```
NameType m_Name;
AgeType m_Age;
```

};函数高级

默认函数参数注意事项

1. 没有默认值的参数必须在有默认参数的左边
2. 函数的声明和实现只能有一个有默认参数

函数占位参数

语法: `返回值类型 函数名（数据类型）{}`

```
void function(int a, int = 10){
    return;
}
```

1. 占位参数还可以用默认参数

函数重载

作用: 函数名相同提高复用性

满足条件:

- 同一个作用域下，如：全局作用域
- 函数名称相同
- 函数参数**类型不同**或者**个数不同**或者**顺序不同**

注意:

1. 函数的返回值不可以作为重载的条件，因为仍然会出现歧义
2. 引用作为重载条件:

```
void func(int &a)
func(a);
void func(const int &a)
func(10);
```

3. 函数重载碰到默认参数：所以尽量不要在重载时候使用默认参数

```
void func(int a, int b = 10)

void func(int a )
{}
func(10);//出现错误，因为出现歧义了
```

类和对象

C++面向对象三大特性：封装、继承、多态

万事万物都是对象

封装

意义：

- 将属性和行为作为一个整体，表现生活中的事物
- 将属性和行为加以权限控制

语法：`class 类名{ 访问权限： 属性/行为};`

类中的属性和行为统一称为成员：

- 属性： 又称 成员属性 成员变量
- 行为： 又称 成员函数 成员方法

访问权限有三个

- public：公共权限　　成员：类内外都可以访问
- protected: 保护权限　　成员：类内可以访问，类外不行；继承的类内可以
- private : 私有权限　　成员：类内可以访问；继承的类内也不行

struct 和 class的区别

唯一区别就是默认的访问权限不一样：

- struct 默认公共权限
- class 默认私有权限

成员属性设置为私有

好处：

- 可以自己控制读写权限
- 可以检测数据的有效性，可以自己写代码检测输入的数值是否合理

对象的初始化和清理

——构造函数和析构函数

编译器提供的构造函数和析构函数是空实现

- 构造函数：赋初值等初始化

  语法：`类名（）{}`

  1. 构造函数没有返回值也不写void
  2. 函数名与类名相同
  3. 可以有参数，可以发生重载
  4. 程序自动调用，只会调用一次
- 析构函数：对象销毁前的清理工作

  语法：`~类名（）{}`

  1. 没有返回值也不写void
  2. 函数名与类名相同
  3. 不可以有参数，不可以重载
  4. 程序自动调用，只调用一次
- 构造函数和析构函数都是public权限

构造函数的分类

- 有参无参

  就是有参数和无参数的区别

- 普通构造和拷贝构造

```
class Person{

};
Person( const Person &p){
    //将传入的p的属性都拷贝到我身上
    cout<< "这是一个拷贝构造函数" << endl;
}
```

构造函数的调用

1. 括号法

```
class Person{};
Person p1;  //默认构造函数
Person p2(10); //有参构造函数
Person p3(p2); //拷贝构造函数
```

   - 注意：调用默认构造函数时候不要加（）
   - `Person p1();`编译器会认为是一个函数声明；

2. 显示法

```
Person p1;
Person p2 = Person(10); //有参构造
Person p3 = Person(p2); //拷贝构造
Person(10);    //匿名对象，执行结束后立即回收
```

   - 注意：不要用拷贝构造函数初始化匿名对象；

     编译器会认为`Person (p3) === Person p3`两个p3,重定义

3. 隐式转换法

`Person p4 = 10`

`Person p4 = p3`

拷贝构造函数的调用时机

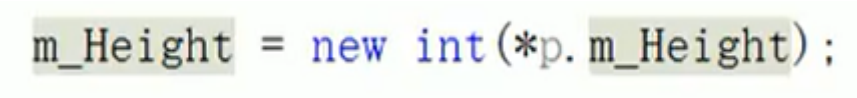
1. 用原有的对象初始化新对象
2. 值传递方式给函数传值
3. 用值的方式返回局部对象

构造函数的调用规则

1. c++至少给一个类添加3个函数：构造函数、析构函数、拷贝构造函数
2. 如果我们写了有参构造，就不默认添加默认构造函数，但是还有拷贝构造函数
3. 如果我们写了拷贝构造函数，编译器不再提供其他普通的构造函数了

深拷贝和浅拷贝

- 浅拷贝：简单的赋值拷贝操作
- 深拷贝：在堆区重新申请空间，进行拷贝操作
- 编译器提供的拷贝构造函数是浅拷贝，浅拷贝带来的问题就是堆区的内存重复释放（析构函数可以用于释放在堆区开辟的数据）



- 上图就是深拷贝，浅拷贝只是简单的赋值操作，面对指针问题，只是赋值了指定内存的地址，并没有产生新的内存空间。深拷贝是产生了新的内存空间来存放相同的内容。

初始化列表

语法： `Person(int a, int b):m_a(a), m_b(b){}`

类对象做为类成员（对象成员）

初始化列表也能给对象成员赋初值，相当于是初始化了对象成员

- 在构造过程中，当其他类对象作为本类成员，先构造对象成员，再构造本类。
- 析构顺序和构造顺序相反。

静态成员

1. **静态成员变量**：
    - `static int m_A;`
    - 所有对象共享同一份数据：静态成员**不属于某个对象**（但是**会属于某一个类，注意类和对象的区别**），所有对象共享一份数据，所以静态成员变量有两种访问方式：
        1. 通过对象进行访问；
        2. 通过类名进行访问：`Person::m_A`
    - 在编译阶段分配内存：程序没有运行之前就开始分配内存
    - 类内声明，类外初始化：类外初始化是一个必须的过程

      `Person::m_a = 100`
    - 静态成员变量也是有访问权限。
2. **静态成员函数**
    - 所有对象共享一个函数，静态成员函数也是有访问权限的。
    - 访问方式：
        1. 通过对象
        2. 通过类名`Person::func();`
    - 静态成员函数只能访问静态成员变量,不能访问非静态成员变量，原因非静态成员变量必须属于某一个对象，静态成员函数无法区分在哪个对象里面。

成员变量和成员函数分开存储

- 成员变量和成员函数是分开存储的，只有**非静态成员变量**才属于类的对象上（静态成员变量和函数都不属于类的对象）
- C++编译器会给每个空对象也分配一个字节的空间，是为了区分空对象占内存的位置，每一个空对象都有独一无二的内存地址

this 指针

this 指针指向被调用的成员函数所属的对象，谁调用，就指向谁

1. 解决名称冲突

2. 返回对象本身用*this，（链式编程思想）

```
Person& PersonAddAage(Person &p){
    this->age += p.age;

    return *this;
}  //返回引用是能够返回对象本身，如果返回值，就会复制一个Person对象出来
```

空指针访问成员函数

报错原因：传入的指针为空，可以访问不加this的成员，但是无法访问加this的属性。

```
if (this == NULL){
    return;
}  //防止传入空指针导致出错
```

const修饰成员函数

常函数：限定一个只读状态

this指针是指针常量，不可以修改指针的指向

```
class Person
{
    //this指针本来是一个指针常量，指针的指向是不能修改的
    //const Person * const this;
   void showPerson() const {//常函数
    //成员函数后面加const 本质上修饰的是this 指针，让指针的指向的值也不可以修改
      //this->m_A = 100;    //不可以修改成员属性
   }
}
```

- 成员函数后加`const`后，称为常函数；
- 常函数内不可以修改成员属性
- 成员属性声明时候加关键字`mutable`后，在常函数中依旧可以修改

常对象：不允许调用普通函数，因为普通函数可能会修改属性值

- 常对象的值也是不能修改的，除非是`mutable int m_B`
- 常对象只能调用常函数

4.4 友元

目的就是让一个函数或者类 访问另一个类中的私有成员

友元的关键字：friend

友元的实现：

- 全局函数做友元

```
class Person{
   //友元 爱人全局函数可以访问private 权限下的属性和行为
    friend void lover(Person *person);

};
```

- 类做友元

```
class Person{
    //让lover类可以访问private权限
    friend class lover;
}
```

- 成员函数做友元

```
class Person{
    //让lover类中的visit函数作为友元
    friend void lover::visit();
}
```

4.5 运算符重载

对已有的运算符重新进行定义，赋予另一种功能，以适用于不同的数据类型

4.5.1 加法重载

```
//成员函数形式
Person operator+(Person &p);
Person p3 = p1.operator+(p2);
Person p3 = p1 + p2;
//全局函数形式
Person operator+(Person &p1, Person &p2);
Person p3 = operator+(p1, p2);
Person p3 = p1 + p2;
//函数重载
Person operator+(Person &p1, int a);
```

运算符重载也可以实现函数重载

4.5.2 左移运算符重载

（`<<`的重载）

```
//不会用成员函数实现重载左移运算符

//利用
ostream & operator<<(ostream &cout, Person p){

    return cout;
}
```

总结：**重载左移运算符**配合**友元**可以实现输出自定义数据类型

4.5.3 递增运算符重载

```
class MyInteger{
    //自定义整型
    //重载前++运算符，返回引用是为了一直对一个数据进行递增操作
    MyInteger& operator++(){
        //
        m_Num++;
        return *this;
    }
    // 加入占位参数，编译器视为后置++
    //后置递增返回的是值，因为对应的要重复处理的数据不是temp
    MyInteger operator++(int){
        MyInteger temp = *this;
        m_Num++;
        return temp;
    }
}
```

