写在前面：

感谢上一期（戳这里）D老师指正我virtual关键字笔误的问题，回去罚抄一百遍orz。以及X学长的鼓励，我感到非常非常开心！

另外特别感谢本期W小哥哥抽空指导我对升级版代码的理解问题，真的万分感谢！。

----------------------------------------------------------

关于上期浅浅地提的一句虚函数，我看见有一个很有趣的描述，分享给大家

C++ 虚函数的作用：

1、为了方便使用多态特性，我们常常需要在基类中定义虚拟函数。

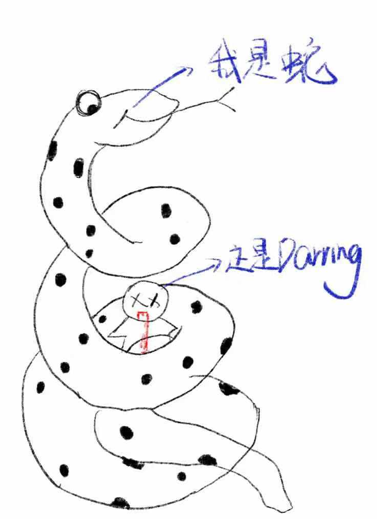
2、在很多情况下，**基类**本身生成对象是不合情理的。例如，动物作为一个基类可以派生出老虎、孔雀等子类，但动物本身生成对象明显不合常理。

为了解决上述问题，引入了**纯虚函数**的概念，将函数定义为纯虚函数（方法：virtual ReturnType Function()= 0;），则编译器要求在派生类中必须予以重写以实现多态性。

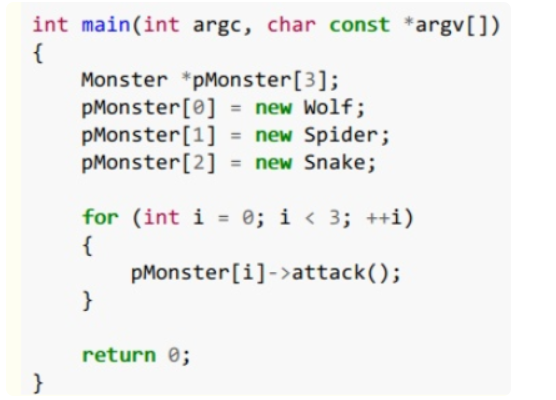
同时含有纯虚拟函数的类称为**抽象类**，它不能生成对象。这样就很好地解决了上述两个问题。

C++中虚函数的用法：

比如你有个游戏，游戏里有个虚基类叫「怪物」，有纯虚函数 「攻击」。然后派生出了三个子类「狼」「蜘蛛」「蟒蛇」，都实现了自己不同的「攻击」函数，比如狼是咬人，蜘蛛是吐丝，蟒蛇把你缠起来。



然后出现好多怪物的时候就可以定义一个 虚基类指针数组，把各种怪物的指针给它，然后迭代循环的时候直接 monster[i]->attack() 攻击玩家就行了，大概见下图：



扩展资料：

使用虚函数的注意事项：

–––––

一、包含虚函数的类指针列表会增大。

二、虚析构函数

三、构造函数不能声明为虚函数

四、不在析构或者构造过程中调用虚函数

以上转自<https://zhidao.baidu.com/question/32424351.html>

1.下面是书p23模拟的一段关于 sizeof 的对话

Q:定义一个空的类型，里面没有任何成员变量和成员函数。对该类型求sizeof,得到结果是什么？

A: 答案是1

Q: 为什么不是0

A: 空类型的实例不包含任何信息，本来求得的sizeof应该是0，但是当我们**声明**该实例时，必须在内存中占有一定空间，否则无法使用这个实例。至于占多少空间，由编译器决定。Visual Studio中，每个空类型的实例占用1字节的空间。

Q: 如果在该类型中添加一个构造函数和析构函数，再求sizeof的结果是多少？

A:还是1。调用构造和析构函数仅需要知道函数地址即可，而这些函数的地址仅与类型相关，与类型的实例无关，编译器也不会因为这两个函数而在实例内添加任何额外的信息。

Q:如果把析构函数标记为虚函数呢？

A：C++编译器一旦发现一个类型中有虚函数，就会为该类型生成虚函数表，并在该类型的每一个实例中添加一个指向虚函数表的指针。

所以：

32位机器，一个指针占用4字节的空间，求得sizeof为4

64位机器，一个指针占用8字节的空间，求得sizeof为8

面试官：你太棒啦，恭喜你被录取啦（误）



2. 赋值运算符函数（高级程序员会另外考虑什么？）

题目： 为该类型添加赋值运算符函数

class CMyString

{

public:

CMyString(char\* pData = nullptr);

CMyString(const CMyString& str);

~CMyString(void);

CMyString& operator = (const CMyString& str);

void Print();

private:

char\* m\_pData;

