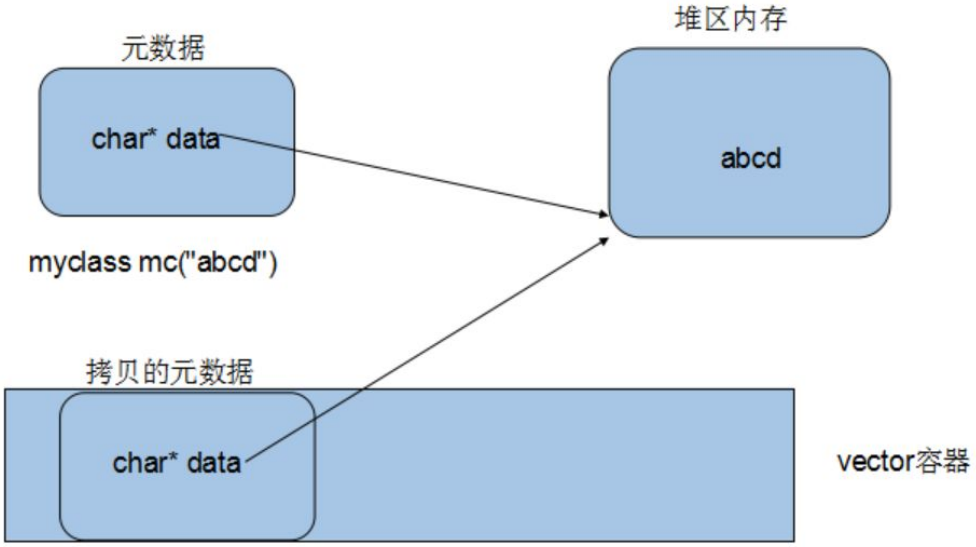
- 除了 queue 和 stack 之外，每个容器都提供可返回迭代器的函数，运用返回的迭代器就可以访问元素。
- 通常 STL 不会抛出异常，需要使用者传入正确参数。
- 每个容器都提供了一个默认的构造函数和默认的拷贝构造函数。
- 大小相关的构造方法: 1 size()返回容器中元素的个数 2 empty()判断容器是否为空

那么当我们在向容器插入元素的时候，需要考虑一种情况，代码：

```
#include<iostream>
#include<vector>
using namespace std;

class myclass{
```

```
public:
    myclass(char* data){
        int len = strlen(data) + 1; //计算传进来的字符串长度
        this->data = new char[len];  //在堆区分配了 len 字节内存
        strcpy(this->data, data); //将数据拷贝到我们在堆分配的内存中
    }
    //既然我们在堆区分配了内存，需要在析构函数中释放内存
    ~myclass(){
        delete[] this->data;
        this->data = NULL;
    }
private:
    char* data;
};

void test_deep_copy(){
    char* data = "abcd";
    myclass mc(data); //创建 myclass 的实例 并用 char*字符串 data 初始化对象

    vector<myclass> v;//创建 vector 容器
    v.push_back(mc); //将 mc 实例插入到 vector 容器尾部
}

int main(){
    test_deep_copy(); //调用测试函数
    return 0;
}
```

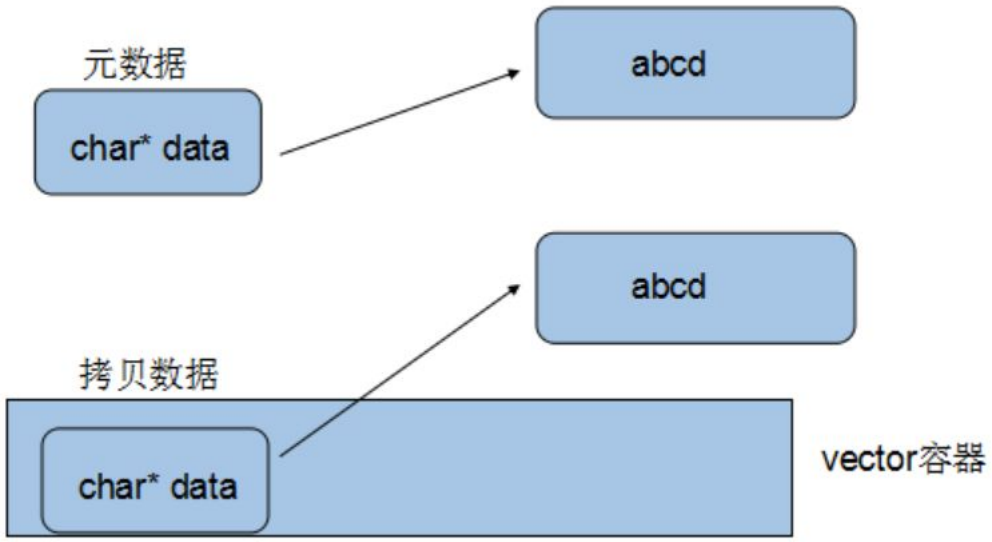
那么以上代码会发生什么问题呢？我们在函数 test_deep_copy()中创建了一个 myclass 对象，vector 对象，这两个对象在函数 test_deep_copy()执行完毕之后，会调用自身的析构函数，我们开题说了，STL 容器都是值引用，再向容器中加入元素的时候，实际上是对元数据进行了一份拷贝，将拷贝的数据放入到容器中，如下图：



