

# C++提高编程

本阶段主要针对C++泛型编程和STL技术，探讨C++更深层的使用

## 1、模板

### 1.1、模板的概念

模板就是建立通用的模具，大大提高复用性

模板的特点：

- 模板不可以直接使用，它只是一个框架
- 模板的通用并不是万能的

### 1.2、函数模板

- C++另一种编程思想称为泛型编程，主要利用的技术就是模板
- 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具体指定，用一个虚拟的类型代表

语法：

```
1  template<typename T>
2  函数声明或定义
```

声明一个模板，告诉编译器后面代码中紧跟着的T不要报错，T是一个通用数据类型

解释：

template -- 声明创建模板

typename -- 表面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，通常为大写字母

```
1  //交换整型函数
2  void swapInt(int& a, int& b) {
3      int temp = a;
4      a = b;
5      b = temp;
6  }
7  //交换浮点型函数
8  void swapDouble(double& a, double& b) {
9      double temp = a;
10     a = b;
11     b = temp;
```

```

12 }
13 //利用模板提供通用的交换函数
14 template<typename T>
15 void mySwap(T& a, T& b)
16 {
17     T temp = a;
18     a = b;
19     b = temp;
20 }
21 void test01()
22 {
23     int a = 10;
24     int b = 20;
25     //swapInt(a, b);
26     //利用模板实现交换
27     //1、自动类型推导
28     mySwap(a, b);
29     //2、显示指定类型
30     mySwap<int>(a, b);
31     cout << "a = " << a << endl;
32     cout << "b = " << b << endl
33 }
34 int main()
35     test01();
36     system("pause");
37     return 0;
38 }

```

总结:

- 函数模板利用关键字template
- 使用函数模板有两种方式：自动类型推导、显示指定类型
- 模板的目的是为了提高复用性，将类型参数化

### 1.2.2、函数模板注意事项

注意事项:

- 自动类型推导，必须推导出一致的数据类型T，才可以使用
- 模板必须要确定出T的数据类型，才可以使用（即使没有使用T也必须指定）

模板指定的数据类型必须和元素定义一致，不能char转int

```

1 //利用模板提供通用的交换函数
2 template<class T>
3 void mySwap(T& a, T& b)
4 {
5     T temp = a;
6     a = b;
7     b = temp;
8 }
9 // 1、自动类型推导，必须推导出一致的数据类型T,才可以使用
10 void test01()
11 {
12     int a = 10;
13     int b = 20;

```

```

14     char c = 'c';
15     mySwap(a, b); // 正确，可以推导出一致的T
16     //mySwap(a, c); // 错误，推导不出一致的T类型
17 }
18 // 2、模板必须要确定出T的数据类型，才可以使用
19 template<class T>
20 void func()
21 {
22     cout << "func 调用" << endl;
23 }
24 void test02()
25 {
26     //func(); //错误，模板不能独立使用，必须确定出T的类型
27     func<int>(); //利用显示指定类型的方式，给T一个类型，才可以使用该模板
28 }
29 int main() {
30     test01();
31     test02();
32     system("pause");
33     return 0;
34 }

```

总结：使用模板时必须确定出通用数据类型T，并且能推导出一致的类型

### 1.2.3、函数模板案例

案例描述：

- 利用函数模板封装一个排序的函数，可以对不同数据类型数组进行排序
- 排序规则从大到小，排序算法为选择排序
- 分别利用char数组和int数组进行测试

```

1 //交换的函数模板
2 template<typename T>
3 void mySwap(T &a, T&b)
4 {
5     T temp = a;
6     a = b;
7     b = temp;
8 }
9 template<class T> // 也可以替换成typename
10 //利用选择排序，进行对数组从大到小的排序
11 void mySort(T arr[], int len)
12 {
13     for (int i = 0; i < len; i++)
14     {
15         int max = i; //最大数的下标
16         for (int j = i + 1; j < len; j++)
17         {
18             if (arr[max] < arr[j])
19             {
20                 max = j;
21             }
22         }
23         if (max != i) //如果最大数的下标不是i，交换两者

```

```

24     {
25         mySwap(arr[max], arr[i]);
26     }
27 }
28 }
29 template<typename T>
30 void printArray(T arr[], int len) {
31
32     for (int i = 0; i < len; i++) {
33         cout << arr[i] << " ";
34     }
35     cout << endl;
36 }
37 void test01()
38 {
39     //测试char数组
40     char charArr[] = "bdcfeagh";
41     int num = sizeof(charArr) / sizeof(char);
42     mySort(charArr, num);
43     printArray(charArr, num);
44 }
45 void test02()
46 {
47     //测试int数组
48     int intArr[] = { 7, 5, 8, 1, 3, 9, 2, 4, 6 };
49     int num = sizeof(intArr) / sizeof(int);
50     mySort(intArr, num);
51     printArray(intArr, num);
52 }
53 int main() {
54     test01();
55     test02();
56     system("pause");
57     return 0;
58 }

```

总结：模板可以提高代码复用，需要熟练掌握

### 1.2.4、普通函数与函数模板的区别

普通函数与函数模板区别：

- 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、普通函数调用时可以自动类型转换

cout << myAdd01(a, c) << endl; //正确，将char类型的'c'隐式转换为int类型 'c' 对应 ASCII码 99

2、使用自动类型推导调用函数模板时，不会发生隐式类型转换

//myAdd02(a, c); // 报错，使用自动类型推导时，不会发生隐式类型转换

//使用两个参数也不会报错

```

1  template<class T1,class T2>
2  void myAdd02(T1 a, T2 b)
3  {
4      return a + b;
5  }
6  myAdd02(a, c);

```

### 3、使用指定类型调用函数模板时，可以发生隐式类型转换

myAdd02(a, c); //正确，如果用显示指定类型，可以发生隐式类型转换

### 引用的方式不可以发生隐式类型转换

```

1  template<class T>
2  T myAdd02(T &a, T &b)
3  {
4      return a + b;
5  }
6  int a=10;
7  char b='a';
8  myAdd02<int>(a,b); //没有与参数列表匹配的 函数模板 "myAdd" 实例

```

```

1  //普通函数
2  int myAdd01(int a, int b)
3  {
4      return a + b;
5  }
6  //函数模板
7  template<class T>
8  T myAdd02(T a, T b)
9  {
10     return a + b;
11 }
12 //使用函数模板时，如果用自动类型推导，不会发生自动类型转换,即隐式类型转换
13 void test01()
14 {
15     int a = 10;
16     int b = 20;
17     char c = 'c';
18     cout << myAdd01(a, c) << endl; //正确，将char类型的'c'隐式转换为int类型 'c'
    对应 ASCII码 99
19     //myAdd02(a, c); // 报错，使用自动类型推导时，不会发生隐式类型转换
20     myAdd02<int>(a, c); //正确，如果用显示指定类型，可以发生隐式类型转换
21 }
22 int main() {
23     test01();
24     system("pause");
25     return 0;
26 }

```

总结：建议使用显示指定类型的方式调用函数模板，因为可以自己确定通用类型T

### 1.2.5、普通函数与函数模板的调用规则

调用规则：

- 1、如果函数模板和普通函数都可以实现，优先调用普通函数（重载）
- 2、可以通过空模板参数列表强制调用函数模板

`myPrint<>(a, b);` //调用函数模板

- 3、函数模板也可以发生重载
- 4、如果函数模板可以产生更好的匹配，优先调用函数模板

普通函数需要隐式类型转换，模板函数不需要

```
1 void myPrint(int a, int b)
2 {
3     cout << "调用的普通函数" << endl;
4 }
5 template<typename T>
6 void myPrint(T a, T b)
7 {
8     cout << "调用的模板" << endl;
9 }
10 char c1 = 'a';
11     char c2 = 'b';
12     myPrint(c1, c2); //调用函数模板
```

```
1 //普通函数与函数模板调用规则
2 void myPrint(int a, int b)
3 {
4     cout << "调用的普通函数" << endl;
5 }
6 template<typename T>
7 void myPrint(T a, T b)
8 {
9     cout << "调用的模板" << endl;
10 }
11 template<typename T>
12 void myPrint(T a, T b, T c)
13 {
14     cout << "调用重载的模板" << endl;
15 }
16 void test01()
17 {
18     //1、如果函数模板和普通函数都可以实现，优先调用普通函数
19     // 注意 如果告诉编译器 普通函数是有的，但只是声明没有实现，或者不在当前文件内实现，就会报错找不到
20     int a = 10;
21     int b = 20;
22     myPrint(a, b); //调用普通函数
23     //2、可以通过空模板参数列表来强制调用函数模板
24     myPrint<>(a, b); //调用函数模板
25     //3、函数模板也可以发生重载
26     int c = 30;
```

```

27     myPrint(a, b, c); //调用重载的函数模板
28     //4、 如果函数模板可以产生更好的匹配,优先调用函数模板
29     char c1 = 'a';
30     char c2 = 'b';
31     myPrint(c1, c2); //调用函数模板
32 }
33 int main() {
34     test01();
35     system("pause");
36     return 0;
37 }

```

### 1.2.6、模板的局限性

局限性：模板的通用性并不是万能的

```

1     template<class T>
2     void f(T a, T b)
3     {
4         a = b;
5     }

```

如果传入的a和b是数组，就无法实现了

可以通过运算符重载和模板特定类型重载解决

```

1     template<class T>
2     void f(T a, T b)
3     {
4         if(a > b) { ... }
5     }

```

如果传入T的数据类型是自定义的，也无法正常运行

C++为了解决这种问题，提供模板的重载，可以为这些特定的类型提供具体化的模板

```
template<> bool myCompare(Person &p1, Person &p2){}
```

```

1     #include<iostream>
2     using namespace std;
3
4     class Person {
5     public:
6         Person(string name, int age) {
7             this->name = name;
8             this->age = age;
9         }
10        string name;
11        int age;
12        bool operator==(const Person& p1) {
13            cout<<"this is operator" << endl;
14            if (this->name == p1.name && this->age == p1.age)    return true;
15            else return false;
16        }
17    };

```

```

18 template<class T>
19 bool myCompare(T& a, T& b) {
20     if (a == b) return true;
21     else return false;
22 }
23 template<> bool myCompare(Person &p1, Person &p2) {
24     cout << "this is muban chongzai" << endl;
25     if (p1.name==p2.name&& p1.age==p2.age) return true;
26     else return false;
27 }
28 int main() {
29     Person p1("Tom", 20);
30     Person p2("Tom", 20);
31     cout << myCompare(p1,p2) << endl;
32     return 0;
33 }

```

总结:

- 利用具体化的模板，可以解决自定义类型的通用化
- 学习模板并不是为了写模板，而是在STL能够运用系统提供的模板

## 1.3、类模板

### 1.3.1、类模板语法

类模板作用:

建立一个通用类，类中的成员数据类型可以不具体制定，用一个虚拟的类型代表

```

1  #include <string>
2  //类模板
3  template<class NameType, class AgeType>
4  class Person
5  {
6  public:
7      Person(NameType name, AgeType age)
8      {
9          this->mName = name;
10         this->mAge = age;
11     }
12     void showPerson()
13     {
14         cout << "name: " << this->mName << " age: " << this->mAge << endl;
15     }
16 public:
17     NameType mName;
18     AgeType mAge;
19 };
20 void test01()
21 {
22     // 指定NameType 为string类型, AgeType 为 int类型
23     Person<string, int>P1("孙悟空", 999);
24     P1.showPerson();
25 }
26 int main() {

```



```

27     test01();
28     system("pause");
29     return 0;
30 }

```

总结：类模板和函数模板语法相似，在声明模板template后面加类，此类称为类模板

### 1.3.2、类模板与函数模板区别

类模板与函数模板区别主要有两点：

1. 类模板没有自动类型推导的使用方式
2. 类模板在模板参数列表中可以有默认参数

类模板中的模板参数列表指定默认参数，调用时可以不加该参数的类型。如果加了参数类型根据加的来忽略默认参数

```

1  template<class NameType, class AgeType = int>
2  Person <string> p("猪八戒", 999); //类模板中的模板参数列表 可以指定默认参数
3  Person <string, char>p("孙悟空", 'f');

```

```

1  #include <string>
2  //类模板
3  template<class NameType, class AgeType = int>
4  class Person
5  {
6  public:
7      Person(NameType name, AgeType age)
8      {
9          this->mName = name;
10         this->mAge = age;
11     }
12     void showPerson()
13     {
14         cout << "name: " << this->mName << " age: " << this->mAge << endl;
15     }
16 public:
17     NameType mName;
18     AgeType mAge;
19 };
20 //1、类模板没有自动类型推导的使用方式
21 void test01()
22 {
23     // Person p("孙悟空", 1000); // 错误 类模板使用时候，不可以用自动类型推导
24     Person <string, int>p("孙悟空", 1000); //必须使用显示指定类型的方式，使用类模板
25     p.showPerson();
26 }
27 //2、类模板在模板参数列表中可以有默认参数
28 void test02()
29 {
30     Person <string> p("猪八戒", 999); //类模板中的模板参数列表 可以指定默认参数
31     p.showPerson();
32 }
33 int main() {
34     test01();

```

```

35     test02();
36     system("pause");
37     return 0;
38 }

```

### 1.3.3、类模板中成员函数创建时机

类模板中成员函数和普通类中成员函数创建时机区别：

- 普通类中的成员函数一开始就可以创建
- 类模板中成员函数在调用时才创建

MyClass m;类模板MyClass只调用了Person1即只生成了Person1的对象，并没有调用过Person2也没有生成Person2的对象，不能使用Person2的成员函数

```

1  class Person1
2  {
3  public:
4      void showPerson1()
5      {
6          cout << "Person1 show" << endl;
7      }
8  };
9  class Person2
10 {
11 public:
12     void showPerson2()
13     {
14         cout << "Person2 show" << endl;
15     }
16 };
17 template<class T>
18 class MyClass
19 {
20 public:
21     T obj;
22
23     //类模板中的成员函数，并不是一开始就创建的，而是在模板调用时再生成
24
25     void fun1() { obj.showPerson1(); }
26     void fun2() { obj.showPerson2(); }
27
28 };
29 void test01()
30 {
31     MyClass<Person1> m;
32     m.fun1();
33     //m.fun2(); //编译会出错，说明函数调用才会去创建成员函数
34 }
35 int main() {
36     test01();
37     system("pause");
38     return 0;
39 }

```

总结：类模板中的成员函数并不是一开始就创建的，在调用时才去创建

### 1.3.4、类模板对象做函数参数

学习目标：类模板实例化出的对象，向函数传参的方式

三种传入方式：

- 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 <string>
2  //类模板
3  template<class NameType, class AgeType = int>
4  class Person
5  {
6  public:
7      Person(NameType name, AgeType age)
8      {
9          this->mName = name;
10         this->mAge = age;
11     }
12     void showPerson()
13     {
14         cout << "name: " << this->mName << " age: " << this->mAge << endl;
15     }
16 public:
17     NameType mName;
18     AgeType mAge;
19 };
20 //1、指定传入的类型
21 void printPerson1(Person<string, int> &p)
22 {
23     p.showPerson();
24 }
25 void test01()
26 {
27     Person <string, int >p("孙悟空", 100);
28     printPerson1(p);    //使用函数模板，可以不指定数据类型
29 }
30 //2、参数模板化
31 template <class T1, class T2>
32 void printPerson2(Person<T1, T2>&p)
33 {
34     p.showPerson();
35     cout << "T1的类型为: " << typeid(T1).name() << endl;
36     cout << "T2的类型为: " << typeid(T2).name() << endl;
37 }
38 void test02()
39 {
40     Person <string, int >p("猪八戒", 90);
41     printPerson2(p);
42 }
43 //3、整个类模板化
44 template<class T>
```

```

45 void printPerson3(T & p)
46 {
47     cout << "T的类型为:  " << typeid(T).name() << endl;
48     p.showPerson();
49
50 }
51 void test03()
52 {
53     Person <string, int >p("唐僧", 30);
54     printPerson3(p);
55 }
56 int main() {
57     test01();
58     test02();
59     test03();
60     system("pause");
61     return 0;
62 }

```

总结：第一种使用比较广泛：指定传入的类型

第二第三种都是类模板配合函数模板，比较复杂

### 1.3.5、类模板与继承

使用类模板时要声明T的类型

类模板碰到继承，需要注意：

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

拷贝构造

MyArray arr3=arr1;

拷贝构造不支持直接赋值arr3=arr1;

运算符重载

MyArray arr4(100);

arr4 = arr1;

```

1 //类模板继承类模板 ,可以用T2指定父类中的T类型
2 template<class T1, class T2>
3 class Son2 :public Base<T2>

```

```

1 template<class T>
2 class Base
3 {
4     T m;
5 };
6 //class Son:public Base //错误, c++编译需要给子类分配内存, 必须知道父类中T的类型才可
  以向下继承
7 class Son :public Base<int> //必须指定一个类型
8 {

```

