

# C++核心编程技术

## 1 内存分区模型

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（全局const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（包括局部const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。（new操作与delete操作）

**内存四分区的意义：**

不同区域存放的数据，赋予不同的生命周期，给程序员更强的灵活编程

### 1.1 程序运行前

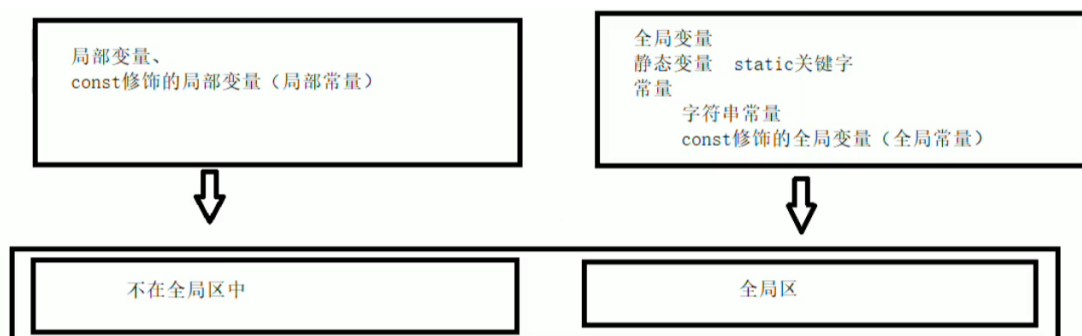
在程序编译后，生成了exe可执行文件，**未执行该程序前**分为两个区域

**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
- 全局区还包含了常量区，字符串常量和常量也存放在此
- 该区域的数据在程序结束后有操作系统释放



**总结：**

- C++中在程序运行前分为全局区和代码区
- 代码区的特点是共享和只读
- 全局区中存放全局变量、静态变量、常量
- 常量区中存放const修饰的全局变量和字符串变量

## 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

**堆区：**

- 由程序员分配释放，若程序员不释放，程序结束有操作系统回收
- 在C++中主要利用new在堆区开辟内存

## 1.3 new操作符

C++中利用new操作符在堆区开辟数据

堆区开辟的数据，由程序员手动开辟，手动释放，释放利用操作符delete

语法： `new 数据类型`

利用new创建的数据，会返回该数据对应的类型的指针

# 2 引用

---

## 2.1引用的基本作用

**作用：**给变量起别名

**语法：** `数据类型 &别名 = 原名`

## 2.2应用注意事项

- 引用必须初始化
- 引用在初始化后，不可以改变

## 2.3 引用左函数参数

**作用：**参数传参时，可以利用引用的技术让形参修饰实参

**优点：**可以简化指针修改实参

**总结：**通过引用参数产生的效果同按地址传递一样的。引用的语法更清楚简单。

## 2.4 引用做函数返回值

**作用：**引用是可以作为函数的返回值存在的

**注意：**不要返回局部变量应用

**用法：**函数调用作为左值

## 2.5引用的本质

**本质：**引用的本质在C++内部实现是一个指针常量。

C++推荐引用技术，因为语法方便，引用的本质是指针常量，但是所有的指针操作编译器帮我们做了。

## 2.6常量引用

**作用：**常量引用主要用来修饰形参，防止误操作

在函数形参列表中，可以加const修饰形参，防止形参改变实参。

# 3 函数提高

---

### 3.1 函数默认参数

在C++中，函数的形参列表中的形参是可以有默认值的

**语法：**返回值类型 函数名 （参数 = 默认值）{ }

**注意：**

1. 如果某个位置已经有了默认值，那么从这个位置往后，从左到右，都必须有默认值。（防止二义性）
2. 如果函数声明有默认参数，函数实现就不能有默认参数（防止重定义错误）

### 3.2 函数占位参数

C++中函数的形参列表可以有占位参数，用来做占位，调用函数时必须填补该位置

**语法：**返回值类型 函数名 （数据类型）{ }

在现阶段函数占位参数存在意义不大，有些地方会用到

### 3.3 函数重载

### 3.3.1 函数重载概述

**作用：** 函数名可以相同，提高复用性

#### 函数重载满足条件

- 同一个作用域下
- 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：** 函数的返回值类型不可以作为函数重载的条件

### 3.3.2 函数重载注意事项

- 引用作为重载条件
- 函数重载碰到函数默认函数

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

//函数重载的注意事项
//1、引用作为重载的条件
void fun(int& a) //int &a =10;不合法
{
    cout << "fun(int& a)函数的调用" << endl;
}
void fun(const int& a) //const int &a =10; 合法
{
    cout << "fun(const int& a)函数的调用" << endl;
}
//2、函数重载碰到默认参数
void fun2(int a,int b)
{
    cout << "fun2(int a, int b)函数的调用" << endl;
}
void fun2(int a)
{
    cout << "fun2(int a)函数的调用" << endl;
}

int main()
{
    int a = 10;
    fun(a); //调用fun(int& a)
    fun(10); //调用fun(const int& a)
    //fun2(a); //当函数重载碰到默认参数，出现二义性，报错，尽量避免这种情况
    fun2(10, 11);
    system("pause");
    return 0;
}
```

## 4 类与对象

C++面向对象的三大特性为：**封装、继承、多态**

C++认为万事万物皆为对象，对象上有其属性和行为

具有相同性质的对象，我们可以抽象为类，人属于人类，车属于车类

## 4.1封装

### 4.1.1 封装的意义

封装是C++面向对象三大特性之一

封装的意义：

- 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

**封装的意义一：**

在设计类的时候，属性和行为写在一起，表现事物

**语法：** `class 类名 { 访问权限：属性 / 行为};`

类中的属性和行为，我们统称为成员

**属性：** 成员属性、成员变量

**行为：** 成员函数、成员方法

**封装意义二：**

类在设计时，可以把属性和行为放在不同的权限下，加以控制

**访问权限有三种**

1. public 公共权限
2. protected 保护权限
3. private 私有权限

- public：可以被该类中的函数、子类的函数、友元函数访问，也可以由该类的对象访问（类外访问）；
- protected：可以被该类中的函数、子类的函数、友元函数访问，但不可以由该类的对象访问（类外访问）；
- private：可以被该类中的函数、友元函数访问，但不可以由子类的函数、该类的对象访问（类外访问）。

**private 关键字的作用在于更好地隐藏类的内部实现。**

### 4.1.2 struct和class的区别

在C++中struct和class唯一的区别在于默认访问权限不同：

struct 默认权限为公共

class 默认权限为私有

### 4.1.3 成员属性设置为私有

**优点1:** 将所有成员函数设置为私有，可以自己控制读写权限

**优点2:** 对于写权限，我们可以检测数据的有效性

## 4.2 对象初始化和清理

(1) 出厂设置与格式化，清理数据

(2) C++中每个对象也都会有初始设置以及对象销毁后的清理数据的设置。

### 4.2.1 构造函数和析构函数

(1) 4.2.1构造函数和析构函数

(2) 对象的**初始化**和**处理**也是两个非常重要的安全问题

一个对象或者变量没有初始状态，对其后果是未知的

同样的使用完一个对象或变量，没有及时清理，也会造成一定的安全问题。

C++利用了**构造函数**和**析构函数**解决上述问题，这两个函数将会被编译器自动调用，完成对象的初始化和清理工作。

对象的初始化操作和清理工作是编译器强制要我们做的事情，因此**如果我们不提供构造和析构，编译器也会提供**

**编译器提供的构造函数和析构函数都是空实现**

- 构造函数：主要作用在于创建对象的成员属性赋值，构造函数由编译器自动调用，无须手动调用。
- 析构函数：主要作用在于对象**销毁前**系统自动调用，执行一些清理操作。

**构造函数语法:** `类名() {}`

1.构造函数，没有返回值也不写void

2.构造函数与类名相同

3.构造函数可以有参数，因此可以发生重载

4.程序在对象销毁前会自动调用构造，无须手动调用，而且只会调用一次。

**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。

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

//构造函数与析构函数

class Person
```

```

{
public:
    Person() //构造函数只会调用一次，在创建对象时进行调用
    {
        cout << "Person 构造函数调用" << endl;
    }
    ~Person() //析构函数只会调用一次，在销毁对象前进行调用
    {
        cout << "Person 析构函数调用" << endl;
    }
};

//构造和析构都必须有的实现，如果我们自己不提供，编译器会提供一个空实现的构造和析构
void p1()
{
    Person p1; //在栈上的数据，p1()函数执行完毕后，释放这个对象。
}
int main()
{
    Person p;
    //p1();
    system("pause");
    return 0;
}

```

## 4.2.2 构造函数的分类及调用

两种分类方式：

- 按参数分为：**有参构造和无参构造**
- 按类型分为：**普通构造和拷贝构造**

三种调用方式：

- 括号法
- 显示法
- 隐式转换法

注意事项：调用默认构造函数，不要加()