};

tips: 注意几点

0. 为什么要重载“=”（或者重载的作用是什么）

重载赋值运算符使对象可以直接赋值。

1. 把返回值的类型声明为该类型的引用，并在函数结束时返回实例自身的引用（\*this）。

只有返回一个引用，才可以允许连续赋值。（为什么？见注1）

如果函数返回值是void，则应用该赋值运算符将不能连续赋值。

例如，假设3个CMyString类型的对象str1，str2，str3，则语句str1 = str2 = str3 就不好。

当然，当你确定 不使用连续赋值时，直接返回void也是可以的。

2. 是否把传入的参数类型声明为常量引用，如果传入的参数不是引用而是实例，从形参到实参会调用一次复制构造函数。而将参数声明为引用时，可以避免这样无谓的消耗。、

同时，我在《剑指offer第一期》里面讲过，如果想让赋值运算符函数内不会改变传入参数实例的状态，应该为传入的引用参数加上const关键字。

3. 是否释放实例自身已有的内存。

4. 判断传入的参数和当前实例（\*this）是不是同一个。若是同一个，不进行赋值了，直接返回。

如果不事先判断就赋值，那么释放实例自身内存就会导致严重的问题：

\*this和传入的参数是同一个时，一旦释放自身内存，传入参数的内存也被释放了，再也找不到需要赋值的内容了。

OK 让我们来看看初级程序员和高级程序员考虑的有什么区别？



经典解法，->初级程序员

CMyString& CMyString::operator = (const CMyString& str)

{

if(this == &str)

return \*this;

delete []m\_pData;

m\_pData = nullptr;

m\_pData = new char[strlen(str.m\_pData) + 1];

strcpy(m\_pData, str.m\_pData);

return \*this;

}

考虑到安全性的解法-> 高级程序员必备

上述代码，分配内存之间先用delete 释放了实例m\_pData的内存。

如果此时**内存不足**，new char 会抛出异常。m\_pData 会成为**空指针**，非常容易导致程序崩溃。

即：一旦赋值运算符函数内部抛出一个异常，CMyString 不会保持有效的状态，违背了**异常安全性**（Exception Safety）原则。

下面采用一种方法：先创建一个临时实例，再交换临时实例和原来的实例。

#include <bits/stdc++.h>

using namespace std;

class CMyString {

public:

CMyString(char \*pData = nullptr);

CMyString(const CMyString & str);

~CMyString(void);

CMyString& operator = (const CMyString &str); //重载赋值运算符函数

void Print();

private:

char \*m\_pData;

};

CMyString::CMyString(char \*pData) {

if(pData == nullptr){

m\_pData = new char[1];

m\_pData[0] = '\0';

}

else{

int length = strlen(pData);

m\_pData = new char[length + 1];

strcpy(m\_pData,pData);

}

}

// 拷贝构造函数，传入的引用一定不为NULL,因为在他的构造函数里面对他进行了空间分配

CMyString::CMyString(const CMyString &str) {

int length = strlen(str.m\_pData);

m\_pData = new char[length + 1];

strcpy(m\_pData,str.m\_pData);

}

//析构函数

CMyString::~CMyString() {

delete[] m\_pData;

}

// 初级程序员的做法

// 当空间不足，容易造成异常安全性问题

//CMyString& CMyString::operator=(const CMyString &str) {

// if(this == &str){

// return \*this;

// }

//

// delete[] m\_pData;

// m\_pData = nullptr;

//

// m\_pData = new char[strlen(str.m\_pData)+1];

// strcpy(m\_pData,str.m\_pData);

//

// return \*this;

//

//}

//高级程序员的做法

CMyString& CMyString::operator = (const CMyString &str) {

if(this != &str){

CMyString strTemp(str);

//采用临时变量pTemp为桥梁，交换实例自身的m\_pData 和 strTemp.m\_pData

char\* pTemp = strTemp.m\_pData;

strTemp.m\_pData = m\_pData;

m\_pData = pTemp;

}

return \*this;

}

//========================测试代码==================

void CMyString::Print() {

printf("%s",m\_pData);

}

void Test1(){

printf("Test1 begins:\n");

char \*text = "Hello world";

CMyString str1(text);

CMyString str2;

str2 = str1;

printf("The expected result is : %s.\n",text);

printf("The actual result is :");

str2.Print();

printf(".\n");

}

//赋值给自己

void Test2(){

printf("Test2 begins:\n");

char \*text = "Hello world";

CMyString str1(text);

str1 = str1;

printf("The expected result is : %s.\n",text);

printf("The actual result is :");

str1.Print();

printf(".\n");

}

//连续赋值

void Test3(){

printf("Test3 begins:\n");

char \*text = "Hello world";

CMyString str1(text);

CMyString str2,str3;

str3 = str2 = str1;

printf("The expected result is: %s.\n",text);

printf("The actual result is :");

str2.Print();

printf(".\n");

printf("The expected result is: %s.\n",text);

printf("The actual result is :");

str3.Print();

printf(".\n");

}

int main(int argc,char \*argv[]) {

Test1();

printf("\n");

Test2();