4.5.4 赋值运算符重载

C++给类提供了4个函数：

1. 默认构造函数
2. 默认析构函数
3. 默认拷贝函数
4. 赋值运算符 `operator=`，对属性进行值拷贝

   存在深浅拷贝的问题。

可以直接写`p1 = p2`，但是这是浅拷贝，会存在堆区问题重复释放的问题，需要利用深拷贝，来解决相关的问题

```
class Person{
   Person& operator=(Person &p) {

      //应该先判断是否有属性在堆区，如果有先释放干净,再进行深拷贝
      if (m_Age!= NULL)
      {
         delete m_Age;
         m_Age = NULL;
      }
      //深拷贝
      m_Age = new int(*p.m_Age);
      //返回对象本身，可以实现链等p3 = p2 = p1;
      return *this;
   }
};
```

4.5.5 关系运算符重载

```
    //重载==
    bool operator==(Person &p) {
        if (this->m_Name == p.m_Name && this->m_Age == p.m_Age)
        {
            return true;
        }
        return false;
    }
    //重载! =
    bool operator!=(Person &p) {
        if (this->m_Name == p.m_Name && this->m_Age == p.m_Age)
        {
            return false;
        }
        return true;
    }

```

4.5.6 函数调用运算符重载

- ()也可以重载
- 重载之后使用方式非常像函数的调用，称为仿函数
- 仿函数没有固定写法，非常灵活

```
class MyPrint {

public:

    //重载函数调用运算符
    void operator()(string text) {
        cout << text << endl;
    }
};
    MyPrint myPrint;
    myPrint("HelloWOtdd");
```

匿名函数对象

```
//MyPrint()这是一个匿名的对象，有括号
cout<<MyPrint()("HelloWorld");
```

4.6 继承

下级的成员除了拥有上级的共性，还有自己的特点

4.6.1 继承的语法

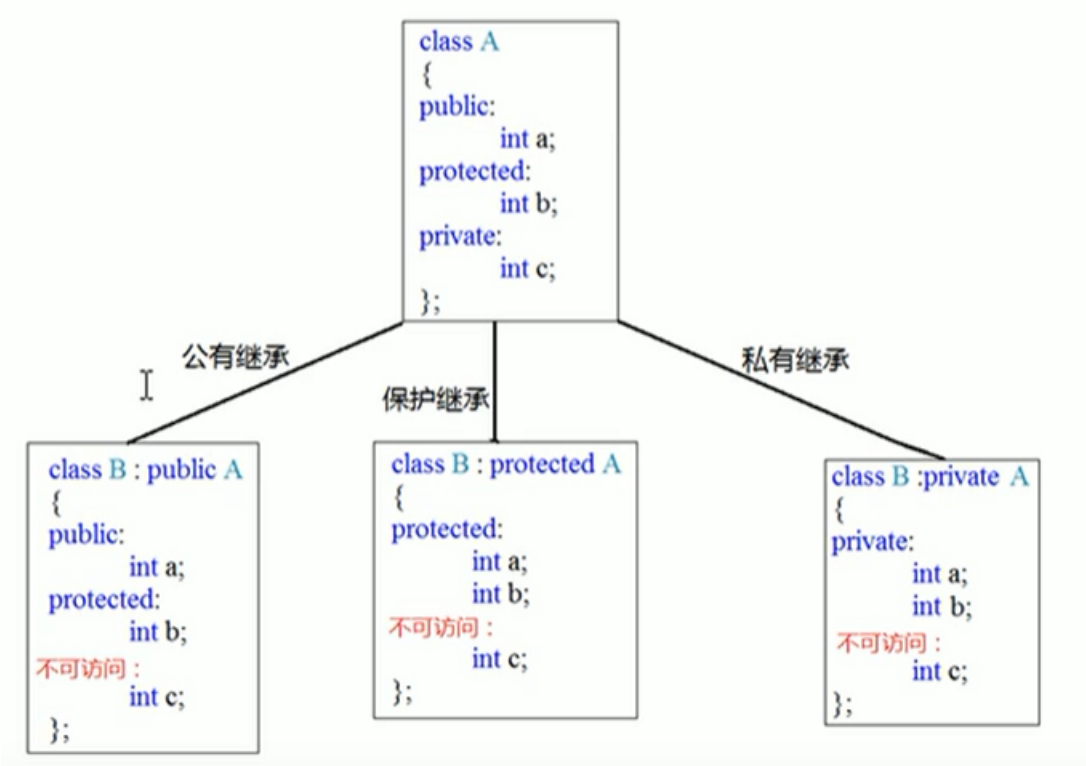
```cpp
//class 子类 : public 父类;
//class 派生类 : public 基类;
class Cpp : public BasePage {
public:
   void content() {

      cout << "Cpp内容" << endl;
   }

};
```

4.6.2 继承方式

继承方式有三种：

- 公共继承
- 保护继承
- 私有继承

三种继承方式的关系：

- 

继承中的对象模型

在父类中所有的非静态成员属性，都会被子类继承下去。

可以利用vs开发人员命令提示符工具：

1. 跳转盘符
2. cd
3. `c1 /d1 reportSingleClassLayout`类名 文件名

继承中的构造和析构顺序

子类继承父类之后，当创建子类对象，也会调用父类对象

继承中的构造时先构造父类，再构造子类，析构与构造相反

4.6.5 继承同名成员处理方式

要是想在子类中访问父类继承下来的同名成员，要加上一个作用域

```
Son.m_A; //直接调用时调用的子类的成员
Son.Father::m_A;  //访问父类继承的同名成员
```

- 如果子类中出现了和父类同名的函数，子类所有同名函数会隐藏掉父类的所有同名函数
- 如果想访问父类中的同名函数，必须要加上作用域。

4.6.6 继承同名静态成员处理方式

静态成员和非静态成员出现同名，处理方式一致。

- 访问子类同名成员 直接访问
- 访问父类同名成员 加作用域

通过类名来访问：

```
Son::m_A;
Son::Base::m_A;
```

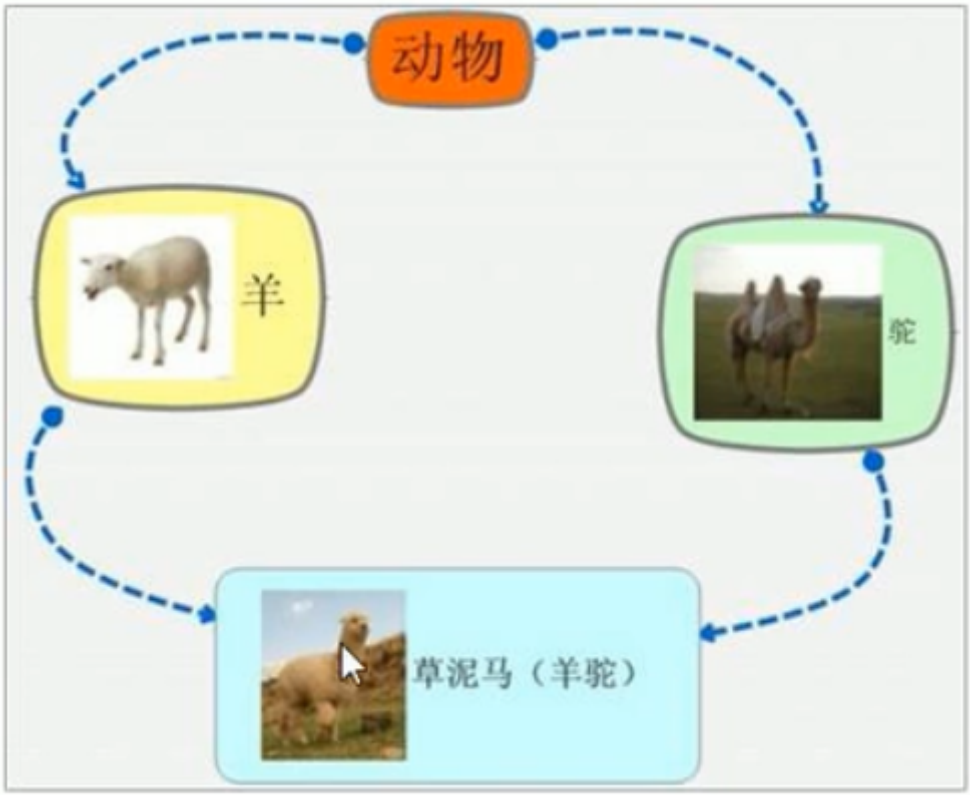
4.6.7 多继承语法

C++允许**一个类继承多个类**

语法：`class 子类：继承方式 父类1， 继承方式 父类2...`

实际开发不建议多继承:可能会引发父类中的同名成员的问题，访问时候要加上作用域

4.6.8 菱形继承（钻石继承）



利用虚继承，可以解决菱形继承问题

- 菱形继承主要问题，子类继承了两份数据，导致资源的浪费以及毫无意义
- 利用虚继承可以解决菱形继承问题

```
class Animal {};
class Sheep : virtual public Animal {};   //Animal 虚基类
class Tuo : virtual public Animal {};
class SheepTuo : public Sheep, public Tuo {};
```

