

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：

```
1  template<typename T>  
2  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

```
1  #include <iostream>  
2  using namespace std;  
3  
4  //函数模板  
5  //交换函数模板  
6  template <typename T>  
7  void mySwap(T& a, T& b) {  
8      T temp = a;  
9      a = b;  
10     b = temp;  
11 }  
12 void test01() {  
13     //函数模板调用  
14     int a = 10;  
15     int b = 20;
```

```

16 //1、自动类型推导
17 mySwap(a, b);
18 //2、显示指定类型
19 mySwap<int>(a, b);
20 cout << "a = " << a << endl;
21 cout << "b = " << b << endl;
22 }
23 int main()
24 {
25     test01();
26     system("pause");
27     return 0;
28 }

```

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

```

1  #include <iostream>
2  using namespace std;
3
4  //函数模板注意事项
5  //交换函数模板
6  template <typename T>
7  void mySwap(T& a, T& b) {
8      T temp = a;
9      a = b;
10     b = temp;
11 }
12 void test01() {
13     int a = 10;
14     int b = 20;
15     char c = 'c';
16     //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
17     //mySwap(a, c);错误
18     mySwap(a, b);
19     cout << "a = " << a << endl;
20     cout << "b = " << b << endl;
21 }
22 //2、函数模板必须确定数据类型T才可以正常使用
23 template <typename T>
24 void func() {
25     cout << "func()函数的调用" << endl;
26 }
27 void test02() {
28     func<int>(); //显示指定类型
29 }
30 int main()
31 {
32     test01();
33
34     system("pause");
35     return 0;
36 }

```

1.2.3 函数模板案例

- 不同数据类型数组排序
- 选择排序

```
1  #include <iostream>
2  using namespace std;
3
4  template <typename T>
5  void mySwap(T& a, T& b) {
6      T temp = a;
7      a = b;
8      b = temp;
9  }
10 //排序算法
11 template <typename T>
12 void mySort(T arr[], int len) {
13     for (int i = 0; i < len; i++) {
14         max = i; //假定的最大值
15         for (int j = i + 1; j < len; j++) {
16             if (arr[j] > arr[max]) {
17                 max = j;
18             }
19         }
20         if (max != i) {
21             mySwap(arr[max], arr[i]);
22         }
23     }
24 }
25 template <typename T>
26 void myPrint(T arr[], int len) {
27     for (int i = 0; i < len; i++) {
28         cout << arr[i] << " ";
29     }
30     cout << endl;
31 }
32 void test01() {
33     //测试char数组
34     char charArr[] = "bshdfisfd";
35     int num = sizeof(charArr) / sizeof(char);
36     mySort(charArr, num);
37     myPrint(charArr, num);
38     //测试int数组
39     int intArr[] = { 2,5,4,8,6,7,3 };
40     int Num = sizeof(intArr) / sizeof(int);
41     mySort(intArr, Num);
42     myPrint(intArr, Num);
43 }
44 int main() {
45
46     test01();
47     system("pause");
48     return 0;
49 }
```

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

```
1  #include <iostream>
2  using namespace std;
3
4  //普通函数和函数模板的区别
5  int myAdd01(int a,int b) {
6      return a + b;
7  }
8  template <class T>
9  T myAdd02(T a, T b) {
10     return a + b;
11 }
12 void test01() {
13     int a = 10;
14     int b = 20;
15     char c = 'c';
16     cout << myAdd01(a, c) << endl; //普通函数可以发生隐式类型转换
17     //自动类型推导
18     //cout << myAdd02(a, c) << endl; //不会发生隐式类型转换
19     //显示指定类型
20     cout << myAdd02<int>(a, c) << endl; //发生自动类型转换
21 }
22 int main() {
23
24     test01();
25     system("pause");
26     return 0;
27 }
```

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

```
1  #include <iostream>
2  using namespace std;
3
4  //普通函数和函数模板的调用规则
5  void myPrint(int a, int b) {
6      cout << "普通函数的调用" << endl;
7  }
8  template <typename T>
9  void myPrint(T a, T b) {
10     cout << "函数模板调用" << endl;
11 }
```

```

11 }
12 //函数模板重载
13 template <typename T>
14 void myPrint(T a, T b, T c) {
15     cout << "重载函数模板调用" << endl;
16 }
17 void test01() {
18     int a = 10;
19     int b = 20;
20     char c1 = 'a';
21     char c2 = 'b';
22     myPrint(a, b);
23     //空模板的参数列表强制调用函数模板
24     myPrint<>(a, b);
25     myPrint(a, b, 100);
26     //更好的匹配
27     myPrint(c1, c2);
28 }
29 int main() {
30
31     test01();
32     system("pause");
33     return 0;
34 }

```

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

```

1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  class Person {
6  public:
7      Person(string name, int age) {
8          this->m_Age = age;
9          this->m_Name = name;
10     }
11     int m_Age;
12     string m_Name;
13 };
14 //模板的局限性
15 template <typename T>
16 bool myCompare(T& a, T& b) {
17     if (a == b) {
18         return true;
19     }
20     return false;
21 }
22 //具体化版本的函数模板
23 template<>bool myCompare(Person& p1, Person& p2) {
24     if (p1.m_Name == p2.m_Name && p1.m_Age == p2.m_Age) {
25         return true;
26     }
27     return false;

```

```

28 }
29 void test01() {
30     int a = 10;
31     int b = 20;
32     bool ret = myCompare(a, b);
33     if (ret) {
34         cout << "a == b" << endl;
35     }
36     else {
37         cout << "a != b" << endl;
38     }
39 }
40 void test02() {
41     Person p1("Tom", 18);
42     Person p2("Tom", 18);
43     bool ret = myCompare(p1, p2);
44     if (ret) {
45         cout << "p1 == p2" << endl;
46     }
47     else {
48         cout << "p1 != p2" << endl;
49     }
50 }
51 int main() {
52
53     test01();
54     test02();
55     system("pause");
56     return 0;
57 }

```

1.3 类模板

1.3.1 类模板语法

语法:

```

1  template <typename T>
2  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

```

1  #include <iostream>
2  using namespace std;
3  #include <string>
4  //类模板
5  template<class NameType, class AgeType>
6  class Person {
7  public:
8      Person(NameType name, AgeType age) {
9          this->m_Age = age;

```

```

10         this->m_Name = name;
11     }
12     void showPerson() {
13         cout << "name:" << this->m_Name << endl;
14         cout << "age:" << this->m_Age << endl;
15     }
16     NameType m_Name;
17     AgeType m_Age;
18 };
19 void test01() {
20     Person<string, int>p1("Tom", 18);
21     p1.showPerson();
22 }
23 int main() {
24     test01();
25     system("pause");
26     return 0;
27 }

```

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

```

1  #include <iostream>
2  using namespace std;
3  #include <string>
4  //类模板与函数模板的区别
5  template<class NameType, class AgeType = int> //类模板的模板参数列表中可以有默认参数
6  class Person {
7  public:
8      Person(NameType name, AgeType age) {
9          this->m_Age = age;
10         this->m_Name = name;
11     }
12     void showPerson() {
13         cout << "name:" << this->m_Name << endl;
14         cout << "age:" << this->m_Age << endl;
15     }
16     NameType m_Name;
17     AgeType m_Age;
18 };
19 void test01() {
20     //Person p("Tom", 18); 类模板没有自动类型推导
21     Person<string, int>p("Tom", 18); //显示指定类型
22     p.showPerson();
23 }
24 void test02() {
25     Person <string>p("Jerry", 18);
26     p.showPerson();
27 }
28 int main() {
29     test01();
30     test02();
31     system("pause");

```

```
32     return 0;
33 }
```

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

```
1  #include <iostream>
2  using namespace std;
3  #include <string>
4  //类模板中成员函数的创建时机
5  class Person1 {
6  public:
7      void showPerson1() {
8          cout << "Person1 show" << endl;
9      }
10 };
11 class Person2 {
12 public:
13     void showPerson2() {
14         cout << "Person2 show" << endl;
15     }
16 };
17 template<class T>
18 class MyClass {
19 public:
20     T obj;
21     //类模板中的成员函数
22     void func1() {
23         obj.showPerson1();
24     }
25     void func2() {
26         obj.showPerson2();
27     }
28 };
29 void test01() {
30     MyClass <Person1>m;
31     m.func1();
32     //m.func2();调用时创建,
33     MyClass<Person2>n;
34     n.func2();
35     //n.func1();调用时创建
36 }
37 int main() {
38     test01();
39     system("pause");
40     return 0;
41 }
```

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

```
1  #include <iostream>
2  using namespace std;
3  #include <string>
4  //类模板对象做函数参数
5  template<class NameType, class AgeType>
6  class Person {
7  public:
8      Person(NameType name, AgeType age) {
9          this->m_Name = name;
10         this->m_Age = age;
11     }
12     void showPerson() {
13         cout << "name:" << this->m_Name << endl;
14         cout << "age:" << this->m_Age << endl;
15     }
16     NameType m_Name;
17     AgeType m_Age;
18 };
19 //1、指定传入类型
20 void printPerson1(Person <string, int>& p){
21     p.showPerson();
22 }
23 //2、参数模板化
24 template<class NameType, class AgeType>
25 void printPerson2(Person<NameType, AgeType>& p) {
26     p.showPerson();
27     //模板类型
28     cout << "NameType = " << typeid(NameType).name() << endl;
29     cout << "AgeType = " << typeid(AgeType).name() << endl;
30 }
31 //3、整个类模板化
32 template<class T>
33 void printPerson3(T& p) {
34     p.showPerson();
35     cout << "T = " << typeid(T).name() << endl;
36 }
37 void test01() {
38     Person<string, int>p("Tom", 18);
39     printPerson1(p);
40     printPerson2(p);
41     printPerson3(p);
42 }
43 int main() {
44     test01();
45     system("pause");
46     return 0;
47 }
```

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

```
1  #include <iostream>
2  using namespace std;
3  #include <string>
4  //类模板与继承
5  template<class T>
6  class Base {
7  public:
8      T m;
9  };
10 class Son1 :public Base<int> {}; //指定父类的类型
11 //灵活指定父类中的类型---子类写成类模板
12 template<class T1, class T2>
13 class Son2:public Base<T2>{
14 public:
15     Son2() {
16         cout << "T1 = " << typeid(T1).name() << endl;
17         cout << "T2 = " << typeid(T2).name() << endl;
18     }
19     T1 obj;
20 };
21 void test01() {
22     //Son1 s1;
23     Son2<int, char>s2;
24 }
25 int main() {
26     test01();
27     system("pause");
28     return 0;
29 }
```

1.3.6 类模板成员函数类外实现

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4  //类模板中成员函数类外实现
5  template<class T1, class T2>
6  class Person {
7  public:
8      Person(T1 name, T2 age);
9      /*{
10         this->m_Age = age;
11         this->m_Name = name;
12     }*/
13     void showPerson();
14     /*{
15         cout << "name:" << this->m_Name << endl;
16         cout << "age:" << this->m_Age << endl;
17     }
```

```

18     */
19     T1 m_Name;
20     T2 m_Age;
21 };
22 //构造函数的类外实现
23 template<class T1, class T2>
24 Person<T1,T2>::Person(T1 name, T2 age) {
25     this->m_Age = age;
26     this->m_Name = name;
27 }
28 //成员函数的类外实现
29 template<class T1, class T2>
30 void Person<T1, T2>::showPerson() {
31     cout << "name:" << this->m_Name << endl;
32     cout << "age:" << this->m_Age << endl;
33 }
34 void test01() {
35     Person<string, int>p("Tom", 18);
36     p.showPerson();
37 }
38 int main()
39 {
40     test01();
41     system("pause");
42     return 0;
43 }

```

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

```

1  ///person.hpp
2  #pragma once
3  #include <iostream>
4  using namespace std;
5  #include <string>
6  template<class T1, class T2>
7  class Person {
8  public:
9      Person(T1 name, T2 age);
10     void showPerson();
11     T1 m_Name;
12     T2 m_Age;
13 };
14 // #include "person.h"
15 template<class T1, class T2>
16 Person<T1, T2>::Person(T1 name, T2 age) {
17     this->m_Name = name;

```

```

18     this->m_Age = age;
19 }
20 template<class T1, class T2>
21 void Person<T1, T2>::showPerson() {
22     cout << "name: " << this->m_Name << endl;
23     cout << "age: " << this->m_Age << endl;
24 }

```

```

1 //main.cpp
2 #include <iostream>
3 #include <string>
4 using namespace std;
5 //类模板分文件编写
6 //2、将.h和.cpp中的内容写到一起，将后缀名改为.hpp文件(主流)
7 #include "person.hpp"
8 void test01() {
9     Person<string, int>p("Tom", 18);
10    p.showPerson();
11 }
12 int main()
13 {
14     test01();
15     system("pause");
16     return 0;
17 }

```

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

```

1 #include <iostream>
2 #include <string>
3 using namespace std;
4 //类模板和友元
5 template<class T1, class T2>
6 class Person;           //提前让编译器知道
7 //类外实现
8 template<class T1, class T2>
9 void printPerson2(Person<T1, T2>p) {
10     cout << "类外实现: " << endl;
11     cout << "name: " << p.m_Name << endl;
12     cout << "age: " << p.m_Age << endl;
13 }
14 //通过全局函数
15 template<class T1, class T2>
16 class Person {
17     //全局函数类内实现
18     friend void printPerson(Person<T1, T2>p) {
19         cout << "类内实现: " << endl;
20         cout << "name: " << p.m_Name << endl;
21         cout << "age: " << p.m_Age << endl;
22     }
23     //全局函数类外实现
24     //加一个空模板的参数列表
25     //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

```

```

26     friend void printPerson2<>(Person<T1, T2>p);
27
28 public:
29     Person(T1 name, T2 age) {
30         this->m_Name = name;
31         this->m_Age = age;
32     }
33 private:
34     T1 m_Name;
35     T2 m_Age;
36 };
37 void test01() {
38     Person<string, int>p1("Tom", 18);
39     printPerson(p1);
40     Person<string, int>p2("Jerry", 18);
41     printPerson2(p2);
42 }
43 int main()
44 {
45     test01();
46     system("pause");
47     return 0;
48 }

```

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

