C++提高编程

——泛型编程和STL技术

1.模板

1.1 模板的概念

模板：通用的模具，大大提高复用性

模板的特点：

- 模板不可以直接使用，只是一个框架
- 模板的通用不是万能的

1.2 函数模板

C++的另一种编程思想为**泛型编程**，主要利用的技术是模板

C++提供两种模板机制：**函数模板**和**类模板**

1.2.1函数模板语法

函数模板作用：

建立一个通用函数，其函数返回值类型和形参类型可以不确定，用一个**虚拟的类型**来代表

语法：

```
template<typename T>
函数声明或定义
```

解释：

template——声明创建模板

typename——表明其后面的符号是一种数据类型，可以用class代替

T——通用的数据类型，名称可以替换，通常是大写字母

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

//函数模板
//交换函数模板
template <typename T>
void mySwap(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}
void test01() {
    //函数模板调用
    int a = 10;
    int b = 20;
```

```
    //1、自动类型推导
    mySwap(a, b);
    //2、显示指定类型
    mySwap<int>(a, b);
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

1.2.2 函数模板的注意事项

- 自动类型推导，必须推导出一致的数据类型T，才可以正常使用
- 模板必须确定出T的数据类型，才可以正常使用

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

//函数模板注意事项
//交换函数模板
template <typename T>
void mySwap(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}
void test01() {
    int a = 10;
    int b = 20;
    char c = 'c';
    //1、自动类型推导必须推导出一致的数据类型T才能正常使用
    //mySwap(a, c);错误
    mySwap(a, b);
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
}
//2、函数模板必须确定数据类型T才可以正常使用
template <typename T>
void func() {
    cout << "func()函数的调用" << endl;
}
void test02() {
    func<int>();//显示指定类型
}
int main()
{
    test01();

    system("pause");
    return 0;
}
```

1.2.3 函数模板案例

- 不同数据类型数组排序
- 选择排序

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

template <typename T>
void mySwap(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}
//排序算法
template <typename T>
void mySort(T arr[], int len) {
    for (int i = 0; i < len; i++) {
        max = i;//假定的最大值
        for (int j = i + 1; j < len; j++) {
            if (arr[j] > arr[max]) {
                max = j;
            }
        }
        if (max != i) {
            mySwap(arr[max], arr[i]);
        }
    }
}
template <typename T>
void myPrint(T arr[], int len) {
    for (int i = 0; i < len; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}
void test01() {
    //测试char数组
    char charArr[] = "bshdfifsfd";
    int num = sizeof(charArr) / sizeof(char);
    mySort(charArr, num);
    myPrint(charArr, num);
    //测试int数组
    int intArr[] = { 2,5,4,8,6,7,3 };
    int Num = sizeof(intArr) / sizeof(int);
    mySort(intArr, Num);
    myPrint(intArr, Num);
}
int main() {

    test01();
    system("pause");
    return 0;
}
```

1.2.4 普通函数和函数模板的区别

区别：

- 普通函数调用时可以发生自动类型转换（隐式类型转换）
- 函数模板调用时，如果利用自动类型推导，不会发生隐式类型转换
- 如果利用显示指定类型的方式，可以发生隐式类型转换

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

//普通函数和函数模板的区别
int myAdd01(int a,int b) {
    return a + b;
}
template <class T>
T myAdd02(T a, T b) {
    return a + b;
}
void test01() {
    int a = 10;
    int b = 20;
    char c = 'c';
    cout << myAdd01(a, c) << endl;//普通函数可以发生隐式类型转换
    //自动类型推导
    //cout << myAdd02(a, c) << endl;//不会发生隐式类型转换
    //显示指定类型
    cout << myAdd02<int>(a, c) << endl;//发生自动类型转换
}
int main() {

    test01();
    system("pause");
    return 0;
}
```

总结：建议使用显示指定类型的方式调用函数模板

1.2.5 普通函数和函数模板的调用规则

调用规则：

- 如果函数模板和普通函数都可以实现，优先调用普通函数
- 可以通过空模板参数列表来强制调用函数模板
- 函数模板也可以发生重载
- 如果函数模板可以产生更好的匹配，优先调用函数模板

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

//普通函数和函数模板的调用规则
void myPrint(int a, int b) {
    cout << "普通函数的调用" << endl;
}
template <typename T>
void myPrint(T a, T b) {
    cout << "函数模板调用" << endl;
```

```
}
//函数模板重载
template <typename T>
void myPrint(T a, T b,T c) {
    cout << "重载函数模板调用" << endl;
}
void test01() {
    int a = 10;
    int b = 20;
    char c1 = 'a';
    char c2 = 'b';
    myPrint(a, b);
    //空模板的参数列表强制调用函数模板
    myPrint<>(a, b);
    myPrint(a, b, 100);
    //更好的匹配
    myPrint(c1, c2);
}
int main() {

    test01();
    system("pause");
    return 0;
}
```

1.2.6 模板的局限性

- 模板的通用性不是万能的
    - 特殊的数据类型

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

class Person {
public:
    Person(string name, int age) {
        this->m_Age = age;
        this->m_Name = name;
    }
    int m_Age;
    string m_Name;
};
//模板的局限性
template <typename T>
bool myCompare(T& a, T& b) {
    if (a == b) {
        return true;
    }
    return false;
}
//具体化版本的函数模板
template<>bool myCompare(Person& p1, Person& p2) {
    if (p1.m_Name == p2.m_Name && p1.m_Age == p2.m_Age) {
        return true;
    }
    return false;
```

```
}
void test01() {
    int a = 10;
    int b = 20;
    bool ret = myCompare(a, b);
    if (ret) {
        cout << "a == b" << endl;
    }
    else {
        cout << "a != b" << endl;
    }
}
void test02() {
    Person p1("Tom", 18);
    Person p2("Tom", 18);
    bool ret = myCompare(p1, p2);
    if (ret) {
        cout << "p1 == p2" << endl;
    }
    else {
        cout << "p1 != p2" << endl;
    }
}
int main() {

    test01();
    test02();
    system("pause");
    return 0;
}
```

1.3 类模板

1.3.1类模板语法

语法：

```
template <typename T>
类
```

解释：

template——声明创建模板

typename——表明其后面的符号是一种数据类型，可以用class代替

T——通用的数据类型，名称可以替换，通常是大写字母

```
#include <iostream>
using namespace std;
#include <string>
//类模板
template<class NameType,class AgeType>
class Person {
public:
    Person(NameType name, AgeType age) {
        this->m_Age = age;
```

```
        this->m_Name = name;
    }
    void showPerson() {
        cout << "name:" << this->m_Name << endl;
        cout << "age:" << this->m_Age << endl;
    }
    NameType m_Name;
    AgeType m_Age;
};
void test01() {
    Person<string, int>p1("Tom", 18);
    p1.showPerson();
}
int main() {
    test01();
    system("pause");
    return 0;
}
```

1.3.2 类模板与函数模板区别

区别：

- 类模板没有自动推导的使用方式
- 类模板在模板参数列表中可以有默认参数

```
#include <iostream>
using namespace std;
#include <string>
//类模板与函数模板的区别
template<class NameType,class AgeType = int>//类模板的模板参数列表中可以有默认参数
class Person {
public:
    Person(NameType name, AgeType age) {
        this->m_Age = age;
        this->m_Name = name;
    }
    void showPerson() {
        cout << "name:" << this->m_Name << endl;
        cout << "age:" << this->m_Age << endl;
    }
    NameType m_Name;
    AgeType m_Age;
};
void test01() {
    //Person p("Tom", 18);类模板没有自动类型推导
    Person<string, int>p("Tom", 18);//显示指定类型
    p.showPerson();
}
void test02() {
    Person <string>p("Jerry", 18);
    p.showPerson();
}
int main() {
    test01();
    test02();
    system("pause");
```

```
    return 0;
}
```

1.3.3 类模板中成员函数的创建时机

- 类模板中成员函数在调用时创建
- 普通类中的成员函数一开始就创建了

```
#include <iostream>
using namespace std;
#include <string>
//类模板中成员函数的创建时机
class Person1 {
public:
    void showPerson1() {
        cout << "Person1 show" << endl;
    }
};
class Person2 {
public:
    void showPerson2() {
        cout << "Person2 show" << endl;
    }
};
template<class T>
class MyClass {
public:
    T obj;
    //类模板中的成员函数
    void func1() {
        obj.showPerson1();
    }
    void func2() {
        obj.showPerson2();
    }
};
void test01() {
    MyClass <Person1>m;
    m.func1();
    //m.func2();调用时创建,
    MyClass<Person2>n;
    n.func2();
    //n.func1();调用时创建
}
int main() {
    test01();
    system("pause");
    return 0;
}
```

1.3.4 类模板对象做函数参数

- 类模板实例化出的对象，向函数传参的方式

传入方式:

- 指定传入的类型——直接显示对象的数据类型

- 参数模板化——将对象中的参数变为模板进行传递
- 整个类模板化——将这个对象类型模板化进行传递

```cpp
#include <iostream>
using namespace std;
#include <string>
//类模板对象做函数参数
template<class NameType, class AgeType>
class Person {
public:
    Person(NameType name, AgeType age) {
        this->m_Name = name;
        this->m_Age = age;
    }
    void showPerson() {
        cout << "name:" << this->m_Name << endl;
        cout << "age:" << this->m_Age << endl;
    }
    NameType m_Name;
    AgeType m_Age;
};
//1、指定传入类型
void printPerson1(Person <string, int>& p){
    p.showPerson();
}
//2、参数模板化
template<class NameType, class AgeType>
void printPerson2(Person<NameType, AgeType>& p) {
    p.showPerson();
    //模板类型
    cout << "NameType = " << typeid(NameType).name() << endl;
    cout << "AgeType = " << typeid(AgeType).name() << endl;
}
//3、整个类模板化
template<class T>
void printPerson3(T& p) {
    p.showPerson();
    cout << "T = " << typeid(T).name() << endl;
}
void test01() {
    Person<string, int>p("Tom", 18);
    printPerson1(p);
    printPerson2(p);
    printPerson3(p);
}
int main() {
    test01();
    system("pause");
    return 0;
}
```

1.3.5 类模板与继承

注意事项

- 当子类继承的父类是一个类模板时，子类在声明是，需要指出父类中T的类型
- 如果不指定，编译器无法给子类分配内存
- 如果想灵活指出父类中的T的类型，子类也需要变为类模板

```
#include <iostream>
using namespace std;
#include <string>
//类模板与继承
template<class T>
class Base {
public:
    T m;
};
class Son1 :public Base<int> {};//指定父类的类型
//灵活指定父类中的类型---子类写成类模板
template<class T1, class T2>
class Son2:public Base<T2>{
public:
    Son2() {
        cout << "T1 = " << typeid(T1).name() << endl;
        cout << "T2 = " << typeid(T2).name() << endl;
    }
    T1 obj;
};
void test01() {
    //Son1 s1;
    Son2<int, char>s2;
}
int main() {
    test01();
    system("pause");
    return 0;
}
```

1.3.6 类模板成员函数类外实现

```
#include <iostream>
#include <string>
using namespace std;
//类模板中成员函数类外实现
template<class T1,class T2>
class Person {
public:
    Person(T1 name, T2 age);
        /*{
            this->m_Age = age;
            this->m_Name = name;

        }*/
        void showPerson();
    /*{
        cout << "name:" << this->m_Name << endl;
        cout << "age:" << this->m_Age << endl;
```

```
    }*/
    T1 m_Name;
    T2 m_Age;
};
//构造函数的类外实现
template<class T1, class T2>
Person<T1,T2>::Person(T1 name, T2 age) {
    this->m_Age = age;
    this->m_Name = name;
}
//成员函数的类外实现
template<class T1, class T2>
void Person<T1, T2>::showPerson() {
    cout << "name:" << this->m_Name << endl;
    cout << "age:" << this->m_Age << endl;
}
void test01() {
    Person<string, int>p("Tom", 18);
    p.showPerson();
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

总结：类模板在类外实现时，需要加上模板的参数列表

1.3.7 类模板分文件编写

问题：

- 类模板中成员函数创建时机是在调用阶段，导致分文件编写调用不到

解决：

- 解决方式1：直接包含.cpp文件
- 解决方式2：将声明和实现写到同一个文件中，并更改后缀名为.hpp，.hpp是约定的名称，不是强制

```
///person.hpp
#pragma once
#include <iostream>
using namespace std;
#include <string>
template<class T1, class T2>
class Person {
public:
    Person(T1 name, T2 age);
    void showPerson();
    T1 m_Name;
    T2 m_Age;
};
//#include "person.h"
template<class T1, class T2>
Person<T1, T2>::Person(T1 name, T2 age) {
    this->m_Name = name;
```

```
    this->m_Age = age;
}
template<class T1, class T2>
void Person<T1, T2>::showPerson() {
    cout << "name: " << this->m_Name << endl;
    cout << "age: " << this->m_Age << endl;
}
```

```
//main.cpp
#include <iostream>
#include <string>
using namespace std;
//类模板分文件编写
//2、将.h和.cpp中的内容写到一起，将后缀名改为.hpp文件(主流)
#include "person.hpp"
void test01() {
    Person<string, int>p("Tom", 18);
    p.showPerson();
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

1.3.8 类模板与友元

- 全局函数类内实现，直接在内类声明友元即可
- 全局函数类外实现，需要提前让编译器知道全局函数的存在

```
#include <iostream>
#include <string>
using namespace std;
//类模板和友元
template<class T1,class T2>
class Person;            //提前让编译器知道
//类外实现
template<class T1, class T2>
void printPerson2(Person<T1, T2>p) {
    cout << "类外实现：" << endl;
    cout << "name: " << p.m_Name << endl;
    cout << "age: " << p.m_Age << endl;
}
//通过全局函数
template<class T1,class T2>
class Person {
    //全局函数类内实现
    friend void printPerson(Person<T1, T2>p) {
        cout << "类内实现：" << endl;
        cout << "name: " << p.m_Name << endl;
        cout << "age: " << p.m_Age << endl;
    }
    //全局函数类外实现
    //加一个空模板的参数列表
    //如果全局函数是类外实现，需要让编译器提前知道这个函数的存在
```

```
    friend void printPerson2<>(Person<T1, T2>p);

public:
    Person(T1 name, T2 age) {
        this->m_Name = name;
        this->m_Age = age;
    }
private:
    T1 m_Name;
    T2 m_Age;
};
void test01() {
    Person<string, int>p1("Tom", 18);
    printPerson(p1);
    Person<string, int>p2("Jerry", 18);
    printPerson2(p2);
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

