

## 一、C++核心编程

笔记本： My Notebook

创建时间： 2023/2/24 11:11

更新时间： 2023/3/13 21:38

作者： 

URL： [https://blog.csdn.net/qq\\_51604330/article/details/118607922](https://blog.csdn.net/qq_51604330/article/details/118607922)

---

# 一、C++核心编程

## C++是面向对象的编程

### 1 内存分区模型<sup>+</sup>

C++程序在执行时，将内存大方向划分为4个区域

- 代码区：存放函数体的二进制代码，由操作系统进行管理的
- 全局区：存放全局变量和静态变量以及常量
- 栈区：由编译器自动分配释放，存放函数的参数值，局部变量等
- 堆区：由程序员分配和释放，若程序员不释放，程序结束时由操作系统回收

内存四区意义：

不同区域存放的数据，赋予不同的生命周期，给我们更大的灵活编程

### 1、代码区、全局区

## 1.1 程序运行前

在程序编译后，生成了exe可执行程序，未执行该程序前分为两个区域

代码区：

存放 CPU 执行的机器指令

代码区是共享的，共享的目的是对于频繁被执行的程序，只需要在内存中有一份代码即可

代码区是只读的，使其只读的原因是防止程序意外地修改了它的指令

全局区：

全局变量和静态变量存放在此.

const修饰的变量

全局区还包含了常量区，字符串常量和其他常量也存放在此.

该区域的数据在程序结束后由操作系统释放.

```
局部变量a的地址为: 5635316
局部变量b的地址为: 5635304
全局变量g_a的地址为: 6094904
静态变量s_a的地址为: 6094908
静态变量s_b的地址为: 6094912
字符串常量的地址为: 6085608
const修饰的局部变量x的地址为: 5635292
const修饰的全局变量c_g_x的地址为: 6085436
```

## 2、栈区

## 1.2 程序运行后

栈区：]

由编译器自动分配释放，存放函数的参数值、局部变量等

注意事项：不要返回局部变量的地址，栈区开辟的数据由编译器自动释放

```
int* func()
{
    int a = 10; //局部变量 存放在栈区，栈区的数据在函数执行完后自动释放
    return &a; //返回局部变量的地址
}

int main()
{
    //接受func函数的返回值
    int * p = func();

    cout << *p << endl; //第一次可以打印正确的数字，是因为编译器做了保留
    cout << *p << endl; //第二次这个数据就不再保留了|
}
```

## 3、堆区

堆区：

由程序员分配释放，若程序员不释放，程序结束时由操作系统回收

在C++中主要利用new在堆区开辟内存

在堆区开辟的地址为：10  
在堆区开辟的地址为：10

## 二、C++类和对象

C++面向对象三大特性：封装、继承、多态

(1) 互换两个容器的元素封装\*\*

访问权限：

1. **public** 公共权限：成员类内可以访问，类外可以访问

2. **protected** 公共权限：成员类内可以访问，类外不可以访问，子类可以访问父类的内容

3. **private** 私有权限：成员类内可以访问，类外不可以访问，子类不可以访问父类的内容

### **class和struct的区别：默认访问权限不同**

- **struct** 默认权限为公共
- **class** 默认权限为私有

### **class成员属性设置为私有化的好处：**

1、可以自己控制读写权限

2、对于写可以检测数据的有效性

---

## **三、构造函数和析构函数**

- 构造函数：**主要作用在于创建对象时为对象的成员属性赋值，构造函数由编译器自动调用**
- 析构函数：**主要作用在于对象销毁前系统自动调用，执行一些清理工作**

### **(1) 构造函数语法：类名(){}**

1. 构造函数，没有返回值也不写void

2. 函数名称与类名相同

3.构造函数可以有参数，因此可以重载

4.程序在调用对象时候会自动调用构造，无须手动调用，而且只会调用一次

## (2) 析构函数语法：~类名(){}

1.析构函数，没有返回值也不写void

2.函数名称与类名相同，在名称前面加上符号~

3.析构函数不可以有参数，因此不可以发生重载

4.程序在对象销毁前会自动调用析构，无须手动调用，而且只会调用一次

注：默认构造函数、默认析构函数、默认拷贝构造函数都是必须有的，如果自己不构造，编译器会提供一个空实现的构造和析构

## 调用构造函数注意事项：

```
//注意事项  
//调用默认构造函数时，不要加()  
//因为下面这行代码，编译器会认为是一个函数声明  
Person p1();
```

## (3) 拷贝构造函数调用时机

- 使用一个已经创建完毕的对象来初始化一个新对象
- 值传递方式给函数参数传值
- 以值方式返回局部对象

## (4) 拷贝构造函数调用时机

- 如果用户定义有参构造函数，c++不再提供默认无参构造，但是会提供默认拷贝构造

- 如果用户定义拷贝构造函数，c++不会再提供其他构造函数

## (5) 深拷贝与浅拷贝

**深拷贝：简单的赋值拷贝操作**

**浅拷贝：在堆区重新申请空间，进行拷贝操作**

**注：深拷贝是为了解决浅拷贝重复释放堆区内存导致报错的问题**

```
class Student
{
public:
    Student() {
        cout << "Person无参构造函数调用" << endl;
    }
    Student(int age, int height) {
        my_age = age;
        my_height = new int(height); //在堆区创建区域来存my_height
        cout << "Person有参构造函数调用" << endl;
    }
    Student(const Student& p) {
        my_age = p.my_age;
        my_height = p.my_height;
    }
    ~Student() {
        //释放在堆区开辟的数据
        if (my_height != NULL) {
            delete my_height;
            my_height = NULL;
        }
        cout << "Person析构函数调用" << endl;
    }

    int my_age;
    int* my_height;
};
```

```

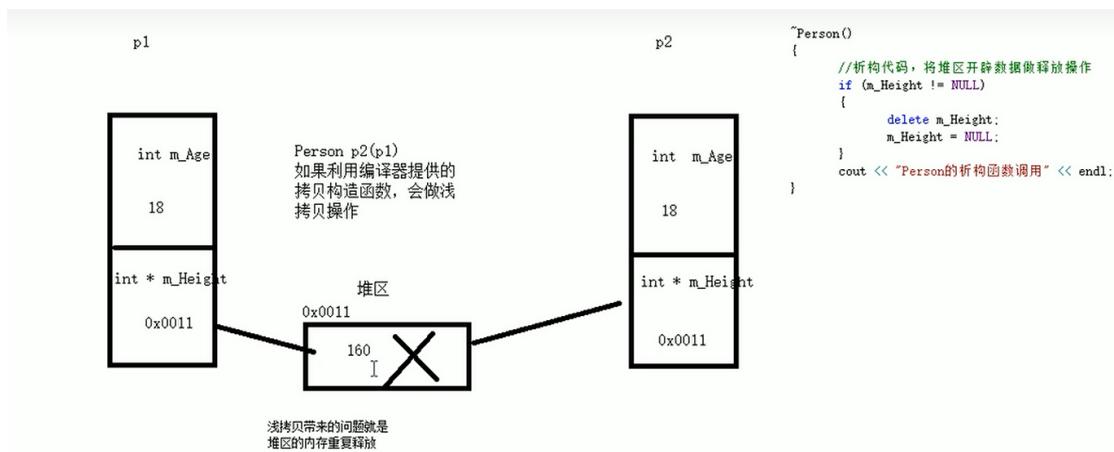
void test() {
    Student p1(18, 160);
    cout << "p1的年龄为: " << p1.my_age << "身高为: " << *p1.my_height << endl;
    Student p2(p1);
    cout << "p2的年龄为: " << p2.my_age << "身高为: " << *p2.my_height << endl;
}

```

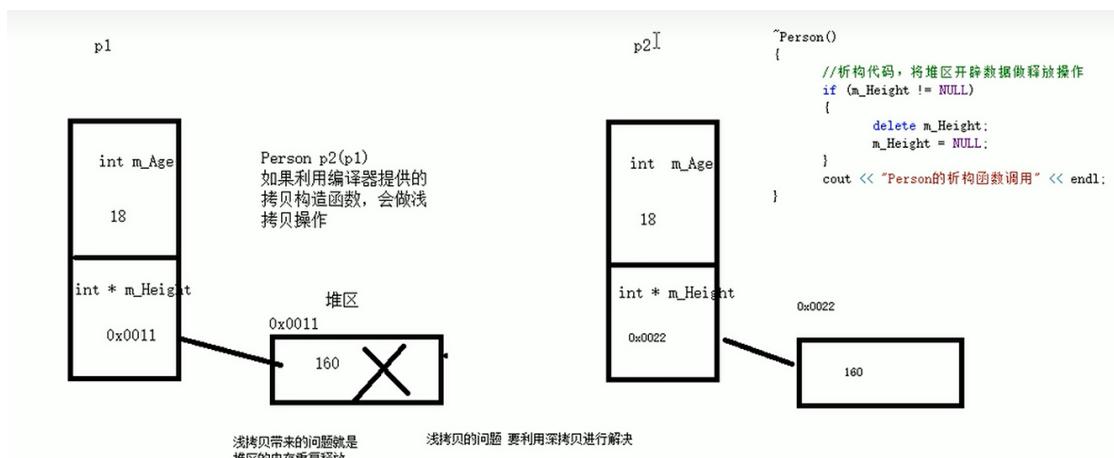
**test () 函数执行完毕之后需要执行类的析构函数;**

**由于函数在堆栈中是后进先出，因此p2先执行析构函数，先释放堆区内存；**

**当p1要执行析构函数释放同一个堆区的内存时，会因为重复释放报错**



**解决办法：在拷贝时在堆区重新开辟一块区域**



```

Student(const Student& p) {
    my_age = p.my_age;
    my_height = new int(*p.my_height); //在堆区重新开辟一块空间
    //my_height = p.my_height;
}