```

9   };
10  void test01()
11  {
12      Son c;
13  }
14  //类模板继承类模板 ,可以用T2指定父类中的T类型
15  template<class T1, class T2>
16  class Son2 :public Base<T2>
17  {
18  public:
19      Son2()
20      {
21          cout << typeid(T1).name() << endl;
22          cout << typeid(T2).name() << endl;
23      }
24  };
25  void test02()
26  {
27      Son2<int, char> child1;
28  }
29  int main() {
30      test01();
31      test02();
32      system("pause");
33      return 0;
34  }

```

总结：如果父类是类模板，子类需要指定出父类中T的数据类型

### 1.3.6、类模板成员函数类外实现

学习目标：能够掌握类模板中成员函数类外实现

```

1  #include <string>
2  //类模板中成员函数类外实现
3  template<class T1, class T2>
4  class Person {
5  public:
6      //成员函数类内声明
7      Person(T1 name, T2 age);
8      void showPerson();
9  public:
10     T1 m_Name;
11     T2 m_Age;
12 };
13 //构造函数 类外实现
14 template<class T1, class T2>
15 Person<T1, T2>::Person(T1 name, T2 age) {
16     this->m_Name = name;
17     this->m_Age = age;
18 }
19 //成员函数 类外实现
20 template<class T1, class T2>
21 void Person<T1, T2>::showPerson() {
22     cout << "姓名: " << this->m_Name << " 年龄:" << this->m_Age << endl;

```

```

23 }
24 void test01()
25 {
26     Person<string, int> p("Tom", 20);
27     p.showPerson();
28 }
29 int main() {
30     test01();
31     system("pause");
32     return 0;
33 }

```

总结：类模板中成员函数类外实现时，需要加上模板参数列表

```

1  template<class T1, class T2>
2  Person<T1, T2>::Person(T1 name, T2 age)

```

### 1.3.7、类模板分文件编写

学习目标：掌握类模板成员函数分文件编写产生的问题及解决方式

问题：

- 类模板中成员函数创建时机是在调用阶段，导致分文件编写时链接不到

解决：

- 方案一：直接包含.cpp源文件
- 方案二：将声明和实现写到同一个文件中，并更改后缀名为.hpp, .hpp是约定的名称并不是强制

person.hpp中代码：

```

1  #pragma once
2  #include <iostream>
3  using namespace std;
4  #include <string>
5
6  template<class T1, class T2>
7  class Person {
8  public:
9      Person(T1 name, T2 age);
10     void showPerson();
11 public:
12     T1 m_Name;
13     T2 m_Age;
14 };
15
16 //构造函数 类外实现
17 template<class T1, class T2>
18 Person<T1, T2>::Person(T1 name, T2 age) {
19     this->m_Name = name;
20     this->m_Age = age;
21 }
22
23 //成员函数 类外实现
24 template<class T1, class T2>

```

```

25 void Person<T1, T2>::showPerson() {
26     cout << "姓名: " << this->m_Name << " 年龄:" << this->m_Age << endl;
27 }

```

类模板分文件编写.cpp中代码

一般不这样做，不会头文件包含源码

```

1  #include<iostream>
2  using namespace std;
3
4  // #include "person.h"
5  #include "person.cpp" //解决方式1，包含cpp源文件
6
7  //解决方式2，将声明和实现写到一起，文件后缀名改为.hpp
8  #include "person.hpp"
9  void test01()
10 {
11     Person<string, int> p("Tom", 10);
12     p.showPerson();
13 }
14
15 int main() {
16
17     test01();
18
19     system("pause");
20
21     return 0;
22 }

```

总结：主流的解决方式是第二种，将类模板成员函数写到一起，并将后缀名改为.hpp

### 1.3.8、类模板与友元

学习目标：掌握类模板配合友元函数的类内和类外实现

全局函数类内实现：直接在类内声明友元即可

全局函数类外实现：需要提前让编译器知道全局函数存在（类外实现麻烦很多）

全局函数类外实现，加空模板参数列表，如果全局函数是类外实现，需要让编译器提前知道这个函数的存在

```

1  #include <string>
2
3  //2、全局函数配合友元 类外实现 - 先做函数模板声明，下方在做函数模板定义，在做友元
4  template<class T1, class T2> class Person;
5
6  //如果声明了函数模板，可以将实现写到后面，否则需要将实现体写到类的前面让编译器提前看到
7  //template<class T1, class T2> void printPerson2(Person<T1, T2> & p);
8
9  template<class T1, class T2>
10 void printPerson2(Person<T1, T2> & p)
11 {

```

```

12     cout << "类外实现 ---- 姓名: " << p.m_Name << " 年龄: " << p.m_Age <<
    endl;
13 }
14 template<class T1, class T2>
15 class Person
16 {
17     //1、全局函数配合友元    类内实现
18     friend void printPerson(Person<T1, T2> & p)
19     {
20         cout << "姓名: " << p.m_Name << " 年龄: " << p.m_Age << endl;
21     }
22     //全局函数配合友元    类外实现
23     friend void printPerson2<>(Person<T1, T2> & p);
24 public:
25     Person(T1 name, T2 age)
26     {
27         this->m_Name = name;
28         this->m_Age = age;
29     }
30 private:
31     T1 m_Name;
32     T2 m_Age;
33
34 };
35 //1、全局函数在类内实现
36 void test01()
37 {
38     Person <string, int >p("Tom", 20);
39     printPerson(p);
40 }
41 //2、全局函数在类外实现

42 void test02()
43 {
44     Person <string, int >p("Jerry", 30);
45     printPerson2(p);
46 }
47 int main() {
48     //test01();
49     test02();
50     system("pause");
51     return 0;
52 }

```

总结：建议全局函数做类内实现，用法简单而且编译器可以直接识别

### 1.3.9、类模板案例

案例描述：实现一个通用的数组类，要求如下：

- 可以对内置数据类型以及自定义数据类型的数据进行存储
- 将数组中的数据存储到堆区
- 构造函数中可以传入数组的容量



- 提供对应的拷贝构造函数以及operator=防止浅拷贝问题
- 提供尾插法和尾删法对数组中的数据进行增加和删除
- 可以通过下标的方式访问数组中的元素
- 可以获取数组中当前元素个数和数组的容量

myArray.hpp中代码

```

1  #pragma once
2  #include <iostream>
3  using namespace std;
4
5  template<class T>
6  class MyArray
7  {
8  public:
9      //构造函数
10     MyArray(int capacity)
11     {
12         this->m_Capacity = capacity;
13         this->m_Size = 0;
14         pAddress = new T[this->m_Capacity];
15     }
16     //拷贝构造
17     MyArray(const MyArray & arr)
18     {
19         this->m_Capacity = arr.m_Capacity;
20         this->m_Size = arr.m_Size;
21         this->pAddress = new T[this->m_Capacity];
22         for (int i = 0; i < this->m_Size; i++)
23         {
24             //如果T为对象，而且还包含指针，必须需要重载 = 操作符，因为这个等号不是 构造
而是赋值，
25             // 普通类型可以直接= 但是指针类型需要深拷贝
26             this->pAddress[i] = arr.pAddress[i];
27         }
28     }
29     //重载= 操作符 防止浅拷贝问题
30     MyArray& operator=(const MyArray& myarray) {
31
32         if (this->pAddress != NULL) {
33             delete[] this->pAddress;
34             this->m_Capacity = 0;
35             this->m_Size = 0;
36         }
37         this->m_Capacity = myarray.m_Capacity;
38         this->m_Size = myarray.m_Size;
39         this->pAddress = new T[this->m_Capacity];
40         for (int i = 0; i < this->m_Size; i++) {
41             this->pAddress[i] = myarray[i];
42         }
43         return *this;
44     }
45     //重载[] 操作符 arr[0]
46     T& operator [](int index)
47     {

```

```

48         return this->pAddress[index]; //不考虑越界，用户自己去处理
49     }
50     //尾插法
51     void Push_back(const T & val)
52     {
53         if (this->m_Capacity == this->m_Size)
54         {
55             return;
56         }
57         this->pAddress[this->m_Size] = val;
58         this->m_Size++;
59     }
60     //尾删法
61     void Pop_back()
62     {
63         if (this->m_Size == 0)
64         {
65             return;
66         }
67         this->m_Size--;
68     }
69     //获取数组容量
70     int getCapacity()
71     {
72         return this->m_Capacity;
73     }
74     //获取数组大小
75     int getSize()
76     {
77         return this->m_Size;
78     }
79     //析构
80     ~MyArray()
81     {
82         if (this->pAddress != NULL)
83         {
84             delete[] this->pAddress;
85             this->pAddress = NULL;
86             this->m_Capacity = 0;
87             this->m_Size = 0;
88         }
89     }
90 private:
91     T * pAddress; //指向一个堆空间，这个空间存储真正的数据
92     int m_Capacity; //容量
93     int m_Size; //大小
94 };

```

类模板案例—数组类封装.cpp中

```

1  #include "myArray.hpp"
2  #include <string>
3  void printIntArray(MyArray<int>& arr) {
4      for (int i = 0; i < arr.getSize(); i++) {
5          cout << arr[i] << " ";

```

```

6     }
7     cout << endl;
8 }
9 //测试内置数据类型
10 void test01()
11 {
12     MyArray<int> array1(10);
13     for (int i = 0; i < 10; i++)
14     {
15         array1.Push_back(i);
16     }
17     cout << "array1打印输出: " << endl;
18     printIntArray(array1);
19     cout << "array1的大小: " << array1.getSize() << endl;
20     cout << "array1的容量: " << array1.getCapacity() << endl;
21     cout << "-----" << endl;
22     MyArray<int> array2(array1);
23     array2.Pop_back();
24     cout << "array2打印输出: " << endl;
25     printIntArray(array2);
26     cout << "array2的大小: " << array2.getSize() << endl;
27     cout << "array2的容量: " << array2.getCapacity() << endl;
28 }
29 //测试自定义数据类型
30 class Person {
31 public:
32     Person() {}
33     Person(string name, int age) {
34         this->m_Name = name;
35         this->m_Age = age;
36     }
37 public:
38     string m_Name;
39     int m_Age;
40 };
41 void printPersonArray(MyArray<Person>& personArr)
42 {
43     for (int i = 0; i < personArr.getSize(); i++) {
44         cout << "姓名: " << personArr[i].m_Name << " 年龄: " <<
personArr[i].m_Age << endl;
45     }
46 }
47 }
48 void test02()
49 {
50     //创建数组
51     MyArray<Person> pArray(10);
52     Person p1("孙悟空", 30);
53     Person p2("韩信", 20);
54     Person p3("妲己", 18);
55     Person p4("王昭君", 15);
56     Person p5("赵云", 24);
57     //插入数据
58     pArray.Push_back(p1);
59     pArray.Push_back(p2);

```

```

60     pArray.Push_back(p3);
61     pArray.Push_back(p4);
62     pArray.Push_back(p5);
63     printPersonArray(pArray);
64     cout << "pArray的大小: " << pArray.getSize() << endl;
65     cout << "pArray的容量: " << pArray.getCapacity() << endl;
66 }
67 int main() {
68     //test01();
69     test02();
70     system("pause");
71     return 0;
72 }

```

## 2、STL

### 2.1 STL的诞生

- 长久以来，软件界一直希望建立一种可重复利用的东西
- C++的**面向对象**和**泛型编程**思想，目的就是**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，分别是:**容器、算法、迭代器、仿函数、适配器（配接器）、空间配置器**

1. 容器：各种数据结构，如vector、list、deque、set、map等,用来存放数据。
2. 算法：各种常用的算法，如sort、find、copy、for\_each等
3. 迭代器：扮演了容器与算法之间的胶合剂。
4. 仿函数：行为类似函数，可作为算法的某种策略。
5. 适配器：一种用来修饰容器或者仿函数或迭代器接口的东西。
6. 空间配置器：负责空间的配置与管理。

### 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

这些容器分为**序列式容器**和**关联式容器**两种：

**序列式容器**:强调值的排序，序列式容器中的每个元素均有固定的位置。

**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，解决逻辑或数学上的问题，这一门学科我们叫做算法(Algorithms)

算法分为:**质变算法**和**非质变算法**。

质变算法：是指运算过程中会更改区间内的元素的内容。例如拷贝，替换，删除等等

非质变算法：是指运算过程中不会更改区间内的元素内容，例如查找、计数、遍历、寻找极值等等

**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，而又无需暴露该容器的内部表示方式。

**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，初学阶段我们可以先理解迭代器为指针

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

### 2.5.1、vector存放内置数据类型

容器：vector

算法：for\_each

迭代器：vector::iterator

//每一个容器都有自己的迭代器，迭代器是用来遍历容器中的元素

//v.begin()返回迭代器，这个迭代器指向容器中第一个数据

//v.end()返回迭代器，这个迭代器指向容器元素的**最后一个元素的下一个位置**

//vector::iterator 拿到vector这种容器的迭代器类型

```
1 #include <vector>
2 #include <algorithm>
3
4 void MyPrint(int val)
```

```

5  {
6      cout << val << endl;
7  }
8  void test01() {
9      //创建vector容器对象，并且通过模板参数指定容器中存放的数据的类型
10     vector<int> v;
11     //向容器中放数据
12     v.push_back(10);
13     v.push_back(20);
14     v.push_back(30);
15     v.push_back(40);
16     //每一个容器都有自己的迭代器，迭代器是用来遍历容器中的元素
17     //v.begin()返回迭代器，这个迭代器指向容器中第一个数据
18     //v.end()返回迭代器，这个迭代器指向容器元素的最后一个元素的下一个位置
19     //vector<int>::iterator 拿到vector<int>这种容器的迭代器类型
20     vector<int>::iterator pBegin = v.begin();
21     vector<int>::iterator pEnd = v.end();
22     //第一种遍历方式：
23     while (pBegin != pEnd) {
24         cout << *pBegin << endl;
25         pBegin++;
26     }
27     //第二种遍历方式：
28     for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
29         cout << *it << endl;
30     }
31     cout << endl;
32     //第三种遍历方式：
33     //使用STL提供标准遍历算法 头文件 algorithm
34     for_each(v.begin(), v.end(), MyPrint);
35 }
36 int main() {
37     test01();
38     system("pause")
39     return 0;
40 }

```

## 2.5.2、vector存放自定义数据类型

vector::iterator it=nums.begin();

it是地址

\*it的本质是vector::iterator中<>内的内容

for (vector<Person\*>::iterator it = v.begin(); it != v.end(); it++)

(\*\*it).m\_name

(\*it)->m\_name (一般用这个)

(\*(\*it)).m\_name

```

1  #include <vector>
2  #include <string>
3
4  //自定义数据类型

```

```

5  class Person {
6  public:
7      Person(string name, int age) {
8          mName = name;
9          mAge = age;
10     }
11 public:
12     string mName;
13     int mAge;
14 };
15 //存放对象
16 void test01() {
17     vector<Person> v;
18     //创建数据
19     Person p1("aaa", 10);
20     Person p2("bbb", 20);
21     Person p3("ccc", 30);
22     Person p4("ddd", 40);
23     Person p5("eee", 50);
24
25     v.push_back(p1);
26     v.push_back(p2);
27     v.push_back(p3);
28     v.push_back(p4);
29     v.push_back(p5);
30     for (vector<Person>::iterator it = v.begin(); it != v.end(); it++) {
31         cout << "Name:" << (*it).mName << " Age:" << (*it).mAge << endl;
32     }
33 }
34 }
35 //放对象指针
36 void test02() {
37     vector<Person*> v;
38     //创建数据
39     Person p1("aaa", 10);
40     Person p2("bbb", 20);
41     Person p3("ccc", 30);
42     Person p4("ddd", 40);
43     Person p5("eee", 50);
44
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     for (vector<Person*>::iterator it = v.begin(); it != v.end(); it++) {
51         Person * p = (*it);
52         cout << "Name:" << p->mName << " Age:" << (*it)->mAge << endl;
53     }
54 }
55 int main() {
56     test01();
57     test02();
58     system("pause");
59     return 0;

```

### 2.5.3、vector容器嵌套容器

```

1  #include <vector>
2
3  //容器嵌套容器
4  void test01() {
5      vector< vector<int> > v;
6      vector<int> v1;
7      vector<int> v2;
8      vector<int> v3;
9      vector<int> v4;
10     for (int i = 0; i < 4; i++) {
11         v1.push_back(i + 1);
12         v2.push_back(i + 2);
13         v3.push_back(i + 3);
14         v4.push_back(i + 4);
15     }
16     //将容器元素插入到vector v中
17     v.push_back(v1);
18     v.push_back(v2);
19     v.push_back(v3);
20     v.push_back(v4);
21     for (vector<vector<int>>::iterator it = v.begin(); it != v.end(); it++)
22     {
23         for (vector<int>::iterator vit = (*it).begin(); vit != (*it).end();
24             vit++) {
25             cout << *vit << " ";
26         }
27         cout << endl;
28     }
29     }
30     int main() {
31         test01();
32         system("pause");
33         return 0;
34     }

```

## 3、STL常用容器

### 3.1、string容器

#### 3.1.1、string基本概念

string是C++风格的字符串，本质上是一个类

string和char\*区别：

- char \*是一个指针
- string是一个类，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

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

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

### 3.1.2、string构造函数

构造函数原型：

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