1.3.9 类模板案例

案例描述：实现一个通用的数组类，要求如下：

- 可以对内置数据类型以及自定义数据类型的数据进行存储
- 将数组中的数据存储到堆区
- 构造函数中可以传入数组的容量
- 提供对应的拷贝构造函数以及operator=防止浅拷贝问题
- 提供尾插法和尾删法对数组中的数据进行增加和删除
- 可以通过下标的方式访问数组中的元素
- 可以获取数组中当前元素个数和数组的容量

```
//MyArray.hpp
//通用的数组类
#pragma once
#include <iostream>
using namespace std;

template<class T>
class MyArray {
public:
    MyArray(int capacity) {
        //cout << "MyArray有参构造函数调用" << endl;
        this->m_Capacity = capacity;
        this->m_Size = 0;
```

```
        this->pAddress = new T[this->m_Capacity];
    }

    MyArray(const MyArray& arr) {
        //cout << "MyArray拷贝构造函数调用" << endl;
        this->m_Capacity = arr.m_Capacity;
        this->m_Size = arr.m_Size;
        this->pAddress = new T[arr.m_Capacity];//深拷贝
        //拷贝arr中的数据
        for (int i = 0; i < this->m_Size; i++) {
            this->pAddress[i] = arr.pAddress[i];
        }
    }

    MyArray& operator=(const MyArray& arr) {
        //cout << "MyArray&operator函数调用" << endl;
        //判断原来堆区是否有数据，如果有先释放
        if (this->pAddress != NULL) {
            delete[] this->pAddress;
            this->pAddress = NULL;
            this->m_Capacity = 0;
            this->m_Size = 0;
        }
        this->m_Capacity = arr.m_Capacity;
        this->m_Size = arr.m_Size;
        this->pAddress = new T[arr.m_Capacity];
        for (int i = 0; i < this->m_Size; i++) {
            this->pAddress[i] = arr.pAddress[i];
        }
        return *this;
    }
    //尾插法
    void Push_Back(const T& value) {
        //判断容量是否等于大小
        if (this->m_Capacity == this->m_Size) {
            cout << "容量已满，无法插入" << endl;
            return;
        }
        this->pAddress[this->m_Size] = value;//数组末尾插入数据
        this->m_Size++;//更新数组大小
    }
    //尾删法
    void Pop_Back() {
        //让用户访问不到最后一个元素，逻辑删除
        if (this->m_Size == 0) {
            cout << "数组为空，无法删除" << endl;
            return;
        }
        this->m_Size--;
    }
    //通过下标方式访问数组中元素,作为左值存在返回
    T& operator[](int index) {
        return this->pAddress[index];
    }

    //返回数组的容量
    int getCapacity() { return this->m_Capacity; }
    //返回数组大小
```

```
    int getSize() { return this->m_Size; }
    ~MyArray() {
        if (this->pAddress != NULL) {
            //cout << "MyArray析构函数调用" << endl;
            delete[]this->pAddress;
            this->pAddress = NULL;
        }
    }
private:
    T* pAddress;//指针指向堆区开辟的真实数组
    int m_Capacity;//容量
    int m_Size;
};
```

```
//main.cpp
#include<iostream>
#include <string>
using namespace std;
#include "MyArray.hpp"

void printIntArray(MyArray<int>& arr) {
    for (int i = 0; i < arr.getSize(); i++) {
        cout << arr[i] << endl;
    }
}
void test01() {
    MyArray<int> arr1(5);
    for (int i = 0; i < 5; i++) {
        arr1.Push_Back(1);//尾插法想数组中插入数据
    }
    cout << "arr1的打印输出为：" << endl;
    printIntArray(arr1);
    cout << "arr1的容量 = " << arr1.getCapacity()  << endl;
    cout << "arr1的大小 = " << arr1.getSize() << endl;

    MyArray <int>arr2(arr1);
    cout << "arr2的打印输出为：" << endl;
    printIntArray(arr2);
    //尾删
    arr2.Pop_Back();
    cout << "arr2尾删后：" << endl;
    cout << "arr2的容量 = " << arr2.getCapacity() << endl;
    cout << "arr2的大小 = " << arr2.getSize() << endl;
}
//测试自定义数据类型
class Person {
public:
    Person() {}
    Person(string name, int age) {
        this->m_Name = name;
        this->m_Age = age;
    }
    string m_Name;
    int m_Age;
};
void printPerson(MyArray<Person>& arr) {
    for (int i = 0; i < arr.getSize(); i++) {
```

```
        cout << "name:" << arr[i].m_Name << endl;
        cout << "age:" << arr[i].m_Age << endl;
    }
}
void test02() {
    MyArray<Person>arr(10);
    Person p1("Tom", 18);
    Person p2("Jerry",20);
    Person p3("Nancy",20);
    Person p4("Jack", 20);
    Person p5("Jonh", 20);
    arr.Push_Back(p1);
    arr.Push_Back(p2);
    arr.Push_Back(p3);
    arr.Push_Back(p4);
    arr.Push_Back(p5);
    printPerson(arr);
    cout << "arr的容量 = " << arr.getCapacity() << endl;
    cout << "arr的大小 = " << arr.getSize() << endl;
}
int main() {
    test01();
    test02();
    system("pause");
    return 0;
}
```

2 STL初识

2.1 STL的诞生

- 长久以来，软件界一直希望建立一种重复利用的东西
- C++**面向对象**和**泛型编程**思想，目的就是为了**复用性的提升**
- 大多情况下，数据结构和算法都未能有一套标准，导致被迫从事大量重复工作
- 为了建立数据结构和算法的一套标准，诞生了STL

2.2 STL基本概念

- STL（Standard Template Library,**标准模板库**）
- STL从广义上分为：**容器（container）算法（algorithm）迭代器（iterator）**
- **容器**和**算法**之间通过**迭代器**进行无缝连接
- STL几乎所有的代码都采用了模板类或者模板函数

2.3 STL六大组件

STL大体分为六大组件

- **容器**：各种数据结构，如vector、list、deque、set、map等，用来存放数据
- **算法**：各种常用的算法，如sort、find、copy、for_each等
- **迭代器**：扮演了容器和算法之间的胶合剂
- **仿函数**：行为类似函数，可作为算法的某种策略
- **适配器（配接器）**：一种用来修饰容器或者仿函数或者迭代器接口的东西
- **空间配置器**：负责空间的配置和管理

2.4 STL中容器、算法、迭代器

容器：置物之所也

STL**容器**就是将运用**最广泛的一些数据结构**实现出来

常用的数据结构：数组、链表、树、栈、队列、集合、映射表等

容器分类：

- **序列式容器**：强调值的排序，序列式容器中的每一个元素均有一个固定的位置
- **关联式容器**：二叉树结构，各元素之间没有严格的物理上的顺序关系

算法：问题之解法也

有限的步骤，解决逻辑或者数学上的问题，这一门学科——算法（Algorithms）

算法分类：

- **质变算法**：指运算过程中会更改区间内的元素的内容，例如拷贝、替换、删除等
- **非质变算法**：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

迭代器：容器和算法之间粘合剂

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，而又无需暴露改容器内部表达方式

每个容器都有自己专属的迭代器

迭代器使用非常类似于指针

迭代器种类：

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

常用的容器中迭代器种类为：双向迭代器和随机访问迭代器

2.5 容器算法迭代器处事

STL中最常用的容器为Vector，可以理解为数组

2.5.1 vector存放内置数据类型

容器：**vector**

算法：**for_each**

迭代器：**vector::iterator**

```
#include<iostream>
using namespace std;
#include<vector>
#include<algorithm>//标准算法头文件
//vector容器存放内置的数据类型
```

```cpp
void myPrint(int value) {
    cout << value << endl;
}
void test01() {
    //创建一个vector容器
    vector<int>v;
    //向容器中插入数据
    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    v.push_back(40);
    //通过迭代器访问容器中的数据
    vector<int>::iterator itBegin = v.begin();//起始迭代器，指向容器中第一个元素
    vector<int>::iterator itEnd = v.end();//结束迭代器，指向容器中最后一个元素的下一个位置
    //第一种遍历方式
    while (itBegin != itEnd) {
        cout << *itBegin << endl;
        itBegin++;
    }
    //第二种遍历
    for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
        cout << *it << endl;
    }
    //第三种遍历方式，利用STL提供的遍历算法
    for_each(v.begin(), v.end(), myPrint);//回调
}

int main()
{
    test01();
    system("pause");
    return 0;
}
```

2.5.2 Vector存放自定义数据类型

```cpp
#include <iostream>
using namespace std;
#include <string>
#include<vector>
//vector容器中存放自定义数据类型
class Person {
public:
    Person(string name, int age) {
        this->m_Name = name;
        this->m_Age = age;
    }

    string m_Name;
    int m_Age;
};
void test01() {
    vector<Person>v;
    Person p1("Tom", 18);
    Person p2("Jerry", 20);
    Person p3("Nancy", 20);
```

```
    Person p4("Jack", 20);
    Person p5("Jonh", 20);
    //向容器中添加数据
    v.push_back(p1);
    v.push_back(p2);
    v.push_back(p3);
    v.push_back(p4);
    v.push_back(p5);
    //遍历容器中的数据
    for (vector<Person>::iterator it = v.begin(); it != v.end(); it++) {
        /*cout << "name: " << (*it).m_Name << "\t"
            << "age: " << (*it).m_Age << endl;*/
        cout << "name: " << it->m_Name << "\t"
            << "age: " << it->m_Age << endl;
    }
}
void test02() {
    vector<Person*>v;
    Person p1("Tom", 18);
    Person p2("Jerry", 20);
    Person p3("Nancy", 20);
    Person p4("Jack", 20);
    Person p5("Jonh", 20);
    //向容器中添加数据
    v.push_back(&p1);
    v.push_back(&p2);
    v.push_back(&p3);
    v.push_back(&p4);
    v.push_back(&p5);
    //遍历容器
    for (vector<Person*>::iterator it = v.begin(); it != v.end(); it++) {
        cout << "name: " << (*it)->m_Name << "\t"
            << "age: " << (*it)->m_Age << endl;
    }
}
int main()
{
    //test01();
    test02();
    system("pause");
    return 0;
}
```

2.5.3 Vector容器嵌套容器

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

//容器嵌套容器
void test01() {
    vector<vector<int>>v;
    //创建小容器
    vector<int>v1;
    vector<int>v2;
    vector<int>v3;
    vector<int>v4;
```

```
    for (int i = 0; i < 4; i++) {
        v1.push_back(i + 1);
        v2.push_back(i + 2);
        v3.push_back(i + 3);
        v4.push_back(i + 4);
    }
    //放入大容器
    v.push_back(v1);
    v.push_back(v2);
    v.push_back(v3);
    v.push_back(v4);
    //遍历大容器
    for (vector<vector<int>>::iterator it = v.begin(); it != v.end(); it++)
{
        for (vector<int>::iterator vit = (*it).begin(); vit != (*it).end();
vit++) {
            cout << *vit << " ";
        }
        cout << endl;
    }
}
int main() {
    test01();
    system("pause");
    return 0;
}
```

3 STL常用容器

3.1 string容器

3.1.1 string基本概念

本质：

- string是C++风格的字符串，而string本质上是一个类

string和char*的区别：

- char*是一个指针
- string是一个类，内部封装了char*，管理这个字符串，是一个char*型的容器

特点：

string类内部封装了很多成员方法

例如：查找find、拷贝copy、删除delete、替换replace、插入insert

string管理char*所分配的内存，不用担心复制越界和取值越界等，由类内部进行负责

3.1.2 string构造函数

构造函数原型：

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

```cpp
#include <iostream>
using namespace std;
#include <string>

//string构造函数
void test01() {
    string s1;//默认构造

    const char* str = "Hello World";
    string s2(str);
    cout << "s2 = " << s2 << endl;

    string s3(s2);
    cout << "s3 = " << s3 << endl;

    string s4(10, 'a');
    cout << "s4 = " << s4 << endl;
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

3.1.3 string赋值操作

赋值的函数原型

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

```cpp
#include <iostream>
using namespace std;
#include <string>

//string赋值函数
void test01() {
    string str1;
    str1 = "Hello World";
    cout << "str1 = " << str1 << endl;

    string str2;
    str2 = str1;
    cout << "str2 = " << str2 << endl;

    string str3;
    str3 = 'a';
    cout << "str3 = " << str3 << endl;

    string str4;
```

```
    str4.assign("HelloC++");
    cout << "str4 = " << str4 << endl;

    string str5;
    str5.assign("HelloC++",5);
    cout << "str5 = " << str5 << endl;

    string str6;
    str6.assign(str5);
    cout << "str6 = " << str6 << endl;

    string str7;
    str7.assign(10, 'a');
    cout << "str7 = " << str7 << endl;
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

3.1.4 string字符串拼接

函数原型：

```
string& operator+=(const char* str);
string& operator+=(const char c);
string& operator+=(const string& str);
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个字符串连接到当前字符字符串结尾
string& append(const string& s);        //同operator+=(const string& str)
string& append(const string& s,int pos,int n);//字符串s中pos开始的n个字符连接到字符串结尾
```

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

//string字符串拼接
void test01() {
    string str1 = "I ";
    str1 += "Love You";
    cout << "str1 = " << str1 << endl;

    str1 += ':';
    cout << "str1 = " << str1 << endl;
    string str2 = "MC,LOL";
    str1 += str2;
    cout << "str1 = " << str1 << endl;

    string str3 = "I ";
    str3.append("Love");
    cout << "str3 = " << str3 << endl;

```

```
    str3.append("game abcde", 4);
    cout << "str3 = " << str3 << endl;

    //str3.append(str2, 0, 2);//只截取MC
    str3.append(str2, 3, 3);//只截取LOL
    cout << "str3 = " << str3 << endl;
}
int main() {
    test01();
    system("pause");
    return 0;
}
```

3.1.5 string查找和替换

- 查找：查找指定字符串是否存在
- 替换：在指定的位置替换字符串

函数原型：

```
int find(const string& str,int pos = 0)const;//查找str第一次出现位置，从pos开始查找
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的第一次出现位置
int rfind(const string& str,int pos = npos)const;//查找str最后一次出现位置，从pos开始查找
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
```