```

## (6) 初始化列表初始化属性:

```
//初始化列表初始化属性
Teacher(string n, int a, string p):name(n), age(a), phone(p)
{}

string name;
int age;
string phone;
};

void test_init() {
    Teacher t1("teacher1", 18, "123145");
    cout << t1.name << " " << t1.age << " " << t1.phone << endl;
}
```

## 四、类对象作为类成员

```
1 class A {}
2 class B
3 {
4     A a;
5 }

class Stu
{
public:

    Stu(string name, Teacher tch): stu_name(name), stu_tch(tch)
    {
    }
    string stu_name;
    Teacher stu_tch;
};

}
```

- 当其他类对象作为本类成员，构造时候先构造类对象，再构造自身
- 析构时相反，先析构自身，再析构类成员

## **静态成员：**

**静态成员就是在成员变量和成员函数前加上关键字static，称为静态成员**

**静态成员分为：**

### **静态成员变量**

- 所有对象共享同一份数据
- 在编译阶段分配内存（全局区）
- 类内声明，类外初始化

### **静态成员函数**

- 所有对象共享同一个函数
- 静态成员函数只能访问静态成员变量

**注：因为静态成员共享，如果是非静态成员，它不知道要访问哪个对象的成员变量**

### **静态成员的两种访问方式**

- 1、通过对象访问
- 2、通过类名访问

```
void test_human2()
{
    //通过对象访问静态变量
    Human h3;
    cout << h3.a << endl;
    //通过类名进行访问
    cout << Human::a << endl;
}
```

注：静态成员变量和静态成员函数也是有访问权限的，私有权限（private）类型的静态成员变量类外无法访问。

## 成员变量和成员函数分开存储：

```
void test_phone() {
    Phone ph1;
    //空对象占内存空间为：1
    //C++编译器会给每个空对象也分配一个字节空间，是为了区分空对象占内存的位置
    //每个空对象也有独一无二的内存地址
    cout << "size of Phone1 is: " << sizeof(ph1) << endl;
}
```

## 五、this指针

- **this指针指向被调用的成员函数所属的对象**
- **this指针是隐含每一个非静态成员内的一种指针**
- **this指针不需要定义，直接使用即可**

---

## 六、const修饰成员函数

### 1、常函数

- 成员函数后加const称该函数为常函数
- 常函数内不可以修改成员属性
- 成员属性声明是加关键字mutable后，在常函数中依然可以修改

## 2、常对象

- 声明对想前加const称该对象为常对象
- 常对象只能调用常函数

```
//常函数
class Item
{
public:
    void showPerson() const //用const修饰成员函数（常函数），其实修饰的是this指针
    //让指针指向的值也不可以修改
    {
        //this指针的本质 是指针常量 指针的指向是不可以修改的
        //I_a = 100;//会报错，相当于this->I_a = 100;
        this->I_b = 100;//不会报错，有关键字mutable修饰
    }
    int I_a;
    mutable int I_b; //加上关键字mutable，使this指针指向的值也可以修改
};

void testItem() {
    const Item i1; //用const修饰对象（常对象）
    i1.I_a = 100; //会报错
    i1.I_b = 10; //不会报错（mutable）
}
```

## 七、友元

友元的目的就是让一个函数或类访问另一个类中的私有成员

友元的关键字为： *friend*

友元的三种实现：

- 全局函数做友元
- 类做友元

- 成员函数做友元
- 

## 八、运算符重载

对已有二点运算符重新定义，赋予其另一种功能，以适应不同的数据类型

以加号运算符重载为例

### (1) 成员函数重载

```
//1、成员函数重载+号
Person operator+(Person& p)
{
    Person temp;
    temp.m_A = this->m_A + p.m_A;
    temp.m_B = this->m_B + p.m_B;
    return temp;
}
```

### (2) 全局函数重载

```
//2、全局函数重载+
Person operator+(Person& p1, Person& p2)
{
    Person temp;
    temp.m_A = p1.m_A + p2.m_A;
    temp.m_B = p1.m_B + p2.m_B;
    return temp;
}
```

## 九、继承

## (一) 语法: class 子类: 继承方式 父类

子类也称为: 派生类

父类也称为: 基类

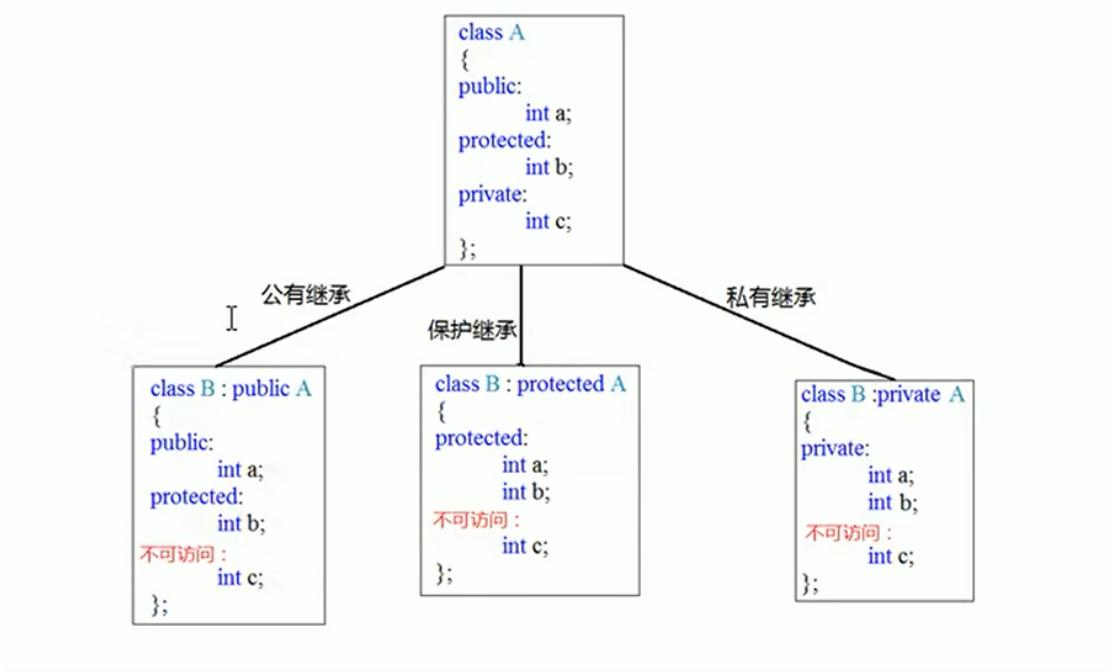
```
class basepage
{
public:
    void begin1() {
        cout << "this is the begin" << endl;
    }
    void end1() {
        cout << "this is the end" << endl;
    }
};

class java:public basepage
{
public:
    void java1()
    {
        cout << "this is the java page" << endl;
    }
};
```

## (二) 继承方式:

继承方式有三种:

- 公共继承
- 保护继承
- 私有继承



### (三) 继承中的对象模型

1、父类中所有的非静态成员都会被子类继承

2、父类中私有成员属性是被编译器给隐藏了，因此虽然访问不到，但确实被继承了

开发人员命令行中查看对象模型

跳转到对应文件夹：

```
D:\Visual Studio>cd D:\vs c++ code\c++HelloWorld\c++HelloWorld
```

cl /d1 reportSingleClassLayout类名 文件名

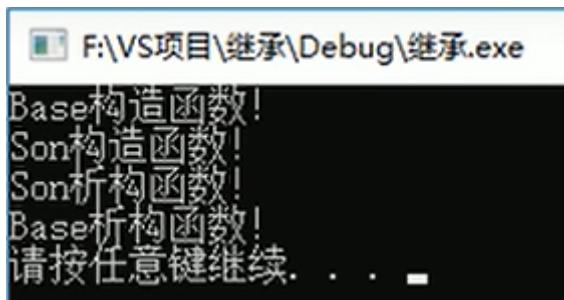
```
D:\vs c++ code\c++HelloWorld\c++HelloWorld>cl /d1 reportSingleClassLayoutSon 继承.cpp
```

用于 x86 的 Microsoft (R) C/C++ 优化编译器 19.28.29333 版  
版权所有 (C) Microsoft Corporation。保留所有权利。

继承.cpp

```
class Son      size(16):
+---          (base class Basel)
0           +--- b_A
0           |   b_B
4           |   b_C
8           +--- s_A
12          +---
```

