## C++与C的比较

C和C++一样都是静态编译型语言

1. **C没有对全局变量重复定义的检测**

定义两个相同的变量，一个初始化一个不初始化，不会出现重复定义的错误

1. C的一个结构体，在定义结构体变量时，struct必须加上，除非用typedef定义一个新类型.而C++可以直接拿struct 后的名字来定义结构体变量
2. **C的三目运算符不能当作左值**，如：

A=10；

B=20；

C=0;

C=A<b?a:b; //返回a的值.如果 A<b?a:b = 50，则错误，那么可以返回地址来改变。\*(A<b?&a:&b) = 50,那么a的值变成了50.

在C++中，可直接 ((a<b?a:b)) = 50;

1. C中const修饰变量：const int a=10; a值不可直接赋值，相当于加了一个**只读**修饰。但是**可以通过指针改变** ,用同样的方法C++中不能改变常变量，只改变了这个指针的值.

Const int a=10; //在C++中常变量一定要初始化值

Int \*p = (int \*)&a; //c++中就变成了对常量取地址，那么编译器会开辟一个临时空间

\*p = 20;

**在c中，a和\*p的值都变为20，但是在C++中,a=10,\*p=20.**

**原因**：C++中编译时发现一个常量，就将常量放置在一个符号表(**常量区**)内，key对相应的value ，**常量没有空间和地址**. 然后在后面使用到时从符号表提取替换.(也是编译阶段)

**与define的区别**. Define的数据是在**预处理阶段**把宏展开，而常变量是在**编译时**扫描符号表，将常变量替换成对应的value.

1. C中枚举类型的变量可以给其赋值数字或者枚举的成员。但是在C++中，只能给枚举变量赋值枚举定义的成员

Enum num

{

One=0;

Two,

Tre,

fou

};

Enum num n = 2 或者Enum num n = tre 在C中都可以，C++中只有第二种可以

6. C宏没有语法检测，所以有了内联函数

## 2.C++的特性

### 1.引用可以理解为一个变量的别名(重要)!!!!!!!

int &a = b;代表a是b的一个引用 int&是引用的数据类型.

1.引用没有定义，是一种关系型声明，不分配内存，与被引用的变量地址相同

2.声明的时候必须初始化，引用不可修改成别变量的引用

3.可对引用再次引用，多次引用的结果是一个变量有多个别名

4.&前有数据类型时，代表引用

**要对常量进行引用，必须加上const    如const int &＝10。**

形参类：**通过引用可以不使用指针传递形参来改变值**。

Void change\_val(int \*a) 可以改成：void change\_val(int &r)

{ {

\*a=10; r=10;

} }

Change\_val(&a); change\_val(a);

打印结构体时：

struct student s = {123,”yjw”};

Printf\_S(struct student s1)//传入形参时有一个复制过程 Printf\_S1(struct student \*s1)

{ { //没有复制过程，推荐使用

Cout << “id: ”<<s1.id<<endl; Cout << “id: ”<<s1->id<<endl;

} }

**Printf\_S(s);** **print\_S1(&s);**

**第一种传入形参有数值复制过程，效率低**

如果**使用引用**更加方便

Printf\_S(struct student &s1)//传入形参时有一个复制过程

{

Cout << “id: ”<<s1.id<<endl;

}

**print\_S1(s);**

int \* get\_malloc(int num)

{

Int \*p = NULL;

P = (int \*) malloc(num);

If(p==null)

Return NULL;

Return p;

} //可以返回地址

Int & getA(void)

{

Int a=10; //如果返回的是static类型的则下面两个方式接收都没影响

Return a;

}

Int b = 0; b=getA(); 在这句执行完局部变量空间才被回收，这个过程有赋值过程，没影响，所有正确 .

#### 返回值为引用

**返回值类型为引用时，这个函数也可以作为左值 如getA = 1000;正确**

**但是要想使得函数可以作为左值，必须返回类型为引用。且返回的变量应该为全局变量或者static或者malloc的类型。因为作为函数左值其实是给作为返回的那个变量赋值的。**

**也就是返回值类型为引用，该函数当作左值可以对返回的static变量进行赋值，作为右值可以得到该变量的值!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!**

**返回引用类型，必须在用一个变量来接收，不接受之间拿该函数进行代码编写，会造成非法地址。如下**

Int &b = getA();或者int b = getA()+c; 产生错误。对函数内局部变量进行引用。该语句执行完之后，内存被释放，然后引用一个未知的地址

**所以:如果引用作为函数返回值类型，不能用引用类型的变量来接收返回值**

如果对一个常变量进行引用，那必须是const引用

### 2.声明命名空间的三种方式：

1.直接使用 std::cout << “aaaaa” << std::endl; //：：代表作用域

2.声明命名空间的一个变量 头文件下声明 using std::cout

cout << “aaaaa” << std::endl;

3.声明命名空间中的全部变量 using namespace std;

cout << “aaaaa” << endl;

