C++提高编程

- 本阶段主要针对**C++**泛型编程和**STL**技术做详细讲解，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**通用的摸具**，大大**提高复用率**

例如：PPT模板

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，其函数返回类型和形参类型可以不具体制定，用一个**虚拟的类型**来代表。

语法：

```
template<typename T>
函数声明或者定义
```

解释：

template --- 声明创建模板

typename --- 表示其后面的符号是一种数据类型，可以用class 来代替

T --- 通用的数据类型，名称可以替换，通常为大写字母

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

//函数模板
/*
//交换两个整型函数
void swapInt(int& a, int& b)
{
    int temp = a;
```

```
    a = b;
    b = temp;
}

//交换两个浮点型函数
void swapDouble(double& a, double& b)
{
    double temp = a;
    a = b;
    b = temp;
}
*/
// 函数模板
template<typename T> //声明一个模板，告诉编译器后面代码紧跟着的T不要报错，T是一个通用数据类型
void mySwap(T& a, T& b)
{
    T temp = a;
    a = b;
    b = temp;
}

void test01()
{
    int a = 10;
    int b = 20;

    //swapInt(a, b);
    //利用函数模板交换
    //两个字方式使用函数模板
    //1、自动类型推导
    //mySwap(a, b);

    //2、显示指定类型
    mySwap<int>(a, b); //指定T的类型
    cout << "a = " << a << " , b = " << b << endl;

    double c = 1.1;
    double d = 2.2;
    mySwap(c, d);
    //swapDouble(c, d);
    cout << "c = " << c << " , d = " << d << endl;
}

int main()
{
    test01();
    return 0;
}
```

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

```
#include<iostream>
using namespace std;
//函数模板注意事项
template<class T>  //typename可以替换成class
void mySwap(T& a, T& b)
{
    T temp = a;
    a = b;
    b = temp;
}

//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
void test01()
{
    int a = 10;
    int b = 20;
    char c = 'c';
    //mySwap(a, c); 错误，推导出类型不一致
    mySwap(a, b);
    cout << "a=" << a << ",b=" << b << endl;
}

//2、模板必须要确定出T的数据类型才可以使用
template<class T>
void func()
{
    cout << "func 调用" << endl;
}

void test02()
{
    //func(); 错误，模板不能独立使用，必须确定出T的类型
    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
}

int main()
{
    test01();
    test02();
    return 0;
}
```

总结：

- 使用模板时，必须确定出通用数据类型T，并且能够推导出一致的类型

1.2.3 普通函数与函数模板的区别

普通函数与函数模板的区别：

- 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
- 如果利用显式指定类型的方式，可以发生隐式类型转换

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

//普通函数与函数模板的区别

//- 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
//- 如果利用显式指定类型的方式，可以发生隐式类型转换

//普通函数
int myAdd01(int a, int b)
{
    return a + b;
}

//函数模板
template<class T>
T myAdd02(T a,T b)
{
    return a + b;
}

void test01()
{
    int a = 10;
    int b = 20;
    char c = 'c'; //c- 99
    cout << myAdd01(a, c) << endl;  //这里把字符型变量自动（隐式）转换成整型

    //cout << myAdd02(a, c) << endl; //错误，存在二义性

    //显式指定类型
    cout << myAdd02<int>(a, c) << endl;
}
int main()
{
    test01();
    return 0;
}
```

总结：建议使用显式指定类型的方式，调用函数模板，因为可以自己确定通用类型T

1.2.4 普通函数与函数模板的调用规则

调用规则如下：

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
2. 可以通过空模板参数列表来强制调用函数模板
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

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

//普通函数和函数模板的调用规则

//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

//2. 可以通过空模板参数列表来强制调用函数模板

//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

void myPrint(int a, int b)  //
{
    cout << "调用普通函数" << endl;
}

template<class T>
void myPrint(T a, T b)
{
    cout << "调用函数模板" << endl;
}

template<class T>
void myPrint(T a, T b,T c)
{
    cout << "调用重载的函数模板" << endl;
}

void test01()
{
    int a = 10;
    int b = 20;
    myPrint(a,b); // 如果函数模板和普通函数都可以实现，优先调用普通函数。如果普通函数只有声明没有定义，则会报错。
    myPrint<>(a, b);   // 可以通过空模板参数列表来强制调用函数模板
    myPrint(a, b, 100);  //函数模板也可以发生重载
    // 如果函数模板可以产生更好的匹配，优先调用函数模板
    char c1 = 'a';
    char c2 = 'b';
    myPrint(c1, c2);  //此时函数模板更加匹配，优先调用函数模板
}

int main()
{
    test01();
    return 0;
```

```
}
```

总结：既然提供了函数模板，最好就不要提供普通函数了，否则会出现二义性。

1.2.5函数模板的局限性

局限性：

- 模板的通用性并不是万能的

例如：

```
template<class T>
void f(T a,T,b)
{
    a = b;
}
```

在上述代码中提供的赋值操作，如果传入的a和b是一个数组，就无法实现了。

再例如：

```
template<class T>
void f(T a,T,b)
{
    if(a > b) {...}
}
```

在上述代码中，如果T的数据类型传入的是像Person这样的自定义数据源类型，也无法正常运行

因此C++为了解决这种问题，提供了模板的重载，可以为特**定的类型**提供**具体化的模板**

示例：

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

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

//模板局限性
//模板并不是万能的，有些特定数据类型，需要用具体化方式做特殊实现

template<class T>
bool myCompare(T& a, T& b)
```

```
{
    if (a == b)
    {
        return true;
    }
    else
    {
        return false;
    }
}

//利用具体化Person的版本实现代码，具体化优先调用，模板重载
template<> bool myCompare(Person& p1, Person& p2)
{
    if (p1.m_name == p2.m_name && p1.m_age == p2.m_age)
    {
        return true;
    }
    else
    {
        return false;
    }
}

void test01()
{
    int a = 10;
    int b = 20;
    bool ret = myCompare(a, b);
    if (ret)
    {
        cout << "a==b" << endl;
    }
    else
    {
        cout << "a!=b" << endl;
    }

}

//解决办法 1、运算符重载
//2、模板重载
void test02() //自定义数据类型比较
{
    Person p1("Tom", 10);
    Person p2("Tom", 11);
    bool ret = myCompare(p1, p2);
    if (ret)
    {
        cout << "p1==p2" << endl;
    }
    else
    {
        cout << "p1!=p2" << endl;
    }
}
int main()
{
```

```
    test01();
    test02();
    return 0;
}
```

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（Standard Template Library 标准模板库）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：

- 建议一个通用类，类中的成员 数据类型可以不具体制定，用一个**虚拟的类型**来代表

语法：

```
template<typename T>
类
```

解释：

template --- 声明创建模板

typename --- 表示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

示例：

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

//类模板
template<class NameType,class AgeType>
class Person
{
public:
    Person(NameType name,AgeType age)
    {
        this->m_name = name;
        this->m_age = age;
    }
    void showPerson()
    {
        cout << "name: " << this->m_name << " ,`age: " << this->m_age << endl;
    }
    NameType m_name;
    AgeType m_age;
};
```

```
void test01()
{
    Person<string, int>p1("孙悟空", 999);
    p1.showPerson();
}

int main()
{
    test01();
    return 0;
}
```

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

1.3.2 类模板与函数模板的区别

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

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

//类模板与函数模板的区别
template<class NameType,class AageType =int> //可以有默认参数
class Person
{
public:
    Person(NameType name, AageType age)
    {
        this->m_name = name;
        this->m_age = age;
    }
    void showPerson()
    {
        cout << "name: " << this->m_name << ", age: " << this->m_age << endl;
    }
    NameType m_name;
    AageType m_age;
};

//1、类模板没有自动类推导使用方式
void test01()
{
    // Person p("孙悟空", 100); //报错，无法使用自动类型推导
    Person<string, int >p("孙悟空", 100);
    p.showPerson();
}

//2、类模板在参数列表中可以有默认参数,函数模板中不可以有默认参数。
void test02()
{
    Person <string>p2("猪八戒", 999);
    p2.showPerson();
```

```
}

int main()
{
    test01();
    test02();
    return 0;
}
```

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是有区别的：

- 普通类中的成员函数一开始就可以创建
- 类模板中的成员函数调用时才创建

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

//类模板中成员函数创建时机

//类模板中的成员函数在调用时才会去创建

class Person1
{
public:
    void showPerson1()
    {
        cout << "Person1 show" << endl;
    }
};

class Person2
{
public:
    void showPerson()
    {
        cout << "Person2 show" << endl;
    }
};

template<class T>
class Myclass
{
public:
    T obj;
    //类模板中的成员函数，一开始是不会被创建，因为还没被调用，会在模板调用时再生成
    void func1()
    {
        obj.showPerson1();
    }
    void func2()
    {
        obj.showPerosn2();
```

```
    }
};

void test01()
{
    Myclass<Person1>m;
    m.func1();
    //m.func2(); 编译会出错，说明函数调用才会去创建成员函数
}
int main()
{
    test01();
    return 0;
}
```

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，像函数传参的方式

一共有三种传入方式：

1. 指定传入的类型 --- 直接显示对象的数据类型
2. 参数模板化 --- 将对象中的参数变为模板进行传递
3. 整个类模板化 --- 将这个对象类型 模板化进行传递

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

//类模板对象做函数参数
template<class T1,class T2>
class Person
{
public:
    Person(T1 name, T2 age)
    {
        this->m_name = name;
        this->m_age = age;
    }
    void showPerson()
    {
        cout << "name: " << this->m_name << " ,age: " << this->m_age << endl;
    }
    T1 m_name;
    T2 m_age;
};

//1、指定传入类型
void printPerson1(Person<string, int>& p)
{
    p.showPerson();
}
void test01()
```

```
{
    Person<string, int>p("孙悟空", 100);
    printPerson1(p);
}

//2、参数模板化
template<class T1,class T2>
void printPerson2(Person<T1, T2>& p)
{
    p.showPerson();
    cout << "T1的类型为：" << typeid(T1).name() << endl;
    cout << "T2的类型为：" << typeid(T2).name() << endl;
}
void test02()
{
    Person<string, int>p("猪八戒", 90);
    printPerson2(p);
}

//3、整个类模板化
template<class T>
void printPerson3(T &p)
{
    p.showPerson();
    cout << "T1的类型为：" << typeid(T).name() << endl;
}

void test03()
{
    Person<string, int>p("唐僧", 30);
    printPerson3(p);

}

int main()
{
    test01();
    test02();
    test03();
    return 0;
}
```

总结：

- 通过类模板创建的对象，可以有三种方式向函数中进行传参
- 使用比较广泛是：指定传入的类型

1.3.5 类模板与继承

当类模板碰到继承时，需要注意以下几点：

- 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
- 如果想灵活指定父类中T的类型，子类也需要变为模板

```
#include<iostream>
```

```cpp
using namespace std;

//类模板与继承

template<class T>
class Base
{
    T m;
};
class Son :public Base<int> //必须知道父类中的T类型，才能继承给子类
{

};

void test01()
{
    Son s1;
}
//如果想灵活指定父类中T类型，子类也需要变类模板
template<class T1,class T2>
class Son2 :public Base<T2> //T2是给父类指定T类型
{
public:
    Son2()
    {
        cout << "T1的数据类型为: " << typeid(T1).name() << endl;
        cout << "T2的数据类型为: " << typeid(T2).name() << endl;
    }
    T1 obj;
};
void test02()
{
    Son2<int, char>s2;
}

int main()
{
    test01();
    test02();
    return 0;
}
```

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：能够掌握类模板中的成员函数类外实现

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

//类模板成员函数类外实现
template<class T1,class T2>
class Person
{
```

```
public:
    Person(T1 name, T2 age); //函数声明
        //{
        //  this->m_name = name;
        //  this->m_age = age;
        //}
    void showPerson(); //函数声明
    //{
    //  cout << "姓名：" << this->m_name << " ，年龄：" << this->m_age << endl;
    //}

    T1 m_name;
    T2 m_age;
};

//构造函数类外实现
template<class T1, class T2>
Person<T1,T2>::Person(T1 name, T2 age)
{
    this->m_name = name;
    this->m_age = age;
}

//成员函数类外实现
template<class T1, class T2>
void Person<T1,T2>::showPerson()
{
    cout << "姓名：" << this->m_name << " ，年龄：" << this->m_age << endl;
}

void test01()
{
    Person<string, int>p("Tom", 20);
    p.showPerson();
}

int main()
{
    test01();
    return 0;
}
```

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以及解决方式

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：

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

实例：

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

//类模板分文件编写问题以及解决

//1、第一种解决方式，直接包含.cpp源文件
//#include "person.cpp"

//2、将声明和实现写到同一个文件中，并更改改后缀名为.hpp，.hpp是约定的名称，并不是强制的
#include "person.hpp"
//template<class T1,class T2>
//class Person
//{
//public:
//  Person(T1 name, T2 age);
//  void showPerson();
//  T1 m_name;
//  T2 m_age;
//};
//
//template<class T1, class T2>
//Person<T1,T2>::Person(T1 name, T2 age)
//{
//  this->m_name = name;
//  this->m_age = age;
//}
//
//
//template<class T1, class T2>
//void Person<T1,T2>::showPerson()
//{
//  cout << "姓名：" << this->m_name << " ，年龄：" << this->m_age << endl;
//}

void test01()
{
    Person<string, int>p("jerry", 10);
    p.showPerson();

}

int main()
{
    test01();
    return 0;
}
```