## (四) 继承中的构造和析构函数顺序



## (五) 继承中同名成员的继承方式

- 访问子类同名成员，直接访问即可
- 访问父类同名成员，需要加作用域

```
Son s;
cout << "Son 下 m_A = " << s.m_A << endl;
cout << "Base 下 m_A = " << s.Base::m_A << endl;
```

s. m\_A 代表的是子类中的成员属性

s. Base::m\_A 代表的是父类中的成员属性

同名成员函数同理：

```
Son s;
s. func(); //直接调用 调用是子类中的同名成员
//如何调用到父类中同名成员函数?
s. Base::func();
```

## (六) 继承中同名静态成员的继承方式

静态成员的特点，回顾之前的笔记

- 访问子类同名静态成员，直接访问即可

- 访问父类同名静态成员，需要加作用域

```
//1、通过对象访问
```

```
cout << "通过对象访问：" << endl;
Son s;
cout << "Son 下 m_A = " << s.m_A << endl;
cout << "Base 下 m_A = " << s.Base::m_A << endl;
```

```
//2、通过类名访问
```

```
cout << "通过类名访问：" << endl;
cout << "Son 下 m_A = " << Son::m_A << endl;
//第一个::代表通过类名方式访问 第二个::代表访问父类作用域下
cout << "Base 下 m_A = " << Son::Base::m_A << endl;
```

同名成员函数同理。

## (六) 多继承语法

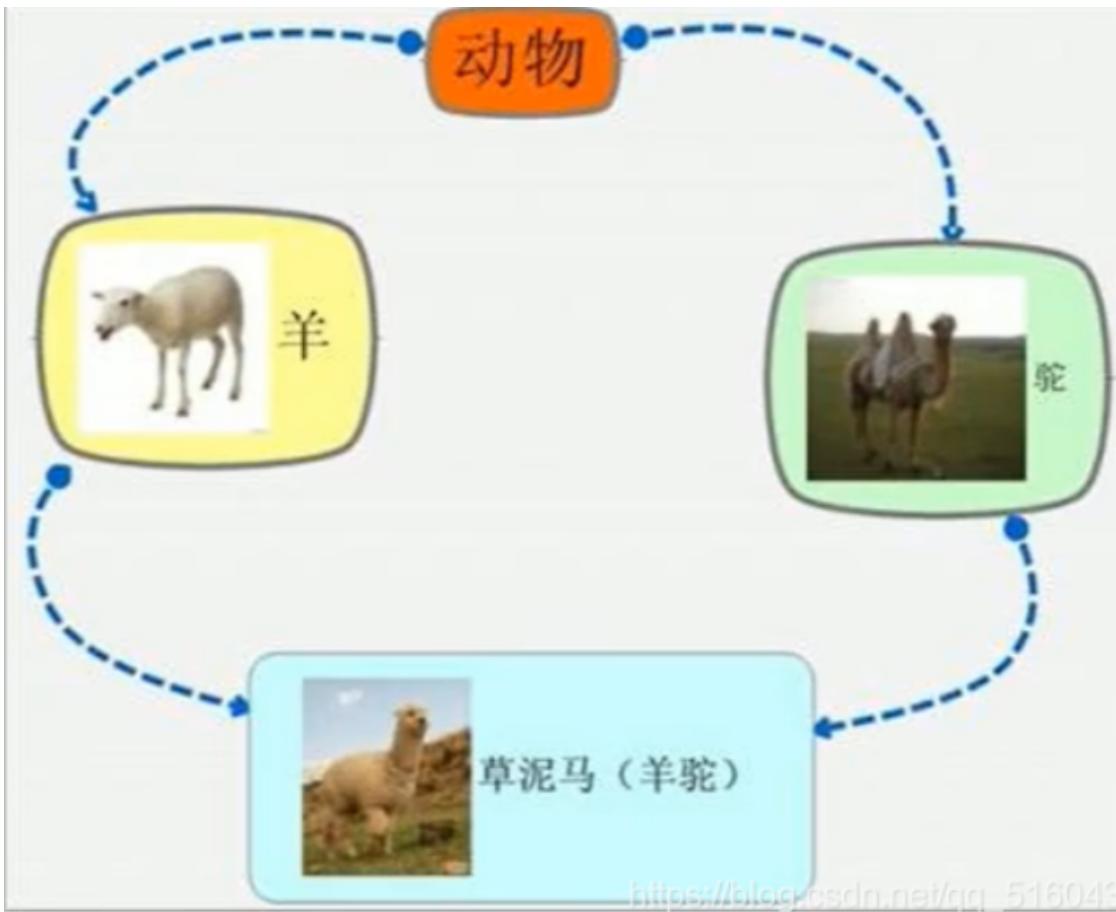
C++允许一个类继承多个类

语法： class 子类：继承方式 父类1，继承方式 父类2....

多继承可能会引发父类中有同名成员出现，需要加作用域区分，C++实际开发中不建议用多继承

## (七) 菱形继承问题

概念：两个派生类继承同一个基类，又有某个类同时继承这两个派生类，这种继承称为菱形继承，或者钻石继承。



问题：

- 1、羊继承了动物的数据，驼同样继承了动物的数据，当草泥马使用数据时，就会产生二义性。
- 2、草泥马继承动物的数据继承了两份，其实我们应该清楚，这份数据我们只需要一份就可以。

解决：

```
class Animal
{
public:
    int m_Age;
};

//利用虚继承可以解决菱形继承问题
//在继承之前加上关键字virtual变为虚继承
// Animal类称为虚基类
//羊
class Sheep:virtual public Animal
{

};

//驼
class Tuo:virtual public Animal
{

};

//羊驼
class SheepTuo :public Sheep,public Tuo
{
```

```
void test01()
{
    SheepTuo st;
    st.Sheep::m_Age = 18;
    st.Tuo::m_Age = 28;
    //当菱形继承，当两个父类拥有相同的数据，需要加作用域来区分
    cout << st.Sheep::m_Age << endl;
    cout << st.Tuo::m_Age << endl;
    cout << st.m_Age << endl;
    //这份数据我们知道，只有一份就可以了，菱形继承导致数据有两份，资源浪费
}
```

## 十、多态

## (一) 概念

多态分为两类：

- 静态多态：函数重载和 运算符载属于静态多态，复用函数名
- 动态多态：派生类和虚函数实现运行时多态

静态多态和动态多态区别：

- 静态多态的函数地址早绑定——编译阶段确定函数地址
- 动态多态的函数地址晚绑定——运行阶段确定函数地址

## (二) 动态多态满足条件：

1、有继承关系

2、子类重写父类的虚函数

## (三) 动态多态使用：

父类的指针或引用 执行子类对象

## (四) 多态原理：

虚函数的本质是添加了一个指向虚函数表的指针，当子类中的函数对父类中的函数进行重写时，会对原来继承的函数进行覆盖

重写前：

Animal类内部结构

vftptr



vftable 表内记录虚函数的地址

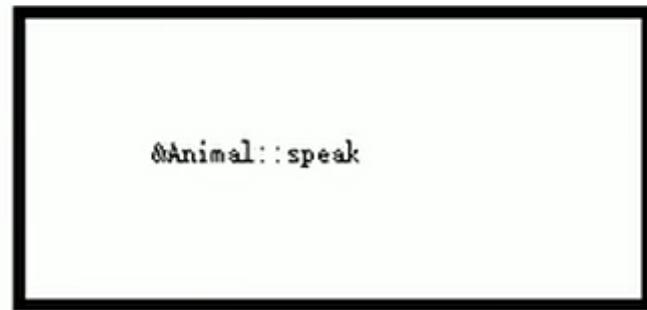


Cat 类内部结构

vftptr



vftable



重写后：

Animal类内部结构

vftptr



vftable 表内记录虚函数的地址



&Animal::speak

Cat 类内部结构

vftptr



vftable



&Cat::speak

以计算器为例：

1、虚函数

```
class AbstractCalculator
{
public:
    virtual int getResult()
    {
        return 0;
    }

    int m_a;
    int m_b;
};
```

## 2、重写覆盖

```
class AddCalculator: public AbstractCalculator
{
public:
    int getResult()
    {
        return m_a + m_b;
    }
};
```

## 3、调用：父类指针执行子类对象

```
void test_calculator()
{
    AbstractCalculator *c1 = new AddCalculator();
    c1->m_a = 10;
    c1->m_b = 5;
    cout << c1->getResult() << endl;
}
```

## (五) 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，主要都是调用子类重写的内容。因此可以将虚函数改为纯虚函数

**纯虚函数语法:** `virtual 返回值类型 函数名 (参数列表) = 0;`

**当类中有了纯虚函数，这个类也称为抽象类**

**抽象类特点:**

- 无法实例化对象
- 子类必须重写抽象类中的纯虚函数，否则也属于抽象类

```
//纯虚函数写法  
virtual int getResult() = 0;
```