**定义一个命名空间**

Namespace space\_1

{

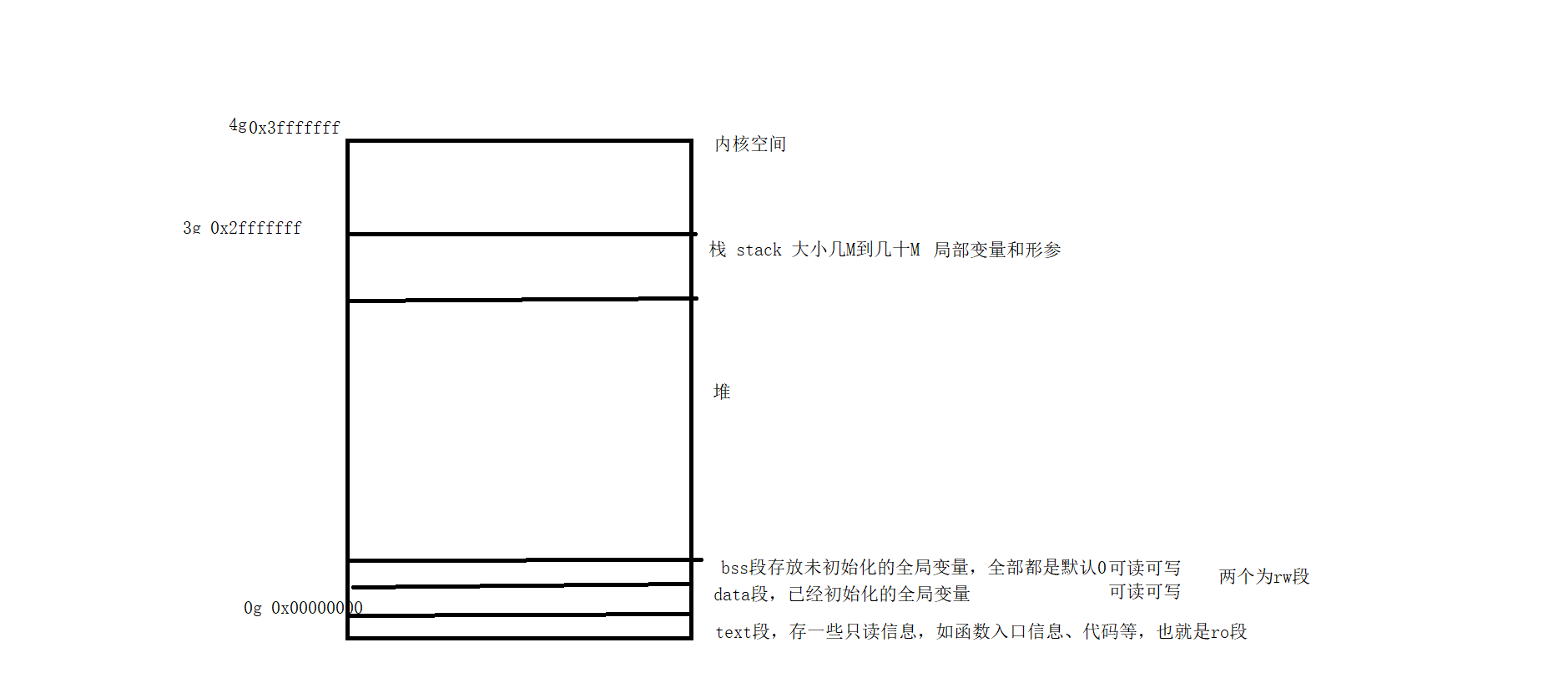
Int s\_1 = 10;

} //不用加分号

### 3.虚拟内存：

读写速度顺序：

寄存器>缓存(几k)>内存(几G)>硬盘>网盘或网线



### 4.内联函数inline

函数前加个inline则声明该函数也产生和宏一样的效果(展开操作)。不需要进行压栈出栈操作。减小内存开销.适用于比较简单且频繁使用的函数

这个展开操作属于编译器阶段。并不是预处理阶段.好处是: 能够校验语法

内联函数的特点:

1. 内联函数声明时inline关键字必须和函数定义结合在一起，否则编译器会直接忽略内敛请求
2. C++编译器直接将函数体插入在函数调用的地方
3. 内联函数没有普通函数调用时的额外开销(压栈、跳转、返回)
4. 内联函数是一种特殊的函数，具有普通函数的特征(参数检查，返回类型等)
5. 内联函数由编译器处理，直接将编译后的函数体插入调用的地方，宏代码片段由预处理处理，进行简单的文本替换，没有任何编译过程。
6. C++中内联编译的限制：
   1. 不能存在任何形式的循环语句
   2. 不能存在过多的条件判断语句
   3. 函数体不能过于庞大
   4. 不能对函数进行取址操作
   5. 函数内联声明必须在调用语句之前

总结：

优点：避免调用时的额外开销

代价：代码段中出现多个“副本”，因此会增加代码段的空间

本质：以牺牲代码段空间为代价，提高程序的运行时间的效率

适用场景：函数体很**小**，且被**频繁**调用

### 5.默认参数和占位参数、函数重载

Void func(int b ,int a = 999) //默认参数，调用时不传递形参默认a=999 默认规则从右到左,左边的有默认值，那么右边的参数必须有默认值.方便传入参数

Void func2(int x，int ) //占位参数 在重载时用到.(亚元) 调用时可以不传递参数

Int func (int a , char b)

返回值 函数名 形参列表

Int func(int a , int b) //函数重载，根据传入参数类型，自动选择调用的函数

//函数重载只能形参列表不同，形参列表相同但是返回值不同的话也还是重定义 也就是函数重载跟函数名和形参列表有关

重载底层实现：

C++利用name mangling (倾轧)技术，来改名函数名，区分参数不同的同名函数.