虚继承的底层：

- `vbptr`: virtual base pointer, 指向`vbtable`虚基列表

4.7 多态

4.7.1 多态的基本概念

多态分为两类：

- 静态多态：函数重载 和 运算符重载属于静态多态 复用函数名
- 动态多态：派生类、虚函数

静态多态和动态多态区别：

- 静态多态的函数地址早绑定：编译时候
- 动态多态的函数地址晚绑定：运行的时候

C++中允许在调用父类时候使用子类

父类的指针或引用指向子类的对象实现多态：

```
Base base = new Son;
base->func();
```

动态多态满足的条件：

1. 有继承关系
2. 子类**重写（函数返回值类型 函数名称 参数列表完全相同）**父类中的虚函数
3. 子类重写时候的`virtual`可写可不写

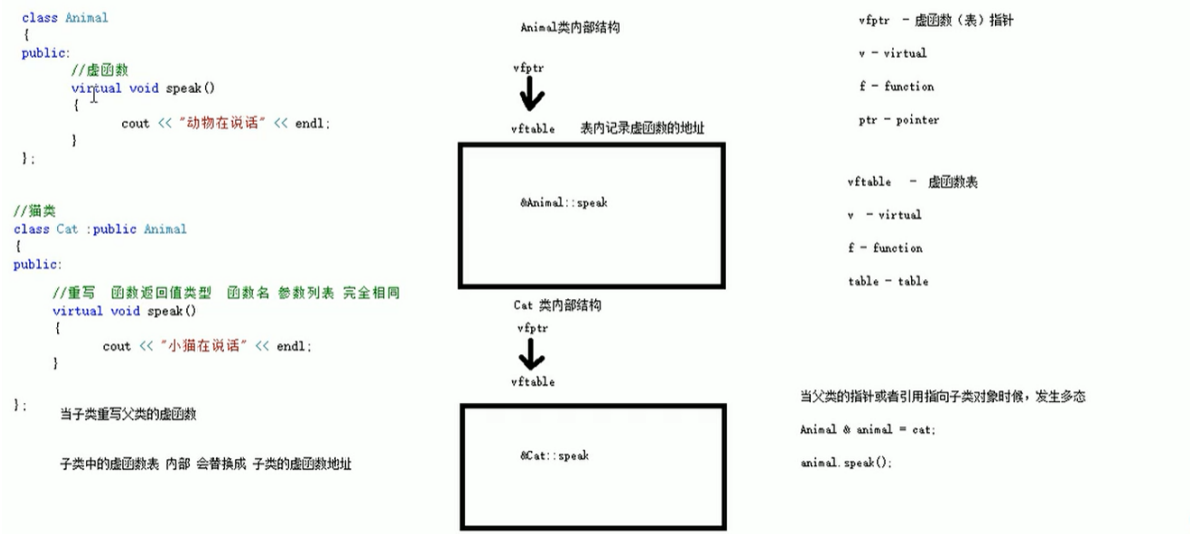
动态多态的使用：

- 父类的指针或者引用 指向 子类对象

```
class Animal {
public:
   //虚函数
   virtual void speak() {

      cout << "动物在说话" << endl;
   }
};

class Dog :public Animal {
public:
    //此处的重写virtual 可写可不写
   void speak() {

      cout << "Dog在说话" << endl;
   }
};

//父类的指针或者引用指向子类的对象
void doSpeak(Animal &animal) {
   animal.speak();
}
void test01() {

   Dog dog;
   doSpeak(dog);
}
```

4.7.2 多态的原理

加上`virtual`之后会有一个虚函数指针`virtual function pointer(vfptr)`虚函数指针指向虚函数表，里面记录的是虚函数的地址



多态的优点：

- 代码组织结构清晰
- 可读性强
- 利于前期和后期的扩展和实现

在开发中，提倡开闭原则：对扩展进行开放，对修改进行关闭

4.7.3 纯虚函数和抽象类

在多态中，通常父类中虚函数的实现是毫无意义的，主要是调用子类重写的内容

因此可以将虚函数改为纯虚函数

语法：`virtual 返回值类型 函数名 （参数列表） = 0;`

只要有一个纯虚函数，就称这个类为抽象类：

- 抽象类无法实例化对象
- 子类必须要重写纯虚函数，否则同样是抽象类

```
Base base = new Son;
base->func();
```

4.7.4 虚析构和纯虚析构

多态使用时候，如果子类有属性开辟在堆区，父类指针无法调用子类的析构函数，无法释放子类对象

父类指针在析构的时候 不会调用子类的析构函数，解决办法，虚析构,纯虚析构

虚析构：使用后如果没有虚函数，还可以实例化对象。

纯虚析构：使用后该类就成为抽象类，无法实例化对象

都要有声明还需要有实现。

```
class Animal
{
    virtual ~Animal()
    {
    //这是一个虚析构函数
   }
   //纯虚析构的声明
   virtual ~Animal() = 0;
```

```
}

//纯虚析构函数的实现
Animal::~Animal(){

}
```

总结：

1. 虚析构和纯虚析构就是用来解决通过父类指针释放子类对象
2. 如果子类中没有堆区数据，可以不写为虚析构或者纯虚析构
3. 拥有纯虚析构函数的类也属于抽象类

5. 文件操作

程序运行的文件属于临时文件，文件可以将数据持久化

用的`<fstream>`文件流

文件类型：

1. 文本文件 -文本AsCII码形式存储
2. 二进制文件 -文件以文本的二进制形式存储在计算机中

操作文件三大类：

1. `ofstream`： 写操作
2. `ifstream`：读操作
3. `fstream`：读写操作

5.1 文本文件

5.1.1 写文件

写文件的步骤：

1. 包含头文件

   `#include <fstream>`

2. 创建流对象

   `ofstream ofs`

3. 打开文件

   `ofs.open("文件路径", 打开方式)`

   - 不存在文件就创建文件

   文件打开方式：

打开方式	解释
`ios::in`	为读文件而打开文件
`ios::out`	为写文件而打开文件
`ios::ate`	初始位置：文件尾
`ios::app`	追加方式写文件
`ios::trunc`	如果文件存在，则先删除后创建
`ios::binary`	二进制方式

注意：文件打开方式可以配合使用，利用|操作符

`ios::binary|ios::in`

4. 写数据

`ofs << “写入的数据”`

5. 关闭文件

`ofs.close()`

5.1.2 读文件

1. 包含头文件

`#include <fstream>`

2. 创建流对象

`ifstream ifs;`

3. 打开文件并判断文件是否打开成功

`ifs.open("文件路径",打开方式)`

4. 读数据

四种方式读取：

```
    //读数据

    //第一种方式
    char buf[1024] = { 0 };
    while (ifs>>buf)
    {
    //ifs>>buf 会将文件中数据一行行读，读到最后(EOF = -1)
     //会返回 EOF, 否则会继续读下一行
        cout << buf << endl;
    }

    //第二种方式
    char buf[1024] = { 0 };
    while (ifs.getline(buf, sizeof(buf)))
    {
     //getline 获取一行数据，
     //getline(要存的数组首地址，数组长度)
        cout << buf << endl;
    }
```

```
20
21      //第三种方式
22      string buf;
23
24      while ( getline(ifs, buf))
25      {
26          cout << buf << endl;
27      }
28
29      ////第四种方式
30      //文件中的数据一个
31      char c;
32      while ((c = ifs.get()) != EOF )//EOF : 文件尾
33      {
34          cout << c;
35      }
36
```

5. 关闭文件

   `ifs.close()`

5.2 二进制文件

以二进制的方式对文件进行读写操作

打开方式指定为 `ios::binary`

5.2.1 写文件

利用流对象调用成员函数write

函数原型：`ostream& write(const char buffer, int len);`

1. 包含头文件
2. 创造输出流对象
3. 打开文件
4. 写文件

```
1  void test01()
2  {
3      //两种办法都可以实现文件的打开
4      //1.
5      //ofstream ofs("Person.txt", ios::out | ios::binary);
6      //2.
7      ofstream ofs;
8      ofs.open("Person.txt", ios::out | ios::binary);
9
10     //这是什么初始化方式?
11     //沿用 struct的初始化方式
12     Person p = { "zhangsan", 18 };
13      //写文件的地址：强制转换为一个常量指针
14     ofs.write((const char *) &p, sizeof(Person));
15     ofs.close();
16 }
```

5.2.2 读文件

函数原型

`istream & read(char * buffer, int len)`

字符指针buffer指向内存中一段存储空间，`len`是读写文件的字节数

```
void test01() {
   ifstream ifs;

   ifs.open("Person.txt", ios::in | ios::binary);

   if (!ifs.is_open())
   {
      cout << "wenjian dakai shibai " << endl;
      return;
   }

   Person p;
   //*************
   ifs.read((char *)&p, sizeof(Person));
   //**************
   cout << p.m_Name << "------" << p.m_Age << endl;

   ifs.close();
}
```

C++提高编程

- C++泛型编程 STL

泛型编程：基于模板技术的编程

- “>>、<<”除了 cin>> 输入流；cout<<输出流 的意思外还有另一个是：>>向右位移、<<向左位移；就是一个整数，如10，二进制为1010,向右位移一位就是0101,既是10>>1=5

1. 模板

1.1 模板的概念

模板的特点：

- 模板不可以直接使用，它只是一个框架
- 模板通用性但不是万能的

1.2 函数模板

- C++中有两种模板：函数模板和类模板

1.2.1 函数模板语法

作用：建立一个通用函数，其返回值类型和形参类型可以不具体制定，用一个虚拟类型来代表。

目的：将类型参数化，提高复用性

语法：

```
template <typename T>
函数声明或者定义
```

```
template <typename T>   //声明模板，T表示一个类型名
//typename 可以用 class代替
void mySwap(T &a, T &b)
{
   T temp = a;
   a = b;
   b = temp;
}

   int a = 1;
   int b = 2;
   //两种方式使用模板
   //1. 自动类型推导
   mySwap(a, b);

   //2. 显示指定类型
   mySwap<int>(a, b);


```

1.2.2 函数模板注意事项

- 自动类型推导要推导出一致的数据类型
- 模板必须要确定出T的数据类型才可以使用（即使给出的函数没有用到模板）

1.2.4 普通函数和函数模板的区别

1. 普通模板调用可以发生隐式类型转换
2. 函数模板 用自动类型推导 不可以发生隐式类型转换
3. 函数模板 用显示制定类型 可以发生隐式类型转换

总结：建议使用显示指定类型的方式使用模板

1.2.5 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的方式来强制调用模板函数

```
//强制调用模板
myPrint<>(a,b);
```

3. 函数模板可以发生函数重载
4. 如果函数模板可以更好的匹配，优先调用函数模板

总结：为了防止二义性，不要同时写相同的普通函数和函数模板

1.2.6 模板的局限性

- 模板的通用性是有限的；
- 利用具体化的模板可以解决自定义类型的通用化

```
template<class T>
bool myCompare(T &a, T &b)
{
}
//如果是两个自定义的类型，就会走下面的代码
template<>bool myCompare(Person &p1, Person &p2);
```