匿名对象 特点：当前行执行结束后，系统会立即回收掉匿名对象

注意：不要利用拷贝构造函数初始化匿名对象。编译器会认为 `Person(p3) == Person p3`;对象声明，重定义。

```

#include<iostream>
using namespace std;
// 1、构造函数分类
// 按照参数分类分为 有参和无参构造 无参又称为默认构造函数
// 按照类型分类分为 普通构造和拷贝构造
class Person
{
public:
    //构造函数
    Person()
    {

```

```

        cout << "Person的无参构造函数" << endl;
    }
    //有参构造函数
    Person(int a)
    {
        age = a;
        cout << "Person的有参构造函数" << endl;
    }
    //拷贝构造函数
    Person(const Person &p) //拷贝辅助所有属性，由于防止修改属性，加以const与引用加以限制。
    {
        age = p.age;
        cout << "Person的拷贝构造函数" << endl;
    }
    ~Person() //析构函数
    {
        cout << "Person的析构函数" << endl;
    }
    int age = 0;
};

void test01()
{
    //调用方法
    //1.括号法
    //Person p1; //默认构造函数调用
    //Person p2(10); //有参构造函数
    //Person p3(p2); //拷贝构造函数
    //cout << "p2的年龄为" << p2.age<<endl;
    //cout << "p3的年龄为" << p3.age << endl; //拷贝函数复制数据
    //注意事项：调用默认构造函数，不要加()
    //Person p4(); //因为编译器会认为这行代码是一个函数的声明，不会认为在创建对象。

    //2.显示法
    //Person p1;
    //Person p2 = Person(100); //调用有参构造
    //Person p3 = Person(p2); //调用拷贝构造

    //Person(10); //匿名对象 特点：当前行执行结束后，系统会立即回收掉匿名对象。
    //cout << "aaaa" << endl;

    //Person(p3); //注意：不要利用拷贝构造函数初始化匿名对象。编译器会认为 Person(p3) ==
    //Person p3;对象声明，重定义。

    //3.隐式转换法
    Person p4 = 10; //相当于 Person p4 = Person(10); 有参构造
    Person p5 = p4; //拷贝构造
}

int main()
{
    test01();
    system("pause");
    return 0;
}

```



### 4.2.3 拷贝构造函数调用规则

C++中拷贝构造函数调用时机通常有三种情况

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

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

//拷贝构造函数调用时机

class Person
{
public:
    Person()
    {
        cout << "Person默认构造函数调用" << endl;
    }
    Person(int age)
    {
        m_Age = age;
        cout << "Person有参构造函数调用" << endl;
    }
    Person(const Person& p)
    {
        m_Age = p.m_Age;
        cout << "Person拷贝构造函数调用" << endl;
    }
    ~Person()
    {
        cout << "Person析构函数调用" << endl;
    }

    int m_Age = 0;
};

//1、 使用一个已经创建完毕的对象来初始化一个新对象
void test01()
{
    Person p1(20);
    Person p2(p1);
    cout << "p2的年龄为: " << p2.m_Age << endl;
}

//2、 值传递的方式给函数参数传值
void dowork02(Person p) //拷贝临时副本
{

}

void test02()
{
    Person p;
    dowork02(p);
}
```

```

}
//3、 以值方式返回局部对象

Person dowork03() //
{
    Person p3;
    cout << (int*)&p3 << endl;
    return p3;
}

void test03()
{
    Person p = dowork03(); // 拷贝临时副本
    cout << (int*)&p << endl;
}

int main()
{
    //test01();
    //test02();
    test03();
    system("pause");
    return 0;
}

```

#### 4.2.4 构造函数调用规则

默认情况下，C++编译器至少给一个类添加3个函数

1. 默认构造函数（无参，函数体为空）
2. 默认析构函数（无参，函数体为空）
3. 默认拷贝构造函数，对属性进行值拷贝

构造函数调用规则如下：

- 如果用户定义有参构造函数，C++不在提供默认无参构造，但是会提供默认拷贝构造
- 如果用户定义拷贝构造函数，C++不会提供其他构造函数。

```

#include<iostream>
using namespace std;
//构造函数的调用规则
//默认情况下，C++编译器至少给一个类添加3个函数
//
//1. 默认构造函数（无参，函数体为空）
//2. 默认析构函数（无参，函数体为空）
//3. 默认拷贝构造函数，对属性进行值拷贝

//如果我们写了有参构造函数，编译器就不再提供默认构造，依然提供拷贝构造
//如果写了拷贝构造函数，编译器就不会提供其他的构造函数
class Person
{
public:
    Person()
    {
        cout << "Person默认构造函数调用" << endl;
    }
}

```

```

Person(int age)
{
    m_age=age;
    cout << "Person有参构造函数调用" << endl;
}
~Person()
{
    cout << "Person析构造函数调用" << endl;
}
/*Person(const Person& p)
{
    m_age = p.m_age;
    cout << "Person拷贝构造函数调用" << endl;
}*/

int m_age = 0;
};

void test01()
{
    Person p;
    p.m_age = 18;
    Person p2(p);
    cout << "p2的年龄为: " << p2.m_age<<endl;
}
void test02()
{
    Person p(28);
    Person p2(p);
    cout << "p2的年龄为: " << p2.m_age << endl;
}
int main()
{
    //test01();
    test02();
    system("pause");
    return 0;
}

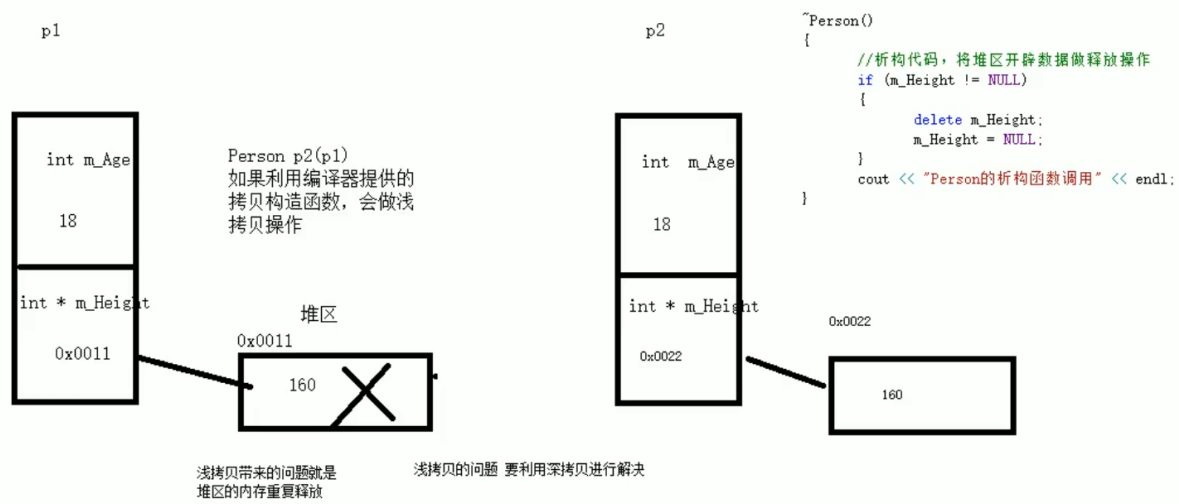
```

## 4.2.5 深拷贝与浅拷贝

深浅拷贝是经典面试问题，也是常见的一个坑。

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



```

#include<iostream>
using namespace std;
//深拷贝和浅拷贝

class Person
{
public:
    Person()
    {
        cout << "Person默认构造函数调用" << endl;
    }
    Person(int age,int height)
    {
        m_age = age;
        m_height = new int(height); //指针接收堆区数据
        cout << "Person有参构造函数调用" << endl;
    }

    //自己实现拷贝构造函数，解决浅拷贝带来的问题
    Person(const Person &p)
    {
        cout << "Person拷贝构造函数调用" << endl;
        m_age = p.m_age;
        //m_height = p.m_height, 编译器默认实现就是这行代码，浅拷贝操作

        //编写深拷贝操作
        m_height = new int(*p.m_height);
    }

    ~Person()
    {
        //析构函数，将堆区开辟数据做释放操作
        if (m_height != NULL)
        {
            delete m_height;
            m_height = NULL;
        } //这段代码在浅拷贝的情况下会导致程序崩溃，因为p2已经释放过一次内存了，p1又再一次释放
        // 浅拷贝带来的问题就是堆区的内存重复释放
        // 浅拷贝的问题要利用深拷贝进行解决。
        cout << "Person析构函数调用" << endl;
    }
    int m_age = 0;
}

```

