**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

- 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

1.1 程序运行前：

程序未执行前分为代码区和全局区两个区域

**\*\*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（放在函数体外面的）和静态变量（用static修饰的）（  
可在函数体中修饰）存放在此.

​       全局区还包含了常量区, 字符串常量和其他常量(const修饰的全局变量 也是常量)也存放在此.

注：const修饰的局部变量（即局部常量）不在全局区

​       ==该区域的数据在程序结束后由操作系统释放==.

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修饰的全局常量  和 字符串常量



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,程序结束时由操作系统回收

​       在C++中主要利用new在堆区开辟内存

**\*\*总结：\*\***

堆区数据由程序员管理开辟和释放

堆区数据利用new关键字进行开辟内存

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！返回的是指针！

用法：

int\* a = new int(10);

int\* a=new int； //加不加括号在于有没有给a这个指针所指向的数据赋初值

而不是new int a=10；

//利用delete释放堆区数据

delete a;

注：用delete命令处理某个指针，说是把那个指针删除了是不正确的。

delete命令指示释放了那个指针原本所指的那部分内存而已。被delete后的指针p的值（地址值）并非就是NULL，而是随机值。

也就是被delete后，如果不再加上一句p=NULL，p就成了“野指针”，在内存里乱指一通。

堆区开辟数组

int\* arr = new int[10];

数据类型 数组元素个数

注意数组是中括号，普通变量是小括号

赋值要用for循环，即需要对数组元素单个单个地赋值，不能直接arr[ ]={xxx…}赋值

释放数组 delete 后加 []

    delete[] arr;

2.引用

2.1**引用的基本使用**

作用：给变量起别名

**语法：** 数据类型 &别名 = 原名

**2.2 引用注意事项**

\* 引用必须初始化 即必须要有=，不可以单纯的 int &b; 这样

//int &c; //错误，引用必须初始化

必须要 int &b=a; 这样

\* 引用在初始化后，不可以改变

即

若已经有 int &b=a;

不可以有 int &b=c;

注：可以b=c，但是这是赋值操作，不是改变了。

**2.3 引用做函数参数**

**作用：**函数传参时，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

3. 引用传递

void mySwap03(int& a, int& b)

使用时：mySwap03(a,b);

总结：通过引用参数产生的效果同按地址传递是一样的。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 即：

**int& test()**

**{**

**static int a=10; //对比下面的加了static，放在了全局区，不会被释放**

**return a;**

**}**

**Void main()**

**{**

**test()=100; //这样a就被赋值为100**

**}**

注意：**\*\*不要返回局部变量引用\*\*:**

**即不能：**

**int& test()**

**{**

**int a=10;**

**return a;**

**}**

用法：函数调用作为左值

返回静态变量引用

int& test02() {

    static int a = 20;

    return a;

}

**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（指针常量即指针变量是一个定值）.\*\***

//发现是引用，转换为 int\* const ref = &a;

void func(int& ref){

    ref = 100; // ref是引用，转换为\*ref = 100

}

int main(){

    int a = 10;

    //自动转换为 int\* const ref = &a; 指针常量是指针指向不可改，也说明为什么引用不可更改

    int& ref = a;

    ref = 20; //内部发现ref是引用，自动帮我们转换为: \*ref = 20;

结论：C++推荐用引用技术，因为语法方便，引用本质是指针常量，但是所有的指针操作编译器都帮我们做了

**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

//引用使用的场景，通常用来修饰形参

void showValue(const int& v) //用了const ,这个函数就不能修改v了

{

    //v += 10;

    cout << v << endl;

}

int main() {

    //int& ref = 10;  引用本身需要一个合法的内存空间，因此这行错误

    //加入const就可以了，编译器优化代码，int temp = 10; const int& ref = temp;

    const int& ref = 10;

    //ref = 100;  //加入const后不可以修改变量

    cout << ref << endl;

    //函数中利用常量引用防止误操作修改实参

    int a = 10;

    showValue(a);

    system("pause");

    return 0;

}

3 函数提高

3.1函数默认参数：

语法：返回值类型  函数名 （参数= 默认值）{}

注：//1. 如果某个位置参数有默认值，那么从这个位置往后，从左向右，必须都要有默认值 否则就会报错

//2. 如果函数声明有默认值，函数实现的时候就不能有默认参数 即声明和定义只能一个有默认参数。

注：有时候不用默认参数，调用函数的时候如果赋值了，优先用赋值的数，而不用默认的

3.2函数占位参数

C++中函数的形参列表里可以有占位参数，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){}

函数占位参数 ，占位参数也可以有默认参数

例子：void func(int a,int=10)

调用函数的时候一定要填补占位参数

后面有默认值的话，调用函数传参时赋值位数必须大于等于占位的参数，后面的一样是优先用赋值的数

3.3函数重载

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 参数**个数不同\*\*** 或者 参数**顺序不同（即参数类型的顺序）\*\***