- 学习模板不是为了写模板，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

```
template <class NameType, class AgeType>
class Person
{
public:
   NameType m_Name;
   AgeType m_Age;
};

//调用
Person<string, int> p1("houzi", 99);

```

1.3.2 类模板和函数模板的区别

1. 类模板没有自动类型推导使用方式

   只能用显示指定类型方式

2. 类模板在模板参数列表中可以有默认参数

```
template <class NameType, class AgeType= int>
class Person
{
public:
   NameType m_Name;
   AgeType m_Age;
};
Person<string> p1("houzi", 99);
```

1.3.3 类模板中的成员函数创建时机

1. 类模板中成员函数调用时才去创建；
2. 普通类成员函数在编译时就创建了；

1.3.4 类模板对象做函数参数

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. 将参数模板化：

3. 整个类都模板化

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

//2、参数模板化
template<class T1, class T2>
void printPerson2(Person<T1, T2> &p)
{
   p.showPerson();
    //看数据的类型名字
   cout << "T1 的类型为 " << typeid(T1).name() << endl;
   cout << "T2 的类型为 " << typeid(T2).name() << endl;
}

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


void test01()
{
   Person<string, int> p1("猴子", 99);
   printPerson1(p1);
   Person<string, int> p2("猪猪", 34);
   printPerson2(p2);
   Person<string, int> p3("唐僧", 23);
   printPerson3(p3);
}
```

1.3.5 类模板与继承

如果父类是类模板，子类需要指定出父亲中T的数据类型

```
template <class T>
class Base
{
public:
   T m;
};


```

```cpp
//class Son :public Base   //错误，必须知道父类中的数据类型
class Son :public Base<int>
{

};

//如果想灵活指定父类中的T类型，子类也需要变成模板
template<class T1, class T2>
class Son2 :public Base<T1>
{
   T2 obj;
};

```

1.3.6 类模板成员函数的类外实现

```cpp
template <class NameType, class AgeType>
class Person
{
public:
   Person(NameType name, AgeType age);
   void showPerson();

   NameType m_Name;
   AgeType m_Age;
};

//构造函数的类外实现
template<class T1, class T2>
Person<T1, T2>::Person(T1 name, T2 age)
{
   this->m_Name = name;
   this->m_Age = age;
}

//成员函数的类外实现
template<class T1, class T2>
void Person<T1,T2>::showPerson()
{
   cout << "name: " << this->m_Name << " age: " << this->m_Age << endl;
}

```

1.3.7 类模板分文件编写

- 类模板中成员函数创建时机是在调用阶段，导致分文件编写时候的链接不到

解决方法：

- 直接包含Cpp文件
- 将函数的声明和定义写在同一个文件中，使用hpp后缀。

```cpp
//直接包含cpp，用以解决类模板的份文件编写
#include "Person.cpp"

//第二种解决方式：将.h和.cpp写到一起，后缀名hpp
#include "Person.hpp"

```

1.3.8 类模板与友元

1. 全局函数类内实现
2. 全局函数类外实现：需要提前让编译器知道有一个全局函数存在

```cpp
//为了让类外实现的全局函数知道有这么一个Person类，需要先有一个类模板的声明
template<class T1, class T2>
class Person;

//全局函数类外实现，需要让编译器先知道有这么一个函数模板
template<class T1, class T2>
void printPerson2(Person<T1,T2> p)
{
     cout<<endl;
}

template<class T1, class T2>
class Person
{
    //全局函数类内实现
  friend void printPerson1(Person<T1,T2> p)
  {
      cout<<endl;
  }
  //全局函数类外实现，==需要让编译器提前知道这个函数的存在==
  //变成了一个函数模板
    friend void printPerson2<>(Person<T1,T2> p)
};
```

总结：类外比较麻烦-----

2. STL 初始

2.1 STL的诞生

- 可重复利用的追求：面向对象、泛型编程
- **面向对象**的三大特性：封装、继承、多态
- 大多数情况下，数据结构和算法没有一套标准，导致被迫运行大量重复工作

  如：`myAdd myPlus相同的功能`

2.2 STL基本概念

- Standard Template Library
- 分为：容器、算法、迭代器
- 容器和算法通过迭代器进行无缝连接
- STL几乎所有的代码都采用模板类或者模板函数

2.3 六大组件

容器、算法、迭代器、仿函数、适配器（配接器）、空间配置器

- 容器：各种数据结构，存放数据用的
- 算法：各种sort等
- 迭代器：扮演容器和算法之间的胶合剂
- 仿函数：行为类似函数，可作为算法的某种策略
- 适配器：一种用来修饰容器或者仿函数或者迭代器接口的某种东西
- 空间配置器：负责空间的配置与管理

2.4 容器算法和迭代器

容器：用来放东西的

```
1．序列式容器：强调值得排序，序列式容器每个元素有固定的位置

2．关联式容器：二叉树结构，各个元素之间没有严格的物理上的顺序关系
```

算法（Algorithms）：解决问题的，有限的步骤解决逻辑或者数学问题

```
1．质变算法：运算会更改区间内的元素的内容

2．非质变算法：运算过程中不会更改区间内的元素内容
```

迭代器：算法要通过迭代器才能访问到容器

- 每个容器都有自己专属的迭代器
- 迭代器类似于指针

迭代器种类：

种类	功能	支持运算
输入迭代器	对数据的只读访问	只读，支持++ == ！=
输出迭代器	对数据的只写访问	只写，支持++
前向迭代器	读写，并且能向前推进迭代器	读写
双向迭代器	读写，并且能向前向后推进迭代器	读写
随机访问迭代器	读写，可以以跳跃的方式访问任意数据，最强的迭代器	读写

2.5 初识

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`

`vector<int>::reverse_iterator`\`v1.rbegin(),v1.rend()`

回调函数：回调函数就是一个通过函数指针调用的函数。如果你把函数的指针（地址）作为参数传递给另一个函数，当这个指针被用来调用其所指向的函数时，我们就说这是回调函数。回调函数不是由该函数的实现方直接调用，而是在特定的事件或条件发生时由另外的一方调用的，用于对该事件或条件进行响应。

```
void printFunc(int val)
{
   cout << val << endl;
}

void test01()
{
   //创建vector容器，数组
   vector<int> v;

   //向容器中插入数据
   v.push_back(10);
   v.push_back(20);
   v.push_back(30);
   v.push_back(40);
   v.push_back(50);

   //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数据的后一个位置
   //
   ////第一种遍历方式
```

```
    //while (itBegin != itEnd)
    //{
    // cout << *itBegin << endl;
    // itBegin++;
    //}

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

    //第三种遍历方式 利用遍历算法for_each
    for_each(v.begin(), v.end(), printFunc);

}
```

2.5.2 存放自定义数据类型

```
void test01()
{
    //创建vector容器，数组
    vector<Person> v;

    Person p1("a", 10);
    Person p2("b", 20);
    Person p3("c", 30);
    Person p4("d", 40);
    Person p5("e", 50);
    //向容器中插入数据

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

    //第二种遍历方式
    for (vector<Person>::iterator it = v.begin(); it != v.end(); it++)
    {
        cout << "姓名：  " << (*it).m_Name << "  ";
        cout << "年龄：  " << it->m_Age << endl;
    }
}

void test02()
{
    vector<Person*> v;

    Person p1("a", 10);
    Person p2("b", 20);
    Person p3("c", 30);
    Person p4("d", 40);
    Person p5("e", 50);
    //向容器中插入数据

```

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

    for (vector<Person*>::iterator it = v.begin(); it != v.end(); it++)
    {
        cout << "姓名:  " << (*it)->m_Name << "  ";
        cout << "年龄:  " << (**it).m_Age << endl;
    }
}
```

2.5.3 Vector容器嵌套容器

```
void test01()
{
    //创建vector容器，数组
    vector<vector<int>> v;

    vector<int> v1;
    vector<int> v2;
    vector<int> v3;
    vector<int> v4;

    for (int i = 0; i < 3; i++)
    {
        v1.push_back(i);
        v2.push_back(i + 3);
        v3.push_back(i + 6);
    }

    v.push_back(v1);
    v.push_back(v2);
    v.push_back(v3);
    v.push_back(v4);

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

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

}

```

3. STL 常用容器

3.1 string容器

3.1.1 string基本概念

string 本质：

- C++风格字符串，是个类

string 和 char *的区别：

- char * 是一个指针
- string 是一个类，里面封装了char *

3.1.2 string构造函数

```
string s1;  //默认构造

//string(const char* s);
//用char *来初始化string
char * str = "nigeshazi"
string s2(str);

//string s2(const string &str);
//拷贝构造
string s3(s2);

//string(int num, char c);
string (10, "a"); //"aaaaaaaaaa"
```

3.1.3 string赋值操作

```
//1.
string str1;
str1 = "hello world";
//2.
string str2;
str2 = str1;
//3.
string str3;
str3 = 'a';
"=========================="
//4.
string str4;
str4.assign("Hello");
//5.string &
string str5;
str5.assign("Hello", 4);//用前4个赋值给str5
//6.
str6.assign(str5);
//7.
str7.assign(10, 'w');
```

3.1.4 string 字符串拼接

```
str1 = "hello ";
str2 = "world";
str1 += str2;//str1 = "hello world"
str1 += '!';//str1 = "hello world!"
str1 += "DaShaZi";//str1 = "hello world!DaShaZi"

string str3 = "I";
str3.append(" love ");
str3.append("ni eewew",2);//str3 = I love ni
str3.append(str2);////str3 = I love ni world
//从0号位置截取3位
str3.append(str2, 0, 3);////str3 = I love ni worldwor
```