```

    int* m_height = 0;
};

void test01()
{
    Person p1(18,160);
    cout << "p1的年龄为: " << p1.m_age << " ,p1的身高为: " << *p1.m_height << endl;
    Person p2(p1);
    cout << "p2的年龄为: " << p2.m_age << " ,p2的身高为: " << *p2.m_height << endl;
}

int main()
{
    test01();
    system("pause");
    return 0;
}

```

## 4.2.6 初始化列表

**作用:**

C++提供了初始化列表语法, 用来初始化属性

**语法:**

构造函数(): 属性1(值1), 属性2(值2).....{}

```

#include<iostream>
using namespace std;

//初始化列表
class Person
{
public:

    //传统初始化操作
    /*Person(int a, int b, int c)
    {
        m_A = a;
        m_B = b;
        m_C = c;
    }*/

    //初始化列表初始化属性
    Person(int a,int b,int c) :m_A(a),m_B(b),m_C(c)
    {

    }
    int m_A;
    int m_B;
    int m_C;
};

```

```

void test01()
{
    //Person p(10,20,30);
    Person p;
    cout << "m_A = " << p.m_A << endl;
    cout << "m_B = " << p.m_B << endl;
    cout << "m_C = " << p.m_C << endl;
}

int main()
{
    test01();
    system("pause");
    return 0;
}

```

## 4.2.7 类对象作为类成员

C++类中的成员可以是另一个类的对象，我们称为该成员为对象成员

例如：

```


class A{}
class B
{
    A a;
}

```

B类中有对象A作为成员，A作为对象成员

那么当创建B对象时，A与B的构造与析构顺序是谁先谁后？

当其他类对象作为本类成员，构造时先构造类对象，在构造自身  
析构是就是先析构自身，再析构类对象

 D:\Code\C++\C++核心编程\_002\Debug\C++核心编程

```

Phone有参构造函数调用
Person有参构造函数调用
张三拿着IPhone MAX
Person析构函数调用
Phone析构函数调用
请按任意键继续. . .

```

```

#include<iostream>
using namespace std;

//类对象作为类成员
class Phone
{
public:
    Phone(string pName):m_PName(pName) //有参构造函数
    {
        cout << "Phone有参构造函数调用" << endl;;
        //m_PName = pName;
    }
}

```

```

    }
    ~Phone()
    {
        cout << "Phone析构函数调用" << endl;
    }
    string m_PName;
};

class Person
{
public:
    //初始化列表 Phone m_Phone = pName 隐式转换法
    Person(string name, string pName): m_Name(name), m_Phone(pName)
    {
        cout << "Person有参构造函数调用" << endl;;
    }
    string m_Name; //姓名
    Phone m_Phone; //手机
    ~Person()
    {
        cout << "Person析构函数调用" << endl;
    }
};

//当其他类对象作为本类成员，构造时先构造类对象，在构造自身
//析构是就是先析构自身，再析构类对象

void test01()
{
    //先走Person类的构造还是先走Phone类的构造
    Person p("张三", "iPhone MAX");
    cout << p.m_Name << "拿着" << p.m_Phone.m_PName << endl;
}

int main()
{
    test01();
    system("pause");
    return 0;
}

```

## 4.2.8 静态对象

静态成员就是在成员变量和成员函数前加上关键字static，成为静态成员

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
  - 类内声明，类外初始化

```

#include<iostream>
using namespace std;

```

```

//静态成员变量
class Person
{
public:
    // 1、所有对象都共享同一份数据
    // 2、编译阶段就分配内存
    // 3、类内声明，类外初始化操作
    static int m_A;

private: //静态变量也是有访问权限的
    static int m_B;
};
int Person::m_A = 100;
int Person::m_B = 200;
void test01()
{
    Person p;
    cout << p.m_A << endl;
    Person p2;
    p2.m_A = 200;
    cout << p.m_A << endl; //输出为200，共享的一份内存
}

void test02()
{
    /// 静态成员变量，不属于某个对象上，所有对象都共享同一份数据
    //因此静态变量成员变量有两种访问方式

    // 1、通过对象进行访问
    Person p;
    cout << "通过对象进行访问" << p.m_A << endl;
    // 2、通过类名进行访问
    cout << "通过类名进行访问" << p.m_A << endl;
    //cout << "通过类名进行访问" << p.m_B << endl; 类外访问不到私有静态变量
}
int main()
{
    //test01();
    test02();
    system("pause");
    return 0;
}

```

- 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

```

#include<iostream>
using namespace std;

```

```

//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

```



```

class Person
{
public:
    //静态成员函数
    static void func()
    {
        m_a = 100; //静态成员函数可以访问静态成员变量（共享数据）
        // m_b = 200; //静态成员函数不可以访问非静态成员变量（特定数据，不好区分）
        cout << "static void func调用" << endl;
    }
    static int m_a;
    int m_b;

    //静态成员函数也有访问权限
private:
    static void func2()
    {
        cout << "static void func2调用" << endl;
    }
};

int Person::m_a = 0;
void test01()
{
    //1、通过对象访问
    Person p;
    p.func();
    //2、通过类名访问
    Person::func();
    // Person::func2(); //类外访问不到私有静态成员函数
}

int main()
{
    test01();
    system("pause");
    return 0;
}

```

## 4.3 C++对象模型和this指针

### 4.3.1 成员变量和成员函数分开存储

在C++中，类内的成员变量和成员函数分开存储

只有非静态成员变量才属于类的对象上。

```

#include<iostream>
using namespace std;

// 成员变量和成员函数分开存储
class Person
{

```

```

    int m_A; // 非静态成员变量，属于类的对象上
    static int m_B; // 静态成员变量，不属于类的对象上
    void func() {} // 非静态成员函数，不属于类的对象上
    static void func02() {} // 静态成员函数，不属于类的对象上

};
int Person::m_B = 10;

void test01()
{
    Person p;
    // 空对象占用内存空间为: 1 Bytes
    // 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
    // 每个空对象也应该有一个独一无二的内存地址
    cout << "size of p is " << sizeof(p) << endl;
}

void test02()
{
    Person p;
    cout << "size of p is " << sizeof(p) << endl;
}

int main()
{
    // test01();
    test02();
    system("pause");
    return 0;
}

```

### 4.3.2 this指针

通过4.3.1我们知道在C++中成员变量和成员函数是分开存储的

每一个非静态成员函数只会诞生一份函数实例，也就是说多个同类型的对象会共用一块代码

那么问题是，这一块代码是如何区分哪个对象自己的？

C++通过提供特殊的对象指针，this指针，解决上述问题，**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函数内的一种指针

this指针不需要定义，直接使用即可

this指针的用途

- 当形参和成员变量同名时，可用this指针来区分
- 在类的非静态成员函数中返回对象本身，可使用 return \*this

```

#include<iostream>
using namespace std;

class Person
{
public:
    Person(int age) // 有参构造函数

```

```

{
    // age = age ; //这里形参与成员变量名称相同，编译器不能区分
    this->age = age; //用this指针指向被调用的成员函数的所属的对象，用以区分
}
Person& PersonAddAge(Person& p) //值返回一个新的对象，引用返回的是原来的对象
{
    this->age += p.age;

    //this指向p2的指针，而*this指向的就是p2这对象本体
    return *this; //深拷贝与浅拷贝
}
int age;
};

//1、解决名称冲突
void test01()
{
    Person p1(18);
    cout << "p1的年龄为" << p1.age << endl;
}

//2、返回对象本身用*this
void test02()
{
    Person p1(10);
    Person p2(10);
    p2.PersonAddAge(p1).PersonAddAge(p1).PersonAddAge(p1); //链式编程思想
    cout << "p2的年龄为" << p2.age << endl;
}

int main()
{
    //test01();
    test02();
    system("pause");
    return 0;
}

```

### 4.3.3 空指针访问成员函数

C++中空指针也可以调用成员函数，但是也要注意有没有用到this指针

如果用到this指针，需要加以判断保证代码的健壮性

```

#include<iostream>
using namespace std;

//空指针调用成员函数

class Person
{
public:
    void showClassName()
    {

```

```

        cout << "this is Person class" << endl;
    }
    void showPersonAge()
    {
        if (this == NULL) //提高健壮性, 防止传入空指针
        {
            return;
        }
        //报错原因是因为传入的指针为NULL
        cout << "age=" << this->m_Age << endl;
    }
    int m_Age;
};

void test01()
{
    Person* p = NULL;
    p->showClassName();
    p->showPersonAge(); //空指针读取权限异常
}

int main()
{
    test01();
    system("pause");
    return 0;
}

```

#### 4.3.4 const修饰成员函数

常函数:

- 成员函数后加const后我们称为这个函数为**常函数**
- 常函数内不可以修改成员属性
- 成员属性声明时加关键词mutable, 在常函数中依然可以修改

//this指针的本质 是指针常量 指针的指向是不可以修改的  
 // **const** Person \* const this;  
 void showPerson() **const**  
 {  
this->m\_A = 100;  
 //this = NULL; //this指针不可以修改指针的指向的  
 }

常对象:

- 声明对象前加const称该对象为常对象
- 常对象只能调用常函数