**\*\*注意:\*\***  函数的返回值不可以作为函数重载的条件

例：

void func()

void func(int a)

void func(double a)

void func(int a,double b)

void func(double a,int b)

可以分别这样调用

func();

    func(10);

    func(3.14);

    func(10,3.14);

    func(3.14 , 10);

3.3.2函数重载的注意事项：

\*1、引用作为重载条件

例子：

void func(int &a)

void func(const int &a)

int a = 10;

    func(a); //调用无const

    func(10);//调用有const

2、函数重载碰到函数默认参数

void func2(int a, int b = 10)

void func2(int a)

func2(10); //碰到默认参数产生歧义，需要避免

不过传入两个参数时仍然可以调用第一个函数

func2(1,2)

4 类和对象

C++面向对象的三大特性为：封装、继承、多态

C++认为万事万物都皆为对象，对象上有其属性（成员变量）和行为（成员行为）

**\*\*例如：\*\***

​   人可以作为对象，属性有姓名、年龄、身高、体重...，行为有走、跑、跳、吃饭、唱歌...

​   车也可以作为对象，属性有轮胎、方向盘、车灯...,行为有载人、放音乐、放空调...

​   具有相同性质的对象，我们可以抽象称为类，人属于人类，车属于车类

4.1封装

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

意义1：

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  };

例：

学生类

class Student{

public：

//属性：

string name；

int ID；

//行为：

void showstdudent()

{cout<<name<<ID<<endl;}

}

意义2：

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：

//三种权限

//公共权限  public     类内可以访问  类外可以访问

//保护权限  protected  类内可以访问  类外不可以访问

//私有权限  private    类内可以访问  类外不可以访问

其中protected和private在继承上有所区别。

4.1.2 struct 和 class的区别

在C++中 struct和class唯一的**\*\*区别\*\***就在于 **\*\*默认的访问权限不同\*\***

区别：

\* 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：\*\***将所有成员属性设置为私有，可以自己控制读写权限

**\*\*优点2：\*\***对于写的权限，可以设置数据的有效范围，检测数据的有效性，防止数据超出有效范围

Private和 Protect只有类内可以访问

文件的拆分：

一．头文件（xxx.h）:

1.#once pragma

include <iostream>

using namespace std;

注：这个类中如果用到别的类，还要包含头文件#include “xxxx.h”

2.把类里面内容除了函数的实现部分全部保留，同时要在函数声明后加；

二．源文件（.cpp）

1.只用#include “xxx.h” ，不在需要别的如using… //xxx是这个源文件对应的头文件

2.对应的类只保留函数部分，（即：不用有属性）同时在函数的数据类型后面加上 xxx：：

（xxx为这个头文件的名称）

4.2

**4.2.1 构造函数和析构函数**

对象的**\*\*初始化和清理\*\***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，对其使用后果是未知

​   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，完成对象初始化和清理工作。

对象的初始化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，编译器会提供\*\***

**\*\*编译器提供的构造函数和析构函数是空实现。\*\* //空实现即函数的大括号里面什么都没有**

\* 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，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，执行一些清理工作。

**\*\*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，因此可以发生重载

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

例子：class Person{……

Person(); 或者 Person（int a…）{}

};

**\*\*析构函数语法：\*\*** ~类名(){}`

例子：

class Person{…

~Person()

{……

}

};