```

1 //MyArray.hpp
2 //通用的数组类
3 #pragma once
4 #include <iostream>
5 using namespace std;
6
7 template<class T>
8 class MyArray {
9 public:
10     MyArray(int capacity) {
11         //cout << "MyArray有参构造函数调用" << endl;
12         this->m_Capacity = capacity;
13         this->m_Size = 0;

```

```

14     this->pAddress = new T[this->m_Capacity];
15 }
16
17 MyArray(const MyArray& arr) {
18     //cout << "MyArray拷贝构造函数调用" << endl;
19     this->m_Capacity = arr.m_Capacity;
20     this->m_Size = arr.m_Size;
21     this->pAddress = new T[arr.m_Capacity]; //深拷贝
22     //拷贝arr中的数据
23     for (int i = 0; i < this->m_Size; i++) {
24         this->pAddress[i] = arr.pAddress[i];
25     }
26 }
27
28 MyArray& operator=(const MyArray& arr) {
29     //cout << "MyArray&operator函数调用" << endl;
30     //判断原来堆区是否有数据，如果有先释放
31     if (this->pAddress != NULL) {
32         delete[] this->pAddress;
33         this->pAddress = NULL;
34         this->m_Capacity = 0;
35         this->m_Size = 0;
36     }
37     this->m_Capacity = arr.m_Capacity;
38     this->m_Size = arr.m_Size;
39     this->pAddress = new T[arr.m_Capacity];
40     for (int i = 0; i < this->m_Size; i++) {
41         this->pAddress[i] = arr.pAddress[i];
42     }
43     return *this;
44 }
45 //尾插法
46 void Push_Back(const T& value) {
47     //判断容量是否等于大小
48     if (this->m_Capacity == this->m_Size) {
49         cout << "容量已满，无法插入" << endl;
50         return;
51     }
52     this->pAddress[this->m_Size] = value; //数组末尾插入数据
53     this->m_Size++; //更新数组大小
54 }
55 //尾删法
56 void Pop_Back() {
57     //让用户访问不到最后一个元素，逻辑删除
58     if (this->m_Size == 0) {
59         cout << "数组为空，无法删除" << endl;
60         return;
61     }
62     this->m_Size--;
63 }
64 //通过下标方式访问数组中元素，作为左值存在返回
65 T& operator[](int index) {
66     return this->pAddress[index];
67 }
68
69 //返回数组的容量
70 int getCapacity() { return this->m_Capacity; }
71 //返回数组大小

```

```

72     int getSize() { return this->m_Size; }
73     ~MyArray() {
74         if (this->pAddress != NULL) {
75             //cout << "MyArray析构函数调用" << endl;
76             delete[] this->pAddress;
77             this->pAddress = NULL;
78         }
79     }
80 private:
81     T* pAddress;//指针指向堆区开辟的真实数组
82     int m_Capacity;//容量
83     int m_Size;
84 };

```

```

1  //main.cpp
2  #include<iostream>
3  #include <string>
4  using namespace std;
5  #include "MyArray.hpp"
6
7  void printIntArray(MyArray<int>& arr) {
8      for (int i = 0; i < arr.getSize(); i++) {
9          cout << arr[i] << endl;
10     }
11 }
12 void test01() {
13     MyArray<int> arr1(5);
14     for (int i = 0; i < 5; i++) {
15         arr1.Push_Back(1);//尾插法想数组中插入数据
16     }
17     cout << "arr1的打印输出为: " << endl;
18     printIntArray(arr1);
19     cout << "arr1的容量 = " << arr1.getCapacity() << endl;
20     cout << "arr1的大小 = " << arr1.getSize() << endl;
21
22     MyArray <int>arr2(arr1);
23     cout << "arr2的打印输出为: " << endl;
24     printIntArray(arr2);
25     //尾删
26     arr2.Pop_Back();
27     cout << "arr2尾删后: " << endl;
28     cout << "arr2的容量 = " << arr2.getCapacity() << endl;
29     cout << "arr2的大小 = " << arr2.getSize() << endl;
30 }
31 //测试自定义数据类型
32 class Person {
33 public:
34     Person() {}
35     Person(string name, int age) {
36         this->m_Name = name;
37         this->m_Age = age;
38     }
39     string m_Name;
40     int m_Age;
41 };
42 void printPerson(MyArray<Person>& arr) {
43     for (int i = 0; i < arr.getSize(); i++) {

```

```

44         cout << "name:" << arr[i].m_Name << endl;
45         cout << "age:" << arr[i].m_Age << endl;
46     }
47 }
48 void test02() {
49     MyArray<Person>arr(10);
50     Person p1("Tom", 18);
51     Person p2("Jerry", 20);
52     Person p3("Nancy", 20);
53     Person p4("Jack", 20);
54     Person p5("Jonh", 20);
55     arr.Push_Back(p1);
56     arr.Push_Back(p2);
57     arr.Push_Back(p3);
58     arr.Push_Back(p4);
59     arr.Push_Back(p5);
60     printPerson(arr);
61     cout << "arr的容量 = " << arr.getCapacity() << endl;
62     cout << "arr的大小 = " << arr.getSize() << endl;
63 }
64 int main() {
65     test01();
66     test02();
67     system("pause");
68     return 0;
69 }

```

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**

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

```

6 void myPrint(int value) {
7     cout << value << endl;
8 }
9 void test01() {
10     //创建一个vector容器
11     vector<int>v;
12     //向容器中插入数据
13     v.push_back(10);
14     v.push_back(20);
15     v.push_back(30);
16     v.push_back(40);
17     //通过迭代器访问容器中的数据
18     vector<int>::iterator itBegin = v.begin();//起始迭代器, 指向容器中第一个元素
19     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
20     //第一种遍历方式
21     while (itBegin != itEnd) {
22         cout << *itBegin << endl;
23         itBegin++;
24     }
25     //第二种遍历
26     for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
27         cout << *it << endl;
28     }
29     //第三种遍历方式, 利用STL提供的遍历算法
30     for_each(v.begin(), v.end(), myPrint);//回调
31 }
32
33 int main()
34 {
35     test01();
36     system("pause");
37     return 0;
38 }

```

2.5.2 Vector存放自定义数据类型

```

1 #include <iostream>
2 using namespace std;
3 #include <string>
4 #include <vector>
5 //vector容器中存放自定义数据类型
6 class Person {
7 public:
8     Person(string name, int age) {
9         this->m_Name = name;
10        this->m_Age = age;
11    }
12
13    string m_Name;
14    int m_Age;
15 };
16 void test01() {
17     vector<Person>v;
18     Person p1("Tom", 18);
19     Person p2("Jerry", 20);
20     Person p3("Nancy", 20);

```

```

21     Person p4("Jack", 20);
22     Person p5("Jonh", 20);
23     //向容器中添加数据
24     v.push_back(p1);
25     v.push_back(p2);
26     v.push_back(p3);
27     v.push_back(p4);
28     v.push_back(p5);
29     //遍历容器中的数据
30     for (vector<Person>::iterator it = v.begin(); it != v.end(); it++) {
31         /*cout << "name: " << (*it).m_Name << "\t"
32             << "age: " << (*it).m_Age << endl;*/
33         cout << "name: " << it->m_Name << "\t"
34             << "age: " << it->m_Age << endl;
35     }
36 }
37 void test02() {
38     vector<Person*>v;
39     Person p1("Tom", 18);
40     Person p2("Jerry", 20);
41     Person p3("Nancy", 20);
42     Person p4("Jack", 20);
43     Person p5("Jonh", 20);
44     //向容器中添加数据
45     v.push_back(&p1);
46     v.push_back(&p2);
47     v.push_back(&p3);
48     v.push_back(&p4);
49     v.push_back(&p5);
50     //遍历容器
51     for (vector<Person*>::iterator it = v.begin(); it != v.end(); it++) {
52         cout << "name: " << (*it)->m_Name << "\t"
53             << "age: " << (*it)->m_Age << endl;
54     }
55 }
56 int main()
57 {
58     //test01();
59     test02();
60     system("pause");
61     return 0;
62 }

```

2.5.3 Vector容器嵌套容器

```

1  #include <iostream>
2  using namespace std;
3  #include <vector>
4
5  //容器嵌套容器
6  void test01() {
7      vector<vector<int>>>v;
8      //创建小容器
9      vector<int>v1;
10     vector<int>v2;
11     vector<int>v3;
12     vector<int>v4;

```

```

13     for (int i = 0; i < 4; i++) {
14         v1.push_back(i + 1);
15         v2.push_back(i + 2);
16         v3.push_back(i + 3);
17         v4.push_back(i + 4);
18     }
19     //放入大容器
20     v.push_back(v1);
21     v.push_back(v2);
22     v.push_back(v3);
23     v.push_back(v4);
24     //遍历大容器
25     for (vector<vector<int>>::iterator it = v.begin(); it != v.end(); it++)
26     {
27         for (vector<int>::iterator vit = (*it).begin(); vit != (*it).end();
28             vit++) {
29             cout << *vit << " ";
30         }
31         cout << endl;
32     }
33 }
34 int main() {
35     test01();
36     system("pause");
37     return 0;
38 }

```

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：

```

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

```

```

1  #include <iostream>
2  using namespace std;
3  #include <string>
4
5  //string构造函数
6  void test01() {
7      string s1; //默认构造
8
9      const char* str = "Hello world";
10     string s2(str);
11     cout << "s2 = " << s2 << endl;
12
13     string s3(s2);
14     cout << "s3 = " << s3 << endl;
15
16     string s4(10, 'a');
17     cout << "s4 = " << s4 << endl;
18 }
19 int main()
20 {
21     test01();
22     system("pause");
23     return 0;
24 }

```

3.1.3 string赋值操作

赋值的函数原型

1	string& operator=(const char* s);	//char*类型字符串赋值给当前字符串
2	string& operator=(const string& s);	//把字符串s赋给当前的字符串
3	string& operator=(char c);	//字符赋值给当前字符串
4	string& assign(const char* s);	//把字符串s赋给当前的字符串
5	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
6	string& assign(const string& s);	//把字符串s赋值给当前字符串
7	string& assign(int n, char c);	//把n个字符c赋给当前字符串

```

1  #include <iostream>
2  using namespace std;
3  #include <string>
4
5  //string赋值函数
6  void test01() {
7      string str1;
8      str1 = "Hello world";
9      cout << "str1 = " << str1 << endl;
10
11     string str2;
12     str2 = str1;
13     cout << "str2 = " << str2 << endl;
14
15     string str3;
16     str3 = 'a';
17     cout << "str3 = " << str3 << endl;
18
19     string str4;

```

```

20     str4.assign("HelloC++");
21     cout << "str4 = " << str4 << endl;
22
23     string str5;
24     str5.assign("HelloC++",5);
25     cout << "str5 = " << str5 << endl;
26
27     string str6;
28     str6.assign(str5);
29     cout << "str6 = " << str6 << endl;
30
31     string str7;
32     str7.assign(10, 'a');
33     cout << "str7 = " << str7 << endl;
34 }
35 int main()
36 {
37     test01();
38     system("pause");
39     return 0;
40 }

```

3.1.4 string字符串拼接

函数原型:

```

1 string& operator+=(const char* str);
2 string& operator+=(const char c);
3 string& operator+=(const string& str);
4 string& append(const char* s);           //把字符串s连接到当前字符串结尾
5 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串
    结尾
6 string& append(const string& s);         //同operator+=(const string& str)
7 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

```

```

1 #include <iostream>
2 using namespace std;
3 #include <string>
4
5 //string字符串拼接
6 void test01() {
7     string str1 = "I ";
8     str1 += "Love You";
9     cout << "str1 = " << str1 << endl;
10
11     str1 += ':';
12     cout << "str1 = " << str1 << endl;
13     string str2 = "MC,LOL";
14     str1 += str2;
15     cout << "str1 = " << str1 << endl;
16
17     string str3 = "I ";
18     str3.append("Love");
19     cout << "str3 = " << str3 << endl;
20

```

```

21     str3.append("game abcde", 4);
22     cout << "str3 = " << str3 << endl;
23
24     //str3.append(str2, 0, 2); //只截取MC
25     str3.append(str2, 3, 3); //只截取LOL
26     cout << "str3 = " << str3 << endl;
27 }
28 int main() {
29     test01();
30     system("pause");
31     return 0;
32 }

```

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：

```

1  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
2  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
3  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
4  int find(const char c, int pos = 0) const; //查找字符c的第一次出现位置
5  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
6  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
7  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
8  int rfind(const char c, int pos = 0) const; //查找字符c最后一次出现位置
9  string& replace(int pos, int n, const string& str); //替换从pos开始n个字符为字符串str
10 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

```

```

1  #include <iostream>
2  using namespace std;
3  #include <string>
4
5  //字符串的查找和替换
6  //查找
7  void test01() {
8      string str1 = "abcdefgde";
9      int pos = str1.find("de");
10     if (pos == -1) {
11         cout << "未找到该字符串" << endl;
12     }
13     else {
14         cout << "pos = " << pos << endl;
15     }
16     //rfind 和 find的区别
17     //rfind从右往左查找，find从左往右查找
18     pos = str1.rfind("de");
19     cout << "pos = " << pos << endl;
20 }
21 //替换

```

```

22 void test02() {
23     string str1 = "abcdefgde";
24     cout << "str1 = " << str1 << endl;
25     str1.replace(1, 3, "1111");//1号位置开始的3个字符替换成"1111"
26     cout << "str1 = " << str1 << endl;
27 }
28 int main()
29 {
30     test01();
31     test02();
32     system("pause");
33     return 0;
34 }

```

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

```

1 int compare(const string& s) const;
2 int compare(const char* s) const;

```

```

1 #include <iostream>
2 using namespace std;
3 #include<string>
4
5 //字符串的比较
6 void test01() {
7     string str1 = "abcdefg";
8     string str2 = "abcdefg";
9     int ret = str1.compare(str2);
10    if (ret == 0) {
11        cout << "str1 == str2" << endl;
12    }
13    else if (ret == 1) {
14        cout << "str1 > str2" << endl;
15    }
16    else if (ret == -1) {
17        cout << "str1 < str2" << endl;
18    }
19 }
20 int main()
21 {
22     test01();

```