由于我们没有提供拷贝构造函数,没有重载=操作符，vector 对我们的 mc 对象进行的简单的浅拷贝，将拷贝的对象插入到容器中，导致我们的 mc 对象的 data 指针和容器中 mc 对象的拷贝对象中的 data 指针都指向了我们在堆区分配的内存，当函数结束，两个对象都调用了析构函数，先调用析构函数的对象成功释放了堆区内存，后调用析构函数的对象一释放，程序挂掉了。

原因在于两个指针指向了同一块堆区内存，这样会导致不可预知的结果，函数结束其中一个调用析构函数，销毁了 data 指向的内存空间，而另一个对象析构的时候就会挂掉。

问题的解决办法就是，给我们的对象提供一个拷贝构造函数，并且重载=操作符,两个指针分别指向自己的那一块内存，互不影响。




```
#include<iostream>
#include<vector>
using namespace std;

class myclass{
public:
    myclass(char* data){
        int len = strlen(data) + 1; //计算传进来的字符串长度
        this->data = new char[len];  //在堆区分配了 len 字节内存
        strcpy(this->data, data); //将数据拷贝到我们在堆分配的内存中
    }
    //增加拷贝构造函数
    myclass(const myclass& mc){
        int len = strlen(mc.data) + 1;
        this->data = new char[len];
        strcpy(this->data, mc.data);
    }
    //重载 operator=操作符
    myclass& operator=(const myclass& mc){
        int len = strlen(mc.data) + 1;
        this->data = new char[len];
        strcpy(this->data, mc.data);
        return *this;
    }
    //既然我们在堆区分配了内存，需要在析构函数中释放内存
    ~myclass(){
        if (NULL != this->data){
            delete[] this->data;
            this->data = NULL;
```

```
        }
    }
private:
    char* data;
};

void test_deep_copy(){
    char* data = “abcd”;
    myclass mc(data); //创建 myclass 的实例 并用 char*字符串 data 初始化对象

    vector<myclass> v;//创建 vector 容器
    v.push_back(mc); //将 mc 实例插入到 vector 容器尾部

}

int main(){
    test_deep_copy(); //调用测试函数
    return 0;
}
```

2.10 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：是	否
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，我们经常要查看历史记录，比如上一次的记录，上上次的记录，但却不会去删除记录，因为记录是事实的描述。
- 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 deque，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：

  一：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。

  二：如果有大量释放操作的话，vector 花的时间更少，这跟二者的内部实现有关。

  三：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，随时可能有乘客下车，支持频繁的不确实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，想要快速要通过 ID 查找对应的用户。二叉树的查找效率，这时就体现出来了。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。

第三章 常用算法

- 掌握函数对象适配器
- 了解算法基本分类
- 掌握常用遍历算法
- 掌握常用查找算法
- 掌握常用排序算法
- 掌握常用拷贝和替换算法
- 掌握常用算数生成算法
- 掌握集合常用算法

3.1 函数对象

3.1.1 函数对象的概念

重载函数调用操作符的类，其对象常称为函数对象（function object），即它们是行为类似函数的对象，也叫仿函数(functor),其实就是重载“()”操作符，使得类对象可以像函数那样调用。

注意:1.函数对象(仿函数)是一个类，不是一个函数。

2.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。

假定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，就将这个类称为“二元仿函数”（binary functor）。