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

**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

**\*\*示例：\*\***

```C++

//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 //默认构造函数不是系统的函数，只是系统的函数是默认（无参）构造函数

// 按照类型分类分为 普通构造和拷贝构造

class Person {

public:

    //无参（默认）构造函数

    Person() {

        cout << "无参构造函数!" << endl;

    }

    //有参构造函数

    Person(int a) {

        age = a;

        cout << "有参构造函数!" << endl;

    }

//拷贝构造函数

一定要注意：自己写拷贝构造的时候，要写给类内属性赋值的函数，不要以为调用了拷贝构造成员函数就自动会赋值。

    Person(const Person& p) {

        age = p.age;

        cout << "拷贝构造函数!" << endl;

    }

    //析构函数

    ~Person() {

        cout << "析构函数!" << endl;

    }

public:

    int age;

};

//2、构造函数的调用

//调用无参构造函数

void test01() {

    Person p; //调用无参构造函数

}

//调用有参的构造函数

void test02() {

    //2.1  括号法，常用

Person p1(10); //调用有参构造函数

Person p2; //调用无参（默认）构造函数

//注意1：调用无参构造函数不能加括号，如果加了编译器认为这是一个函数声明

//Person p2();

Person p3(p2); //调用拷贝构造函数

//2.2 显式法

Person p1; //调用无参构造

Person p2 = Person(10); //调用有参构造

    Person p3 = Person(p2); //调用拷贝构造

//Person(10)单独写就是匿名对象  当前行结束之后，马上析构

当它放在等号右侧时，则等号的左侧就是它的名

即此时匿名对象Person（10）的名是P2。

//注意2：不能利用 拷贝构造函数 初始化匿名对象

因为编译器认为是 Person（P3）等价于 Person P3

这样的话此时相当于创建了一个对象无参构造P3，而上面有P3了，就有两个P3,这样就重定义了。这样编译器就会认为Person（P3）这是对象的声明

    //Person p5(p4);

    //2.3 隐式转换法

    Person p4 = 10; //等价于 Person p4 = Person(10); // 调用了有参构造

    Person p5 = p4; //等价于 Person p5 = Person(p4); // 调用了拷贝构造

}

int main() {

    test01();

    //test02();

    system("pause");

    return 0;

}

**#### 4.2.3 拷贝构造函数调用时机**

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

Person man(100); //p对象已经创建完毕

    Person newman(man); //调用拷贝构造函数

\* 值传递的方式给函数参数传值

void doWork(Person p1) {}//调用这个函数时会自动调用拷贝构造函数

\* 以值方式返回局部对象

Person doWork2()

{

    Person p1;

    cout << (int \*)&p1 << endl;

return p1; //返回值时调用拷贝构造函数

//注：这里返回到主函数的p1和原来的p1不是一样的，返回主函数的p1是这个函数里面的p1的一个值的copy，他们的地址不一样

}

**#### 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++不会再提供其他构造函数

即：用户写了3，编译器就不会再提供12，写了2就不会默认提供1，但会提供3

**#### 4.2.5 深拷贝与浅拷贝**

深浅拷贝是面试经典问题，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

深拷贝例子：

Person(const Person& p) {

        m\_height = new int(\*p.m\_height);//这里就是深拷贝的操作

        //注：m\_height在这个类中是指针，做笔记时后面成员属性的代码被省略了

    }

//如果不利用深拷贝在堆区创建新内存，会导致浅拷贝带来的重复释放堆区问题

析构函数可用来释放堆区开辟的数据：

例：//析构函数

    ~Person() {

        cout << "析构函数!" << endl;

        if (m\_height != NULL)

        {

            delete m\_height;

        }

    }

注：栈有个规则：先进后出，先进的数据后出来

> 总结：如果属性有在堆区开辟的，一定要自己提供拷贝构造函数，防止浅拷贝带来的问题

**#### 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）... {}`

例：

Person(int a, int b, int c) :m\_A(a), m\_B(b), m\_C(c) {}

**#### 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，我们称该成员为 对象成员

例如：

```C++

class A {}

class B

{

    A a；

}

类A就是类B 中的一个成员

例子：

Class Phone

{

Public:

string PhoneID

Phone(){};

Phone(string Pname)

{PhoneID=Pname

}

};

class Person

{

string name;

Phone iphone;

//方法一：用列表去初始化，这样Phone这个类中可以没有默认构造函数

Person(string Name,string tempname):name(Name),iphone(tempname)

//注，这里iphone（tempname）相当于写了iphone=tempname(隐式转换法)

{

}

方法二：这样写，Phone这个类里面必须要写有默认构造，只写参数构造会报错

Person（string Name,string tempname）{name=Name；iphone=tempname}

};

**#### 4.2.8 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

  //静态成员变量特点：

    //1 在编译阶段分配内存：即在成勋没有运行前就分配有内存，在全局区

//2 类内声明，类外初始化：

即例：

class Person

{

static int a; //类内声明

};

int Person：： a=100； //类外初始化

//3 所有对象共享同一份数据

即如果有另一个类p2，让a=200后，所有类中的a都变为200了

静态成员变量不属于某个对象上，所有对象都共享一份数据

故静态成员变量有两种访问方式；

1、通过对象进行访问：

cout<<p1.a<<endl;

2、通关类名进行访问：

cout<<Person::a<<endl;

注：静态成员变量也有访问权限的，如果是private下的静态成员变量，就不能类外直接访问

**\*\***静态成员函数

//静态成员函数特点：

//1 所有对象（这个成员函数所属的类的对象）共享一个函数：

//静态成员变量两种访问方式

    //1、通过对象

    Person p1;

    p1.func();

    //2、通过类名

    Person::func();

//2 静态成员函数只能访问静态成员变量

static void func()

    {

        cout << "func调用" << endl;

        m\_A = 100;

        //m\_B = 100; //错误，不可以访问非静态成员变量

原因：函数体无法区分道理是哪个对象的m\_B，而m\_A是共享的，所以可以

    }

    static int m\_A; //静态成员变量

    int m\_B; //

注：静态成员函数也是有访问权限的，同静态成员变量

**### 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**

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

即：//非静态成员变量占对象空间

//静态成员变量不占对象空间

//函数也不占对象空间，所有函数共享一个函数实例

//静态成员函数也不占对象空间

注：空对象占内存大小为1个字节

//c++编译器会给每个空对象也分配一个字节空间，为了区分空对象所占内存的位置，每个空对象也应该有一个独一无二的内存地址

**#### 4.3.2 this指针概念**

**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：

1、当形参和成员变量同名时，可用this指针来区分

2、在类的非静态成员函数中返回对象本身，可使用return \*this

例：

class Person

{

public:

    Person(int age)

    {

//1、当形参和成员变量同名时，可用this指针来区分

        this->age = age; //这样this指向的age是类中的属性，而等号后的age是形参

    }

//2、  在类的非静态成员函数中返回对象本身，可使用return \*this

    Person& PersonAddPerson(Person p)

    {

        this->age += p.age;

        //返回对象本身

        return \*this;

    }

    int age;

};

**#### 4.3.3 空指针访问成员函数**

C++中空指针也是可以调用成员函数的，但是也要注意有没有用到this指针

/空指针访问成员函数

class Person {

public:

    void ShowClassName() {

        cout << "我是Person类!" << endl;

    }

    void ShowPerson() {

        if (this == NULL) { //保证代码的健壮性

            return;

        }

        cout << mAge << endl;

    }

public:

    int mAge;

};

void test01()

{

    Person \* p = NULL;

    p->ShowClassName(); //空指针，可以调用成员函数

    p->ShowPerson();  //但是如果成员函数中用到了this指针，就不可以了

}

int main() {

    test01();

    system("pause");

    return 0;

}

如果用到this指针，需要加以判断保证代码的健壮性

**#### 4.3.4 const修饰成员函数**

//this指针的本质是一个指针常量，相当于：Person\* const this 指针的指向不可修改，指向的值可以修改

**\*\*常函数：\*\***

\* 成员函数后加const后我们称为这个函数为**\*\*常函数\*\***

例：

void ShowPerson() const //常函数语法，const加在（）后

{

        this->m\_B = 100;

    }

\* 常函数内不可以修改成员属性

在成员函数后面加const，修饰的是this的指向，就变成了const Person\* const this使得指针指向的值也不能修改

\* 成员属性声明时加关键字mutable后，在常函数中依然可以修改

例：mutable int m\_B；

//const修饰成员函数，表示指针指向的内存空间的数据不能修改，除了mutable修饰的变量

**\*\*常对象：\*\***

\* 声明对象前加const称该对象为常对象

常对象不是class const Person这样定义；而是在创建一个类变量的时候使用，例：

const Person p1;

常对象不能修改成员变量的值,但是可以访问, 但是常对象可以修改mutable修饰成员变量

例：mutable int m\_B; //可修改 可变的

\* 常对象只能调用常函数

常对象只能调用const的函数（常函数），不能调用普通函数

原因：常对象中的成员变量为定值，不可修改，如果能调用普通函数，那么普通函数如果包含修改成员变量式子，就会产生冲突，所以不能调用普通函数。

而常函数本身就有不可修改成员属性的性质，不会产生冲突。

例：

class Person {

public:

    Person() {

        m\_A = 0;

        m\_B = 0;

    }

void ShowPerson() const //常函数语法，const加在（）后

{

        this->m\_B = 100;

    }

public:

int m\_A;

例：

    mutable int m\_B; //可修改 可变的

};

//const修饰对象  常对象

void test01() {

    const Person person; //常量对象

    cout << person.m\_A << endl;

    //person.mA = 100; //常对象不能修改成员变量的值,但是可以访问

    person.m\_B = 100; //但是常对象可以修改mutable修饰成员变量

person.MyFunc(); //常对象只能调用const的函数（常函数），不能调用普通函数

}

int main() {

    test01();

    system("pause");

    return 0;

}

**### 4.4 友元**

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

\* 全局函数做友元

\* 类做友元

\* 成员函数做友元

插入一种类内写函数声明，类外写函数实现的语法，友元中不用这种语法就会报错，至于原因还得以后问老师

例：

class Person{

public:

Person(); //类内声明

void test(); //类内声明

Private:

int Age;

};

Person::Person()

{ Age=10;}

void Person::test()

{cout<<”test”<<endl;

}

**#### 4.4.1 全局函数做友元**

友元的三种实现

1.全局函数做友元

class Building

{

    //告诉编译器 goodGay全局函数 是 Building类的好朋友，可以访问类中的私有内容

    friend void goodGay(Building \* building);

public:

…}

void goodGay(Building \* building)

{

    xxxx……

}

2.类做友元

class Building

{

    //告诉编译器 goodGay类是Building类的好朋友，可以访问到Building类中私有内容

    friend class goodGay;

}

3. 成员函数做友元

class Building

{

    //告诉编译器  goodGay类中的visit成员函数 是Building好朋友，可以访问私有内容

    friend void goodGay::visit();

}

**### 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

**#### 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:有两种方式：

1、通关成员函数实现

2、通关全局函数实现

class Person {

public:

    Person() {};

    Person(int a, int b)

    {

        this->m\_A = a;

        this->m\_B = b;

    }

1、成员函数实现 + 号运算符重载

    Person operator+(const Person& p) {

        Person temp;

        temp.m\_A = this->m\_A + p.m\_A;

        temp.m\_B = this->m\_B + p.m\_B;

        return temp;