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的存在

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

//提前让编译器知道Person类的存在
template<class T1,class T2>
class Person;
//2、全局函数在类外实现
template<class T1, class T2>
void printPerson2(Person<T1, T2>p)
{
    cout << "全局函数类外实现：姓名：" << p.m_name << " ,年龄：" << p.m_age << endl;
}

//通过全局函数，打印Person信息

template<class T1,class T2>
class Person
{
    //全局函数类内实现
    friend void printPerson(Person<T1, T2>p) //参数模板化传参
    {
        cout << "全局函数类内实现：姓名：" << p.m_name << " ,年龄：" << p.m_age <<
endl;
    }

    //全局函数类外实现
    //加空模板参数列表，来优先调用函数模板
    //如果全局函数 是类外实现，需要让编译器提前知道这个函数的存在
    friend void printPerson2<>(Person<T1, T2>p);
public:
    Person(T1 name, T2 age)
    {
        this->m_name = name;
        this->m_age = age;
    }
private:
    T1 m_name;
    T2 m_age;
};

//1、全局函数在类内实现
void test01()
{
    Person<string, int>p("Tom", 20);
    printPerson(p);
}

void test02()
```

```
{
    Person<string, int>p("Jerry", 19);
    printPerson2(p);
}

int main()
{
    test01();
    test02();
    return 0;
}
```

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

编程感想：当你使用一个函数或者一个变量时，应该考虑当前的执行文件中是否引用或者链接到这些函数以及变量，以及应该考虑声明函数以及变量的时机与使用函数与变量的时机是否冲突，否则运行会出错。

2 STL初识

2.1 STL的诞生

- 长久以来，软件界一直希望建立一种可重复利用的东西
- C++的**面向对象**和**泛型编程思想**，目的就是**复用性的提升**
- 大多数情况下，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的基本概念

- 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：扮演了容器和算法之间的胶合剂
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者迭代器接口的东西
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，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：强调知道配许，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

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、技术、遍历、寻找极值等。

迭代器：容器和算法之间的粘合剂

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

每个容器都有自己的专属迭代器

迭代器使用非常类似于指针，初学阶段我们可以理解迭代器为指针

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	读数据的只写访问	只读，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容器算法迭代器初识

了解STL中容器、算法、迭代器概念之后，我们利用代码感受STL的魅力

STL中最常用的容器为Vector，可以理解为数组，下面我们将学习如何想这个容器中插入数据，并遍历这个容器

2.5.1 vector存放内置数据类型

容器：`vector`

算法：`for_each`

迭代器：`vector<int>::iterator`

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

void myPrint(int val)
{
    cout << val << endl;
}

void test01()
{
    //创建一个vector容器，数组
    vector<int> v;
    //向容器中尾插数据
    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    v.push_back(40);
    v.push_back(50);

    //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，这个给迭代器指向容器中第一个元素
    //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<int>::iterator 拿到vector<int>这种容器的迭代器类型


    //通过迭代器访问容器中的数据
    vector<int>::iterator itBegin = v.begin(); //起始迭代器 指向容器中第一个元素
    vector<int>::iterator itEnd = v.end(); //结束迭代器 指向容器中最后一个元素的下一个位置

    //第一种遍历方式
    cout << "第一种遍历方式" << endl;
    while (itBegin != itEnd)
    {
        cout << *itBegin << endl;
        itBegin++;
    }
```

```
    //第二种遍历方式
    cout << "第二种遍历方式" << endl;
    for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
    {
        cout << *it << endl;
    }

    //第三种遍历方式，利用STL提供的遍历算法
    cout << "第二种遍历方式" << endl;
    for_each(v.begin(), v.end(), myPrint); //利用了函数回调的方式
}

int main()
{
    test01();
    return 0;
}
```

2.5.2 Vector存放自定义数据类型

学习目标：vector中存放自定义数据类型，并打印输出

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

//vector容器中存放自定义数据类型

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

void test01()
{
    vector<Person> v;

    Person p1("aaa", 10);
    Person p2("bbb", 20);
    Person p3("ccc", 30);
    Person p4("ddd", 40);
    Person p5("eee", 50);

    //向容器中添加数据
    v.push_back(p1);
    v.push_back(p2);
    v.push_back(p3);
```

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

    //遍历容器中的数据
    for (vector<Person>::iterator it = v.begin(); it != v.end(); it++)
    {
        //cout << "姓名：" << (*it).m_name << " ,年龄：" << (*it).m_age << endl;
        cout << "姓名：" << it->m_name << " ,年龄：" << it->m_age << endl;
    }

}

//存放自定义数据类型 指针
void test02()
{
    vector<Person*> v;

    Person p1("aaa", 10);
    Person p2("bbb", 20);
    Person p3("ccc", 30);
    Person p4("ddd", 40);
    Person p5("eee", 50);

    //向容器中添加数据
    v.push_back(&p1);
    v.push_back(&p2);
    v.push_back(&p3);
    v.push_back(&p4);
    v.push_back(&p5);


    //遍历容器中的数据
    for (vector<Person*>::iterator it = v.begin(); it != v.end(); it++)
    {
        //cout << "姓名：" << (*it).m_name << " ,年龄：" << (*it).m_age << endl;
        cout << "第二个：姓名：" << (*it)->m_name << " ,年龄：" << (*it)->m_age <<
endl;
    }
}

int main()
{
    test01();
    cout << endl;
    test02();
    return 0;
}
```

2.5.3 Vector容器嵌套容器

学习目标：容器中嵌套容器，我们将所有数据进行遍历输出

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

```cpp
#include<algorithm>


//容器嵌套容器
void test01()
{
    vector <vector<int>> v;

    //创建小容器
    vector<int>v1;
    vector<int>v2;
    vector<int>v3;
    vector<int>v4;

    //小容器中添加数据
    for (int i= 0; i < 4; i++)
    {
        v1.push_back(i + 1);
        v2.push_back(i + 2);
        v3.push_back(i + 3);
        v4.push_back(i + 4);
    }

    //将小容器插入到大容器中
    v.push_back(v1);
    v.push_back(v2);
    v.push_back(v3);
    v.push_back(v4);

    //通过大容器，把所有数据遍历一遍
    for (vector<vector<int>>::iterator it = v.begin(); it != v.end(); it++)
    {
        //(*it) ---- 容器vector<int>
        for (vector<int>::iterator vit = (*it).begin(); vit != (*it).end();
vit++)
        {
            cout << *vit << " ";
        }
        cout << endl;
    }
}

int main()
{
    test01();
    return 0;
}
```

3 STL-常用容器

3.1 string容器

3.1.1 string基本概念

本质：

- string是C++风格的字符串，而string本质上是一个类

string和char *的区别：

- char *是一个指针
- sting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

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

string管理char *所分配的内存，不用担心复制越界和取值越界，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`

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

//string构造函数

void test01()
{
    //string();                         //创建一个空的字符串，例如string str;
    string s1;                          //创建空字符串，调用无参构造函数

    //string(const char* s);            //使用字符串s初始化
    const char* str = "hello world!";
    string s2(str);                     //把c_string转换成string
    cout << "s2 = " << s2 << endl;

    //string(const string& str);        //使用一个string对象初始化另一个string对象
    string s3(s2);                      //调用拷贝构造函数
    cout << "s3 = " << s3 << endl;

    //string(int n,char c);              //使用n个字符c初始化
    string s4(10, 'a');
    cout << "s4 = " << s4 << endl;
}

int main()
{
```

```
    test01();
    return 0;
}
```

3.1.3 string赋值操作

功能描述：

- 给string字符串进行赋值操作

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付给当前的字符串
- `string& operator=(char c);` //字符赋值给当前的字符串
- `string& assgin(const char *s);` //把字符串s赋给当前的字符串
- `string& assgin(const char *s, int n);` //把当前s的前n个字符赋给当前的字符串
- `string& assgin(const string &s);` //把字符串s赋给当前字符串
- `string& assgin(int n, char c);` //把n个字符c赋给当前字符串

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

//string赋值操作

void test01()
{
    //string& operator=(const char* s);         //char*类型字符串赋值给当前的字符串
    string str1;
    str1 = "hello world!";
    cout << "str1 = " << str1 << endl;

    //string& operator=(const string &s);       //把字符串s付给当前的字符串
    string str2;
    str2 = str1;
    cout << "str2 = " << str2 << endl;

    //string& operator=(char c);                //字符赋值给当前的字符串
    string str3;
    str3 = 'a';
    cout << "str3 = " << str3 << endl;

    //string& assgin(const char *s);            //把字符串s赋给当前的字符串
    string str4;
    str4.assign("hello C++");
    cout << "str4 = " << str4 << endl;

    //string& assgin(const char *s, int n);     //把当前s的前n个字符赋给当前的字符串
    string str5;
    str5.assign("hello C++", 5);
    cout << "str5 = " << str5 << endl;

    //string& assgin(const string &s);          //把字符串s赋给当前字符串
    string str6;
    str6.assign(str5);
```

```
    cout << "str6 = " << str6 << endl;

    //string& assgin(int n, char c);            //把n个字符c赋给当前字符串
    string str7;
    str7.assign(10, 'w');
    cout << "str7 = " << str7 << endl;
}

int main()
{
    test01();
    return 0;
}
```

3.1.4 string字符串拼接

功能实现：

- 实现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 );` //字符串中从pos开始的n个字符连接到字符串结尾

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

//string字符串拼接操作

void test01()
{
    //string& operator+=(const char* str);                //重载+=操作符
    string str1 = "我";
    str1 += "爱玩游戏";
    cout << "str1 = " << str1 << endl;

    //string& operator+=(const char c);                   //重载+=操作符
    str1 += ": ";
    cout << "str1 = " << str1 << endl;

    //string& operator+=(const string& str);              //重载+=操作符
    string str2 = "LOL和DNF";
    str1 += str2;
    cout << "str1 = " << str1 << endl;
```

```
    //string& append(const char *s);                    //把字符串s连接到当前字符串结尾
    string str3 = "I";
    str3.append(" LOVE ");
    cout << "str3 = " << str3 << endl;

    //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
    str3.append("GAME, ABCD", 4);
    cout << "str3 = " << str3 << endl;

    //string& append(const string &s);                  //同operator+=(const string& str)
    str3.append(str2);
    cout << "str1 = " << str3 << endl;

    //string& append(const string &s, int pos, int n );  //字符串中从pos开始的n个字符连接到字符串结尾
    string str4 = "Hello C++";
    str3.append(str4, 6, 3);
    cout << "str1 = " << str3 << endl;
}

int main()
{
    test01();
    return 0;
}
```

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：在指定位置替换字符串

函数原型：

- `int find(const string& str, int pos = 0) cosnt;` //查找str第一次出现的位置，从pos开始查找
- `int find(const char * s, int pos = 0) cosnt;` //查找s第一次出现的位置，从pos开始查找
- `int find(const char * s, int pos, int n) cosnt;` //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) cosnt;` //查找字符c第一次出现的位置
- `int rfind(const string& str, int pos = npos) cosnt;` //查找str最后一次出现的位置，从pos开始查找
- `int rfind(const char * s, int pos = npos) cosnt;` //查找s最后一次出现的位置，从pos开始查找
- `int rfind(const char * s, int pos, int n) cosnt;` //从pos位置查找s的前n个字符最后一次位置
- `int rfind(const char c, int pos = 0) cosnt;` //查找字符c最后一次出现的位置

- `string& replace(int pos, int n, const string& str);     //替换从pos开始n个字符为字符串str`
- `string& replace(int pos, int n, const char *s);     //替换从pos开始的n个字符为字符串s`

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

//字符串查找和替换

//1、查找
void test01()
{
    //int find(const string& str, int pos = 0) cosnt;          //查找str第一次出现的位置，从pos开始查找`
    string str1 = "abcdefgde";
    int pos = str1.find("de");
    if (pos == -1)
    {
        cout << "未找到字符串" << endl;
    }
    else
    {
        cout << "find查找位置pos为：" << pos << endl;
    }

    //find和rfind的区别
    //rfind从右往左查找，find从左往右查找
    //int rfind(const string& str, int pos = npos) cosnt;     //查找str最后一次出现的位置，从pos开始查找
    pos = str1.rfind("de");
    cout << "rfind查找位置pos为：" << pos << endl;
}

//2、替换
void test02()
{
    //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 str2 = "abcdefgde";
    //从1号位置起3个字符，替换成"1111"
    str2.replace(1, 3, "rrrr");
    cout << "str2替换后为: " << str2 << endl;

    string str3 = "c";
    str2.replace(6, 2, str3);
    cout << "str2替换后为: " << str2 << endl;
}
int main()
{
    test01();
    test02();
    return 0;
}
```