函数对象基本概念

函数对象也可以有参数和返回值

函数对象超出函数概念，可以保存函数调用状态

函数对象做参数和返回值

```
#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
#include<vector>
#include<algorithm>
#include<functional>
using namespace std;

class FuncObject01{
public:
    void operator()(){
        cout << "hello world" << endl;
    }
};

void FuncObject02(){
    cout << "hello world" << endl;
}

//函数对象概念
void test01(){

    FuncObject01 fobj;
    fobj();
    FuncObject02();

}

class FuncObject03{
public:
    int operator()(int a, int b){
```

```
        return a + b;
    }
};

int FuncObject04(int a,int b){
    return a + b;
}
//函数对象也可以像普通函数一样 具有返回值和参数
void test02(){

    FuncObject03 fobj;
    int ret = fobj(10,20);
    cout << "ret :" << ret << endl;

    ret = FuncObject04(10,20);
    cout << "ret :" << ret << endl;

}
//函数对象超出了普通函数的功能，可以具有保存函数调用状态
//例如 我们要统计函数调用次数

class FuncObject05{
public:
    FuncObject05() :count(0){}
    void operator()(){
        cout << "hello world" << endl;
        count++;
    }
    int count;
};

//普通函数要统计调用次数 需要一个全局变量
int g_count = 0;
void FuncObject06(){
    cout << "hello world" << endl;
    g_count++;
}
void test03(){

    FuncObject06();
    FuncObject06();
    cout << "函数调用次数: " << g_count << endl;

    //使用函数对象 不需要使用全局变量
```

```
    FuncObject05 fobj;
    fobj();
    fobj();
    fobj();

    cout << "函数调用次数：" << fobj.count << endl;

}

//函数对象做参数和返回值
class print{
public:
    print() :count(0){}
    void operator()(const int& val){
        cout << val << " ";
        count++;
    }
    int count;
};
void test04(){

    vector<int> v;
    v.push_back(1);
    v.push_back(3);
    v.push_back(5);
    v.push_back(2);

    //通过 for_each 算法 遍历容器元素
    print myprint;
    //函数对象做返回值和参数
    myprint = for_each(v.begin(), v.end(), myprint);
    cout << endl;

    cout << "函数对象调用次数:" << myprint.count << endl;
}

int main(){
    test01();
    test02();
    test03();
    test04();

    system("pause");
    return EXIT_SUCCESS;
```

```
}
```

3.1.2 谓词概念

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 接受一个参数，那么叫做一元谓词,如果接受两个参数，那么叫做二元谓词，谓词可作为一个判断式。

例如：

```
struct myfuncobj01{
	bool operator(int v){}  //接受一个参数，并且返回值为 Bool 即一元谓词
}
bool compare01(int v); //同样是叫做一元谓词

struct myfuncobj02{
	bool operator(int v1, int v2){}  //接受两个参数，返回值为 Bool 即二元谓词
}
bool compare02(int v1, int v2); //同样是叫做二元谓词
```

一元函数对象 应用举例: for_each

一元谓词 应用举例：find_if

二元函数对象 应用举例: transform

二元谓词 应用举例 ：sort

```
#define _CRT_SECURE_NO_WARNINGS

#include<iostream>
#include<vector>
#include<algorithm>
#include<functional>

using namespace std;

//一元函数对象
class print{
public:
```

```
    void operator()(const int& v){
        cout << v << " ";
    }
};
void test01(){

    vector<int> v;

    v.push_back(1);
    v.push_back(2);
    v.push_back(5);
    v.push_back(3);

    //一元函数对象
    for_each(v.begin(),v.end(), print());
    cout << endl;

}

//一元谓词
class mygreater{
public:
    bool operator()(const int& v){
        return v > 2;
    }
};
void test02(){

    vector<int> v;
    v.push_back(1);
    v.push_back(2);
    v.push_back(5);
    v.push_back(3);

    vector<int>::iterator it = find_if(v.begin(), v.end(), mygreater()); //匿名函数对象
    cout << *it << endl;
}

//二元函数对象
class myplus{
public:
    int operator()(int v1,int v2){
        return v1 + v2;
    }
```

```
};
void test03(){

    vector<int> v1, v2,v3;
    v1.push_back(3);
    v1.push_back(4);
    v1.push_back(5);

    v2.push_back(7);
    v2.push_back(8);
    v2.push_back(2);


    v3.resize(v1.size()+ v2.size()); // 给 v3 开辟空间

    transform(v1.begin(), v1.end(), v2.begin(),v3.begin(), myplus());
    for_each(v3.begin(), v3.end(), print());
    cout << endl;
}

//二元谓词
class mycompare{
public:
    bool operator()(int v1,int v2){
        return v1 > v2;
    }
};
void test04(){

    vector<int> v;
    v.push_back(1);
    v.push_back(2);
    v.push_back(5);
    v.push_back(3);

    sort(v.begin(), v.end(), mycompare());
    for_each(v.begin(), v.end(), print());
}

int main(){

    test01();
    test02();
    test03();
```

```
    test04();

    system("pause");
    return EXIT_SUCCESS;
}
```