```
1  #include <string>
2  //string构造
3  void test01()
4  {
5      string s1; //创建空字符串，调用无参构造函数
6      cout << "str1 = " << s1 << endl;
7      const char* str = "hello world";
8      string s2(str); //把c_string转换成了string
9      cout << "str2 = " << s2 << endl;
10     string s3(s2); //调用拷贝构造函数
11     cout << "str3 = " << s3 << endl;
12     string s4(10, 'a');
13     cout << "str3 = " << s3 << endl;
14 }
15 int main() {
16     test01();
17     system("pause");
18     return 0;
19 }
```

### 3.1.3、string赋值操作

给string字符串进行赋值

赋值的函数原型：

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

```
1  //赋值
2  void test01()
3  {
4      string str1;
5      str1 = "hello world";
6      cout << "str1 = " << str1 << endl;
```

```

7
8     string str2;
9     str2 = str1;
10    cout << "str2 = " << str2 << endl;
11
12    string str3;
13    str3 = 'a';
14    cout << "str3 = " << str3 << endl;
15
16    string str4;
17    str4.assign("hello c++");
18    cout << "str4 = " << str4 << endl;
19
20    string str5;
21    str5.assign("hello c++",5);
22    cout << "str5 = " << str5 << endl;
23
24    string str6;
25    str6.assign(str5);
26    cout << "str6 = " << str6 << endl;
27
28    string str7;
29    str7.assign(5, 'x');
30    cout << "str7 = " << str7 << endl;
31 }
32
33 int main() {
34     test01();
35     system("pause");
36     return 0;
37 }

```

总结:

string的赋值方式很多, `operator=` 这种方式是比较实用的

### 3.1.4、string字符串拼接

功能描述:

- 实现在字符串末尾拼接字符串

函数原型:

- `string& operator+=(const char* str);` //重载+=操作符
- `string& operator+=(const char c);` //重载+=操作符
- `string& operator+=(const string& str);` //重载+=操作符
- `string& append(const char *s);` //把字符串s连接到当前字符串结尾
- `string& append(const char *s, int n);` //把字符串s的前n个字符连接到当前字符串结尾
- `string& append(const string &s);` //同operator+=(const string& str)
- `string& append(const string &s, int pos, int n);` //字符串s中从pos开始的n个字符连接到字符串结尾

```

2 void test01()
3 {
4     string str1 = "我";
5     str1 += "爱玩游戏";
6     cout << "str1 = " << str1 << endl;
7     str1 += ':';
8     cout << "str1 = " << str1 << endl;
9     string str2 = "LOL DNF";
10    str1 += str2;
11    cout << "str1 = " << str1 << endl;
12    string str3 = "I";
13    str3.append(" love ");
14    str3.append("game abcde", 4);
15    //str3.append(str2);
16    str3.append(str2, 4, 3); // 从下标4位置开始，截取3个字符，拼接字符串末尾
17    cout << "str3 = " << str3 << endl;
18 }
19 int main() {
20     test01();
21     system("pause");
22     return 0;
23 }

```

### 3.1.5、string查找和替换

功能描述：

- 查找：查找指定字符串是否存在
- 替换：在指定的位置替换字符串

函数原型：

- `int find(const string& str, int pos = 0) const;` //查找str第一次出现位置,从pos开始查找
- `int find(const char* s, int pos = 0) const;` //查找s第一次出现位置,从pos开始查找
- `int find(const char* s, int pos, int n) const;` //从pos位置查找s的前n个字符第一次位置
- `int find(const char c, int pos = 0) const;` //查找字符c第一次出现位置
- `int rfind(const string& str, int pos = npos) const;` //查找str最后一次位置,从pos开始查找
- `int rfind(const char* s, int pos = npos) const;` //查找s最后一次出现位置,从pos开始查找
- `int rfind(const char* s, int pos, int n) const;` //从pos查找s的前n个字符最后一次位置
- `int rfind(const char c, int pos = 0) const;` //查找字符c最后一次出现位置
- `string& replace(int pos, int n, const string& str);` //替换从pos开始n个字符为字符串str
- `string& replace(int pos, int n, const char* s);` //替换从pos开始的n个字符为字符串s

```

1 //查找和替换
2 void test01()

```

```

3  {
4      //查找
5      string str1 = "abcdefgde";
6      int pos = str1.find("de");
7      if (pos == -1)
8      {
9          cout << "未找到" << endl;
10     }
11     else
12     {
13         cout << "pos = " << pos << endl;
14     }
15     pos = str1.rfind("de");
16     cout << "pos = " << pos << endl;
17 }
18 void test02()
19 {
20     //替换
21     string str1 = "abcdefgde";
22     //从1号位置起3个字符替换为"1111"
23     str1.replace(1, 3, "1111");
24     //a1111efgde
25     cout << "str1 = " << str1 << endl;
26 }
27 int main() {
28     //test01();
29     //test02();
30     system("pause");
31     return 0;
32 }

```

总结:

- find查找是从左往后, rfind从右往左
- find找到字符串后返回查找的第一个字符位置, 找不到返回-1
- replace在替换时, 要指定从哪个位置起, 多少个字符, 替换成什么样的字符串

### 3.1.6、string字符串比较

功能描述: 字符串之间的比较

比较方式: 字符串比较根据字符的ASCII码进行对比

=返回0

>返回1

<返回-1

函数原型:

- `int compare(const string &s) const;` //与字符串s比较
- `int compare(const char *s) const;` //与字符串s比较

```

1  //字符串比较
2  void test01()
3  {

```

```

4     string s1 = "hello";
5     string s2 = "aello";
6     int ret = s1.compare(s2);
7     if (ret == 0) {
8         cout << "s1 等于 s2" << endl;
9     }
10    else if (ret > 0)
11    {
12        cout << "s1 大于 s2" << endl;
13    }
14    else
15    {
16        cout << "s1 小于 s2" << endl;
17    }
18 }
19 int main() {
20     test01();
21     system("pause");
22     return 0;
23 }

```

总结：字符串对比主要是用于比较两个字符串是否相等，判断谁大谁小意义并不大

### 3.1.7、字符存取

string中单个字符获取方式有两种

- `char& operator[](int n);` //通过[]方式取字符
- `char& at(int n);` //通过at方法获取字符

```

1 void test01()
2 {
3     string str = "hello world";
4     for (int i = 0; i < str.size(); i++)
5     {
6         cout << str[i] << " ";
7     }
8     cout << endl;
9     for (int i = 0; i < str.size(); i++)
10    {
11        cout << str.at(i) << " ";
12    }
13    cout << endl;
14    //字符修改
15    str[0] = 'x';
16    str.at(1) = 'x';
17    cout << str << endl;
18 }
19 int main() {
20     test01();
21     system("pause");
22     return 0;
23 }

```

总结：string字符串中单个字符存取有两种方式，利用[]或at

### 3.1.8、string插入和删除

功能描述：对string字符串进行插入和删除字符操作

append和+一样都是直接加到结尾的

函数原型：

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

```
1 //字符串插入和删除
2 void test01()
3 {
4     string str = "hello";
5     str.insert(1, "111");
6     cout << str << endl;
7     str.erase(1, 3); //从1号位置开始3个字符
8     cout << str << endl;
9 }
10 int main() {
11     test01();
12     system("pause");
13     return 0;
14 }
```

插入和删除的起始下标都是从0开始

### 3.1.9、string子串

功能：从字符串中获取想要的子串

函数原型：

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

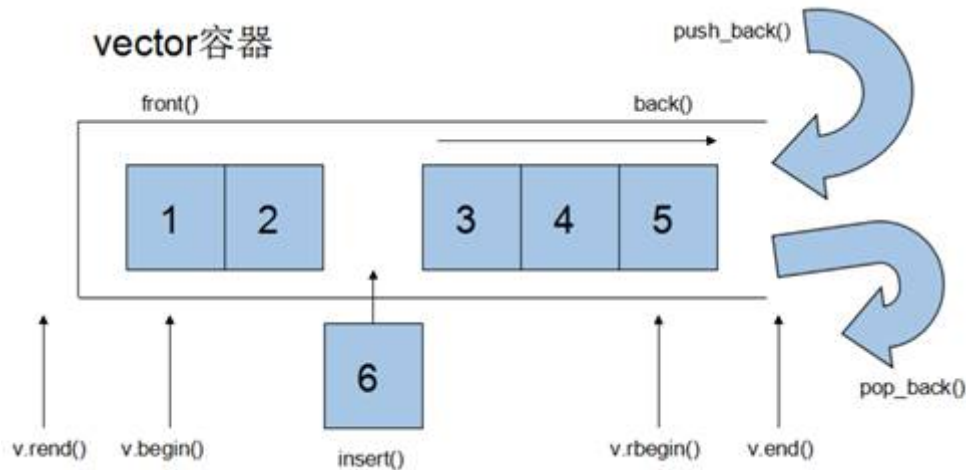
```
1 //子串
2 void test01()
3 {
4     string str = "abcdefg";
5     string subStr = str.substr(1, 3);
6     cout << "subStr = " << subStr << endl;
7     string email = "hello@sina.com";
8     int pos = email.find("@");
9     string username = email.substr(0, pos);
10    cout << "username: " << username << endl;
11 }
12 int main() {
13     test01();
14     system("pause");
15     return 0;
16 }
```

## 3.2、vector容器

### 3.2.1、vector基本概念

vector与普通函数区别：数组是静态空间，vector可以动态扩展

动态扩展：并不是在原空间之后续接新空间，而是找更大的内存空间，然后将原数据拷贝新空间，释放原空间



vector容器的迭代器支持随机访问的迭代器

### 3.2.2、vector构造函数

函数原型：

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

```
1  #include <vector>
2
3  void printVector(vector<int>& v) {
4
5      for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
6          cout << *it << " ";
7      }
8      cout << endl;
9  }
10 void test01()
11 {
12     vector<int> v1; //无参构造
13     for (int i = 0; i < 10; i++)
14     {
15         v1.push_back(i);
16     }
17     printVector(v1);
18     vector<int> v2(v1.begin(), v1.end());
19     printVector(v2);
20     vector<int> v3(10, 100);
```

```

21     printVector(v3);
22     vector<int> v4(v3);
23     printVector(v4);
24 }
25 int main() {
26     test01();
27     system("pause");
28     return 0;
29 }

```

### 3.2.3、vector赋值操作

功能描述：给vector容器进行赋值

函数原型：

- `vector& operator=(const vector &vec);` //重载等号操作符
- `assign(begin, end);` //将[beg, end)区间中的数据拷贝赋值给本身。（string不支持）
- `assign(n, elem);` //将n个elem拷贝赋值给本身。

```

1  #include <vector>
2
3  void printVector(vector<int>& v) {
4      for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
5          cout << *it << " ";
6      }
7      cout << endl;
8  }
9  //赋值操作
10 void test01()
11 {
12     vector<int> v1; //无参构造
13     for (int i = 0; i < 10; i++)
14     {
15         v1.push_back(i);
16     }
17     printVector(v1);
18     vector<int>v2;
19     v2 = v1;
20     printVector(v2);
21     vector<int>v3;
22     v3.assign(v1.begin(), v1.end());
23     printVector(v3);
24     vector<int>v4;
25     v4.assign(10, 100);
26     printVector(v4);
27 }
28 int main() {
29     test01();
30     system("pause");
31     return 0;
32 }

```



### 3.2.4、vector容量和大小

功能描述：对vector容器的容量和大小操作

函数原型：

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

```
1  #include <vector>
2
3  void printVector(vector<int>& v) {
4      for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
5          cout << *it << " ";
6      }
7      cout << endl;
8  }
9  void test01()
10 {
11     vector<int> v1;
12     for (int i = 0; i < 10; i++)
13     {
14         v1.push_back(i);
15     }
16     printVector(v1);
17     if (v1.empty())
18     {
19         cout << "v1为空" << endl;
20     }
21     else
22     {
23         cout << "v1不为空" << endl;
24         cout << "v1的容量 = " << v1.capacity() << endl;
25         cout << "v1的大小 = " << v1.size() << endl;
26     }
27     //resize 重新指定大小，若指定的更大，默认用0填充新位置，可以利用重载版本替换默认填充
28     v1.resize(15,10);
29     printVector(v1);
30     //resize 重新指定大小，若指定的更小，超出部分元素被删除
31     v1.resize(5);
32     printVector(v1);
33 }
34 int main() {
35     test01();
36     system("pause");
37     return 0;
38 }
```

总结:

- 判断是否为空 --- empty
- 返回元素个数 --- size
- 返回容器容量 --- capacity
- 重新指定大小 --- resize

### 3.2.5、vector插入和删除

功能描述: 对vector容器进行插入、删除操作

函数原型:

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

```
1
2 #include <vector>
3
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 //插入和删除
11 void test01()
12 {
13     vector<int> v1;
14     //尾插
15     v1.push_back(10);
16     v1.push_back(20);
17     v1.push_back(30);
18     v1.push_back(40);
19     v1.push_back(50);
20     printVector(v1);
21     //尾删
22     v1.pop_back();
23     printVector(v1);
24     //插入
25     v1.insert(v1.begin(), 100);
26     printVector(v1);
27     v1.insert(v1.begin(), 2, 1000);
28     printVector(v1);
29     //删除
30     v1.erase(v1.begin());
31     printVector(v1);
32     //清空
33     v1.erase(v1.begin(), v1.end());
```

```

34     v1.clear();
35     printVector(v1);
36 }
37 int main() {
38     test01();
39     system("pause");
40     return 0;
41 }

```

总结:

- 尾插 --- push\_back
- 尾删 --- pop\_back
- 插入 --- insert (位置迭代器)
- 删除 --- erase (位置迭代器)
- 清空 --- clear

### 3.2.6、vector数据存取

功能描述: 对vector中的数据的存取操作

函数原型:

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

```

1  #include <vector>
2
3  void test01()
4  {
5      vector<int>v1;
6      for (int i = 0; i < 10; i++)
7      {
8          v1.push_back(i);
9      }
10     for (int i = 0; i < v1.size(); i++)
11     {
12         cout << v1[i] << " ";
13     }
14     cout << endl;
15     for (int i = 0; i < v1.size(); i++)
16     {
17         cout << v1.at(i) << " ";
18     }
19     cout << endl;
20     cout << "v1的第一个元素为: " << v1.front() << endl;
21     cout << "v1的最后一个元素为: " << v1.back() << endl;
22 }
23 int main() {
24     test01();
25     system("pause");
26     return 0;
27 }

```

总结:

- 除了用迭代器获取vector容器中元素, [ ]和at也可以
- front返回容器第一个元素
- back返回容器最后一个元素

### 3.2.7、vector互换容器

功能描述: 实现两个容器内元素进行互换

函数原型: swap(vec); //将vec与本身的元素互换

resize重新指定大小, 但是不能改变容量

**vector(v).swap(v); 收缩内存**

vector(v)匿名对象

.swap(v)容器交换

匿名对象当前行执行完会被系统回收, 不用担心浪费空间

```
1  #include <vector>
2
3  void printVector(vector<int>& v) {
4      for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
5          cout << *it << " ";
6      }
7      cout << endl;
8  }
9  void test01()
10 {
11     vector<int>v1;
12     for (int i = 0; i < 10; i++)
13     {
14         v1.push_back(i);
15     }
16     printVector(v1);
17     vector<int>v2;
18     for (int i = 10; i > 0; i--)
19     {
20         v2.push_back(i);
21     }
22     printVector(v2);
23     //互换容器
24     cout << "互换后" << endl;
25     v1.swap(v2);
26     printVector(v1);
27     printVector(v2);
28 }
29 void test02()
30 {
31     vector<int> v;
32     for (int i = 0; i < 100000; i++) {
33         v.push_back(i);
34     }
35     cout << "v的容量为: " << v.capacity() << endl;
```

```

36     cout << "v的大小为: " << v.size() << endl;
37     v.resize(3);
38     cout << "v的容量为: " << v.capacity() << endl;
39     cout << "v的大小为: " << v.size() << endl;
40     //收缩内存
41     vector<int>(v).swap(v); //匿名对象
42     cout << "v的容量为: " << v.capacity() << endl;
43     cout << "v的大小为: " << v.size() << endl;
44 }
45 int main() {
46     test01();
47     test02();
48     system("pause");
49     return 0;
50 }

```

总结: swap可以使两个容器互换, 可以达到实用的收缩内存效果

### 3.2.8、vector预留空间

功能描述: 减少vector在动态扩展容量时的扩展次数

函数原型:

- `reserve(int len);` //容器预留len个元素长度, 预留位置不初始化, 元素不可访问。