    }

public:

    int m\_A;

    int m\_B;

};

2、全局函数实现 + 号运算符重载

Person operator+(const Person& p1, const Person& p2) {

  Person temp(0, 0);

  temp.m\_A = p1.m\_A + p2.m\_A;

  temp.m\_B = p1.m\_B + p2.m\_B;

  return temp;

}

//成员函数重载本质调用： Person p3=p1+p2=p1.operator+(p2);

对<<:本质：p.operator(cout)所以只能有p在前的顺序，不能实现cout在前

//全局函数重载本质调用： Person p3=p1+p2=operator+(p1，p2);

对<<:本质：cout<<p 相当于operator<<(cout,p)

注：运算符重载 可以发生函数重载 ，即可以通关参数数据类型的不同，实现operator+函数的重载，用+的时候自动调用对应的函数

例：

Person operator+(const Person& p2, int val)

{

    Person temp;

    temp.m\_A = p2.m\_A + val;

    temp.m\_B = p2.m\_B + val;

    return temp;

}

调用的本质：

void test() {

    Person p1(10, 10);

    Person p2(20, 20);

    Person p3 = p2 + p1;  //相当于 p2.operaor+(p1)

    cout << "mA:" << p3.m\_A << " mB:" << p3.m\_B << endl;

Person p4 = p3 + 10; //相当于 operator+(p3,10)

    cout << "mA:" << p4.m\_A << " mB:" << p4.m\_B << endl;

}

> 总结1：对于内置的数据类型的表达式的的运算符是不可能改变的

即：如整型浮点型这样内置自带的运算符不可以改变

> 总结2：不要滥用运算符重载

即函数operator内部是减法，但是用+的运算符。+-\*/乱用。

**#### 4.5.2 左移运算符重载**

作用：可以输出自定义数据类型

前提 注：对：cout，我们可以鼠标右键，选转到定义，然后再下面的栏中打开，可以看到cout属于ostream（输出流//是一个类），cout是输出流中的 标准输出流对象

用成员函数实现<<重载的话：

有p.operator<<(cout) 但是这样的话，简化的版本就会变为 p<<cout 即这样的话，p<<cout 可以实现输出，但和预期的cout<<p不同，所以不用这个方法

不能实现cout<<p

所以不能用成员函数实现<<运算符，因为无法实现cout再左侧。

用全局函数实现<<重载：

//全局函数实现左移重载

//ostream对象cout只能有一个 所以要用&，不能直接传

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

    out << "a:" << p.m\_A << " b:" << p.m\_B;

    return cout; //只有有return cout，才能实现连续输出，不然的话实现不了cout<<p<<xxx<<xxx<<endl；的功能

}

注，这里函数里的cout可以改为out，a……用啥都行，不影响使用

注：全局函数访问类中的priavte时，要用friend 友元实现。

**#### 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

前置先++，再返回；所以直接对传入的数据进行++操作就行，最后返回传入的数据的引用

而后置，先返回再++，但是返回后就退出函数了，所以要创建一个临时的变量记录传入的数据，然后对传入的数据++，最后返回临时变量

而由于前后置的返回的东西不同，导致了前置可以有++++++++..双数个+，最后也能返回正确的结果，而后置返回的是临时变量，在函数运行完后就会消失，导致了，只能++而不能++++…而出现正确的结果，对自定义的后置++++不报错，但是结果也只会和++一样。

注，对后置，要在opertor++（int）中加入占位参数int，这样编译器就会自动识别这是++后置运算。

总结：对只用一个属性的类时前置递增返回引用，后置递增返回值

而对有多个属性的类时，前置的和前面一样，后置的可以在堆区开辟新的临时类，然后返回临时类回去，就可以解决函数返回只能返回一个值的问题。（自己得出的结论）

**#### 4.5.4 赋值运算符重载**

c++编译器一共给一个类添加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

所以当有属性指向堆区时，就需要我们写赋值运算符重载函数

例：

Person& operator=(Person &p)

    {

        if (m\_Age != NULL) //清空创建了的类中的属性

        {

            delete m\_Age;

            m\_Age = NULL;

        }

        //编译器提供的代码是浅拷贝

        //m\_Age = p.m\_Age;

        //提供深拷贝 解决浅拷贝的问题

        m\_Age = new int(\*p.m\_Age);

        //返回自身

        return \*this; //反复自身才能连续赋值a=b=c=d；

    }

**#### 4.5.5 关系运算符重载**

**\*\*作用：\*\***重载关系运算符，可以让两个自定义类型对象进行对比操作

注：返回bool类型

例：

bool operator!=(Person & p)

    {

        if (this->m\_Name == p.m\_Name && this->m\_Age == p.m\_Age)

        {

            return false;

        }

        else

        {

            return true;

        }

    }

**#### 4.5.6 函数调用运算符重载**

\* 函数调用运算符 ()  也可以重载

\* 由于重载后使用的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

例：

class MyPrint

{

public:

    void operator()(string text)