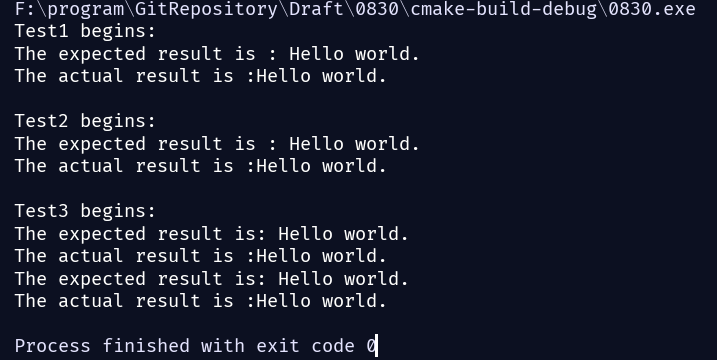
printf("\n");

Test3();

return 0;

}

运行结果：



strTemp是个局部变量，出了if的作用域后，自动调用strTemp的析构函数，将strTemp.m\_pData所指向的内存释放掉。由于strTemp.m\_pData指向的内存就是实例之前的内存，相当于自动调用析构函数释放实例的内存。

C++有六个类的默认函数：构造、析构、拷贝构造、赋值运算符重载。

程序第几行，会首先声明一个临时变量CMyString的对象strTemp，**自动**调用构造函数，构造函数里面肯定会为m\_pData new空间，一旦内存不足new char失败，程序会终止在第几行，自然后面的就不会执行了。自然也就不会影响到

比如有两个对象A和B， 重载赋值运算符之后，A = B； 若

此时内存空间不够抛出诸如bad\_alloc等异常，等号左边的A不受任何影响，我们还没有修改原来的实例的状态。

这就保证了异常安全性。

如果应聘者面试的时候能够考虑到这个层面，面试官就会想。。。。



（误）

注1：为什么只有返回一个引用，才可以允许连续赋值？

事实上，我们的重载运算符返回void、返回对象本身、返回对象引用都是可以的，并不是说一定要返回一个引用，只不过在不同的情况下需要不同的返回值。

那么什么情况下要返回对象的引用呢？

原因有两个：

* 允许进行连续赋值
* 防止返回对象（返回对象也可以进行连续赋值（常规的情况，如a = b = c，而不是（a = b） = c））的时候调用拷贝构造函数和析构函数导致不必要的开销，降低赋值运算符的效率。

对于第二点原因：如果用”值传递“的方式，虽然功能仍然正确，但由于return语句要把\*this拷贝到保存返回值的外部存储单元之中，增加了不必要的开销，会降低赋值函数的效率。

场景：

　　需要返回对象引用或者返回对象（效率没有返回引用高），需要实现连续赋值，使重载的运算符更符合C++本身的运算符语意，如连续赋值 = += -= \*= 、=，<<输出流

　　关于赋值 =，我们知道赋值=有连续等于的特性

1 int x,y,z;

2 x=y=z=15;

同样有趣的是，赋值采用的是右结合律，所以上述连锁赋值被解析为：

1 x=(y=(z=15));//赋值连锁形式

这里15先被赋值给z，然后其结果（更新后的z）再被赋值给y，然后其结果（更新后的y）再被赋值给x。

　　为了实现”连锁赋值“，赋值操作符号返回一个reference(引用)指向操作符号的左侧实参（而事实上重载运算符的左侧实参就是调用对象本身，比如= += -=等），这是你为classes实现赋值操作符时应该遵循的协议：这个协议不仅仅适用于以上的标准赋值形式，也适用于所有赋值运算。

注意，这只是个协议，并无强制性，如果不遵循它，代码一样可以通过编译，然而这份协议被所有内置类型和标准程序库提供的类型入string，vector，complex，std::trl::shared\_ptr或者即将提供的类型共同遵守。因此除非你有一个标新立异的好理由，不然还是随众吧。

对于连续赋值，常规的情况（a = b = c）

1. 返回对象的情况：当运算符重载返回的是对象时，会在连续赋值运算过程的返回途中，调用两次拷贝构造函数和析构函数（因为return的是个新的对象）

2. 如果采用String& operator = (const String& str)这样就不会有多余的调用（因为这里直接return一个已经存在的对象的引用）

3. 另外，当你确定不使用连续赋值时，直接返回void也是可以的。

要明白一点：

　　运算符左侧的对象就是操作对象，比如

1 ObjectA = ObjectB 等同于ObjectA.operator=（ObjectB）

2 ObjectA+=ObjectB 等同于ObjectA.operator+（ObjectB）

总结

并非必须返回引用，返回引用的好处既可以避免拷贝构造函数和析构函数的调用，又可以保证= +=等运算符的原始语义清晰。

　所以，对此类运算符重载时，还是老老实实的返回引用，少搞事，做个好guy ：）

以上参考<https://www.cnblogs.com/codingmengmeng/p/5871254.html>



好了今天的分享时间就到这里了，欢迎批评欢迎建议。嘻嘻~