```

1  #include <vector>
2
3  void test01()
4  {
5      vector<int> v;
6      //预留空间
7      v.reserve(100000);
8      int num = 0;
9      int* p = NULL;
10     for (int i = 0; i < 100000; i++) {
11         v.push_back(i);
12         if (p != &v[0]) {
13             p = &v[0];
14             num++;
15         }
16     }
17     cout << "num:" << num << endl;
18 }
19 int main() {
20     test01();
21     system("pause");
22     return 0;
23 }

```

总结: 数据量较大, 可以一开始利用reserve预留空间

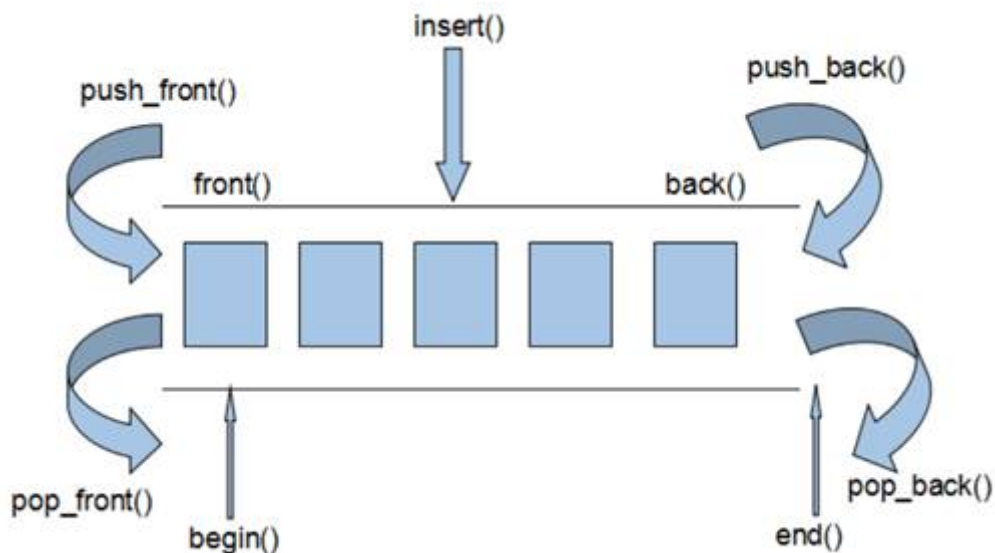
## 3.3、deque容器

### 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与vector区别：

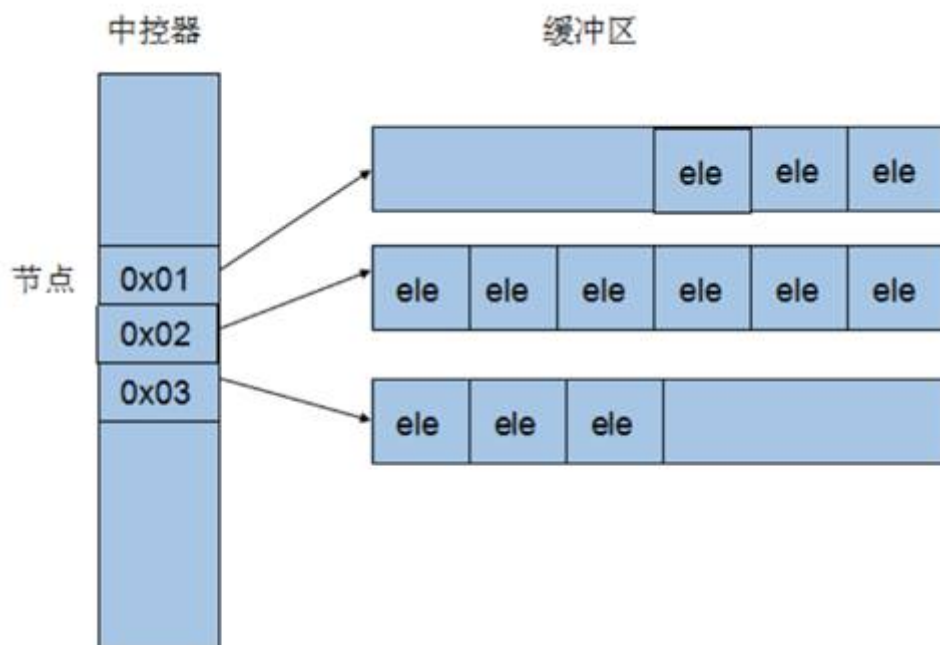
- vector对头部的插入删除效率低，数据量越大，效率越低
- deque相对而言，对头部的插入删除速度比vector快
- vector访问元素时的速度会比deque快，这和两者内部实现有关



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区的地址，使得使用deque时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

### 3.2.2、deque构造函数

函数原型:

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

只读迭代器

```
1 void printDeque(const deque<int>& d)
2 {
3     for (deque<int>::const_iterator it = d.begin(); it != d.end(); it++) {
4         cout << *it << " ";
5     }
6     cout << endl;
7 }
```

```
1 #include <deque>
2
3 void printDeque(const deque<int>& d)
4 {
5     for (deque<int>::const_iterator it = d.begin(); it != d.end(); it++) {
6         cout << *it << " ";
7     }
8     cout << endl;
9 }
10 //deque构造
11 void test01() {
12     deque<int> d1; //无参构造函数
13     for (int i = 0; i < 10; i++)
14     {
15         d1.push_back(i);
16     }
17     printDeque(d1);
18     deque<int> d2(d1.begin(), d1.end());
19     printDeque(d2);
20     deque<int> d3(10, 100);
21     printDeque(d3);
22     deque<int> d4 = d3;
23     printDeque(d4);
24 }
25 int main() {
26     test01();
27     system("pause");
28     return 0;
29 }
```

总结: deque容器和vector容器的构造方式几乎一致

### 3.3.3、deque赋值操作

函数原型：

- `deque& operator=(const deque &deq);` //重载等号操作符
- `assign(beg, end);` //将[beg, end)区间中的数据拷贝赋值给本身。
- `assign(n, elem);` //将n个elem拷贝赋值给本身。

```
1  #include <deque>
2
3  void printDeque(const deque<int>& d)
4  {
5      for (deque<int>::const_iterator it = d.begin(); it != d.end(); it++) {
6          cout << *it << " ";
7      }
8      cout << endl;
9  }
10 //赋值操作
11 void test01()
12 {
13     deque<int> d1;
14     for (int i = 0; i < 10; i++)
15     {
16         d1.push_back(i);
17     }
18     printDeque(d1);
19     deque<int> d2;
20     d2 = d1;
21     printDeque(d2);
22     deque<int> d3;
23     d3.assign(d1.begin(), d1.end());
24     printDeque(d3);
25     deque<int> d4;
26     d4.assign(10, 100);
27     printDeque(d4);
28 }
29 int main() {
30     test01();
31     system("pause");
32     return 0;
33 }
```

总结：deque赋值操作也与vector相同

### 3.3.4、deque大小操作

函数原型：

- `deque.empty();` //判断容器是否为空
- `deque.size();` //返回容器中元素的个数
- `deque.resize(num);` //重新指定容器的长度为num,若容器变长，则以默认值填充新位置。

//如果容器变短，则末尾超出容器长度的元素被删除。



- `deque.resize(num, elem);` //重新指定容器的长度为num,若容器变长,则以elem值填充新位置。

//如果容器变短,则末尾超出容器长度的元素被删除。

deque可以无限开辟新的缓冲区,中控器加一个地址维护新开辟的缓冲区空间,故deque没有容量的概念

```
1  #include <deque>
2
3  void printDeque(const deque<int>& d)
4  {
5      for (deque<int>::const_iterator it = d.begin(); it != d.end(); it++) {
6          cout << *it << " ";
7      }
8      cout << endl;
9  }
10 //大小操作
11 void test01()
12 {
13     deque<int> d1;
14     for (int i = 0; i < 10; i++)
15     {
16         d1.push_back(i);
17     }
18     printDeque(d1);
19     //判断容器是否为空
20     if (d1.empty()) {
21         cout << "d1为空!" << endl;
22     }
23     else {
24         cout << "d1不为空!" << endl;
25         //统计大小
26         cout << "d1的大小为: " << d1.size() << endl;
27     }
28     //重新指定大小
29     d1.resize(15, 1);
30     printDeque(d1);
31     d1.resize(5);
32     printDeque(d1);
33 }
34 int main() {
35     test01();
36     system("pause");
37     return 0;
38 }
```

总结:

- deque没有容量的概念
- 判断是否为空 --- empty
- 返回元素个数 --- size
- 重新指定个数 --- resize

### 3.3.5、deque插入和删除

函数原型：

两端插入操作：

- `push_back(elem);` //在容器尾部添加一个数据
- `push_front(elem);` //在容器头部插入一个数据
- `pop_back();` //删除容器最后一个数据
- `pop_front();` //删除容器第一个数据

指定位置操作：

pos是迭代器

- `insert(pos,elem);` //在pos位置插入一个elem元素的拷贝，返回新数据的位置。
- `insert(pos,n,elem);` //在pos位置插入n个elem数据，无返回值。
- `insert(pos,beg,end);` //在pos位置插入[beg,end)区间的数据，无返回值。
- `clear();` //清空容器的所有数据
- `erase(beg,end);` //删除[beg,end)区间的数据，返回下一个数据的位置。
- `erase(pos);` //删除pos位置的数据，返回下一个数据的位置。

```
1  #include <deque>
2
3  void printDeque(const deque<int>& d)
4  {
5      for (deque<int>::const_iterator it = d.begin(); it != d.end(); it++) {
6          cout << *it << " ";
7
8      }
9      cout << endl;
10 }
11 //两端操作
12 void test01()
13 {
14     deque<int> d;
15     //尾插
16     d.push_back(10);
17     d.push_back(20);
18     //头插
19     d.push_front(100);
20     d.push_front(200);
21
22     printDeque(d);
23
24     //尾删
25     d.pop_back();
26     //头删
27     d.pop_front();
28     printDeque(d);
29 }
30 //插入
31 void test02()
32 {
33     deque<int> d;
```

```

34     d.push_back(10);
35     d.push_back(20);
36     d.push_front(100);
37     d.push_front(200);
38     printDeque(d);
39
40     d.insert(d.begin(), 1000);
41     printDeque(d);
42
43     d.insert(d.begin(), 2, 10000);
44     printDeque(d);
45
46     deque<int> d2;
47     d2.push_back(1);
48     d2.push_back(2);
49     d2.push_back(3);
50
51     d.insert(d.begin(), d2.begin(), d2.end());
52     printDeque(d);
53 }
54 //删除
55 void test03()
56 {
57     deque<int> d;
58     d.push_back(10);
59     d.push_back(20);
60     d.push_front(100);
61     d.push_front(200);
62     printDeque(d);
63
64     d.erase(d.begin());
65     printDeque(d);
66
67     d.erase(d.begin(), d.end());
68     d.clear();
69     printDeque(d);
70 }
71 int main() {
72     //test01();
73     //test02();
74     test03();
75     system("pause");
76     return 0;
77 }

```

总结:

- 插入和删除提供的位置是迭代器
- 尾插 --- push\_back
- 尾删 --- pop\_back
- 头插 --- push\_front
- 头删 --- pop\_front

### 3.3.6、deque数据存取

功能描述：对deque中的数据的存取操作

函数原型：

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

`deq.back()`

迭代器存取`*(deq.end()-1)`

```
1  #include <deque>
2
3  void printDeque(const deque<int>& d)
4  {
5      for (deque<int>::const_iterator it = d.begin(); it != d.end(); it++) {
6          cout << *it << " ";
7
8      }
9      cout << endl;
10 }
11 //数据存取
12 void test01()
13 {
14     deque<int> d;
15     d.push_back(10);
16     d.push_back(20);
17     d.push_front(100);
18     d.push_front(200);
19
20     for (int i = 0; i < d.size(); i++) {
21         cout << d[i] << " ";
22     }
23     cout << endl;
24     for (int i = 0; i < d.size(); i++) {
25         cout << d.at(i) << " ";
26     }
27     cout << endl;
28     cout << "front:" << d.front() << endl;
29     cout << "back:" << d.back() << endl;
30 }
31 int main() {
32     test01();
33     system("pause");
34     return 0;
35 }
```

总结：

- 除了用迭代器获取deque容器中元素，`[]`和`at`也可以
- `front`返回容器第一个元素
- `back`返回容器最后一个元素

deque插入删除比vector快，但是访问元素速度比vector慢

### 3.3.7、deque排序

算法:

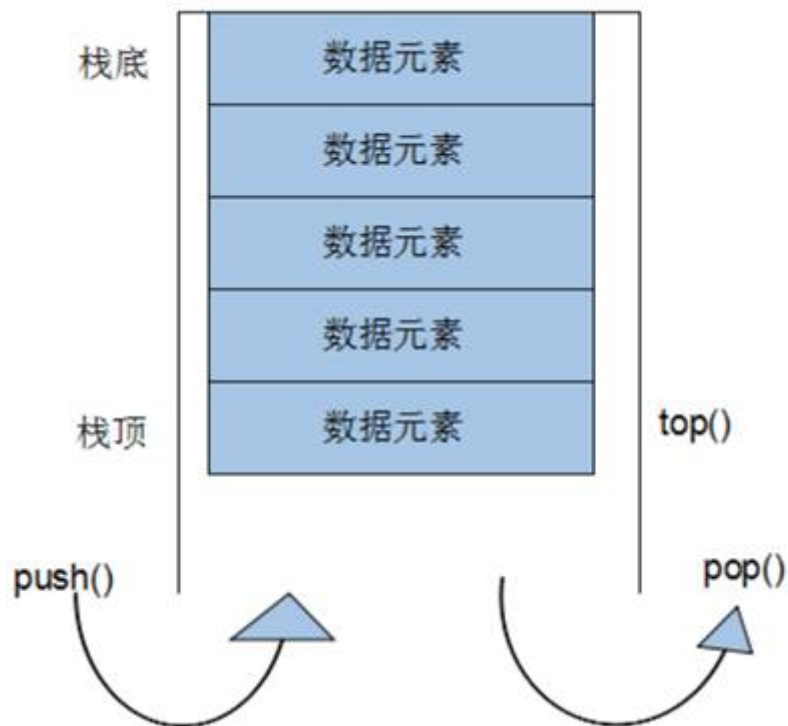
- `sort(iterator beg, iterator end)` //对beg和end区间内元素进行排序

```
1  #include <deque>
2  #include <algorithm>
3
4  void printDeque(const deque<int>& d)
5  {
6      for (deque<int>::const_iterator it = d.begin(); it != d.end(); it++) {
7          cout << *it << " ";
8      }
9      cout << endl;
10 }
11
12 void test01()
13 {
14
15     deque<int> d;
16     d.push_back(10);
17     d.push_back(20);
18     d.push_front(100);
19     d.push_front(200);
20
21     printDeque(d);
22     sort(d.begin(), d.end());
23     printDeque(d);
24
25 }
26 int main() {
27     test01();
28     system("pause");
29     return 0;
30 }
```

## 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, 它只有一个出口



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

功能描述：栈容器常用的对外接口

构造函数：

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

赋值操作：

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

数据存取：

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

大小操作：

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

示例：

```
1 #include <stack>
2
3 //栈容器常用接口
4 void test01()
5 {
```

```

6      //创建栈容器 栈容器必须符合先进后出
7      stack<int> s;
8      //向栈中添加元素,叫做 压栈 入栈
9      s.push(10);
10     s.push(20);
11     s.push(30);
12     while (!s.empty()) {
13         //输出栈顶元素
14         cout << "栈顶元素为: " << s.top() << endl;
15         //弹出栈顶元素
16         s.pop();
17     }
18     cout << "栈的大小为: " << s.size() << endl;
19 }
20 int main() {
21     test01()
22     system("pause");
23     return 0;
24 }

```

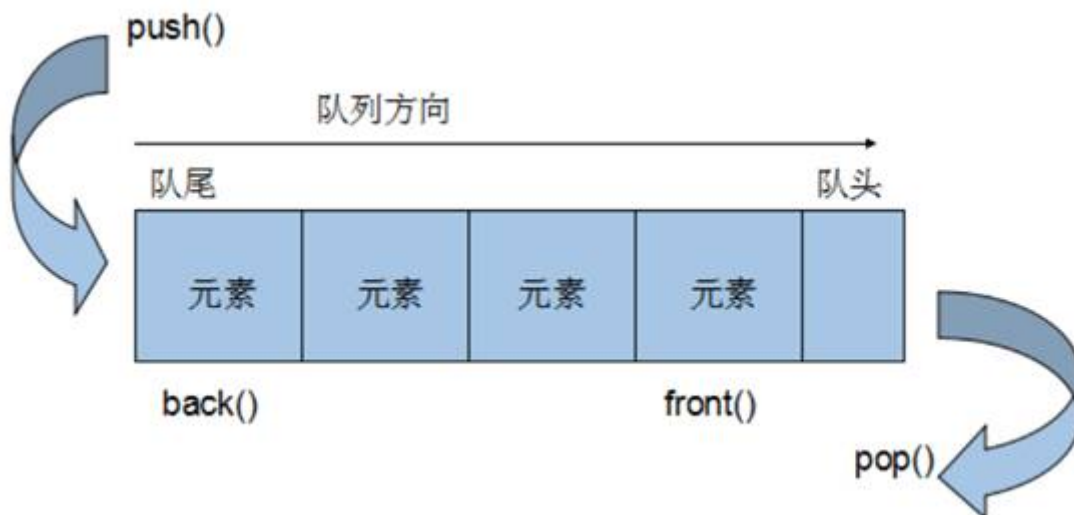
总结:

- 入栈 --- push
- 出栈 --- pop
- 返回栈顶 --- top
- 判断栈是否为空 --- empty
- 返回栈大小 --- size

## 3.6 queue 容器

### 3.6.1 queue 基本概念

**概念:** Queue是一种**先进先出**(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**

队列可以获取双端元素的值

队列中只有队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

功能描述：栈容器常用的对外接口

构造函数：

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

赋值操作：

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

数据存取：

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

大小操作：

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

示例：