实现原理：用v c i f l d 分别表示 void char in float long double 及其引用

Void func(char a); //底层改名为func\_c(char a)

Void func(char a ,int b,double c); //func\_cid(char a,int b,double c)

### 6.类和对象

类相当于C中的结构体类型，对象相当于这个类型定义的变量

Class student {

Char name[64];

Int sex;

};

Student stu; 可以理解分别为类和对象

**和C的区别：**

类具有访问权限

类内可以包含成员函数

类内有四种默认函数：默认构造、默认拷贝构造、默认等号操作符、默认析构

构造函数：三种-无参构造、有参构造、拷贝构造

构造函数可以重载。构造函数结构:没有返回值，函数名和类名相同，参数列表不同时可以重载.在定义对象时调用

一般用来做初始化

每个类在创建时，如果没有定义构造函数，在编译时会默认调用一个Test{} 的无参构造。

在构造函数内再调用构造函数是危险的行为

析构函数：

在一个对象临死(生命周期结束)之前，会自动调用析构函数.

析构函数无参，不存在重载

一般用来做为释放空间使用.

析构函数也有一个默认的。在两个函数出现定义时，默认的两个就不存在

拷贝构造函数:

Test t1;

Test t2(t1); 这样不会报错，因为会调用一个默认的拷贝构造函数,和Test t2=t1相同,都是调用拷贝构造函数.当定义完t2后,再进行 t2 = t1 不一样，后者是调用一个赋值操作符重载。

Void operator = (Test &another){ … }

**注意(深拷贝与浅拷贝):**

当初始化对象时，有分配空间过程，且在初始化另一个对象时，使用的是默认拷贝构造函数(Test t2(t1))。 那么这过程为一个浅拷贝过程，单纯的把t1分配的地址让t2的成员函数指向它，当两个生命周期结束时，t1析构函数执行正常(free)，t2free过程会出现错误。造成程序崩溃

**如:**

Class Test

{

Public:

Test(int a,char \*b)

{

\_num = a;

Int len = strlen(b);

\_name = (char \*)malloc(len+1);

Strcpy(\_name,b);

}

~Test()

{

If(\_name!=NULL)

Free(\_name);

}

Test(Test &anther) //默认的拷贝构造函数

{

\_num = anther.\_num;

\_name = anther.\_name;

}

Test(Test &anther) //自定义的拷贝构造函数

{

\_num = anther.\_num;

Int len = strlen(anther.\_name);

\_name = (char \*)malloc(len+1);

Strcpy(\_name,anther.\_name);

}

Private:

Int \_num;

Char \*\_name;

}

定义对象:

Test t1(10,”yjw”);

Test t2(t1); //当没有定义拷贝构造函数时，销毁该对象时，程序崩溃，定义后正常。

**构造函数初始化列表**：

Test A

{

Public:

Test(int a)

{

\_a = a;

}

Privare:

Int \_a;

}

Test B

{

Public:

Test(int a int c,int b):\_a(a),\_c(c) //构造函数初始化列表。初始化顺序按privare //声明顺序来.先初始化\_a,再初始化\_c.列表初始化完成后，初始化构造函数体内

{

\_b = b;

}

//也可以写成下面的形式

Test(int a,int b,int c):\_a(a),\_c(c),\_b(b)

{

}

Privare:

Test \_a

Int \_b;

Test \_c;

Const int d; //因为在定义变量时不能初始化，所以常变量只能放在构造函数初始化列表内初始化 （:\_d(d)）

}

### new和delete

C++里面的new和delete是一个关键字，并非像malloc一样是函数。

1.分配一个空间：

Int \*p = new int; //给p分配一个sizeof(int)大小的空间

Delete p; //释放分配给p的空间

2.分配多个空间（10）:

**C语法**:

**int \*p = (int \*)malloc(sizeof(int)\*10);**

**赋值**：for(i=0;i<10;i++) \*(p+i) = I;

Char \*p = (char \*)malloc(sizeof(char)\*10);

Strcpy(p,”12345”);

**释放**: free(p);

**C++语法**：

**Int \*p = new int[10];** //若是int \*p = new int(10)这个意思是初始化为10，长度还

是sizeof(int)

**赋值**：和C一样

**释放**:**delete[] p;**

**区别：**

1. malloc 的空间可以用delete来释放
2. new的空间也可以用free来释放

由此可见他们实现的方法可能是一致的。但:

区别1：malloc是函数，new是关键字,因为没有压栈出栈所有速度快

区别2：malloc在给一个指针对象分配空间时，不能初始化，当没有init\_data()的接口时，

造成数据乱码，但是new在分配空间时可以之间初始化如 Test \*tp = new

Test(10,20)也就是可以触发类的构造函数;

区别3:同理free时不会触发类的析构函数，而delete可以;

### 8.静态成员函数和静态成员变量

Class Test

{

Public:

Test(int a,int b)

{

\_a=a;

\_b=b;

}

Static int c;

Int get\_c(void){ c++;return c; }

Private:

Int \_a;

Int \_b;

};

Int Test::c = 0; //静态变量一定要初始化，且在类外面进行初始化,不管私有还是公有都如此

静态成员变量说明：

1.该静态成员变量属于整个类，所有该类定义的对象，所访问的这个静态成员变量都指向

同一地址。如 Test a1(10,20); Test a2(100,200);(a1.get\_c ==1) (a2.get\_c == 2)

2.如果变成私有的静态成员变量，所有该变量对类外的接口函数前都要加上static

静态成员变量特点：

1. 静态成员实现了同类对象之间的信息共享
2. **Static成员类外储存，计算类的大小，并不把其计算在内**
3. Static成员是命名空间属于类的全局变量，存储在data区
4. Static成员只能在类外初始化
5. 可以通过类名访问(无对象定义时也可以)，也可通过对象访问\

静态成员函数：

1. 静态成员函数的意义不在于信息共享，数据沟通，而是在于管理静态数据成员，完成对静态数据成员的封装
2. 静态成员函数只能访问静态数据成员,原因：非静态成员函数在调用时this指针被当作参数传进。而静态成员函数属于类，不属于对象，从而没有this指针.

### 9.this指针及返回对象本身

this指针在类中，被隐藏起来，是一个指针，指向这个对象的地址。

函数可以准确访问类的成员变量，就是通过this来进行区分不同对象的成员

且类型是一个 Test \*const this类型

在

Class Test

{

Public:

Int get\_a()

{

//这里this->\_a可以被赋值，代表this不是一个 const Test \*类型

//这里this++出错，代表是一个常指针 Test \*const

//如果这里不想使得this指针的成员值可以被修改，变成一个只读类型

//的指针可以再对其修饰为const Test \*const 类型。但是this指针是被隐//藏起来的，想要修饰可以在这个函数后直接加上const 如 int get\_a() //const {}

Return \_a; // 相当于 return this->\_a;

//如果要在一个函数内返回对象本身，那么要使用 \*this返回.如 Test &

// get\_a() { return \*this; }

}

Private:

};

### 10.友元函数和友元类

在类内，用friend关键字声明一个函数为该类的友元函数。且是一个全局函数。那么在该函数体内可以直接访问类的私有成员。声明函数可以在类内任意地方进行声明。

好处是当该函数需要大量进行引用时，通过对象的成员函数进行访问会有过多的压栈出栈过程、类型检查、安全性检查。声明为友元函数时，可以减少内存的开销，提高程序效率.

坏处是破坏了类的封装性和隐藏性，使得非成员函数可以访问私有成员

同类对象间无私处，异类对象间有友元。

如果想在另一个类内访问一个类的私有成员，那么可以将另一个类声明为这个类的**友元类**。方法，如果想让B能访问A的私有成员，可以在类A中声明 friend class B; 这是B不能访问A私有成员。

友元的注意事项：

1. 友元关系不能被继承
2. 友元关系是单向的，不具有交换性
3. 友元关系不具传递性，如B是A的友元类，C是B的友元类，若A中没有C的友元声明，那么C不是A的友元类

### 11.操作符重载

#### 1操作符重载基本语法

Test operator+(Test &t1,Test &t2) //对+进行操作符重载，关键字operator与操作符之间不能有空格。调用时 Test t3 = t1+t2; //相当于Test t3 = operator+(t1,t2);

**注意：**

Test

{

Public: //调用时相当于 Test t3 = t1.operator+(t2);

Test operator+(Test &anther){} //类内定义和全局定义(上面的基本语法)这些格

//式最终会被翻译成同一种重载，在调用时会冲突，只能定义一个。

Private:

Int a;

Int b;

}

#### 2.操作符重载规则

1. C++不允许用户自己定义新的操作符，只能对已有的C++操作符进行重载

2. C++中允许的重载操作符：

绝大部分都可以，不能重载的只有四个：‘.’成员选择符、‘.\*’成员对象选择符、‘::’域解析操作符、‘?:’条件操作符 Class A{char \*p}; A a; a.\*p==🡺\*(a.p);

3. 重载不能改变运算符的运算个数:根据运算个数可以分为单目运算符和双目运算符

4. 重载不能改变运算符的优先级、不能改变运算符的结合特性

5. 操作符重载不能有默认参数

6. 重载时必须有一个或多个参数为自定义类型 如 int operator(int a,Test b)