3.1.3 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>。

- 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 minute<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
```

使用例子:

```
//使用内建函数对象声明一个对象
plus<int> myPlus;
cout << myPlus(5, 3) << endl;
//使用匿名临时对象
cout << plus<int>()(5, 6) << endl;
sort 排序使用预定义函数对象进行排序。
count_if equal_to 参数绑定
```

3.1.4 函数对象适配器

函数对象适配器是完成一些配接工作，这些配接包括绑定(bind)，否定(negate),以及对一般函数或成员函数的修饰，使其成为函数对象，重点掌握函数对象适配器(红色字体):

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

not1 ： 对一元函数对象取反

not2：对二元函数对象取反

ptr_fun ： 将普通函数修饰成函数对象

mem_fun ： 修饰成员函数

mem_fun_ref ： 修饰成员函数

预定义函数对象

仿函数适配器 bind1st bind2nd

仿函数适配器 not1 not2

仿函数适配器 ptr_fun

成员函数适配器 mem_fun mem_fun_ref

```
#define _CRT_SECURE_NO_WARNINGS

#include<iostream>
#include<functional>
#include<algorithm>
```

```
#include<vector>
#include<string>
using namespace std;

/*
    template<class T> T plus<T>//加法仿函数
    template<class T> T minute<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
*/
//预定义函数对象
class print{
public:
    void operator()(int v){
        cout << v << "    ";
    }
};
void test01(){

    plus<int> myplus; //实例化一个对象
    int ret = myplus(10, 20);
    cout << "ret : " << ret << endl;

    cout << plus<int>()(30, 40) << endl;

    vector<int> v1, v2, v3;
    for (int i = 0; i < 10;i++){
        v1.push_back(i);
        v2.push_back(i + 1);
    }

    v3.resize(v1.size());
    transform(v1.begin(),v1.end(),v2.begin(),v3.begin(),plus<int>());

    for_each(v1.begin(), v1.end(), print());
    cout << endl;

    for_each(v2.begin(), v2.end(), print());
    cout << endl;

    for_each(v3.begin(), v3.end(), print());
    cout << endl;
```

```
}

//函数适配器 bind1st bind2nd
//现在我有这个需求 在遍历容器的时候，我希望将容器中的值全部加上 100 之后显示出来，怎么做哇？
struct myprint : public binary_function<int,int,void>{   //二元函数对象 所以需要继承 binary_fucntion<参数类型,参数类型,返回值类型>
    void operator()(int v1 ,int v2) const{
        cout << v1 + v2 << " ";
    }
};
void test02(){

    vector<int> v;
    v.push_back(1);
    v.push_back(2);
    v.push_back(3);
    v.push_back(4);

    //我们直接给函数对象绑定参数 编译阶段就会报错
    //for_each(v.begin(), v.end(), bind2nd(myprint(),100));
    //如果我们想使用绑定适配器,需要我们自己的函数对象继承 binary_function 或者 unary_function
    //根据我们函数对象是一元函数对象 还是二元函数对象
    for_each(v.begin(), v.end(), bind2nd(myprint(), 100));
    cout << endl;

    //总结：  bind1st 和 bind2nd 区别?
    //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
    //bind1st bind2nd 将二元函数对象转为一元函数对象

}

//函数对象适配器 not1 not2
struct myprint02 {
    void operator()(int v1) const{
        cout << v1 << " ";
    }
};
void test03(){

    vector<int> v;
```

```
    v.push_back(2);
    v.push_back(1);
    v.push_back(5);
    v.push_back(4);

    vector<int>::iterator it =  find_if(v.begin(), v.end(),
not1(bind2nd(less_equal<int>(), 2)));
    cout << "it:" << *it << endl;
    sort(v.begin(),v.end(),not2(greater<int>()));

    for_each(v.begin(), v.end(), myprint02());
    cout << endl;