```
1  #include <queue>
2  #include <string>
3  class Person
4  {
5  public:
6      Person(string name, int age)
7      {
8          this->m_Name = name;
9          this->m_Age = age;
10     }
11     string m_Name;
12     int m_Age;
13 };
14
15 void test01() {
16     //创建队列
17     queue<Person> q;
18     //准备数据
19     Person p1("唐僧", 30);
20     Person p2("孙悟空", 1000);
21     Person p3("猪八戒", 900);
22     Person p4("沙僧", 800);
23     //向队列中添加元素 入队操作
24     q.push(p1);
25     q.push(p2);
26     q.push(p3);
27     q.push(p4);
28     //队列不提供迭代器，更不支持随机访问
```



```

29     while (!q.empty()) {
30         //输出队头元素
31         cout << "队头元素-- 姓名: " << q.front().m_Name
32             << " 年龄: " << q.front().m_Age << endl;
33         cout << "队尾元素-- 姓名: " << q.back().m_Name
34             << " 年龄: " << q.back().m_Age << endl;
35         cout << endl;
36         //弹出队头元素
37         q.pop();
38     }
39     cout << "队列大小为: " << q.size() << endl;
40 }
41 int main() {
42     test01();
43     system("pause");
44     return 0;
45 }

```

总结:

- 入队 --- push
- 出队 --- pop
- 返回队头元素 --- front
- 返回队尾元素 --- back
- 判断队是否为空 --- empty
- 返回队列大小 --- size

## 3.7 list容器

### 3.7.1 list基本概念

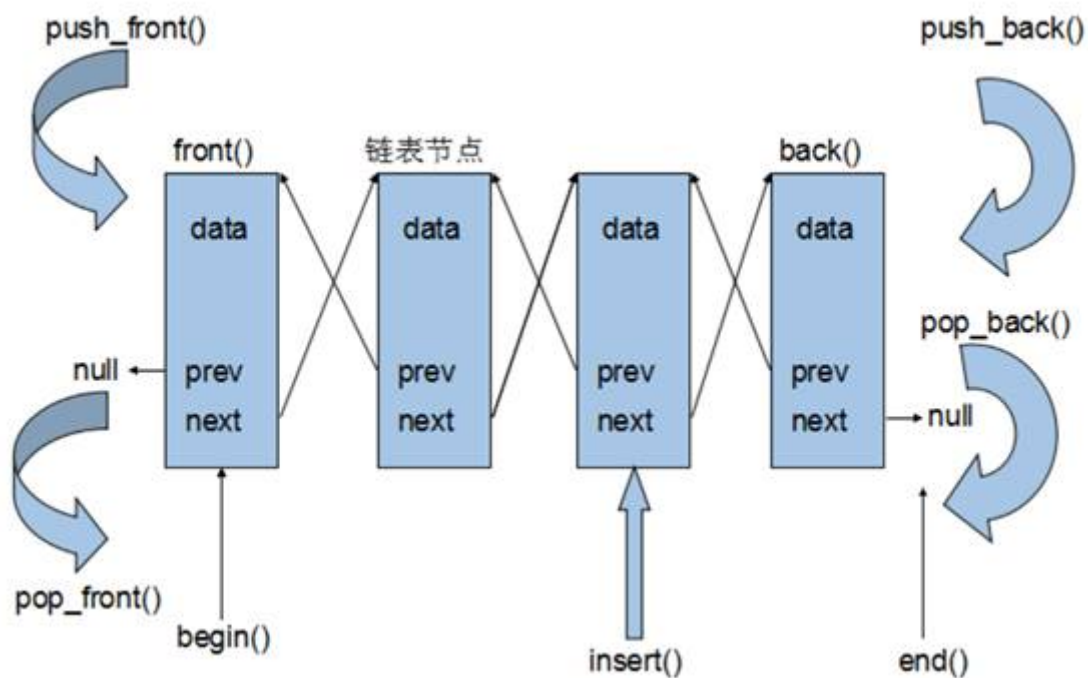
**功能:** 将数据进行链式存储

**链表** (list) 是一种物理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中的指针链接实现的

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元素的**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中的迭代器只支持前移和后移，不可以随机访问属于双向迭代器

list的优点：

- 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十分方便，修改指针即可，不需要移动大量元素

list的缺点：

- 链表灵活，但是空间(指针域) 和时间（遍历）额外耗费较大，因为要额外存储指针
- 遍历速度没有数组快

List有一个重要的性质，插入操作和删除操作都不会造成原有list迭代器的失效，这在vector是不成立的。

总结：STL中List和vector是两个最常被使用的容器，各有优缺点

### 3.7.2 list构造函数

功能描述：

- 创建list容器

函数原型：

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

示例：

```
1 #include <list>
2
3 void printList(const list<int>& L) {
4     for (list<int>::const_iterator it = L.begin(); it != L.end(); it++) {
5         cout << *it << " ";
6     }
```

```

7     cout << endl;
8 }
9 void test01()
10 {
11     list<int>L1;
12     L1.push_back(10);
13     L1.push_back(20);
14     L1.push_back(30);
15     L1.push_back(40);
16     printList(L1);
17     list<int>L2(L1.begin(),L1.end());
18     printList(L2);
19     list<int>L3(L2);
20     printList(L3);
21     list<int>L4(10, 1000);
22     printList(L4);
23 }
24 int main() {
25     test01();
26     system("pause");
27     return 0;
28 }

```

总结：list构造方式同其他几个STL常用容器，熟练掌握即可

### 3.7.3 list 赋值和交换

功能描述：

- 给list容器进行赋值，以及交换list容器

函数原型：

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

示例：

```

1  #include <list>
2
3  void printList(const list<int>& L) {
4      for (list<int>::const_iterator it = L.begin(); it != L.end(); it++) {
5          cout << *it << " ";
6      }
7      cout << endl;
8  }
9  //赋值和交换
10 void test01()
11 {
12     list<int>L1;
13     L1.push_back(10);
14     L1.push_back(20);
15     L1.push_back(30);
16     L1.push_back(40);

```

```

17     printList(L1);
18
19     //赋值
20     list<int>L2;
21     L2 = L1;
22     printList(L2);
23
24     list<int>L3;
25     L3.assign(L2.begin(), L2.end());
26     printList(L3);
27
28     list<int>L4;
29     L4.assign(10, 100);
30     printList(L4);
31 }
32 //交换
33 void test02()
34 {
35     list<int>L1;
36     L1.push_back(10);
37     L1.push_back(20);
38     L1.push_back(30);
39     L1.push_back(40);
40
41     list<int>L2;
42     L2.assign(10, 100);
43
44     cout << "交换前: " << endl;
45     printList(L1);
46     printList(L2);
47
48     cout << endl;
49
50     L1.swap(L2);
51
52     cout << "交换后: " << endl;
53     printList(L1);
54     printList(L2);
55 }
56
57 int main() {
58     //test01();
59     test02();
60     system("pause");
61     return 0;
62 }

```

总结：list赋值和交换操作能够灵活运用即可

### 3.7.4 list 大小操作

功能描述：

- 对list容器的大小进行操作

函数原型：

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

#### 示例:

```

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

```

#### 总结:

- 判断是否为空 --- empty
- 返回元素个数 --- size
- 重新指定个数 --- resize

### 3.7.5 list 插入和删除

#### 功能描述:

- 对list容器进行数据的插入和删除

#### 函数原型:

- push\_back(elem); //在容器尾部加入一个元素
- pop\_back(); //删除容器中最后一个元素
- push\_front(elem); //在容器开头插入一个元素
- pop\_front(); //从容器开头移除第一个元素
- insert(pos,elem); //在pos位置插elem元素的拷贝, 返回新数据的位置。
- insert(pos,n,elem); //在pos位置插入n个elem数据, 无返回值。
- insert(pos,beg,end); //在pos位置插入[beg,end)区间的数据, 无返回值。
- clear(); //移除容器的所有数据
- erase(beg,end); //删除[beg,end)区间的数据, 返回下一个数据的位置。
- erase(pos); //删除pos位置的数据, 返回下一个数据的位置。
- remove(elem); //删除容器中所有与elem值匹配的元素。

#### 示例:

```
1  #include <list>
2
3  void printList(const list<int>& L) {
4
5      for (list<int>::const_iterator it = L.begin(); it != L.end(); it++) {
6          cout << *it << " ";
7      }
8      cout << endl;
9  }
10 //插入和删除
11 void test01()
12 {
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     printList(L);
23     //尾删
24     L.pop_back();
25     printList(L);
26     //头删
27     L.pop_front();
28     printList(L);
29     //插入
30     list<int>::iterator it = L.begin();
31     L.insert(++it, 1000);
32     printList(L);
33     //删除
```

```

34     it = L.begin();
35     L.erase(++it);
36     printList(L);
37     //移除
38     L.push_back(10000);
39     L.push_back(10000);
40     L.push_back(10000);
41     printList(L);
42     L.remove(10000);
43     printList(L);
44     //清空
45     L.clear();
46     printList(L);
47 }
48
49 int main() {
50     test01();
51     system("pause");
52     return 0;
53 }

```

总结:

- 尾插 --- push\_back
- 尾删 --- pop\_back
- 头插 --- push\_front
- 头删 --- pop\_front
- 插入 --- insert
- 删除 --- erase
- 移除 --- remove
- 清空 --- clear

### 3.7.6 list 数据存取

功能描述:

- 对list容器中数据进行存取

函数原型:

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

示例:

```

1  #include <list>
2
3  //数据存取
4  void test01()
5  {
6      list<int>L1;
7      L1.push_back(10);
8      L1.push_back(20);
9      L1.push_back(30);
10     L1.push_back(40);
11     //cout << L1.at(0) << endl; //错误 不支持at访问数据

```

```

12 //cout << L1[0] << endl; //错误 不支持[]方式访问数据
13 cout << "第一个元素为: " << L1.front() << endl;
14 cout << "最后一个元素为: " << L1.back() << endl;
15 //list容器的迭代器是双向迭代器, 不支持随机访问
16 list<int>::iterator it = L1.begin();
17 //it = it + 1; //错误, 不可以跳跃访问, 即使是+1
18 }
19 int main() {
20     test01();
21     system("pause");
22     return 0;
23 }

```

总结:

- list容器中不可以通过[]或者at方式访问数据
- 返回第一个元素 --- front
- 返回最后一个元素 --- back

### 3.7.7 list 反转和排序

功能描述:

- 将容器中的元素反转, 以及将容器中的数据进行排序

函数原型:

- reverse(); //反转链表
- sort(); //链表排序

示例:

指定规则排序

```

1 bool myCompare(int val1 , int val2)
2 {
3     return val1 > val2;
4 }
5 L.sort(myCompare); //指定规则, 从大到小

```

```

1 void printList(const list<int>& L) {
2
3     for (list<int>::const_iterator it = L.begin(); it != L.end(); it++) {
4         cout << *it << " ";
5     }
6     cout << endl;
7 }
8 bool myCompare(int val1 , int val2)
9 {
10     return val1 > val2;
11 }
12 //反转和排序
13 void test01()
14 {
15     list<int> L;
16     L.push_back(90);

```



```

17     L.push_back(30);
18     L.push_back(20);
19     L.push_back(70);
20     printList(L);
21
22     //反转容器的元素
23     L.reverse();
24     printList(L);
25
26     //排序
27     L.sort(); //默认的排序规则 从小到大
28     printList(L);
29
30     L.sort(myCompare); //指定规则，从大到小
31     printList(L);
32 }
33 int main() {
34     test01();
35     system("pause");
36     return 0;
37 }

```

总结：

- 反转 --- reverse
- 排序 --- sort (成员函数)

### 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降序

示例：

```

1  #include <list>
2  #include <string>
3  class Person {
4  public:
5      Person(string name, int age , int height) {
6          m_Name = name;
7          m_Age = age;
8          m_Height = height;
9      }
10 public:
11     string m_Name; //姓名
12     int m_Age; //年龄
13     int m_Height; //身高
14 };
15 bool ComparePerson(Person& p1, Person& p2) {
16     if (p1.m_Age == p2.m_Age) {
17         return p1.m_Height > p2.m_Height;
18     }
19     else
20     {
21         return p1.m_Age < p2.m_Age;

```

```

22     }
23
24 }
25 void test01() {
26     list<Person> L;
27
28     Person p1("刘备", 35, 175);
29     Person p2("曹操", 45, 180);
30     Person p3("孙权", 40, 170);
31     Person p4("赵云", 25, 190);
32     Person p5("张飞", 35, 160);
33     Person p6("关羽", 35, 200);
34
35     L.push_back(p1);
36     L.push_back(p2);
37     L.push_back(p3);
38     L.push_back(p4);
39     L.push_back(p5);
40     L.push_back(p6);
41
42     for (list<Person>::iterator it = L.begin(); it != L.end(); it++) {
43         cout << "姓名: " << it->m_Name << " 年龄: " << it->m_Age
44             << " 身高: " << it->m_Height << endl;
45     }
46     cout << "-----" << endl;
47     L.sort(ComparePerson); //排序
48     for (list<Person>::iterator it = L.begin(); it != L.end(); it++) {
49         cout << "姓名: " << it->m_Name << " 年龄: " << it->m_Age
50             << " 身高: " << it->m_Height << endl;
51     }
52 }
53
54 int main() {
55     test01();
56     system("pause");
57     return 0;
58 }

```

总结:

- 对于自定义数据类型，必须要指定排序规则，否则编译器不知道如何进行排序
- 高级排序只是在排序规则上再进行一次逻辑规则制定，并不复杂

## 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，底层构造是用红黑树实现

**序列式容器**:强调值的排序，序列式容器中的每个元素均有固定的位置。

**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

set和multiset区别：

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

### 3.8.2、set构造和赋值

构造：

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

赋值：

- `set& operator=(const set &st);` //重载等号操作符

```

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

总结：

- set容器插入数据时用insert
- set容器插入数据的数据会自动排序

set不支持头插/尾插，插入数据时只有insert

### 3.8.3 set大小和交换

#### 功能描述:

- 统计set容器大小以及交换set容器

#### 函数原型:

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

#### 示例:

```
1  #include <set>
2
3  void printSet(set<int> & s)
4  {
5      for (set<int>::iterator it = s.begin(); it != s.end(); it++)
6      {
7          cout << *it << " ";
8      }
9      cout << endl;
10 }
11 //大小
12 void test01()
13 {
14     set<int> s1;
15
16     s1.insert(10);
17     s1.insert(30);
18     s1.insert(20);
19     s1.insert(40);
20
21     if (s1.empty())
22     {
23         cout << "s1为空" << endl;
24     }
25     else
26     {
27         cout << "s1不为空" << endl;
28         cout << "s1的大小为: " << s1.size() << endl;
29     }
30 }
31 //交换
32 void test02()
33 {
34     set<int> s1;
35
36     s1.insert(10);
37     s1.insert(30);
38     s1.insert(20);
39     s1.insert(40);
40
41     set<int> s2;
42     s2.insert(100);
```

```

43     s2.insert(300);
44     s2.insert(200);
45     s2.insert(400);
46
47     cout << "交换前" << endl;
48     printSet(s1);
49     printSet(s2);
50     cout << endl;
51
52     cout << "交换后" << endl;
53     s1.swap(s2);
54     printSet(s1);
55     printSet(s2);
56 }
57 int main() {
58     //test01();
59     test02();
60     system("pause");
61     return 0;
62 }

```

总结:

- 统计大小 --- size
- 判断是否为空 --- empty
- 交换容器 --- swap

### 3.8.4 set插入和删除

功能描述:

- set容器进行插入数据和删除数据

函数原型:

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

示例:

```

1  #include <set>
2
3  void printSet(set<int> & s)
4  {
5      for (set<int>::iterator it = s.begin(); it != s.end(); it++)
6      {
7          cout << *it << " ";
8      }
9      cout << endl;
10 }
11 //插入和删除
12 void test01()
13 {