总结：

- find查找是从左往右，rfind查找是从右往左
- find找到字符字符串后返回查找的第一个字符位置（从0开始计数），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：

- 字符串比较是按字符的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`

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

//字符串比较，逐个字符的比较

void test01()
{
    //string str1 = "hello";
    string str1 = "xello";
    //string str2 = "hello";
    string str2 = "zello";

    if (str1.compare(str2) == 0)
    {
        cout << "str1等于str2" << endl;
    }
    else if (str1.compare(str2) > 0)
    {
        cout << "str1大于str2" << endl;
    }
    else
    {
        cout << "str1小于str2" << endl;
    }
}

int main()
{
    test01();
    return 0;
}
```

总结：字符串比较主要是用于比较两个字符串是否相等，判断谁大谁小的意义不大。

3.1.7 string字符存取

string中单个字符存取方式有两种：

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

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

//string字符存取
void test01()
{
    string str = "hello";
    cout << "str = " << str << endl;

    //char& operator[](int n);       //通过[]方式获取字符
    //修改字符
    str[3] = 'a';
    for (int i = 0; i< str.size(); i++)
    {
        cout << str[i] << " ";
    }
    cout << endl;

    //char& at(int);                 //通过at方法获取字符
    //修改字符
    str.at(3) = 'b';
    for (int i = 0; i < str.size(); i++)
    {
        cout << str.at(i) << " ";
    }
    cout << endl;
}

int main()
{
    test01();
    return 0;
}
```

注意：下标类似数组，从0开始计数。

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

函数原型：

- `string& insert(int pos, const char* s);` //插入字符串
- `string& insert(int pos, const string& str);` //插入字符串
- `string& insert(int pos, int n, char c);` //在指定位置插入n个字符c

- `string& erase(int pos, int n = npos);` //删除从pos开始的n个字符

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

//字符串 插入和删除
void test01()
{
    string str = "hello";

    //string& insert(int pos, const char* s);                //插入字符串
    str.insert(1, "111");
    cout << "str = " << str << endl;

    //string& erase(int pos, int n = npos);                  //删除从pos开始的n个字符
    str.erase(1, 3);
    cout << "str = " << str << endl;
}

int main()
{
    test01();
    return 0;
}
```

注意：下标类似数组，从0开始计数。

3.1.9 string子串

功能描述：

- 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

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

//stirng子串
void test01()
{
    //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
    string str = "abcdef";
    string subStr = str.substr(1, 3);
    cout << "substr = " << subStr << endl;
}
//实用操作
void test02()
{
    string email = "zhangsan@sina.com";
    //从邮件地址中获取用户名信息
    int pos = email.find('@');
```

```
    string userName = email.substr(0,pos);
    cout << "用户名为: " << userName << endl;
}

int main()
{
    //test01();
    test02();
    return 0;
}
```

总结：灵活的运用求子串功能，可以在实际开发中获取到有用的信息

3.2 vector容器

3.2.1 vector基本概念

功能：

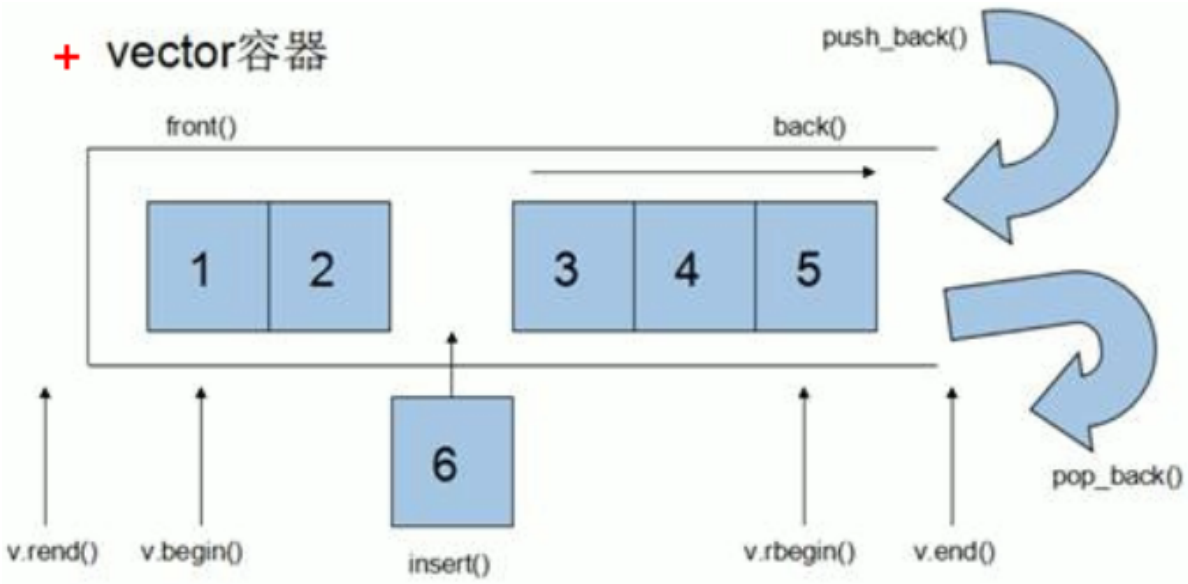
- vector数据结构和**数组非常相似**，也称为**单端数组**

vector与普通数组的区别：

- 不同之处在于数组是静态空间，而vector可以**动态扩展**

动态扩展：

- 并不是在原空间之后续接新空间，而是找更大的空间，然后将原数据拷贝新空间，释放原空间。



- vector容器的迭代器是支持随机访问的迭代器

3.2.2 vector构造函数

功能描述：

- 创建vector容器

函数原型：

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

```
#include<iostream>
using namespace std;
#include<vector>
//vector容器构造
void printVector(vector<int> &v)
{
    for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;
}

void test01()
{
    //vector<T> v;                    //采用模板实现类实现，默认构造函数
    vector<int> v1; //默认构造，无参构造
    for (int i = 0; i < 10; i++)
    {
        v1.push_back(i);
    }
    printVector(v1);

    //vector(v.begin(), v.end());    //将v[begin(), end())区间中的元素拷贝给本身
    //通过区间方式进行构造
    vector<int>v2(v1.begin(), v1.end());
    printVector(v2);

    //vector(n, elem);               //构造函数将n个elem拷贝给本身。
    //n个elem方式构造
    vector<int>v3(10, 100); //10个100
    printVector(v3);

    //vector(const vector &vec);     //拷贝构造函数。
    //拷贝构造
    vector<int>v4(v3);
    printVector(v4);
}

int main()
{
    test01();
    return 0;
}
```

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(beg,end);  //将[beg, end) 区间中的数据拷贝赋值给本身`
- `assign(n,elem);  //将n个elem拷贝赋值给本身。`

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

//vector赋值
void printVector(vector<int>& v)
{
    for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;
}

void test01()
{
    vector<int> v1;
    for (int i = 0; i < 10; i++)
    {
        v1.push_back(i);
    }
    printVector(v1);

    //vector& operator=(const vector& vec);    //重载等号操作符
    vector<int> v2;
    v2 = v1;
    printVector(v2);

    //assign(beg,end);                         //将[beg, end) 区间中的数据拷贝赋值给
本身
    vector<int> v3;
    v3.assign(v1.begin(), v1.end());
    printVector(v3);

    //assign(n,elem);                          //将n个elem拷贝赋值给本身。
    vector<int> v4;
    v4.assign(10, 100); //10个100
    printVector(v4);
}

int main()
{
    test01();
    return 0;
}
```

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会被删除
  ```
- ```
  reszie(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会被删除
  ```

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

//vector容器的大小和操作
void printVector(vector<int>& v)
{
    for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;
}

void test01()
{
    vector<int> v1;
    for (int i = 0; i < 10; i++)
    {
        v1.push_back(i);
    }
    printVector(v1);

    //empty();                    //判断容器是否为空
    if (v1.empty()) //为真，代表容器为空
    {
        cout << "容器为空" << endl;
    }
    else
    {
        cout << "容器不为空" << endl;
        //capacity();               //容器的容量
```

```
        cout << "容器容量为：" << v1.capacity() << endl; //容器容量一般大于等于容器大小，容量值由底层代码决定
        //size();                    //返回容器元素的个数
        cout << "容器大小为：" << v1.size() << endl;
    }

    //重新指定容器大小
    //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会被删除
    v1.resize(13);
    printVector(v1);  //默认用0填充多余的位置
    v1.resize(15, 100);  //函数重载，用100填充多余的位置
    printVector(v1);
    cout << "容器容量为：" << v1.capacity() << endl;
    v1.resize(5);
    printVector(v1);
    cout << "容器容量为：" << v1.capacity() << endl;

    //resize增加容器长度，会使容器的容量值也跟着增加
    //resize减小容器长度，容器的容量值不受影响。
}

int main()
{
    test01();
    return 0;
}
```

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();` //删除所有元素

```
#include<iostream>
using namespace std;
#include<vector>
//vector插入和删除
void printVector(vector<int>& v)
{
    for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
    {
```

```
        cout << *it << " ";
    }
    cout << "vector " << endl;
}

void test01()
{
    vector<int> v1;
    for (int i = 1; i <= 5; i++)
    {
    //push_back(ele);                                   //尾部插入元素ele
        v1.push_back(i * 10); //尾插数据
    }
    printVector(v1);
    //pop_back();                                       //删除最后一个元素
    v1.pop_back(); //尾删
    printVector(v1);

    //插入，第一个参数迭代器
    //insert(const_iterator pos, ele);                  //迭代器指向位置pos插入元素ele
    v1.insert(v1.begin(), 100);
    printVector(v1);

    //insert(const_iterator pos, int count, ele);   //迭代器指向位置pos插入count个元素ele
    v1.insert(v1.begin(), 2, 200);
    printVector(v1);

    //erase(const_iterator pos);                        //删除迭代器指向的元素
    v1.erase(v1.begin());//删除
    printVector(v1);

    //清空操作
    //erase(const_iterator start, const_iterator end);    //删除迭代器从start到end之间的元素
    //v1.erase(v1.begin(), v1.end());

    //clear();                                          //删除所有元素
    v1.clear();
    printVector(v1);
}

int main()
{
    test01();
    return 0;
}
```

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();` //返回容器中第一个元素数据
- `back();` //返回容器中最后一个数据元素

```
#include<iostream>
using namespace std;
#include<vector>
//vectors容器数据存取

void test01()
{
    vector<int>v1;
    for (int i = 0; i < 10; i++)
    {
        v1.push_back(i);
    }

    //利用[]方式访问数组中元素
    //operator[ ];        //返回索引idx所指的数据
    for (int i = 0;i<v1.size(); i++)
    {
        cout << v1[i] << " ";
    }
    cout << endl;

    //利用at方式访问元素
    //at(int idx);        //返回索引idx所指的数据
    for (int i = 0; i < v1.size(); i++)
    {
        cout << v1.at(i) << " ";
    }
    cout << endl;

    //获取第一个元素
    //front();            //返回容器中第一个元素数据
    cout << "容器第一个元素为: " << v1.front() << endl;

    //获取最后一个元素
    //back();             //返回容器中最后一个数据元素
    cout << "容器最后一个元素为: " << v1.back() << endl;
}

int main()
{
    test01();
    return 0;
}
```

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与本身的元素进行互换`

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

//vector容器互换
void printVector(vector<int> &v)
{
    for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;
}

//1、基本使用
void test01()
{
    cout << "-----------交换前---------" << endl;
    vector<int>v1;
    for (int i = 0; i < 10; i++)
    {
        v1.push_back(i);
    }
    cout << "v1为：";
    printVector(v1);


    vector<int>v2;
    for (int i = 0; i < 10; i++)
    {
        v2.push_back(10-i);
    }
    cout << "v2为：";
    printVector(v2);

    cout << endl;
    v1.swap(v2);
    cout << "-----------交换后---------" << endl;
    cout << "v1为：";
    printVector(v1);
    cout << "v2为：";
    printVector(v2);
}

//2、实际用途
//巧用swap可以收缩内存空间
void test02()
{
```

```
    vector<int> v;
    for (int i = 0; i < 100000; i++)
    {
        v.push_back(i);
    }
    cout << "容器v的容量为: " << v.capacity() << endl;
    cout << "容器v的大小为: " << v.size() << endl;

    //重新v的指定大小，大小改变，容量不变
    cout << "-----------重新v的指定大小---------" << endl;
    v.resize(3);
    cout << "容器v的容量为: " << v.capacity() << endl;
    cout << "容器v的大小为: " << v.size() << endl;

    //巧用swap收缩内存
    cout << "-----------巧用swap收缩内存---------" << endl;
    vector<int>(v).swap(v);  //vecor<int>(v) 匿名对象
                             //用v当前所用元素个数来初始化匿名对象，匿名对象会自动销毁
    cout << "容器v的容量为: " << v.capacity() << endl;
    cout << "容器v的大小为: " << v.size() << endl;
}

int main()
{
    //test01();
    test02();
    return 0;
}
```