```

#include<iostream>
using namespace std;

//常函数

```

```

class Person
{
public:
    //this指针的本质是指针常量，指针的指向是不可以修改的(即地址不可以修改，但是地址内 存的值可以修改)
    //Person * const this; //this指针的本质
    //常函数的实质 const Person * const this ,此时指针指向的值也不允许修改了
    // 在成员函数后面加const，修饰的是this指向，让指针指向的值也不可以修改
    void showPerson() const
    {
        this->m_B = 100;
        //this->m_A = 100;
        //this = NULL; //this指针不可以修改指针的指向的
        cout << "调用常函数" << endl;
    }
    void func()
    {
        cout << "调用普通函数" << endl;
    }
    int m_A;
    mutable int m_B; //特殊变量，即使在常函数中，也可以修改这个值，加关键字mutable
};

//常对象
void test01()
{
    Person p;
    p.showPerson(); //普通对象可以调用常函数
}

void test02()
{
    const Person p; //在对象前加const，变为常对象
    // p.m_A = 100; //不可修改
    p.m_B = 100; //m_B是特殊值，在常对象下也可以修改
    p.showPerson();
    //p.func(); //常对象只能调用常函数，不可以调用普通成员函数，因为普通成员函数可以修改属性
}

int main()
{
    test01();
    test02();
    system("pause");
    return 0;
}

```

## 4.4 友元

生活中你的家有客厅（Public），有你的卧室（Private）

客厅所有来的客人都可以进去，但是你的卧室是私有的，也就是说只有你能进去

但是呢，你也可以允许你的好朋友进去

在程序里，有些私有属性也想让类外特殊的一些函数或者类进行访问，就需要用到友元的技术

友元的目的就是让一个函数或者类访问另一个类中私有成员

友元的关键字为 **friend**

友元的三种实现：

1. 全局函数做友元
2. 类做友元
3. 成员函数做友元

#### 4.4.1 全局函数做友元

全局函数做友元的情况

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

class Buliding
{
    //goodFriend全局函数是Buliding好朋友，可以访问Buliding中私有成员
    friend void goodFriend(Buliding* buliding);
public:
    string m_SittingRoom; //公用
    Buliding() //构造函数
    {
        m_SittingRoom = "客厅";
        m_BedRoom = "卧室";
    }
private:
    string m_BedRoom; //私有
};

void goodFriend(Buliding* buliding)
{
    cout << "好朋友全局函数正在访问：" << buliding->m_SittingRoom << endl;
    //cout << "好朋友全局函数正在访问：" << buliding->m_BedRoom << endl; //正常情况下，
    //全局函数不能访问私有变量
    cout << "好朋友全局函数正在访问：" << buliding->m_BedRoom << endl; //在类中进行友元
    //声明后，全局函数可以访问私有变量
}

void test01()
{
    Buliding buliding;
    goodFriend(&buliding);
}

int main()
{
    test01();
    system("pause");
    return 0;
}
```

## 4.4.2 类做友元

类做友元的情况

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

//类做友元
class Buliding;
class GoodFriend
{
public:
    GoodFriend();//构造函数
    void visit(); //参观函数 访问Buliding中的属性
    Buliding* buliding;
};

class Buliding
{
    //GoodFriend类作为Buliding类的友元，可以访问奔雷中私有成员
    friend class GoodFriend;
public:
    Buliding(); //构造函数
    string m_SittingRoom;
private:
    string m_BedRoom;
};

//类外写成员函数
Buliding::Buliding()
{
    m_SittingRoom = "客厅";
    m_BedRoom = "卧室";
}
GoodFriend::GoodFriend()
{
    buliding = new Buliding; //堆区创建一个Buliding对象
}
void GoodFriend::visit()
{
    cout << "好朋友类正在访问: " << buliding->m_SittingRoom << endl;
    //cout << "好朋友类正在访问: " << buliding->m_BedRoom << endl; //正常情况下，其他类
    不能访问私有变量
    cout << "好朋友类正在访问: " << buliding->m_BedRoom << endl; //在类中进行友元声明
    后，全局函数可以访问私有变量
    delete buliding; //回收堆区内存
}
void test01()
{
    GoodFriend gg;
    gg.visit();
}

int main()
```

```

{
    test01();
    system("pause");
    return 0;
}

```

#### 4.3.4 成员函数做友元

成员函数做友元的情况，编程心得，注意声明的顺序，如果使用了相关内容却未声明，会报错。

```

#include<iostream>
using namespace std;

// 成员函数做友元

class Buliding;
class GoodFriend
{
public:
    GoodFriend(); //构造函数
    void visit(); //让visit函数可以访问Buliding中私有成员
    void visit2(); //让visit2函数不可以访问Buliding中私有成员

    Buliding * buliding;
};

class Buliding
{
    //告诉编译器 GoodFriend下的visit成员函数作为本类的友元，可以访问私有成员
    friend void GoodFriend::visit();
public:
    Buliding();
    string m_SittingRoom;
private:
    string m_BedRoom;
};

// 类外实现成员函数
Buliding::Buliding()
{
    m_SittingRoom = "客厅";
    m_BedRoom = "卧室";
}
GoodFriend::GoodFriend()
{
    buliding = new Buliding; //在堆区创建一个buliding对象
}
void GoodFriend::visit()
{
    cout << "visit函数正在访问 :" << buliding->m_SittingRoom << endl;
    cout << "visit函数正在访问 :" << buliding->m_BedRoom << endl;
}
void GoodFriend::visit2()
{

```



```

        cout << "visit2函数正在访问 :" << buliding->m_SittingRoom << endl;
        //cout << "visit2函数正在访问 :" << buliding->m_BedRoom << endl;
    }

    void test01()
    {
        GoodFriend gg;
        gg.visit();
        gg.visit2();
    }

    int main()
    {
        test01();
        system("pause");
        return 0;
    }

```

## 4.5 运算符重载

运算符重载概念：对已有的运算符重新进行定义，赋予其另一种功能，以适应不同的数据类型  
用于类与对象的操作

### 4.5.1 加号运算符重载

作用：实现两个自定义数据类型相加的运算

对于内置的数据类型，编译器知道如何进行运算

对于内置数据类型，编译器知道如何进行运算

```

int a = 10;
int b = 10;
int c = a + b;

```

```

class Person
{
public:
    int m_A;
    int m_B;
}

```

```

Person p1;
p1.m_A = 10;
p1.m_B = 10;

Person p2;
p2.m_A = 10;
p2.m_B = 10;

Person p3 = p1 + p2;

```

通过自己写成员函数，实现两个对象相加属性后返回新的对象

```

Person PersonAddPerson(Person &p)
{
    Person temp;
    temp.m_A = this->m_A + p.m_A;
    temp.m_B = this->m_B + p.m_B;
    return temp;
}

```

编译器给起了一个通用名称

通过成员函数重载+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;
}

```

```

Person p3 = p1.operator+(p2);
简化为
Person p3 = p1 + p2;

```

通过全局函数重载+

```

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;
}

```

```

Person p3 = operator+ (p1, p2)
简化为
Person p3 = p1 + p2;

```

```

#include<iostream>
using namespace std;

```

//加号运算符重载

```

class Person
{
public:
    int m_A;

```

```

    int m_B;
    //1、成员函数重载+号
    /* 本质调用 Person p3 = p1.operator+(p2);
    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+号
//本质调用 Person p3 = operator+(p1,p2)
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;
}
//函数进行重载
Person operator+(Person& p1, int num)
{
    Person temp;
    temp.m_A = p1.m_A + num;
    temp.m_B = p1.m_B + num;
    return temp;
}
void test01()
{
    Person p1;
    p1.m_A = 11;
    p1.m_B = 12;
    Person p2;
    p2.m_A = 21;
    p2.m_B = 22;
    Person p3 = p1 + p2;
    // 运算符重载也可以发生函数重载
    Person p4 = p1 + 100;
    cout << "p3的m_A = " << p3.m_A << endl;
    cout << "p3的m_B = " << p3.m_B << endl;
    cout << "p4的m_A = " << p4.m_A << endl;
    cout << "p4的m_B = " << p4.m_B << endl;
}
int main()
{
    test01();
    system("pause");
    return 0;
}

```

总结1：对于内置的数据类型的表达式的运算符是不可能改变的

总结2：不要滥用运算符重载

## 4.5.2 左移运算符重载

作用：可以输出自定义数据类型

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

//左移运算符重载

class Person
{
    friend ostream& operator<<(ostream& cout, Person p); //友元
public:
    Person(int a, int b) // 有参构造函数
    {
        m_A = a;
        m_B = b;
    }
private:
    int m_A;
    int m_B;
    //成员函数重载 左移运算符
    // p.operator<<(cout) ----> 简化版本 p<< cout
    // void operator<<(cout) {} 不会利用成员运算符重载<<运算符，因为无法实现cout在左侧
};