## (六) 虚析构和纯虚析构

**多态使用时，如果子类中有属性开辟到堆区，那么父类指针在释放时无法调用到子类的析构代码**

**解决方式：将父类中的析构函数改为虚析构或者纯虚析构**

**虚析构和纯虚析构共性:**

- 可以解决父类指针释放子类对象
- 都需要有具体的函数实现

**虚析构和纯虚析构区别:**

- 如果是纯虚析构，该类属于抽象类，无法实例化对象

**虚析构语法:** `virtual ~类名 () {}`

**纯虚析构语法:** `virtual ~类名 () = 0 ;`

**类名:: ~类名 () {}**

- 如果是纯虚析构，该类属于抽象类，无法实例化对象

**若不使用虚/纯虚析构函数：**

```
Fish构造函数调用  
Shark构造函数调用  
小shark会游泳  
Fish析构函数调用  
请按任意键继续. . .
```

父类指针在析构时候，不会调用子类中析构函数，导致子类如果有堆区属性，出现内存泄露

**解决办法：将父类中的析构函数改为虚析构函数/纯虚析构函数，解决父类指针释放子类对象时不干净的问题**

**虚析构解决：**

```
virtual ~Fish()  
{  
    cout << "Fish析构函数调用" << endl;  
}
```

**纯虚析构解决：**

```
//纯虚析构，需要声明也需要实现  
//有了纯虚析构之后，这个类也属于抽象类，无法实例化对象  
virtual ~Fish() = 0;  
  
};  
  
Fish::~Fish()  
{  
    cout << "Fish纯虚析构函数调用" << endl;  
}
```

```
Fish构造函数调用  
Shark构造函数调用  
小shark会游泳  
Shark析构函数调用  
Fish析构函数调用  
请按任意键继续. . .
```

# 十一、文件操作

C++中对文件操作需要包含头文件 <fstream>

## (一) 文件类型分为两种:

- 1、文本文件：文件以文本的ASCII码形式存储在计算机中
- 2、二进制文件：文件以文本的二进制形式存储在计算机中，用户一般不能直接读懂它们

## (二) 操作文件的三大类:

- 1、ofstream: 写操作
- 2、ifstream: 读操作
- 3、fstream : 读写操作

## (三) 文本文件

写文件步骤如下：

- 1、包含头文件 #include <fstream>
- 2、创建流文件 ofstream ofs;
- 3、打开文件 ofs.open("文件路径", 打开方式);
- 4、写数据 ofs << "写入的数据";
- 5、关闭文件 ofs.close();

文件打开方式

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	如果文件存在先删除, 再创建
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

例如: 用二进制方式写文件 `ios::binary | ios:: out`

## 读文件步骤如下:

1、包含头文件 `#include <fstream>`

2、创建流文件 `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 四种读取方式:

```
//读数据
//第一种
char buf[1024] = { 0 };
while (ifs >> buf)
{
    cout << buf << endl;
}

//第二种
char buf[1024] = { 0 };
while (ifs.getline(buf, sizeof(buf)))
{
    cout << buf << endl;
}
```

```
//第三种
string buf;
while (getline(ifs, buf))
{
    cout << buf << endl;
}

//第四种
char c;
while ((c = ifs.get()) != EOF) // EOF end of file
{
    cout << c;
}
```

5、关闭文件 `ofs.close();`

## (四) 二进制文件

打开方式要指定为 `ios::binary`

### 写文件

二进制方式写文件主要利用流对象调用成员函数**write**

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

参数解释: 字符指针buffer指向内存中一段存储空间。len是读写的字节数

### 读文件

二进制方式读文件主要利用流对象调用成员函数**read**

函数原型: `istream& read(char *buffer, int len);`

参数解释: 字符指针buffer指向内存中一段存储空间, len是读写的字节数

---

# 十二、模板

**概念：模板就是建立通用的模具，大大提高复用性**

**模板的特点：**

- 模板不可以直接使用，它只是一个框架
- 模板的通用并不是万能的

## **(一) 函数模板概念**

- C++除了面向对象另一种编成思想称为泛型编程，主要利用的技术就是模板

**函数模板的作用：**建立一个通用函数，其函数返回值类型和形参类型可以不具体制定，用一个虚拟的类型来代表.

**语法：**

```
template<typename T>
函数声明或定义
```

**解释：**

**template ---声明创建模板**

**typename ---表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**

**两种调用模板函数类型：**

```
//第一种：自动类型推导  
mySwap(a, b);
```

```
//第二种：显示指定类型  
mySwap<int>(a, b);
```

## 函数模板注意事项：

- 自动类型推导，必须推导出一致的数据类型T，才可以使用
- 模板必须要确定出T的数据类型，才可以使用

## (二) 普通函数与函数模板的区别：

### 1、普通函数调用时可以发生自动类型转换（隐式类型转换）

```
//普通函数  
int myAdd01(int a, int b)  
{  
    return a + b;  
}  
  
void test01()  
{  
    int a = 10;  
    int b = 20;  
    char c = 'c'; // a - 97 c - 99  
    cout << myAdd01(a, c) << endl;  
}
```

### 2、函数模板用自动类型推导不可以发生隐式类型转换

```
//自动类型推导  
//cout << myAdd02(a, c) << endl;
```

### 3、函数模板用显示指定类型，会发生隐式类型转换

```
//显示指定类型 会发生隐式类型转换  
cout << myAdd02<int>(a, c) << endl;
```

## (三) 普通函数与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

## (四) 类模板

语法：

```
template <typename T>  
类
```

typename可以用class代替

## (五) 类模板与函数模板区别

- 1、类模板没有自动类型推导的使用方式

```
Doctor<string, int> d1("xg", 45); //正确  
//Doctor d1("xg", 45); //错误，类模板无法自动推导
```

- 2、类模板在模板参数列表中可以有默认参数

```
//类模板
template <class NameType, class AgeType = int>
class Doctor
{
public:
    Doctor(NameType name, AgeType age)
    {
        this->m_Name = name;
        this->m_Age = age;
    }
}
```

## (六) 类模板对象做函数参数

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

```
//1、指定传入类型
void printDoctor1(Doctor<string, int>& p)
{
    p.showDoctor();
}

void test_print1()
{
    Doctor<string, int> p("test1", 44);
    printDoctor1(p);
}
```

2、数模板化--- 将对象中的参数变为模板进行传递

```
//2、参数化模板
template <class T1, class T2>
void printDoctor2(Doctor<T1, T2>& p)
{
    p.showDoctor();
}
void test_print2()
{
    Doctor<string, int> p2("test2", 40);
    printDoctor2(p2);
}
```

### 3、整个类模板化--- 将这个对象类型 模板化进行传递

## (七) 类模板与继承

- 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

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

class Son :public Base<int> //指定父类中的T的数据类型，才能继承
{};


```

- 如果想灵活指定出父类中T的类型，子类也需变为类模板

```
//如果想灵活指定出父类中T的类型，子类也需变为类模板
template <class T1, class T2>
class Son1 :public Base<T2>
{
    T1 obj;
};

void test_Son2()
{
    Son1 <int, char>s2;
```

## (八) 类模板成员函数类外实现

```
//构造函数类外实现
template<class T1, class T2>
Driver<T1, T2>::Driver(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 Driver<T1, T2>::showDriver()
{
    cout << this->m_Name << " " << this->m_Age << endl;
```

## (九) 类模板分文件编写

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

//第二种解决方式，将.h和.cpp中的内容写到一起，将后缀名改为.hpp文件
```

# 十三、STL

## STL基本概念：

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

## STL六大组件:

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

---

## 容器:

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，栈，队列，集合，映射表等。

**这些容器分为序列式容器和关联式容器两种：**

**序列式容器：强调值的排序，序列式容器中的每个元素均有固定的位置。**

**关联式容器：二叉树结构，各元素之间没有严格的物理上的顺序关系。**

## **算法：**

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

**算法分为：质变算法和非质变算法。**

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

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

## **迭代器：**

提供一种方法，使之能够依序寻访某个容器所含的各个元素，而又无需暴露该容器的内部表示方式。

**每个容器都有自己专属的迭代器，迭代器的使用非常类似于指针**

## **迭代器种类：**

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

常用的容器中迭代器种类为双向迭代器和随机访问迭代器。

## (一)Vector

### 1、vector存放内置数据类型

```
容器: vector
算法: for_each
迭代器: vector<int>::iterator
```

### 包含头文件vector、algorithm

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

//创建一个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);
```

### 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元素

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

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

```
//第三种遍历方式 利用STL提供的遍历算法
for_each(v.begin(), v.end(), myPrint);