    //not1 对一元函数对象取反
    //not2 对二元函数对象取反
}

//如何给一个普通函数使用绑定适配器(bind1st bind2nd)绑定一个参数？(拓展)
//ptr_fun
void myprint04(int v1,int v2){
    cout << v1 + v2 << " ";
}
void test04(){

    vector<int> v;
    v.push_back(2);
    v.push_back(1);
    v.push_back(5);
    v.push_back(4);


    //1 将普通函数适配成函数对象
    //2 然后通过绑定器绑定参数
    for_each(v.begin(), v.end(), bind2nd(ptr_fun(myprint04),100));
    cout << endl;

    //总结: ptr_fun 将普通函数转变为函数对象
}

//mem_fun mem_fun_ref
//如果我们容器中存储的是对象或者对象指针，如果能指定某个成员函数处理成员数据。
class student{
public:
    student(string name, int age) :name(name), age(age){}
```

```
    void print(){
        cout << "name:" << name << " age:" << age << endl;;
    }
    void print2(int a){
        cout << "name:" << name << " age:" << age << " a:" << a << endl;
    }
    int age;
    string name;
};
void test05(){


    //mem_fun : 如果存储的是对象指针，需要使用 mem_fun
    vector<student*> v;
    student* s1 = new student("zhaosi",10);
    student* s2 = new student("liuneng", 20);
    student* s3 = new student("shenyang", 30);
    student* s4 = new student("xiaobao", 40);

    v.push_back(s1);
    v.push_back(s2);
    v.push_back(s3);
    v.push_back(s4);

    for_each(v.begin(), v.end(), mem_fun(&student::print));
    cout << "------------------------------" << endl;

    //mem_fun_ref : 如果存储的是对象，需要使用 mem_fun_ref

    vector<student> v2;
    v2.push_back(student("zhaosi",50));
    v2.push_back(student("liuneng", 60));
    v2.push_back(student("shenyang", 70));
    v2.push_back(student("xiaobao", 80));

    for_each(v2.begin(), v2.end(), mem_fun_ref(&student::print));

}


int main(){

    //test01();
    //test02();
```

```
    //test03();
    //test04();
    test05();

    system("pause");
    return EXIT_SUCCESS;
}
```

如果希望函数对象适配器能对我们自己编写的函数对象有效 ,我们需要根据我们的函数对象类型继承 STL 的父类对象。

如果你本身是二元函数对象　需要继承

二元函数继承：public binary_function<参数类型,参数类型,返回类型>

一元函数继承：public unary_function<参数类型,返回类型>

3.2 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,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...

<numeric>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.

<functional> 定义了一些模板类,用以声明函数对象。

STL 算法分为：**质变算法**和**非质变算法**。

所有的 STL 算法都作用在由迭代器[first,end)所标示出来的区间上，所谓质变算法，是指运算过程中会改变区间内的(迭代器所指)的元素内容。比如，拷贝(copy)、互换(swap)、替换(replace)、填写(fill)、删除(remove)、排序(sort)等算法都属于此类。

非质变算法是指是指在运算过程中不会区间内(迭代器所指)的元素内容，比如查找

(find)、计数(count)、遍历(for_each)、寻找极值(max,min)等，都属于此类。但是如果你在 for_each 遍历每个元素的时候试图应用一个会改变元素内容的仿函数，那么元素当然也会改变。

3.3 常用遍历算法

```
/*
    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);
/*
    transform 算法 将指定容器区间元素搬运到另一容器中
    注意 : 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 _cakkback 回调函数或者函数对象
    @return 返回目标容器迭代器
*/
transform(iterator beg1, iterator end1, iterator beg2, _callbakc)
```

3.3.1 for_each 练习代码：

基本正向遍历和逆向遍历
for_each 绑定参数输出
for_each 修改容器元素
for_each 返回值

```
#define _CRT_SECURE_NO_WARNINGS

#include<iostream>
#include<vector>
#include<algorithm>
#include<functional>
```

```
using namespace std;

//for_each 正向遍历 反向遍历
struct print01{
    void operator()(int v){
        cout << v << " ";
    }
};
void test01(){

    vector<int> v;
    for (int i = 0; i < 10;i++){
        v.push_back(rand() % 100);
    }

    //正向遍历
    for_each(v.begin(), v.end(), print01());
    cout << endl;
    //反向遍历
    for_each(v.rbegin(), v.rend(), print01());
    cout << endl;
}