//只能利用全局函数重载左移运算符
// 本质 operator<<(cout,p) 简化cout<<p;
//注意ostream对象只能有一个
ostream & operator<<(ostream &cout, Person p)
{
    cout << "m_A = " << p.m_A << " ,m_B = " << p.m_B ;
    return cout;
}

void test01()
{
    Person p(10,11);
    cout << p << endl << "hello world!" << endl;; //链式编程
}

int main()
{
    test01();
    system("pause");
    return 0;
}
```

### 4.5.3 递增运算符重载

作用：通过重载递增运算符，实现自己的整型数据

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

// 重载递增运算符
//自定义整型

class MyInteger
{
    friend ostream& operator<<(ostream& cout, MyInteger myint); //友元
public:
    MyInteger()
    {
        m_Num = 0;
    }
    //重载前置++运算符
    MyInteger & operator++() //返回引用是为了一直对一个数据进行递增操作
    {
        m_Num++;
        return *this;
    }
    //重载后置++运算符
    // void operator++(int) int 代表占位参数，可以用于区分前置和后置递增
    MyInteger operator++(int) //注意返回的是局部变量，因此需要返回值（局部对象），而不是
    返回引用（非法操作）
    {
        //先 记录当时的结果
        MyInteger temp = *this;
        //后 递增
        m_Num++;
        //最后将记录结果做返回
        return temp;
    }
private:
    int m_Num;
};

//重载左移运算符
ostream& operator<<(ostream& cout, MyInteger myint) //注意myint引用与值的形参
{
    cout << myint.m_Num;
    return cout;
}

void test01()
{
    MyInteger myint;
    cout << ++(++myint) << endl;
    cout << myint << endl;
}

void test02()
{
    MyInteger myint;
```

```

        cout << myint++ << endl;
        cout << myint << endl;
    }
    int main()
    {
        test01();
        test02();
        system("pause");
        return 0;
    }

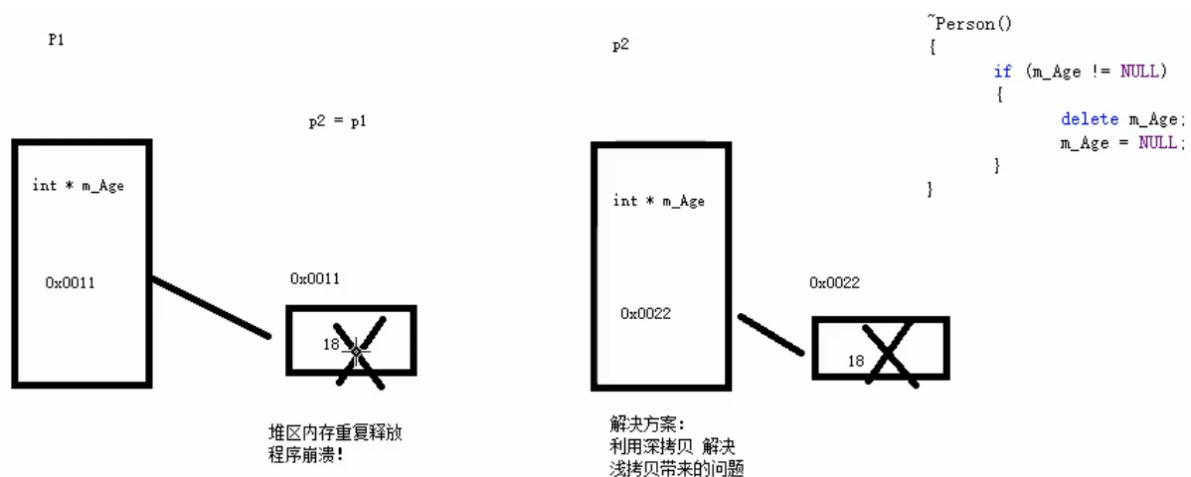
```

#### 4.5.4 赋值运算符重载

C++编译器至少会给一个类添加4个函数

1. 默认构造函数（无参，函数体为空）
2. 默认析构函数（无参，函数体为空）
3. 默认拷贝构造函数，对属性进行**值拷贝**
4. 赋值运算符 operator=，对属性进行值拷贝

如果类中有属性指向堆区，做赋值操作时也会出现深浅拷贝问题



```

#include<iostream>
using namespace std;

//赋值运算符重载

class Person
{
public:
    Person(int age) //有参构造函数
    {
        m_Age = new int(age); //堆区开辟一个在指针变量
        //程序员手动创建，手动释放
    }
    int* m_Age;
    ~Person()
    {
        if (m_Age != NULL)
        {
            delete m_Age; //析构函数 ---->程序崩溃---->原因：堆区内存重复释放
            m_Age = NULL;
        }
    }
}

```

```

    }
}

//重载 赋值运算符
Person & operator=(Person& p) //返回引用，若是返回值，则是调用拷贝构造函数创建一个新的副本
{
    //编译器是提供浅拷贝
    //m_Age = p.m_Age;

    //应先判断是否有属性在堆区，如果有先释放干净，然后再深拷贝
    if (m_Age != NULL)
    {
        delete m_Age;
        m_Age = NULL;
    }

    //深拷贝
    m_Age = new int(*p.m_Age); //在堆区申请一块新的内存

    return *this; //返回对象本身
}
};

void test01()
{
    Person p1(18);
    Person p2(20);
    Person p3(30);
    p3 = p2 = p1; //默认赋值操作是浅拷贝
    cout << "p1的年龄为: " << *p1.m_Age << endl;
    cout << "p2的年龄为: " << *p2.m_Age << endl;
    cout << "p3的年龄为: " << *p3.m_Age << endl;
}

int main()
{
    test01();
    int a = 10;
    int b = 20;
    int c = 30;
    c = b = a;
    cout << "a = " << a << ",b = " << b << ",c = " << c << endl;
    system("pause");
    return 0;
}

```

### 4.5.5 关系运算符重载

**作用：** 重载关系运算符，可以让两个自定义对象进行比较操作

```

#include<iostream>
using namespace std;

//关系运算符重载
class Person

```

```

{
public:
    Person(string name, int age) //有参构造函数
    {
        m_name = name;
        m_age = age;
    }
    //重载 == 号
    bool operator==(Person &p)
    {
        if (this->m_age == p.m_age && this->m_name == p.m_name)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
    string m_name;
    int m_age;
};

void test01()
{
    Person p1("tom", 18);
    Person p2("Jerry", 18);
    if (p1 == p2)
    {
        cout << "对象p1与p2是相等的" << endl;
    }
    else
    {
        cout << "对象p1和p2是不相等的" << endl;
    }
}

int main()
{
    test01();
    system("pause");
    return 0;
}

```

#### 4.5.6 函数调用运算符重载

- 函数调用运算符()也可以重载
- 由于重载后使用方式非常像函数的调用，因此称为仿函数
- 仿函数没有固定写法，非常灵活

```

#include<iostream>
using namespace std;

//函数调用运算符重载

```

```

//打印输出类
class MyPrint
{
public:
    //重载函数调用运算符
    void operator()(string test)
    {
        cout << test << endl;
    }
};

void MyPrint02(string test)
{
    cout << test << endl;
}

void test01()
{
    MyPrint myprint;
    myprint("hello,world!"); //由于使用起来非常类似于函数调用，因此称为仿函数
    MyPrint02("hello,master!");
}

//仿函数非常灵活，没有固定的写法
//例子：加法类

class MyAdd
{
public:
    int operator()(int num1, int num2)
    {
        return num1 + num2;
    }
};

void test02()
{
    MyAdd myadd;
    cout << "结果为" << myadd(10, 20) << endl;

    //匿名函数对象，当前操作执行完后，立即被释放
    cout << "匿名函数对象结果为" << MyAdd()(100, 200) << endl;
}

int main()
{
    test01();
    test02();
    system("pause");
    return 0;
}

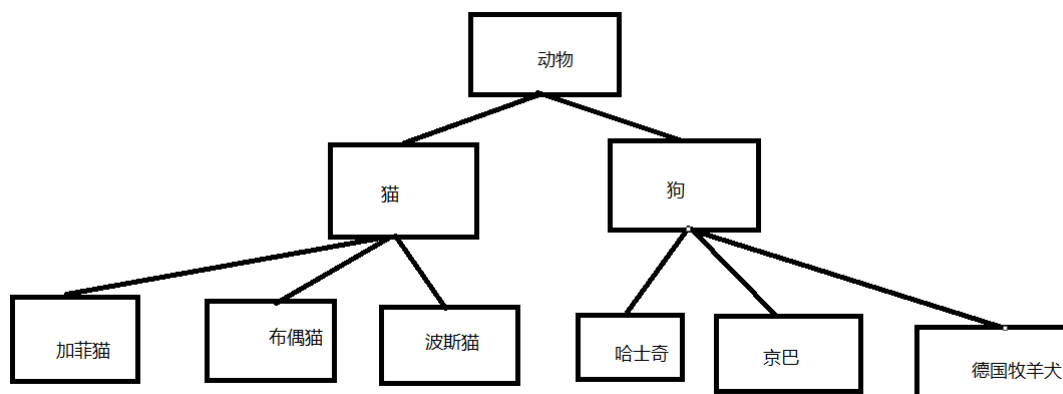
```