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

//字符串的查找和替换
//查找
void test01() {
    string str1 = "abcdefgde";
    int pos = str1.find("de");
    if (pos == -1) {
        cout << "未找到该字符串" << endl;
    }
    else {
        cout << "pos = " << pos << endl;
    }
    //rfind 和 find的区别
    //rfind从右往左查找，find从左往右查找
    pos = str1.rfind("de");
    cout << "pos = " << pos << endl;
}
//替换
```

```
void test02() {
    string str1 = "abcdefgde";
    cout << "str1 = " << str1 << endl;
    str1.replace(1, 3, "1111");//1号位置开始的3个字符替换成"1111"
    cout << "str1 = " << str1 << endl;
}
int main()
{
    test01();
    test02();
    system("pause");
    return 0;
}
```

总结：

- find查找是从左往右，rfind从右往左
- find找不到返回-1
- replace在替换时，指定位置开始，多少个字符，替换成什么字符

3.1.6 string字符串比较

比较方式：

- 字符串比较按字符的ASCLL码进行对比
  - 相等——返回0
  - 大于——返回1
  - 小于——返回-1

函数原型

```
int compare(const string& s)const;
int compare(const char* s)const;
```

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

//字符串的比较
void test01() {
    string str1 = "abcdefg";
    string str2 = "abcdefg";
    int ret = str1.compare(str2);
    if (ret == 0) {
        cout << "str1 == str2" << endl;
    }
    else if (ret == 1) {
        cout << "str1 > str2" << endl;
    }
    else if (ret == -1) {
        cout << "str1 < str2" << endl;
    }
}
int main()
{
    test01();
```

```
    system("pause");
    return 0;
}
```

3.1.7 string字符串存取

```
//string中单个字符串存取方式
char& operator[](int n);
char& at(int n);
```

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

//string字符存取
void test01() {
    string str = "Hello World";
    //通过[]访问
    for (int i = 0; i < str.size(); i++) {
        cout << str[i];
    }
    cout << endl;
    //通过at访问
    for (int i = 0; i < str.size(); i++) {
        cout << str.at(i);
    }
    cout << endl;
    //修改单个字符
    str[0] = 'x';
    cout << "str = " << str << endl;
    str.at(1) = 'x';
    cout << "str = " << str << endl;
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

3.1.8 string的插入和删除

函数原型:

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

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

//string插入和删除
void test01() {
```

```
    string str = "Hello World";
    str.insert(1,"aee");
    cout << "str = " << str << endl;
    str.erase(1, 3);
    cout << "str = " << str << endl;
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

3.1.9 string子串

函数原型

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

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

//string子串
void test01() {
    string str = "Hello World";
    string substr = str.substr(0, str.size() / 2);
    cout << "substr = " << substr << endl;
}
//实用操作
void test02() {
    string email = "muffinhead@sina.com";
    //从邮件地址中获取用户名信息
    int pos = email.find("@");
    string username = email.substr(0, pos);
    cout << "username = " << username << endl;
}
int main()
{
    test01();
    test02();
    system("pause");
    return 0;
}
```

3.2 vector容器

3.2.1 vector基本概念

功能:

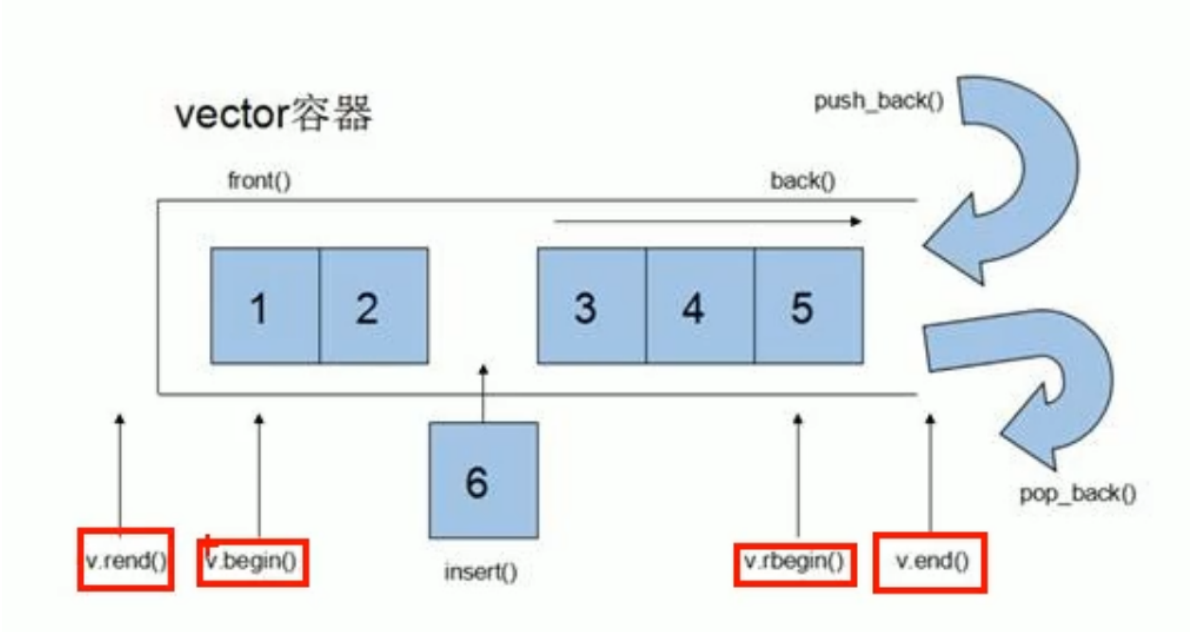
- vector数据结构和数组非常相似，也称为**单端数组**

vector和普通数组的区别:

- 不同之处在于数组是静态空间，而vector可以**动态扩展**

动态扩展：

- 并不是在原空间扩展新的空间，而是寻找更大的空间，然后将原有的数据拷贝到新空间下，释放原空间
- vector容器的迭代器是支持随机访问的迭代器



3.2.2 vector构造函数

函数原型

```
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自身
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
```

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

void printVector(vector<int>&v) {
    for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}
//vector构造函数
void test01() {
    vector<int>v1;//默认构造
    for (int i = 0; i < 10; i++) {
        v1.push_back(i);
    }
    printVector(v1);

    //通过区间方式进行构造
    vector<int>v2(v1.begin(), v1.end());
    printVector(v2);

    //n个elem方式构造
    vector<int>v3(10, 100);
```

```
    printVector(v3);

    //拷贝构造函数
    vector<int>v4(v3);
    printVector(v4);
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

3.2.3 vector赋值操作

函数原型：

```
vector& operator=(const vector& vec);//重载等号运算符
assign(beg,end);//将[beg,end)区间中的数据拷贝赋值给本身
assign(n,elem);//将n个elem拷贝赋值给本身
```

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

void printVector(vector<int>&v) {
    for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}
//vector赋值函数
void test01() {
    vector<int>v1;
    for (int i = 0; i < 10; i++) {
        v1.push_back(i);
    }
    printVector(v1);
    //赋值
    vector<int>v2 = v1;
    printVector(v2);
    //assign
    vector<int>v3;
    v3.assign(v1.begin(), v1.end());
    printVector(v3);

    vector<int>v4;
    v4.assign(10, 100);
    printVector(v4);
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

3.2.4 vector容量和大小

函数原型

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

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

void printVector(vector<int>&v) {
    for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}
//vector容量和大小
void test01() {
    vector<int>v1;
    for (int i = 0; i < 10; i++) {
        v1.push_back(i);
    }
    printVector(v1);
    if (!v1.empty()) {
        cout << "v1不为空" << endl;
        cout << "v1的容量为：" << v1.capacity() << endl;
        cout << "v1的大小为：" << v1.size() << endl;
    }
    else {
        cout << "v1为空" << endl;
    }
    //重新指定大小
    v1.resize(15);
    printVector(v1);
    v1.resize(16, 1);
    printVector(v1);
    v1.resize(5);
    printVector(v1);
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

3.2.5 vector插入和删除

函数原型:

```
push_back(ele);//尾部插入元素ele
pop_back();//删除最后一个元素
insert(const_interator pos,ele);//迭代器指向位置pos插入元素ele
insert(const_interator pos,int count,ele);//迭代器指向位置pos插入count个元素ele
erase(const_interator pos);//删除迭代器指向的元素
erase(const_interator start,const_interator end);//删除迭代器从start到end之间的元素
clear();//删除容器中所有元素
```

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

void printVector(vector<int>&v) {
    for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}
//vector插入和删除
void test01() {
    vector<int>v1;
    for (int i = 0; i < 10; i++) {
        v1.push_back(i);//尾插法插入数据
    }
    printVector(v1);
    v1.pop_back();//尾删法

    v1.insert(v1.begin(), 100);//提供迭代器插入
    printVector(v1);
    v1.insert(v1.begin(), 2, 100);
    printVector(v1);

    //删除
    v1.erase(v1.begin());
    printVector(v1);

    //v1.erase(v1.begin(), v1.end());//类似于清空
    //printVector(v1);

    v1.clear();//清空
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

3.2.6 数据获取

函数原型：

```
at(int idx);//返回索引idx所指的数据
operator[];//返回索引idx所值的数据
front();//返回容器中第一个数据元素
back();//返回容器中最后一个数据元素
```

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

//vector数据存取
void test01() {
    vector<int>v1;
    for (int i = 0; i < 10; i++) {
        v1.push_back(i);
    }
    for (int i = 0; i < v1.size(); i++) {
        //cout << v1[i] << " ";//利用[]访问
        cout << v1.at(i) << " ";//利用成员函数at
    }
    cout << endl;
    //获取第一个元素
    cout << "v1的第一个元素：" << v1.front() << endl;
    //获取最后一个元素
    cout << "v1的最后一个元素：" << v1.back() << endl;
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

3.2.7 vector互换容器

函数原型：

```
swap(vec);//将vec与本身的元素互换
```

```
#include <iostream>
using namespace std;
#include<vector>
void printVector(vector<int>& v) {
    for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}
//vector数据存取
void test01() {
    vector<int>v1;
    for (int i = 0; i < 10; i++) {
```

```
        v1.push_back(i);
    }
    cout << "互换前：" << endl;
    printVector(v1);
    vector<int>v2;
    for (int i = 10; i > 0; i--) {
        v2.push_back(i);
    }
    printVector(v2);
    cout << "交换后：" << endl;
    v1.swap(v2);
    printVector(v1);
    printVector(v2);
}
//实际用途
void test02() {
    //巧用swap 可以收缩内存空间
    vector<int>v;
    for (int i = 0; i < 10000; i++) {
        v.push_back(i);
    }
    cout << "v.capacity() = " << v.capacity() << endl;
    cout << "v.size() = " << v.size() << endl;
    v.resize(3);//重新指定大小
    cout << "v.capacity() = " << v.capacity() << endl;
    cout << "v.size() = " << v.size() << endl;
    //巧用swap收缩内存
    vector<int>(v).swap(v);
    //vector<int>(v);//匿名对象以v来初始化
    cout << "v.capacity() = " << v.capacity() << endl;
    cout << "v.size() = " << v.size() << endl;
}
int main()
{
    test01();
    test02();
    system("pause");
    return 0;
}
```

3.2.8 vector预留空间

功能：

- 减少vector在动态扩展时的扩展次数

函数原型：

```
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
```

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

//vector预留空间
void test01() {
    vector<int>v;
```

```cpp
    //利用reserve预留空间
    v.reserve(10000);
    int num = 0;//统计开辟次数
    int* p = NULL;
    for (int i = 0; i < 10000; i++) {
        v.push_back(i);
        if (p != &v[0]) {
            p = &v[0];
            num++;
        }
    }
    cout << "num = " << num << endl;
}

int main()
{
    test01();
    system("pause");
    return 0;
}
```

- 总结：如果数据量较大，可以一开始就使用reserve预留空间

3.3 deque容器

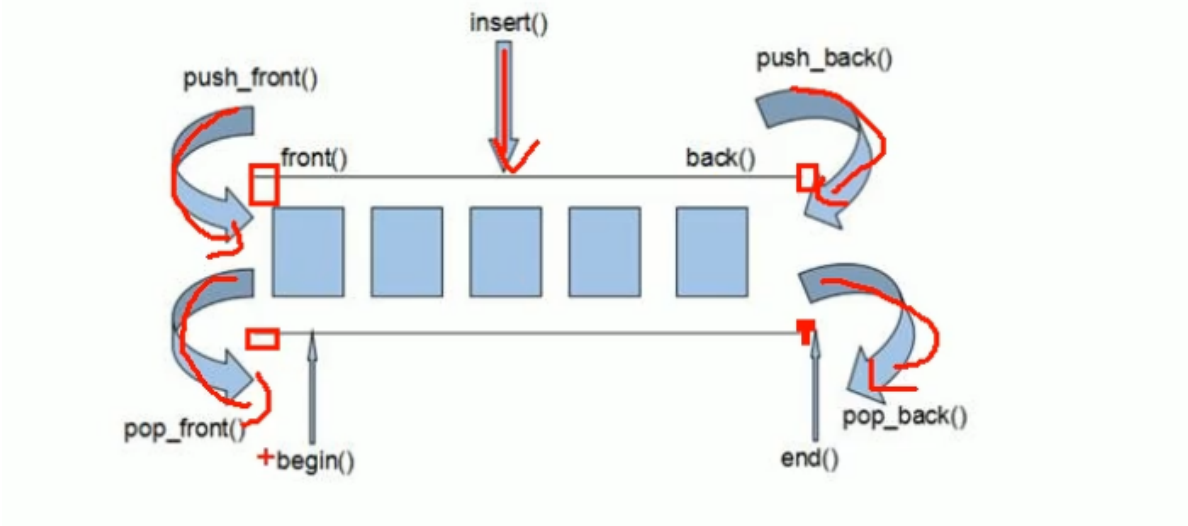
3.3.1deque容器基本概念

功能：

- 双端数组，可以对头端进行插入删除操作

deque和vector的区别：

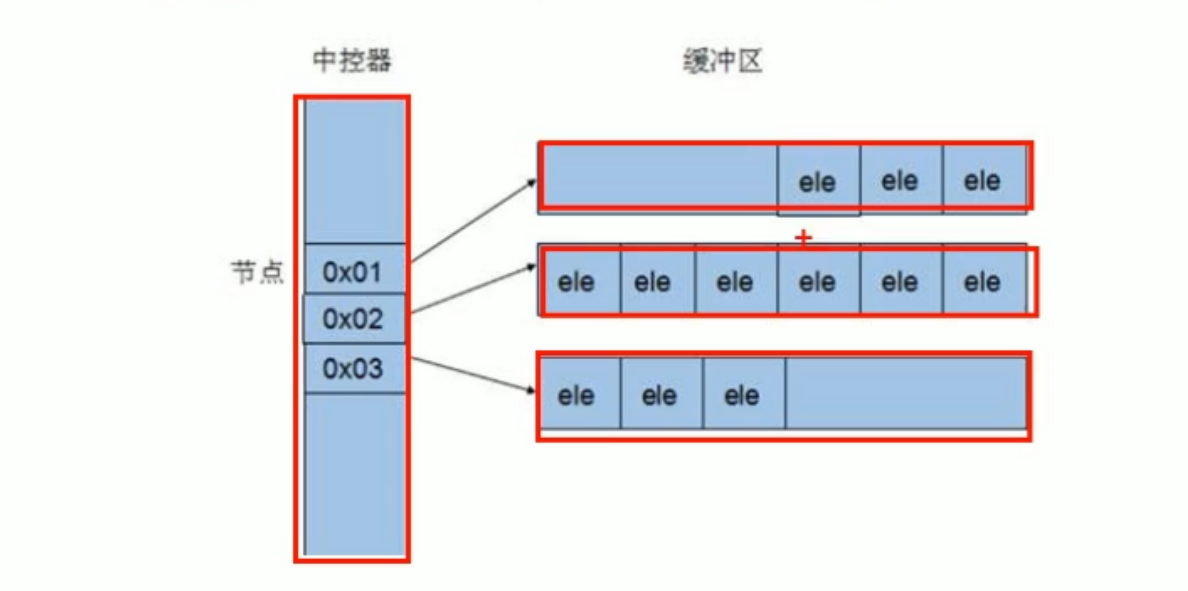
- vector对于头部的插入删除效率低，数据量越大，效率越低
- deque相对而言，对头部的插入删除效率会比vector快
- vector访问元素的速度会比deque快，这和两者的内部实现有关