3.1.5 string字符串查找和替换

```
string str1 = "abcdefg";
//rfind 和 find 的区别：
//rfind 从右往左查找，find从左往右查找
st1.find("de");   //返回第一个"de"的下标，没有返回-1
str1.rfind("de");//返回倒数第一个"de"的下标，没有返回-1

//替换
str1.replace(1,3,"1111");//1号位置起3个字符，替换成1111
```

3.1.6 string字符串比较

比较方式：

- 比较ASCII码
- = 0 两个相等
- > 0 string1比string2大
- < 0

```
str1.compare(str2);

```

3.1.7 string 字符存取

```
string str = "hello";
str.size();//返回长度5
//读
str[2];// =l;
str.at(1);//e
//写
str[0]= 'x';
str.at(1) = 'x';
```

3.1.8 string 插入和删除

```
string str = "hello";
str.insert(1, "***");//"h***ello"
str.erase(1, 3);//从第一个位置删除3个字符
```

3.1.9 string 子串

```
string str = "abcdef";
//从1位置截取3个字符
string suBstr = str.substr(1,3);//"bcd"

//应用
string email = "zhangsan@sina.com";
int pos = email.find("@");
string userName = email.substr(0,pos);//zhangsan
```

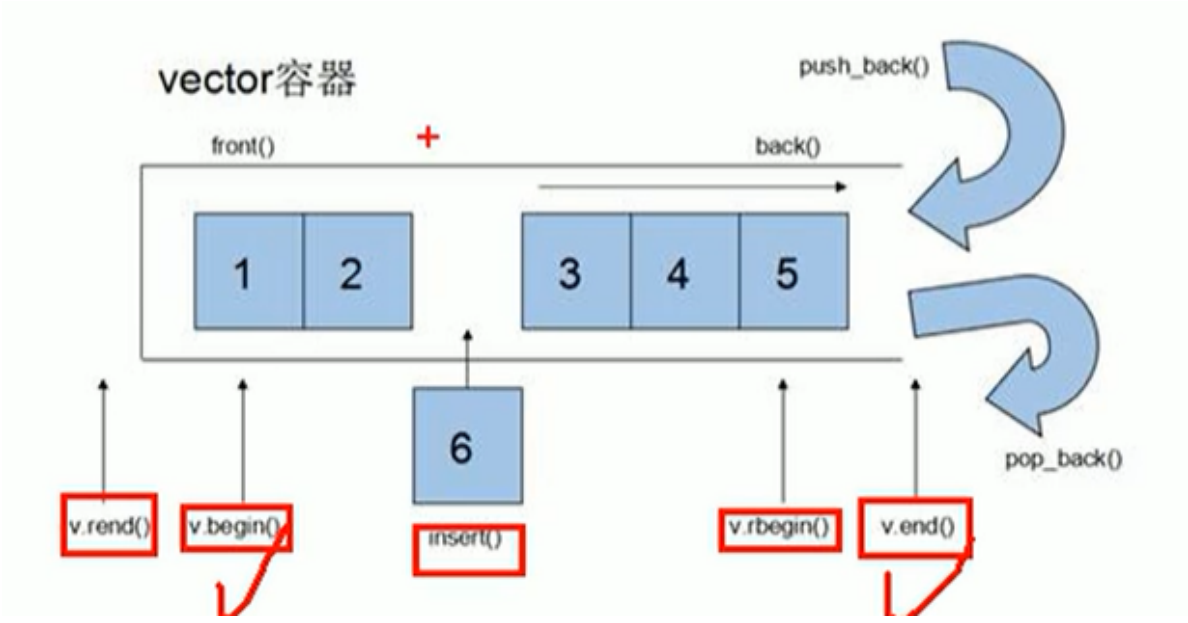
3.2 vector 容器

功能：

- vector数据结构和数组非常相似，也称单端数组

动态扩展：

- 不是续接空间，而是找到更大的空间然后拷贝到新空间，释放原空间



3.2.2 vector 的构造函数

```cpp
vector<int> v1;//无参构造
vector<int>::iterator it;  //迭代器

//通过区间构造
vector<int>v2(v1.begin(),v1.end());

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

//拷贝构造
vector<int>v4(v3);
```

3.2.3 vector 赋值操作

```cpp
vector<int> v1;
for(int i = 0; i< 10, i++)
{
    v1.push_back(i);
}
//1
vector<int> v2 = v1;
//2
vector<int> v3;
v3.assign(v1.begin(),v1.end());
//3
vector<int> v4;
v4.assign(10, 100); //10个100
```

3.2.4 vector 容量和大小

```cpp
vector<int> v1;

//判断容器是否为空
v1.empty();//true:为空；false:不为空

//容量
v1.capacity();

//v1的大小个数
v1.size();
//容量>=大小
//重新制定大小
v1.resize(15); //多的位置默认用0填充
v1.resize(15, 100);//指定默认填充的值为100
v1.resize(5);//超出的部分被删除


```

3.2.5 vector 插入和删除

```cpp
vector<int>v1;

v1.push_back(10);//尾插

```

```
v1.pop_back();//尾删法

//插入，第一个参数是迭代器
v1.insert(v1.begin(),100);//在第一个位置插100
v1.insert(v1.begin(),2, 1000);//在第一个位置插入两个1000

//删除,参数也是迭代器
v1.erase(v1.begin());
//删除某个区间
v1.erase(v1.begin(),v1.end());
//清空
v1.clear();
```

3.2.6 vector数据存取

```
vector<int> v1;

v1[10];  //利用中括号访问第11个元素
v1.at(10); //利用at方式访问11元素

//获取第一个元素
v1.front();
//获取最后一个元素
v1.back();
```

3.2.7 vector互换容器

- 实现两个容器的元素进行互换

```
vector<int> v1;
vector<int> v2;
/*
赋值v1 0、1、2、3、4、5、6、7、8、9
v2:9、8、7、6、5、4、3、2、1、0
*/
v1.swap(v2);//将v1 和 v2

//巧用swap()可以收缩空间
vector<int> v;
/*

*/

vector<int>(v).swap(v);
//vector<int>(v)是匿名对象，利用拷贝构造出一个v,和v交换之后匿名对象就被回收掉了
```

3.2.8 vector预留空间

功能描述：

- 减少vector在**动态扩展容量**时候的扩展次数

```
vector<int> v;
v.reserve(10000);//预留10万位置
```

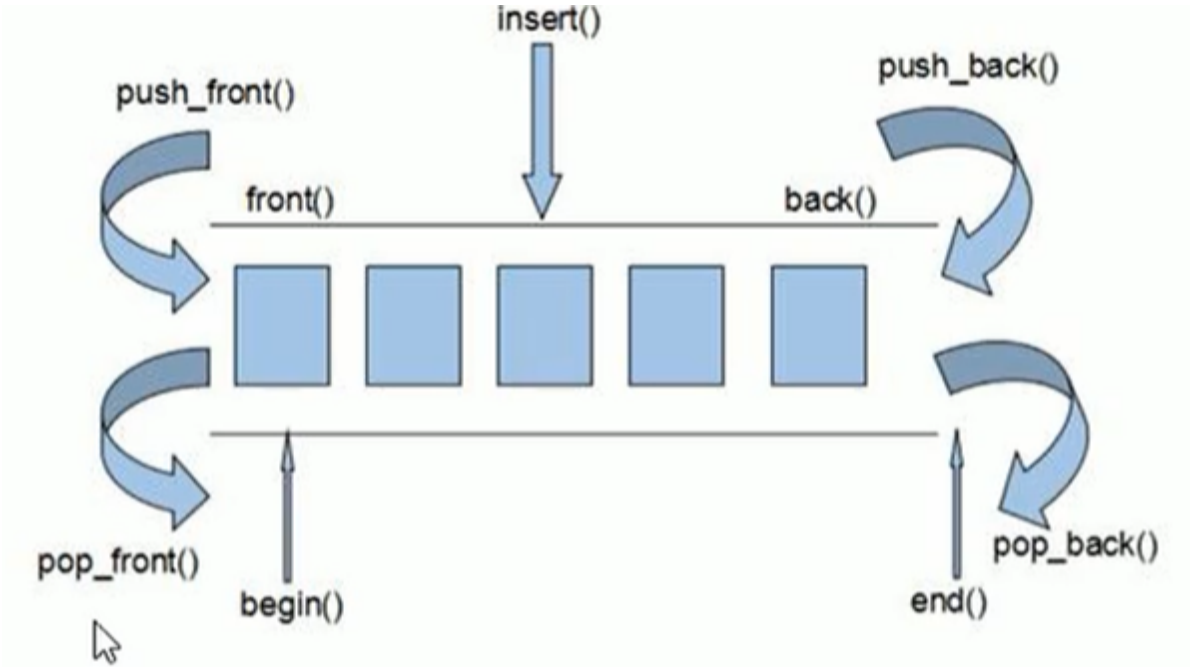
3.3 deque容器

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和删除操作

deque和vector的区别：

- vector 对于头部的插入效率很低
- deque相对而言，头部插入速度更快
- vector访问元素速度更快一些



deque内部工作原理：

中控器：维护每段缓冲区的内容，缓冲区中存放真实数据

deque使用时候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

```
#include <deque>
//普通无参构造
deque<int> d1;

//区间构造
deque<int>d2(d1.begin(),d1.end());

//10个100构造
deque<int>d3(10,100);

//拷贝构造
deque<int>d4(d3);

d1.push_back(i);  //尾插

//deque迭代器和vector一样，是iterator
```

```
17
18  //打印deque中的元素
19  for(deque<int>::iterator it = d1.begin(); it!=d1.end();it++)
20  {
21      cout<< *it<<" ";
22  }
23  cout << endl;
24
25  //只读打印元素，只读迭代器要使用const_iterator
26  void printDeque(const deque<int>&d)
27  {
28      for(deque<int>::const_iterator it= d.begin(); it!= d.end(); it++)
29      {
30          cout << *it << " ";
31      }
32      cout << endl;
33  }
```