    {

        cout << text << endl;

    }

};

void test01()

{

    //重载的（）操作符 也称为仿函数

    MyPrint myFunc;

    myFunc("hello world");

}

这样，就实现了（）的重载，实现了打印功能，同理，还可以实现其他功能的重载

class MyAdd

{

public:

    int operator()(int v1, int v2)

    {

        return v1 + v2;

    }

};

用法：

void test02()

{

    MyAdd add;

    int ret = add(10, 10);

    cout << "ret = " << ret << endl;

    //匿名对象调用

    cout << "MyAdd()(100,100) = " << MyAdd()(100, 100) << endl;

}

**### 4.6  继承**

**\*\*继承是面向对象三大特性之一\*\***

**#### 4.6.1 继承的基本语法**

**class 子类：继承方式 父类**

class A : public B //public不唯一，可以是其他的方式

{…};

A 类称为子类 或 派生类

B 类称为父类 或 基类

例：

class Father{…};

class son : public Father

{…};

继承的好处：==可以减少重复的代码==

**\*派生类中的成员，包含两大部分\*\***：

一类是从基类继承过来的，一类是自己增加的成员。

从基类继承过过来的表现其共性，而新增的成员体现了其个性。

**#### 4.6.2 继承方式**

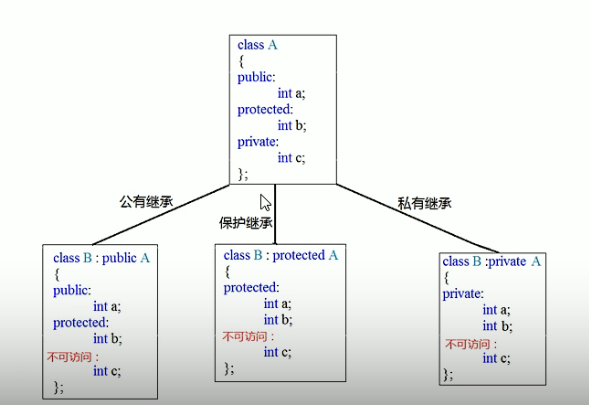
继承的语法：class 子类 : 继承方式  父类

**\*\*继承方式一共有三种：\*\***

\* 公共继承

\* 保护继承

\* 私有继承



注：父类中的私有成员只是被隐藏了，但实际上也是被继承了的。

**#### 4.6.3 继承中的对象模型**

**用vs开发人员命令提示符工具**

1. **复制文件路径**
2. 跳转盘符 G： 回车 //这样路径就跳转到G盘

3.cd 粘贴路径 回车

4.cl/d1 reportSingleClassLayout需要打印的类名 xx(Tab键补全) 回车

注：cl中的l是L的小写，d1是数字1

总结：父类中的所以非静态成员属性都会被子类继承下去

父类中私有成员属性是被编译器给隐藏了，所以访问不到，但是确实是被子类继承下去了

**#### 4.6.4 继承中构造和析构顺序**

> 总结：继承中 先调用父类构造函数，再调用子类构造函数，析构顺序与构造相反

**#### 4.6.5 继承同名成员处理方式**

\* 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

语法：s.Base：：m\_a;

//当子类与父类拥有同名的成员函数，子类会隐藏父类中所有版本的同名成员函数

即如果父类中有 void func();void func(int a);

子类中有void func（）

那么在用子类调用时，即使是s.func(10)；都调用不了func（int a），因为父类中全部函数都被隐藏掉了。

需要s.Base::func(10);

 //如果想访问父类中被隐藏的同名成员函数，需要加父类的作用域

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，子类会隐藏父类中同名成员函数，加作用域可以访问到父类中同名函数

**#### 4.6.6 继承同名静态成员处理方式**

**同上**

**区别之处在于：静态有通过对象和通过类名两种访问方式**

**静态成员在内类声明，内外定义**

**class Base{**

**public:**

**static int m\_A**

**};**

**int Base::m\_a=100;**

**class Son::public Base**

**{**

**Public:**

**int Son::m\_A=200;**

**};**

**两种访问方式：**

test01()

{

1、通过对象访问：

Son s;

s.m\_A; //访问Son中的m\_A

s.Base::m\_A; //访问Base中的m\_A

2、通过类名访问：

Son::m\_A; //访问Son中的m\_A

Son::Base::m\_A； //访问Base中的m\_A

注：Base：：m\_A也可以访问Base中的m\_A,但是意义和Son::Base::m\_A；不一样，前者是以Base作类名访问，后着是以Son作类名，访问Base作用域下面的m\_A