```

```

14     set<int> s1;
15     //插入
16     s1.insert(10);
17     s1.insert(30);
18     s1.insert(20);
19     s1.insert(40);
20     printSet(s1);
21
22     //删除
23     s1.erase(s1.begin());
24     printSet(s1);
25     s1.erase(30);
26     printSet(s1);
27     //清空
28     //s1.erase(s1.begin(), s1.end());
29     s1.clear();
30     printSet(s1);
31 }
32
33 int main() {
34     test01();
35     system("pause");
36     return 0;
37 }

```

总结:

- 插入 --- insert
- 删除 --- erase
- 清空 --- clear

### 3.8.5 set查找和统计

功能描述:

- 对set容器进行查找数据以及统计数据

函数原型:

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

示例:

```

1  #include <set>
2
3  //查找和统计
4  void test01()
5  {
6      set<int> s1;
7      //插入
8      s1.insert(10);
9      s1.insert(30);
10     s1.insert(20);
11     s1.insert(40);

```

```

12
13 //查找
14 set<int>::iterator pos = s1.find(30);
15 if (pos != s1.end())
16 {
17     cout << "找到了元素 : " << *pos << endl;
18 }
19 else
20 {
21     cout << "未找到元素" << endl;
22 }
23 //统计
24 int num = s1.count(30);
25 cout << "num = " << num << endl;
26 }
27 int main() {
28     test01();
29     system("pause");
30     return 0;
31 }

```

总结:

- 查找 --- find (返回的是迭代器)
- 统计 --- count (对于set, 结果为0或者1)

### 3.8.6 set和multiset区别

学习目标:

- 掌握set和multiset的区别

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pair<set::iterator, bool> ret = s.insert(10);

示例:

```

1 #include <set>
2
3 //set和multiset区别
4 void test01()
5 {
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
22     multiset<int> ms;
23     ms.insert(10);
24     ms.insert(10);
25     for (multiset<int>::iterator it = ms.begin(); it != ms.end(); it++) {
26         cout << *it << " ";
27     }
28     cout << endl;
29 }
30 int main() {
31     test01();
32     system("pause");
33     return 0;
34 }

```

总结:

- 如果不允许插入重复数据可以利用set
- 如果需要插入重复数据利用multiset

### 3.8.7 pair对组创建

功能描述:

- 成对出现的数据, 利用对组可以返回两个数据

两种创建方式:

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

```
pair<string, int> p2 = make_pair("Jerry", 10);
```

示例:

```

1 unordered_map<string, int> hash;
2 hash.insert(pair<string, int>("hello world", 1));
3 cout << ( * hash.begin()).first << "    " << hash.begin()->second <<
endl;

```

```

1 #include <string>
2
3 //对组创建
4 void test01()
5 {
6     pair<string, int> p(string("Tom"), 20);
7     cout << "姓名: " << p.first << " 年龄: " << p.second << endl;
8
9     pair<string, int> p2 = make_pair("Jerry", 10);
10    cout << "姓名: " << p2.first << " 年龄: " << p2.second << endl;
11 }
12 int main() {

```



```

13     test01();
14     system("pause");
15     return 0;
16 }

```

总结：

两种方式都可以创建对组，记住一种即可

### 3.8.8 set容器排序

学习目标：

- set容器默认排序规则为从小到大，掌握如何改变排序规则

主要技术点：

- 利用仿函数，可以改变排序规则

**示例一** set存放内置数据类型

```

1  class MyCompare
2  {
3  public:
4      bool operator()(int v1, int v2) {
5          return v1 > v2;
6      }
7  };
8      set<int, MyCompare> s2;

```

```

1  #include <set>
2
3  class MyCompare
4  {
5  public:
6      bool operator()(int v1, int v2) {
7          return v1 > v2;
8      }
9  };
10 void test01()
11 {
12     set<int> s1;
13     s1.insert(10);
14     s1.insert(40);
15     s1.insert(20);
16     s1.insert(30);
17     s1.insert(50);
18
19     //默认从小到大
20     for (set<int>::iterator it = s1.begin(); it != s1.end(); it++) {
21         cout << *it << " ";
22     }
23     cout << endl;
24
25     //指定排序规则
26     set<int, MyCompare> s2;

```

```

27     s2.insert(10);
28     s2.insert(40);
29     s2.insert(20);
30     s2.insert(30);
31     s2.insert(50);
32
33     for (set<int, MyCompare>::iterator it = s2.begin(); it != s2.end();
it++) {
34         cout << *it << " ";
35     }
36     cout << endl;
37 }
38
39 int main() {
40     test01();
41     system("pause");
42     return 0;
43 }

```

总结：利用仿函数可以指定set容器的排序规则

## 示例二 set存放自定义数据类型

```

1  #include <set>
2  #include <string>
3
4  class Person
5  {
6  public:
7      Person(string name, int age)
8      {
9          this->m_Name = name;
10         this->m_Age = age;
11     }
12     string m_Name;
13     int m_Age;
14 };
15 class comparePerson
16 {
17 public:
18     bool operator()(const Person& p1, const Person &p2)
19     {
20         //按照年龄进行排序 降序
21         return p1.m_Age > p2.m_Age;
22     }
23 };
24 void test01()
25 {
26     set<Person, comparePerson> s;
27     Person p1("刘备", 23);
28     Person p2("关羽", 27);
29     Person p3("张飞", 25);
30     Person p4("赵云", 21);
31
32     s.insert(p1);

```

```

33     s.insert(p2);
34     s.insert(p3);
35     s.insert(p4);
36     s.insert(Person("Mike", 17));
37     for (set<Person, comparePerson>::iterator it = s.begin(); it != s.end();
it++)
38     {
39         cout << "姓名:  " << it->m_Name << " 年龄:  " << it->m_Age << endl;
40     }
41 }
42 int main() {
43     test01();
44     system("pause");
45     return 0;
46 }

```

总结:

对于自定义数据类型, set必须指定排序规则才可以插入数据

## 3.9 map/ multimap容器

### 3.9.1 map基本概念

简介:

- map中所有元素都是pair
- pair中第一个元素为key (键值), 起到索引作用, 第二个元素为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

功能描述:

- 对map容器进行构造和赋值操作

函数原型:

构造:

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

赋值:

- `map& operator=(const map &mp);` //重载等号操作符

示例:

```
1  #include <map>
2
3  void printMap(map<int,int>&m)
4  {
5      for (map<int, int>::iterator it = m.begin(); it != m.end(); it++)
6      {
7          cout << "key = " << it->first << " value = " << it->second << endl;
8      }
9      cout << endl;
10 }
11 void test01()
12 {
13     map<int,int>m; //默认构造
14     m.insert(pair<int, int>(1, 10));
15     m.insert(pair<int, int>(2, 20));
16     m.insert(pair<int, int>(3, 30));
17     printMap(m);
18
19     map<int, int>m2(m); //拷贝构造
20     printMap(m2);
21
22     map<int, int>m3;
23     m3 = m2; //赋值
24     printMap(m3);
25 }
26
27 int main() {
28     test01();
29     system("pause");
30     return 0;
31 }
```

总结: map中所有元素都是成对出现, 插入数据时候要使用对组

### 3.9.3 map大小和交换

功能描述:

- 统计map容器大小以及交换map容器

函数原型:

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

示例:

```
1  #include <map>
2
3  void printMap(map<int,int>&m)
4  {
5      for (map<int, int>::iterator it = m.begin(); it != m.end(); it++)
6      {
```

```

7         cout << "key = " << it->first << " value = " << it->second << endl;
8     }
9     cout << endl;
10 }
11
12 void test01()
13 {
14     map<int, int>m;
15     m.insert(pair<int, int>(1, 10));
16     m.insert(pair<int, int>(2, 20));
17     m.insert(pair<int, int>(3, 30));
18
19     if (m.empty())
20     {
21         cout << "m为空" << endl;
22     }
23     else
24     {
25         cout << "m不为空" << endl;
26         cout << "m的大小为: " << m.size() << endl;
27     }
28 }
29
30
31 //交换
32 void test02()
33 {
34     map<int, int>m;
35     m.insert(pair<int, int>(1, 10));
36     m.insert(pair<int, int>(2, 20));
37     m.insert(pair<int, int>(3, 30));
38
39     map<int, int>m2;
40     m2.insert(pair<int, int>(4, 100));
41     m2.insert(pair<int, int>(5, 200));
42     m2.insert(pair<int, int>(6, 300));
43
44     cout << "交换前" << endl;
45     printMap(m);
46     printMap(m2);
47
48     cout << "交换后" << endl;
49     m.swap(m2);
50     printMap(m);
51     printMap(m2);
52 }
53 int main() {
54     test01();
55     test02();
56     system("pause");
57     return 0;
58 }

```

总结:

- 统计大小 --- size

- 判断是否为空 --- empty
- 交换容器 --- swap

### 3.9.4 map插入和删除

功能描述:

- map容器进行插入数据和删除数据

函数原型:

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

```
m.insert(make_pair(2, 20));
```

示例:

```
1  #include <map>
2
3  void printMap(map<int, int>&m)
4  {
5      for (map<int, int>::iterator it = m.begin(); it != m.end(); it++)
6      {
7          cout << "key = " << it->first << " value = " << it->second << endl;
8      }
9      cout << endl;
10 }
11
12 void test01()
13 {
14     //插入
15     map<int, int> m;
16     //第一种插入方式
17     m.insert(pair<int, int>(1, 10));
18     //第二种插入方式
19     m.insert(make_pair(2, 20));
20     //第三种插入方式
21     m.insert(map<int, int>::value_type(3, 30));
22     //第四种插入方式
23     m[4] = 40;
24     printMap(m);
25
26     //删除
27     m.erase(m.begin());
28     printMap(m);
29
30     m.erase(3);
31     printMap(m);
32
33     //清空
34     m.erase(m.begin(), m.end());
35     m.clear();
```

```

36     printMap(m);
37 }
38
39 int main() {
40     test01();
41     system("pause");
42     return 0;
43 }

```

总结:

- map插入方式很多, 记住其一即可
- 插入 --- insert
- 删除 --- erase
- 清空 --- clear

### 3.9.5 map查找和统计

功能描述:

- 对map容器进行查找数据以及统计数据

函数原型:

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

示例:

```

1  #include <map>
2
3  //查找和统计
4  void test01()
5  {
6      map<int, int>m;
7      m.insert(pair<int, int>(1, 10));
8      m.insert(pair<int, int>(2, 20));
9      m.insert(pair<int, int>(3, 30));
10     //查找
11     map<int, int>::iterator pos = m.find(3);
12     if (pos != m.end())
13     {
14         cout << "找到了元素 key = " << (*pos).first << " value = " <<
(*pos).second << endl;
15     }
16     else
17     {
18         cout << "未找到元素" << endl;
19     }
20     //统计
21     int num = m.count(3);
22     cout << "num = " << num << endl;
23 }
24 int main() {
25     test01();

```

```

26     system("pause");
27     return 0;
28 }

```

总结:

- 查找 --- find (返回的是迭代器)
- 统计 --- count (对于map, 结果为0或者1)

### 3.9.6 map容器排序

学习目标:

- map容器默认排序规则为 按照key值进行 从小到大排序, 掌握如何改变排序规则

主要技术点:

- 利用仿函数, 可以改变排序规则

示例:

```

1  #include <map>
2
3  class MyCompare {
4  public:
5      bool operator()(int v1, int v2) {
6          return v1 > v2;
7      }
8  };
9
10 void test01()
11 {
12     //默认从小到大排序
13     //利用仿函数实现从大到小排序
14     map<int, int, MyCompare> m;
15
16     m.insert(make_pair(1, 10));
17     m.insert(make_pair(2, 20));
18     m.insert(make_pair(3, 30));
19     m.insert(make_pair(4, 40));
20     m.insert(make_pair(5, 50));
21
22     for (map<int, int, MyCompare>::iterator it = m.begin(); it != m.end();
it++) {
23         cout << "key:" << it->first << " value:" << it->second << endl;
24     }
25 }
26 int main() {
27     test01();
28     system("pause");
29     return 0;
30 }

```

总结:

- 利用仿函数可以指定map容器的排序规则
- 对于自定义数据类型, map必须要指定排序规则, 同set容器



## 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（ABCDEFGHJIJ），10名员工进入公司之后，需要指派员工在那个部门工作
- 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

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

案例代码：

```
1  #include<iostream>
2  using namespace std;
3  #include <vector>
4  #include <string>
5  #include <map>
6  #include <ctime>
7
8  /*
9   - 公司今天招聘了10个员工（ABCDEFGHJIJ），10名员工进入公司之后，需要指派员工在那个部门工
   作
10  - 员工信息有：姓名 工资组成；部门分为：策划、美术、研发
11  - 随机给10名员工分配部门和工资
12  - 通过multimap进行信息的插入 key(部门编号) value(员工)
13  - 分部门显示员工信息
14  */
15
16  #define CEHUA  0
17  #define MEISHU 1
18  #define YANFA  2
19
20  class worker
21  {
22  public:
23      string m_Name;
24      int m_Salary;
25  };
26
27  void createWorker(vector<worker>&v)
28  {
29      string nameSeed = "ABCDEFGHJIJ";
30      for (int i = 0; i < 10; i++)
31      {
32          worker worker;
33          worker.m_Name = "员工";
34          worker.m_Name += nameSeed[i];
```

```

35
36     worker.m_Salary = rand() % 10000 + 10000; // 10000 ~ 19999
37     //将员工放入到容器中
38     v.push_back(worker);
39 }
40 }
41
42 //员工分组
43 void setGroup(vector<Worker>&v,multimap<int,Worker>&m)
44 {
45     for (vector<Worker>::iterator it = v.begin(); it != v.end(); it++)
46     {
47         //产生随机部门编号
48         int deptId = rand() % 3; // 0 1 2
49
50         //将员工插入到分组中
51         //key部门编号, value具体员工
52         m.insert(make_pair(deptId, *it));
53     }
54 }
55
56 void showWorkerByGourp(multimap<int,Worker>&m)
57 {
58     // 0  A  B  C   1  D  E   2  F  G ...
59     cout << "策划部门: " << endl;
60
61     multimap<int,Worker>::iterator pos = m.find(CEHUA);
62     int count = m.count(CEHUA); // 统计具体人数
63     int index = 0;
64     for (; pos != m.end() && index < count; pos++, index++)
65     {
66         cout << "姓名: " << pos->second.m_Name << " 工资: " << pos-
67         >second.m_Salary << endl;
68     }
69
70     cout << "-----" << endl;
71     cout << "美术部门: " << endl;
72     pos = m.find(MEISHU);
73     count = m.count(MEISHU); // 统计具体人数
74     index = 0;
75     for (; pos != m.end() && index < count; pos++, index++)
76     {
77         cout << "姓名: " << pos->second.m_Name << " 工资: " << pos-
78         >second.m_Salary << endl;
79     }
80
81     cout << "-----" << endl;
82     cout << "研发部门: " << endl;
83     pos = m.find(YANFA);
84     count = m.count(YANFA); // 统计具体人数
85     index = 0;
86     for (; pos != m.end() && index < count; pos++, index++)
87     {
88         cout << "姓名: " << pos->second.m_Name << " 工资: " << pos-
89         >second.m_Salary << endl;

```

```

87     }
88
89 }
90
91 int main() {
92     srand((unsigned int)time(NULL));
93     //1、创建员工
94     vector<Worker>vWorker;
95     createWorker(vWorker);
96     //2、员工分组
97     multimap<int, Worker>mWorker;
98     setGroup(vWorker, mWorker);
99     //3、分组显示员工
100    showWorkerByGourp(mWorker);
101    //测试
102    //for (vector<Worker>::iterator it = vWorker.begin(); it !=
vWorker.end(); it++)
103        //{
104        //    cout << "姓名:  " << it->m_Name << " 工资:  " << it->m_Salary <<
endl;
105        //}
106    system("pause");
107    return 0;
108 }

```

总结:

- 当数据以键值对形式存在, 可以考虑用map 或 multimap

## 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, 其对象常称为**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 <string>
2
3  //1、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
4  class MyAdd
5  {
6  public :

```