## 4.6 继承

### 继承是面向对象三大特性之一

有些类与类之间存在特殊的关系，例如下图



我们发现，定义这些类时，下级别的成员除了拥有上一级的共性，还有自己的特性

这个时候，我们就可以考虑利用继承的技术，减少重复代码。

### 4.6.1 继承的基本语法

例如我们看到很多网站中，都有公共的头部，公共的底部，甚至公共的左侧列表，只有中心内容不同

接下来我们分别利用普通写法和继承的写法来实现网页中的内容，看一下继承存在的意义及好处

#### 继承的好处：减少重复代码

**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

#### 派生类的成员，包含两大部分：

一类是基类继承过来的，一类是自己增加的成员。

从基类继承过来的表现其共性，而新增的成员体现了其个性。

```
#include<iostream>
using namespace std;
//普通实现页面 ----> 辅助粘贴

// 继承实现
class BasePage
{
public:
    void header()
    {
        cout << "首页、公开课、登录、注册...(公共头部)" << endl;
    }
    void footer()
    {
        cout << "帮助中心、交流合作、站内地图...(公共底部)" << endl;
    }
    void left()
    {
```

```

        cout << "Java、Python、C++、...(公共类列表)" << endl;
    }
};
//继承的好处：减少重复代码
//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
//Java页面
class Java:public BasePage
{
public:
    void content()
    {
        cout << "Java学科视频" << endl;
    }
};

class Python:public BasePage
{
public:
    void content()
    {
        cout << "Python学科视频" << endl;
    }
};

void test01()
{
    cout << "Java下载视频页面如下：" << endl;
    Java ja;
    ja.header();
    ja.footer();
    ja.content();
    cout << "-----" << endl;
    cout << "Python下载视频页面如下：" << endl;
    Python py;
    py.header();
    py.footer();
    py.content();
}

int main()
{
    test01();
    system("pause");
    return 0;
}

```

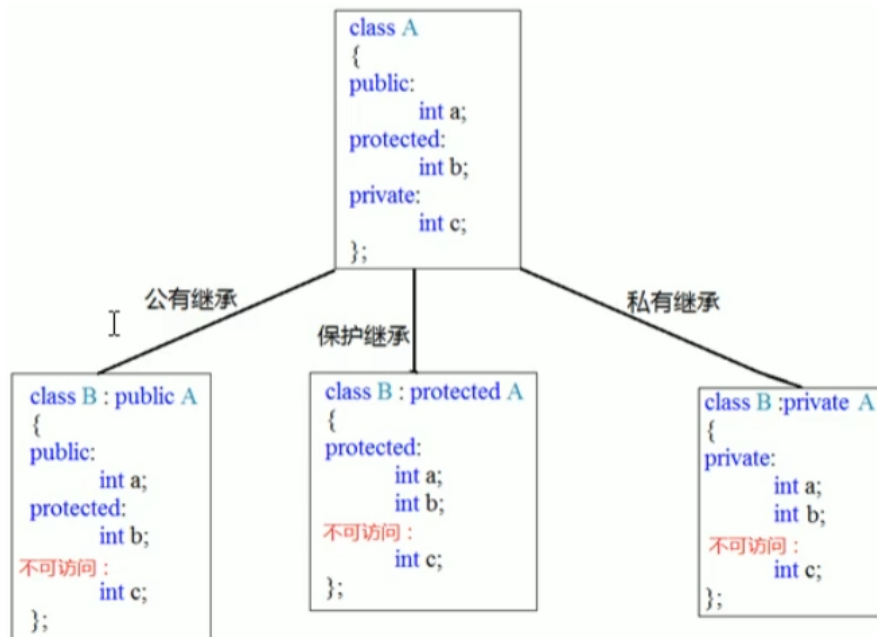
## 4.6.2 继承方式

继承的语法：class 子类 : 继承方式 父类

**继承方式一共有三种：**

- 公共继承
- 保护继承

- 私有继承



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

//继承方式
class Father //定义父类
{
    //friend class Son1;
public:
    int m_a;
protected:
    int m_b;
private:
    int m_c;
};

//公共继承
class Son1:public Father
{
public:
    void func()
    {
        m_a = 10; //父类中的公共权限成员 到子类中依然是公共权限
        m_b = 10; //父类中的保护权限成员 到子类中依然是保护权限
        //m_c = 10; //父类的私有权限 子类访问不到，但是可通过声明友元来进行访问
    }
};

//保护继承
class Son2 :protected Father
{
    void func()
    {
        m_a = 100; //父类中的公共成员，到子类中变为保护权限
        m_b = 100; //父类中保护成员，到子类变为保护权限
        //m_c = 100; //父类中的私有成员，子类访问不到
    }
};
```

```

//私有继承
class Son3 : private Father
{
    void func()
    {
        m_a = 100; //父类公共成员 到子类中变为 私有成员
        m_b = 100; //父类保护成员 到子类变为 私有成员
        // m_c = 100; //父类中私有权限成员，子类访问不到
    }
};

class GrandSon3 :public Son3
{
    void func()
    {
        // m_a = 100; //到了Son3中,m_a变为私有，即使是子类，也访问不到
        // m_b = 100; //到了Son3中,m_b变为私有，即使是子类，也访问不到
    }
};

void test01()
{
    Son1 s1;
    s1.m_a = 100; //公共权限，类外可访问
    //s1.m_b = 100; //保护权限，类外不可访问，子类可访问
}

void test02()
{
    Son2 s2;
    // s2.m_a = 100; //Son2中m_a变为保护权限，因此类外访问不到
    // s2.m_b = 100; //Son2中m_b变为保护权限，因此类外不可访问
}

void test03()
{
    Son3 s3;
    //s3.m_a = 100; //到Son3中，变为 私有成员 ，类外访问不到
    //s3.m_b = 100; //到Son3中，变为 私有成员 ，类外访问不到
}

int main()
{
    system("pause");
    return 0;
}

```

### 4.6.3 继承中的对象模型

**问题：** 从父类继承过来的成员，哪些属于子类对象中？

```
D:\Code\C++\继承>cl /dl reportSingleClassLayoutSon "03继承中的对象模型.cpp"
用于 x86 的 Microsoft (R) C/C++ 优化编译器 19.29.30133 版
版权所有(C) Microsoft Corporation。保留所有权利。
```

03继承中的对象模型.cpp

```
class Son          size(16):
+---
0      +--- (base class Base)
0      |   m_a
4      |   m_b
8      |   m_c
      +---
12     |   m_d
      +---
```

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

//继承中的对象模型
class Base
{
public:
    int m_a;
private:
    int m_b;
public:
    int m_c;
};

class Son :public Base
{
public:
    int m_d;
};

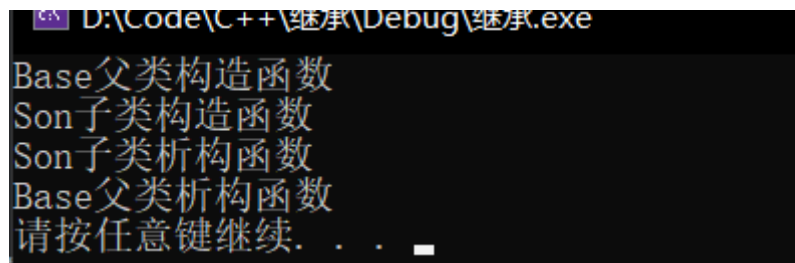
// 利用开发者人员命令提示工具查看对象模型
void test01()
{
    //在父类中所有非静态成员都会被子类继承下去
    //在父类中私有成员属性 是被编译器给隐藏了，因此是访问不到，但是确实被继承下去了
    cout << "size of Son = " << sizeof(Son) << endl; //结果为16，4个整数
}

int main()
{
    test01();
    system("pause");
    return 0;
}
```