deque内部工作原理：

deque内部有个中控器，维护每段缓冲区中的内容，缓冲区中存放真实数据

中控器维护的是每个缓冲器的地址，使得使用deque像一片连续的内存空间



- deque容器的迭代器也是支持随机访问的

3.3.2 deque构造函数

函数原型：

```
deque<T>deqT;//默认构造形式
deque(beg,end);//构造函数将[beg,end)区间中的元素拷贝给自身
deque(n,elem);//构造函数将n个elem拷贝给自身
deque(const deque& deq);//拷贝构造函数
```

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

//只读状态
void printDeque(const deque<int>& deq) {
    for (deque<int>::const_iterator it = deq.begin(); it != deq.end(); it++)
{
        cout << *it << " ";
    }
    cout << endl;
}
//deque构造函数
void test01() {
    deque<int>deq1;//默认构造函数
    for (int i = 0; i < 10; i++) {
        deq1.push_back(i);
    }
    printDeque(deq1);

    deque<int>deq2(deq1.begin(), deq1.end());
    printDeque(deq2);

    deque<int>deq3(10, 100);
    printDeque(deq3);

    deque<int>deq4(deq3);
    printDeque(deq4);
}
```

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

3.3.3 deque赋值操作

函数原型：

```
1 deque& operator=(const deque& deq);//重载等号运算符
2 assign(beg,end);//将[beg,end)区间中的数据拷贝赋值给本身
3 assign(n,elem);//将n个elem拷贝赋值给本身
```

```
1  #include <iostream>
2  using namespace std;
3  #include<deque>
4
5  //只读状态
6  void printDeque(const deque<int>& deq) {
7      for (deque<int>::const_iterator it = deq.begin(); it != deq.end(); it++)
   {
8          cout << *it << " ";
9      }
10     cout << endl;
11 }
12 //deque赋值操作
13 void test01() {
14     deque<int>deq1;
15     for (int i = 0; i < 10; i++) {
16         deq1.push_back(i);
17     }
18     printDeque(deq1);
19
20     deque<int>deq2 = deq1;
21     printDeque(deq2);
22
23     deque<int>deq3;
24     deq3.assign(deq1.begin(), deq1.end());
25     printDeque(deq3);
26
27     deque<int>deq4;
28     deq4.assign(10, 100);
29     printDeque(deq4);
30
31 }
32
33 int main()
34 {
35     test01();
36     system("pause");
37     return 0;
38 }
```

3.3.4 deque大小操作

函数原型：

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

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

//只读状态
void printDeque(const deque<int>& deq) {
    for (deque<int>::const_iterator it = deq.begin(); it != deq.end(); it++)
{
        cout << *it << " ";
    }
    cout << endl;
}
//deque大小操作
void test01() {
    deque<int>deq1;
    for (int i = 0; i < 10; i++) {
        deq1.push_back(i);
    }
    printDeque(deq1);
    if (!deq1.empty()) {
        cout << "deq1不为空" << endl;
        cout << "deq1.size() = " << deq1.size() << endl;
        //deque没有容量的概念
    }
    else {
        cout << "deq1为空" << endl;

    }
    //deq1.resize(15);
    deq1.resize(15,1);
    printDeque(deq1);

    deq1.resize(5);
    printDeque(deq1);
}

int main()
{
    test01();
    system("pause");
    return 0;
}
```

3.3.5 deque插入和删除

函数原型：

```
//两端插入操作：
push_back(elem);//在容器尾部添加一个数据
push_front(elem);//在容器头部插入一个数据
pop_back();//删除容器最后一个数据
pop_front();//删除容器第一个数据
//指定位置操作
insert(pop,elem);//在pos位置插入一个elem元素的拷贝，返回新数据的位置
insert(pop,n,elem);//在pos位置插入n个elem数据，无返回值
insert(pop,beg,end);//在pos位置插入[beg,end)区间的数据，无返回值
clear();//清空容器的所有数据
erase(beg,end);//删除[beg,end)区间的数据，返回下一个数据的位置
erase(pos);//删除pos位置的数据，返回下一个数据的位置
```

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

//只读状态
void printDeque(const deque<int>& deq) {
    for (deque<int>::const_iterator it = deq.begin(); it != deq.end(); it++)
{
        cout << *it << " ";
    }
    cout << endl;
}
//deque插入和删除
void test01() {
    deque<int>deq1;

    //尾插
    deq1.push_back(10);
    deq1.push_back(20);

    //头插
    deq1.push_front(100);
    deq1.push_front(200);

    printDeque(deq1);

    //头删
    deq1.pop_front();
    //尾删
    deq1.pop_back();
    printDeque(deq1);
}
void test02() {
    deque<int>deq1;
    deq1.push_back(10);
    deq1.push_back(20);
    deq1.push_front(100);
    deq1.push_front(200);
    printDeque(deq1);

```

```
    //insert插入
    deq1.insert(deq1.begin(), 1000);
    printDeque(deq1);
    deq1.insert(deq1.begin(), 2, 10000);
    printDeque(deq1);

    //按照区间插入
    deque<int>deq2;
    for (int i = 0; i < 10; i++) {
        deq2.push_back(i);
    }
    deq1.insert(deq1.begin(), deq2.begin(), deq2.end());
    printDeque(deq1);
}
void test03() {
    deque<int>deq3;
    for (int i = 0; i < 10; i++) {
        deq3.push_back(i);
    }
    //删除
    deque<int>::iterator it = deq3.begin();
    it++;
    deq3.erase(it);
    printDeque(deq3);

    //按照区间方式删除
    //deq3.erase(deq3.begin(), deq3.end());//类似于清空
    deq3.clear();//清空
    printDeque(deq3);
}
int main()
{
    test01();
    test02();
    test03();
    system("pause");
    return 0;
}
```

3.3.6 deque数据存取

函数原型：

```
at(int idx);//返回索引idx所指的数据
operator[];//返回索引idx所值的数据
front();//返回容器中第一个数据元素
back();//返回容器中最后一个数据元素
```

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

//deque数据存取
void test01() {
    deque<int>deq1;
    for (int i = 0; i < 3; i++) {
```

```
        deq1.push_back(i);
    }
    for (int i = 0; i < 3; i++) {
        deq1.push_front(i+3);
    }
    //通过[]和at方式访问元素
    for (int i = 0; i < deq1.size() ; i++) {
        //cout << deq1[i] << " ";
        cout << deq1.at(i) << " ";
    }
    cout << endl;

    cout << "第一个元素：" << deq1.front() << endl;
    cout << "最后一个元素：" << deq1.back() << endl;
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

3.3.7 deque排序

```
//算法
sort(iterator beg,iterator end)//对beg和end区间内元素进行排序
```

```
#include <iostream>
using namespace std;
#include<deque>
#include <algorithm>
void printDeque(const deque<int>& deq) {
    for (deque<int>::const_iterator it = deq.begin(); it != deq.end(); it++)
{
        cout << *it << " ";
    }
    cout << endl;
}
//deque排序
void test01() {
    deque<int>deq1;
    deq1.push_back(10);
    deq1.push_back(100);
    deq1.push_back(160);
    deq1.push_back(20);
    deq1.push_back(90);
    deq1.push_back(70);
    deq1.push_back(30);
    cout << "排序前：" << endl;
    printDeque(deq1);
    //排序：默认升序
    //对于支持随机访问的迭代器的容器，都支持用sort算法对其进行排序
    sort(deq1.begin(), deq1.end());
    cout << "排序后：" << endl;
    printDeque(deq1);
}
```

```cpp
int main()
{
    test01();
    system("pause");
    return 0;
}
```

3.4 容器案例——评委打分

3.4.1 案例描述

- 有五名选手ABCDE，10个评委分别对每一位选手打分，去除最高分和最低分，取平均分

3.4.2 实现步骤

- 创建五名选手，放到vector中
- 遍历vector容器，取出来每一个选手，执行for循环。可以把10个评委打分存放到deque容器中
- sort算法对deque容器遍历中分数排序，去除最高分和最低分
- deque容器遍历一遍，累加总分
- 获取平均分

```cpp
#include<iostream>
using namespace std;
#include<string>
#include<vector>
#include<deque>
#include<ctime>
#include <algorithm>
class Person {
public:
    Person(string name) {
        this->m_Name = name;
    }
    string m_Name;
    int m_Score;
};
void CreatPerson(vector<Person>& v) {
    string nameSeed = "ABCDE";
    for (int i = 0; i < 5; i++) {
        Person p("选手");
        p.m_Name += nameSeed.at(i);
        p.m_Score = 0;//初始化为0
        v.push_back(p);
    }
}
void setScore(vector<Person>& v) {
    for (vector<Person>::iterator it = v.begin(); it != v.end(); it++) {
        deque<int>d;
        for (int i = 0; i < 10; i++) {
            int score = rand() % 41 + 60;
            d.push_back(score);
        }
        //排序
        sort(d.begin(), d.end());
        //去除最高分和最低分
        d.pop_back();
        d.pop_front();
```

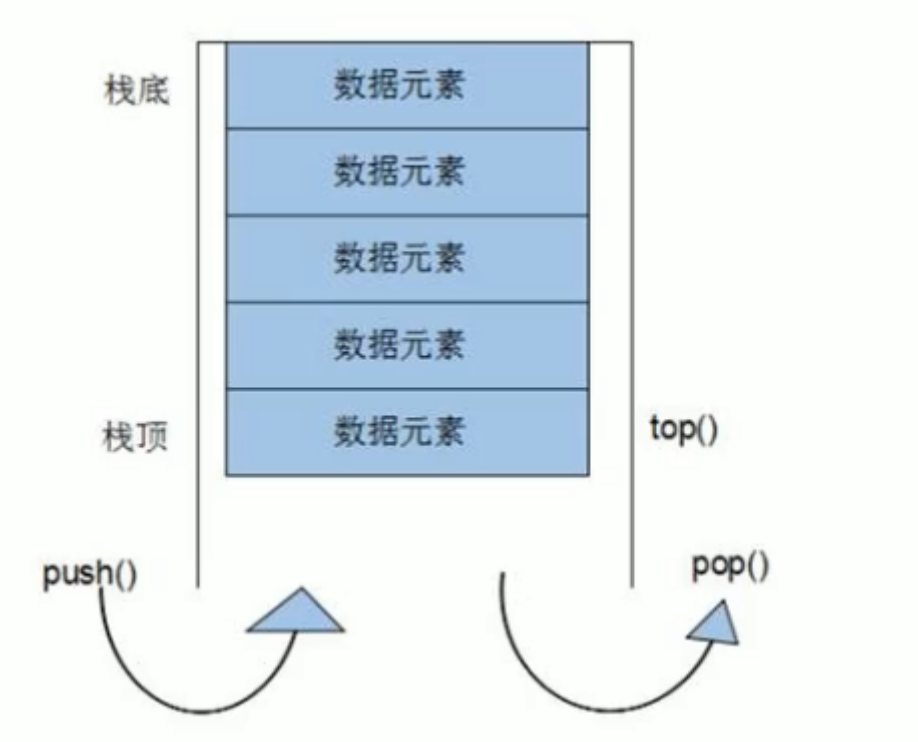
```
37
38          //均分
39          int sum = 0;
40          for (deque<int>::iterator it = d.begin(); it != d.end();it++) {
41              sum += (*it);
42          }
43          int avg = sum / d.size();
44          it->m_Score = avg;
45      }
46  }
47  void showScore(vector<Person>& v) {
48      for (vector<Person>::iterator it = v.begin(); it != v.end(); it++) {
49          cout << "name:" << it->m_Name << " score:" << it->m_Score << endl;
50      }
51  }
52  int main()
53  {
54      srand((unsigned)time(NULL));
55      //创建五名选手，放到vector容器中
56      vector<Person>v;
57      CreatPerson(v);
58      //给五名选手打分
59      setScore(v);
60      //展示分数
61      showScore(v);
62
63      system("pause");
64      return 0;
65  }
```

3.5 stack容器

3.5.1 stack基本概念

概念:

stack是一种**先进后出**（First In Last Out,FILO)的数据结构，他只有一个出口



栈中只有顶端的元素才可以被外界使用，因此栈不允许有遍历行为

栈中进入数据称为——**入栈push**

栈中弹出数据称为——**出栈pop**

3.5.2 stack常用接口

构造函数：

```
stack<T> stk;//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
```

赋值操作：

```
stack& operator=(const stack& stk);//重载等号运算符
```

数据存取：

```
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
```

大小操作：

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

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

void test01() {
    stack<int>stk;
```