总结：swap可以使两个容器互换，可以达到实用的收缩空间内存的效果

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`

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

//vector容器 预留空间
void test01()
{
    vector<int>v1;

    //利用reserve预留空间
    v1.reserve(100000);
    int num = 0;//统计开辟空间次数
    int* p = NULL;
    for (int i = 0; i < 100000; i++)
    {
```

```
            v1.push_back(i);
            if (p != &v1[0]) //&v1[0] 容器的首地址
            {
                p = &v1[0];
                num++;
            }
        }
        cout << "num = " << num << endl;  //没用reserve，开辟空间次数为30，使用reserve预留空间后，开辟空间次数为1
}

int main()
{
    test01();
    return 0;
}
```

总结：如果数据量比较大，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

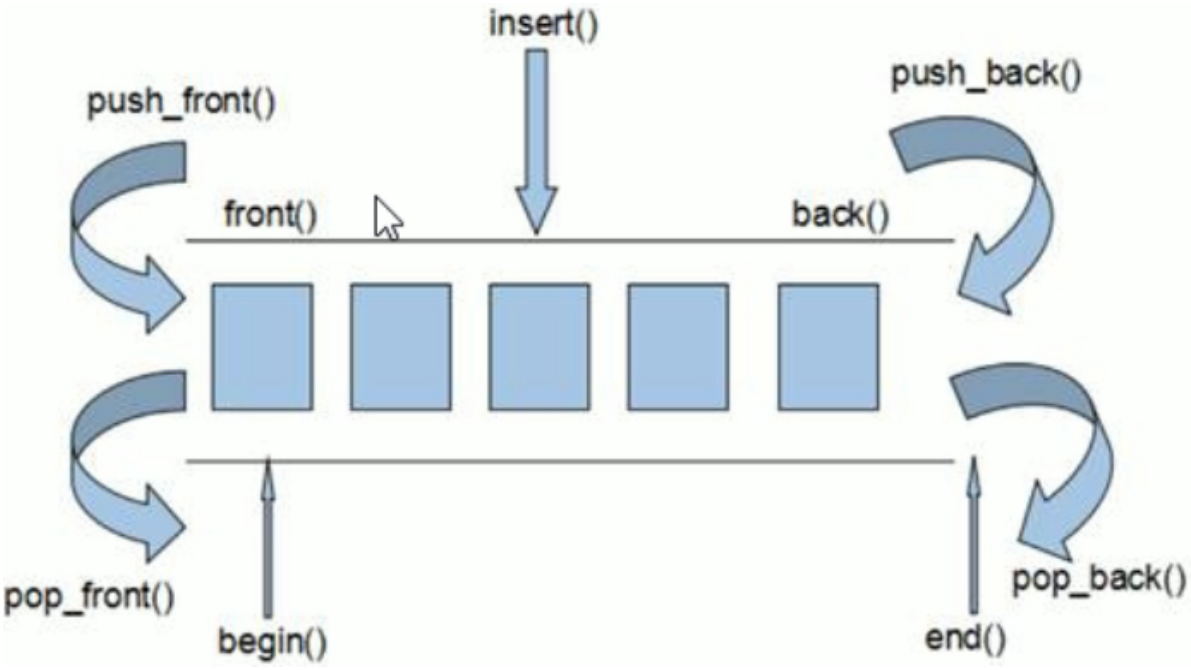
功能：

- 双端数组，可以对头端进行插入删除操作

deque与vector区别：

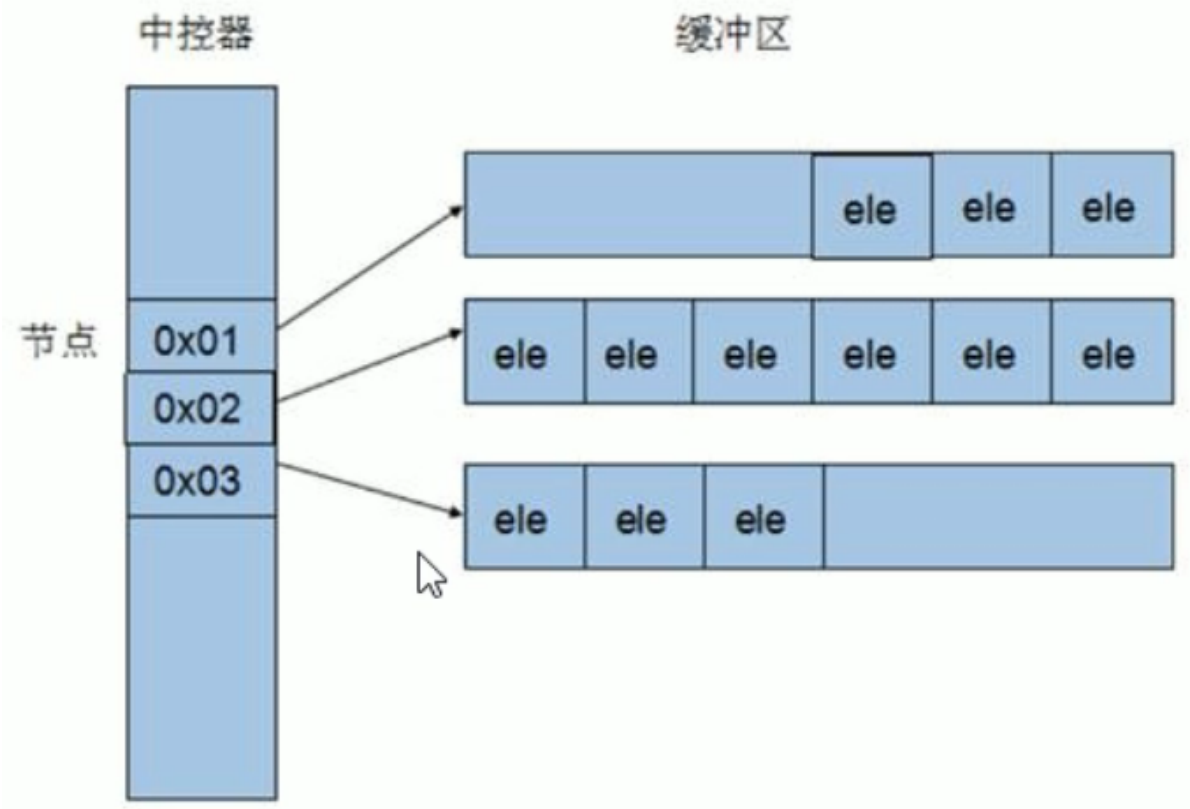
- vector对于头部的插入删除效率低，数据量越大，效率越低。
- deque相对而言，对头部的插入删除速度会比vector快
- vector访问元素时的速度会比deque快，这和两者内部实现有关

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，缓冲区存放真实数据

中控器维护的是每个缓存区的地址，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。

3.3.2 deque构造函数

功能描述：

- deque容器构造

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

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

//deque 构造函数
void printDeque(const deque<int> &d) //加cosnt 防止被修改
{
    for (deque<int>::const_iterator it = d.begin(); it != d.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;
}

void test01()
```

```
{
    //deque<T> deqT;             //默认构造形式
    deque<int> d1;
    for (int i = 0; i < 10; i++)
    {
        d1.push_back(i);
    }
    printDeque(d1);

    //deque(beg, end);           //构造函数将[beg, end)区间中的元素拷贝给本身
    deque<int>d2(d1.begin(), d1.end());
    printDeque(d2);

    //deque(n, elem);             //构造函数将n个elem拷贝给本身
    deque<int>d3(10, 100);
    printDeque(d3);

    //deque(const deque &deq);       //拷贝构造函数
    deque<int>d4(d3);
    printDeque(d4);
}

int main()
{
    test01();
    return 0;
}
```

3.3.3 deque赋值操作

功能描述：

- 给deque容器进行赋值

函数原型：

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

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

//deque容器赋值操作
void printDeque(const deque<int>& d)
{
    for (deque<int>::const_iterator it = d.begin(); it != d.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;
}

void test01()
{
```

```
    deque<int>d1;
    for (int i = 0; i < 10; i++)
    {
        d1.push_back(i);
    }
    printDeque(d1);

    //deque& operator=(const deque &deq); //重载等号操作
    deque<int>d2;
    d2 = d1;
    printDeque(d2);

    //assign(beg, end);       //将[beg,end)区间中的数据拷贝赋值给本身
    deque<int>d3;
    d3.assign(d1.begin(), d1.end());
    printDeque(d3);

    //assign(n, elem);          //将n个elem拷贝赋值给本身
    deque<int>d4;
    d4.assign(10, 100);
    printDeque(d4);
}

int main()
{
    test01();
    return 0;
}
```

3.3.4 deque大小操作

功能描述：

- 对deque容器的大小进行操作

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.reszie(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。 //默认值一般为0
  //如果容器变短，则末尾超出容器长度的元素会被删除
- `deque.reszie(num, elem);` //重新指定容器的长度为num，若容器变长，则以elem填充新位置。
  //如果容器变短，则末尾超出容器长度的元素会被删除

```
#include<iostream>
#include<deque>
using namespace std;
//deque容器大小操作
void printDeque(const deque<int>& d)
{
    for (deque<int>::const_iterator it = d.begin(); it != d.end(); it++)
```

```
    {
        cout << *it << " ";
    }
    cout << endl;
}

void test01()
{
    deque<int> d1;
    for (int i = 0; i < 10; i++)
    {
        d1.push_back(i);
    }
    printDeque(d1);

    if (d1.empty())
    {
        cout << "d1为空" << endl;
    }
    else
    {
        cout << "d1不为空" << endl;
        cout << "d1的大小为：" << d1.size() << endl;
        //deque容器没有容量概念
    }
    //d1.resize(15);//默认用0填充
    d1.resize(15, 11);
    printDeque(d1);

    d1.resize(5);
    printDeque(d1);
}

int main()
{
    test01();
    return 0;
}
```

3.3.5 deque插入和删除

功能描述：

- 向deque容器中插入和删除数据

函数原型：

两端插入操作：

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

指定位置操作：

- `insert(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

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

void printDeque(const deque<int> d)
{
    for (deque<int>::const_iterator it = d.begin(); it != d.end(); it++)
    {
        cout << *it << " ";
    }
    cout <<" deque " << endl;
}

void test01()
{
    deque<int> d1;

    //尾插
    d1.push_back(10);
    d1.push_back(20);

    //头插
    d1.push_front(100);
    d1.push_front(200);

    printDeque(d1);

    //尾删
    d1.pop_back();
    printDeque(d1);

    //头删
    d1.pop_front();
    printDeque(d1);

}

void test02()
{
    deque<int>d1;
    d1.push_back(10);
    d1.push_back(20);
    d1.push_front(100);
    d1.push_front(200);
    printDeque(d1);

    //insert插入
    d1.insert(d1.begin(), 1000);
    printDeque(d1);
```

```
    d1.insert(d1.begin(), 2, 10000);
    printDeque(d1);

    //按照区间进行插入
    deque<int>d2;
    d2.push_back(1);
    d2.push_back(2);
    d2.push_back(3);

    d1.insert(d1.begin(), d2.begin(), d2.end());
    printDeque(d1);
}

void test03()
{
    deque<int>d1;
    d1.push_back(10);
    d1.push_back(20);
    d1.push_front(100);
    d1.push_front(200);

    //删除
    deque<int>::iterator it = d1.begin();
    it++;
    d1.erase(it);
    printDeque(d1);

    //按区间删除
    //d1.erase(d1.begin(), d1.end());
    //清空
    d1.clear();
    printDeque(d1);
}

int main()
{
    //test01();
    //test02();
    test03();
    return 0;
}
```

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`

```
#include<iostream>
```

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

//deque容器数据存取

void test01()
{
    deque<int> d;
    d.push_back(10);
    d.push_back(20);
    d.push_back(30);
    d.push_front(100);
    d.push_front(200);
    d.push_front(300);

    //通过[]方式访问元素
    for (int i = 0; i < d.size(); i++)
    {
        cout << d[i] << " ";
    }
    cout << endl;