#### 4.6.4继承中构造和析构顺序

子类继承父类后，当创建子类对象，也会调用父类的构造函数

问题：父类和子类的构造和析构顺序是谁先谁后？



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

//继承中的构造和析构顺序
class Base
{
public:
    Base()
    {
        cout << "Base父类构造函数" << endl;
    }
    ~Base()
    {
        cout << "Base父类析构函数" << endl;
    }
};

class Son :public Base
{
public:
    Son()
    {
        cout << "Son子类构造函数" << endl;
    }
    ~Son()
    {
        cout << "Son子类析构函数" << endl;
    }
};

void test01()
{
    Son s1;
}

int main()
{
    test01();
    system("pause");
    return 0;
}
```

#### 4.6.5 继承同名成员处理方式

问题：当子类与父类出现同名的成员，如何通过子类对象，访问到子类或者父类中同名的数据？

- 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

总结：

1. 子类对象可以直接访问到子类中同名成员
2. 子类对象加作用域可以访问到父类同名成员
3. 当子类与父类拥有同名的成员函数，子类会隐藏父类中的成员函数，加作用域可以访问到父类中同名函数。

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

//继承中的同名成员处理
class Base
{
public:
    Base()
    {
        m_a = 100;
    }
    void func()
    {
        cout << "Base下的func函数调用" << endl;
    }
    void func(int a)
    {
        cout << "Base下的func(int a)函数调用" << endl;
    }
    int m_a;
};

class Son :public Base
{
public:
    Son()
    {
        m_a = 200;
    }
    void func()
    {
        cout << "Son下的func函数调用" << endl;
    }
    int m_a;
};

class GrandSon :public Son
{
    //默认声明为:private
public:
    GrandSon()
    {
        m_a = 300;
    }
    void func()
    {
        cout << "GrandSon下的func函数调用" << endl;
    }
    int m_a;
};
```

//同名成员属性处理方式

```

void test01()
{
    Son s;
    cout << "Son的m_a = " << s.m_a << endl;
    //如果通过子类对象访问父类的同名成员，需要加作用域
    cout << "Base的m_a = " << s.Base::m_a << endl;
}
//同名成员函数处理方式
void test02()
{
    Son s;
    s.func(); //访问子类的同名函数
    s.Base::func(); //访问父类的同名函数

    //如果子类中出现和父类同名的成员函数，子类的同名函数会隐藏掉父类中所有的同名成员函数
    //如果想访问父类中被隐藏的同名函数，需要添加作用域
    s.Base::func(100);
}
void test03()
{
    GrandSon gs;
    gs.func();
    gs.Son::func();
    gs.Son::Base::func(); //孙子类-->子类-->父类
}
int main()
{
    test01();
    test02();
    test03();
    system("pause");
    return 0;
}

```

#### 4.6.6 继承同名静态成员处理方式

问题：继承同名的静态成员在子类对象上如何进行访问？

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，直接访问即可
- 访问父类同名成员，需要加作用域

```

#include<iostream>
using namespace std;

//继承中的同名静态成员处理方式
class Base
{
public:
    static int m_a;
    static void func()
    {
        cout << "Base - static void func()" << endl;
    }
}

```



```

static void func(int a)
{
    cout << "Base - static void func(int a)" << endl;
}

};

int Base::m_a = 100;

class Son :public Base
{
public:
    static int m_a;
    static void func()
    {
        cout << "Son - static void func()" << endl;
    }
};

int Son::m_a = 200;
//同名静态成员属性
void test01()
{
    //1、通过对象访问
    cout << "通过对象访问: " << endl;
    Son s;
    cout << "Son下的m_a = " << s.m_a << endl;
    cout << "Base下的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;
}

//同名静态成员函数
void test02()
{
    // 1、通过对象访问
    cout << "通过对象访问: " << endl;
    Son s;
    s.func();
    s.Base::func();

    //2、通过类名访问
    cout << "通过类名访问: " << endl;
    Son::func();
    Son::Base::func();
    //子类出现和父类同名静态成员函数，也会隐藏父类中所有同名成员函数
    //如果想访问父类中被隐藏同名成员，需要加作用域
    Son::Base::func(100);
}

int main()
{
    test01();
    cout << endl;
    test02();
    system("pause");
    return 0;
}

```

```
}
```

### 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...`

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

总结: 多继承中如果父类中出现了同名情况, 子类使用时需要加作用域

**C++实际开发中不建议用多继承**

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

//多继承语法

class Base1
{
public:
    Base1()
    {
        m_a = 100;
    }
    int m_a;
};

class Base2
{
public:
    Base2()
    {
        m_a = 200;
    }
    int m_a;
};

class Son :public Base1, public Base2
{
public:
    Son()
    {
        m_c = 300;
        m_d = 400;
    }
    int m_c;
    int m_d;
};

void test01()
{
    Son s;
    cout << "sizeof Son is " << sizeof(s) << endl; //结果为16
    cout << "Base1下的m_a = " << s.Base1::m_a << endl;
```

```

        cout << "Base2下的m_a = " << s.Base2::m_a << endl;
    }
    int main()
    {
        test01();
        system("pause");
        return 0;
    }

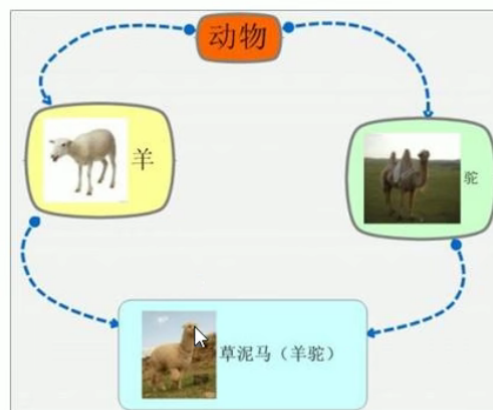
```

## 4.6.8 菱形继承

菱形继承概念：

- 两个派生类继承同一个基类
- 又有某个类同时继承两个派生类
- 这种继承称为菱形继承，或者钻石继承

典型的菱形继承案例：



菱形继承问题：

1. 羊继承了动物的数据，驼同样继承了动物的数据，当羊驼使用数据时，就会产生二义性。
2. 草泥马继承自动物的数据继承了两份，其实我们应该清楚，这份数据我们应该只需要一份就行了。

```

D:\Code\C++\继承>cl /dl reportSingleClassLayoutAlpaca "08菱形继承.cpp"
用于 x86 的 Microsoft (R) C/C++ 优化编译器 19.29.30133 版
版权所有 (C) Microsoft Corporation。保留所有权利。

```

08菱形继承.cpp

```

class Alpaca    size(8):
+---
0      +--- (base class Camel)
0      | +--- (base class Animal)
0      | | m_a
      | +---
+---
4      +--- (base class Sheep)
4      | +--- (base class Animal)
4      | | m_a
      | +---
+---
+---

```

利用虚继承 解决菱形继承的问题

继承之前加上关键字virtual 变为虚继承

此时基类称为虚基类

用于 x86 的 Microsoft (R) C/C++ 优化编译器 19.29.30133 版  
版权所有(C) Microsoft Corporation。保留所有权利。

08菱形继承.cpp

```
class Alpaca    size(12):
+----
0      +---- (base class Camel)
0      | {vbptr}
      +----
4      +---- (base class Sheep)
4      | {vbptr}
      +----
+----
      +---- (virtual base Animal)
8      | m_a
      +----

Alpaca::$vbtable@Camel@:
0      0
1      8 (Alpaca@Camel+0)Animal)

Alpaca::$vbtable@Sheep@:
0      0
1      4 (Alpaca@Sheep+0)Animal)
vbi:    class offset o.vbptr  o.vbte fVtorDisp
      Animal      8      0      4 0
```

vbptr (virtual base pointer) (指向)---> vbtable(virtual base table)

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

//菱形继承

//动物类
class Animal
{
public:
    int m_a;
};

//利用虚继承 解决菱形继承的问题
//继承之前加上关键字virtual 变为虚继承
//此时Animal类称为虚基类
//此时m_a只有一份
class Sheep :virtual public Animal{}; //羊类
class Camel :virtual public Animal{}; //驼类
class Alpaca :public Camel, public Sheep {}; //羊驼类

void test01()
{
    Alpaca a1;
    a1.Sheep::m_a = 10;
    a1.Camel::m_a = 18;
    // 当菱形继承, 两个父类拥有相同数据, 需要加以作用域区分
    cout << "a1.Sheep::m_a = " << a1.Sheep::m_a << endl;
    cout << "a1.Camel::m_a = " << a1.Camel::m_a << endl;
    cout << "s1.m_a = " << a1.m_a << endl;
    //当这份数据我们知道只有一份就可以, 菱形继承导致数据有两份, 资源浪费
```

```

}
int main()
{
    test01();
    system("pause");
    return 0;
}

```

**总结:**

- 菱形继承带来的主要问题是子类继承两份相同的数据，导致资源浪费以及毫无意义
- 利用虚继承可以解决菱形继承问题

## 4.7 多态

### 4.7.1 多态的基本概念

**多态是C++面向对象三大特性之一**

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：派生类和虚函数实现运行时多态

静态多态和动态多态的区别：

- 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

```

#include<iostream>
using namespace std;

// 多态
class Animal
{
public:
    // 虚函数，这样就可以实现地址晚绑定了
    virtual void speak()
    {
        cout << "动物在说话" << endl;
    }
};

class Cat :public Animal
{
public:
    // 重写 函数返回值类型 函数名 参数列表 完全相同
    void speak()
    {
        cout << "小猫在说话" << endl;
    }
};

class Dog :public Animal
{
public:

```

```

void speak()
{
    cout << "小狗在说话" << endl;
}

//执行说话的函数
//地址早绑定，在编译阶段就确定了函数地址
//如果想执行让猫说话，那么这个函数就不能提前绑定，需要在执行阶段进行绑定，地址晚绑定

//动态多态满足条件
//1、有继承关系
//2、子类复写父类的虚函数

// 动态多态使用
// 父类的指针或者引用
，指向子类对象
void doSpeak(Animal& animal) //Animal & animal = cat;
{
    animal.speak();
}

void test01()
{
    Cat cat;
    doSpeak(cat);
    Dog doge;
    doSpeak(doge);
}

int main()
{
    test01();
    system("pause");
    return 0;
}

```

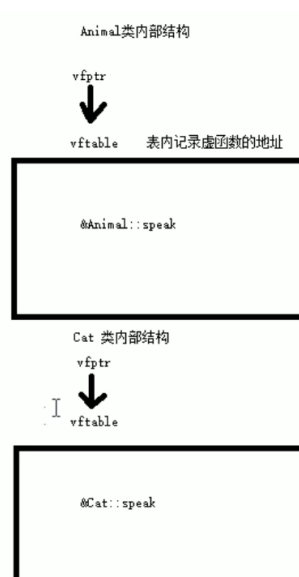
```

class Animal
{
public:
    //虚函数
    virtual void speak()
    {
        cout << "动物在说话" << endl;
    }
};

//猫类
class Cat :public Animal
{
public:
    //重写 函数返回值类型 函数名 参数列表 完全相同
    virtual void speak()
    {
        cout << "小猫在说话" << endl;
    }
};

当子类重写父类的虚函数
    子类中的虚函数表 内部 会替换成 子类的虚函数地址

```



vfptr - 虚函数（表）指针

v - virtual

f - function

ptr - pointer

vftable - 虚函数表

v - virtual

f - function

table - table

当父类的指针或者引用指向子类对象时候，发生多态

```
Animal & animal = cat;
animal.speak();
```

当子类重写父类的虚函数，子类中的虚函数表内部会替换成子类的虚函数地址，没有重写时就是继承虚函数。

当父类的指针或者引用指向子类对象时，发生多态。

### 4.7.2 纯虚函数和抽象类

在多态中，通常父类中的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

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

// 纯虚函数和抽象类

class Base
{
public:
    //纯虚函数
    //只要有一个纯虚函数，这个类就称为抽象类
    //抽象类特点：
    //1、无法实例化对象
    //2、抽象类的子类，必须要重写父类中的纯虚函数，否则也属于抽象类
    virtual void func() = 0;
};

class Son :public Base
{
public:
    virtual void func()
    {
        cout << "func函数调用" << endl;
    };
};

void test01()
{
    //Base b;    //
    //new Base; // 不管在栈区还是对辖区，抽象类都无法实例化对象
    Son s;
    //new Son; // 子类必须重写父类中的纯虚函数，否则无法实例化对象
    Base* base = new Son;
    base->func(); //当父类的指针或者引用指向子类对象时，发生多态
}

int main()
{
    test01();
    system("pause");
    return 0;
}
```

```
}
```

### 4.7.3 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

虚析构和纯虚析构的区别：

- 如果是纯虚析构，该类属于抽象类，无法实例化对象

虚析构函数语法：

```
virtual ~类名() {}
```

纯虚析构语法：

```
virtual ~类名() = 0;
```

```
类名::~~类名() {}
```

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

//虚析构和纯虚析构

class Animal
{
public:
    Animal()
    {
        cout << "Animal构造函数调用" << endl;
    }
    //利用虚析构可以解决父类指针释放子类对象时不干净的问题
    //virtual ~Animal()
    //{
    //    cout << "Animal虚析构函数调用" << endl;
    //}
    //纯虚析构函数，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 ~Animal() = 0;
    virtual void speak() = 0;
};

Animal::~~Animal()
{
    cout << "Animal纯虚析构函数调用" << endl;
}

class Cat : public Animal
{
public:
```



```

    Cat(string name) //构造函数
    {
        cout << "Cat构造函数调用" << endl;
        m_name = new string(name);
    }
    virtual void speak()
    {
        cout << *m_name << "小猫在说话" << endl;
    }
    string *m_name;
    ~Cat() //析构函数
    {
        if (m_name != NULL)
        {
            cout << "Cat析构函数调用" << endl;
            delete m_name;
            m_name = NULL;
        }
    }
};

void test01()
{
    Animal* animal = new Cat("Tom");
    animal->speak();
    //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，出现内存泄漏
    delete animal;
}

int main()
{
    test01();
    system("pause");
    return 0;
}

```

总结：

1. 虚析构或者纯虚析构都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，无法实例化对象

## 5 文件操作

程序运行时产生的数据都属于临时数据，程序一旦运行结束都会被释放

通过文件可以将数据持久化

C++中对文件操作需要包含头文件

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

操作文件的三大类：

- 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

写文件步骤如下：

- 1. 包含头文件

```
#include<fstream>
```

- 2. 创建流对象

```
ofstream ofsf;
```

- 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：

打开方式	解释
ios::in	为读文件而打开文件
ios::out	为写文件二打开文件
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`

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

// 文本文件 写文件
void test01()
{
    //1、包含头文件 fstream
    //2、创建流对象
```

```

    ofstream ofs;
    //3、指定打开方式
    ofs.open("test.txt", ios::out);
    //4、写内容
    ofs << "姓名: 张三" << endl;
    ofs << "性别: 男" << endl;
    ofs << "年龄: 18" << endl;
    //5、关闭文件
    ofs.close();
}

int main()
{
    test01();
    system("pause");
    return 0;
}

```

总结:

- 文件操作必须包含头文件fstream
- 读文件可以利用ifstream, 或者fstream类
- 打开文件时候需要指定操作文件的路径, 以及打开方式
- 利用<<可以向文件中写数据
- 操作完毕, 要关闭文件

### 5.1.2 读文件

读文件与写文件步骤相似, 但是读取方式相对于比较多

读文件步骤如下:

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

5. 关闭文件

```
ifs.close();
```

实例:

```

#include<iostream>
using namespace std;
#include<fstream>
#include<string>

```

```

void test01()
{
    ifstream ifs;
    ifs.open("test.txt", ios::in);
    if (!ifs.is_open())
    {
        cout << "文件打开失败" << endl;
        return;
    }
    //四种读取数据方式
    //1、第一种方式，数组，ifs指针右移
    //char buf[1024] = { 0 };
    //while (ifs >> buf) //右移运算符，ifs指针右移
    //{
    //    cout << buf << endl;
    //}

    //2、第二种方式
    //char buf[1024] = { 0 };
    //while (ifs.getline(buf, sizeof(buf)))
    //{
    //    cout << buf << endl;
    //}

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

    //第四种方式，不太推荐使用，效率较低
    //char c;
    //while ((c = ifs.get()) != EOF) //EOF end of file
    //{
    //    cout << c;
    //}
    ifs.close();
}

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

```

总结：

- 读文件可以利用ifstream，或者fstream类
- 利用is\_open函数可以判断文件是否打开成功
- close关闭文件

## 5.2 二进制文件

以二进制的方式对文件进行读写操作

打开方式要指定为`ios::binary`

### 5.2.1 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。

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

//二进制文件 写文件
class Perosn
{
public:
    char m_name[64];
    int m_age;
};

void test01()
{
    //1、包含头文件

    //2、创建流对象
    ofstream ofs;
    //3、打开文件
    ofs.open("person.txt", ios::out | ios::binary);
    //4、写文件
    Perosn p = { "张三", 18 };
    ofs.write((const char*)&p, sizeof(Perosn));
    //5、关闭文件
    ofs.close();
}

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

总结：

- 文件输出流对象可以通过write函数，以二进制方式写数据

## 5.2.2 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: 字符串buffer指向内存中一段存储空间, len是读写的字节数。

```
#include<iostream>
using namespace std;
#include<fstream>
//二进制文件 读文件

class Person
{
public:
    char m_name[64];
    int m_age;
};

void test01()
{
    //1、包含头文件

    //2、创建流对象
    ifstream ifs;
    //3、打开文件, 判断文件是否打开成功
    ifs.open("person.txt", ios::in | ios::binary);

    if (!ifs.is_open())
    {
        cout << "文件打开失败" << endl;
        return;
    }
    //4、读文件
    Person p;
    ifs.read((char*)&p, sizeof(Person));
    cout << "姓名: " << p.m_name << " 年龄: " << p.m_age << endl;
    //5、关闭文件
    ifs.close();
}

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

总结:

- 文件输入流对象, 可以通过read函数, 以二进制方式读数据。

去1