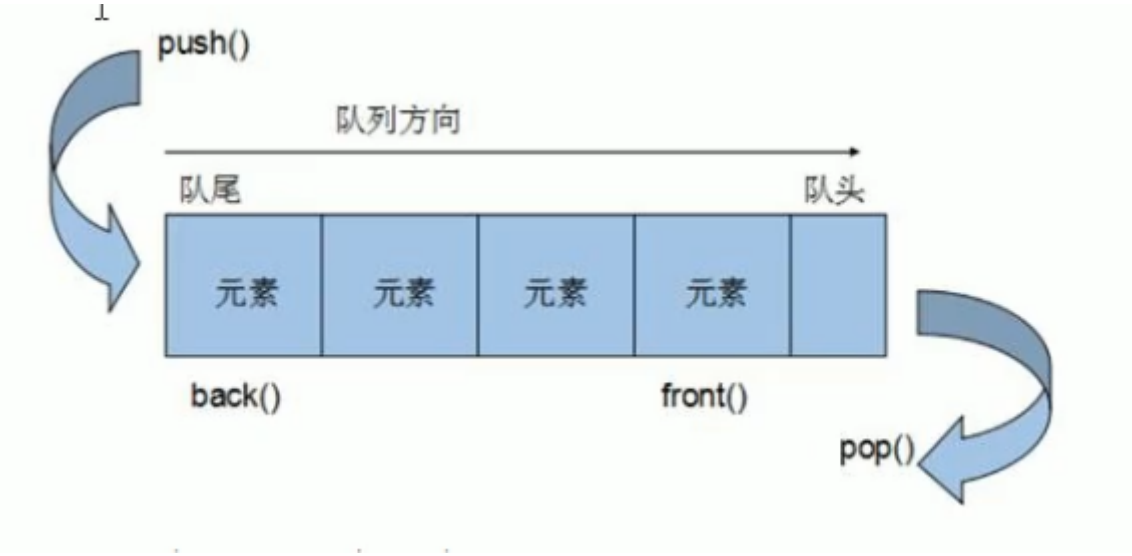
```
    //入栈
    stk.push(10);
    stk.push(20);
    stk.push(30);
    stk.push(40);
    cout << "stk.size() = " << stk.size() << endl;

    while (!stk.empty()) {
        cout << stk.top() << " ";
        stk.pop();//出栈
    }
    cout << endl << "stk.size() = " << stk.size() << endl;
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

3.6 queue容器

3.6.1 queue基本概念

概念：Queue是一种**先进后出**（First In First Out,FIFO)的数据结构,它有两个出口



队列容器允许从一端新增元素，从另一端移除元素

队列中只有对头和队尾才可以被外界使用，因此队列不允许有遍历行为

队列中进数据称为——**入队push**

队列中出数据称为——**出队pop**

3.6.2 queue常用接口

构造函数：

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

赋值操作：

```
queue& operator=(const queue& queue);//重载等号运算符
```

数据存取：

```
push(elem);//向队尾添加元素
pop();//从对头移除第一个元素
back();//返回最后一个元素
front();//返回第一个元素
```

大小操作：

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

```
#include <iostream>
using namespace std;
#include <queue>
#include <string>
class Person {
public:
    Person(string name, int age) {
        this->m_Name = name;
        this->m_Age = age;
    }
    string m_Name;
    int m_Age;
};
//队列Queue
void test01() {
    queue<Person>que;
    Person p1("Tom", 18);
    Person p2("Jerry", 18);
    Person p3("Mary", 18);
    Person p4("John", 18);

    //入队
    que.push(p1);
    que.push(p2);
    que.push(p3);
    que.push(p4);

    cout << "que.size() = " << que.size() << endl;
    while (!que.empty()) {
        //输出对头
        cout << "队头元素--name:" << que.front().m_Name << " age:" << que.front().m_Age << endl;
        cout << "队尾元素--name:" << que.back().m_Name << " age:" << que.back().m_Age << endl;
        //出队
        que.pop();
    }
    cout << "que.size() = " << que.size() << endl;
}
int main()
{
```

```
    test01();
    system("pause");
    return 0;
}
```

3.7 list容器

3.7.1 list基本概念

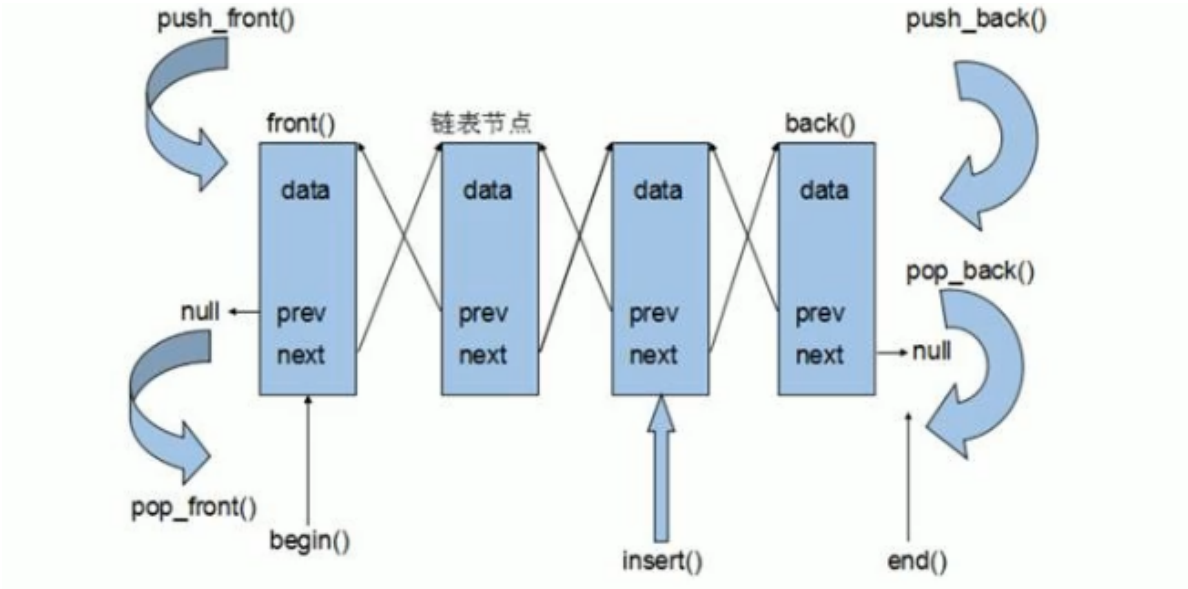
功能：将数据进行链式存储

链表（list）：一种五里存储单元上非连续的存储结构，数据元素的逻辑顺序是通过链表的指针链接实现的

链表的组成：链表由一系列**结点**组成

结点的组成：一个是存储数据单元的**数据域**，另一个是存储下一个结点地址的**指针域**

STL中的链表是一个双向循环链表



由于链表的存储方式并不是连续的内存空间，因此链表list的中的迭代器只支持前移和后移，属于**双向迭代器**

list优点：

- 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方便

list缺点：

- 链表灵活，但是空间（指针域）和时间（遍历）额外耗费较大

重要性质：插入操作和删除操作都不会造成原有的list迭代器的失效，这在vector中是不成立的

3.7.2 list构造函数

函数原型：

```
list<T>lst;//list采用模板类实现，list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自身
list(n,elem);//构造函数将n个elem拷贝给自身
list(const list& lst);//拷贝构造函数
```

```cpp
#include <iostream>
using namespace std;
#include <list>

void printList(const list<int>& L) {
    for (list<int>::const_iterator it = L.begin(); it != L.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}
//list构造函数
void test01() {
    list<int>L1;
    //添加数据
    L1.push_back(10);
    L1.push_back(20);
    L1.push_back(30);
    L1.push_back(40);
    printList(L1);

    list<int>L2(L1.begin(), L1.end());
    printList(L2);

    list<int>L3(L2);
    printList(L3);

    list<int>L4(10, 100);
    printList(L4);
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

3.7.3 list赋值和交换

函数原型：

```cpp
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 lisy& lst);//重载等号运算符
swap(lst);//将lst与本身元素互换
```

```cpp
#include <iostream>
using namespace std;
#include <list>

void printList(const list<int>& L) {
    for (list<int>::const_iterator it = L.begin(); it != L.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}
//list赋值和交换
```

```cpp
void test01() {
    list<int>L1;
    L1.push_back(10);
    L1.push_back(20);
    L1.push_back(30);
    L1.push_back(40);
    printList(L1);

    list<int>L2 = L1;
    printList(L2);

    list<int>L3;
    L3.assign(L2.begin(), L2.end());
    printList(L3);

    list<int>L4;
    L4.assign(10, 100);
    printList(L4);
};
//交换
void test02() {
    list<int>L1;
    L1.push_back(10);
    L1.push_back(20);
    L1.push_back(30);
    L1.push_back(40);

    list<int>L2;
    L2.assign(10, 100);
    cout << "交换前" << endl;
    printList(L1);
    printList(L2);
    L1.swap(L2);
    cout << "交换后" << endl;
    printList(L1);
    printList(L2);
}
int main()
{
    test01();
    test02();
    system("pause");
    return 0;
}
```

3.7.4 list大小操作

函数原型:

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

```cpp
#include <iostream>
using namespace std;
#include <list>

void printList(const list<int>& L) {
    for (list<int>::const_iterator it = L.begin(); it != L.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}
//list大小操作
void test01() {
    list<int>L1;
    L1.push_back(10);
    L1.push_back(20);
    L1.push_back(30);
    L1.push_back(40);
    printList(L1);
    if (!L1.empty()) {
        cout << "L1不为空" << endl;
        cout << "L1的元素个数: " << L1.size() << endl;
    }
    else {
        cout << "L1为空" << endl;
    }
    //重新指定大小
    //L1.resize(10);
    L1.resize(10,100);
    printList(L1);
    L1.resize(3);
    printList(L1);
};

int main()
{
    test01();
    system("pause");
    return 0;
}
```

3.7.5 list插入和删除

函数原型:

```cpp
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();//在容器开头移除第一个元素
insert(pos,elem);//在pos位置查elem元素的拷贝，返回新数据的位置
insert(pos,n,elem);//在pos位置插入[beg,end)区间的数据，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
```

```cpp
#include <iostream>
using namespace std;
#include <list>

void printList(const list<int>& L) {
    for (list<int>::const_iterator it = L.begin(); it != L.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}
//list插入和删除
void test01() {
    list<int>L;
    //尾插
    L.push_back(10);
    L.push_back(20);
    L.push_back(30);
    //头插
    L.push_front(100);
    L.push_front(200);
    L.push_front(300);

    printList(L);

    //尾删
    L.pop_back();
    //头删
    L.pop_front();
    printList(L);
    //insert插入
    list<int>::iterator it = L.begin();
    L.insert(++it, 1000);
    printList(L);

    //删除
    it = L.begin();
    L.erase(++it);
    printList(L);

    //移除
    L.push_back(10000);
    L.push_back(10000);
    L.push_back(10000);
    L.push_back(10000);
    printList(L);
    L.remove(10000);
    printList(L);

    //清空
    L.clear();
    printList(L);
};

int main()
{
    test01();
    system("pause");
    return 0;
```

```
}
```

3.7.6 list数据存取

函数原型：

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

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

//list数据存取
void test01() {
    list<int>L1;
    L1.push_back(10);
    L1.push_back(20);
    L1.push_back(30);
    L1.push_back(40);

    cout << "L1的第一个元素:" << L1.front() << endl;
    cout << "L1的最后一个元素: " << L1.back() << endl;

    //list迭代器不支持随机访问
    list<int>::iterator it = L1.begin();
    it++;
    it--;   //支持双向
    //it = it + 1;可能是跳跃型访问
};

int main()
{
    test01();
    system("pause");
    return 0;
}
```

3.7.7 list反转和排序

函数原型：

```
reverse();//反转链表
sort();//链表排序
```

```
#include <iostream>
using namespace std;
#include <list>
#include <algorithm>
void printList(const list<int>& L) {
    for (list<int>::const_iterator it = L.begin(); it != L.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}
```

```cpp
bool myCompare(int v1,int v2) {
    //降序
    return v1 > v2;
}
//list反转和排序
void test01() {
    list<int>L1;
    L1.push_back(10);
    L1.push_back(40);
    L1.push_back(60);
    L1.push_back(50);
    L1.push_back(80);
    L1.push_back(400);
    L1.push_back(90);
    L1.push_back(30);
    cout << "排序前：" << endl;
    printList(L1);
    //反转
    cout << "反转后：" << endl;
    L1.reverse();
    printList(L1);

    //排序
    cout << "排序后（默认升序）：" << endl;
    //所有不支持随机访问的迭代器的容器，不可以使用标准算法
    //不支持随机访问的迭代器的容器，内部会提供对应的算法
    //sort(L1.begin(), L1.end());
    L1.sort();//默认升序
    printList(L1);

    cout << "排序后（降序）:" << endl;
    L1.sort(myCompare);
    printList(L1);
};

int main()
{
    test01();
    system("pause");
    return 0;
}
```

3.7.8 排序案例

- 案例描述：将Person自定义数据类型进行排序，Person中属性有姓名、年龄、身高
- 排序规则：按照年龄进行升序，如果年龄相同按照身高进行排序

```cpp
#include<iostream>
#include<string>
#include<list>
using namespace std;

class Person {
public:
    Person(string name, int age, double height) {
        this->m_Name = name;
        this->m_Age = age;
```

```
        this->m_Height = height;
    }
    string m_Name;
    int m_Age;
    double m_Height;
};
bool comparePerson(Person& p1, Person& p2) {
    //按照年龄升序
    if (p1.m_Age == p2.m_Age) {
        return p1.m_Height > p2.m_Height;//按照身高降序
    }
    return p1.m_Age < p2.m_Age;
}
void test01() {
    list<Person>L;
    Person p1("刘备", 35, 175);
    Person p2("曹操", 45, 180);
    Person p3("孙权", 40, 170);
    Person p4("赵云", 25, 190);
    Person p5("张飞", 35, 160);
    Person p6("关羽", 35, 200);
    L.push_back(p1);
    L.push_back(p2);
    L.push_back(p3);
    L.push_back(p4);
    L.push_back(p5);
    L.push_back(p6);

    for (list<Person>::iterator it = L.begin(); it != L.end(); it++) {
        cout << "姓名: " << it->m_Name << " 年龄: " << it->m_Age << " 身高: " << it->m_Height << endl;
    }
    //排序
    cout << "---------------------------------------" << endl;
    cout << "排序后: " << endl;
    L.sort(comparePerson);
    for (list<Person>::iterator it = L.begin(); it != L.end(); it++) {
        cout << "姓名: " << it->m_Name << " 年龄: " << it->m_Age << " 身高: " << it->m_Height << endl;
    }
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