7. 用于类对象的操作符必须重载，但‘=’和‘&’不建议重载

8. 重载后的功能和之前的功能尽量相似，如+重载后还是实西现加的功能

9. 运算符重载可以是类的成员函数、类的友元函数也可以是普通函数

#### 3.重载示例

##### 3.1++操作符重载

**写在全局：**

Test &operator++(Test t1)

{

T1.a++;t1.b++; return t1;

}; //重载的是 ++t2;

Const Test operator++(test t1,int) //没错! Int a=10; ++++a对，a++++错，所以要加const

{

Test temp(t1.a,t1.b); t1.a++; t1.b++; return t1;

}; //**加一个占位符**，这时重载的是后++，t2++；

最后在类内用friend修饰即可

**写成类的成员函数：**

Test & operator++() //重载前++

{

This->a++;

This->b++;

Return \*this;

}

Const Test operator++(int)

{

Test temp(this->a,this->b);

This->a++;

This->b++;

Return temp;

}

##### 3.2左移右移重载比较常用：

比如cout << a << endl; //两个变量属于双目运算符

**写在全局：**

Ostream& operator<<(ostream &os,Test &t1) //cout类型为输出流

{

Os << t1.a << ‘”,” << t1.b << endl;

Return os; //原本的能cout << x << xx << endl; 连续使用<< 说明返值是cout

} //调用方法： cout << c1 << c2 << endl; （operator<<(cout,c1)）

**写在成员方法中：**

Ostream & operator<<(ostream & os)

{

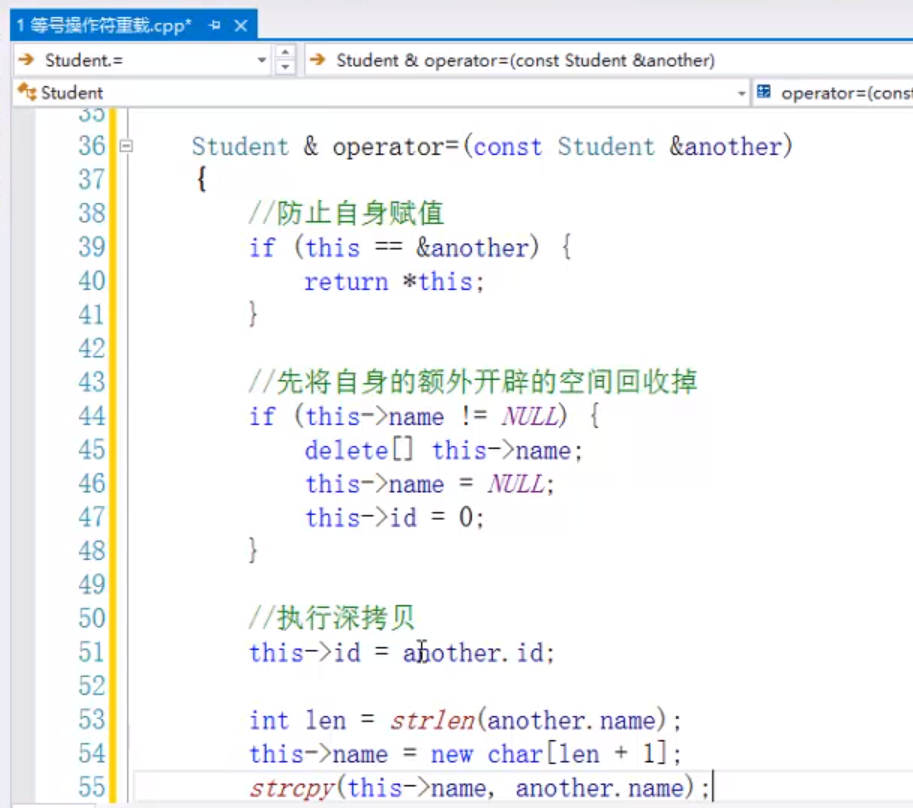
Os << this->a << “,” << this->b << endl;

Return os;

} **//调用方法变成：c1 << cout ; (c1.operator<<(cout)，所以不建议将<<重载写到类**内

右移方法和特性一致!

##### 3.3 等号操作符重载



Return \*this;

}

##### 3.4小括号重载

将一个对象当成一个普通函数来调用时，称这种对象为仿函数。如想实现一个平方。

Class square

{

Public:

Square()

{}

~square()

{}

Int operator()(int a)

{

Return a\*a;

}

Private:

Int \_a;

}

Square s;

Int value = S(3); //value == 9;

##### 3.5重载New、delete

重载new，依然会触发对象的构造函数。重载new和delete时与operator要隔空格

Class Test

{

Public:s

Test() {}

Test(int a)

{

\_a = a;

}

~Test()

{

}

**Void \* operator new(size\_t size)**

**{**

**Return malloc(size);**

**}**

**Void operator delete(void \*p)**

**{**

**If(p!=NULL)**

**{Free(p);p = NULL;}**

**}**

Private:

Int \_a;

}

Int main()

{

Test \*t1 = new Test(10); //会调用构造函数,调用过程 **t1->operator new(sizeof(Test))**

Delete t1; //会触发析构函数 t1.operator delete(t1)

Return 0;