```

23     system("pause");
24     return 0;
25 }

```

3.1.7 string字符串存取

```

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

```

```

1 #include <iostream>
2 using namespace std;
3 #include<string>
4
5 //string字符存取
6 void test01() {
7     string str = "Hello world";
8     //通过[]访问
9     for (int i = 0; i < str.size(); i++) {
10         cout << str[i];
11     }
12     cout << endl;
13     //通过at访问
14     for (int i = 0; i < str.size(); i++) {
15         cout << str.at(i);
16     }
17     cout << endl;
18     //修改单个字符
19     str[0] = 'x';
20     cout << "str = " << str << endl;
21     str.at(1) = 'x';
22     cout << "str = " << str << endl;
23 }
24 int main()
25 {
26     test01();
27     system("pause");
28     return 0;
29 }

```

3.1.8 string的插入和删除

函数原型:

```

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

```

```

1 #include <iostream>
2 using namespace std;
3 #include<string>
4
5 //string插入和删除
6 void test01() {

```

```

7     string str = "Hello world";
8     str.insert(1, "aee");
9     cout << "str = " << str << endl;
10    str.erase(1, 3);
11    cout << "str = " << str << endl;
12 }
13 int main()
14 {
15     test01();
16     system("pause");
17     return 0;
18 }

```

3.1.9 string子串

函数原型

```

1 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

```

```

1 #include <iostream>
2 using namespace std;
3 #include <string>
4
5 //string子串
6 void test01() {
7     string str = "Hello world";
8     string substr = str.substr(0, str.size() / 2);
9     cout << "substr = " << substr << endl;
10 }
11 //实用操作
12 void test02() {
13     string email = "muffinhead@sina.com";
14     //从邮件地址中获取用户名信息
15     int pos = email.find("@");
16     string username = email.substr(0, pos);
17     cout << "username = " << username << endl;
18 }
19 int main()
20 {
21     test01();
22     test02();
23     system("pause");
24     return 0;
25 }

```

3.2 vector容器

3.2.1 vector基本概念

功能:

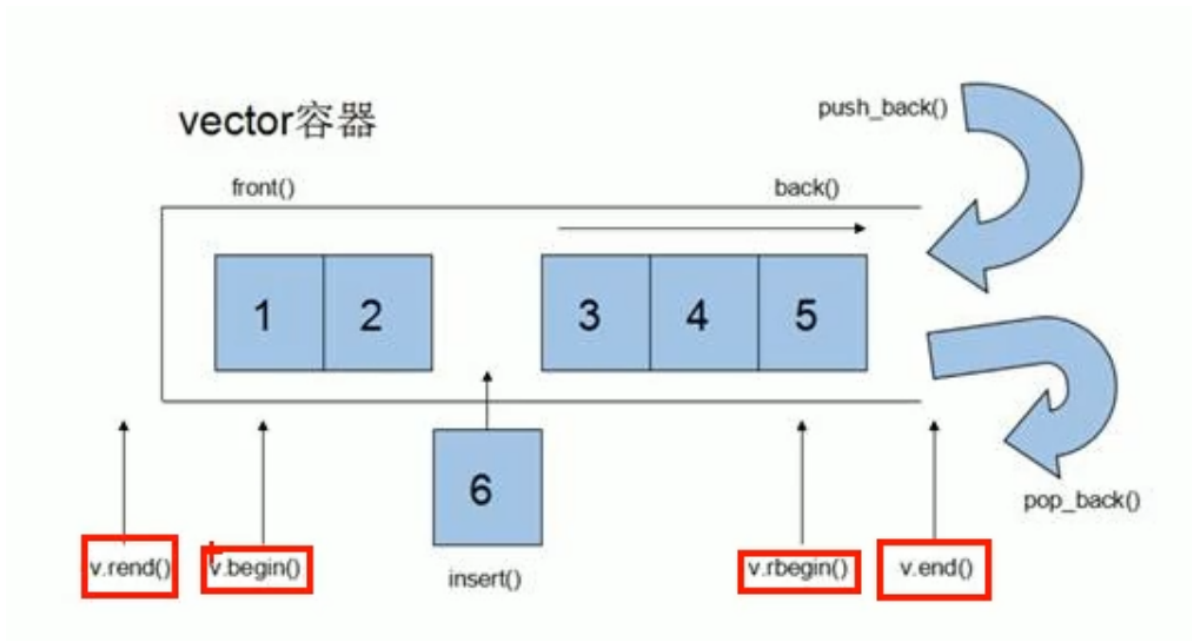
- vector数据结构和数组非常相似，也称为**单端数组**

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

```
1 vector<T>v;//采用模板实现类实现，默认构造函数
2 vector(v.begin(),v.end());//将v[begin(),end())区间中的元素拷贝给自身
3 vector(n,elem);//构造函数将n个elem拷贝给本身
4 vector(const vector& vec);//拷贝构造函数
```

```
1 #include <iostream>
2 using namespace std;
3 #include<vector>
4
5 void printVector(vector<int>&v) {
6     for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
7         cout << *it << " ";
8     }
9     cout << endl;
10 }
11 //vector构造函数
12 void test01() {
13     vector<int>v1;//默认构造
14     for (int i = 0; i < 10; i++) {
15         v1.push_back(i);
16     }
17     printVector(v1);
18
19     //通过区间方式进行构造
20     vector<int>v2(v1.begin(), v1.end());
21     printVector(v2);
22
23     //n个elem方式构造
24     vector<int>v3(10, 100);
```

```

25     printVector(v3);
26
27     //拷贝构造函数
28     vector<int>v4(v3);
29     printVector(v4);
30 }
31 int main()
32 {
33     test01();
34     system("pause");
35     return 0;
36 }

```

3.2.3 vector赋值操作

函数原型:

```

1 vector& operator=(const vector& vec); //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<vector>
4
5 void printVector(vector<int>&v) {
6     for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
7         cout << *it << " ";
8     }
9     cout << endl;
10 }
11 //vector赋值函数
12 void test01() {
13     vector<int>v1;
14     for (int i = 0; i < 10; i++) {
15         v1.push_back(i);
16     }
17     printVector(v1);
18     //赋值
19     vector<int>v2 = v1;
20     printVector(v2);
21     //assign
22     vector<int>v3;
23     v3.assign(v1.begin(), v1.end());
24     printVector(v3);
25
26     vector<int>v4;
27     v4.assign(10, 100);
28     printVector(v4);
29 }
30 int main()
31 {
32     test01();
33     system("pause");
34     return 0;
35 }

```

3.2.4 vector容量和大小

函数原型

```
1 empty();//判断容器是否为空
2 capacity();//容器的容量
3 size();//返回容器中元素的个数
4 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
5 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
```

```
1 #include <iostream>
2 using namespace std;
3 #include<vector>
4
5 void printVector(vector<int>&v) {
6     for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
7         cout << *it << " ";
8     }
9     cout << endl;
10 }
11 //vector容量和大小
12 void test01() {
13     vector<int>v1;
14     for (int i = 0; i < 10; i++) {
15         v1.push_back(i);
16     }
17     printVector(v1);
18     if (!v1.empty()) {
19         cout << "v1不为空" << endl;
20         cout << "v1的容量为: " << v1.capacity() << endl;
21         cout << "v1的大小为: " << v1.size() << endl;
22     }
23     else {
24         cout << "v1为空" << endl;
25     }
26     //重新指定大小
27     v1.resize(15);
28     printVector(v1);
29     v1.resize(16, 1);
30     printVector(v1);
31     v1.resize(5);
32     printVector(v1);
33 }
34 int main()
35 {
36     test01();
37     system("pause");
38     return 0;
39 }
```

3.2.5 vector插入和删除

函数原型:

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

```
1 #include <iostream>
2 using namespace std;
3 #include <vector>
4
5 void printVector(vector<int>&v) {
6     for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
7         cout << *it << " ";
8     }
9     cout << endl;
10 }
11 //vector插入和删除
12 void test01() {
13     vector<int>v1;
14     for (int i = 0; i < 10; i++) {
15         v1.push_back(i); //尾插法插入数据
16     }
17     printVector(v1);
18     v1.pop_back(); //尾删法
19
20     v1.insert(v1.begin(), 100); //提供迭代器插入
21     printVector(v1);
22     v1.insert(v1.begin(), 2, 100);
23     printVector(v1);
24
25     //删除
26     v1.erase(v1.begin());
27     printVector(v1);
28
29     //v1.erase(v1.begin(), v1.end()); //类似于清空
30     //printVector(v1);
31
32     v1.clear(); //清空
33 }
34 int main()
35 {
36     test01();
37     system("pause");
38     return 0;
39 }
```

3.2.6 数据获取

函数原型:

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

```
1 #include <iostream>
2 using namespace std;
3 #include<vector>
4
5 //vector数据存取
6 void test01() {
7     vector<int>v1;
8     for (int i = 0; i < 10; i++) {
9         v1.push_back(i);
10    }
11    for (int i = 0; i < v1.size(); i++) {
12        //cout << v1[i] << " "; //利用[]访问
13        cout << v1.at(i) << " "; //利用成员函数at
14    }
15    cout << endl;
16    //获取第一个元素
17    cout << "v1的第一个元素: " << v1.front() << endl;
18    //获取最后一个元素
19    cout << "v1的最后一个元素: " << v1.back() << endl;
20 }
21 int main()
22 {
23     test01();
24     system("pause");
25     return 0;
26 }
```

3.2.7 vector互换容器

函数原型:

```
1 swap(vec); //将vec与本身的元素互换
```

```
1 #include <iostream>
2 using namespace std;
3 #include<vector>
4 void printVector(vector<int>& v) {
5     for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
6         cout << *it << " ";
7     }
8     cout << endl;
9 }
10 //vector数据存取
11 void test01() {
12     vector<int>v1;
13     for (int i = 0; i < 10; i++) {
```

```

14     v1.push_back(i);
15 }
16 cout << "互换前: " << endl;
17 printVector(v1);
18 vector<int>v2;
19 for (int i = 10; i > 0; i--) {
20     v2.push_back(i);
21 }
22 printVector(v2);
23 cout << "交换后: " << endl;
24 v1.swap(v2);
25 printVector(v1);
26 printVector(v2);
27 }
28 //实际用途
29 void test02() {
30     //巧用swap 可以收缩内存空间
31     vector<int>v;
32     for (int i = 0; i < 10000; i++) {
33         v.push_back(i);
34     }
35     cout << "v.capacity() = " << v.capacity() << endl;
36     cout << "v.size() = " << v.size() << endl;
37     v.resize(3); //重新指定大小
38     cout << "v.capacity() = " << v.capacity() << endl;
39     cout << "v.size() = " << v.size() << endl;
40     //巧用swap收缩内存
41     vector<int>(v).swap(v);
42     //vector<int>(v); //匿名对象以v来初始化
43     cout << "v.capacity() = " << v.capacity() << endl;
44     cout << "v.size() = " << v.size() << endl;
45 }
46 int main()
47 {
48     test01();
49     test02();
50     system("pause");
51     return 0;
52 }

```

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:

```
1 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
```

```

1 #include <iostream>
2 using namespace std;
3 #include<vector>
4
5 //vector预留空间
6 void test01() {
7     vector<int>v;

```



```

8      //利用reserve预留空间
9      v.reserve(10000);
10     int num = 0; //统计开辟次数
11     int* p = NULL;
12     for (int i = 0; i < 10000; i++) {
13         v.push_back(i);
14         if (p != &v[0]) {
15             p = &v[0];
16             num++;
17         }
18     }
19     cout << "num = " << num << endl;
20 }
21
22 int main()
23 {
24     test01();
25     system("pause");
26     return 0;
27 }

```

- 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

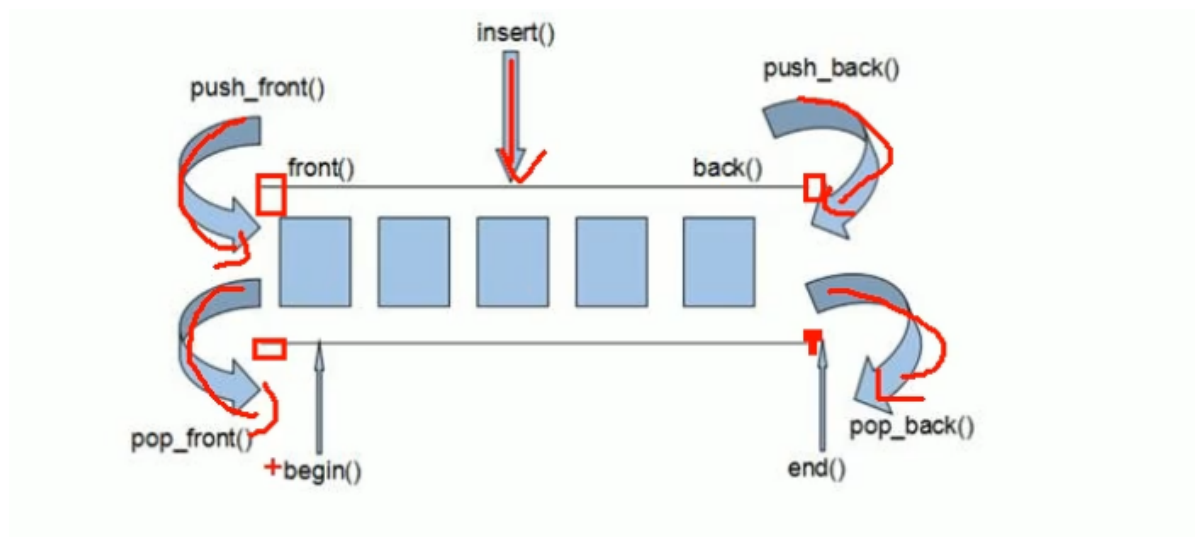
3.3.1 deque容器基本概念

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：

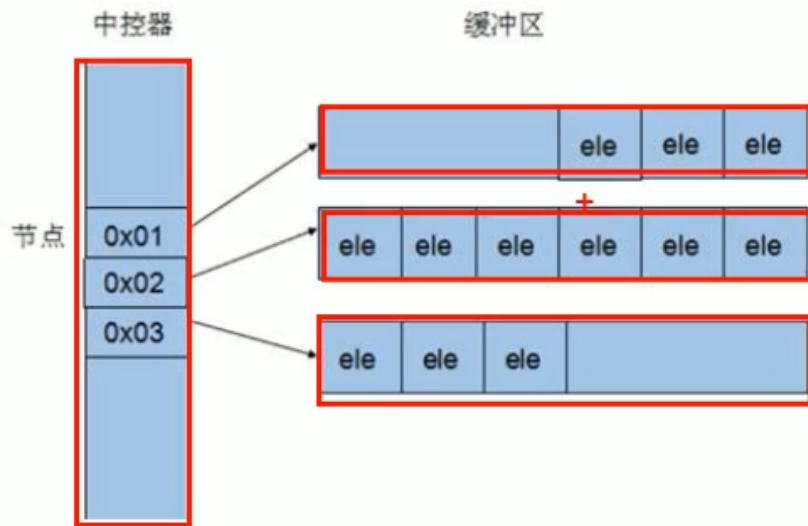
- vector对于头部的插入删除效率低，数据量越大，效率越低
- deque相对而言，对头部的插入删除效率会比vector快
- vector访问元素的速度会比deque快，这和两者的内部实现有关



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:

```

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

```

```

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     {
9         cout << *it << " ";
10    }
11    cout << endl;
12 }
13 //deque构造函数
14 void test01() {
15     deque<int> deq1; //默认构造函数
16     for (int i = 0; i < 10; i++) {
17         deq1.push_back(i);
18     }
19     printDeque(deq1);
20
21     deque<int> deq2(deq1.begin(), deq1.end());
22     printDeque(deq2);
23
24     deque<int> deq3(10, 100);
25     printDeque(deq3);
26
27     deque<int> deq4(deq3);
28     printDeque(deq4);
29 }

```

```

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     {
9         cout << *it << " ";
10    }
11    cout << endl;
12 }
13 //deque赋值操作
14 void test01() {
15     deque<int>deq1;
16     for (int i = 0; i < 10; i++) {
17         deq1.push_back(i);
18     }
19     printDeque(deq1);
20
21     deque<int>deq2 = deq1;
22     printDeque(deq2);
23
24     deque<int>deq3;
25     deq3.assign(deq1.begin(), deq1.end());
26     printDeque(deq3);
27
28     deque<int>deq4;
29     deq4.assign(10, 100);
30     printDeque(deq4);
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:

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

```
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     {
9         cout << *it << " ";
10    }
11    cout << endl;
12 }
13 //deque大小操作
14 void test01() {
15     deque<int>deq1;
16     for (int i = 0; i < 10; i++) {
17         deq1.push_back(i);
18     }
19     printDeque(deq1);
20     if (!deq1.empty()) {
21         cout << "deq1不为空" << endl;
22         cout << "deq1.size() = " << deq1.size() << endl;
23         //deque没有容量的概念
24     }
25     else {
26         cout << "deq1为空" << endl;
27     }
28     //deq1.resize(15);
29     deq1.resize(15,1);
30     printDeque(deq1);
31
32     deq1.resize(5);
33     printDeque(deq1);
34 }
35
36 int main()
37 {
38     test01();
39     system("pause");
40     return 0;
41 }
```

3.3.5 deque插入和删除

函数原型:

```
1 //两端插入操作:
2 push_back(elem); //在容器尾部添加一个数据
3 push_front(elem); //在容器头部插入一个数据
4 pop_back(); //删除容器最后一个数据
5 pop_front(); //删除容器第一个数据
6 //指定位置操作
7 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
8 insert(pos, n, elem); //在pos位置插入n个elem数据, 无返回值
9 insert(pos, beg, end); //在pos位置插入[beg, end)区间的数据, 无返回值
10 clear(); //清空容器的所有数据
11 erase(beg, end); //删除[beg, end)区间的数据, 返回下一个数据的位置
12 erase(pos); //删除pos位置的数据, 返回下一个数据的位置
```

```
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     {
9         cout << *it << " ";
10    }
11    cout << endl;
12 }
13 //deque插入和删除
14 void test01() {
15     deque<int> deq1;
16
17     //尾插
18     deq1.push_back(10);
19     deq1.push_back(20);
20
21     //头插
22     deq1.push_front(100);
23     deq1.push_front(200);
24
25     printDeque(deq1);
26
27     //头删
28     deq1.pop_front();
29     //尾删
30     deq1.pop_back();
31     printDeque(deq1);
32 }
33 void test02() {
34     deque<int> deq1;
35     deq1.push_back(10);
36     deq1.push_back(20);
37     deq1.push_front(100);
38     deq1.push_front(200);
39     printDeque(deq1);
40 }
```

```

40 //insert插入
41 deq1.insert(deq1.begin(), 1000);
42 printDeque(deq1);
43 deq1.insert(deq1.begin(), 2, 10000);
44 printDeque(deq1);
45
46 //按照区间插入
47 deque<int>deq2;
48 for (int i = 0; i < 10; i++) {
49     deq2.push_back(i);
50 }
51 deq1.insert(deq1.begin(), deq2.begin(), deq2.end());
52 printDeque(deq1);
53 }
54 void test03() {
55     deque<int>deq3;
56     for (int i = 0; i < 10; i++) {
57         deq3.push_back(i);
58     }
59     //删除
60     deque<int>::iterator it = deq3.begin();
61     it++;
62     deq3.erase(it);
63     printDeque(deq3);
64
65     //按照区间方式删除
66     //deq3.erase(deq3.begin(), deq3.end()); //类似于清空
67     deq3.clear(); //清空
68     printDeque(deq3);
69 }
70 int main()
71 {
72     test01();
73     test02();
74     test03();
75     system("pause");
76     return 0;
77 }

```

3.3.6 deque数据存取

函数原型:

```

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

```

```

1 #include <iostream>
2 using namespace std;
3 #include<deque>
4
5 //deque数据存取
6 void test01() {
7     deque<int>deq1;
8     for (int i = 0; i < 3; i++) {

```

```

9         deq1.push_back(i);
10    }
11    for (int i = 0; i < 3; i++) {
12        deq1.push_front(i+3);
13    }
14    //通过[]和at方式访问元素
15    for (int i = 0; i < deq1.size() ; i++) {
16        //cout << deq1[i] << " ";
17        cout << deq1.at(i) << " ";
18    }
19    cout << endl;
20
21    cout << "第一个元素: " << deq1.front() << endl;
22    cout << "最后一个元素: " << deq1.back() << endl;
23 }
24 int main()
25 {
26     test01();
27     system("pause");
28     return 0;
29 }

```

3.3.7 deque排序

```

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

```

```

1 #include <iostream>
2 using namespace std;
3 #include<deque>
4 #include <algorithm>
5 void printDeque(const deque<int>& deq) {
6     for (deque<int>::const_iterator it = deq.begin(); it != deq.end(); it++)
7     {
8         cout << *it << " ";
9     }
10    cout << endl;
11 }
12 //deque排序
13 void test01() {
14     deque<int>deq1;
15     deq1.push_back(10);
16     deq1.push_back(100);
17     deq1.push_back(160);
18     deq1.push_back(20);
19     deq1.push_back(90);
20     deq1.push_back(70);
21     deq1.push_back(30);
22     cout << "排序前: " << endl;
23     printDeque(deq1);
24     //排序: 默认升序
25     //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
26     sort(deq1.begin(), deq1.end());
27     cout << "排序后: " << endl;
28     printDeque(deq1);
29 }

```

```

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

```

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

```

1  #include<iostream>
2  using namespace std;
3  #include<string>
4  #include<vector>
5  #include<deque>
6  #include<ctime>
7  #include <algorithm>
8  class Person {
9  public:
10     Person(string name) {
11         this->m_Name = name;
12     }
13     string m_Name;
14     int m_Score;
15 };
16 void CreatPerson(vector<Person>& v) {
17     string nameSeed = "ABCDE";
18     for (int i = 0; i < 5; i++) {
19         Person p("选手");
20         p.m_Name += nameSeed.at(i);
21         p.m_Score = 0; //初始化为0
22         v.push_back(p);
23     }
24 }
25 void setScore(vector<Person>& v) {
26     for (vector<Person>::iterator it = v.begin(); it != v.end(); it++) {
27         deque<int> d;
28         for (int i = 0; i < 10; i++) {
29             int score = rand() % 41 + 60;
30             d.push_back(score);
31         }
32         //排序
33         sort(d.begin(), d.end());
34         //去除最高分和最低分
35         d.pop_back();
36         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 }

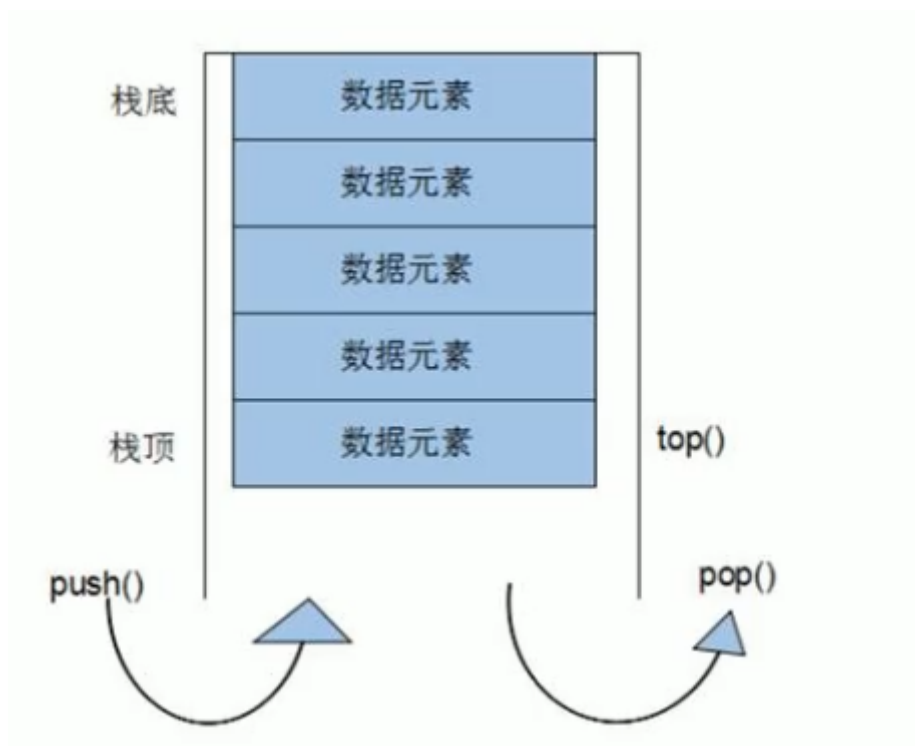
```

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:

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

赋值操作:

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

数据存取:

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

大小操作:

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

```
1 #include <iostream>
2 using namespace std;
3 #include <stack>
4
5 void test01() {
6     stack<int>stk;
```

```

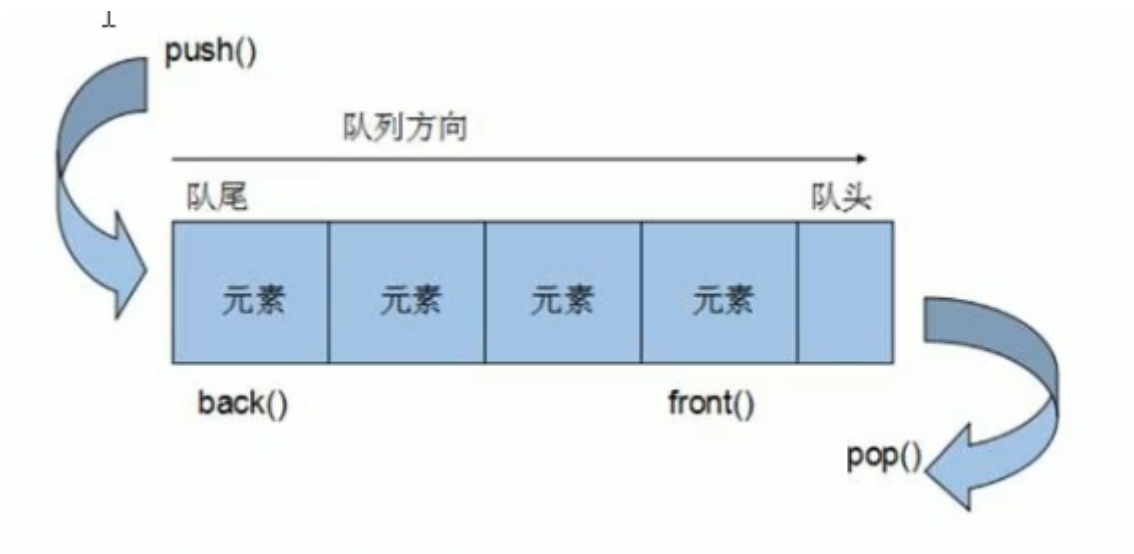
7      //入栈
8      stk.push(10);
9      stk.push(20);
10     stk.push(30);
11     stk.push(40);
12     cout << "stk.size() = " << stk.size() << endl;
13
14     while (!stk.empty()) {
15         cout << stk.top() << " ";
16         stk.pop();//出栈
17     }
18     cout << endl << "stk.size() = " << stk.size() << endl;
19 }
20 int main()
21 {
22     test01();
23     system("pause");
24     return 0;
25 }

```

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：

```

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

```

赋值操作：

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

数据存取:

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

大小操作:

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

```
1 #include <iostream>
2 using namespace std;
3 #include <queue>
4 #include <string>
5 class Person {
6 public:
7     Person(string name, int age) {
8         this->m_Name = name;
9         this->m_Age = age;
10    }
11    string m_Name;
12    int m_Age;
13 };
14 //队列Queue
15 void test01() {
16     queue<Person> que;
17     Person p1("Tom", 18);
18     Person p2("Jerry", 18);
19     Person p3("Mary", 18);
20     Person p4("John", 18);
21
22     //入队
23     que.push(p1);
24     que.push(p2);
25     que.push(p3);
26     que.push(p4);
27
28     cout << "que.size() = " << que.size() << endl;
29     while (!que.empty()) {
30         //输出对头
31         cout << "队头元素--name:" << que.front().m_Name << " age:" <<
que.front().m_Age << endl;
32         cout << "队尾元素--name:" << que.back().m_Name << " age:" <<
que.back().m_Age << endl;
33         //出队
34         que.pop();
35     }
36     cout << "que.size() = " << que.size() << endl;
37 }
38 int main()
39 {
```

```

40     test01();
41     system("pause");
42     return 0;
43 }

```

3.7 list容器

3.7.1 list基本概念

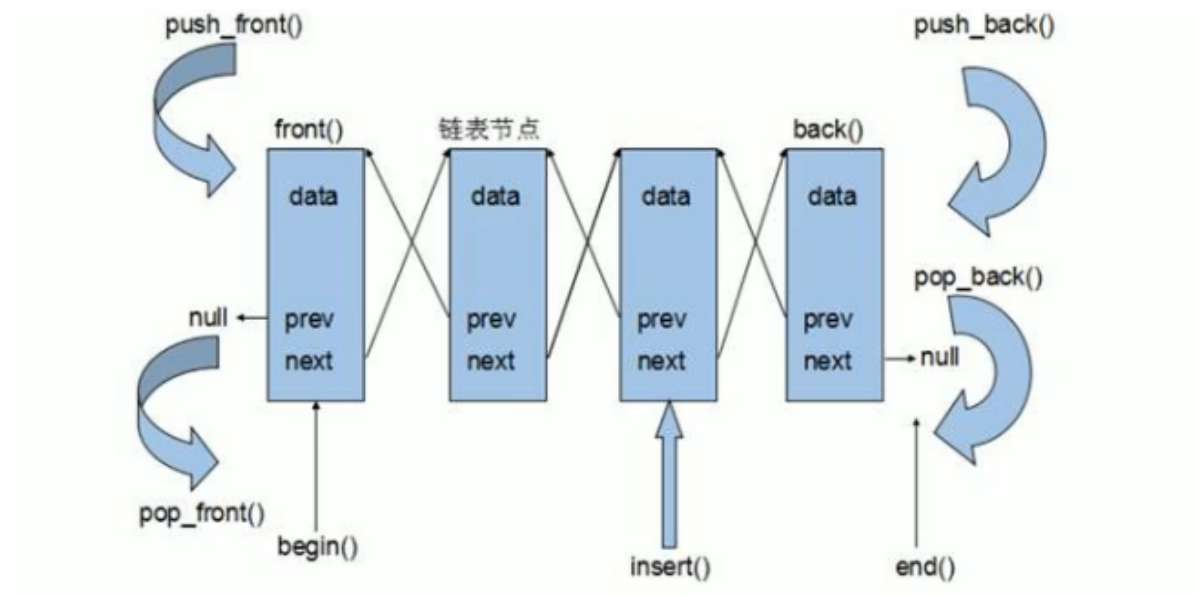
功能：将数据进行链式存储

链表 (list)：一种在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：

```

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

```

```

1  #include <iostream>
2  using namespace std;
3  #include <list>
4
5  void printList(const list<int>& L) {
6      for (list<int>::const_iterator it = L.begin(); it != L.end(); it++) {
7          cout << *it << " ";
8      }
9      cout << endl;
10 }
11 //list构造函数
12 void test01() {
13     list<int>L1;
14     //添加数据
15     L1.push_back(10);
16     L1.push_back(20);
17     L1.push_back(30);
18     L1.push_back(40);
19     printList(L1);
20
21     list<int>L2(L1.begin(), L1.end());
22     printList(L2);
23
24     list<int>L3(L2);
25     printList(L3);
26
27     list<int>L4(10, 100);
28     printList(L4);
29 }
30 int main()
31 {
32     test01();
33     system("pause");
34     return 0;
35 }

```

3.7.3 list赋值和交换

函数原型:

```

1  assign(beg,end); //将[beg,end)区间中的数据拷贝赋值给本身
2  assign(n,elem); //将n个elem拷贝赋值给本身
3  list& operator=(const list& lst); //重载等号运算符
4  swap(lst); //将lst与本身元素互换

```

```

1  #include <iostream>
2  using namespace std;
3  #include <list>
4
5  void printList(const list<int>& L) {
6      for (list<int>::const_iterator it = L.begin(); it != L.end(); it++) {
7          cout << *it << " ";
8      }
9      cout << endl;
10 }
11 //list赋值和交换

```

```

12 void test01() {
13     list<int>L1;
14     L1.push_back(10);
15     L1.push_back(20);
16     L1.push_back(30);
17     L1.push_back(40);
18     printList(L1);
19
20     list<int>L2 = L1;
21     printList(L2);
22
23     list<int>L3;
24     L3.assign(L2.begin(), L2.end());
25     printList(L3);
26
27     list<int>L4;
28     L4.assign(10, 100);
29     printList(L4);
30 };
31 //交换
32 void test02() {
33     list<int>L1;
34     L1.push_back(10);
35     L1.push_back(20);
36     L1.push_back(30);
37     L1.push_back(40);
38
39     list<int>L2;
40     L2.assign(10, 100);
41     cout << "交换前" << endl;
42     printList(L1);
43     printList(L2);
44     L1.swap(L2);
45     cout << "交换后" << endl;
46     printList(L1);
47     printList(L2);
48 }
49 int main()
50 {
51     test01();
52     test02();
53     system("pause");
54     return 0;
55 }

```

3.7.4 list大小操作

函数原型:

```

1 empty(); //判断容器是否为空
2 capacity(); //容器的容量
3 size(); //返回容器中元素的个数
4 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
5 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

```

```

1  #include <iostream>
2  using namespace std;
3  #include <list>
4
5  void printList(const list<int>& L) {
6      for (list<int>::const_iterator it = L.begin(); it != L.end(); it++) {
7          cout << *it << " ";
8      }
9      cout << endl;
10 }
11 //list大小操作
12 void test01() {
13     list<int>L1;
14     L1.push_back(10);
15     L1.push_back(20);
16     L1.push_back(30);
17     L1.push_back(40);
18     printList(L1);
19     if (!L1.empty()) {
20         cout << "L1不为空" << endl;
21         cout << "L1的元素个数: " << L1.size() << endl;
22     }
23     else {
24         cout << "L1为空" << endl;
25     }
26     //重新指定大小
27     //L1.resize(10);
28     L1.resize(10,100);
29     printList(L1);
30     L1.resize(3);
31     printList(L1);
32 };
33
34 int main()
35 {
36     test01();
37     system("pause");
38     return 0;
39 }

```

3.7.5 list插入和删除

函数原型:

```

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

```



```
1  #include <iostream>
2  using namespace std;
3  #include <list>
4
5  void printList(const list<int>& L) {
6      for (list<int>::const_iterator it = L.begin(); it != L.end(); it++) {
7          cout << *it << " ";
8      }
9      cout << endl;
10 }
11 //list插入和删除
12 void test01() {
13     list<int>L;
14     //尾插
15     L.push_back(10);
16     L.push_back(20);
17     L.push_back(30);
18     //头插
19     L.push_front(100);
20     L.push_front(200);
21     L.push_front(300);
22
23     printList(L);
24
25     //尾删
26     L.pop_back();
27     //头删
28     L.pop_front();
29     printList(L);
30     //insert插入
31     list<int>::iterator it = L.begin();
32     L.insert(++it, 1000);
33     printList(L);
34
35     //删除
36     it = L.begin();
37     L.erase(++it);
38     printList(L);
39
40     //移除
41     L.push_back(10000);
42     L.push_back(10000);
43     L.push_back(10000);
44     L.push_back(10000);
45     printList(L);
46     L.remove(10000);
47     printList(L);
48
49     //清空
50     L.clear();
51     printList(L);
52 };
53
54 int main()
55 {
56     test01();
57     system("pause");
58     return 0;
```

3.7.6 list数据存取

函数原型:

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

```
1 #include <iostream>
2 using namespace std;
3 #include <list>
4
5 //list数据存取
6 void test01() {
7     list<int>L1;
8     L1.push_back(10);
9     L1.push_back(20);
10    L1.push_back(30);
11    L1.push_back(40);
12
13    cout << "L1的第一个元素:" << L1.front() << endl;
14    cout << "L1的最后一个元素: " << L1.back() << endl;
15
16    //list迭代器不支持随机访问
17    list<int>::iterator it = L1.begin();
18    it++;
19    it--;    //支持双向
20    //it = it + 1;可能是跳跃型访问
21 };
22
23 int main()
24 {
25     test01();
26     system("pause");
27     return 0;
28 }
```

3.7.7 list反转和排序

函数原型:

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

```
1 #include <iostream>
2 using namespace std;
3 #include <list>
4 #include <algorithm>
5 void printList(const list<int>& L) {
6     for (list<int>::const_iterator it = L.begin(); it != L.end(); it++) {
7         cout << *it << " ";
8     }
9     cout << endl;
10 }
```

```

11 bool myCompare(int v1,int v2) {
12     //降序
13     return v1 > v2;
14 }
15 //list反转和排序
16 void test01() {
17     list<int>L1;
18     L1.push_back(10);
19     L1.push_back(40);
20     L1.push_back(60);
21     L1.push_back(50);
22     L1.push_back(80);
23     L1.push_back(400);
24     L1.push_back(90);
25     L1.push_back(30);
26     cout << "排序前: " << endl;
27     printList(L1);
28     //反转
29     cout << "反转后: " << endl;
30     L1.reverse();
31     printList(L1);
32
33     //排序
34     cout << "排序后（默认升序）: " << endl;
35     //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
36     //不支持随机访问的迭代器的容器，内部会提供对应的算法
37     //sort(L1.begin(), L1.end());
38     L1.sort();//默认升序
39     printList(L1);
40
41     cout << "排序后（降序）:" << endl;
42     L1.sort(myCompare);
43     printList(L1);
44 };
45
46 int main()
47 {
48     test01();
49     system("pause");
50     return 0;
51 }

```

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

```

1  #include<iostream>
2  #include<string>
3  #include<list>
4  using namespace std;
5
6  class Person {
7  public:
8      Person(string name, int age, double height) {
9          this->m_Name = name;
10         this->m_Age = age;

```

```

11         this->m_Height = height;
12     }
13     string m_Name;
14     int m_Age;
15     double m_Height;
16 };
17 bool comparePerson(Person& p1, Person& p2) {
18     //按照年龄升序
19     if (p1.m_Age == p2.m_Age) {
20         return p1.m_Height > p2.m_Height; //按照身高降序
21     }
22     return p1.m_Age < p2.m_Age;
23 }
24 void test01() {
25     list<Person> L;
26     Person p1("刘备", 35, 175);
27     Person p2("曹操", 45, 180);
28     Person p3("孙权", 40, 170);
29     Person p4("赵云", 25, 190);
30     Person p5("张飞", 35, 160);
31     Person p6("关羽", 35, 200);
32     L.push_back(p1);
33     L.push_back(p2);
34     L.push_back(p3);
35     L.push_back(p4);
36     L.push_back(p5);
37     L.push_back(p6);
38
39     for (list<Person>::iterator it = L.begin(); it != L.end(); it++) {
40         cout << "姓名: " << it->m_Name << " 年龄: " << it->m_Age << " 身高: "
41         << it->m_Height << endl;
42     }
43     //排序
44     cout << "-----" << endl;
45     cout << "排序后: " << endl;
46     L.sort(comparePerson);
47     for (list<Person>::iterator it = L.begin(); it != L.end(); it++) {
48         cout << "姓名: " << it->m_Name << " 年龄: " << it->m_Age << " 身高: "
49         << it->m_Height << endl;
50     }
51 }
52 int main()
53 {
54     test01();
55     system("pause");
56     return 0;
57 }

```

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

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

```
1 #include<iostream>
2 using namespace std;
3 #include<set>
4 //set容器构造和赋值
5 void printSet(set<int>& s) {
6     for (set<int>::iterator it = s.begin(); it != s.end(); it++) {
7         cout << *it << " ";
8     }
9     cout << endl;
10 }
11 void test01() {
12     set<int> s1;
13     s1.insert(10);
14     s1.insert(40);
15     s1.insert(30);
16     s1.insert(20);
17     s1.insert(30);
18     //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
19     printSet(s1);
20
21     set<int> s2(s1);
22     printSet(s2);
23
24     set<int> s3;
25     s3 = s2;
26     printSet(s3);
27 }
28 int main()
29 {
30     test01();
31     system("pause");
32     return 0;
33 }
```

3.8.3 set大小和交换

函数原型:

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

```

1  #include<iostream>
2  using namespace std;
3  #include<set>
4  //set大小和交换
5  void printSet(set<int>& s) {
6      for (set<int>::iterator it = s.begin(); it != s.end(); it++) {
7          cout << *it << " ";
8      }
9      cout << endl;
10 }
11 void test01() {
12     set<int>s1;
13     s1.insert(10);
14     s1.insert(40);
15     s1.insert(30);
16     s1.insert(20);
17     //判断是否为空
18     if (!s1.empty()) {
19         cout << "集合容器s1不为空" << endl;
20         cout << "s1.size() = " << s1.size() << endl;
21     }
22     else {
23         cout << "集合容器s1为空" << endl;
24     }
25     set<int>s2;
26     s2.insert(10);
27     s2.insert(40);
28     s2.insert(50);
29     s2.insert(70);
30     s2.insert(80);
31     s2.insert(20);
32
33     cout << "交换前" << endl;
34     printSet(s1);
35     printSet(s2);
36     s1.swap(s2);
37     cout << "交换后" << endl;
38     printSet(s1);
39     printSet(s2);
40 }
41 int main()
42 {
43     test01();
44     system("pause");
45     return 0;
46 }