3.8 set / multiset容器

3.8.1 set基本概念

简介:

- 所有元素都会在插入时自动被排序

本质:

- set / multiset属于**关联式容器**，底层结构是用**二叉树**实现

set和multiset区别

- set不允许容器中有重复的元素
- multiset允许容器中有重复的元素

3.8.2 set构造和赋值

```
//构造
set<T> st;//默认构造函数
set(const set& st);//拷贝构造函数
//赋值
set& operator=(const set& st);//重载等号操作符
```

```
#include<iostream>
using namespace std;
#include<set>
//set容器构造和赋值
void printSet(set<int>& s) {
    for (set<int>::iterator it = s.begin(); it != s.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}
void test01() {
    set<int>s1;
    s1.insert(10);
    s1.insert(40);
    s1.insert(30);
    s1.insert(20);
    s1.insert(30);
    //所有的元素在插入会被默认排序，set容器不允许插入重复值
    printSet(s1);

    set<int>s2(s1);
    printSet(s2);

    set<int>s3;
    s3 = s2;
    printSet(s3);
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

3.8.3 set大小和交换

函数原型：

```
size();//返回容器中元素的数目
empty();//判断容器是否为空
swap(st);//交换两个集合容器
```

```cpp
#include<iostream>
using namespace std;
#include<set>
//set大小和交换
void printSet(set<int>& s) {
    for (set<int>::iterator it = s.begin(); it != s.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}
void test01() {
    set<int>s1;
    s1.insert(10);
    s1.insert(40);
    s1.insert(30);
    s1.insert(20);
    //判断是否为空
    if (!s1.empty()) {
        cout << "集合容器s1不为空" << endl;
        cout << "s1.size() = " << s1.size() << endl;
    }
    else {
        cout << "集合容器s1为空" << endl;
    }
    set<int>s2;
    s2.insert(10);
    s2.insert(40);
    s2.insert(50);
    s2.insert(70);
    s2.insert(80);
    s2.insert(20);

    cout << "交换前" << endl;
    printSet(s1);
    printSet(s2);
    s1.swap(s2);
    cout << "交换后" << endl;
    printSet(s1);
    printSet(s2);
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

3.8.4 set插入和删除

函数原型：

```cpp
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，返回下一个元素迭代器
erase(elem);//删除容器中值为elem的元素
```

```cpp
#include<iostream>
using namespace std;
#include<set>
//set插入和删除
void printSet(set<int>& s) {
    for (set<int>::iterator it = s.begin(); it != s.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}
void test01() {
    set<int>s1;
    //插入
    s1.insert(10);
    s1.insert(40);
    s1.insert(30);
    s1.insert(20);

    printSet(s1);

    //删除
    s1.erase(s1.begin());
    printSet(s1);
    //删除重载的版本
    s1.erase(30);
    printSet(s1);

    //清空
    //s1.erase(s1.begin(), s1.end());
    s1.clear();
    printSet(s1);
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

3.8.5 set查找和统计

函数原型：

```cpp
find(key);//查找key是否存在，返回该元素的迭代器，若不存在，返回set.end()
count(key);//统计key元素的个数
```

```cpp
#include<iostream>
using namespace std;
#include<set>
//set查找和统计

void test01() {
    set<int>s1;
    //插入
    s1.insert(10);
    s1.insert(40);
```

```
    s1.insert(30);
    s1.insert(20);
    //查找
    set<int>::iterator pos = s1.find(300);
    if (pos != s1.end()) {
        cout << "找到该元素: " << *pos << endl;
    }
    else {
        cout << "未找到元素" << endl;
    }
    //统计
    int num = s1.count(30);
    cout << "num = " << num << endl;
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

3.8.6 set和multiset区别

区别:

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

```
#include<iostream>
using namespace std;
#include<set>
//set和multiset区别
void test01() {
    set<int>s;
    pair<set<int>::iterator, bool> ret = s.insert(10);
    if (ret.second) {
        cout << "第一次插入成功" << endl;
    }
    else {
        cout << "第一次插入失败" << endl;
    }
    ret = s.insert(10);
    if (ret.second) {
        cout << "第二次插入成功" << endl;
    }
    else {
        cout << "第二次插入失败" << endl;
    }
    multiset<int>ms;//允许插入重复数据
    ms.insert(10);
    ms.insert(10);
    ms.insert(10);
    ms.insert(10);
    for (multiset<int>::iterator it = ms.begin(); it != ms.end(); it++) {
        cout << *it << " ";
    }
```

```
    cout << endl;
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

3.8.7 pair对组创建

功能描述：

- 成对出现的数据，利用对组可以返回两个数据

两种创建方式：

```
pair<type,type>p(value1,value2);
pair<type,type>p = make_pair(value1,value2);
```

```
#include<iostream>
using namespace std;
#include<string>
//对组创建
void test01() {
    pair<string, int> p(string("Tom"), 20);
    cout << "name:" << p.first << " age:" << p.second << endl;

    pair<string, int>p2 = make_pair("Jerry", 20);
    cout << "name:" << p2.first << " age:" << p2.second << endl;
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

3.8.8 set容器排序

- 利用仿函数，改变默认排序规则

```
//set存放内置数据类型
#include<iostream>
using namespace std;
#include<string>
#include<set>
//set容器排序
class MyCompare {
public:
    bool operator()(int v1,int v2)const {
        return v1 > v2;
    }
};
void test01() {
    set<int>s1;
```

```
    s1.insert(10);
    s1.insert(40);
    s1.insert(30);
    s1.insert(20);
    s1.insert(50);
    for (set<int>::const_iterator it = s1.begin(); it != s1.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
    //指定排序规则为从大到小
    //set插入数据之前改变排序规则
    set<int,MyCompare>s2;
    s2.insert(10);
    s2.insert(40);
    s2.insert(30);
    s2.insert(20);
    s2.insert(50);
    for (set<int,MyCompare>::const_iterator it = s2.begin(); it != s2.end();
it++) {
        cout << *it << " ";
    }
    cout << endl;
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

```
//set存放自定义数据类型
#include<iostream>
using namespace std;
#include<string>
#include<set>
//set容器排序
class Person {
public:
    Person(string name, int age) {
        this->m_Name = name;
        this->m_Age = age;
    }
    string m_Name;
    int m_Age;
};
class MyComparePerson {
public:
    bool operator()(const Person&p1,const Person&p2)const {
        return p1.m_Age > p2.m_Age;
    }
};
void test01() {
    //自定义数据类型都会指定排序规则
    set<Person,MyComparePerson>s;
    Person p1("刘备", 35);
    Person p2("曹操", 45);
    Person p3("孙权", 40);
```

```
    Person p4("赵云", 25);
    s.insert(p1);
    s.insert(p2);
    s.insert(p3);
    s.insert(p4);

    for (set<Person,MyComparePerson>::iterator it = s.begin(); it != s.end(); it++) {
        cout << "name:" << it->m_Name << " age:" << it->m_Age << endl;
    }
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

3.9 map / multimap 容器

3.9.1 map基本概念

简介：

- map中所有容器都是pair
- pair中第一个元素是key（键值），起到索引作用，第二个元素是value（实值）
- 所有元素都会根据元素的键值自动排序

本质：

- map / multimap属于**关联式容器**，底层结构是用**二叉树**实现

优点：

- 可以根据key值快速找到value值

map和multimap**区别**：

- map不允许容器中有重复key值元素
- multimap允许容器中有重复key值元素

3.9.2 map构造和赋值

函数原型：

```
//构造
map<T1,T2>mp;//默认构造函数
map(const map& mp);//拷贝构造函数
//赋值
map& operator=(const map& mp);//重载等号操作符
```

```
#include<iostream>
using namespace std;
#include<map>
//map构造和赋值
void printMap(const map<int, int>& m) {
    for (map<int, int>::const_iterator it = m.begin(); it != m.end(); it++)
{
```

```
        cout << "key = " << it->first << " value = " << it->second << endl;
    }
    cout << endl;
}
void test01() {
    map<int, int>mp;
    //对组插入
    mp.insert(pair<int, int>(1, 10));
    mp.insert(pair<int, int>(3, 30));
    mp.insert(pair<int, int>(4, 40));
    mp.insert(pair<int, int>(2, 20));

    printMap(mp);
    //拷贝构造
    map<int, int>mp2(mp);
    printMap(mp2);

    map<int, int>mp3;
    mp3 = mp;
    printMap(mp3);
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

3.9.3 map大小和交换

函数原型：

```
size();//返回容器中元素的数目
empty();//判断容器是否为空
swap(st);//交换两个集合容器
```

```
#include<iostream>
using namespace std;
#include<map>
//map大小和交换
void printMap(const map<int, int>& m) {
    for (map<int, int>::const_iterator it = m.begin(); it != m.end(); it++)
{
        cout << "key = " << it->first << " value = " << it->second << endl;
    }
    cout << endl;
}
void test01() {
    map<int, int>mp;
    mp.insert(pair<int, int>(1, 10));
    mp.insert(pair<int, int>(3, 30));
    mp.insert(pair<int, int>(4, 40));
    mp.insert(pair<int, int>(2, 20));

    if (!mp.empty()) {
        cout << "mp不为空" << endl;
```

```
        cout << "mp.size() = " << mp.size() << endl;
    }
    else {
        cout << "mp为空" << endl;
    }
    map<int, int>mp2;
    mp2.insert(pair<int, int>(5, 10));
    mp2.insert(pair<int, int>(6, 30));
    mp2.insert(pair<int, int>(7, 40));
    mp2.insert(pair<int, int>(8, 20));
    cout << "交换前：" << endl;
    printMap(mp);
    printMap(mp2);

    mp.swap(mp2);

    cout << "交换后：" << endl;
    printMap(mp);
    printMap(mp2);
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

3.9.4 map插入和删除

函数原型：

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

```
#include<iostream>
using namespace std;
#include<map>
//map插入和删除
void printMap(const map<int, int>& m) {
    for (map<int, int>::const_iterator it = m.begin(); it != m.end(); it++)
{
        cout << "key = " << it->first << " value = " << it->second << endl;
    }
    cout << endl;
}
void test01() {
    map<int, int>mp;
    //插入
    //第一种
    mp.insert(pair<int, int>(1, 10));
    //第二种
    mp.insert(make_pair(3, 30));
    //第三种（可忽略）
```

```
    mp.insert(map<int,int>::value_type(4,40));
    //第四种（不建议）----用途：利用key访问value
    mp[2] = 20;
    //cout << mp[5] << endl;
    printMap(mp);

    //删除
    mp.erase(mp.begin());
    printMap(mp);

    mp.erase(3);//按照key删除
    printMap(mp);

    //清空
    //mp.erase(mp.begin(), mp.end());
    mp.clear();
    printMap(mp);

}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

3.9.5 map查找和统计

函数原型：

```
find(key);//查找key是否存在，若存在，返回该键的元素的迭代器，若不存在，返回set.end();
count(key);//统计key的元素个数
```

```
#include<iostream>
using namespace std;
#include<map>
//map查找和统计
void test01() {
    map<int, int>mp;
    //map不允许插入重复的键值
    mp.insert(pair<int, int>(1, 10));
    mp.insert(pair<int, int>(3, 30));
    mp.insert(pair<int, int>(4, 40));
    mp.insert(pair<int, int>(2, 20));

    //查找
    map<int, int>::iterator pos = mp.find(3);
    if (pos != mp.end()) {
        cout << "查找到了元素 key = " << pos->first << " value = " << pos->second << endl;
    }
    else {
        cout << "未找到元素" << endl;
    }
    //统计
    int num = mp.count(3);//map统计结果  1 or 0
```

```
    cout << "num = " << num << endl;
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

3.9.6 map排序

- 利用仿函数改变默认排序规则

```
#include<iostream>
using namespace std;
#include<map>
//map排序
class MyCompareMap {
public:
    bool operator()(const int& v1, const int& v2)const {
        return v1 > v2;
    }
};
void test01() {
    map<int, int,MyCompareMap>mp;
    //map不允许插入重复的键值
    mp.insert(pair<int, int>(1, 10));
    mp.insert(pair<int, int>(3, 30));
    mp.insert(pair<int, int>(4, 40));
    mp.insert(make_pair(2, 20));
    mp.insert(make_pair(5, 50));
    for (map<int, int,MyCompareMap>::iterator it = mp.begin(); it != mp.end(); it++) {
        cout << "key = " << it->first << " value = " << it->second << endl;
    }
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

总结：

- 对于自定义数据类型，必须指定排序规则

3.10 案例——员工分组

3.10.1 案例描述

- 公司今天招聘了10个员工（ABCDEFGHIJ），10名员工进入公司后，需要指派员工在哪个部门工作
- 员工信息有：姓名、工资组成；部门分为：策划，美术，研发
- 随机给10名员工分配部门和工资
- 通过multimap进行信息的插入，key（部门编号）value（员工）
- 分部门显示员工信息

3.10.2 实现步骤

- 创建10名员工，放到vector中
- 遍历vector容器，取出每个员工，进行随机分组
- 分组后，将员工部门编号作为key，具体员工作为value，放到multimap容器中
- 分部门显示员工信息

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

#define Worker_Num 10
enum {
    CEHUA,
    MEISHU,
    YANFA
};
class Worker {
public:
    string m_Name;//姓名
    int m_Salary;//工资
};
void CreateWorker(vector<Worker>& v) {
    string nameSeed = "ABCDEFGHIJ";
    for (int i = 0; i < Worker_Num; i++) {
        Worker worker;
        worker.m_Name = "员工";
        worker.m_Name += nameSeed.at(i);
        worker.m_Salary = rand() % 10000 + 10000;
        //将员工放入vector容器
        v.push_back(worker);
    }
}
void setGroup(vector<Worker>& v, multimap<int, Worker>& m) {
    //遍历员工随机分组
    for (vector<Worker>::iterator it = v.begin(); it != v.end(); it++) {
        int depId = rand() % 3;
        m.insert(make_pair(depId, *it));
    }
}
void showWokerBydepId(int elem, multimap<int, Worker>& m) {
    multimap<int, Worker>::iterator pos = m.find(elem);
    int count = m.count(elem);
    int index = 0;
    for (; pos != m.end() && index < count; pos++, index++) {
        cout << "姓名：  " << pos->second.m_Name << " 工资：" << pos-
>second.m_Salary << endl;
    }
}
void showWorkerByGroup(multimap<int, Worker>& m) {
    cout << "策划部门：" << endl;
    showWokerBydepId(CEHUA, m);
    cout << "------------------------------" << endl;
    cout << "美术部门：" << endl;
```

```
    showWokerBydepId(MEISHU, m);
    cout << "-----------------------------" << endl;
    cout << "研发部门：" << endl;
    showWokerBydepId(YANFA, m);
}

int main()
{
    srand((unsigned int)time(NULL));
    //创建员工
    vector<Worker>vWorker;
    CreateWorker(vWorker);
    //员工分组
    multimap<int, Worker> mWorker;
    setGroup(vWorker, mWorker);
    //分组显示
    showWorkerByGroup(mWorker);

    system("pause");
    return 0;
}
```