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

## 2、vector容器嵌套容器

```
//创建大容器
vector < vector<int> > v3;

//创建小容器
vector<int> va;
vector<int> vb;
vector<int> vc;
vector<int> vd;

//向小容器中添加数据
va.push_back(5);
va.push_back(10);
va.push_back(15);
va.push_back(25);

//向大容器中添加数据（小容器）
v3.push_back(va);
v3.push_back(vb);
v3.push_back(vc);
v3.push_back(vd);

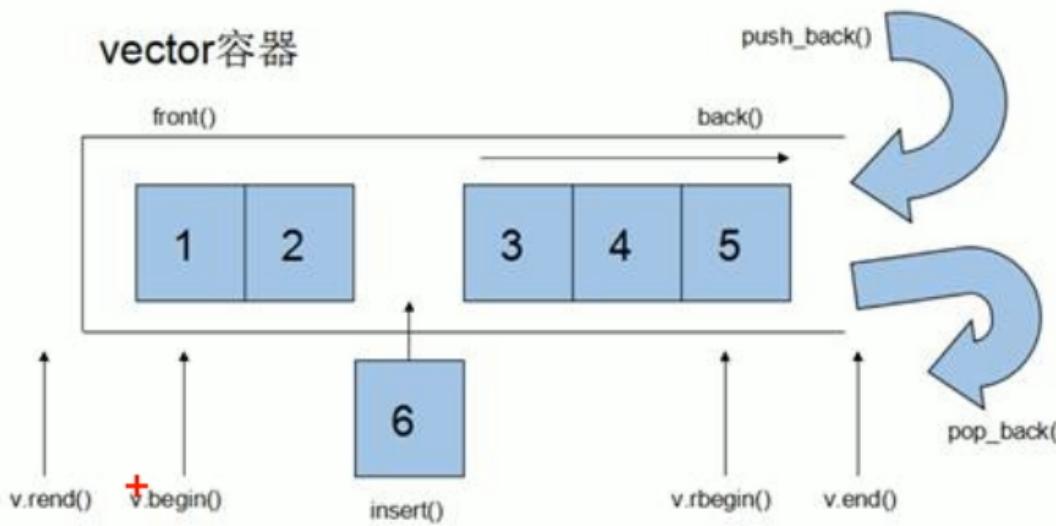
//打印大容器中的数据
for (vector<vector<int>>::iterator it1 = v3.begin(); it1 != v3.end(); it1++)
{
    for (vector<int>::iterator it2 = (*it1).begin(); it2 != (*it1).end(); it2++)
    {
        cout << *it2 << endl;
    }
}
```

### 3、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，而是找更大的内存空间，然后将原数据拷贝新空间，释放原空间。



### 4、vector构造函数

- 默认构造，无参构造

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

- 通过区间方式进行构造

```
vector<int> v2(v1.begin(), v1.end());
```

- n个elem方式构造，生成10个100

```
vector<int>v3(10, 100);
```

### 5、vector赋值操作

- operator=赋值

```
vector<int> v2;
v2 = v1;
```

- assign区间赋值  

```
vector<int> v3;
v3.assign(v1.begin(), v1.end());
```
- n个elem方式赋值, 10个100  

```
vector<int> v4;
v4.assign(10, 100);
```

## 6、vector容量大小

- 判断容器是否为空, 为空返回1, 否则返回0  
`v1.empty();`
- 容器的容量,容量永远大于等于容器大小, 多出来的容量用于扩展  
`v1.capacity();`
- 容器中元素的个数  
`v1.size();`
- 重新指定容器的长度为num, 若容器变长, 则以默认值0填充新位置。
- 如果容器变短, 则末尾超出容器长度的元素被删除  
`v1.resize(int num)`
- 重新指定容器的长度为num, 若容器变长, 则以elem值填充新位置。
- 如果容器变短, 则末尾超出容器长度的元素被删除。  
`v1.resize(int num, elem)`

## 6、vector插入和删除

- 尾部插入元素elem  
`v1.push_back(elem);`
- 删除最后一个元素  
`v1.pop_back();`
- 迭代器指向位置pos插入元素elem(插入位置只能是迭代器指向的位置)  

```
v1.insert(v1.begin(), 9);
```
- 迭代器指向位置pos插入count个元素elem,末尾插入5个0  

```
v1.insert(v1.end(), 5, 0);
```
- 删除迭代器指向的元素  

```
v1.erase(v1.begin());
```
- 删除迭代器从start到end之间的元素(位置只能用迭代器)  

```
v1.erase(v1.begin(), v1.end());
```
- 删除容器中所有元素  

```
v1.clear();
```

## 7、vector数据存取

- at返回索引所指的数据  

```
int i = v1.at(0);
```
- []返回索引所指的数据  

```
int j = v1[1];
```
- front()返回容器中第一个数据元素

```
cout << v1.front() << endl;
```

- back()返回容器中最后一个数据元素  

```
cout << v1.back() << endl;
```

## 8、vector容器互换

`swap(vec);` 实现vec与本身的元素互换

```
v1.swap(v2);
```

- 实际用途：巧用swap可以收缩内存空间

//巧用swap收缩内存

```
vector<int>(v).swap(v);
```

```
cout << "v的容量为: " << v.capacity() << endl;
cout << "v的大小为: " << v.size() << endl;
```

```
v的容量为: 3
v的大小为: 3
```

## 9、vector预留空间

减少vector在动态扩展容量时的扩展次数

```
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。
```

```
vector<int> v;
int num = 0; //统计开辟次数
int* p = NULL;

for (int i = 0; i < 10000; i++)
{
    v.push_back(i);
    if (p != &v[0])
    {
        p = &v[0];
        num++;
    }
}

cout << "总共开辟内存的次数: ";
cout << "num=" << num << endl;
```

```
总共开辟内存的次数: num=24
```

预留空间后：

```
vector<int> v;
v.reserve(10000);

int num = 0; //统计开辟次数
int* p = NULL;
for (int i = 0; i < 10000; i++)
{
    v.push_back(i);
    if (p != &v[0])
    {
        p = &v[0];
        num++;
    }
}

cout << "总共开辟内存的次数: ";
cout << "num=" << num << endl;
```

```
总共开辟内存的次数: num=1
```

## (二)String

**String本质：**

string是C++风格的字符串，而string本质上是一个类

## string和char \*区别

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

特点：

string类内部封装了很多成员方法

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

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

## string构造函数

1、构造函数原型：

- string(); //创建一个空的字符串，例如：string str;  
`string str1 = "asfgag";`
- string(const char\* s); //使用字符串s初始化  
`const char* s1 = "char str2";  
string str2(s1);`
- string(const string& str); //使用一个string 对象初始化另一个string对象  
`string str3(str2);`
- string(int n, char c); //使用n个字符c初始化  
`string str4(5, 'a');`

2、string字符串拼接：

- char\*类型和字符串类型赋值给当前字符串

```
string str1;  
str1 = "aaa";
```

```
string str2;
str2 = str1;
```

- `assign`把字符串赋给当前的字符串

```
string str3;
str3.assign("bbb");
```

```
string str3;
str4.assign(str3);
```

- `assign`将字符串前五个字符赋给当前字符串

```
string str5;
str5.assign("hello world", 5);
```

- `assign`用n个字符c赋值给当前字符串

```
string str6;
str6.assign(10, 'c');
```

### 3、字符串拼接

- 重载`+=`操作符

```
string str1 = "aaa";
string str2 = "bbb";
str1 += "ccc";
str1 += str2;
```

- `append`把字符串s连接到当前字符串结尾

```
str1.append("ddd");
str3.append(str2);
```

- `append`把字符串s的前n个字符连接到当前字符串结尾

```
str4.append("Hello World", 5);
str4.append(str3, 5);
```

- `append`把字符串s中pos开始的（不包括pos）的n个字符串连接到字符串结尾

```
#str4.append(s, pos, n)
str4.append(str3, 0, 5);
```

## 4、string查找和替换

**功能：查找指定字符串是否存在**

**替换：在指定的位置替换字符串**

- `find`查找str在当前字符串中第一次出现,从pos位置开始查找

```
#int find(str, pos)
string s1 = "abcdefg";
pos = s1.find('e', 3);
```

- `rfind`查找str在当前字符串中第一次出现，从pos位置开始查找

```
pos = s1.rfind('e', 3);
```

**find是从左往右查找，rfind是从右往左查找，但是返回的位置都是从左往右计数的。**

- 从ops位置查找s的前n个字符第一次位置

```
string s2 = "apple";
s2.replace(4, 1, "e");
```

apple  
请按任意键继续. . . ■

## 5、string字符串比较

**字符串是按字符的ASCII码进行比较**

```
= 返回 0 ; > 返回 1 ; < 返回 -1
string s1 = "abcdef";
string s2 = "fg";
cout << s1.compare(s2) << endl;
```

D:\vs c++ code\c++HelloWorld\Debug  
-1  
请按任意键继续. . .