```

3.8.4 set插入和删除

函数原型:

```

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

```

```

1  #include<iostream>
2  using namespace std;
3  #include<set>
4  //set插入和删除
5  void printSet(set<int>& s) {
6      for (set<int>::iterator it = s.begin(); it != s.end(); it++) {
7          cout << *it << " ";
8      }
9      cout << endl;
10 }
11 void test01() {
12     set<int>s1;
13     //插入
14     s1.insert(10);
15     s1.insert(40);
16     s1.insert(30);
17     s1.insert(20);
18
19     printSet(s1);
20
21     //删除
22     s1.erase(s1.begin());
23     printSet(s1);
24     //删除重载的版本
25     s1.erase(30);
26     printSet(s1);
27
28     //清空
29     //s1.erase(s1.begin(), s1.end());
30     s1.clear();
31     printSet(s1);
32 }
33 int main()
34 {
35     test01();
36     system("pause");
37     return 0;
38 }

```

3.8.5 set查找和统计

函数原型:

```

1  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()
2  count(key); //统计key元素的个数

```

```

1  #include<iostream>
2  using namespace std;
3  #include<set>
4  //set查找和统计
5
6  void test01() {
7      set<int>s1;
8      //插入
9      s1.insert(10);
10     s1.insert(40);

```

```

11     s1.insert(30);
12     s1.insert(20);
13     //查找
14     set<int>::iterator pos = s1.find(300);
15     if (pos != s1.end()) {
16         cout << "找到该元素: " << *pos << endl;
17     }
18     else {
19         cout << "未找到元素" << endl;
20     }
21     //统计
22     int num = s1.count(30);
23     cout << "num = " << num << endl;
24 }
25 int main()
26 {
27     test01();
28     system("pause");
29     return 0;
30 }

```

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

```

1  #include<iostream>
2  using namespace std;
3  #include<set>
4  //set和multiset区别
5  void test01() {
6      set<int>s;
7      pair<set<int>::iterator, bool> ret = s.insert(10);
8      if (ret.second) {
9          cout << "第一次插入成功" << endl;
10     }
11     else {
12         cout << "第一次插入失败" << endl;
13     }
14     ret = s.insert(10);
15     if (ret.second) {
16         cout << "第二次插入成功" << endl;
17     }
18     else {
19         cout << "第二次插入失败" << endl;
20     }
21     multiset<int>ms;//允许插入重复数据
22     ms.insert(10);
23     ms.insert(10);
24     ms.insert(10);
25     ms.insert(10);
26     for (multiset<int>::iterator it = ms.begin(); it != ms.end(); it++) {
27         cout << *it << " ";
28     }

```



```

29     cout << endl;
30 }
31 int main()
32 {
33     test01();
34     system("pause");
35     return 0;
36 }

```

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：

```

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

```

```

1 #include<iostream>
2 using namespace std;
3 #include<string>
4 //对组创建
5 void test01() {
6     pair<string, int> p(string("Tom"), 20);
7     cout << "name:" << p.first << " age:" << p.second << endl;
8
9     pair<string, int>p2 = make_pair("Jerry", 20);
10    cout << "name:" << p2.first << " age:" << p2.second << endl;
11 }
12 int main()
13 {
14     test01();
15     system("pause");
16     return 0;
17 }

```

3.8.8 set容器排序

- 利用仿函数，改变默认排序规则

```

1 //set存放内置数据类型
2 #include<iostream>
3 using namespace std;
4 #include<string>
5 #include<set>
6 //set容器排序
7 class MyCompare {
8 public:
9     bool operator()(int v1,int v2)const {
10         return v1 > v2;
11     }
12 };
13 void test01() {
14     set<int>s1;

```

```

15     s1.insert(10);
16     s1.insert(40);
17     s1.insert(30);
18     s1.insert(20);
19     s1.insert(50);
20     for (set<int>::const_iterator it = s1.begin(); it != s1.end(); it++) {
21         cout << *it << " ";
22     }
23     cout << endl;
24     //指定排序规则为从大到小
25     //set插入数据之前改变排序规则
26     set<int, MyCompare> s2;
27     s2.insert(10);
28     s2.insert(40);
29     s2.insert(30);
30     s2.insert(20);
31     s2.insert(50);
32     for (set<int, MyCompare>::const_iterator it = s2.begin(); it != s2.end();
it++) {
33         cout << *it << " ";
34     }
35     cout << endl;
36 }
37 int main()
38 {
39     test01();
40     system("pause");
41     return 0;
42 }

```

```

1  //set存放自定义数据类型
2  #include<iostream>
3  using namespace std;
4  #include<string>
5  #include<set>
6  //set容器排序
7  class Person {
8  public:
9      Person(string name, int age) {
10         this->m_Name = name;
11         this->m_Age = age;
12     }
13     string m_Name;
14     int m_Age;
15 };
16 class MyComparePerson {
17 public:
18     bool operator()(const Person&p1, const Person&p2) const {
19         return p1.m_Age > p2.m_Age;
20     }
21 };
22 void test01() {
23     //自定义数据类型都会指定排序规则
24     set<Person, MyComparePerson> s;
25     Person p1("刘备", 35);
26     Person p2("曹操", 45);
27     Person p3("孙权", 40);

```

```

28     Person p4("赵云", 25);
29     s.insert(p1);
30     s.insert(p2);
31     s.insert(p3);
32     s.insert(p4);
33
34     for (set<Person, MyComparePerson>::iterator it = s.begin(); it !=
s.end(); it++) {
35         cout << "name:" << it->m_Name << " age:" << it->m_Age << endl;
36     }
37 }
38 int main()
39 {
40     test01();
41     system("pause");
42     return 0;
43 }

```

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:

```

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

```

```

1 #include<iostream>
2 using namespace std;
3 #include<map>
4 //map构造和赋值
5 void printMap(const map<int, int>& m) {
6     for (map<int, int>::const_iterator it = m.begin(); it != m.end(); it++)
7     {

```

```

7         cout << "key = " << it->first << " value = " << it->second << endl;
8     }
9     cout << endl;
10 }
11 void test01() {
12     map<int, int>mp;
13     //对组插入
14     mp.insert(pair<int, int>(1, 10));
15     mp.insert(pair<int, int>(3, 30));
16     mp.insert(pair<int, int>(4, 40));
17     mp.insert(pair<int, int>(2, 20));
18
19     printMap(mp);
20     //拷贝构造
21     map<int, int>mp2(mp);
22     printMap(mp2);
23
24     map<int, int>mp3;
25     mp3 = mp;
26     printMap(mp3);
27 }
28 int main()
29 {
30     test01();
31     system("pause");
32     return 0;
33 }

```

3.9.3 map大小和交换

函数原型:

```

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

```

```

1 #include<iostream>
2 using namespace std;
3 #include<map>
4 //map大小和交换
5 void printMap(const map<int, int>& m) {
6     for (map<int, int>::const_iterator it = m.begin(); it != m.end(); it++)
7     {
8         cout << "key = " << it->first << " value = " << it->second << endl;
9     }
10    cout << endl;
11 }
12 void test01() {
13     map<int, int>mp;
14     mp.insert(pair<int, int>(1, 10));
15     mp.insert(pair<int, int>(3, 30));
16     mp.insert(pair<int, int>(4, 40));
17     mp.insert(pair<int, int>(2, 20));
18
19     if (!mp.empty()) {
20         cout << "mp不为空" << endl;
21     }
22 }

```

```

20         cout << "mp.size() = " << mp.size() << endl;
21     }
22     else {
23         cout << "mp为空" << endl;
24     }
25     map<int, int>mp2;
26     mp2.insert(pair<int, int>(5, 10));
27     mp2.insert(pair<int, int>(6, 30));
28     mp2.insert(pair<int, int>(7, 40));
29     mp2.insert(pair<int, int>(8, 20));
30     cout << "交换前: " << endl;
31     printMap(mp);
32     printMap(mp2);
33
34     mp.swap(mp2);
35
36     cout << "交换后: " << endl;
37     printMap(mp);
38     printMap(mp2);
39 }
40 int main()
41 {
42     test01();
43     system("pause");
44     return 0;
45 }

```

3.9.4 map插入和删除

函数原型:

```

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

```

```

1 #include<iostream>
2 using namespace std;
3 #include<map>
4 //map插入和删除
5 void printMap(const map<int, int>& m) {
6     for (map<int, int>::const_iterator it = m.begin(); it != m.end(); it++)
7     {
8         cout << "key = " << it->first << " value = " << it->second << endl;
9     }
10    cout << endl;
11 }
12 void test01() {
13     map<int, int>mp;
14     //插入
15     //第一种
16     mp.insert(pair<int, int>(1, 10));
17     //第二种
18     mp.insert(make_pair(3, 30));
19     //第三种 (可忽略)

```

```

19     mp.insert(map<int,int>::value_type(4,40));
20     //第四种（不建议）----用途：利用key访问value
21     mp[2] = 20;
22     //cout << mp[5] << endl;
23     printMap(mp);
24
25     //删除
26     mp.erase(mp.begin());
27     printMap(mp);
28
29     mp.erase(3); //按照key删除
30     printMap(mp);
31
32     //清空
33     //mp.erase(mp.begin(), mp.end());
34     mp.clear();
35     printMap(mp);
36
37 }
38 int main()
39 {
40     test01();
41     system("pause");
42     return 0;
43 }

```

3.9.5 map查找和统计

函数原型：

```

1 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();
2 count(key); //统计key的元素个数

```

```

1 #include<iostream>
2 using namespace std;
3 #include<map>
4 //map查找和统计
5 void test01() {
6     map<int, int> mp;
7     //map不允许插入重复的键值
8     mp.insert(pair<int, int>(1, 10));
9     mp.insert(pair<int, int>(3, 30));
10    mp.insert(pair<int, int>(4, 40));
11    mp.insert(pair<int, int>(2, 20));
12
13    //查找
14    map<int, int>::iterator pos = mp.find(3);
15    if (pos != mp.end()) {
16        cout << "查找到了元素 key = " << pos->first << " value = " << pos->second << endl;
17    }
18    else {
19        cout << "未找到元素" << endl;
20    }
21    //统计
22    int num = mp.count(3); //map统计结果 1 or 0

```

```

23     cout << "num = " << num << endl;
24 }
25 int main()
26 {
27     test01();
28     system("pause");
29     return 0;
30 }

```

3.9.6 map排序

- 利用仿函数改变默认排序规则

```

1  #include<iostream>
2  using namespace std;
3  #include<map>
4  //map排序
5  class MyCompareMap {
6  public:
7      bool operator()(const int& v1, const int& v2)const {
8          return v1 > v2;
9      }
10 };
11 void test01() {
12     map<int, int, MyCompareMap> mp;
13     //map不允许插入重复的键值
14     mp.insert(pair<int, int>(1, 10));
15     mp.insert(pair<int, int>(3, 30));
16     mp.insert(pair<int, int>(4, 40));
17     mp.insert(make_pair(2, 20));
18     mp.insert(make_pair(5, 50));
19     for (map<int, int, MyCompareMap>::iterator it = mp.begin(); it !=
mp.end(); it++) {
20         cout << "key = " << it->first << " value = " << it->second << endl;
21     }
22 }
23 int main()
24 {
25     test01();
26     system("pause");
27     return 0;
28 }

```

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 (ABCDEFGHJI), 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

```
1  #include <iostream>
2  #include <string>
3  #include <vector>
4  #include <map>
5  #include <ctime>
6  using namespace std;
7
8  #define Worker_Num 10
9  enum {
10     CEHUA,
11     MEISHU,
12     YANFA
13 };
14 class Worker {
15 public:
16     string m_Name; //姓名
17     int m_Salary; //工资
18 };
19 void CreateWorker(vector<Worker>& v) {
20     string nameSeed = "ABCDEFGHJIJ";
21     for (int i = 0; i < Worker_Num; i++) {
22         Worker worker;
23         worker.m_Name = "员工";
24         worker.m_Name += nameSeed.at(i);
25         worker.m_Salary = rand() % 10000 + 10000;
26         //将员工放入vector容器
27         v.push_back(worker);
28     }
29 }
30 void setGroup(vector<Worker>& v, multimap<int, Worker>& m) {
31     //遍历员工随机分组
32     for (vector<Worker>::iterator it = v.begin(); it != v.end(); it++) {
33         int depId = rand() % 3;
34         m.insert(make_pair(depId, *it));
35     }
36 }
37 void showWorkerBydepId(int elem, multimap<int, Worker>& m) {
38     multimap<int, Worker>::iterator pos = m.find(elem);
39     int count = m.count(elem);
40     int index = 0;
41     for (; pos != m.end() && index < count; pos++, index++) {
42         cout << "姓名: " << pos->second.m_Name << " 工资: " << pos-
43 >second.m_Salary << endl;
44     }
45 }
46 void showWorkerByGroup(multimap<int, Worker>& m) {
47     cout << "策划部门: " << endl;
48     showWorkerBydepId(CEHUA, m);
49     cout << "-----" << endl;
50     cout << "美术部门: " << endl;
```



```

50     showWorkerBydepId(MEISHU, m);
51     cout << "-----" << endl;
52     cout << "研发部门: " << endl;
53     showWorkerBydepId(YANFA, m);
54 }
55
56 int main()
57 {
58     srand((unsigned int)time(NULL));
59     //创建员工
60     vector<Worker> vWorker;
61     CreateWorker(vWorker);
62     //员工分组
63     multimap<int, Worker> mWorker;
64     setGroup(vWorker, mWorker);
65     //分组显示
66     showWorkerByGroup(mWorker);
67
68     system("pause");
69     return 0;
70 }

```

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

```

1  #include<iostream>
2  using namespace std;
3  #include <string>
4  //函数对象
5  class MyAdd {
6  public:
7      int operator()(int v1, int v2) {
8          return v1 + v2;
9      }
10 };
11 class MyPrint {
12 public:
13     MyPrint() {

```

```

14         this->count = 0;
15     }
16     void operator()(string str) {
17         cout << str << endl;
18         count++;
19     }
20     int count;//内部自己状态
21 };
22 void doPrint(MyPrint& mp, string str) {
23     mp(str);
24 }
25 void test01() {
26     MyAdd myAdd;
27     cout << myAdd(10, 10) << endl;
28
29     MyPrint myPrint;
30     myPrint("Hello C++");
31     cout << "myPrint调用次数 = " << myPrint.count << endl;
32
33     doPrint(myPrint, "Hello C++");
34 }
35 int main()
36 {
37     test01();
38     system("pause");
39     return 0;
40 }