//如果想写 Test \*t1 = new Test[n],则在重载new的基础上在new后直接跟上[]即可，

//完全不用改变 (void \* operator new[](size\_t size){})

}

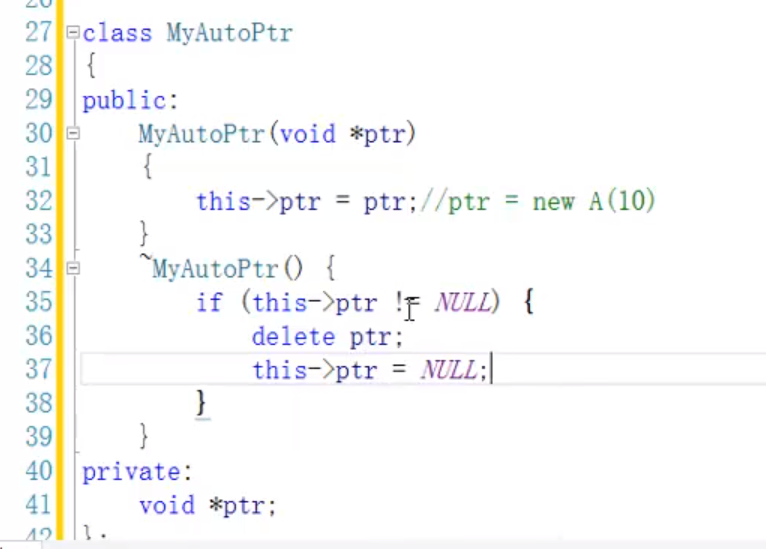
##### 3.6注意

1.不建议重载&& 和|| 操作符,因为会失去操作符原有的一些特性，比如&&，if(0 && 1)如果前半部分已经可以判断出改语句的结果，那么后半部分不会被执行到

### 12.智能指针

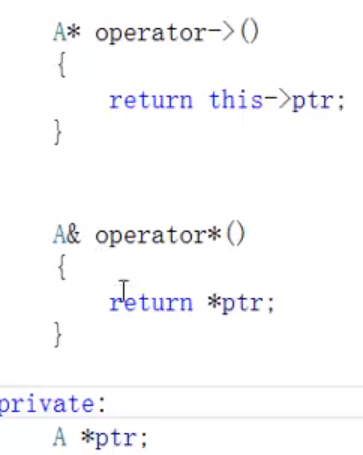
**开辟的空间会自动回收的指针，引用时需要引入的头文件 <memory>**

**智能指针的实质是一个类**

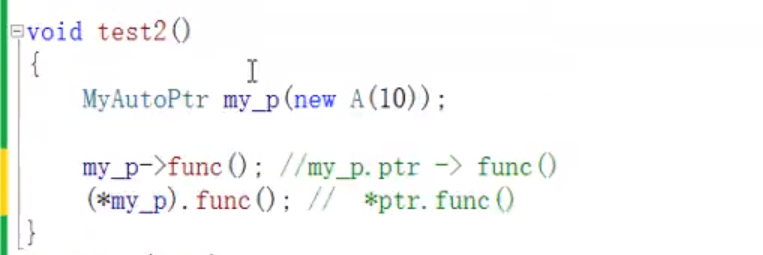


**如这个类，不过还要重载一些必须的操作符，如->、\* 、= 等等**

**如重载->和\*的方法：**



**自定义类调用方法：**



使用：

Auto\_ptr<int> 🡸🡺 int \*

Auto\_ptr<int> ptr(new int) //定义一个int类型的指针ptr并分配空间

使用起来与普通的指针无异

### 13.string类

引入的头文件<string>，是一个类

任务：

写一个自定义的string类



**Private:**

**Int** len;

**Char** \*str;

去实现的其中一些功能。

### 14.继承

编程追求的是高内聚，低耦合。继承的耦合度很高

**类和类之间有三种关系**

#### 类B拥有类A的成员变量(B has A)

Class B

{

Public:

Private:

**A \_a;**

}

#### 类C拥有类A的形参或者定义友元(C uses A)

class C

{

Public:

Void print(**A & a**)

{}

Private:

Int \_a;

}

#### 3.类D继承于A(D is A)

Class A

{

Public:

Private:

Int \_a;

}

**Class D : publicA** //D继承于A，可以表示为 D is A

{

Public:

Void print(void)