    //通过at方式访问元素
    for (int i = 0; i < d.size(); i++)
    {
        cout << d.at(i) << " ";
    }
    cout << endl;

    cout << "第一个元素为: " << d.front() << endl;
    cout << "最后一个元素为: " << d.back() << endl;
}

int main()
{
    test01();
    return 0;
}
```

3.3.7 deque排序

功能描述:

- 利用算法实现对deque容器进行排序

函数原型:

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

```cpp
#include<iostream>
#include<deque>
#include<algorithm> #标准算法头文件
using namespace std;

void printDeque(const deque<int> &d)
{
```

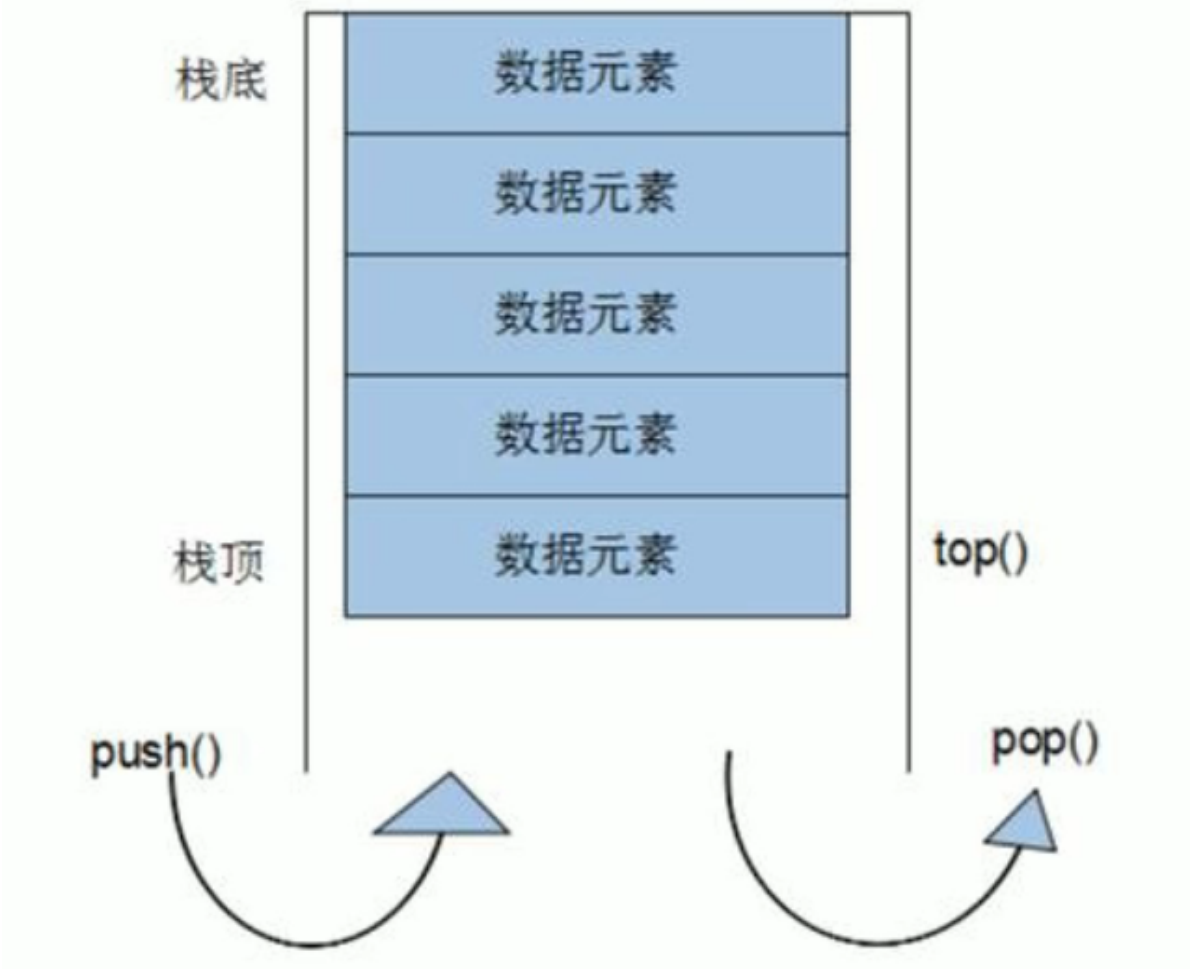
```
    for (deque<int>::const_iterator it = d.begin(); it != d.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;
}
//deque容器排序
void test01()
{
    deque<int>d;
    d.push_back(30);
    d.push_back(10);
    d.push_back(20);
    d.push_back(100);
    d.push_back(300);
    d.push_back(200);
    printDeque(d);

    //排序 默认排序规则 从小到大 升序
    //对于支持随机访问的迭代器的容器，都可以利用sort算法直接对其进行排序
    //vector容器也可以利用，sort进行排序
    sort(d.begin(), d.end());
    cout << "从小到大排序后：" << endl;
    printDeque(d);
}

int main()
{
    test01();
    return 0;
}
```

总结：sort算法非常实用，使用时包含头文件algorithm即可

3.4 stack容器

3.4.1 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`

生活中的栈的应用：手枪弹匣

3.4.2 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

```cpp
#include<iostream>
#include<stack>
using namespace std;
//栈stack容器
void test01()
{
    //特点：符合先进后出数据结构
    stack<int>s;
    s.push(10);
    s.push(20);
    s.push(30);
    s.push(40);
    cout << "栈的大小：" << s.size() << endl;
    while (!s.empty())
    {
        cout << "栈顶元素为：" << s.top() << endl;
        s.pop();
    }
    cout << "栈的大小：" << s.size() << endl;
}

int main()
{
    test01();
    return 0;
}
```

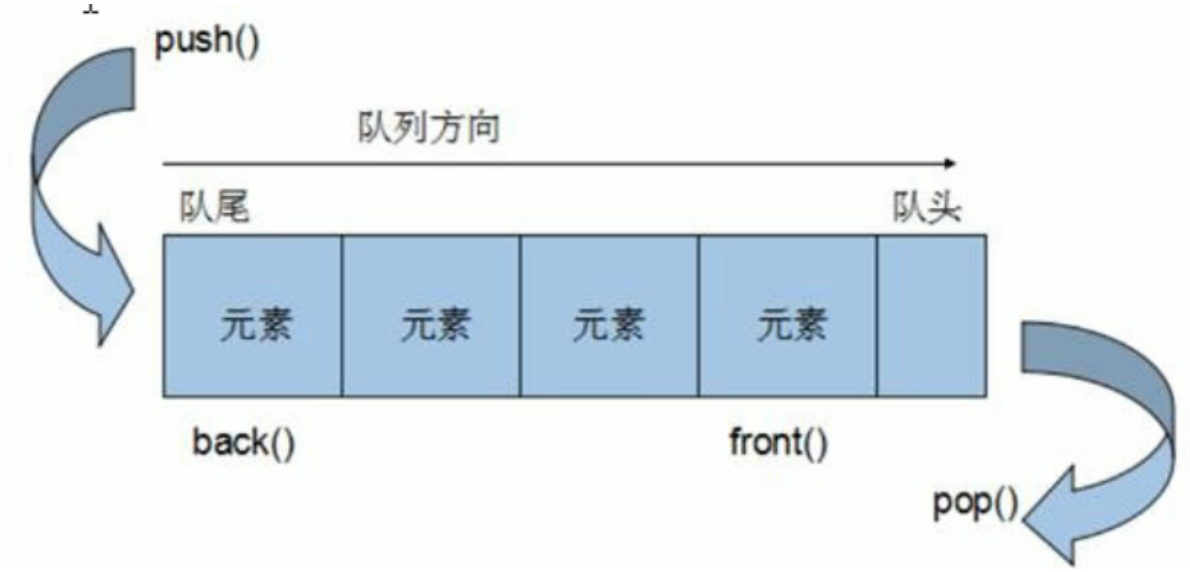
总结：

- 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

3.5 queue容器

3.5.1 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



队列容器允许一端新增数据，从另一端移除数据

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`

3.5.2 queue 常用接口

功能描述：队列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);` //向队尾添加元素
- `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();` //返回栈的大小

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

//队列 Queue

class Person
{
public:
```

```
    Person(string name, int age)
    {
        this->m_name = name;
        this->m_age = age;
    }
    string m_name;
    int m_age;
};

void test01()
{
    //创建队列
    queue<Person>q;
    Person p1("张三", 16);
    Person p2("李四", 17);
    Person p3("王五", 18);
    Person p4("赵六", 19);
    //队列
    q.push(p1);
    q.push(p2);
    q.push(p3);
    q.push(p4);

    //判断只要队列不为空，查看队头，查看队尾，出队
    while (!q.empty())
    {
        //查看 队头
        cout << "队头元素--姓名：" << q.front().m_name << "，年龄：" <<
q.front().m_age << endl;

        //查看 队尾
        cout << "队头元素--姓名：" << q.back().m_name << "，年龄：" <<
q.back().m_age << endl;

        q.pop();
    }
    cout << "队列大小" << q.size() << endl;
}

int main()
{
    test01();
    return 0;
}
```

3.6 list容器

3.6.1 list基本概念

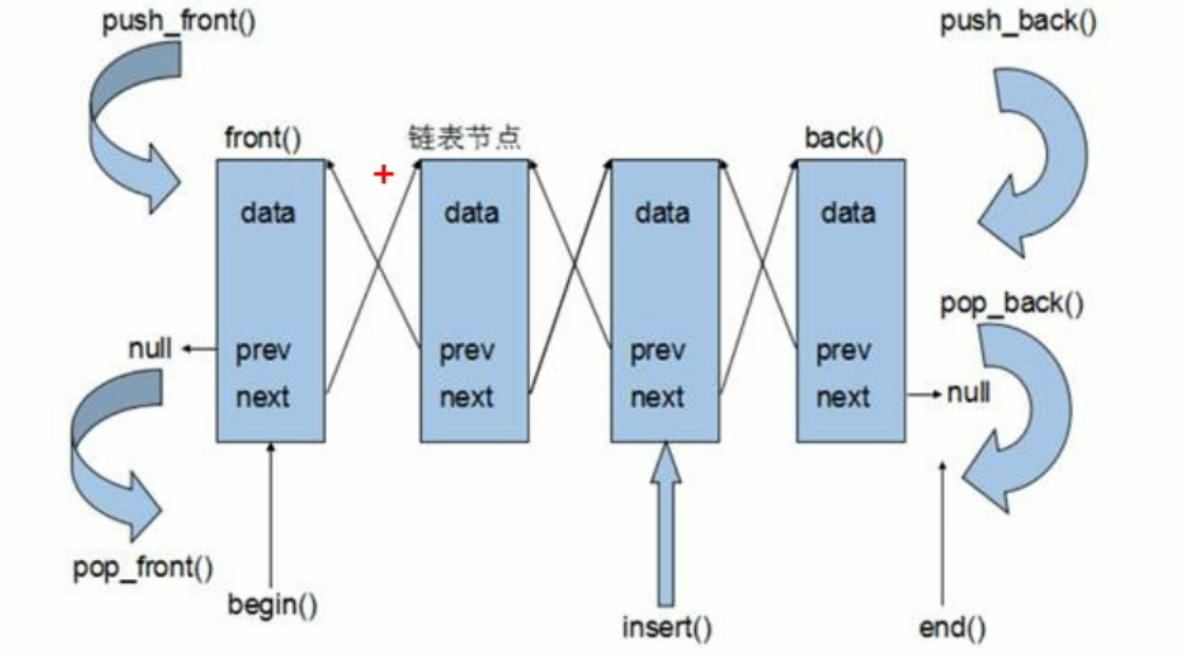
功能：将数据进行链式存储

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连边中的指针链接实现的

链表的组成：链表由一系列组成

结点的组成：一个是存储数据单元的数据域，另一个是存储下一个结点地址的指针域

STL中的链表是一个双向循环链表



由于链表的存储方式并不是连续的内存空间，因此链表中list中的迭代器只支持前移和后移，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移除
- 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）额外消耗较大

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

3.6.2 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

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

```
//list构造函数
void printList(const list<int>& l)
{
    for (list<int>::const_iterator lt = l.begin(); lt != l.end(); lt++)
    {
        cout << *lt << " ";
    }
    cout << endl;
}

void test01()
{
    //默认构造List容器
    list<int>L1;
    L1.push_back(10);
    L1.push_back(20);
    L1.push_back(30);
    L1.push_back(40);

    printList(L1);

    //区间方式构造
    list<int>L2(L1.begin(), L1.end());
    printList(L2);

    //拷贝构造
    list<int>L3(L2);
    printList(L3);

    //n个elem构造
    list<int>L4(10, 1000);
    printList(L4);
}

int main()
{
    test01();
    return 0;
}
```

3.6.3 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拷贝给赋值给本身
- `list& operator=(const list &lst);` //重载等号操作符
- `swap(lst);` //将lst与本身的元素进行互换