```

7     int operator()(int v1,int v2)
8     {
9         return v1 + v2;
10    }
11 };
12 void test01()
13 {
14     MyAdd myAdd;
15     cout << myAdd(10, 10) << endl;
16 }
17 //2、函数对象可以有自己的状态
18 class MyPrint
19 {
20 public:
21     MyPrint()
22     {
23         count = 0;
24     }
25     void operator()(string test)
26     {
27         cout << test << endl;
28         count++; //统计使用次数
29     }
30     int count; //内部自己的状态
31 };
32 void test02()
33 {
34     MyPrint myPrint;
35     myPrint("hello world");
36     myPrint("hello world");
37     myPrint("hello world");
38     cout << "myPrint调用次数为:  " << myPrint.count << endl;
39 }
40 //3、函数对象可以作为参数传递
41 void doPrint(MyPrint &mp , string test)
42 {
43     mp(test);
44 }
45 void test03()
46 {
47     MyPrint myPrint;
48     doPrint(myPrint, "Hello C++");
49 }
50 int main() {
51     //test01();
52     //test02();
53     test03();
54     system("pause");
55     return 0;
56 }

```

总结:

- 仿函数写法非常灵活, 可以作为参数进行传递。

## 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 <vector>
2  #include <algorithm>
3
4  //1.一元谓词
5  struct GreaterFive{
6      bool operator()(int val) {
7          return val > 5;
8      }
9  };
10 void test01() {
11     vector<int> v;
12     for (int i = 0; i < 10; i++)
13     {
14         v.push_back(i);
15     }
16     vector<int>::iterator it = find_if(v.begin(), v.end(), GreaterFive());
17     if (it == v.end()) {
18         cout << "没找到!" << endl;
19     }
20     else {
21         cout << "找到:" << *it << endl;
22     }
23 }
24 int main() {
25     test01();
26     system("pause");
27     return 0;
28 }
```

### 4.2.3、二元谓词

```
1  #include <vector>
2  #include <algorithm>
3  //二元谓词
4  class MyCompare
5  {
6  public:
7      bool operator()(int num1, int num2)
8      {
9          return num1 > num2;
10     }
11 };
```

```

12 void test01()
13 {
14     vector<int> v;
15     v.push_back(10);
16     v.push_back(40);
17     v.push_back(20);
18     v.push_back(30);
19     v.push_back(50);
20
21     //默认从小到大
22     sort(v.begin(), v.end());
23     for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
24     {
25         cout << *it << " ";
26     }
27     cout << endl;
28     cout << "-----" << endl;
29     //使用函数对象改变算法策略，排序从大到小
30     sort(v.begin(), v.end(), MyCompare());
31     for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
32     {
33         cout << *it << " ";
34     }
35     cout << endl;
36 }
37 int main() {
38     test01();
39     system("pause");
40     return 0;
41 }

```

## 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：

- 算术仿函数
- 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 `#include<functional>`

### 4.3.2、算符仿函数

#### 功能描述：

- 实现四则运算
- 其中negate是一元运算，其他都是二元运算

#### 仿函数原型：

- `template<class T> T plus<T>` //加法仿函数
- `template<class T> T minus<T>` //减法仿函数
- `template<class T> T multiplies<T>` //乘法仿函数
- `template<class T> T divides<T>` //除法仿函数
- `template<class T> T modulus<T>` //取模仿函数
- `template<class T> T negate<T>` //取反仿函数

```
1  #include <functional>
2  //negate
3  void test01()
4  {
5      negate<int> n;
6      cout << n(50) << endl;
7  }
8  //plus
9  void test02()
10 {
11     plus<int> p;
12     cout << p(10, 20) << endl;
13 }
14 int main() {
15     test01();
16     test02();
17     system("pause");
18     return 0;
19 }
```

### 4.3.3 关系仿函数

#### 功能描述：

- 实现关系对比

#### 仿函数原型：

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

#### 示例：

```
1  #include <functional>
2  #include <vector>
```

```

3  #include <algorithm>
4
5  class MyCompare
6  {
7  public:
8      bool operator()(int v1,int v2)
9      {
10         return v1 > v2;
11     }
12 };
13 void test01()
14 {
15     vector<int> v;
16
17     v.push_back(10);
18     v.push_back(30);
19     v.push_back(50);
20     v.push_back(40);
21     v.push_back(20);
22
23     for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
24         cout << *it << " ";
25     }
26     cout << endl;
27
28     //自己实现仿函数
29     //sort(v.begin(), v.end(), MyCompare());
30     //STL内建仿函数 大于仿函数
31     sort(v.begin(), v.end(), greater<int>());
32
33     for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
34         cout << *it << " ";
35     }
36     cout << endl;
37 }
38
39 int main() {
40     test01();
41     system("pause");
42     return 0;
43 }

```

总结：关系仿函数中最常用的就是greater<>大于

#### 4.3.4 逻辑仿函数

功能描述：

- 实现逻辑运算

函数原型：

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



示例:

```
1  #include <vector>
2  #include <functional>
3  #include <algorithm>
4  void test01()
5  {
6      vector<bool> v;
7      v.push_back(true);
8      v.push_back(false);
9      v.push_back(true);
10     v.push_back(false);
11
12     for (vector<bool>::iterator it = v.begin(); it != v.end(); it++)
13     {
14         cout << *it << " ";
15     }
16     cout << endl;
17
18     //逻辑非 将v容器搬运到v2中，并执行逻辑非运算
19     vector<bool> v2;
20     v2.resize(v.size());
21     transform(v.begin(), v.end(), v2.begin(), logical_not<bool>());
22     for (vector<bool>::iterator it = v2.begin(); it != v2.end(); it++)
23     {
24         cout << *it << " ";
25     }
26     cout << endl;
27 }
28
29 int main() {
30     test01();
31     system("pause");
32     return 0;
33 }
```

总结: 逻辑仿函数实际应用较少, 了解即可

## 5、STL常用算法

概述:

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

## 5.1 常用遍历算法

### 学习目标:

- 掌握常用的遍历算法

### 算法简介:

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

### 5.1.1 for\_each

#### 功能描述:

- 实现遍历容器

#### 函数原型:

- `for_each(iterator beg, iterator end, _func);`  
// 遍历算法 遍历容器元素  
// beg 开始迭代器  
// end 结束迭代器  
// \_func 函数或者函数对象 (仿函数), 可以用类名创建的匿名对象如Func()

#### 示例:

```
1  #include <algorithm>
2  #include <vector>
3
4  //普通函数
5  void print01(int val)
6  {
7      cout << val << " ";
8  }
9  //函数对象
10 class print02
11 {
12     public:
13     void operator()(int val)
14     {
15         cout << val << " ";
16     }
17 };
18 //for_each算法基本用法
19 void test01() {
20     vector<int> v;
21     for (int i = 0; i < 10; i++)
22     {
23         v.push_back(i);
24     }
25     //遍历算法
26     for_each(v.begin(), v.end(), print01);
27     cout << endl;
28     for_each(v.begin(), v.end(), print02());
```

```

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

```

**总结：**for\_each在实际开发中是最常用遍历算法，需要熟练掌握

### 5.1.2 transform

**功能描述：**

- 搬运容器到另一个容器中

**函数原型：**

- `transform(iterator beg1, iterator end1, iterator beg2, _func);`

//beg1 源容器开始迭代器

//end1 源容器结束迭代器

//beg2 目标容器开始迭代器

//\_func 函数或者函数对象

```

1 vector<int>vTarget; //目标容器
2 vTarget.resize(v.size()); // 目标容器需要提前开辟空间
3 transform(v.begin(), v.end(), vTarget.begin(), TransForm());

```

**示例：**

```

1 #include<vector>
2 #include<algorithm>
3
4 //常用遍历算法 搬运 transform
5 class TransForm
6 {
7 public:
8     int operator()(int val)
9     {
10         return val;
11     }
12 };
13 class MyPrint
14 {
15 public:
16     void operator()(int val)
17     {
18         cout << val << " ";
19     }
20 };
21 void test01()
22 {
23     vector<int>v;

```

```

24     for (int i = 0; i < 10; i++)
25     {
26         v.push_back(i);
27     }
28     vector<int>vTarget; //目标容器
29     vTarget.resize(v.size()); // 目标容器需要提前开辟空间
30     transform(v.begin(), v.end(), vTarget.begin(), Transform());
31     for_each(vTarget.begin(), vTarget.end(), MyPrint());
32 }
33 int main() {
34     test01();
35     system("pause");
36     return 0;
37 }

```

**总结：**搬运的目标容器必须要提前开辟空间，否则无法正常搬运

## 5.2 常用查找算法

学习目标：

- 掌握常用的查找算法

算法简介：

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

### 5.2.1 find

功能描述：

- 查找指定元素，找到返回指定元素的迭代器，找不到返回结束迭代器end()

函数原型：

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

示例：

```

1  #include <algorithm>
2  #include <vector>
3  #include <string>
4  void test01() {
5
6      vector<int> v;
7      for (int i = 0; i < 10; i++) {

```

```
8         v.push_back(i + 1);
9     }
10    //查找容器中是否有 5 这个元素
11    vector<int>::iterator it = find(v.begin(), v.end(), 5);
12    if (it == v.end())
13    {
14        cout << "没有找到!" << endl;
15    }
16    else
17    {
18        cout << "找到:" << *it << endl;
19    }
20 }
21
22 class Person {
23 public:
24     Person(string name, int age)
25     {
26         this->m_Name = name;
27         this->m_Age = age;
28     }
29    //重载==
30    bool operator==(const Person& p)
31    {
32        if (this->m_Name == p.m_Name && this->m_Age == p.m_Age)
33        {
34            return true;
35        }
36        return false;
37    }
38
39 public:
40     string m_Name;
41     int m_Age;
42 };
43
44 void test02() {
45
46     vector<Person> v;
47
48    //创建数据
49    Person p1("aaa", 10);
50    Person p2("bbb", 20);
51    Person p3("ccc", 30);
52    Person p4("ddd", 40);
53
54    v.push_back(p1);
55    v.push_back(p2);
56    v.push_back(p3);
57    v.push_back(p4);
58
59    vector<Person>::iterator it = find(v.begin(), v.end(), p2);
60    if (it == v.end())
61    {
62        cout << "没有找到!" << endl;
```

```

63     }
64     else
65     {
66         cout << "找到姓名:" << it->m_Name << " 年龄: " << it->m_Age << endl;
67     }
68 }

```

总结：利用find可以在容器中找到指定的元素，返回值是迭代器

### 5.2.2 find\_if

功能描述：

- 按条件查找元素

函数原型：

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

只返回第一个找到的元素迭代器

示例：

```

1  #include <algorithm>
2  #include <vector>
3  #include <string>
4
5  //内置数据类型
6  class GreaterFive
7  {
8  public:
9      bool operator()(int val)
10     {
11         return val > 5;
12     }
13 };
14
15 void test01() {
16
17     vector<int> v;
18     for (int i = 0; i < 10; i++) {
19         v.push_back(i + 1);
20     }
21
22     vector<int>::iterator it = find_if(v.begin(), v.end(), GreaterFive());
23     if (it == v.end()) {
24         cout << "没有找到!" << endl;
25     }
26     else {
27         cout << "找到大于5的数字:" << *it << endl;
28     }

```

```

29 }
30
31 //自定义数据类型
32 class Person {
33 public:
34     Person(string name, int age)
35     {
36         this->m_Name = name;
37         this->m_Age = age;
38     }
39 public:
40     string m_Name;
41     int m_Age;
42 };
43
44 class Greater20
45 {
46 public:
47     bool operator()(Person &p)
48     {
49         return p.m_Age > 20;
50     }
51
52 };
53
54 void test02() {
55     vector<Person> v;
56
57     //创建数据
58     Person p1("aaa", 10);
59     Person p2("bbb", 20);
60     Person p3("ccc", 30);
61     Person p4("ddd", 40);
62
63     v.push_back(p1);
64     v.push_back(p2);
65     v.push_back(p3);
66     v.push_back(p4);
67
68     vector<Person>::iterator it = find_if(v.begin(), v.end(), Greater20());
69     if (it == v.end())
70     {
71         cout << "没有找到!" << endl;
72     }
73     else
74     {
75         cout << "找到姓名:" << it->m_Name << " 年龄: " << it->m_Age << endl;
76     }
77 }
78
79 int main() {
80     //test01();
81     test02();
82     system("pause");
83     return 0;

```

总结：find\_if按条件查找使查找更加灵活，提供的仿函数可以改变不同的策略

### 5.2.3 adjacent\_find

功能描述：

- 查找相邻重复元素

函数原型：

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

示例：

```

1  #include <algorithm>
2  #include <vector>
3
4  void test01()
5  {
6      vector<int> v;
7      v.push_back(1);
8      v.push_back(2);
9      v.push_back(5);
10     v.push_back(2);
11     v.push_back(4);
12     v.push_back(4);
13     v.push_back(3);
14
15     //查找相邻重复元素
16     vector<int>::iterator it = adjacent_find(v.begin(), v.end());
17     if (it == v.end()) {
18         cout << "找不到!" << endl;
19     }
20     else {
21         cout << "找到相邻重复元素为:" << *it << endl;
22     }
23 }
```

总结：面试题中如果出现查找相邻重复元素，记得用STL中的adjacent\_find算法

### 5.2.4 binary\_search

功能描述：

- 查找指定元素是否存在

函数原型：

- `bool binary_search(iterator beg, iterator end, value);`



```
// 查找指定的元素，查到 返回true 否则false
// 注意: 在无序序列中不可用
// beg 开始迭代器
// end 结束迭代器
// value 查找的元素
```

**示例:**

```
1  #include <algorithm>
2  #include <vector>
3
4  void test01()
5  {
6      vector<int>v;
7
8      for (int i = 0; i < 10; i++)
9      {
10         v.push_back(i);
11     }
12     //二分查找
13     bool ret = binary_search(v.begin(), v.end(), 2);
14     if (ret)
15     {
16         cout << "找到了" << endl;
17     }
18     else
19     {
20         cout << "未找到" << endl;
21     }
22 }
23 int main() {
24     test01();
25     system("pause");
26     return 0;
27 }
```

**总结:** 二分查找法查找效率很高，值得注意的是查找的容器中元素必须的有序序列

## 5.2.5 count

**功能描述:**

- 统计元素个数

**函数原型:**

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

**示例:**

```

1  #include <algorithm>
2  #include <vector>
3
4  //内置数据类型
5  void test01()
6  {
7      vector<int> v;
8      v.push_back(1);
9      v.push_back(2);
10     v.push_back(4);
11     v.push_back(5);
12     v.push_back(3);
13     v.push_back(4);
14     v.push_back(4);
15     int num = count(v.begin(), v.end(), 4);
16     cout << "4的个数为: " << num << endl;
17 }
18 //自定义数据类型
19 class Person
20 {
21 public:
22     Person(string name, int age)
23     {
24         this->m_Name = name;
25         this->m_Age = age;
26     }
27     bool operator==(const Person & p)
28     {
29         if (this->m_Age == p.m_Age)
30         {
31             return true;
32         }
33         else
34         {
35             return false;
36         }
37     }
38     string m_Name;
39     int m_Age;
40 };
41
42 void test02()
43 {
44     vector<Person> v;
45
46     Person p1("刘备", 35);
47     Person p2("关羽", 35);
48     Person p3("张飞", 35);
49     Person p4("赵云", 30);
50     Person p5("曹操", 25);
51
52     v.push_back(p1);
53     v.push_back(p2);
54     v.push_back(p3);
55     v.push_back(p4);

```

```

56     v.push_back(p5);
57
58     Person p("诸葛亮",35);
59
60     int num = count(v.begin(), v.end(), p);
61     cout << "num = " << num << endl;
62 }
63 int main() {
64     //test01();
65     test02();
66     system("pause");
67     return 0;
68 }

```

**总结：** 统计自定义数据类型时候，需要配合重载 `operator==`

## 5.2.6 count\_if

**功能描述：**

- 按条件统计元素个数

**函数原型：**

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

重载==在数据类型的类中写即可

重载 () 需要另外写一个类

**示例：**

```

1  #include <algorithm>
2  #include <vector>
3
4  class Greater4
5  {
6  public:
7      bool operator()(int val)
8      {
9          return val >= 4;
10     }
11 };
12 //内置数据类型
13 void test01()
14 {
15     vector<int> v;
16     v.push_back(1);
17     v.push_back(2);
18     v.push_back(4);
19     v.push_back(5);

```

```

20     v.push_back(3);
21     v.push_back(4);
22     v.push_back(4);
23
24     int num = count_if(v.begin(), v.end(), Greater4());
25     cout << "大于4的个数为: " << num << endl;
26 }
27
28 //自定义数据类型
29 class Person
30 {
31 public:
32     Person(string name, int age)
33     {
34         this->m_Name = name;
35         this->m_Age = age;
36     }
37     string m_Name;
38     int m_Age;
39 };
40 class AgeLess35
41 {
42 public:
43     bool operator()(const Person &p)
44     {
45         return p.m_Age < 35;
46     }
47 };
48 void test02()
49 {
50     vector<Person> v;
51
52     Person p1("刘备", 35);
53     Person p2("关羽", 35);
54     Person p3("张飞", 35);
55     Person p4("赵云", 30);
56     Person p5("曹操", 25);
57
58     v.push_back(p1);
59     v.push_back(p2);
60     v.push_back(p3);
61     v.push_back(p4);
62     v.push_back(p5);
63
64     int num = count_if(v.begin(), v.end(), AgeLess35());
65     cout << "小于35岁的个数: " << num << endl;
66 }
67
68
69 int main() {
70     //test01();
71     test02();
72     system("pause");
73     return 0;
74 }