}

静态函数同理

注：//出现同名，子类会隐藏掉父类中所有同名成员函数，需要加作作用域访问

> 总结：同名静态成员处理方式和非静态处理方式一样，只不过有两种访问的方式（通过对象 和 通过类名）

**#### 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...

class Son:public Base1,protected Base2…

//多继承容易产生成员同名的情况

//通过使用类名作用域可以区分调用哪一个基类的成员

 总结： 多继承中如果父类中出现了同名情况，子类使用时候要加作用域

一般在开发时不使用多继承

**#### 4.6.8 菱形继承**

**\*\*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，当草泥马使用数据时，就会产生二义性。

解决方法：可以用作用域解决

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，这份数据我们只需要一份就可以。

在继承前加virtual关键字，变为虚继承（注：Animal这个第一部分的类就称为虚基类）可以解决这个问题

注：是在中间部分加

(之所以是在菱形中间部分加，我猜是因为在中间部分加，第二部分的类就产生了虚基类指针，指向第一部分的数据，从而第二部分和第一部分用的是同一个地址上的数据，然后第三部分继承也就直接继承第二部分的虚基类指针，指针还是指向第一份的数据，然后四个数据一起共用第一部分这个数据上的地址的数据。)

例：

class Sheep : virtual public Animal //这个部分是菱形的中间部分，两个角都得

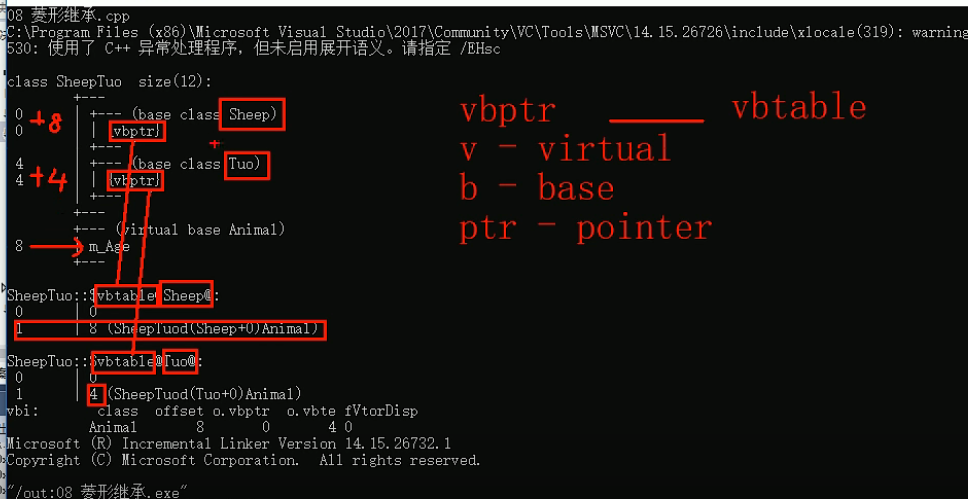
{}; 加，这里做笔记省略了另一个角

此时公共的父类称为虚基类。

不管从谁中继承其实都继承了一个指针，而这个指针指向虚基类表，表中存有偏移量，通过偏移量就找到唯一的数据（整个菱形的数据都是唯一的），从而解决了二义性的问题

vbptr（virtual base pointer）:虚基类指针，指向虚基类表

指向vbtable（虚基类表）：表中记录了 偏移量这个数据，这个vbtur对应的地址加上这个数据就指向了唯一的数据



**## 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

//多态满足条件：

//1、有继承关系

//2、子类重写父类中的虚函数

注：重写：函数返回值类型  函数名 参数列表 完全一致称为重写

重载只是函数名不同。与重写不一样

//动态的多态使用：

//1、有继承关系

class Animal

{

virtual void speak() //父类中需要实现多态的函数前加virtual

{

cout<<”动物在说话”<<endl;

}

}

class Cat

{

virtual void speak() //重写父类的函数，注：这里的virtual可写可不写

{ 但是就算不写，这个子类中的函数仍是虚函数

cout<<”猫在说话”<<endl;

}

};

2、//父类指针或引用指向子类对象 用引用即下例。用指针：Animal\*a=new Cat;

即： 或：Cat cat；

Animal \*animal=&cat；

void DoSpeak(Animal & animal)

{

    animal.speak(); //注：如果不使用virtual，多态，这里就会调用打印”动物”的函数

} 用了virtual，就会调用打印”猫”的函数 （即，tese01（）中传入的子类中对应的函数）

在用这个函数时，传入的是cat，而不是animal

void test01()

{

    Cat cat;

    DoSpeak(cat);

}

用多态方法，在子类重写的函数所对应的父类的函数前面加virtual关键字，子类中的可加可不加

**#### 4.7.1 多态的原理剖析**

**在父类的函数前加virtual关键字后，父类的结构就法师了变化。**

**没加之前这个结构占1个字节的内存空间（有前面类的大小，已经类中函数不是储存在类中的知识，而空类默认占一个字节）**

**加了之后变成了4个字节大小的内存空间，内部结构中对了个虚函数（表）指针（vfptr），而任何指针都是4个字节，而这个指针指向一个虚函数表（vftable），表内部记录一个虚函数（父类中前面加了virtual的函数）的地址，在上例中即记录了：**