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

```
//list赋值和交换

void printList(const list<int>& l)
{
    for (list<int>::const_iterator lt = l.begin(); lt != l.end(); lt++)
    {
        cout << *lt << " ";
    }
    cout << endl;
}

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

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

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

    list<int>L4;
    L4.assign(10, 100);
    printList(L4);

}


void test02()
{
    list<int>L1;
    L1.push_back(10);
    L1.push_back(20);
    L1.push_back(30);
    L1.push_back(40);

    list<int>L2;
    L2.assign(10, 100);

    cout << "交换前------------" << endl;
    printList(L1);
    printList(L2);
    swap(L1, L2);
    cout << "交换后------------" << endl;
    printList(L1);
    printList(L2);

}

int main()
```

```
{
    //test01();
    test02();
    return 0;
}
```

3.6.4 list大小操作

功能描述：

- 对list容器的大小进行操作

函数原型：

- `empty();` //判断容器是否为空
- `size();` //返回容器元素的个数
- `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会被删除
- `reszie(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会被删除

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

void printList(const list<int>& L)
{
    for (list<int>::const_iterator lt = L.begin(); lt != L.end(); lt++)
    {
        cout << *lt << " ";
    }
    cout << endl;
}

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

    //判断容器是否为空
    if (L1.empty())
    {
        cout << "L1为空" << endl;
    }
    else
```

```
    {
        cout << "L1不为空" << endl;
        cout << "L1的元素个数为：" << L1.size() << endl;
    }
    L1.resize(10, 100);
    printList(L1);

    L1.resize(2);
    printList(L1);

}

int main()
{
    test01();
    return 0;
}
```

3.6.5 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位置插入elem元素的拷贝，返回新数据的位置
- `insert(pos, n, elem);` //在pos位置插入n个elem数据，无返回值
- `insert(pos, beg, end);` //在pos位置插入[beg, end)区间的数据，无返回值
- `clear();` //移除容器的所有数据
- `erase(beg, end);` //删除[beg, end)区间的数据，返回下一个数据的位置
- `erase(pos);` //删除pos位置的数据，返回下一个数据的位置
- `remove(elem);` //删除容器中所有与elem值匹配的元素

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

//list插入和删除
void printList(const list<int>& L)
{
    for (list<int>::const_iterator lt = L.begin(); lt != L.end(); lt++)
    {
        cout << *lt << " ";
    }
    cout <<"-list" << endl;
}

void test01()
{
```

```
    list<int>l;
    //尾插
    l.push_back(10);
    l.push_back(20);
    l.push_back(30);
    //头插
    l.push_front(100);
    l.push_front(200);
    l.push_front(300);
    //300 200 100 10 20 30
    printList(l);

    //尾删
    l.pop_back();
    //300 200 100 10 20
    printList(l);

    //头删
    l.pop_front();
    //200 100 10 20
    printList(l);

    //insert插入
    list<int>::iterator it = l.begin();
    l.insert(++it, 1000);
    //200 1000 100 10 20
    printList(l);

    //删除
    it = l.begin();
    l.erase(++it);
    //200 100 10 20
    printList(l);

    //移除
    l.push_back(10);
    l.remove(10);
    //200 100 20
    printList(l);

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

int main()
{
    test01();
    return 0;
}
```

3.6.6 list存取

功能描述：

- 对list容器中数据机械能存取

函数原型：

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

```
#include<iostream>
#include<list>
using namespace std;
//list容器 数据存取
void test01()
{
    list<int>l1;
    l1.push_back(10);
    l1.push_back(20);
    l1.push_back(30);
    l1.push_back(40);


    //l1[0]    不可以用[]访问list容器中的元素

    //l1.at(0) 不可以用at方式访问list容器中的元素

    //原因是list本质是链表，不是用连续线性空间存储数据，迭代器也是不支持随机访问的

    cout << "第一个元素为：" << l1.front() << endl;
    cout << "最后一个元素为：" << l1.back() << endl;

    //验证迭代器是不支持随机访问的
    list<int>::iterator it = l1.begin();
    it++;
    it--; //支持双向
    //it = it + 1; 报错

}

int main()
{
    test01();
    return 0;
}
```

3.6.7 list反转和排序

功能排序：

- 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`

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

void test01()
{
    //反转
    list<int>l1;
    l1.push_back(10);
    l1.push_back(20);
    l1.push_back(30);
    l1.push_back(40);

    cout << "反转前---------" << endl;
    printList(l1);
    l1.reverse();
    cout << "反转后---------" << endl;
    printList(l1);
}

bool myCompare(int v1,int v2)
{
    //降序 就让第一个数 > 第二个数
    return v1 > v2;
}

void test02()
{

    //排序
    list<int>l1;
    l1.push_back(90);
    l1.push_back(20);
    l1.push_back(50);
    l1.push_back(30);
    l1.push_back(40);

    cout << "排序前---------" << endl;
    printList(l1);
    //所有不支持随机访问迭代器的容器，不可以用标准算法
    // 不支持随机访问迭代器的容器，内部会提供对应的一些算法
    //sort(l1.begin(),l1.end()); 报错
    //l1.sort(); //默认升序
    l1.sort(myCompare);
    cout << "排序后---------" << endl;
    printList(l1);
}
```

```
int main()
{
    //test01();
    test02();
    return 0;
}
```

3.7 set/ multiset 容器

3.7.1 set基本概念

简介：

- 所有元素都会在在插入时自动被排序

本质：

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

set和multiset区别：

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

3.7.2 set构造和赋值

功能描述：创建set容器以及赋值

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=(cosnt set& st);　　//重载等号操作符`

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

//set容器构造和赋值
void printSet(const set<int>& s)
{
    cout << "set- ";
    for (set<int>::const_iterator st = s.begin(); st != s.end(); st++)
    {
        cout << *st << " ";
    }
    cout << endl;
}
```

```
void test01()
{
    set<int>s1;

    //插入数据，只有insert方式
    s1.insert(10);
    s1.insert(40);
    s1.insert(20);
    s1.insert(50);
    s1.insert(30);
    s1.insert(40);


    //遍历容器
    //set容器特点：1、所有元素插入时自动被排序
    //            2、set容器不允许插入重复值
    printSet(s1);

    //拷贝构造
    set<int>s2(s1);
    printSet(s2);

    //赋值操作
    set<int>s3;
    s3 = s2;
    printSet(s3);
}

int main()
{
    test01();
    return 0;
}
```

3.7.3 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`

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

//set大小和交换
void printSet(const set<int>& s)
{
    cout << "set- ";
    for (set<int>::const_iterator st = s.begin(); st != s.end(); st++)
    {
```

```
        cout << *st << " ";
    }
    cout << endl;
}

void test01()
{
    set<int>s1;
    s1.insert(10);
    s1.insert(30);
    s1.insert(20);
    s1.insert(40);

    set<int>s2;
    s2.insert(100);
    s2.insert(300);
    s2.insert(200);
    s2.insert(400);

    if (s1.empty())
    {
        cout << "s1为空" << endl;
    }
    else
    {
        cout << "s1不为空" << endl;
        cout << "s1的大小为：" << s1.size() << endl;
    }

    cout << "交换前------------" << endl;
    printSet(s1);
    printSet(s2);

    s1.swap(s2);
    cout << "交换前后-----------" << endl;
    printSet(s1);
    printSet(s2);
}

int main()
{
    test01();
    return 0;
}
```

3.7.4 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();` //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

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

//set 插入和删除
void printSet(const set<int>& s)
{
    cout << "set- ";
    for (set<int>::const_iterator st = s.begin(); st != s.end(); st++)
    {
        cout << *st << " ";
    }
    cout << endl;
}
void test01()
{
    set<int> s1;
    s1.insert(50);
    s1.insert(30);
    s1.insert(20);
    s1.insert(40);
    printSet(s1);

    //删除
    s1.erase(s1.begin());
    printSet(s1);

    //重载删除
    s1.erase(30);
    printSet(s1);

    //清空
    //s1.erase(s1.begin(), s1.end());
    s1.clear();
    printSet(s1);
}

int main()
{
    test01();
    return 0;
}
```

3.7.5 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，返回该key的元素的迭代器；若不存在，返回`set.end();`

- `count(key);　　//统计key的元素个数`

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

//set查找和统计
void test01()
{
    set<int>s1;
    s1.insert(10);
    s1.insert(20);
    s1.insert(30);
    s1.insert(40);

    set<int>::iterator pos = s1.find(30);
    if (pos != s1.end())
    {
        cout << "找到元素" << *(pos) << endl;
    }
    else
    {
        cout << "未找到元素" << endl;
    }
}

void test02()
{
    set<int>s1;
    s1.insert(10);
    s1.insert(20);
    s1.insert(30);
    s1.insert(40);
    s1.insert(40);

    //对于set而言，统计结果要么是0要么是1，不允许重复数值
    cout << "num = " << s1.count(40) << endl;
}

int main()
{
    test01();
    test02();
    return 0;
}
```

3.7.6 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

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

//set与multiset的区别
void test01()
{
    set<int>s1;
    pair<set<int>::iterator,bool> ret = s1.insert(10);
    if (ret.second)
    {
        cout << "第一次插入成功" << endl;
    }
    else
    {
        cout << "第一次插入失败" << endl;
    }
    ret = s1.insert(10);
    if (ret.second)
    {
        cout << "第二次插入成功" << endl;
    }
    else
    {
        cout << "第二次插入失败" << endl;
    }

    multiset<int>ms;
    //multiset允许插入重复的值
    ms.insert(10);
    ms.insert(10);
    ms.insert(10);
    for (multiset<int>::iterator mt = ms.begin(); mt != ms.end(); mt++)
    {
        cout << *mt << " ";
    }
    cout << endl;

}

int main()
{
    test01();
    return 0;
}
```

3.7.7 pair对组创建

功能描述：

- 成对出现的数据，利用队组可以返回两个数据

两种创建方式：

- `pair<type, type> p ( vlaue1, vlaue2 );`
- `pair<type, type> p = make_pair( value1, value2 );`

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

//pair对组的创建
void test01()
{
    //第一种方式
    pair<string, int>p("Tom", 20);
    cout << "姓名: " << p.first << " 年龄: " << p.second << endl;

    //第二种方式
    pair<string, int>p2 = make_pair("Jerry", 30);
    cout << "姓名: " << p2.first << " 年龄: " << p2.second << endl;
}

int main()
{
    test01();
    return 0;
}
```

3.7.8 set容器排序

学习目标:

- set容器默认规则为从小到大，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

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

//set容器排序
class myCompare
{
public:
     bool operator()(int v1, int v2) const
    {
        return v1 > v2;
    }
};
void test01()
{
    set<int>s1;
    s1.insert(10);
    s1.insert(20);
    s1.insert(40);
```

```
    s1.insert(30);
    s1.insert(50);

    //默认从小到大
    for (set<int>::iterator it = s1.begin(); it != s1.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;

    //指定排序规则为从大到小
    //仿函数本质是类，但是功能上确实方法（函数）
    set<int,myCompare>s2;
    s2.insert(10);
    s2.insert(20);
    s2.insert(40);
    s2.insert(30);
    s2.insert(50);
    for (set<int,myCompare>::iterator it = s2.begin(); it != s2.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;

}

int main()
{
    test01();
    return 0;
}
```

示例二：set存放自定义数据类型

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

//set容器排序，存放自定义数据类型

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

    string m_name;
    int m_age;

};

class MyCompare
{
public:
```

```
    bool operator()(const Person& p1, const Person& p2) const
    {
        //按照年龄降序
        return p1.m_age > p2.m_age;
    }
};

void test01()
{
    //自定义数据类型，都会指定排序规则
    set<Person,MyCompare>s;

    //创建Person对象
    Person p1("张三", 24);
    Person p2("李四", 28);
    Person p3("王五", 25);
    Person p4("赵六", 21);

    s.insert(p1);
    s.insert(p2);
    s.insert(p3);
    s.insert(p4);

    for (set<Person,MyCompare>::iterator it = s.begin(); it != s.end(); it++)
    {
        cout << "姓名: " << it->m_name << " 年龄" << it->m_age << endl;
    }
}

int main()
{
    test01();
    return 0;
}
```

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），起索引作用，第二个元素为value（实值）
- 所有元素都会更具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（红黑树）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`

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

//map构造和赋值
void printMap(map<int, int>& m)
{
    for (map<int, int>::iterator it = m.begin(); it != m.end(); it++)
    {
        cout << "key = " << it->first << " ,value = " << it->second << endl;
    }
    cout << endl;
}
void test01()
{
    map<int, int>mp;  //key-value

    //会根据key值排序
    mp.insert(pair<int, int>(1, 10));
    mp.insert(pair<int, int>(2, 20));
    mp.insert(pair<int, int>(3, 30));
    mp.insert(pair<int, int>(4, 40));
    printMap(mp);

    //拷贝构造
    map<int, int>m2(mp);
    printMap(m2);

    //赋值
    map<int, int>m3;
    m3 = m2;
    printMap(m3);
}

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