//for_each 算法 绑定参数
//将容器中的元素加上 100 再输出
struct print2 : public binary_function<int,int,void>{
    void operator()(int v1,int v2) const{
        cout << v1+v2 << " ";
    }
};
void print21(int v1, int v2){
    cout << v1 + v2 << " ";
}
void test02(){

    vector<int> v;
    for (int i = 0; i < 10; i++){
        v.push_back(rand() % 100);
    }

    for_each(v.begin(), v.end(), print01());
    cout << endl;
```

```
    //函数对象做参数
    for_each(v.begin(), v.end(), bind2nd(print2(), 100));
    cout << endl;
    //普通回调函数做参数，并且绑定参数
    for_each(v.begin(), v.end(), bind2nd(ptr_fun(print21), 100));
    cout << endl;

}

//for_each 修改元素值
struct print3 {
    void operator()(int& v1) const{
        v1 = v1 + 100;
        cout << v1 << " ";
    }
};
void test03(){

    vector<int> v;
    for (int i = 0; i < 10; i++){
        v.push_back(rand() % 100);
    }

    for_each(v.begin(), v.end(), print01());
    cout << endl;

    for_each(v.begin(), v.end(), print3());
    cout << endl;

    for_each(v.begin(), v.end(), print01());
    cout << endl;
}

//for_each 返回值
struct print4 {
    print4() :count(0){}
    void operator()(int v1){
        count++;
        cout << v1 << " ";
    }
    int count;
};
void test04(){
```

```
    vector<int> v;
    for (int i = 0; i < 10; i++){
        v.push_back(rand() % 100);
    }
    print4 temp1;
    print4 temp2 = for_each(v.begin(), v.end(), temp1);
    cout << endl;

    cout << "temp1:" << temp1.count << endl;
    cout << "temp2:" << temp2.count << endl;

}
int main(){

    //test01();
    test02();
    test03();
    test04();

    system("pause");
    return EXIT_SUCCESS;
}
```

3.3.2 transform 练习代码：

从一个容器经过处理搬运到另一个容器:
两个容器数据处理搬运到第三个容器

```
#define _CRT_SECURE_NO_WARNINGS

#include<iostream>
#include<vector>
#include<algorithm>
#include<functional>

using namespace std;

//容器中元素加 10 搬运到另一容器中
void print1(int v){
    cout << v << " ";
}
struct myplus01{
```

```
    int operator()(int v1){
        return v1 + 100;
    }
};
void test01(){

    vector<int> v,dest;
    for (int i = 0; i < 10; i++){
        v.push_back(rand() % 100);
    }
    for_each(v.begin(), v.end(), print1);
    cout << endl;
    //首先给 dest 开辟足够内存
    dest.resize(v.size());
    //搬运元素

    /*
        template<class _InIt,
        class _OutIt,
        class _Fn1> inline
        _OutIt _Transform(_InIt _First, _InIt _Last,
        _OutIt _Dest, _Fn1 _Func)
        {   // transform [_First, _Last) with _Func
        for (; _First != _Last; ++_First, ++_Dest)
        *_Dest = _Func(*_First);
        return (_Dest);
        }
    */
    transform(v.begin(), v.end(), dest.begin(), myplus01());
    for_each(dest.begin(), dest.end(), print1);
    cout << endl;

}

//容器 1 的元素 + 容器 2 的元素 搬运到 第三个容器中
struct myplus02{
    int operator()(int v1,int v2){
        return v1 + v2;
    }
};
void test02(){

    vector<int> v1,v2, dest;
    for (int i = 0; i < 10; i++){
```

```
        v1.push_back(i);
        v2.push_back(i + 1);
    }
    for_each(v1.begin(), v1.end(), print1);
    cout << endl;
    /*
        template<class _InIt1,
        class _InIt2,
        class _OutIt,
        class _Fn2> inline
        _OutIt transform(_InIt1 _First1, _InIt1 _Last1,
        _InIt2 _First2, _OutIt _Dest, _Fn2 _Func)
        {   // transform [_First1, _Last1) and [_First2, ...) with _Func
        _DEBUG_RANGE(_First1, _Last1);
        _DEBUG_POINTER(_Dest);
        _DEBUG_POINTER(_Func);
        if (_First1 != _Last1)
        return (_Transform2(_Unchecked(_First1), _Unchecked(_Last1),
        _First2, _Dest, _Func,
        _Is_checked(_Dest)));
        return (_Dest);
        }


        template<class _InIt1,
        class _InIt2,
        class _OutIt,
        class _Fn2> inline
        _OutIt _Transform(_InIt1 _First1, _InIt1 _Last1,
        _InIt2 _First2, _OutIt _Dest, _Fn2 _Func)
        {   // transform [_First1, _Last1) and [_First2, ...) with _Func
        for (; _First1 != _Last1; ++_First1, ++_First2, ++_Dest)
        *_Dest = _Func(*_First1, *_First2);
        return (_Dest);
        }