4 STL-函数对象

4.1 函数对象

4.1.1 函数对象概念

概念：

- 重载**函数调用操作符**的类，其对象称为**函数对象**
- **函数对象**使用重载的()时，行为类似函数调用，也叫**仿函数**

本质：

函数对象（仿函数）是一个**类**，不是一个函数

4.1.2 函数对象使用

特点：

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

```
#include<iostream>
using namespace std;
#include <string>
//函数对象
class MyAdd {
public:
    int operator()(int v1, int v2) {
        return v1 + v2;
    }
};
class MyPrint {
public:
    MyPrint() {
```

```
        this->count = 0;
    }
    void operator()(string str) {
        cout << str << endl;
        count++;
    }
    int count;//内部自己状态
};
void doPrint(MyPrint& mp, string str) {
    mp(str);
}
void test01() {
    MyAdd myAdd;
    cout << myAdd(10, 10) << endl;

    MyPrint myPrint;
    myPrint("Hello C++");
    cout << "myPrint调用次数 = " << myPrint.count << endl;

    doPrint(myPrint, "Hello C++");
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

4.2 谓词

4.2.1 谓词概念

*概念:

- 返回bool类型的仿函数称为**谓词**
- 如果operator()接受一个参数，那么叫做一元谓词
- 如果operator()接受两个参数，那么叫做二元谓词

4.2.2 一元谓词

```
#include<iostream>
using namespace std;
#include <vector>
#include <algorithm>
//一元谓词
class GreaterFive {
public:
    bool operator()(int val) {
        return val > 5;
    }
};
void test01() {
    vector<int>v;
    for (int i = 0; i < 10; i++) {
        v.push_back(i);
    }
    //查找容器中有没有大于5的数字
```

```
    //GreaterFive()--匿名函数对象
    vector<int>::iterator pos = find_if(v.begin(), v.end(), GreaterFive());
    if (pos != v.end()) {
        cout << "找到了, " << *pos << endl;
    }
    else {
        cout << "未找到" << endl;
    }
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

4.2.3 二元谓词

```
#include<iostream>
using namespace std;
#include <vector>
#include <algorithm>
//二元谓词
class MyCompare {
public:
    bool operator()(int v1,int v2) {
        return v1 > v2;
    }
};
void test01() {
    vector<int>v;
    v.push_back(10);
    v.push_back(50);
    v.push_back(40);
    v.push_back(20);
    v.push_back(70);

    sort(v.begin(), v.end());
    for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
        cout << *it << " ";
    }
    cout << endl << "--------------" << endl;
    sort(v.begin(), v.end(), MyCompare());//改变默认排序规则
    for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

4.3 内建函数对象

4.3.1 内建函数对象意义

概念：

- STL内建了一些函数对象

分类：

- 算数仿函数
- 关系仿函数
- 逻辑仿函数

用法：

- 这些仿函数所产生的对象，用法和一般函数完全相同
- 使用内建函数对象，需要引入头文件

4.3.2 算数仿函数

功能描述：

- 实现四则运算
- 其中negate是一元运算，其余都是二元运算

仿函数原型：

```
template<class T> T plus<T>;//加法仿函数
template<class T> T minus<T>;//减法仿函数
template<class T> T multiplies<T>;//乘法仿函数
template<class T> T divide<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
```

```
#include<iostream>
using namespace std;
#include <functional>
//算数仿函数
void test01() {
    negate<int>n;//取反仿函数
    cout << n(10) << endl;

    plus<int>p;//加法仿函数
    cout << "10 + 20 = " << p(10, 20) << endl;
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

4.3.3 关系仿函数

仿函数原型：

```
template<class T>bool equal_to<T>;//等于
template<class T>bool not_equal_to<T>;//不等于
template<class T>bool greater<T>;//大于
template<class T>bool greater_equal<T>;//大于等于
template<class T>bool less<T>;//小于
template<class T>bool less_equal<T>;//小于等于
```

```
#include<iostream>
using namespace std;
#include <functional>
#include<vector>
#include <algorithm>
//关系仿函数
class MyCompare {
public:
    bool operator()(int v1, int v2) {
        return v1 > v2;
    }
};
void printVector(vector<int>&v) {
    for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}
void test01() {
    vector<int>v;
    v.push_back(10);
    v.push_back(50);
    v.push_back(40);
    v.push_back(20);
    v.push_back(70);
    printVector(v);
    //降序
    //sort(v.begin(), v.end(), MyCompare());
    //greater<int>()--内建函数对象
    sort(v.begin(), v.end(), greater<int>());
    printVector(v);

}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

4.3.4 逻辑仿函数

函数原型：

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

```
#include<iostream>
using namespace std;
#include <functional>
#include<vector>
#include <algorithm>
//逻辑仿函数
void printVector(vector<bool>&v) {
    for (vector<bool>::iterator it = v.begin(); it != v.end(); it++) {
        cout << *it << " ";
    }
    cout << endl;
}
void test01() {
    vector<bool>v;
    v.push_back(true);
    v.push_back(false);
    v.push_back(true);
    v.push_back(false);
    printVector(v);
    //利用逻辑非，将容器v搬运到容器v2中，并执行取反操作
    vector<bool>v2;
    v2.resize(v.size());
    transform(v.begin(), v.end(), v2.begin(), logical_not<bool>());
    printVector(v2);
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

5 STL-常用算法

概念：

- 算法主要是由头文件组成
- 是所有STL头文件中最大的一个，范围涉及到比较、交换、查找、遍历操作、复制、修改等
- 体积很小，只包括几个序列上面进行简单数学运算的模板函数
- 定义了一些模板类，用于声明函数对象

5.1 常用遍历算法

算法简介：

```
for_each    //遍历容器
transform   //搬运容器到另一个容器中
```

5.1.1 for_each

函数原型：

```
for_each(iterator beg,iterator end,,_func);
//遍历算法 遍历容器元素
//beg 开始迭代器
//end 结束迭代器
//_func 函数或者函数对象
```

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

//for_each

//普通函数
void print01(int val) {
    cout << val << " ";
}
//仿函数
class print02 {
public:
    void operator()(int val) {
        cout << val << " ";
    }
};
void test01() {
    vector<int>v;
    for (int i = 0; i < 10; i++) {
        v.push_back(i);
    }
    //for_each(v.begin(), v.end(), print01);
    for_each(v.begin(), v.end(), print02());
    cout << endl;
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

5.1.2 transform

功能描述：

- 搬运容器到另一个容器中

函数原型：

```
transform(iterator beg1,iterator end1,iterator beg2,_func);
//beg1  源容器开始迭代器
//end1  源容器结束迭代器
//beg2  目标容器开始迭代器
//_func 函数或者函数对象
```

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

//transform

class Transform {
public:
    int operator()(int val) {
        return val + 10;//搬运过程可以进行一些逻辑运算
    }
};
class MyPrint {
public:
    void operator()(int val) {
        cout << val << " ";
    }
};
void test01() {
    vector<int>v;
    for (int i = 0; i < 10; i++) {
        v.push_back(i);
    }
    vector<int>vTarget;//目标容器
    vTarget.resize(v.size());//目标容器需要提前开辟空间
    transform(v.begin(), v.end(), vTarget.begin(), Transform());
    for_each(vTarget.begin(), vTarget.end(), MyPrint());
    cout << endl;
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

5.2 常用查找算法

算法简介：

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

5.2.1 find

功能描述：

- 查找指定元素，找到返回值指定元素的迭代器，找不到返回结束迭代器end()

函数原型：

```
find(iterator beg,iterator end,value);
//按值查找元素，找到返回指定位置迭代器，找不到返回结束迭代器end()
//beg 开始迭代器
//end 结束迭代器
//value 查找的元素
```

```
#include<iostream>
using namespace std;
#include<vector>
#include <algorithm>
#include <string>
//find
class Person {
public:
    Person(string name, int age) {
        this->m_Name = name;
        this->m_Age = age;
    }
    //重载 == 让底层find知道如何比对Pers自定义数据类型
    bool operator==(const Person& p) {
        if (this->m_Name == p.m_Name && this->m_Age == p.m_Age) {
            return true;
        }
        return false;
    }
    string m_Name;
    int m_Age;
};

void test01() {
    //查找内置数据类型
    vector<int>v;
    for (int i = 0; i < 10; i++) {
        v.push_back(i);
    }
    vector<int>::iterator pos = find(v.begin(), v.end(), 3);
    if (pos != v.end()) {
        cout << "找到了元素: " << *pos << endl;
    }
    else {
        cout << "未找到该元素" << endl;
    }
    //查找自定义数据类型
    vector<Person>vp;
    Person p1("刘备", 35);
    Person p2("曹操", 45);
    Person p3("孙权", 40);
    Person p4("赵云", 25);
    vp.push_back(p1);
```

```
    vp.push_back(p2);
    vp.push_back(p3);
    vp.push_back(p4);

    Person pp("赵云", 25);
    vector<Person>::iterator posp = find(vp.begin(), vp.end(), pp);
    if (posp != vp.end()) {
        cout << "找到了元素--姓名：" << posp->m_Name << " 年龄：" << posp->m_Age
<< endl;
    }
    else {
        cout << "未找到该元素" << endl;
    }
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

5.2.2 find_if

功能描述：

- 按条件查找元素

函数原型：

```
find_if(iterator beg,iterator end,_Pred);
//按值查找元素，找到返回指定位置迭代器，找不到返回结束迭代器位置
//beg 开始迭代器
//end 结束迭代器
//_Pred 函数或者谓词
```

```
#include<iostream>
using namespace std;
#include<vector>
#include <algorithm>
#include <string>
//find_if
class GreaterFive {
public:
    bool operator()(int val) {
        if (val > 5) {
            return true;
        }
        return false;
    }
};
class Person {
public:
    Person(string name, int age) {
        this->m_Name = name;
        this->m_Age = age;
    }
    string m_Name;
```

```cpp
    int m_Age;
};
class Greater40 {
public:
    bool operator()(const Person& p) {
        if (p.m_Age >= 40) {
            return true;
        }
        return false;
    }
};
void test01() {
    //查找内置数据类型
    vector<int>v;
    for (int i = 0; i < 10; i++) {
        v.push_back(i);
    }
    vector<int>::iterator pos = find_if(v.begin(), v.end(), GreaterFive());
    if (pos != v.end()) {
        cout << "找到该元素: " << *pos << endl;
    }
    else {
        cout << "未找到该元素" << endl;
    }
    //查找自定义数据类型
    vector<Person>vp;
    Person p1("刘备", 35);
    Person p2("曹操", 45);
    Person p3("孙权", 40);
    Person p4("赵云", 25);
    vp.push_back(p1);
    vp.push_back(p2);
    vp.push_back(p3);
    vp.push_back(p4);

    //找年龄大于40的人
    vector<Person>::iterator posp = find_if(vp.begin(), vp.end(),
Greater40());
    if (posp != vp.end()) {
        cout << "找到该元素 姓名: " << posp->m_Name << " 年龄:  " << posp->m_Age
<< endl;
    }
    else {
        cout << "未找到该元素" << endl;
    }
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

5.2.3 adjacent_find

功能描述：

- 查找相邻重复元素

函数原型：

```
adjacent(iterator beg,iterator end);
//查找相邻重复元素，返回相邻元素的第一个位置的迭代器
//beg 开始迭代器
//end 结束迭代器
```

```
#include<iostream>
using namespace std;
#include<vector>
#include <algorithm>
//adjacent_find

void test01() {
    vector<int>v;
    v.push_back(0);
    v.push_back(1);
    v.push_back(2);
    v.push_back(3);
    v.push_back(2);
    v.push_back(3);
    v.push_back(3);
    v.push_back(4);

    vector<int>::iterator pos = adjacent_find(v.begin(), v.end());
    if (pos != v.end()) {
        cout << "找到相邻重复元素:" << *pos << endl;
    }
    else {
        cout << "未找到相邻重复元素" << endl;
    }
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

5.2.4 binary_search

功能描述：

- 查找指定元素是否存在

函数原型：

```
bool binary_search(iterator beg,iterator end,value);
//查找指定的元素，查到返回true，否则false
//注意：在无需序列中不可用
//beg 开始迭代器
//end 结束迭代器
//value 查找的元素
```

```
#include<iostream>
using namespace std;
#include<vector>
#include <algorithm>
//binary_search

void test01() {
    vector<int>v;
    for (int i = 0; i < 10; i++) {
        v.push_back(i);
    }
    //查找容器中是否有9元素
    //注意：容器必须是有序序列
    bool ret = binary_search(v.begin(), v.end(), 9);
    if (ret) {
        cout << "找到了该元素" << endl;
    }
    else {
        cout << "未找到该元素" << endl;
    }
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

5.2.5 count

功能描述：

- 统计元素个数

函数原型：

```
count(iterator beg,iterator end,value);
//统计元素出现次数
//beg 开始迭代器
//end 结束迭代器
//value 统计的元素
```

```
#include<iostream>
using namespace std;
#include<vector>
#include <algorithm>
#include <string>
//count
class Person {
```

```
public:
    Person(string name, int age) {
        this->m_Name = name;
        this->m_Age = age;
    }
    //重载==
    bool operator==(const Person& p) {
        return this->m_Age == p.m_Age;
    }
    string m_Name;
    int m_Age;
};
void test01() {
    //统计内置数据类型
    vector<int>v;
    for (int i = 0; i < 10; i++) {
        v.push_back(i);
        v.push_back(9);
    }
    int num = count(v.begin(), v.end(), 9);
    cout << "v中元素9的个数为：" << num << endl;
    //统计自定义数据类型
    vector<Person>vp;
    Person p1("刘备", 35);
    Person p2("曹操", 45);
    Person p3("孙权", 40);
    Person p4("赵云", 25);
    Person p5("曹操", 40);

    Person p("诸葛亮", 35);
    vp.push_back(p1);
    vp.push_back(p2);
    vp.push_back(p3);
    vp.push_back(p4);
    vp.push_back(p5);

    int cnt = count(v.begin(), v.end(), p);
    cout << "vp中与诸葛亮同岁的人有" << cnt << "个" << endl;
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

5.2.6 count_if

功能描述：

- 按条件统计元素个数

函数原型：