```
    return 0;
}
```

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(mp);` //交换两个集合容器

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

void printMap(map<int, int>& m)
{
    cout << endl;
    for (map<int, int>::iterator it = m.begin(); it != m.end(); it++)
    {
        cout << "key = " << it->first << " ,value = " << it->second << endl;
    }
}
void test01()
{
    map<int, int>m;
    m.insert(pair<int, int>(1, 10));
    m.insert(pair<int, int>(2, 20));
    m.insert(pair<int, int>(3, 30));

    if (m.empty())
    {
        cout << "m为空" << endl;
    }
    else
    {
        cout << "m不为空" << endl;
        cout << "m的大小为：" << m.size() << endl;
    }

    map<int, int>m2;
    m2.insert(pair<int, int>(4, 40));
    m2.insert(pair<int, int>(5, 50));
    m2.insert(pair<int, int>(6, 60));

    cout << "交换前--------------" << endl;
    printMap(m);
    printMap(m2);

    m.swap(m2);
```

```
    cout << "交换后--------------" << endl;
    printMap(m);
    printMap(m2);

}

int main()
{
    test01();
    return 0;
}
```

3.9.4map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(pair<type1,type2> (value1, vlaue2) );  //在容器中插入元素`
- `insert(make_pair (value1, vlaue2) );  //在容器中插入元素`
- `clear();  //清除所有元素`
- `erase();  //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`

```
#include<iostream>
#include<map>
using namespace std;
//map插入和删除

void printMap(map<int, int>& m)
{
    cout << "map-"<<endl;

    for (map<int, int>::iterator it = m.begin(); it != m.end(); it++)
    {
        cout << "key = " << it->first << " ,value = " << it->second << endl;
    }
}

void test01()
{
    map<int, int>m;
    //插入，第一种方式
    m.insert(pair<int, int>(4, 40));

    //插入，第二种方式
    m.insert(make_pair(2, 20));

    //插入，第三种
    m.insert(map<int, int>::value_type(3, 30));
```

```
    //第四种，不推荐，若误用[]会创建新键值对，用途是利用key访问到vlaue;
    m[1] = 10;

    //cout << m[5] << endl;
    printMap(m);

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

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

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

int main()
{
    test01();
    return 0;
}
```

3.9.5map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，返回该key的元素的迭代器；若不存在，返回`map.end();`
- `count(key);` //统计key的元素个数

```
#include<iostream>
#include<map>
using namespace std;
//map统计和查找

void test01()
{
    //查找
    map<int, int>m;
    m.insert(make_pair(1, 10));
    m.insert(make_pair(2, 20));
    m.insert(make_pair(3, 30));

    map<int, int>::iterator pos = m.find(3);
    if (pos != m.end())
    {
        cout << "查找了元素 key = " << pos->first << " ,value = " << pos->second
<< endl;
    }
```

```
    else
    {
        cout << "元素未找到" << endl;
    }

    //统计
    cout << "统计num = " << m.count(3) << endl;
}

int main()
{
    test01();
    return 0;
}
```

3.9.6 map容器排序

学习目标：

- map容器默认规则为 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

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

//map自定义排序
class MapCompare //仿函数
{
public:
    bool operator()(int v1, int v2) const
    {
        //降序
        return v1 > v2;
    }
};

void printMap(map<int, int,MapCompare>& m)
{
    cout << " "<<endl;

    for (map<int, int,MapCompare>::iterator it = m.begin(); it != m.end(); it++)
    {
        cout << "key = " << it->first << " ,value = " << it->second << endl;
    }
}

void test01()
{
    map<int, int,MapCompare>m;
    m.insert(make_pair(1, 10));
    m.insert(make_pair(3, 10));
    m.insert(make_pair(4, 10));
```

```
    m.insert(make_pair(5, 10));
    m.insert(make_pair(2, 10));
    printMap(m);
}

int main()
{
    test01();
    return 0;
}
```

4 STL-函数对象

4.1 函数对象

4.1.1 函数对象概念

概念：

- 重载**函数调用操作符**的类，其对象常称为**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，函数对象也可以有自己的状态
- 函数对象可以作为参数传递

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

//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
class MyAdd
{
public:
    int operator()(int v1, int v2)
    {
        return v1 + v2;
    }
};

void test01()
{
    MyAdd myadd;
```

```
    cout << myadd(10 , 20) << endl;
}

//2、函数对象超出普通函数的概念，函数对象也可以有自己的状态
class MyPrint
{
public:
    MyPrint()
    {
        this->count = 0;
    }
    void operator()(string s)
    {
        cout << s << endl;
        this->count++;
    }
    int count;//记录内部自己的状态
};

void test02()
{
    MyPrint myprint;  //只有这里调用了构造函数
    myprint("hello world!");
    myprint("hello world!");

    cout << "mycount调用次数为：" << myprint.count << endl;
}

//3、函数对象可以作为参数传递
void doPrint(MyPrint& mp, string s)
{
    mp(s);
}

void test03()
{
    MyPrint myprint;
    doPrint(myprint, "hello c++!");
}

int main()
{
    test01();
    test02();
    test03();
    return 0;
}
```

4.2 谓词

4.2.1 谓词概念

概念：

- 返回bool类型的仿函数称为谓词
- 如果operator()接受一个参数，那么叫做一元谓词
- 如果operator()接受二个参数，那么叫做二元谓词

4.2.2 一元谓词

```
#include<iostream>
using namespace std;
#include<vector>
//仿函数 返回值类型是bool数据类型，称为谓词
//一元谓词
class FindFive
{
public:
    bool operator()(int v)
    {
        return v > 5;
    }
};
void test01()
{
    vector<int>v;
    for (int i = 0; i < 10; i++)
    {
        v.push_back(i);
    }
    //查找容器中有没有大于5的数字
    //FindFive() 匿名函数对象
    vector<int>::iterator pos = find_if(v.begin(), v.end(), FindFive());
    if ( pos == v.end()) ////返回一个迭代器
    {
        cout << "未找到" << endl;
    }
    else
    {
        cout << "找到了大于5的数字为：" << *pos << endl;
    }
}

int main()
{
    test01();
    return 0;
}
```

4.2.3 二元谓词

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

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

void test01()
{
    vector<int>v;
    v.push_back(10);
    v.push_back(40);
    v.push_back(30);
    v.push_back(50);
    v.push_back(20);
    v.push_back(60);

    sort(v.begin(), v.end());
    for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;

    //使用函数对象 改变算法策略，变为排序规则从大到小
    cout << "降序排列为: " << endl;
    sort(v.begin(), v.end(), MyCompare());
    for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;
}

int main()
{
    test01();
    return 0;
}
```

4.3 内建函数对象

4.3.1 内建函数对象

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，需要引用头文件 `#include<functional>`

4.3.2 算术仿函数

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

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

//内建函数对象 算术仿函数
//negate  一元仿函数 取反仿函数
void test01()
{
    negate<int> n;

    cout << "50取反为: " << n(50)<< endl; //取相反数
}
//plus 二元仿函数 加法
void test02()
{
    plus<int> p;
    cout << p(10, 20) << endl;
}

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

```
    return 0;
}
```

4.3.3 关系仿函数

功能描述：

仿函数原型：

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

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

//内建函数对象 关系仿函数
//大于 greater
class MyCompare
{
public:
    bool operator()(int v1, int v2)
    {
        return v1 > v2;
    }
};

void test01()
{
    vector<int>v;
    v.push_back(10);
    v.push_back(40);
    v.push_back(30);
    v.push_back(50);
    v.push_back(20);
    v.push_back(60);

    sort(v.begin(), v.end());
    for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;

    //使用函数对象 改变算法策略，变为排序规则从大到小
    cout << "降序排列为：" << endl;
    //sort(v.begin(), v.end(), MyCompare());
    //使用内建函数
```

```
    sort(v.begin(), v.end(), greater<int>());

    for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;
}

int main()
{
    test01();
    return 0;
}
```

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`

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

//内建函数对象 逻辑仿函数 逻辑非

void test01()
{
    vector<bool>v;
    v.push_back(true);
    v.push_back(false);
    v.push_back(true);
    v.push_back(false);

    for (vector<bool>::iterator it = v.begin(); it != v.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;

    //利用逻辑非，将容器v搬运到v2容器，同时取反
    vector<bool>v2;
    v2.resize(v.size());
    transform(v.begin(), v.end(), v2.begin(),logical_not<bool>());

    for (vector<bool>::iterator it = v2.begin(); it != v2.end(); it++)
    {
```

```
        cout << *it << " ";
    }
    cout << endl;

}

int main()
{
    test01();
    return 0;
}
```

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、赋值、修改等
- `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中`

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);`
- `//beg开始迭代器`
- `//end结束迭代器`
- `//_func 函数或者函数对象`

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

// for_each 遍历算法
```

```
//普通函数
void print01(int val)
{
    cout << val << " ";
}
//仿函数
class print02
{
public:
    void operator()(int val)
    {
        cout << val << " ";
    }
};

void test01()
{
    vector<int> v;
    for (int i = 0; i < 10; i++)
    {
        v.push_back(i);
    }
    for_each(v.begin(), v.end(), print01); //普通函数
    cout << endl;

    for_each(v.begin(), v.end(), print02()); //仿函数
    cout << endl;
}

int main()
{
    test01();
    return 0;
}
```

5.1.2 transform

功能描述：

- 搬运容器到另一个容器

函数原型：

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

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

//常用容器 transform
```

```
void print01(int val)
{
    cout << val << " ";
}
class TS
{
public:
    int operator()(int val)
    {
        return val;
    }
};

void test01()
{
    vector<int>v;
    for (int i = 0; i < 10; i++)
    {
        v.push_back(i);
    }

    vector<int>vTarget; //目标容器
    vTarget.resize(v.size());  //搬运前需要提前开辟空间
    transform(v.begin(), v.end(), vTarget.begin(),TS());
    for_each(vTarget.begin(), vTarget.end(), print01);
    cout << endl;
}

int main()
{
    test01();
    return 0;
}
```

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：

```
#include <algorithm>
#include <vector>
#include <string>
void test01() {

    vector<int> v;
    for (int i = 0; i < 10; i++) {
        v.push_back(i + 1);
    }
    //查找容器中是否有 5 这个元素
    vector<int>::iterator it = find(v.begin(), v.end(), 5);
    if (it == v.end())
    {
        cout << "没有找到!" << endl;
    }
    else
    {
        cout << "找到:" << *it << endl;
    }
}

class Person {
public:
    Person(string name, int age)
    {
        this->m_Name = name;
        this->m_Age = age;
    }
    //重载==
    bool operator==(const Person& p)
    {
        if (this->m_Name == p.m_Name && this->m_Age == p.m_Age)
        {
            return true;
        }
```

```
        return false;
    }

public:
    string m_Name;
    int m_Age;
};

void test02() {

    vector<Person> v;

    //创建数据
    Person p1("aaa", 10);
    Person p2("bbb", 20);
    Person p3("ccc", 30);
    Person p4("ddd", 40);

    v.push_back(p1);
    v.push_back(p2);
    v.push_back(p3);
    v.push_back(p4);

    vector<Person>::iterator it = find(v.begin(), v.end(), p2);
    if (it == v.end())
    {
        cout << "没有找到!" << endl;
    }
    else
    {
        cout << "找到姓名:" << it->m_Name << " 年龄: " << it->m_Age << endl;
    }
}
```

总结： 利用find可以在容器中找指定的元素，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）

示例：

```
#include <algorithm>
#include <vector>
#include <string>

//内置数据类型
class GreaterFive
{
public:
    bool operator()(int val)
    {
        return val > 5;
    }
};

void test01() {

    vector<int> v;
    for (int i = 0; i < 10; i++) {
        v.push_back(i + 1);
    }

    vector<int>::iterator it = find_if(v.begin(), v.end(), GreaterFive());
    if (it == v.end()) {
        cout << "没有找到!" << endl;
    }
    else {
        cout << "找到大于5的数字:" << *it << endl;
    }
}

//自定义数据类型
class Person {
public:
    Person(string name, int age)
    {
        this->m_Name = name;
        this->m_Age = age;
    }
public:
    string m_Name;
    int m_Age;
};

class Greater20
{
public:
    bool operator()(Person &p)
    {
        return p.m_Age > 20;
    }
```

```
};

void test02() {

    vector<Person> v;

    //创建数据
    Person p1("aaa", 10);
    Person p2("bbb", 20);
    Person p3("ccc", 30);
    Person p4("ddd", 40);

    v.push_back(p1);
    v.push_back(p2);
    v.push_back(p3);
    v.push_back(p4);

    vector<Person>::iterator it = find_if(v.begin(), v.end(), Greater20());
    if (it == v.end())
    {
        cout << "没有找到!" << endl;
    }
    else
    {
        cout << "找到姓名:" << it->m_Name << " 年龄: " << it->m_Age << endl;
    }
}

int main() {

    //test01();

    test02();

    system("pause");