    */

    dest.resize(v1.size());
    transform(v1.begin(), v1.end(), v2.begin(), dest.begin(), myplus02());

    for_each(dest.begin(), dest.end(), print1);
    cout << endl;
}
```

```
int main(){

    //test01();
    test02();

    system("pause");
    return EXIT_SUCCESS;
}
```

3.4 常用查找算法

```
/*
    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)
/*
    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);
/*
    binary_search 算法 二分查找法
    注意: 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 false
*/
bool binary_search(iterator beg, iterator end, value);
/*
    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);
/*
    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 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(iterator beg, iterator end, value);
/*
    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  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);
```

3.4.1 find 算法案例

find 查找基本数据类型，类对象，类指针:

```
#define _CRT_SECURE_NO_WARNINGS

#include<iostream>
#include<vector>
#include<algorithm>
#include<functional>

using namespace std;


//find 算法
void test01(){

    int arr[] = {5,2,8,9,1,3};
    vector<int> v(arr, arr + sizeof(arr) / sizeof(int));

    /*
        template<class _InIt,
```

```
        class _Ty> inline
        _InIt find(_InIt _First, _InIt _Last, const _Ty& _Val)
        {   // find first matching _Val
        _DEBUG_RANGE(_First, _Last);
        return (_Rechecked(_First,
            _Find(_Unchecked(_First), _Unchecked(_Last), _Val)));
        }

    */

    //这里注意 find 返回值 如果没有找到 返回 v.end()
    vector<int>::iterator it =  find(v.begin(),v.end(),1);
    //可以这样判断是否找到元素
    if (it == v.end()){
        cout << "没有找到!" << endl;
    }
    else{
        cout << *it << endl;
    }

}

//find 查找对象
class sutdent{
public:
    sutdent(int age, int salary) :age(age), salary(salary){}
    int age;
    int salary;

    bool operator==(const sutdent& stu){
        if (this->age == stu.age && this->salary == stu.salary){
            return true;
        }
        else{
            return false;
        }
    }
};
void test02(){

    //对象查找  重载==操作符
    sutdent s1(1, 2), s2(3, 4), s3(5, 6);
    vector<sutdent> vs;
    vs.push_back(s1);
```

```
    vs.push_back(s2);
    vs.push_back(s3);

    vector<sutdent>::iterator its = find(vs.begin(), vs.end(), s2);
    if (its == vs.end()){
        cout << "s2 没有找到！" << endl;
    }
    else{
        cout << "s2 找到！" << endl;
    }

}


int main(){

    //test01();
    test02();

    system("pause");
    return EXIT_SUCCESS;
}
```

3.5 常用排序算法

```
/*
    merge 算法 容器元素合并，并存储到另一容器中
    @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)
/*
    sort 算法 容器元素排序
    注意:两个容器必须是有序的
    @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)
/*
    sort 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)
/*
    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)
```

3.6 常用拷贝和替换算法

```
/*
    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 目标容器结束迭代器
*/
copy(iterator beg, iterator end, iterator dest)
/*
    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 oldvalue 新元素
*/
replace(iterator beg, iterator end, oldvalue, newvalue)
/*
    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 oldvalue 新元素
*/
```

```
replace_if(iterator beg, iterator end, _callback, newvalue)
/*
    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)
```

3.7 常用算数生成算法

```
/*
    accumulate 算法 计算容器元素累计总和
    @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)
/*
    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 t 填充元素
*/
fill(iterator beg, iterator end, value)
```

3.8 常用集合算法