```
count_if(iterator beg,iterator end,_Pred);
//按条件统计元素出现次数
//beg 开始迭代器
//end 结束迭代器
//_Pred 谓词
```

```
#include<iostream>
using namespace std;
#include<vector>
#include <algorithm>
#include <string>
//count_if
class Greater20 {
public:
    bool operator()(int val) {
        return val > 20;
    }
};
class Person {
public:
    Person(string name, int age) {
        this->m_Name = name;
        this->m_Age = age;
    }
    int m_Age;
    string m_Name;
};
class AgeGreater20 {
public:
    bool operator()(const Person& p) {
        return p.m_Age > 20;
    }
};
void test01() {
    //统计内置数据类型
    vector<int>v;
    v.push_back(10);
    v.push_back(40);
    v.push_back(30);
    v.push_back(20);
    v.push_back(40);
    v.push_back(20);

    int num = count_if(v.begin(), v.end(), Greater20());
    cout << "v中大于20的元素个数" << num << endl;

    //统计自定义数据类型
    vector<Person>vp;
    Person p1("刘备", 35);
    Person p2("曹操", 45);
    Person p3("孙权", 40);
    Person p4("赵云", 25);
    Person p5("曹操", 40);
    vp.push_back(p1);
    vp.push_back(p2);
    vp.push_back(p3);
```

```
    vp.push_back(p4);
    vp.push_back(p5);

    int cnt = count_if(vp.begin(), vp.end(), AgeGreater20());
    cout << "vp中人物年龄大于20人数为：" << cnt << endl;
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

5.3 常用排序算法

算法简介：

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

5.3.1 sort

功能描述：

- 对容器内元素进行排序

函数原型：

```
sort(iterator beg,iterator end,_Pred);
//按值查找元素，找到返回指定位置迭代器，找不到返回结束迭代器位置
//beg 开始迭代器
//end 结束迭代器
//_Pred 谓词
```

```
#include<iostream>
using namespace std;
#include<vector>
#include <algorithm>
#include <functional>
//sort
void myPrint(int val) {
    cout << val << " ";
}
void test01() {
    vector<int>v;
    v.push_back(10);
    v.push_back(40);
    v.push_back(30);
    v.push_back(20);
    v.push_back(40);
    v.push_back(20);

    //利用sort进行升序
    sort(v.begin(),v.end());
```

```
    for_each(v.begin(), v.end(), myPrint);
    cout << endl;
    //改变为降序
    sort(v.begin(), v.end(), greater<int>());
    for_each(v.begin(), v.end(), myPrint);
    cout << endl;
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

5.3.2 random_shuffle

功能描述：

- 洗牌 指定范围内的元素随机调整次序

函数原型：

```
random_shuffle(iterator beg,iterator end);
//指定范围内的元素随机调整次序
//beg 开始迭代器
//end 结束迭代器
```

```
#include<iostream>
using namespace std;
#include<vector>
#include <algorithm>
#include<ctime>
//random_shuffle
void myPrint(int val) {
    cout << val << " ";
}
void test01() {
    vector<int>v;
    for (int i = 0; i < 10; i++) {
        v.push_back(i);
    }
    //利用洗牌算法，打乱顺序
    random_shuffle(v.begin(), v.end());
    for_each(v.begin(), v.end(), myPrint);
    cout << endl;
}
int main()
{
    srand((unsigned int)time(NULL));
    test01();
    system("pause");
    return 0;
}
```

5.3.3 merge

功能描述：

- 两个容器元素合并，并存储到另一容器中

函数原型：

```
merge(iterator beg1,iterator end1,iterator beg2,iterator end2,iterator dest);
//容器元素合并，并存储到另一个容器中
//注意：两个容器必须是有序的
//beg1 容器1开始迭代器
//end1 容器1结束迭代器
//beg2 容器2开始迭代器
//end2 容器2结束迭代器
//dest 目标容器开始迭代器
```

```
#include<iostream>
using namespace std;
#include<vector>
#include <algorithm>
//merge
class MyPrint {
public:
    void operator()(int val) {
        cout << val << " ";
    }
};
void test01() {
    vector<int>v1;
    vector<int>v2;
    for (int i = 0; i < 10; i++) {
        v1.push_back(i);
        v2.push_back(i + 10);
    }

    vector<int>vTarget;//目标容器
    vTarget.resize(v1.size() + v2.size());//提前分配空间

    merge(v1.begin(), v1.end(), v2.begin(), v2.end(), vTarget.begin());
    for_each(vTarget.begin(), vTarget.end(), MyPrint());
    cout << endl;
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

5.3.4 reverse

功能描述：

- 将容器内元素进行反转

函数原型：

```
reverse(iterator beg,iterator end);
//反转指定范围内的元素
//beg 开始迭代器
//end 结束迭代器
```

```
#include<iostream>
using namespace std;
#include<vector>
#include <algorithm>
//reverse
class MyPrint {
public:
    void operator()(int val) {
        cout << val << " ";
    }
};
void test01() {
    vector<int>v;
    for (int i = 0; i < 10; i++) {
        v.push_back(i);
    }
    for_each(v.begin(), v.end(), MyPrint());
    cout << endl;
    reverse(v.begin(), v.end());
    for_each(v.begin(), v.end(), MyPrint());
    cout << endl;
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

5.4 常用拷贝和替换算法

算法简介：

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

5.4.1 copy

功能描述：

- 容器内指定范围的元素拷贝到另一容器中

函数原型：

```
copy(iterator beg,iterator end,iterator dest);
//按值查找元素，找到返回指定位置迭代器，找不到返回结束迭代器位置
//beg 开始迭代器
//end 结束迭代器
//dest 目标容器开始迭代器
```

```
#include<iostream>
using namespace std;
#include<vector>
#include <algorithm>
//copy
class MyPrint {
public:
    void operator()(int val) {
        cout << val << " ";
    }
};
void test01() {
    vector<int>v;
    for (int i = 0; i < 10; i++) {
        v.push_back(i);
    }
    vector<int>v2;
    v2.resize(v.size());
    copy(v.begin(), v.end(), v2.begin());
    for_each(v2.begin(), v2.end(), MyPrint());
    cout << endl;
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

5.4.2 replace

功能描述：

- 将容器内指定范围的旧元素修改为新元素

函数原型：

```
replace(iterator beg,iterator end,oldvalue,newvalue);
//将区间内的旧元素替换成新元素
//beg 开始迭代器
//end 结束迭代器
//oldvalue 旧元素
//newvalue 新元素
```

```
#include<iostream>
using namespace std;
#include<vector>
#include <algorithm>
//replace
class MyPrint {
```

```
public:
    void operator()(int val) {
        cout << val << " ";
    }
};
void test01() {
    vector<int>v;
    for (int i = 0; i < 10; i++) {
        v.push_back(i);
    }
    for_each(v.begin(), v.end(), MyPrint());
    cout << endl;
    replace(v.begin(), v.end(), 1, 2);//容器中1换成2
    for_each(v.begin(), v.end(), MyPrint());
    cout << endl;
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

5.4.3 replace_if

功能描述：

- 容器内指定范围满足条件的元素替换为新元素

函数原型：

```
replaace_if(iterator beg,iterator end,_Pred,newvalue);
//按条件替换元素，满足条件的替换成指定元素
//beg 开始迭代器
//end 结束迭代器
//_Pred 谓词
//newvalue 新元素
```

```
#include<iostream>
using namespace std;
#include<vector>
#include <algorithm>
//replace
class MyPrint {
public:
    void operator()(int val) {
        cout << val << " ";
    }
};
class Greater5 {
public:
    bool operator()(int val) {
        return val > 5;
    }
};
void test01() {
    vector<int>v;
```

```
    for (int i = 0; i < 10; i++) {
        v.push_back(i);
    }
    for_each(v.begin(), v.end(), MyPrint());
    cout << endl;
    replace_if(v.begin(), v.end(), Greater5(), 0);
    for_each(v.begin(), v.end(), MyPrint());
    cout << endl;
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

5.4.4 swap

功能描述：

- 互换两个容器的元素

函数原型：

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

```
#include<iostream>
using namespace std;
#include<vector>
#include <algorithm>
//replace_if
class MyPrint {
public:
    void operator()(int val) {
        cout << val << " ";
    }
};
class Greater5 {
public:
    bool operator()(int val) {
        return val > 5;
    }
};
void test01() {
    vector<int>v1;
    vector<int>v2;
    for (int i = 0; i < 10; i++) {
        v1.push_back(i);
        v2.push_back(i + 10);
    }
    cout << "互换前：" << endl;
    for_each(v1.begin(), v1.end(), MyPrint());
    cout << endl;
    for_each(v2.begin(), v2.end(), MyPrint());
```

```
    cout << endl;
    swap(v1, v2);
    cout << "互换后：" << endl;
    for_each(v1.begin(), v1.end(), MyPrint());
    cout << endl;
    for_each(v2.begin(), v2.end(), MyPrint());
    cout << endl;
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

5.5 常用算数生成函数

算法简介：

```
accumulate  //计算容器元素累计总和
fill    //向容器中添加元素
```

注意：

- 算术生成算法属于小型算法，使用时包含的头文件为

5.5.1 accumlate

功能描述：

- 计算区间内容器元素的总和

函数原型：

```
accumulate(iterator beg,iterator end,value);
//计算容器元素累计总和
//beg 开始迭代器
//end 结束迭代器
//value 起始值
```

```
#include<iostream>
using namespace std;
#include<vector>
#include <numeric>
//accumulate
void test01() {
    vector<int>v;
    for (int i = 0; i <= 100; i++) {
        v.push_back(i);
    }
    //参数3  起始累加值
    int sum = accumulate(v.begin(), v.end(), 0);
    cout << "sum = " << sum << endl;
}
int main()
{
```

```
    test01();
    system("pause");
    return 0;
}
```

5.5.2 fill

功能描述：

- 向容器中填充指定的值

函数原型：

```
fill(iterator beg,iterator end,value);
//向容器中添加元素
//beg 开始迭代器
//end 结束迭代器
//value 填充的值
```

```
#include<iostream>
using namespace std;
#include<vector>
#include <numeric>
#include <algorithm>
//fill
class MyPrint {
public:
    void operator()(int val) {
        cout << val << " ";
    }
};

void test01() {
    vector<int>v;
    v.resize(10);
    for_each(v.begin(), v.end(), MyPrint());
    cout << endl;
    fill(v.begin(), v.end(), 1);
    for_each(v.begin(), v.end(), MyPrint());
    cout << endl;
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

5.6 常用集合算法

算法简介：

```
set_intersection    //求两个容器的交集
set_union   //求两个容器的并集
set_difference  //求两个容器的差集
```

5.6.1 set_intersection

功能描述：

- 求两个容器的交集

函数原型：

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

```
#include<iostream>
using namespace std;
#include<vector>
#include <algorithm>
//set_intersaction
class MyPrint {
public:
    void operator()(int val) {
        cout << val << " ";
    }
};

void test01() {
    vector<int>v1;
    vector<int>v2;
    for (int i = 0; i < 10; i++) {
        v1.push_back(i);
        v2.push_back(i + 5);
    }
    vector<int>vTarget;
    vTarget.resize(min(v1.size(), v2.size()));//提前开辟空间
    vector<int>::iterator itEnd = set_intersection(v1.begin(), v1.end(), v2.begin(), v2.end(), vTarget.begin());
    for_each(vTarget.begin(), itEnd, MyPrint());
    cout << endl;
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

5.6.2 set_union

功能描述：

- 求两个集合的并集

函数原型：

```
set_union(iterator beg1,iterator end1,iterator beg2,iterator end2,iterator dest);
//求两个集合的并集
//注意：两个集合必须是有序序列
//beg1 容器1开始迭代器
//end1 容器1结束迭代器
//beg2 容器2开始迭代器
//end2 容器2结束迭代器
//dest 目标容器开始迭代器
```

```
#include<iostream>
using namespace std;
#include<vector>
#include <algorithm>
//set_union
class MyPrint {
public:
    void operator()(int val) {
        cout << val << " ";
    }
};

void test01() {
    vector<int>v1;
    vector<int>v2;
    for (int i = 0; i < 10; i++) {
        v1.push_back(i);
        v2.push_back(i + 5);
    }
    vector<int>vTarget;
    vTarget.resize(v1.size()+v2.size());//提前开辟空间
    vector<int>::iterator itEnd = set_union(v1.begin(), v1.end(), v2.begin(), v2.end(), vTarget.begin());
    for_each(vTarget.begin(), itEnd, MyPrint());
    cout << endl;
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

5.6.3 set_difference

功能描述：

- 求两个集合的差集

函数原型：

```
set_difference(iterator beg1,iterator end1,iterator beg2,iterator end2,iterator dest);
//求两个集合的交集
//注意：两个集合必须是有序序列
//beg1 容器1开始迭代器
//end1 容器1结束迭代器
//beg2 容器2开始迭代器
//end2 容器2结束迭代器
//dest 目标容器开始迭代器
```

```
#include<iostream>
using namespace std;
#include<vector>
#include <algorithm>
//set_union
class MyPrint {
public:
    void operator()(int val) {
        cout << val << " ";
    }
};

void test01() {
    vector<int>v1;
    vector<int>v2;
    for (int i = 0; i < 10; i++) {
        v1.push_back(i);
        v2.push_back(i + 5);
    }
    vector<int>vTarget;
    vTarget.resize(max(v1.size(),v2.size()));//提前开辟空间
    cout << "v1和v2的差集为：" << endl;
    vector<int>::iterator itEnd = set_difference(v1.begin(), v1.end(), v2.begin(), v2.end(), vTarget.begin());
    for_each(vTarget.begin(), itEnd, MyPrint());
    cout << endl;
    cout << "v2和v1的差集为：" << endl;
    itEnd = set_difference(v2.begin(), v2.end(), v1.begin(), v1.end(), vTarget.begin());
    for_each(vTarget.begin(), itEnd, MyPrint());
    cout << endl;
}
int main()
{
    test01();
    system("pause");
    return 0;
}
```

总结：

- 集合算法的两个容器必须是有序序列
- 目标容器提前开辟的空间大小需要考虑特殊情况
- 利用集合算法的返回值输出集合算法的结果