## 6、string字符存取

- 通过[]方式取字符

```
for (int i = 0; i < str.size(); i++)
{
    cout << str[i] << " ";
}
cout << endl;
```

//修改单个字符  
str[0] = 'i';  
cout << str << endl;

- 通过at方式获取字符

```
for (int i = 0; i < str.size(); i++)
{
    cout << str.at(i) << endl;
}
cout << endl;
```

str.at(1) = 'i';  
cout << str << endl;

## 7、string插入和删除

- 在pos位置插入字符串

```
string str = "146";
str.insert(2, "5");
```

- 在pos位置插入n个字符c

```
str.insert(6, 3, 'a');
```

- 删除从pos开始的n个字符

```
str.erase(6, 3);
```

## 8、string子串获取

- 利用string.substr(int a,int b)

```
string str = "123456";
string str1 = str.substr(0, 3); // (0, 3]
```

- 获取邮件中用户名信息

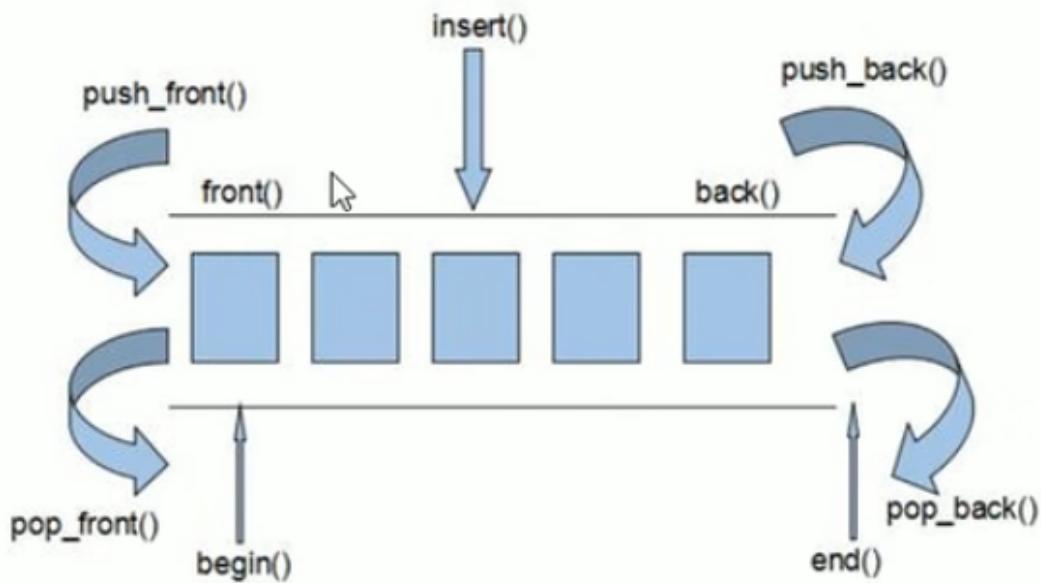
```
string email = "asfngb@qq.com";
int pos = email.find("@");
string userName = email.substr(0, pos);
```

### (三) Deque容器

功能：双端数组，可以对头端进行插入删除操作

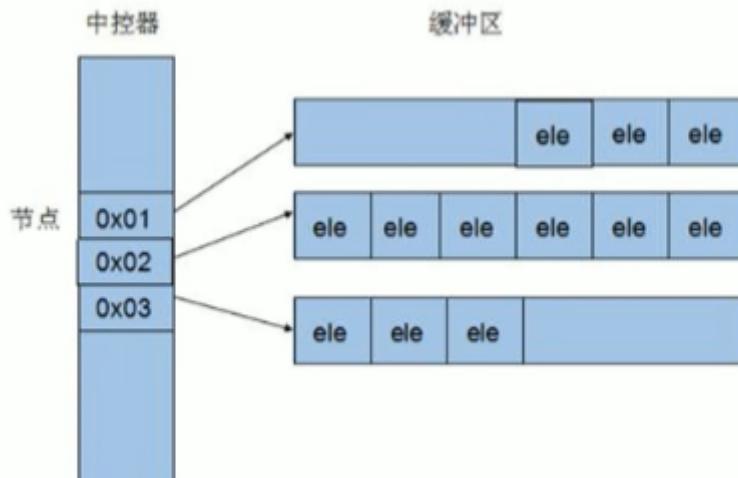
#### 1、deque与vector区别：

- 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



#### 2、deque内部工作原理

deque内部有个中控器，维护每段缓冲区中的内容，缓冲区中存放真实数据。  
中控器维护的是每个缓冲区的地址，使得使用deque时像一片连续的内存空间。



### 3、deque容器构造

- 默认构造形式

```
deque<int> dq1;
```

- 构造函数将[begin,end]区间中的元素拷贝给本身

```
deque<int> dq2(dq1.begin(), dq1.end());
```

- 构造函数将n个elem拷贝给本身

```
deque<int> dq3(10, 100);
```

- 拷贝构造函数

```
deque<int> dq4(dq3);
```

### 4、deque赋值操作

- 重载等号赋值操作

```
deque<int> d1(10, 100);
```

- 将[begin,end]区间中的元素拷贝给本身

```
deque<int>d3;
d3.assign(d2.begin(), d2.end());
```

- 将n个elem拷贝给本身

```
deque<int>d4;
d4.assign(10, 11);
```

## 5、deque大小操作

- 判断容器是否为空

```
deque.empty();
```

- 返回容中元素的个数

```
deque.size();
```

- 重新指定容器的长度为num,若容器变长，则以默认值0填充新位置；如果容器变短，则末尾超出容器长度的元素被删除。

```
deque.resize(num);
d1.resize(15);
0 1 2 3 4 5 6 7 8 9 0 0 0 0 0
```

- 重新指定容器的长度为num,若容器变长，则以elem值填充新位置；如果容器变短，则末尾超出容器长度的元素被删除。

```
deque.resize(num, elem);
d1.resize(15, 9);
0 1 2 3 4 5 6 7 8 9 9 9 9 9 9
```

注意：deque容器没有获取容量函数，因为deque容器没有容量概念。

## 6、deque插入和删除

两端插入操作：

- `push_back(elem);` 在容器尾部添加一个数据
- `push_front(elem);` 在容器头部添加一个数据
- `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, begin, end)` 在pos位置插入[begin,end)区间的数据，无返回值。
- `clear();` 清空容器的所有数据
- `erase(begin, end);` 删除[begin,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。

## 7、`deque`数据存取

- `at(int index);` 返回索引idx所指的数据
- `[]` 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

```
dq1.at(9)  
dq1[8]  
dq1.front()  
dq1.back()
```

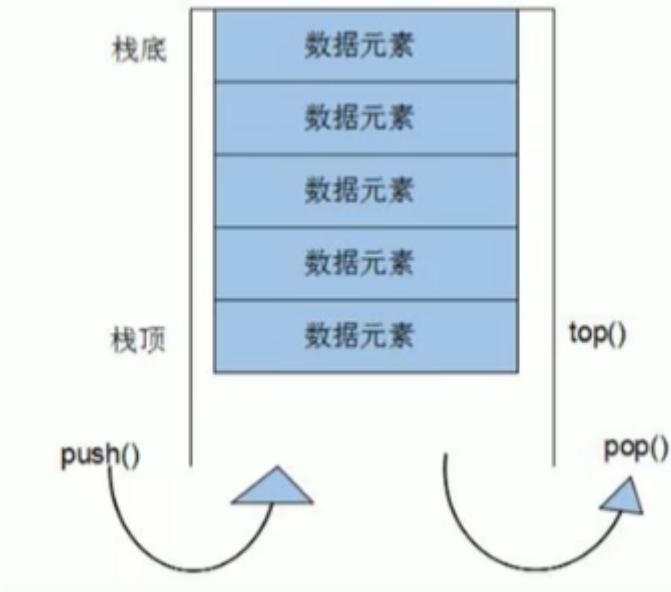
## 8、`deque`排序

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

```
sort(d1.begin(), d1.end());
```

## (四)`stack`容器

**概念：**`stack`是一种先进后出(First In Last Out,FIL)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

## 常用操作：

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

```
stack<int> st1;
st1.push(5);
st1.push(10);
st1.push(15);

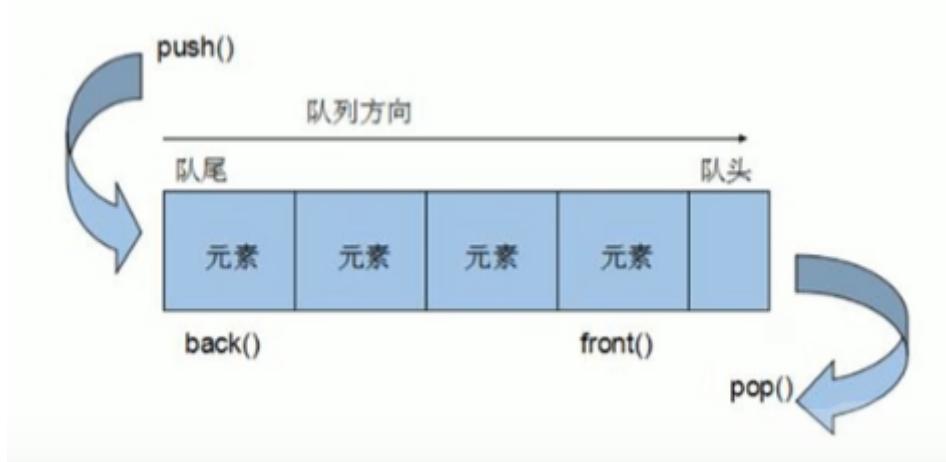
stack<int> st2(st1);
st2.pop();
cout << st2.top() << endl;

stack<int> st3;
cout << st3.empty() << endl;

cout << st2.size() << endl;
```

## (五)queue容器

**概念：** Queue是一种先进先出(First in First Out,FIFO)的数据结构，它有两个出口。



队列容器允许从一端新增元素，从另一端移除元素；队列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。

**常用操作：**

构造函数:

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

赋值操作:

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

数据存取:

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

大小操作:

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

## (六)list容器

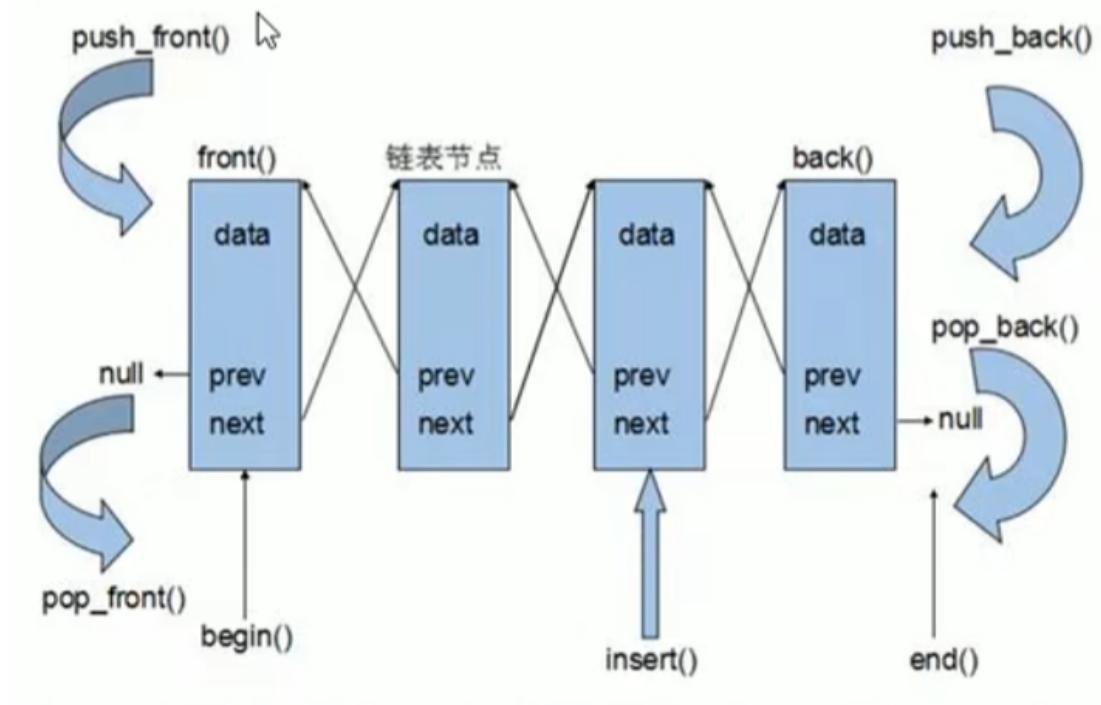
**概念:** 链表(list)是一种物理存储单元上非连续的存储结构, 数据元素的逻辑顺序是通过链表中的指针链接实现的。

**链表的组成:** 链表由一系列结点组成

**结点的组成:** 一个是存储数据元素的数据域, 另一个是存储下一个结点地址的指针域

**STL中的链表是一个双向循环链表**

链表结构：



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

## list的优点:

- 采用动态存储分配，不会造成内存浪费和溢出
- 链表执行插入和删除操作十分方便，修改指针即可，不需要移动大量元素

## list的缺点:

- 链表灵活，但是空间(指针域) 和 时间(遍历) 额外耗费较大

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。

Vector和List的区别:<https://blog.csdn.net/Vcrossover/article/details/111188049>

## 1、list容器构造

- 默认构造

```
list<int> ls1;
```

- 拷贝构造函数

```
list<int> ls2(ls1);
```

- 将n个elem拷贝给本身

```
list<int> ls3(5, 3);
```

- 将[begin,end]区间中的元素拷贝给本身

```
list<int> ls4(ls3.begin(), ls3.end());
```

## 2、list赋值和交换

- push\_back赋值

```
list<int> ls1;
ls1.push_back(5);
ls1.push_back(10);
ls1.push_back(15);
```

- assign将[begin,end]区间中的元素拷贝给本身

```
ls2.assign(ls1.begin(), ls2.end());
```

- assign将n个elem拷贝给本身

```
ls3.assign(10, 3);
```

- 重载等号操作符

```
list<int> ls4;
ls4 = ls3;
```

- 将lst与本身的元素互换

```
list<int> ls5;
ls5.swap(ls4);
printList(ls5);
```

## 3、list容器大小操作

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

## 4、list容器插入和删除

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

## 5、list数据存取

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

list不是连续线性空间存储数据，迭代器也不支持随机访问(遍历时不能用指针`it++`)，不能用`at`或`[]`访问list容器中指定位置的元素。

## 6、list反转和排序

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

所有不支持随机访问迭代器的容器，不可以用标准算法。

例：`sort()`括号中不需要`iterator begin(), iterator end()`  
直接使用：

L1.sort();

## (七)set容器

概念：

- 所有元素都会在插入时自动被排序
- 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

### 1、set构造和赋值

创建set容器

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

赋值

```
s1.insert(10);
s1.insert(30);
s1.insert(20);
s1.insert(20);
s1.insert(20);
```

```
10 20 30
10 20 30
```

请按任意键继续. .

由上图可知，set容器所有元素插入时自动排序，且不会插入重复的值。

### 2、set大小和交换

- `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

### 3、set插入和删除

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

- 删除迭代器所指向的位置的元素

```
s1.erase(s1.begin());
```

- 删除迭代器所指向的区间的元素

```
s1.erase(s1.begin(), s1.end());
```

- 删除容器中值为elem的元素

```
s1.erase(10);
```

### 4、set查找和统计

- 查找key是否存在,若存在，返回该键的元素的迭代器; 若不存在，返回set.end():

```
set<int>::iterator pos = s1.find(30);
```

- 统tkey的元素个数

```
cout << s1.count(10) << endl;
```

### 5、multiset

```
multiset<int> ms;
ms.insert(10);
ms.insert(15);
ms.insert(15);
ms.insert(15);
```

10 15 15 15  
请按任意键继续. .

允许插入重复的值

### 6、pair对组创建

成对出现的数据，利用对组可以返回两个数据

## 创建

```
pair<string, int> n("Lin", 18);
pair<string, string> m = make_pair("Zhu", "pig");
```

## 获取值

```
cout << n.first << ":" << n.second << endl;
```

## 7、set容器排序

利用仿函数，改变set容器从大到小的排序规则。

```
//set利用仿函数改变排序规则
class MyCompare
{
public:
    //第一个()代表重载的符号，第二个()代表函数的参数列表
    bool operator()(int v1, int v2)
    {
        return v1 > v2;
    }
};
```

```
void test_set4()
{
    set<int> s1;

    s1.insert(10);
    s1.insert(30);
    s1.insert(20);
    s1.insert(60);
    s1.insert(25);
    printSet(s1);

    //指定排序规则为从大到小

    set<int, MyCompare> s2;

    s2.insert(10);
    s2.insert(30);
    s2.insert(20);
    s2.insert(60);
    s2.insert(25);

    for (set<int, MyCompare>::iterator it = s2.begin(); it != s2.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;
}
```

## (八)map容器

### 概念

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

### 本质

map/multimap属于关联式容器, 底层结构是用二叉树实现。

### 优点:

- 可以根据key值快速找到value值

### map和multimap区别:

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

## 1、map构造和赋值

```
map<string, int> m1;  
  
m1.insert(pair<string, int>("oin", 18));  
m1.insert(pair<string, int>("lin", 38));  
m1.insert(pair<string, int>("asg", 68));  
  
//拷贝构造  
map<string, int> m2(m1);
```

## 2、map大小和交换

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

## 3、map插入和删除

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

## 4、map查找和统计

- 查找key是否存在，若存在，返回改键的元素的迭代器；若不存在，返回`set.end();`

```
map<string, int>::iterator pos = m1.find("oin");
if (pos != m1.end())
{
    cout << "该元素的值为: " << (*pos).second << endl;
}
```

- 统计key的元素个数

```
cout << "键为oin的元素个数为: " << m1.count("oin") << endl;
```

map的count只能为1，因为map不允许插入重复key元素的值，如果重复插入，只有第一次插入的值生效；multimap的count可以大于1。

## 5、排序

- 利用仿函数改变排序规则

```
class MyCompare
{
public:
    bool operator()(int v1, int v2)
    {
        //降序
        return v1 > v2;
    }
};

//map 利用仿函数改变排序规则
void test_map3()
{
    map<int, int, MyCompare>m;
    m.insert(pair<int, int>(1, 10));
    m.insert(make_pair(9, 5));
    m.insert(pair<int, int>(7, 15));
    m.insert(make_pair(6, 73));

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

---

# 十四、STL-函数对象

## 1、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(仿函数)是一个类，不是一个函数

## 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。

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

void test_MyAdd()
{
    MyAdd myAdd;
    cout<<myAdd(10, 10)<<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 test_MyPrint()
{
    MyPrint myPrint;
    myPrint("aaaa");
    myPrint("aaaa");
    myPrint("aaaa");
    myPrint("aaaa");
    myPrint("aaaa");
    cout << "总共调用次数: " << myPrint.count << endl;
}
```

- 函数对象可以作为参数传递。

```
void doPrint(MyPrint &mp, string test)
{
    mp(test);
}

void test_doPrint()
{
    MyPrint myPrint;
    doPrint(myPrint, "Hello c++");
}

int main()
{
    //test_MyAdd();
    //test_MyPrint();

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

## 十五、谓词

### 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

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

- 如果operator()接受两个参数，那么叫做二元谓词