```

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

```

1  #include<iostream>
2  using namespace std;
3  #include <vector>
4  #include <algorithm>
5  //一元谓词
6  class GreaterFive {
7  public:
8      bool operator()(int val) {
9          return val > 5;
10     }
11 };
12 void test01() {
13     vector<int>v;
14     for (int i = 0; i < 10; i++) {
15         v.push_back(i);
16     }
17     //查找容器中有没有大于5的数字

```

```

18 //GreaterFive()--匿名函数对象
19 vector<int>::iterator pos = find_if(v.begin(), v.end(), GreaterFive());
20 if (pos != v.end()) {
21     cout << "找到了, " << *pos << endl;
22 }
23 else {
24     cout << "未找到" << endl;
25 }
26 }
27 int main()
28 {
29     test01();
30     system("pause");
31     return 0;
32 }

```

4.2.3 二元谓词

```

1  #include<iostream>
2  using namespace std;
3  #include <vector>
4  #include <algorithm>
5  //二元谓词
6  class MyCompare {
7  public:
8      bool operator()(int v1,int v2) {
9          return v1 > v2;
10     }
11 };
12 void test01() {
13     vector<int>v;
14     v.push_back(10);
15     v.push_back(50);
16     v.push_back(40);
17     v.push_back(20);
18     v.push_back(70);
19
20     sort(v.begin(), v.end());
21     for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
22         cout << *it << " ";
23     }
24     cout << endl << "-----" << endl;
25     sort(v.begin(), v.end(), MyCompare()); //改变默认排序规则
26     for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
27         cout << *it << " ";
28     }
29     cout << endl;
30 }
31 int main()
32 {
33     test01();
34     system("pause");
35     return 0;
36 }

```

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：

```
1  template<class T> T plus<T>;//加法仿函数
2  template<class T> T minus<T>;//减法仿函数
3  template<class T> T multiplies<T>;//乘法仿函数
4  template<class T> T divide<T>;//除法仿函数
5  template<class T> T modulus<T>;//取模仿函数
6  template<class T> T negate<T>;//取反仿函数
```

```
1  #include<iostream>
2  using namespace std;
3  #include <functional>
4  //算数仿函数
5  void test01() {
6      negate<int>n;//取反仿函数
7      cout << n(10) << endl;
8
9      plus<int>p;//加法仿函数
10     cout << "10 + 20 = " << p(10, 20) << endl;
11 }
12 int main()
13 {
14     test01();
15     system("pause");
16     return 0;
17 }
```

4.3.3 关系仿函数

仿函数原型:

```
1 template<class T>bool equal_to<T>;//等于
2 template<class T>bool not_equal_to<T>;//不等于
3 template<class T>bool greater<T>;//大于
4 template<class T>bool greater_equal<T>;//大于等于
5 template<class T>bool less<T>;//小于
6 template<class T>bool less_equal<T>;//小于等于
```

```
1 #include<iostream>
2 using namespace std;
3 #include <functional>
4 #include<vector>
5 #include <algorithm>
6 //关系仿函数
7 class MyCompare {
8 public:
9     bool operator()(int v1, int v2) {
10         return v1 > v2;
11     }
12 };
13 void printVector(vector<int>&v) {
14     for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
15         cout << *it << " ";
16     }
17     cout << endl;
18 }
19 void test01() {
20     vector<int>v;
21     v.push_back(10);
22     v.push_back(50);
23     v.push_back(40);
24     v.push_back(20);
25     v.push_back(70);
26     printVector(v);
27     //降序
28     //sort(v.begin(), v.end(), MyCompare());
29     //greater<int>()—内建函数对象
30     sort(v.begin(), v.end(), greater<int>());
31     printVector(v);
32 }
33 }
34 int main()
35 {
36     test01();
37     system("pause");
38     return 0;
39 }
```

4.3.4 逻辑仿函数

函数原型:

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

```
1 #include<iostream>
2 using namespace std;
3 #include <functional>
4 #include<vector>
5 #include <algorithm>
6 //逻辑仿函数
7 void printVector(vector<bool>&v) {
8     for (vector<bool>::iterator it = v.begin(); it != v.end(); it++) {
9         cout << *it << " ";
10    }
11    cout << endl;
12 }
13 void test01() {
14     vector<bool>v;
15     v.push_back(true);
16     v.push_back(false);
17     v.push_back(true);
18     v.push_back(false);
19     printVector(v);
20     //利用逻辑非, 将容器v搬运到容器v2中, 并执行取反操作
21     vector<bool>v2;
22     v2.resize(v.size());
23     transform(v.begin(), v.end(), v2.begin(), logical_not<bool>());
24     printVector(v2);
25 }
26 int main()
27 {
28     test01();
29     system("pause");
30     return 0;
31 }
```

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:

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

5.1.1 for_each

函数原型:

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

```
1 #include<iostream>
2 using namespace std;
3 #include<vector>
4 #include <algorithm>
5
6 //for_each
7
8 //普通函数
9 void print01(int val) {
10     cout << val << " ";
11 }
12 //仿函数
13 class print02 {
14 public:
15     void operator()(int val) {
16         cout << val << " ";
17     }
18 };
19 void test01() {
20     vector<int>v;
21     for (int i = 0; i < 10; i++) {
22         v.push_back(i);
23     }
24     //for_each(v.begin(), v.end(), print01);
25     for_each(v.begin(), v.end(), print02());
26     cout << endl;
27 }
28 int main()
29 {
30     test01();
31     system("pause");
32     return 0;
33 }
```

5.1.2 transform

功能描述:

- 搬运容器到另一个容器中

函数原型:

```

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

```

```

1 #include<iostream>
2 using namespace std;
3 #include<vector>
4 #include <algorithm>
5
6 //transform
7
8 class Transform {
9 public:
10     int operator()(int val) {
11         return val + 10; //搬运过程可以进行一些逻辑运算
12     }
13 };
14 class MyPrint {
15 public:
16     void operator()(int val) {
17         cout << val << " ";
18     }
19 };
20 void test01() {
21     vector<int>v;
22     for (int i = 0; i < 10; i++) {
23         v.push_back(i);
24     }
25     vector<int>vTarget; //目标容器
26     vTarget.resize(v.size()); //目标容器需要提前开辟空间
27     transform(v.begin(), v.end(), vTarget.begin(), Transform());
28     for_each(vTarget.begin(), vTarget.end(), MyPrint());
29     cout << endl;
30 }
31 int main()
32 {
33     test01();
34     system("pause");
35     return 0;
36 }

```

5.2 常用查找算法

算法简介：

```

1 find                //查找元素
2 find_if             //按条件查找元素
3 adjacent_find       //查找相邻重复元素
4 binary_search        //二分查找法
5 count               //统计元素个数
6 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:

```
1 find(iterator beg,iterator end,value);
2 //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()
3 //beg 开始迭代器
4 //end 结束迭代器
5 //value 查找的元素
```

```
1 #include<iostream>
2 using namespace std;
3 #include<vector>
4 #include <algorithm>
5 #include <string>
6 //find
7 class Person {
8 public:
9     Person(string name, int age) {
10         this->m_Name = name;
11         this->m_Age = age;
12     }
13     //重载 == 让底层find知道如何比对Pers自定义数据类型
14     bool operator==(const Person& p) {
15         if (this->m_Name == p.m_Name && this->m_Age == p.m_Age) {
16             return true;
17         }
18         return false;
19     }
20     string m_Name;
21     int m_Age;
22 };
23
24 void test01() {
25     //查找内置数据类型
26     vector<int>v;
27     for (int i = 0; i < 10; i++) {
28         v.push_back(i);
29     }
30     vector<int>::iterator pos = find(v.begin(), v.end(), 3);
31     if (pos != v.end()) {
32         cout << "找到了元素: " << *pos << endl;
33     }
34     else {
35         cout << "未找到该元素" << endl;
36     }
37     //查找自定义数据类型
38     vector<Person>vp;
39     Person p1("刘备", 35);
40     Person p2("曹操", 45);
41     Person p3("孙权", 40);
42     Person p4("赵云", 25);
43     vp.push_back(p1);
```

```

44     vp.push_back(p2);
45     vp.push_back(p3);
46     vp.push_back(p4);
47
48     Person pp("赵云", 25);
49     vector<Person>::iterator posp = find(vp.begin(), vp.end(), pp);
50     if (posp != vp.end()) {
51         cout << "找到了元素--姓名: " << posp->m_Name << " 年龄: " << posp->m_Age
<< endl;
52     }
53     else {
54         cout << "未找到该元素" << endl;
55     }
56 }
57 int main()
58 {
59     test01();
60     system("pause");
61     return 0;
62 }

```

5.2.2 find_if

功能描述:

- 按条件查找元素

函数原型:

```

1 find_if(iterator beg, iterator end, _Pred);
2 //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
3 //beg 开始迭代器
4 //end 结束迭代器
5 //_Pred 函数或者谓词

```

```

1 #include<iostream>
2 using namespace std;
3 #include<vector>
4 #include <algorithm>
5 #include <string>
6 //find_if
7 class GreaterFive {
8 public:
9     bool operator()(int val) {
10         if (val > 5) {
11             return true;
12         }
13         return false;
14     }
15 };
16 class Person {
17 public:
18     Person(string name, int age) {
19         this->m_Name = name;
20         this->m_Age = age;
21     }
22     string m_Name;

```

```

23     int m_Age;
24 };
25 class Greater40 {
26 public:
27     bool operator()(const Person& p) {
28         if (p.m_Age >= 40) {
29             return true;
30         }
31         return false;
32     }
33 };
34 void test01() {
35     //查找内置数据类型
36     vector<int>v;
37     for (int i = 0; i < 10; i++) {
38         v.push_back(i);
39     }
40     vector<int>::iterator pos = find_if(v.begin(), v.end(), GreaterFive());
41     if (pos != v.end()) {
42         cout << "找到该元素: " << *pos << endl;
43     }
44     else {
45         cout << "未找到该元素" << endl;
46     }
47     //查找自定义数据类型
48     vector<Person>vp;
49     Person p1("刘备", 35);
50     Person p2("曹操", 45);
51     Person p3("孙权", 40);
52     Person p4("赵云", 25);
53     vp.push_back(p1);
54     vp.push_back(p2);
55     vp.push_back(p3);
56     vp.push_back(p4);
57
58     //找年龄大于40的人
59     vector<Person>::iterator posp = find_if(vp.begin(), vp.end(),
Greater40());
60     if (posp != vp.end()) {
61         cout << "找到该元素 姓名: " << posp->m_Name << " 年龄: " << posp->m_Age
<< endl;
62     }
63     else {
64         cout << "未找到该元素" << endl;
65     }
66 }
67 int main()
68 {
69     test01();
70     system("pause");
71     return 0;
72 }

```

5.2.3 adjacent_find

功能描述:

- 查找相邻重复元素

函数原型:

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

```
1 #include<iostream>
2 using namespace std;
3 #include<vector>
4 #include <algorithm>
5 //adjacent_find
6
7 void test01() {
8     vector<int>v;
9     v.push_back(0);
10    v.push_back(1);
11    v.push_back(2);
12    v.push_back(3);
13    v.push_back(2);
14    v.push_back(3);
15    v.push_back(3);
16    v.push_back(4);
17
18    vector<int>::iterator pos = adjacent_find(v.begin(), v.end());
19    if (pos != v.end()) {
20        cout << "找到相邻重复元素:" << *pos << endl;
21    }
22    else {
23        cout << "未找到相邻重复元素" << endl;
24    }
25 }
26 int main()
27 {
28     test01();
29     system("pause");
30     return 0;
31 }
```

5.2.4 binary_search

功能描述:

- 查找指定元素是否存在

函数原型:

```

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

```

```

1 #include<iostream>
2 using namespace std;
3 #include<vector>
4 #include <algorithm>
5 //binary_search
6
7 void test01() {
8     vector<int>v;
9     for (int i = 0; i < 10; i++) {
10         v.push_back(i);
11     }
12     //查找容器中是否有9元素
13     //注意：容器必须是有序序列
14     bool ret = binary_search(v.begin(), v.end(), 9);
15     if (ret) {
16         cout << "找到了该元素" << endl;
17     }
18     else {
19         cout << "未找到该元素" << endl;
20     }
21 }
22 int main()
23 {
24     test01();
25     system("pause");
26     return 0;
27 }

```

5.2.5 count

功能描述：

- 统计元素个数

函数原型：

```

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

```

```

1 #include<iostream>
2 using namespace std;
3 #include<vector>
4 #include <algorithm>
5 #include <string>
6 //count
7 class Person {

```

```

8 public:
9     Person(string name, int age) {
10         this->m_Name = name;
11         this->m_Age = age;
12     }
13     //重载==
14     bool operator==(const Person& p) {
15         return this->m_Age == p.m_Age;
16     }
17     string m_Name;
18     int m_Age;
19 };
20 void test01() {
21     //统计内置数据类型
22     vector<int>v;
23     for (int i = 0; i < 10; i++) {
24         v.push_back(i);
25         v.push_back(9);
26     }
27     int num = count(v.begin(), v.end(), 9);
28     cout << "v中元素9的个数为: " << num << endl;
29     //统计自定义数据类型
30     vector<Person>vp;
31     Person p1("刘备", 35);
32     Person p2("曹操", 45);
33     Person p3("孙权", 40);
34     Person p4("赵云", 25);
35     Person p5("曹操", 40);
36
37     Person p("诸葛亮", 35);
38     vp.push_back(p1);
39     vp.push_back(p2);
40     vp.push_back(p3);
41     vp.push_back(p4);
42     vp.push_back(p5);
43
44     int cnt = count(v.begin(), v.end(), p);
45     cout << "vp中与诸葛亮同岁的人有" << cnt << "个" << endl;
46 }
47 int main()
48 {
49     test01();
50     system("pause");
51     return 0;
52 }

```

5.2.6 count_if

功能描述:

- 按条件统计元素个数

函数原型:

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