{

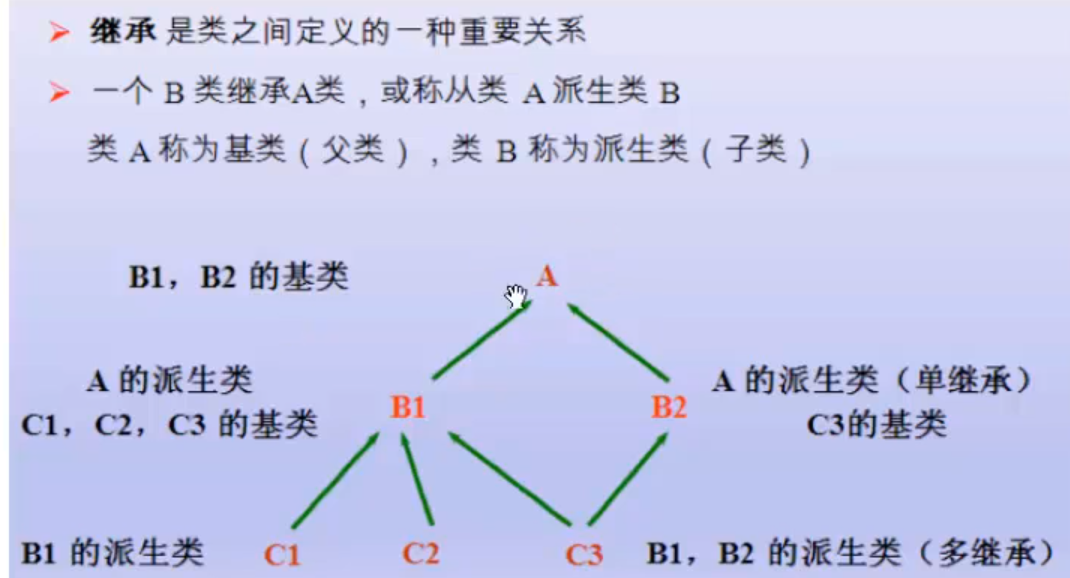
Cout << \_a << endl; //可以直接调用类A的成员

}

}

继承关系具有传递性，不具对称性

在c++中可重用性就是通过继承来实现的



##### 3.1-类的继承方式

###### 1.公有继承

**Class B : public A //这两个public是两个完全不同的概念，如void 和void\*.一个表示**

**//继承方式，一个表示访问权限**

{

**Public**:

Protected:

Private:

}

###### 2.保护继承

**Class C:protected A**

{

Public:

Protected:

Private:

}

###### 私有继承

Class D : private A

{

Public:

Protected:

Private:

}

**总结：**

(**public类内外都可访问，protected、private类内可访问类外不可，不可见类内外都不可**)



1.公有继后父类的成员变量保持原来的访问权限

2.保护继承后在子类中父类的访问权限除private外都变成保护权限(即只能在类内访问)

3.私有继承后在子类中父类的访问权限都变成私有

**4.不管哪种继承方式父类的私有成员在子类中都变为不可见**

#### 4.类的赋值兼容原则

1.子类对象可以当作父类对象使用

Child c;

c.print(); //**子类对象可以直接调用父类的成员方法**

2.子类对象可以直接赋值给父类对象

Child c; //子类

Parent p; //父类

P = c; //赋值

反过来不行，因为子类对象的空间包含有父类对象，可以将父类对象所占用的空间

填满。类似包含和包含于的关系

3.子类对象可以直接初始化父类对象

Child c;

Parent p = c; //初始化

4.父类指针可以直接指向子类对象

和2中的解释相似

Parent \*pp; //父类指针

Child \*cc; //子类指针

Child c; //子类对象

Parent p; //父类对象

Cc = &c; //子类指针指向子类对象肯定没问题的

Pp = &c; //父类指针同样可以指向子类对象

// 反过来 cc = &p 错误

好处可以定义一个函数接收多种对象

Void print(parent \*ppp)

{

Ppp->func();

}

调用时：

Print( &p); //可以

Print( &c); //可以

5.父类引用可以直接引用子类对象

引用和指针类似。所以大致原理相同

#### 5.子类的构造和析构

子类的构造和析构会自动调用父类的构造和析构。

先触发父类的构造再进行子类的构造，先调用子类的析构再调用父类的析构

#### 6.父类和子类出现变量重名的情况

当变量名相同时,父类成员变量为私有，父类中要提供该变量的调用方法.

若父类的成员变量为公有的，那么直接通过作用域域名直接调用即可(parent::a)

#### 7.继承中的static

静态成员变量也继承了下来，整个家族指向同一个静态成员变量的地址

#### 8.多继承与虚继承

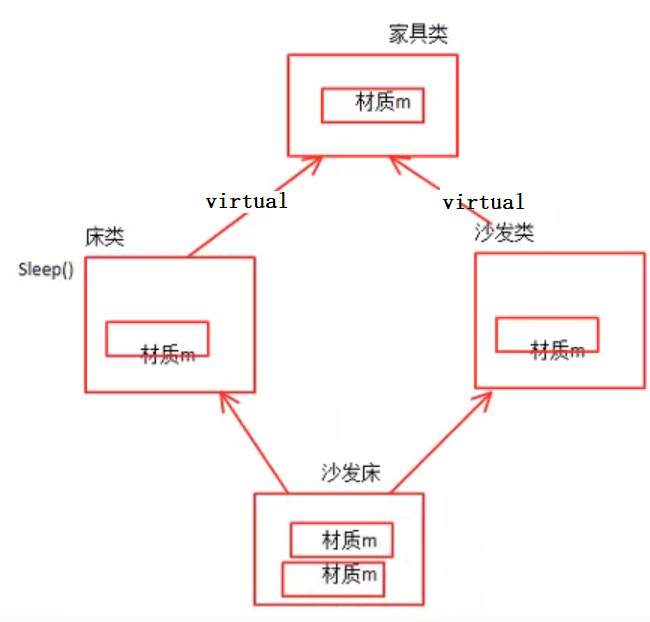
多继承的语法：

**Class C : public A,public B**

{

}

虚继承概念：多继承带来以下的问题，那么引入虚继承的方法来解决



子类的子类会产生两个变量并有两个空间。

在调用时产生歧义,造成不能访问。目标变量不明确。（SofaBed sb;sb.m; //出错）

那么**这时只能通过作用域来访问： sb.sofa::m**;