3.3.3 deque赋值操作

```
1   deque<int> d1;
2   for(int i = 0; i<10; i++)
3   {
4       d1.push_back(i);
5   }
6   //operator= 赋值
7   d2 = d1;
8
9   //assign 区间赋值
10  d3.assign(d1.begin(),d1.end());
11  //assign 10个100赋值操作
12  d4.assign(10, 100);
```

3.3.4 deque容器大小操作

```
1   deque<int> d;
2
3   //判断是否为空，空返回true,非空返回false
4   d.empty();
5
6   //判断大小
7   d1.size();
8
9   //重新制定大小
10  d1.resize(15);//用0 来填充
11  d1.resize(15, 1);//用1填充
12
```

deque没有容量的概念，因为容量是无限大

3.3.5 deque的插入和删除

```
deque<int> d;

//尾插
d.push_back(i);
//头插
d.push_front(i);

//尾删
d.pop_back();

//头删
d.pop_front();

//插入
//insert插入一个100
d1.insert(d1.begin(), 100);
//插入2个1000
d1.insert(d1.end(),2,1000);
//按照区间进行插入
d1.insert(d1.begin(), d2.begin(), d2.end());

//删除
d1.erase(d1.begin());
//按照区间删除
d1.erase(d1.begin(),d1.end());

//清空
d1.clear();

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

- 插入和删除位置指定要用迭代器

3.3.6 deque 数据存取

```
//通过中括号访问每一个元素
for(int i=0;i<d.size();i++)
{
    cout<<d[i]<< " ";
    cout<<d.at(i)<<" ";
}
cout<<endl;

//通过at访问
```

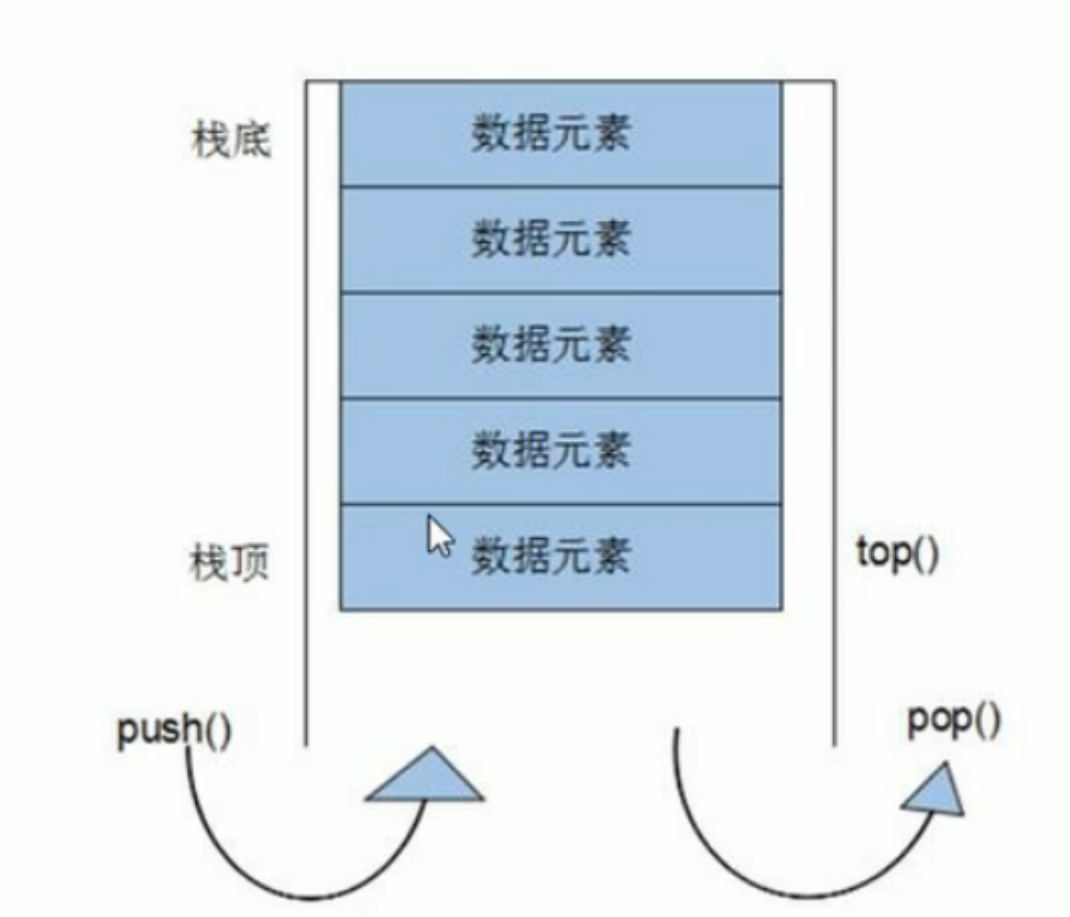
```
d.at(i);
//首尾元素
//得到第一个元素
d.front();
//得到最后一个元素
d.back();

```

3.3.7 deque容器的排序

```
#inclde <deque>
//使用标准算法sort要包含algorithm
#include <algorithm>
deque<int> d;
//默认从小到大排序
sort(d.begin(),d.end());
```

- 对于支持随机访问的迭代器的容器，都可以用sort进行排序

3.5 stack容器



First in Last out

栈不允许有遍历行为，只有栈顶元素可以被外界访问到

3.5.2 stack 常用接口

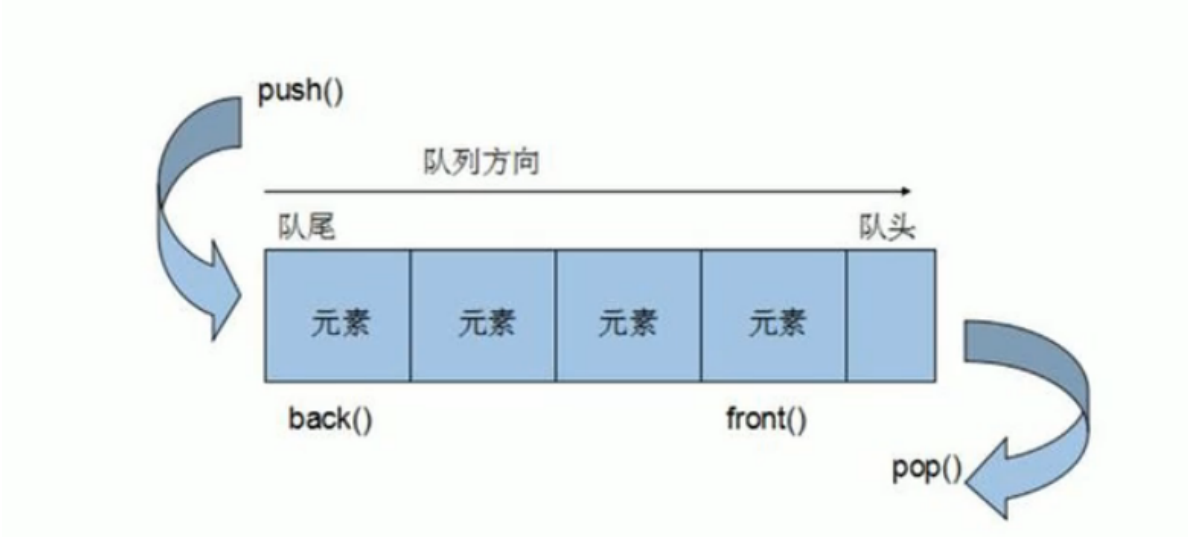
```
//构造函数:默认构造+拷贝构造
```

```
stack<T> stk;
stack(const stack &stk);

//入栈
stk.push(100);
//出栈
stk.pop();

//判断栈是否为空,空返回true,非空返回false
stk.empty();

//栈的大小
stk.size();

//查看栈顶元素
stk.top();
```

3.6 queue 容器

3.6.1 基本概念

FIFO: first in first out




```
#include <queue>
queue<T> que;
queue(const queue &que);

//入队
q.push();

//出队
q.pop();

//是否为空
q.empty();

//队头元素访问
q.front();

```

```cpp
//队尾元素访问
q.back();