    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

示例：

```
#include <algorithm>
#include <vector>

void test01()
{
    vector<int> v;
    v.push_back(1);
    v.push_back(2);
    v.push_back(5);
    v.push_back(2);
    v.push_back(4);
    v.push_back(4);
    v.push_back(3);

    //查找相邻重复元素
    vector<int>::iterator it = adjacent_find(v.begin(), v.end());
    if (it == v.end()) {
        cout << "找不到!" << endl;
    }
    else {
        cout << "找到相邻重复元素为:" << *it << endl;
    }
}
```

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：

```
#include <algorithm>
#include <vector>

void test01()
{
    vector<int>v;

    for (int i = 0; i < 10; i++)
    {
        v.push_back(i);
    }
    //二分查找
    bool ret = binary_search(v.begin(), v.end(),2);
    if (ret)
    {
        cout << "找到了" << endl;
    }
    else
    {
        cout << "未找到" << endl;
    }
}

int main() {

    test01();

    system("pause");

    return 0;
}
```

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：

```
#include <algorithm>
#include <vector>

//内置数据类型
void test01()
{
    vector<int> v;
    v.push_back(1);
    v.push_back(2);
    v.push_back(4);
    v.push_back(5);
    v.push_back(3);
    v.push_back(4);
    v.push_back(4);

    int num = count(v.begin(), v.end(), 4);

    cout << "4的个数为：  " << num << endl;
}

//自定义数据类型
class Person
{
public:
    Person(string name, int age)
    {
        this->m_Name = name;
        this->m_Age = age;
    }
```

```
    bool operator==(const Person & p)
    {
        if (this->m_Age == p.m_Age)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
    string m_Name;
    int m_Age;
};

void test02()
{
    vector<Person> v;

    Person p1("刘备", 35);
    Person p2("关羽", 35);
    Person p3("张飞", 35);
    Person p4("赵云", 30);
    Person p5("曹操", 25);

    v.push_back(p1);
    v.push_back(p2);
    v.push_back(p3);
    v.push_back(p4);
    v.push_back(p5);

    Person p("诸葛亮",35);

    int num = count(v.begin(), v.end(), p);
    cout << "num = " << num << endl;
}
int main() {

    //test01();

    test02();

    system("pause");

    return 0;
}
```

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

示例：

```
#include <algorithm>
#include <vector>

class Greater4
{
public:
    bool operator()(int val)
    {
        return val >= 4;
    }
};

//内置数据类型
void test01()
{
    vector<int> v;
    v.push_back(1);
    v.push_back(2);
    v.push_back(4);
    v.push_back(5);
    v.push_back(3);
    v.push_back(4);
    v.push_back(4);

    int num = count_if(v.begin(), v.end(), Greater4());

    cout << "大于4的个数为：  " << num << endl;
}

//自定义数据类型
class Person
{
public:
    Person(string name, int age)
```

```
    {
        this->m_Name = name;
        this->m_Age = age;
    }

    string m_Name;
    int m_Age;
};

class AgeLess35
{
public:
    bool operator()(const Person &p)
    {
        return p.m_Age < 35;
    }
};
void test02()
{
    vector<Person> v;

    Person p1("刘备", 35);
    Person p2("关羽", 35);
    Person p3("张飞", 35);
    Person p4("赵云", 30);
    Person p5("曹操", 25);

    v.push_back(p1);
    v.push_back(p2);
    v.push_back(p3);
    v.push_back(p4);
    v.push_back(p5);

    int num = count_if(v.begin(), v.end(), AgeLess35());
    cout << "小于35岁的个数：" << num << endl;
}


int main() {

    //test01();

    test02();

    system("pause");

    return 0;
}
```

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：

```
#include <algorithm>
#include <vector>

void myPrint(int val)
{
    cout << val << " ";
}

void test01() {
    vector<int> v;
    v.push_back(10);
    v.push_back(30);
    v.push_back(50);
    v.push_back(20);
```

```
    v.push_back(40);

    //sort默认从小到大排序
    sort(v.begin(), v.end());
    for_each(v.begin(), v.end(), myPrint);
    cout << endl;

    //从大到小排序
    sort(v.begin(), v.end(), greater<int>());
    for_each(v.begin(), v.end(), myPrint);
    cout << endl;
}

int main() {

    test01();

    system("pause");

    return 0;
}
```

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

示例：

```
#include <algorithm>
#include <vector>
#include <ctime>

class myPrint
```

```
{
public:
    void operator()(int val)
    {
        cout << val << " ";
    }
};

void test01()
{
    srand((unsigned int)time(NULL));
    vector<int> v;
    for(int i = 0 ; i < 10;i++)
    {
        v.push_back(i);
    }
    for_each(v.begin(), v.end(), myPrint());
    cout << endl;

    //打乱顺序
    random_shuffle(v.begin(), v.end());
    for_each(v.begin(), v.end(), myPrint());
    cout << endl;
}

int main() {

    test01();

    system("pause");

    return 0;
}
```

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 目标容器开始迭代器

示例:

```
#include <algorithm>
#include <vector>

class myPrint
{
public:
    void operator()(int val)
    {
        cout << val << " ";
    }
};

void test01()
{
    vector<int> v1;
    vector<int> v2;
    for (int i = 0; i < 10 ; i++)
    {
        v1.push_back(i);
        v2.push_back(i + 1);
    }

    vector<int> vtarget;
    //目标容器需要提前开辟空间
    vtarget.resize(v1.size() + v2.size());
    //合并  需要两个有序序列
    merge(v1.begin(), v1.end(), v2.begin(), v2.end(), vtarget.begin());
    for_each(vtarget.begin(), vtarget.end(), myPrint());
    cout << endl;
}

int main() {

    test01();

    system("pause");

    return 0;
}
```

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：

```
#include <algorithm>
#include <vector>

class myPrint
{
public:
    void operator()(int val)
    {
        cout << val << " ";
    }
};

void test01()
{
    vector<int> v;
    v.push_back(10);
    v.push_back(30);
    v.push_back(50);
    v.push_back(20);
    v.push_back(40);

    cout << "反转前：  " << endl;
    for_each(v.begin(), v.end(), myPrint());
    cout << endl;

    cout << "反转后：  " << endl;

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

```
}

int main() {

    test01();

    system("pause");

    return 0;
}
```

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：

```
#include <algorithm>
#include <vector>
```

```
class myPrint
{
public:
    void operator()(int val)
    {
        cout << val << " ";
    }
};

void test01()
{
    vector<int> v1;
    for (int i = 0; i < 10; i++) {
        v1.push_back(i + 1);
    }
    vector<int> v2;
    v2.resize(v1.size());
    copy(v1.begin(), v1.end(), v2.begin());

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

int main() {

    test01();

    system("pause");

    return 0;
}
```

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：

```
#include <algorithm>
#include <vector>

class myPrint
{
public:
    void operator()(int val)
    {
        cout << val << " ";
    }
};

void test01()
{
    vector<int> v;
    v.push_back(20);
    v.push_back(30);
    v.push_back(20);
    v.push_back(40);
    v.push_back(50);
    v.push_back(10);
    v.push_back(20);

    cout << "替换前： " << endl;
    for_each(v.begin(), v.end(), myPrint());
    cout << endl;

    //将容器中的20 替换成 2000
    cout << "替换后： " << endl;
    replace(v.begin(), v.end(), 20,2000);
    for_each(v.begin(), v.end(), myPrint());
    cout << endl;
}

int main() {

    test01();

    system("pause");

    return 0;
}
```

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:

```
#include <algorithm>
#include <vector>

class myPrint
{
public:
    void operator()(int val)
    {
        cout << val << " ";
    }
};

class ReplaceGreater30
{
public:
    bool operator()(int val)
    {
        return val >= 30;
    }

};

void test01()
{
    vector<int> v;
    v.push_back(20);
    v.push_back(30);
    v.push_back(20);
    v.push_back(40);
```

```
    v.push_back(50);
    v.push_back(10);
    v.push_back(20);

    cout << "替换前：" << endl;
    for_each(v.begin(), v.end(), myPrint());
    cout << endl;

    //将容器中大于等于的30 替换成 3000
    cout << "替换后：" << endl;
    replace_if(v.begin(), v.end(), ReplaceGreater30(), 3000);
    for_each(v.begin(), v.end(), myPrint());
    cout << endl;
}

int main() {

    test01();

    system("pause");

    return 0;
}
```

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：

```
#include <algorithm>
#include <vector>

class myPrint
{
public:
    void operator()(int val)
    {
```

```
        cout << val << " ";
    }
};

void test01()
{
    vector<int> v1;
    vector<int> v2;
    for (int i = 0; i < 10; i++) {
        v1.push_back(i);
        v2.push_back(i+100);
    }

    cout << "交换前： " << endl;
    for_each(v1.begin(), v1.end(), myPrint());
    cout << endl;
    for_each(v2.begin(), v2.end(), myPrint());
    cout << endl;

    cout << "交换后： " << endl;
    swap(v1, v2);
    for_each(v1.begin(), v1.end(), myPrint());
    cout << endl;
    for_each(v2.begin(), v2.end(), myPrint());
    cout << endl;
}

int main() {

    test01();

    system("pause");

    return 0;
}
```

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：

```
#include <numeric>
#include <vector>
void test01()
{
    vector<int> v;
    for (int i = 0; i <= 100; i++) {
        v.push_back(i);
    }

    int total = accumulate(v.begin(), v.end(), 0);

    cout << "total = " << total << endl;
}

int main() {

    test01();

    system("pause");

    return 0;
}
```

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：

```
#include <numeric>
#include <vector>
#include <algorithm>

class myPrint
{
public:
    void operator()(int val)
    {
        cout << val << " ";
    }
};

void test01()
{

    vector<int> v;
    v.resize(10);
    //填充
    fill(v.begin(), v.end(), 100);

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

int main() {

    test01();

    system("pause");

    return 0;
}
```

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：

```
#include <vector>
#include <algorithm>

class myPrint
{
public:
    void operator()(int val)
    {
        cout << val << " ";
    }
};

void test01()
{
```

```
    vector<int> v1;
    vector<int> v2;
    for (int i = 0; i < 10; i++)
    {
        v1.push_back(i);
        v2.push_back(i+5);
    }

    vector<int> vTarget;
    //取两个里面较小的值给目标容器开辟空间
    vTarget.resize(min(v1.size(), v2.size()));

    //返回目标容器的最后一个元素的迭代器地址
    vector<int>::iterator itEnd =
        set_intersection(v1.begin(), v1.end(), v2.begin(), v2.end(),
vTarget.begin());

    for_each(vTarget.begin(), itEnd, myPrint());
    cout << endl;
}

int main() {

    test01();

    system("pause");

    return 0;
}
```

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:

```
#include <vector>
#include <algorithm>

class myPrint
{
public:
    void operator()(int val)
    {
        cout << val << " ";
    }
};

void test01()
{
    vector<int> v1;
    vector<int> v2;
    for (int i = 0; i < 10; i++) {
        v1.push_back(i);
        v2.push_back(i+5);
    }

    vector<int> vTarget;
    //取两个容器的和给目标容器开辟空间
    vTarget.resize(v1.size() + v2.size());

    //返回目标容器的最后一个元素的迭代器地址
    vector<int>::iterator itEnd =
        set_union(v1.begin(), v1.end(), v2.begin(), v2.end(), vTarget.begin());

    for_each(vTarget.begin(), itEnd, myPrint());
    cout << endl;
}

int main() {

    test01();

    system("pause");

    return 0;
}
```

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：

```
#include <vector>
#include <algorithm>

class myPrint
{
public:
    void operator()(int val)
    {
        cout << val << " ";
    }
};

void test01()
{
    vector<int> v1;
    vector<int> v2;
    for (int i = 0; i < 10; i++) {
        v1.push_back(i);
        v2.push_back(i+5);
    }

    vector<int> vTarget;
    //取两个里面较大的值给目标容器开辟空间
    vTarget.resize( max(v1.size() , v2.size()));
```

```
    //返回目标容器的最后一个元素的迭代器地址
    cout << "v1与v2的差集为：  " << endl;
    vector<int>::iterator itEnd =
        set_difference(v1.begin(), v1.end(), v2.begin(), v2.end(),
vTarget.begin());
    for_each(vTarget.begin(), itEnd, myPrint());
    cout << endl;


    cout << "v2与v1的差集为：  " << endl;
    itEnd = set_difference(v2.begin(), v2.end(), v1.begin(), v1.end(),
vTarget.begin());
    for_each(vTarget.begin(), itEnd, myPrint());
    cout << endl;
}

int main() {

    test01();

    system("pause");

    return 0;
}
```

总结：

- 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