```
/*
    set_intersection 算法 求两个 set 集合的交集
    注意: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)
/*
    set_union 算法 求两个 set 集合的并集
    注意: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_union(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest)
/*
    set_difference 算法 求两个 set 集合的差集
    注意: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_difference(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest)
```

第四章 STL 综合案例

4.1 案例学校演讲比赛

4.2 学校演讲比赛介绍

1）某市举行一场演讲比赛（ speech_contest ），共有 24 个人参加。比赛共三轮，前两轮为淘汰赛，第三轮为决赛。

2）比赛方式：分组比赛，每组 6 个人；选手每次要随机分组，进行比赛；

第一轮分为 4 个小组，每组 6 个人。比如 100-105 为一组，106-111 为第二组，依次类推，

每人分别按照抽签（draw）顺序演讲。当小组演讲完后，淘汰组内排名最后的三个选手，然后继续下一个小组的比赛。

第二轮分为 2 个小组，每组 6 人。比赛完毕，淘汰组内排名最后的三个选手，然后继续下一个小组的比赛。

第三轮只剩下 6 个人，本轮为决赛，选出前三名。

4）比赛评分：10 个评委打分，去除最低、最高分，求平均分

每个选手演讲完由 10 个评委分别打分。该选手的最终得分是去掉一个最高分和一个最低分，求得剩下的 8 个成绩的平均分。

选手的名次按得分降序排列，若得分一样，按参赛号升序排名。

用 STL 编程，求解这个问题
1） 请打印出所有选手的名字与参赛号，并以参赛号的升序排列。
2） 打印每一轮比赛后，小组比赛成绩和小组晋级名单
3） 打印决赛前三名，选手名称、成绩。

4.3 需求分析

```
//产生选手 （ ABCDEFGHIJKLMNOPQRSTUVWXYZ ） 姓名、得分；选手编号
//第 1 轮 选手抽签 选手比赛 查看比赛结果
//第 2 轮 选手抽签 选手比赛 查看比赛结果
//第 3 轮 选手抽签 选手比赛 查看比赛结果
```

4.4 实现思路

需要把选手信息、选手得分信息、选手比赛抽签信息、选手的晋级信息保存在容器中，需要涉及到各个容器的选型。（相当于信息的数据库 E-R 图设计）

选手可以设计一个类 Speaker（姓名和得分）
所有选手编号和选手信息，可以放在容器内：map<int, Speaker>
所有选手的编号信息，可以放在容器：vecter<int> v1 中
第 1 轮晋级名单，可以放在容器 vecter<int> v2 中
第 2 轮晋级名单，可以放在容器 vecter<int> v3 中
第 3 轮前三名名单，可以放在容器 vecter<int> v4 中

每个小组的比赛得分信息，按照从小到大的顺序放在
multimap<成绩, 编号, greater<int>> multmapGroup
也就是：multimap<int, int, greater<int> > multmapGroup;
每个选手的得分，可以放在容器 deque<int> dscore; 方便去除最低最高分

4.5 实现细节

1） 搭建框架

2） 完善业务函数
random_shuffle

3） 测试

```
void main()
{
    //定义数据结构 所有选手放到容器中
    map<int, Speaker> mapSpeaker;
```

```
    vector<int>                 v1; //第 1 轮演讲比赛  名单
    vector<int>                 v2; //第 2 轮演讲比赛  名单
    vector<int>                 v3; //第 3 轮演讲比赛  名单
    vector<int>                 v4; //最后  演讲比赛  名单

    //产生选手
    GenSpeaker(mapSpeaker, v1);

    //第 1 轮  选手抽签  选手比赛  查看比赛结果(晋级名单  得分情况)
    cout << "\n\n\n 任意键,开始第一轮比赛" << endl;
    cin.get();
    speech_contest_draw(v1);
    speech_contest(0,    v1, mapSpeaker, v2);
    speech_contest_print(0, v2, mapSpeaker);

    //第 2 轮  选手抽签  选手比赛  查看比赛结果
    cout << "\n\n\n 任意键,开始第二轮比赛" << endl;
    cin.get();
    speech_contest_draw(v2);
    speech_contest(1,    v2, mapSpeaker, v3);
    speech_contest_print(1, v3, mapSpeaker);

    //第 3 轮  选手抽签  选手比赛  查看比赛结果
    cout << "\n\n\n 任意键,开始第三轮比赛" << endl;
    cin.get();
    speech_contest_draw(v3);
    speech_contest(2,    v3, mapSpeaker, v4);
    speech_contest_print(2, v4, mapSpeaker);

    system("pause");
}

//产生选手
int GenSpeaker(map<int, Speaker> &mapSpeaker, vector<int> &v1)

//选手抽签
int speech_contest_draw(vector<int>    &v)

//选手比赛
int speech_contest(int index,    vector<int> &v1, map<int, Speaker> &mapSpeaker, vector<int>
&v2)

//打印选手比赛晋级名单
int speech_contest_print(int index, vector<int> v, map<int, Speaker> & mapSpeaker)
```