//队大小
q.size();
```

3.7 list容器

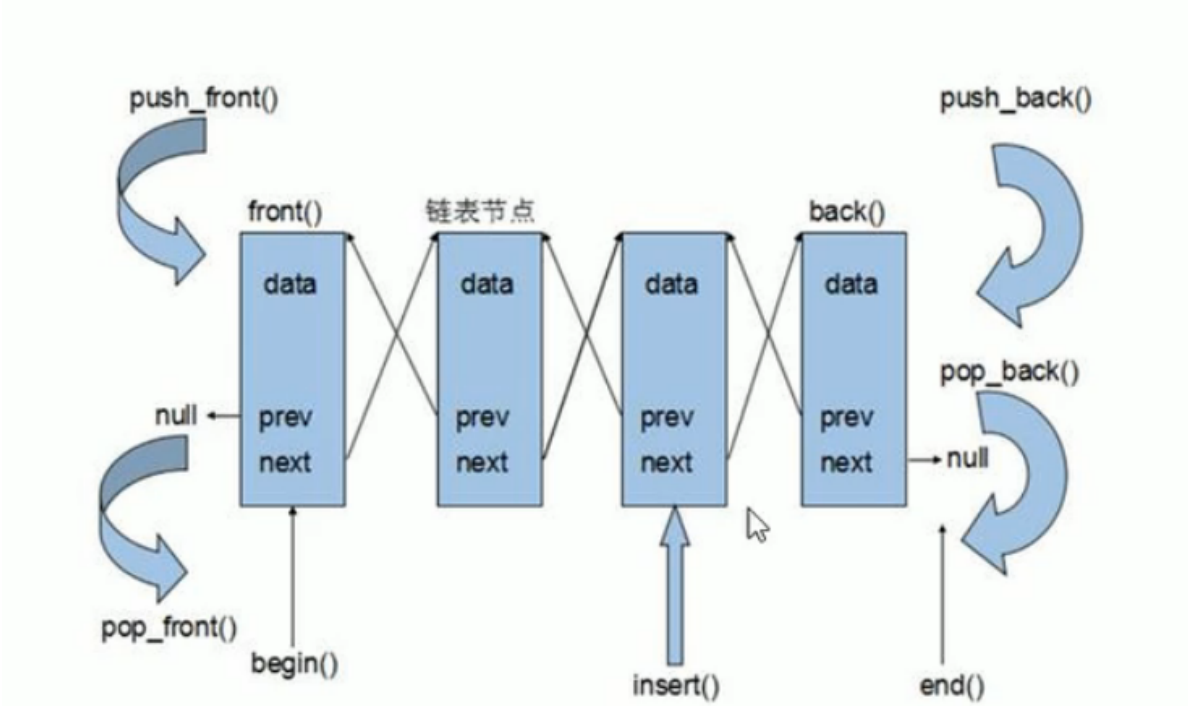
功能：将数据进行链式存储

链表是一种物理存储单位上非连续的方式

结点 = 数据 域+ 指针域；

优点：快速进行插入删除的操作

缺点：遍历速度慢，占用空间比数组大



STL链表是一个双向链表

链表的迭代器**双向迭代器**，而vector等是**随机迭代器**

list和vector是最常用的两种容器

3.7.2 list构造函数

```cpp
#include <list>
//默认构造
list<int>L1;
L1.push_back(1);
//区间方式构造
list<int>L2(L1.begin(),L1.end());
//拷贝构造
list<int>L3(L2);
//n个几的构造
```

```
list<int>L4(10,1000);
//遍历list
void printList(const list<int> &L)
{
    for(list<int>::const_iterator it = L.begin();it!=L.end();it++)
    {

    }
}
```

3.7.3 list赋值与交换

```
//赋值
list<int>l1;
list<int>l2;
list<int>l3;
list<int>l4;
l2 = l1;
l3.assign(l2.begin(),l2.end());
l4.assign(10,100);

//交换
l1.swap(l4);

```

3.7.4 list大小操作

```
//大小，元素个数
l1.size();

//判断是否为空，空返回1，非空返回0
l1.empty();

//重新指定大小
l1.resize(10);
l1.resize(10,1000);

//
```

3.7.5 list 插入和删除

```
//头部
l1.push_back(1);
l1.pop_back();

//尾部
l1.push_front(1);
l1.pop_front();

//插入
l1.insert(pos, elem);
l1.insert(pos, n, elem);
l1.insert(pos,begin,end);

```

```
//删除
l1.erase(begin, end);
l1.erase(pos);
//删除链表中的所有elem
l1.remove(elem);

//清空
l1.clear();
```

3.7.6 list 数据存取

```
//返回第一个元素
l1.front();

//返回最后一个元素
l1.back();

list<int>::iterator it;
it++;it--;
//支持佳佳减减但是不支持加号
```

3.7.7 list的反转和排序

```
//反转,要与reserve区别
l1.reverse();

//排序，默认排序：从小到大
l1.sort();

//排序从大到小
bool myCompare(int v1, int v2)
{
    //降序 就让第一个数 > 第二个数
    return v1>v2;
}
l1.sort(myCompare);
```

- **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，内部会提供一些对应的算法**

3.7.8 自定义类型排序

```
//list 对自定义排序
class Person
{
public:
   Person(string name, int age, int height)
   {
      this->m_Name = name;
      this->m_Age = age;
      this->m_Height = height;
   }
   string m_Name;
   int m_Age;
   int m_Height;
```

```
};

//指定排序规则
bool myCompare(Person &p1, Person &p2)
{
   if (p1.m_Age == p2.m_Age)
   {
      return p1.m_Height > p2.m_Height;
   }
   return p1.m_Age < p2.m_Age;
}

void test01()
{
   list<Person>L;

   Person p1("aaa", 23, 143);
   Person p2("bbb", 54, 123);
   Person p3("ccc", 23, 145);
   Person p4("ddd", 56, 165);
   Person p5("eee", 65, 189);
   Person p6("fff", 23, 200);

   L.push_back(p1);
   L.push_back(p2);
   L.push_back(p3);
   L.push_back(p4);
   L.push_back(p5);
   L.push_back(p6);

   for (list<Person>::iterator it = L.begin();  it != L.end(); it++)
   {
      cout << "姓名:  " << it->m_Name << " 年龄:  " << it->m_Age << " 身高:  "
<< it->m_Height << endl;
   }
   //排序
    L.sort(myCompare);
   cout << "排序后============================" << endl;
   for (list<Person>::iterator it = L.begin(); it != L.end(); it++)
   {
      cout << "姓名:  " << it->m_Name << " 年龄:  " << it->m_Age << " 身高:  "
<< it->m_Height << endl;
   }

}

```

3.8 set/multiset 容器

3.8.1 set基本概念

- 所有元素插入时候自动排序
- set容器不允许插入重复的值，但是不会报错,也不会有这个值

本质:

- set/multiset属于关联式容器，底层结构是用二叉树实现

set 、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
- set插入数据会返回插入结果，表示插入是否成功

```
//set容器的构造
#include <set>

set<int> s1;

//拷贝构造
set<int>s2(s1);

//赋值
set<int>s3;
s3 = s2;

//插入数据只有insert方式，返回对组数据
pair<set<int>::iterator, bool>ret = set.insert(10);

//遍历容器
void printSet(set<int>&s)
{
    for(set<int>::iterator it=s.begin();it!=s.end();it++)
    {
        cout<<*it<< " ";
    }
    cout<<endl;
}

```

3.8.3 set大小和交换

- 统计set容器大小以及交换set容器

函数：没有重新指定大小

```
set<int>s1;

s1.insert(10);

//判断是否为空,空返回true，非空返回false
s1.empty();

//元素个数
s1.size();

//交换
set<int>s2;
s2.insert(20);
s1.swap(s2);
```

3.8.4 set的插入和删除

```
//插入
s1.insert(elem);

//清空
s1.clear();

//删除,传入的是iterator
s1.erase(pos);
s1.erase(pos1,pos2);
//指定删除某个元素
s1.erase(elem);
```

3.8.5 set的查找和统计

功能:

- 对set容器进行查找数据以及统计数据

函数:

```

set<int> s1;

//查找
set<int>::iterator pos = s1.find(30);
if(pos!=s1.end())
{
    cout<<"找到元素，返回的是迭代器"<<*pos<<endl;
}
else
{
    cout<<"没有查找到元素，返回的是s1.end()"<<endl;
}

//统计
//统计20的个数，返回的是一个int
int num = s1.count(20);
```

3.8.7 pair对组创建

```
//不需要包含头文件

//第一种方式
pair<string,int>p("Tom", 20);
pair<string,int>p2 = make_pair("Jerry",30);
cout<<"姓名： "<<p.first<<" 年龄： "<<p.second<<endl;
```

3.8.8 set容器排序

- set容器默认排序从小到大

技术点：

- 利用仿函数，可以改变排序规则

```
//从小到大
set<int>s1;

//从大到小
class MyCompare
{
public:
    bool operator()(int v1, int v2)
    {
        return v1>v2;
    }
}
set<int, MyCompare>s2;

for(set<int, MyCompare>::iterator it=s2.begin();it!=s2.end();it++)
{

}


//set存放自定义类型的时候指定排序规则
class Person
{
public:
    Person(string name, int age)
    {
        this->m_Name = name;
        this->m_Age = age;
    }
    string m_Name;
    int m_Age;
}
class comparePerson
{
public:
    bool operator()(const Person &p1, const Person &p2)
    {
        //按照年龄降序
        return p1.m_Age>p2.m_Age;
      //按照年龄升序
        //return p1.m_Age<p2.m_Age;
    }
}
//不指定规则，对于自定义数据类型可能无法进行插入操作
set<Person, comparePerson>s1;
s1.insert(p1);
```

3.9 map / multimap容器

（使用率仅次于vector 和list）

3.9.1 map基本概念

- map 中所有的元素都是pair
- key - value (pair中第一个元素为键，第二个元素为值)
- 所有的元素都会根据元素的键进行自动排序

本质：

- map/multimap属于关联式容器，底层结构是二叉树

优点：

- 快速通过key 找到 value

map和multimap的区别：

- map不允许有重复的key值
- multimap可以有重复的key值

3.9.2 构造和赋值

```
#include<map>

map<int,int> m;

//拷贝构造
map<int,int>m2(m);

//插入
m.insert(pair<int,int>(1,10));

//赋值
map<int,int>m3;
ma = m2;


void printMap(const map<int,int> &m)
{
    for(map<int,int>::const_iterator it=m.begin();it!=m.end();it++)
    {
        cout<<it->first<<it->secon;

    }
    cout<<endl;
}
```

3.9.3 map大小和交换

```
map<int,int>m;
map<int,int>m1;
//判断是否为空
m.empty();
//大小
m.size();

m.swap(m1);
```

3.9.4 map插入和删除

```
map<int,int>m1;

//插入
//第一种
m.insert(pair<int,int>(1,10));
//第二种
m.insert(make_pair(2,20));
//第三种
m.insert(map<int,int>::value_type(3,30));
//第四种,如果键不存在，就往里面添加
m[4] = 40;
//[]如果不存在，创建这个键，存在了就访问他



//删除
//删除位置
m.erase(m.begin());
//删除区间
m.erase(m.begin(),m.end());
//删除key值对应的对组
m.erase(3);
//清空
m.clear();
```

3.9.5 map查找和统计