```
class MyCompare
{
public:
    bool operator() (int val1, int val2)
    {
        return val1 > val2;
    }
};
```

## 十六、内建函数对象

### 概念：

STL内建了一些函数对象

### 分类：

- 算数仿函数
- 关系仿函数
- 逻辑仿函数

### 用法：

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

```
include<functional>
```

## 1、算术仿函数

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：

- 一元取反仿函数

```
negate<int>n;
cout << n(50) << endl;
```

- 算术仿函数：

加法：

```
//plus 二元仿函数 加法
plus<int>p1;
cout << p1(10, 30) << endl;
```

- 减法：minus
- 乘法：multiplies
- 除法：divides
- 取模：modulus

## 2、关系仿函数

仿函数原型：

```
vector<int>v;
v.push_back(86);
v.push_back(64);
v.push_back(12);
v.push_back(16);
v.push_back(54);

//升序
sort(v.begin(), v.end(), less<int>());
```

- 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

## 3、逻辑仿函数

- `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

```
//利用逻辑非 将容器v1 搬运到容器v2中，并执行取反操作
vector<bool>v2;
v2.resize(v1.size());
transform(v1.begin(), v1.end(), v2.begin(), logical_not<bool>());
```

## 十七、STL-常用算法

概述：

- 算法主要是由头文件`<algorithm><functional><numeric>`组成。
- `<algorithm>`是所有STL头文件中最大的一个，范围涉及到比较、交换、查找、遍历操作、复制、修改。
- `<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。
- `<functional>`定义了一些模板类，用以声明函数对象。

### (1) `for_each` 遍历

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

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

    for_each(v1.begin(), v1.end(), print01);
    cout << endl;

    for_each(v1.begin(), v1.end(), print02());
    cout << endl;
}
```

## (2) transform 搬运容器到另一个容器

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

```
//常用遍历算法transform
class Transform
{
public:
    int operator() (int val)
    {
        return val + 100;
    }
};

void test_transform()
{
    vector<int>v1;
    vector<int>v2;
    for (int i = 0; i < 10; i++) {
        v1.push_back(i);
    }
    v2.resize(v1.size()); //目标容器需要提前开辟空间

    transform(v1.begin(), v1.end(), v2.begin(), Transform());
    for_each(v2.begin(), v2.end(), print02());
    cout << endl;
}
```

### (3) 查找算法

#### 1. find 查找元素

- 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);`

```

class Person2
{
public:
    Person2(string name, int age)
    {
        this->p_name = name;
        this->p_age = age;
    }

    //重载==，让底层find知道如何对比person数据类型
    bool operator==(const Person2& p)
    {
        if (this->p_name == p.p_name && this->p_age == p.p_age)
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    string p_name;
    int p_age;
};

void test_find()
{
    Person2 p1("Li", 16);
    Person2 p2("Lin", 18);
    Person2 p3("Li", 20);

    vector<Person2> v;
    v.push_back(p1);
    v.push_back(p2);
    v.push_back(p3);
    vector<Person2>::iterator it = find(v.begin(), v.end(), Person2("Li", 20));

    if (it == v.end())
    {
        cout << "没有找到" << endl;
    }
    else
    {
        cout << "找到了" << endl;
    }
}

```

## 2. find\_if 按条件查找元素

- 函数原型

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

```

//查找年龄大于20的人
class Greater20
{
public:
    bool operator () (Person2& p)
    {
        return p.p_age > 20;
    }
};

void test_findif()
{
    Person2 p1("Li", 16);
    Person2 p2("Lin", 18);
    Person2 p3("Li", 20);

    vector<Person2> v;
    v.push_back(p1);
    v.push_back(p2);
    v.push_back(p3);
    vector<Person2>::iterator it = find_if(v.begin(), v.end(), Greater20());

    if (it == v.end())
    {
        cout << "没有找到" << endl;
    }
    else
    {
        cout << "找到了" << endl;
    }
}

```

### 3. adjacent\_find 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`

```

vector<int>::iterator it = adjacent_find(v.begin(), v.end());

```

### 4. binary\_search 二分查找法

- 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`

```

bool isInside1 = binary_search(v.begin(), v.end(), 5);

```

输出：

```
0  
1  
请按任意键继续. .
```

## 5. count 统计元素个数

- 函数原型

### 函数原型：

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

// 统计元素出现次数

- 统计内置数据类型

```
int count1 = count(v.begin(), v.end(), 0);
```

- 统计自定义数据类型

```
class Person2  
{  
public:  
    Person2(string name, int age)  
    {  
        this->p_name = name;  
        this->p_age = age;  
    }  
  
    //重载==，让底层find知道如何对比person数据类型  
    bool operator==(const Person2& p)  
    {  
        if (this->p_name == p.p_name && this->p_age == p.p_age)  
        {  
            return true;  
        }  
        else  
        {  
            return false;  
        }  
    }  
  
    int count1 = count(v.begin(), v.end(), pp);
```

## 6. count\_if 按条件统计元素个数

### 函数原型：

- `count_if(iterator beg, iterator end, _Pred);`

- 统计内置数据类型

```
class Greater2
{
public:
    bool operator() (int val)
    {
        return val > 2;
    }
};
```

```
//按条件统计内置数据类型元素个数—count_if
int count2 = count_if(v.begin(), v.end(), Greater2());
```

- 统计自定义数据类型

```
//查找年龄大于20的人
class Greater20
{
public:
    bool operator() (Person2& p)
    {
        return p.p_age > 20;
    }
};
```

```
//按条件统计自定义数据类型元素个数—count_if
int count2 = count_if(v.begin(), v.end(), Greater20());
```

## (4) 排序算法

### 1. sort 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`

- 升序`sort(v.begin(), v.end());`
- 降序`sort(v.begin(), v.end(), greater<int>());`
- `greater<int>()`为系统自带的谓词，需要`#include<functional>`

### 2. random\_shuffle 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`

```
random_shuffle(v.begin(), v.end());
```

3. merge 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);
```

```
merge(v.begin(), v.end(), v2.begin(), v2.end(), v3.begin());
```

注：1、合并的两个容器必须是有序的；2、合并前目标容器必须要先分配空间

4. reverse 反转指定范围的元素

- 函数原型

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

```
reverse(v3.begin(), v3.end());
```

## (5) 拷贝和替换算法

1. copy 容器内指定范围的元素贝到另一容器中

- 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

```
copy(v1.begin(), v1.end(), v2.begin());
```

注：拷贝前目标容器必须要先分配空间

2. replace 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);
```

```
replace(v2.begin(), v2.end(), 9, 999);
```

3. replace\_if 容器内指定范围满足条件的元素替换为新元素

- 函数原型

```
replace_if(iterator beg, iterator end, _pred, newvalue);
```

```

class Greater2
{
public:
    bool operator() (int val)
    {
        return val > 2;
    }
};

replace_if(v2.begin(), v2.end(), Greater2(), 0);

```

#### 4. swap 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

## (6) 算术生成算法

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

#### 1. accumulate 计算容器元素累计总和

- 函数原型

`accumulate(iterator beg, iterator end, value);`

```
int total = accumulate(v1.begin(), v1.end(), 0);
```

#### 2. fill 将指定区间填充成指定数据

- 函数原型

`fill(iterator beg, iterator end, value);`

```
fill(v2.begin(), v2.end(), 0);
```

v2为：

0 1 2 3 4 5 6 7 8 9

将v2从头到尾填充为0

0 0 0 0 0 0 0 0 0 0

## (7) 常用集合算法

以下三个算法共同特点：1、两个集合必须是有序序列；2、目标容器必须先开辟空间；3、该算法返回的是目标容器的结束迭代器

### 1. set\_intersection 求两个容器的交集

- 函数原型

```
set_intersection(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest);
```

// 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

```
//获取交集
vector<int>v3;
v3.resize(min(v1.size(), v2.size()));
vector<int>::iterator it1 = set_intersection(v1.begin(), v1.end(), v2.begin(), v2.end(), v3.begin());
```

### 2. set\_union 求两个容器的并集

- 函数原型

```
set_union(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest);
```

```
//获取并集
vector<int>v4;
v4.resize(v1.size() + v2.size());
vector<int>::iterator it2 = set_union(v1.begin(), v1.end(), v2.begin(), v2.end(), v4.begin());
```

### 3. set\_difference 求两个容器的差集

- 函数原型

```
set_difference(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest);
```

```
vector<int>v5;
v5.resize(max(v1.size(), v2.size()));
vector<int>::iterator it3 = set_difference(v1.begin(), v1.end(), v2.begin(), v2.end(), v5.begin());
vector<int>v6;
v6.resize(max(v1.size(), v2.size()));
vector<int>::iterator it4 = set_difference(v2.begin(), v2.end(), v1.begin(), v1.end(), v6.begin());
```

```
v1:  
0 1 2 3 4 5 6 7 8 9  
v2:  
5 6 7 8 9 10 11 12 13 14  
v1和v2的差集:  
0 1 2 3 4  
v2和v1的差集:  
10 11 12 13 14
```