或者**在子类继承父类时加一个关键字virtual，将子类变成一个虚继承** 如上图

**产生的结果**，子类的子类发现有相同的变量，那么只会出现一个变量空间

**用法**： class B : virtual public parent{…} 子类1

class C : virtual public parent{…}子类2

Class D : public B,public A{…} 孙子

### 15.多态

##### 1.多态现象

如果有几个不完全相同的对象，有时人们向它们发出同一个消息时，他们的反应各不相同且分别执行不同的操作.这种现象就是多态现象。如上课铃声响，不同的班级走向各自的教室。

##### 2.定义

C++中所谓的多态是指，由继承而产生的相关的不同的类，其对象对同一消息做出的不同反应

##### 3.意义

多态性是面向对象程序设计的一个重要特性，能增加程序的灵活性，可以减轻系统的升级、维护、调试的工作量和复杂度

多态是设计模式的基础，是框架的基础

##### 4.应用和虚函数

Class parent //父类

{

Public:

Void Func(void){…}

Private:

};

Class child

{

Public:

Void func(void){…}

Private:

};

问题：

Void print( parent \* p) //父类指针

{

p->func(); //**希望传递进来的如果是子类，就调用子类的成员方法**

//**如果是父类，就执行父类的成员方法，这种行为就是多态**

}

**在调用该函数时，编译器做了一个安全的处理，编译任务，不管传入的是子类对象还是父类对象，如果都执行父类的成员方法，那么一定会被成功执行。**

**所以为了解决这种方法，要将重名的函数前加 virtual,如 virtual void func(void){…**}

**只在父类这个相同的函数前声明为虚函数即可**

##### 5.多态发生的条件

1.要有继承

2.父类和子类中有函数名一模一样(虚函数重写行为)

3.父类指针或引用指向子类对象

##### 6.多态的原理vptr指针\*

如果说一个类有virtual 关键字声明的虚函数，那么在编译器给这个类定义的对象开辟空间时，会默认增加一个指针vptr（虚函数表指针）,并指向该类的虚函数表头,且处在开辟的空间头。调用对象的成员方法时会先遍历vptr指向的表，来查找是否有对应的调用

如果一个类包含有virtual那么类的空间将多四个字节

Vptr初始化过程是分步初始化，在调用子类构造函数时，先触发父类的构造函数，这时vptr指针是指向父类的虚函数表，这时如果在构造函数中调用类的业务(函数)，那么调用的都是父类的成员函数。在父类构造函数调用完成后再触发子类的构造，这时vptr又重新指向子类的虚函数表

### 16.静态联编和动态联编

联编是指一个程序模块、代码之间相互关联的过程

静态联编是程序的匹配、链接在编译阶段实现，也成为早期匹配。**重载函数使用静态联编**

动态联编是指程序联编推迟到程序运行时进行，所以又称为晚期联编(迟绑定)。**Switch语句和if语句是动态联编的例子，多态对于编译器来讲也是联编**

### 17.虚析构函数

前面有了虚继承、虚函数现在又有了虚析构



发生多态后，调用delete，这时编译器只会把变量当作父类来处理，所以只会调用父类的析构，所以要在父类的析构函数前加关键字virtual

实质就是多态

### 18.重载、重写、重定义区别

1.重载一定是发生在同一作用域下

2.重定义发生在两个不同的类中，且一个是父类，一个是子类。其实就是父类中没有声明为虚函数，且在子类中写了一个名字一模一样的函数。

3.重写和重定义一样，不过被重写的是一个虚函数。其实就是声明为虚函数后,写子类中和父类中重名的函数叫做重写

### 19.纯虚函数和抽象类

**纯虚函数**是一个在基类中说明的虚函数，在基类中没有定义，要求任何派生类都定义自己的版本

纯虚函数为派生类提供一个公共界面(接口的封装和设计、软件的模块功能划分)

**语法：**

Virtual 类型 函数名(形参列表) = 0

一个具有纯虚函数的基类称为**抽象类** 抽象类的存在就是为了提供接口，为了被继承

如果一个普通类去继承一个抽象类，那么必须去重写这个纯虚函数,如果不重写，那这个子类还是一个抽象类。

抽象类不允许来创建对象。也就是不能被实例化。但是可以用来定义一个指针来指向子类对象。也就是多态

面向抽象类编程:可以将编写过程分为三层：main()函数的高级业务逻辑层，设计抽象类的抽象层，和实现抽象类的实现层