```
map<int,int>m;
m.insert(make_pair(1,10));

//找到元素返回的是键所在的位置，没有找到返回m.end();
map<int,int>::iterator pos = m.find(3);

//统计，统计键为3的个数
int num = m.count(3);
//map的count是0或者1，multimap可能大于1
```

3.9.6 map容器的排序

- 利用仿函数，改变排序规则

```
#include<map>

class myCompare
{
public:
    bool operator()(int v1, int v2)
    {
        return v1>v2;
    }
}

map<int,int,myCompare>m;

m.insert(make_pair(1,20));
m.insert(make_pair(2,10));
m.insert(make_pair(4,59));

for(map<int,int,myCompare>::iterator it=m.begin();it!=m.end();it++)
{

}
```

- 自定义类型必须制定排序规则

4. STL-函数对象

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：

函数对象是一个类，不是函数

4.1.2 函数对象使用

特点：

- 函数对象使用时候可以像普通函数那样调用，可以有参数和返回值
- 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

```
class MyAdd
{

public:
    int operator()(int v1, int v2)
    {
        return v1 +v2;
    }
```

```cpp
};

//函数对象使用
void test01()
{
    myAdd myAdd;
    cout<< myAdd(10,20)<<endl;
}

//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
class MyPrint
{
  public:
    MyPrint()
    {
        this->count = 0;
    }
    void operator()(string test)
    {
        cout<<test <<endl;
        this->count ++;
    }
    int count; //内部自己的状态
};

void test02()
{
    MyPrint myPrint;
    myPrint("HelloWorld");
    cout<<"myPrint的调用次数："<<myPrint.count<<endl;
}

//函数对象可以作为参数传递
void doPrint(MyPrint & mp, string test)
{
    mp(test);
}
void test03()
{
    MyPrint myPrint;
    doPrint(myPrint, "Hello cpp");
}
```

4.2 谓词(pred)

4.2.1 谓词概念

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

4.2.2 一元谓词

```
#include <algorithm>

class GreaterFive
{
  public:
    bool operator()(int val)
    {
        return val>5;
    }
};

void test01()
{
    vector<int>v;
    for(int i; i<10;i++)
    {
        v.push_back(i);
    }
    //GreaterFive()这是匿名函数对象
    vector<int>::ierator it = find_if(v.begin(),v.end(),GreaterFive());

}
```

算法 `find_if`

`Pred`：说明想要的是一个谓词

4.2.3 二元谓词

```
//指定sort的排序规则
class MyCompare
{
public:
   bool operator() (int val1, int val2)
   {
      return val1 > val2;
   }
private:

};

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

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

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

}

```

4.3 内建函数对象

4.3.1 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`include<functional>`

4.3.2 算术仿函数

功能：

- 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>`
- `minus`
- `multiplies`
- `divides`
- `modulus`
- `negate`

4.3.3 关系仿函数

功能：

仿函数原型：

- `template<class T> bool equal_to<T>`
- `not_equal_to`
- `greater`
- `greater_equal`

- `less`
- `less_equal`

4.3.4 逻辑仿函数

- `logical_not`
- `logical_and`
- `logical_or`

5 STL-常用算法

概述:

- 算法主要是由头文件`<algorithm><functional><numeric>`组成

5.1 常用遍历算法

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

5.1.1 `for_each`

```
#pragma region for_each
void print01(int val)
{
   cout << val << " ";
}
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);
   }
   //利用普通的函数实现遍历操作
   for_each(v.begin(), v.end(), print01);
   cout << endl;
   //仿函数实现遍历操作
   for_each(v.begin(), v.end(), print02());
   cout << endl;
}

#pragma endregion
```

5.1.2 transform

- 搬运容器到另一个容器中

```
/*////////////////////////////////
    template < class InputIterator, class OutputIterator, class UnaryOperator >
      OutputIterator transform ( InputIterator first1,  // 源容器的起始地址
                                 InputIterator last1,    // 源容器的终止地址
                                 OutputIterator result,  // 目标容器的起始地址
                                 UnaryOperator op );     // 函数指针
    // typedef 目标容器元素类型 (*UnaryOperator)(源容器元素类型);

    template < class InputIterator1, class InputIterator2,
               class OutputIterator, class BinaryOperator >
      OutputIterator transform ( InputIterator1 first1,     // 源容器1的起始地址
                                 InputIterator1 last1,       // 源容器1的终止地址
                                 InputIterator2 first2,      // 源容器2的起始地址，元素个数与1相同
                                 OutputIterator result,      // 目标容器的起始地址，元素个数与1相同
                                 BinaryOperator binary_op ); // 函数指针
    // typedef 目标容器元素类型 (*BinaryOperator)(源容器1元素类型，源容器2元素类型);
    //*////////////////////////////////

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

    int op_increase (int i)
    {
        return i+1;
    }

    int op_sum (int i, int j)
    {
        return i+j;
    }

    int to_upper(int c)
    {
        if (islower(c))
        {
            return (c-32);
        }

        return c;
    }

```

```
    int to_lower(int c)
    {
        if (isupper(c))
        {
            return c+32;
        }

        return c;
    }

    int main () {
        vector<int> first;
        vector<int> second;
        vector<int>::iterator it;

        // set some values:
        for (int i=1; i<6; i++) first.push_back (i*10); //  first: 10 20 30 40 50

        ///将first容器的元素加1赋值给second容器
        second.resize(first.size());        // allocate space !!!必须预先设置一个大小与first相同
        transform (first.begin(), first.end(), second.begin(), op_increase); // second: 11 21 31 41 51
        cout << "second contains:";
        for (it=second.begin(); it!=second.end(); ++it)
        {
            cout << " " << *it;
        }
        cout << endl;
        //*/////////////////////////////////////////////

        ///将first容器的元素与second容器的元素相加，并将得到的结果重新赋值给first

        transform (first.begin(), first.end(), second.begin(), first.begin(), op_sum); //  first: 21 41 61 81 101
        cout << "first contains:";
        for (it=first.begin(); it!=first.end(); ++it)
            cout << " " << *it;
        cout << endl;

 //*////////////////////////////////////////////////////////////////////////////////

        ///大小写转换////////////////////////////////////
        string strsrc("Hello, World!");
        string strdest;
        strdest.resize(strsrc.size());      // !!!必须预先设置一个大小与strsrc相同
        transform(strsrc.begin(), strsrc.end(), strdest.begin(), to_upper); // 转换为大写
        cout << strdest << endl;

        transform(strsrc.begin(), strsrc.end(), strdest.begin(), to_lower); // 转换为小写
        cout << strdest << endl;
        //*/////////////////////////////////////////
```

```
93          return 0;
94      }
```

5.2 常用查找算法

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end()

函数原型：

- `find(iterator beg, iterator end, value);`

  按值查找，找到返回指定位置迭代器，找不到返回结束迭代器

注意：

- 查找自定义数据，要重载==符号，否则find不知道怎么对比数据

5.2.2 `find_if`

`find_if(iterator beg, iterator end, _Pred);`

5.2.3 `adjacent_find`

- 查找相邻重复元素
- `adjacent_find(iterator beg, iterator end);`

  返回相邻元素的第一个位置的迭代器

5.2.4 binary_search 二分查找法

功能：

- 查找指定元素是否存在

函数：

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

  查找到返回true;

  注意：在无序序列中是不可用的

  查找的速度很快，指数速度

5.2.5 count

功能描述：

- 统计元素个数

函数原型

- `count(iterator beg, iterator end, value)`

- 统计内置数据类型
- 统计自定义数据类型

5.2.6 count_if

功能描述：

- 按条件统计

函数原型

- `count_if(iterator beg, iterator end, _Pread)`
- 自定义数据类型
- 内置数据类型

5.3 常用排序算法

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)`

5.3.2 random_shuffle 洗牌算法

功能描述：

- 对容器内的元素进行打乱

函数原型：

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

  记得加随机数种子，才能做到真正的随机

5.3.3 merge

功能

- 两个容器中元素合并，存储到另一个容器中

函数原型

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

  将两个有序序列输入，得到的也是有序序列

5.3.4 reverse

功能

- 将容器内元素进行反转
- `reverse(iterator beg, iterator end);`

5.4 常用拷贝和替换算法

5.4.1 copy

功能描述

- 拷贝到另一个容器中
- `copy(v1.begin()，v1.end()，v2.begin());`

  v2必须要先resize,提前开辟空间

5.4.2 replace

- 指定范围内的所有旧元素替换为新元素
- `replace(iterator beg, iterator end, value_old,value_new);`
- 满足条件的都要替换

5.4.3 replace_if

- 条件替换
- `replace_if(iterator beg, iteractor end, _Pred, newValue);`
-

5.4.4 swap

功能描述

- 互换两个容器中的元素,同种类型的容器
- `swap(container c1, container c2);`

5.5 常用算数生成算法（在`<numeric>`中)

5.5.1 accumulate

- `accumulate(iterator beg, iterator end, value);`
- 计算区间内容器元素总和
- value初始值

5.5.2 fill

- `fill(iter beg, iter end, value);`

  向容器中填充value

5.6 常用集合算法

5.6.1 set_intersection

- 求交集
- 必须是有序序列
- 目标容器需要提前开辟空间
- min(a, b)在algorithm中
- `set_intersection(v1.begin(),v1.end(),v2.begin(),v2.end(),vTarget.begin());`
- 返回值为itend迭代器位置

5.6.2 set_Union

- 求两个集合的并集
- 必须是有序序列
- vTarget需要提前开辟空间
- `set_union(v1.begin(),v1.end(),v2.begin(),v2.end(),vtarget.begin());`
- 返回迭代器位置

5.6.3 set_difference

- 必须是有序序列
- `set_difference(beg1,end1,beg2,end2,tar);`
- 返回的itend