```

**总结：**按值统计用count，按条件统计用count\_if

## 5.3 常用排序算法

**学习目标：**

- 掌握常用的排序算法

**算法简介：**

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

### 5.3.1 sort

**功能描述：**

- 对容器内元素进行排序

**函数原型：**

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

**示例：**

```
1  #include <algorithm>
2  #include <vector>
3
4  void myPrint(int val)
5  {
6      cout << val << " ";
7  }
8
9  void test01() {
10     vector<int> v;
11     v.push_back(10);
12     v.push_back(30);
13     v.push_back(50);
14     v.push_back(20);
15     v.push_back(40);
16
17     //sort默认从小到大排序
18     sort(v.begin(), v.end());
19     for_each(v.begin(), v.end(), myPrint);
20     cout << endl;
21
22     //从大到小排序
23     sort(v.begin(), v.end(), greater<int>());
24     for_each(v.begin(), v.end(), myPrint);
```

```

25     cout << endl;
26 }
27
28 int main() {
29     test01();
30     system("pause");
31     return 0;
32 }

```

**总结：**sort属于开发中最常用的算法之一，需熟练掌握

### 5.3.2 random\_shuffle

**功能描述：**

- 洗牌 指定范围内的元素随机调整次序

**函数原型：**

- `random_shuffle(iterator beg, iterator end);`

// 指定范围内的元素随机调整次序

// beg 开始迭代器

// end 结束迭代器

加随机数种子，保证每次的随机结果都不一样

`srand((unsigned int)time(NULL));`

**示例：**

```

1  #include <algorithm>
2  #include <vector>
3  #include <ctime>
4
5  class myPrint
6  {
7  public:
8      void operator()(int val)
9      {
10         cout << val << " ";
11     }
12 };
13
14 void test01()
15 {
16     srand((unsigned int)time(NULL));
17     vector<int> v;
18     for(int i = 0 ; i < 10;i++)
19     {
20         v.push_back(i);
21     }
22     for_each(v.begin(), v.end(), myPrint());
23     cout << endl;
24
25     //打乱顺序
26     random_shuffle(v.begin(), v.end());

```

```

27     for_each(v.begin(), v.end(), myPrint());
28     cout << endl;
29 }
30
31 int main() {
32     test01();
33     system("pause");
34     return 0;
35 }

```

**总结：** random\_shuffle洗牌算法比较实用，使用时记得加随机数种子

### 5.3.3 merge

**功能描述：**

- 两个容器元素合并，并存储到另一容器中

**函数原型：**

- `merge(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest);`

// 容器元素合并，并存储到另一容器中

// 注意: 两个容器必须是有序的

// beg1 容器1开始迭代器

// end1 容器1结束迭代器

// beg2 容器2开始迭代器

// end2 容器2结束迭代器

// dest 目标容器开始迭代

目标容器必须提前开辟空间，合并的两个容器必须是有序的且顺序必须相同

```

1     vector<int> vtarget;
2     //目标容器需要提前开辟空间
3     vtarget.resize(v1.size() + v2.size());

```

**示例：**

```

1     #include <algorithm>
2     #include <vector>
3
4     class myPrint
5     {
6     public:
7         void operator()(int val)
8         {
9             cout << val << " ";
10        }
11    };
12
13    void test01()
14    {
15        vector<int> v1;
16        vector<int> v2;

```

```

17     for (int i = 0; i < 10 ; i++)
18     {
19         v1.push_back(i);
20         v2.push_back(i + 1);
21     }
22     vector<int> vtarget;
23     //目标容器需要提前开辟空间
24     vtarget.resize(v1.size() + v2.size());
25     //合并 需要两个有序序列
26     merge(v1.begin(), v1.end(), v2.begin(), v2.end(), vtarget.begin());
27     for_each(vtarget.begin(), vtarget.end(), myPrint());
28     cout << endl;
29 }
30 int main() {
31     test01();
32     system("pause");
33     return 0;
34 }

```

**总结：** merge合并的两个容器必须的有序序列

### 5.3.4 reverse

**功能描述：**

- 将容器内元素进行反转

**函数原型：**

- `reverse(iterator beg, iterator end);`

// 反转指定范围的元素

// beg 开始迭代器

// end 结束迭代器

**示例：**

```

1  #include <algorithm>
2  #include <vector>
3
4  class myPrint
5  {
6  public:
7      void operator()(int val)
8      {
9          cout << val << " ";
10     }
11 };
12 void test01()
13 {
14     vector<int> v;
15     v.push_back(10);
16     v.push_back(30);
17     v.push_back(50);
18     v.push_back(20);
19     v.push_back(40);

```



```

20     cout << "反转前: " << endl;
21     for_each(v.begin(), v.end(), myPrint());
22     cout << endl;
23     cout << "反转后: " << endl;
24     reverse(v.begin(), v.end());
25     for_each(v.begin(), v.end(), myPrint());
26     cout << endl;
27 }
28 int main() {
29     test01();
30     system("pause");
31     return 0;
32 }

```

**总结：**reverse反转区间内元素，面试题可能涉及到

## 5.4 常用拷贝和替换算法

**学习目标：**

- 掌握常用的拷贝和替换算法

**算法简介：**

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

### 5.4.1 copy

**功能描述：**

- 容器内指定范围的元素拷贝到另一容器中

**函数原型：**

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

**示例：**

```

1  #include <algorithm>
2  #include <vector>
3
4  class myPrint
5  {
6  public:
7      void operator()(int val)
8      {
9          cout << val << " ";
10     }

```

```

11 };
12
13 void test01()
14 {
15     vector<int> v1;
16     for (int i = 0; i < 10; i++) {
17         v1.push_back(i + 1);
18     }
19     vector<int> v2;
20     v2.resize(v1.size());
21     copy(v1.begin(), v1.end(), v2.begin());
22     for_each(v2.begin(), v2.end(), myPrint());
23     cout << endl;
24 }
25 int main() {
26     test01();
27     system("pause");
28     return 0;
29 }

```

**总结：**利用copy算法在拷贝时，目标容器记得提前开辟空间

## 5.4.2 replace

**功能描述：**

- 将容器内指定范围的旧元素修改为新元素

**函数原型：**

- `replace(iterator beg, iterator end, oldvalue, newvalue);`

// 将区间内旧元素 替换成 新元素

// beg 开始迭代器

// end 结束迭代器

// oldvalue 旧元素

// newvalue 新元素

**示例：**

```

1  #include <algorithm>
2  #include <vector>
3
4  class myPrint
5  {
6  public:
7      void operator()(int val)
8      {
9          cout << val << " ";
10     }
11 };
12
13 void test01()
14 {
15     vector<int> v;

```

```

16     v.push_back(20);
17     v.push_back(30);
18     v.push_back(20);
19     v.push_back(40);
20     v.push_back(50);
21     v.push_back(10);
22     v.push_back(20);
23     cout << "替换前: " << endl;
24     for_each(v.begin(), v.end(), myPrint());
25     cout << endl;
26     //将容器中的20 替换成 2000
27     cout << "替换后: " << endl;
28     replace(v.begin(), v.end(), 20, 2000);
29     for_each(v.begin(), v.end(), myPrint());
30     cout << endl;
31 }
32 int main() {
33     test01();
34     system("pause");
35     return 0;
36 }

```

### 自定义对象replace

```

1  #include<iostream>
2  #include <algorithm>
3  #include <vector>
4  using namespace std;
5
6  class Person {
7  public:
8      Person(string name,int age) {
9          this->name = name;
10         this->age = age;
11     }
12     Person(const Person& p) {
13         this->name = p.name;
14         this->age = p.age;
15     }
16     bool operator==(const Person& p) {
17         if (this->age == p.age&&this->name==p.name) {
18             return true;
19         }
20         else return false;
21     }
22     string name;
23     int age;
24 };
25 int main() {
26     vector<Person> vec;
27     vec.push_back(Person("张飞",28));
28     vec.push_back(Person("赵云", 23));
29     vec.push_back(Person("马超", 23));
30     vec.push_back(Person("诸葛亮", 20));
31     vec.push_back(Person("黄忠", 30));

```

```

32     replace(vec.begin(), vec.end(), Person("黄忠", 23), Person("黄忠", 40));
33     for (auto it = vec.begin(); it != vec.end(); it++) {
34         cout << "name:" << it->name << "    age:" << it->age << endl;
35     }
36     return 0;
37 }

```

**总结：**replace会替换区间内满足条件的元素

### 5.4.3 replace\_if

**功能描述：**

- 将区间内满足条件的元素，替换成指定元素

**函数原型：**

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

**示例：**

```

1  #include <algorithm>
2  #include <vector>
3
4  class myPrint
5  {
6  public:
7      void operator()(int val)
8      {
9          cout << val << " ";
10     }
11 };
12 class ReplaceGreater30
13 {
14 public:
15     bool operator()(int val)
16     {
17         return val >= 30;
18     }
19 };
20
21 void test01()
22 {
23     vector<int> v;
24     v.push_back(20);
25     v.push_back(30);
26     v.push_back(20);
27     v.push_back(40);
28     v.push_back(50);

```

```

29     v.push_back(10);
30     v.push_back(20);
31
32     cout << "替换前: " << endl;
33     for_each(v.begin(), v.end(), myPrint());
34     cout << endl;
35     //将容器中大于等于的30 替换成 3000
36     cout << "替换后: " << endl;
37     replace_if(v.begin(), v.end(), ReplaceGreater30(), 3000);
38     for_each(v.begin(), v.end(), myPrint());
39     cout << endl;
40 }
41 int main() {
42     test01();
43     system("pause");
44     return 0;
45 }

```

**总结:** replace\_if按条件查找, 可以利用仿函数灵活筛选满足的条件

## 5.4.4 swap

**功能描述:**

- 互换两个容器的元素

**函数原型:**

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

**示例:**

```

1  #include <algorithm>
2  #include <vector>
3
4  class myPrint
5  {
6  public:
7      void operator()(int val)
8      {
9          cout << val << " ";
10     }
11 };
12 void test01()
13 {
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+100);
19     }
20     cout << "交换前: " << endl;

```

```

21     for_each(v1.begin(), v1.end(), myPrint());
22     cout << endl;
23     for_each(v2.begin(), v2.end(), myPrint());
24     cout << endl;
25
26     cout << "交换后: " << endl;
27     swap(v1, v2);
28     for_each(v1.begin(), v1.end(), myPrint());
29     cout << endl;
30     for_each(v2.begin(), v2.end(), myPrint());
31     cout << endl;
32 }
33 int main() {
34     test01();
35     system("pause");
36     return 0;
37 }

```

**总结:** swap交换容器时, 注意交换的容器要同种类型

## 5.5 常用算术生成算法

**学习目标:**

- 掌握常用的算术生成算法

**注意:**

- 算术生成算法属于小型算法, 使用时包含的头文件为 `#include <numeric>`

**算法简介:**

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

### 5.5.1 accumulate

**功能描述:**

- 计算区间内 容器元素累计总和

**函数原型:**

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

**示例:**

```

1  #include <numeric>
2  #include <vector>
3  void test01()
4  {
5      vector<int> v;

```

```

6     for (int i = 0; i <= 100; i++) {
7         v.push_back(i);
8     }
9     int total = accumulate(v.begin(), v.end(), 0);
10    cout << "total = " << total << endl;
11 }
12 int main() {
13     test01();
14     system("pause");
15     return 0;
16 }

```

**总结：**accumulate使用时头文件注意是 numeric，这个算法很实用

## 5.5.2 fill

**功能描述：**

- 向容器中填充指定的元素

**函数原型：**

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

**示例：**

```

1  #include <numeric>
2  #include <vector>
3  #include <algorithm>
4
5  class myPrint
6  {
7  public:
8      void operator()(int val)
9      {
10         cout << val << " ";
11     }
12 };
13
14 void test01()
15 {
16     vector<int> v;
17     v.resize(10);
18     //填充
19     fill(v.begin(), v.end(), 100);
20     for_each(v.begin(), v.end(), myPrint());
21     cout << endl;
22 }
23 int main() {
24     test01();
25     system("pause");

```

```
26     return 0;
27 }
```

**总结:** 利用fill可以将容器区间内元素填充为 指定的值

## 5.6 常用集合算法

**学习目标:**

- 掌握常用的集合算法

**算法简介:**

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

### 5.6.1 set\_intersection

**功能描述:**

- 求两个容器的交集

**函数原型:**

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

**示例:**

itEnd并不等于VTarget.end(),itEnd是交集最后一个迭代器,并不一定是VTarget的最后一个,因为VTarget的size开辟的空间不会被全部填满,除非一个集合是另一个的子集

```
1 vector<int>::iterator itEnd =
2     set_intersection(v1.begin(), v1.end(), v2.begin(), v2.end(),
3     vTarget.begin());
```

```
1 #include <vector>
2 #include <algorithm>
3
4 class myPrint
5 {
6 public:
7     void operator()(int val)
8     {
9         cout << val << " ";
10    }
11};
```



```

12
13 void test01()
14 {
15     vector<int> v1;
16     vector<int> v2;
17     for (int i = 0; i < 10; i++)
18     {
19         v1.push_back(i);
20         v2.push_back(i+5);
21     }
22     vector<int> vTarget;
23     //取两个里面较小的值给目标容器开辟空间
24     vTarget.resize(min(v1.size(), v2.size()));
25     //返回目标容器的最后一个元素的迭代器地址
26     vector<int>::iterator itEnd =
27         set_intersection(v1.begin(), v1.end(), v2.begin(), v2.end(),
28         vTarget.begin());
29     for_each(vTarget.begin(), itEnd, myPrint());
30     cout << endl;
31 }
32 int main() {
33     test01();
34     system("pause");
35     return 0;
36 }

```

#### 总结:

- 求交集的两个集合必须的有序序列
- 目标容器开辟空间需要从**两个容器中取小值**
- **set\_intersection返回值既是交集中最后一个元素的位置**

### 5.6.2 set\_union

#### 功能描述:

- 求两个集合的并集

#### 函数原型:

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

#### 示例:

```

1 #include <vector>
2 #include <algorithm>
3

```

```

4  class myPrint
5  {
6  public:
7      void operator()(int val)
8      {
9          cout << val << " ";
10     }
11 };
12
13 void test01()
14 {
15     vector<int> v1;
16     vector<int> v2;
17     for (int i = 0; i < 10; i++) {
18         v1.push_back(i);
19         v2.push_back(i+5);
20     }
21
22     vector<int> vTarget;
23     //取两个容器的和给目标容器开辟空间
24     vTarget.resize(v1.size() + v2.size())
25     //返回目标容器的最后一个元素的迭代器地址
26     vector<int>::iterator itEnd =
27         set_union(v1.begin(), v1.end(), v2.begin(), v2.end(),
28 vTarget.begin());
29     for_each(vTarget.begin(), itEnd, myPrint());
30     cout << endl;
31 }
32 int main() {
33     test01();
34     system("pause");
35     return 0;
36 }

```

总结:

- 求并集的两个集合必须的有序序列
- 目标容器开辟空间需要两个容器相加
- set\_union返回值既是并集中最后一个元素的位置

### 5.6.3 set\_difference

功能描述:

- 求两个集合的差集

函数原型:

- set\_difference(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest);  
// 求两个集合的差集  
// 注意:两个集合必须是有序序列

```
// beg1 容器1开始迭代器
// end1 容器1结束迭代器
// beg2 容器2开始迭代器
// end2 容器2结束迭代器
// dest 目标容器开始迭代器
```

#### 示例:

```
1  #include <vector>
2  #include <algorithm>
3
4  class myPrint
5  {
6  public:
7      void operator()(int val)
8      {
9          cout << val << " ";
10     }
11 };
12
13 void test01()
14 {
15     vector<int> v1;
16     vector<int> v2;
17     for (int i = 0; i < 10; i++) {
18         v1.push_back(i);
19         v2.push_back(i+5);
20     }
21
22     vector<int> vTarget;
23     //取两个里面较大的值给目标容器开辟空间
24     vTarget.resize( max(v1.size() , v2.size()));
25
26     //返回目标容器的最后一个元素的迭代器地址
27     cout << "v1与v2的差集为: " << endl;
28     vector<int>::iterator itEnd =
29         set_difference(v1.begin(), v1.end(), v2.begin(), v2.end(),
30 vTarget.begin());
31     for_each(vTarget.begin(), itEnd, myPrint());
32     cout << endl;
33
34     cout << "v2与v1的差集为: " << endl;
35     itEnd = set_difference(v2.begin(), v2.end(), v1.begin(), v1.end(),
36 vTarget.begin());
37     for_each(vTarget.begin(), itEnd, myPrint());
38     cout << endl;
39 }
40
41 int main() {
42     test01();
43     system("pause");
44     return 0;
45 }
```

#### 总结:

- 求差集的两个集合必须的有序序列
- 目标容器开辟空间需要从**两个容器取较大值**
- `set_difference`返回值既是差集中最后一个元素的位置