**&Animal：：speak（） 这个虚函数的入口地址**

**注：子类中的vfptr和父类中的vfptr是两个不同的指针，各自指向自己的vftable**

**关于继承的类需要重写：**

**如果不重写，那么Cat这个子类中的 虚函数表 就会直接继承 上面的虚函数表中的内容 ,也就是和父类的一样的，记录了 &Animal：：speak（） ，故子类中的vfptr指向的子类的虚函数表中 和父类中虚函数表一样的内容：&Animal：：speak（）**

**而如果子类中重写了父类虚函数，那子类中的虚函数表内部会替换成子类的虚函数地址，即&Cat：：speak()**

**所以当父类的 指针或者引用 指向子类的对象时，由于本身还是一个子类对象，所以调用公共的speak（）接口的时候，编译器会在子类的虚函数表中确定函数的入口地址，就发生了多态**

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以及维护

**#### 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

**#### 4.7.5 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\*\***

是在父类的析构函数中用虚析构，子类不需要写（纯）虚析构，但是子类还是要写析构函数，（纯）虚析构解决的是：子类中的析构函数不被调用，

（注：注意区分多态和继承中构造和析构不同，在继承中会父构子构子析父析，但是多态是在继承的基础上还多了个用父类指针指向子类对象，而父类指针在指向子类对象的时候不会调用子类的析构函数，这个就是子类析构函数不被调用的原因）

用来（纯）虚析构后，子类中的析构函数才会被调用。

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 空实现{}也行

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：

类内：

virtual ~类名() = 0;`

内外

类名::~类名(){}`

总结：

​   1. 虚析构或纯虚析构就是用来解决通过父类指针释放子类对象

虚和纯虚的区别：拥有纯虚析构函数的类就变成了抽象类。而如果类中只有虚析构而没有别的虚函数，那这个类不是抽象类

​   2. 如果子类中没有堆区数据，可以不写为虚析构或纯虚析构

​

**## 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#include<fstream>

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 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：

| 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

**\*\*示例：\*\***

```C++

#include <fstream>

void test01()

{

    ofstream ofs;

ofs.open("test.txt", ios::out); //注：test.txt不加路径就默认保存至

vs项目所在的路径下，这里可以添加路径

例：

ofs.open(“G:\\C++\\text.txt”,ios::out); 注：要用[\\而不是\](file:///\\\\而不是\\)

    ofs << "姓名：张三" << endl;

    ofs << "性别：男" << endl;

    ofs << "年龄：18" << endl;

    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 ofstream  ，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);

3.5.判断文件是否打开成功

if (!ifs.is\_open()) //返回bool值

    {

        cout << "文件打开失败" << endl;

        return;

    }

4. 读数据

   四种方式读取

5. 关闭文件

   ifs.close();

**\*\*示例：\*\***

```C++

#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;

    }

    //第一种方式 一个一个字节

    //char buf[1024] = { 0 };

    //while (ifs >> buf) 把ifs写入buf中，然后传回bool类型，文件尾部会传回0；

    //{

    //  cout << buf << endl;

    //}

    //第二种 //用getline函数一行一行地获取

    //char buf[1024] = { 0 };

    //while (ifs.getline(buf,sizeof(buf)))

    //{

    //  cout << buf << endl;

    //}

    //第三种

    //string buf;

    //while (getline(ifs, buf)) //这个getline函数是全局函数，与三不同

    //{

    //  cout << buf << endl;

    //}

第四种： //不推荐用，一个一个字符读，效率低

    char c;

    while ((c = ifs.get()) != EOF) //get函数每次读一个字符

    {

        cout << c;

    }

    ifs.close();

}

int main() {

    test01();

    system("pause");

    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

**\*\*示例：\*\***

```C++

#include <fstream>

#include <string>

class Person

{

public:

    char m\_Name[64];

    int m\_Age;

};

//二进制文件  写文件

void test01()

{

    //1、包含头文件

    //2、创建输出流对象

    ofstream ofs("person.txt", ios::out | ios::binary);

    //3、打开文件

    //ofs.open("person.txt", ios::out | ios::binary);

    Person p = {"张三"  , 18};

    //4、写文件

    ofs.write((const char \*)&p, sizeof(p));

    //5、关闭文件

    ofs.close();

}

int main() {

    test01();

    system("pause");

    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：字符指针buffer指向内存中一段存储空间。len是读写的字节数

示例：

```C++

#include <fstream>

#include <string>

class Person

{

public:

    char m\_Name[64];

    int m\_Age;

};

void test01()

{

    ifstream ifs("person.txt", ios::in | ios::binary);

    if (!ifs.is\_open())

    {

        cout << "文件打开失败" << endl;

    }

    Person p;

    ifs.read((char \*)&p, sizeof(p));

    cout << "姓名： " << p.m\_Name << " 年龄： " << p.m\_Age << endl;

}

int main() {

    test01();

    system("pause");

    return 0;

}

```

- 文件输入流对象 可以通过read函数，以二进制方式读数据