```
1 #include<iostream>
2 using namespace std;
3 #include<vector>
4 #include <algorithm>
5 #include <string>
6 //count_if
7 class Greater20 {
8 public:
9     bool operator()(int val) {
10         return val > 20;
11     }
12 };
13 class Person {
14 public:
15     Person(string name, int age) {
16         this->m_Name = name;
17         this->m_Age = age;
18     }
19     int m_Age;
20     string m_Name;
21 };
22 class AgeGreater20 {
23 public:
24     bool operator()(const Person& p) {
25         return p.m_Age > 20;
26     }
27 };
28 void test01() {
29     //统计内置数据类型
30     vector<int>v;
31     v.push_back(10);
32     v.push_back(40);
33     v.push_back(30);
34     v.push_back(20);
35     v.push_back(40);
36     v.push_back(20);
37
38     int num = count_if(v.begin(), v.end(), Greater20());
39     cout << "v中大于20的元素个数" << num << endl;
40
41     //统计自定义数据类型
42     vector<Person>vp;
43     Person p1("刘备", 35);
44     Person p2("曹操", 45);
45     Person p3("孙权", 40);
46     Person p4("赵云", 25);
47     Person p5("曹操", 40);
48     vp.push_back(p1);
49     vp.push_back(p2);
50     vp.push_back(p3);
```

```

51     vp.push_back(p4);
52     vp.push_back(p5);
53
54     int cnt = count_if(vp.begin(), vp.end(), AgeGreater20());
55     cout << "vp中人物年龄大于20人数为: " << cnt << endl;
56 }
57 int main()
58 {
59     test01();
60     system("pause");
61     return 0;
62 }

```

5.3 常用排序算法

算法简介：

```

1  sort      //对容器内元素进行排序
2  random_shuffle  //洗牌，指定范围内的元素随机调整次序
3  merge     //容器元素合并，并存储到另一容器中
4  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：

```

1  sort(iterator beg,iterator end,_Pred);
2  //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
3  //beg 开始迭代器
4  //end 结束迭代器
5  //_Pred 谓词

```

```

1  #include<iostream>
2  using namespace std;
3  #include<vector>
4  #include <algorithm>
5  #include <functional>
6  //sort
7  void myPrint(int val) {
8      cout << val << " ";
9  }
10 void test01() {
11     vector<int>v;
12     v.push_back(10);
13     v.push_back(40);
14     v.push_back(30);
15     v.push_back(20);
16     v.push_back(40);
17     v.push_back(20);
18
19     //利用sort进行升序
20     sort(v.begin(),v.end());

```



```

21     for_each(v.begin(), v.end(), myPrint);
22     cout << endl;
23     //改变为降序
24     sort(v.begin(), v.end(), greater<int>());
25     for_each(v.begin(), v.end(), myPrint);
26     cout << endl;
27 }
28 int main()
29 {
30     test01();
31     system("pause");
32     return 0;
33 }

```

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:

```

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

```

```

1 #include<iostream>
2 using namespace std;
3 #include<vector>
4 #include <algorithm>
5 #include<ctime>
6 //random_shuffle
7 void myPrint(int val) {
8     cout << val << " ";
9 }
10 void test01() {
11     vector<int>v;
12     for (int i = 0; i < 10; i++) {
13         v.push_back(i);
14     }
15     //利用洗牌算法, 打乱顺序
16     random_shuffle(v.begin(), v.end());
17     for_each(v.begin(), v.end(), myPrint);
18     cout << endl;
19 }
20 int main()
21 {
22     srand((unsigned int)time(NULL));
23     test01();
24     system("pause");
25     return 0;
26 }

```

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:

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

```
1 #include<iostream>
2 using namespace std;
3 #include<vector>
4 #include <algorithm>
5 //merge
6 class MyPrint {
7 public:
8     void operator()(int val) {
9         cout << val << " ";
10    }
11 };
12 void test01() {
13     vector<int>v1;
14     vector<int>v2;
15     for (int i = 0; i < 10; i++) {
16         v1.push_back(i);
17         v2.push_back(i + 10);
18     }
19
20     vector<int>vTarget;//目标容器
21     vTarget.resize(v1.size() + v2.size());//提前分配空间
22
23     merge(v1.begin(), v1.end(), v2.begin(), v2.end(), vTarget.begin());
24     for_each(vTarget.begin(), vTarget.end(), MyPrint());
25     cout << endl;
26 }
27 int main()
28 {
29     test01();
30     system("pause");
31     return 0;
32 }
```

5.3.4 reverse

功能描述:

- 将容器内元素进行反转

函数原型:

```

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

```

```

1 #include<iostream>
2 using namespace std;
3 #include<vector>
4 #include <algorithm>
5 //reverse
6 class MyPrint {
7 public:
8     void operator()(int val) {
9         cout << val << " ";
10    }
11 };
12 void test01() {
13     vector<int>v;
14     for (int i = 0; i < 10; i++) {
15         v.push_back(i);
16     }
17     for_each(v.begin(), v.end(), MyPrint());
18     cout << endl;
19     reverse(v.begin(), v.end());
20     for_each(v.begin(), v.end(), MyPrint());
21     cout << endl;
22 }
23 int main()
24 {
25     test01();
26     system("pause");
27     return 0;
28 }

```

5.4 常用拷贝和替换算法

算法简介：

```

1 copy //容器内指定范围的元素拷贝到另一容器中
2 replace //将容器内指定范围的旧元素修改为新元素
3 replace_if //容器内指定范围满足条件的元素替换为新元素
4 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：

```

1 copy(iterator beg,iterator end,iterator dest);
2 //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
3 //beg 开始迭代器
4 //end 结束迭代器
5 //dest 目标容器开始迭代器

```

```

1 #include<iostream>
2 using namespace std;
3 #include<vector>
4 #include <algorithm>
5 //copy
6 class MyPrint {
7 public:
8     void operator()(int val) {
9         cout << val << " ";
10    }
11 };
12 void test01() {
13     vector<int>v;
14     for (int i = 0; i < 10; i++) {
15         v.push_back(i);
16     }
17     vector<int>v2;
18     v2.resize(v.size());
19     copy(v.begin(), v.end(), v2.begin());
20     for_each(v2.begin(), v2.end(), MyPrint());
21     cout << endl;
22 }
23 int main()
24 {
25     test01();
26     system("pause");
27     return 0;
28 }

```

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:

```

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

```

```

1 #include<iostream>
2 using namespace std;
3 #include<vector>
4 #include <algorithm>
5 //replace
6 class MyPrint {

```

```

7 public:
8     void operator()(int val) {
9         cout << val << " ";
10    }
11 };
12 void test01() {
13     vector<int>v;
14     for (int i = 0; i < 10; i++) {
15         v.push_back(i);
16     }
17     for_each(v.begin(), v.end(), MyPrint());
18     cout << endl;
19     replace(v.begin(), v.end(), 1, 2); //容器中1换成2
20     for_each(v.begin(), v.end(), MyPrint());
21     cout << endl;
22 }
23 int main()
24 {
25     test01();
26     system("pause");
27     return 0;
28 }

```

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:

```

1 replace_if(iterator beg, iterator end, _Pred, newvalue);
2 //按条件替换元素，满足条件的替换成指定元素
3 //beg 开始迭代器
4 //end 结束迭代器
5 //_Pred 谓词
6 //newvalue 新元素

```

```

1 #include<iostream>
2 using namespace std;
3 #include<vector>
4 #include <algorithm>
5 //replace
6 class MyPrint {
7 public:
8     void operator()(int val) {
9         cout << val << " ";
10    }
11 };
12 class Greater5 {
13 public:
14     bool operator()(int val) {
15         return val > 5;
16     }
17 };
18 void test01() {
19     vector<int>v;

```

```

20     for (int i = 0; i < 10; i++) {
21         v.push_back(i);
22     }
23     for_each(v.begin(), v.end(), MyPrint());
24     cout << endl;
25     replace_if(v.begin(), v.end(), Greater5(), 0);
26     for_each(v.begin(), v.end(), MyPrint());
27     cout << endl;
28 }
29 int main()
30 {
31     test01();
32     system("pause");
33     return 0;
34 }

```

5.4.4 swap

功能描述:

- 互换两个容器的元素

函数原型:

```

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

```

```

1 #include<iostream>
2 using namespace std;
3 #include<vector>
4 #include <algorithm>
5 //replace_if
6 class MyPrint {
7 public:
8     void operator()(int val) {
9         cout << val << " ";
10    }
11 };
12 class Greater5 {
13 public:
14     bool operator()(int val) {
15         return val > 5;
16    }
17 };
18 void test01() {
19     vector<int>v1;
20     vector<int>v2;
21     for (int i = 0; i < 10; i++) {
22         v1.push_back(i);
23         v2.push_back(i + 10);
24     }
25     cout << "互换前: " << endl;
26     for_each(v1.begin(), v1.end(), MyPrint());
27     cout << endl;
28     for_each(v2.begin(), v2.end(), MyPrint());

```

```

29     cout << endl;
30     swap(v1, v2);
31     cout << "互换后: " << endl;
32     for_each(v1.begin(), v1.end(), MyPrint());
33     cout << endl;
34     for_each(v2.begin(), v2.end(), MyPrint());
35     cout << endl;
36 }
37 int main()
38 {
39     test01();
40     system("pause");
41     return 0;
42 }

```

5.5 常用算数生成函数

算法简介:

```

1 accumulate //计算容器元素累计总和
2 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 accumulate

功能描述:

- 计算区间内容器元素的总和

函数原型:

```

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

```

```

1 #include<iostream>
2 using namespace std;
3 #include<vector>
4 #include <numeric>
5 //accumulate
6 void test01() {
7     vector<int>v;
8     for (int i = 0; i <= 100; i++) {
9         v.push_back(i);
10    }
11    //参数3 起始累加值
12    int sum = accumulate(v.begin(), v.end(), 0);
13    cout << "sum = " << sum << endl;
14 }
15 int main()
16 {

```

```

17     test01();
18     system("pause");
19     return 0;
20 }

```

5.5.2 fill

功能描述:

- 向容器中填充指定的值

函数原型:

```

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

```

```

1 #include<iostream>
2 using namespace std;
3 #include<vector>
4 #include <numeric>
5 #include <algorithm>
6 //fill
7 class MyPrint {
8 public:
9     void operator()(int val) {
10         cout << val << " ";
11     }
12 };
13
14 void test01() {
15     vector<int>v;
16     v.resize(10);
17     for_each(v.begin(), v.end(), MyPrint());
18     cout << endl;
19     fill(v.begin(), v.end(), 1);
20     for_each(v.begin(), v.end(), MyPrint());
21     cout << endl;
22 }
23 int main()
24 {
25     test01();
26     system("pause");
27     return 0;
28 }

```

5.6 常用集合算法

算法简介:

```

1 set_intersection //求两个容器的交集
2 set_union       //求两个容器的并集
3 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:

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

```
1 #include<iostream>
2 using namespace std;
3 #include<vector>
4 #include <algorithm>
5 //set_intersaction
6 class MyPrint {
7 public:
8     void operator()(int val) {
9         cout << val << " ";
10    }
11 };
12
13 void test01() {
14     vector<int>v1;
15     vector<int>v2;
16     for (int i = 0; i < 10; i++) {
17         v1.push_back(i);
18         v2.push_back(i + 5);
19     }
20     vector<int>vTarget;
21     vTarget.resize(min(v1.size(), v2.size()));//提前开辟空间
22     vector<int>::iterator itEnd = set_intersection(v1.begin(), v1.end(),
v2.begin(), v2.end(), vTarget.begin());
23     for_each(vTarget.begin(), itEnd, MyPrint());
24     cout << endl;
25 }
26 int main()
27 {
28     test01();
29     system("pause");
30     return 0;
31 }
```

5.6.2 set_union

功能描述:

- 求两个集合的并集

函数原型:

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

```
1 #include<iostream>
2 using namespace std;
3 #include<vector>
4 #include <algorithm>
5 //set_union
6 class MyPrint {
7 public:
8     void operator()(int val) {
9         cout << val << " ";
10    }
11 };
12
13 void test01() {
14     vector<int>v1;
15     vector<int>v2;
16     for (int i = 0; i < 10; i++) {
17         v1.push_back(i);
18         v2.push_back(i + 5);
19     }
20     vector<int>vTarget;
21     vTarget.resize(v1.size()+v2.size());//提前开辟空间
22     vector<int>::iterator itEnd = set_union(v1.begin(), v1.end(),
23     v2.begin(), v2.end(), vTarget.begin());
24     for_each(vTarget.begin(), itEnd, MyPrint());
25     cout << endl;
26 }
27
28 int main()
29 {
30     test01();
31     system("pause");
32     return 0;
33 }
```

5.6.3 set_difference

功能描述:

- 求两个集合的差集

函数原型:

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

```
1 #include<iostream>
2 using namespace std;
3 #include<vector>
4 #include <algorithm>
5 //set_union
6 class MyPrint {
7 public:
8     void operator()(int val) {
9         cout << val << " ";
10    }
11 };
12
13 void test01() {
14     vector<int>v1;
15     vector<int>v2;
16     for (int i = 0; i < 10; i++) {
17         v1.push_back(i);
18         v2.push_back(i + 5);
19     }
20     vector<int>vTarget;
21     vTarget.resize(max(v1.size(),v2.size()));//提前开辟空间
22     cout << "v1和v2的差集为: " << endl;
23     vector<int>::iterator itEnd = set_difference(v1.begin(), v1.end(),
v2.begin(), v2.end(), vTarget.begin());
24     for_each(vTarget.begin(), itEnd, MyPrint());
25     cout << endl;
26     cout << "v2和v1的差集为: " << endl;
27     itEnd = set_difference(v2.begin(), v2.end(), v1.begin(), v1.end(),
vTarget.begin());
28     for_each(vTarget.begin(), itEnd, MyPrint());
29     cout << endl;
30 }
31 int main()
32 {
33     test01();
34     system("pause");
35     return 0;
36 }
```

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