计算机学院 高级程序语言设计 课程实验报告

实验题目：多态（二） 动态_运行时多态		学号：202300130150
日期：2024.4.23	班级： 4	姓名： 王成意

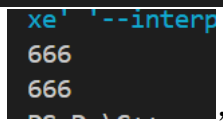
实验目的：
理解并掌握动态_运行时多态。

实验步骤与内容：
1. 编写一个哺乳动物类 Mammal，再由此派生出狗类 Dog，二者都声明 speak()成员函数，该函数在基类中被声明为虚函数，声明一个 Dog 类的对象，通过此对象调用 speak 函数，观察运行结果。（习题 8.5）

代码：

```
#include <cstdio>

using namespace std;

class mammal

{

public:

    mammal() = default;

    virtual void speak()

    {

        printf("hahaha\n");

    }

};

class dog : public mammal

{

public:
```

```
    dog() = default;
    void speak()
    {
        printf("666\n");
    }
};
int main()
{
    dog x;
    mammal *y = new dog();
    x.speak();
    y->speak();
    return 0;
```

结果： 

，由此可派生类对象调用的 speak 函数是其在派生类内重新声明的函数。

2. 请编写一个抽象类 Shape，在此基础上派生出类 Rectangle 和 Circle，二者都有计算对象面积的函数 getArea()、计算对象周长的函数 getPerim()。注意不同的派生类可能需要额外的成员数据。（Shape 的析构函数之前应该加 virtual）（习题 8.6）

代码：

```
#include <cstdio>
using namespace std;
class shape
{
```

```
public:
    virtual float getArea() = 0;
    virtual float getPerim() = 0;
    virtual void show() = 0;
    virtual ~shape();
};
shape::~shape()
{
    printf("deleting shape!\n");
}
class rectangle : public shape
{
private:
    float x, y;

public:
    rectangle(float x, float y) : x(x), y(y) {}
    float getArea()
    {
        return x * y;
    }
    float getPerim()
```

```
    {
        return ((x + y) * 2);
    }
    virtual void show()
    {
        printf("%f %f\n", this->getArea(),
this->getPerim());
    }
    ~rectangle();
};
rectangle::~rectangle()
{
    printf("deleting rectangle!\n");
}
const float pi = 3.1416;
class circle : public shape
{
private:
    float r;

public:
    circle(float r) : r(r) {}
```

```
    float getArea()
    {
        return pi * r * r;
    }
    float getPerim()
    {
        return 2 * pi * r;
    }
    virtual void show()
    {
        printf("%f %f\n", this->getArea(),
this->getPerim());
    }
    ~circle();
};
circle::~circle()
{
    printf("deleting circle!\n");
}
int main()
{
    rectangle x(3, 4);
```

```
    circle y(2);
    shape *a = &x;
    shape *b = &y;
    a->show(), b->show();
    return 0;
}
```

结果：

```
xe' '--interpreter=mi'
12.000000 14.000000
12.566400 12.566400
deleting circle!
deleting shape!
deleting rectangle!
deleting shape!
```

由此可见，shape 的虚析构函数实现了与其派生类的动态绑定，也就是调用派生类的析构函数时，会调用基类的虚构函数。

3. 定义一个基类 BaseClass，从它派生出类 DerivedClass。BaseClass 有成员函数 fnl()、fn2()打印该函数调用信息，fn1()是虚函数；DerivedClass 也有成员函数 fnl()、fn2()。在主函数中声明一个 DerivedClass 的对象，分别用 BaseClass 和 DerivedClass 的指针指向 DerivedClass 的对象，并通过指针调用 fn1()、fn2()，观察运行结果。父指子（父类指针指子类对象）实现动态多态练习。（习题 8.8）

代码：

```
#include<cstdio>

using namespace std;

class BaseClass

{

private:

    /* data */
```

```
public:
    BaseClass(/* args */);
    ~BaseClass();
    virtual void fn1()
    {
        printf("base fn1\n");
    }
    void fn2()
    {
        printf("base fn2\n");
    }
};

BaseClass::BaseClass(/* args */)
{
}

BaseClass::~BaseClass()
{
}

class DerivedClass:public BaseClass
```

```
{
private:
    /* data */
public:
    DerivedClass(/* args */);
    ~DerivedClass();
    void fn1()
    {
        printf("derived fn1\n");
    }
    void fn2()
    {
        printf("derived fn2\n");
    }
};

DerivedClass::DerivedClass(/* args */)
{
}

DerivedClass::~DerivedClass()
{
```

```
}


int main()
{
    DerivedClass a;
    BaseClass *b = &a;
    DerivedClass *c = &a;
    b->fn1(), b->fn2();
    c->fn1(), c->fn2();
    return 0;
}
```

结果：

```
interpreter=mi'
derived fn1
base fn2
derived fn1
derived fn2
```

，其中：

```
BaseClass *b = &a;
```

这一语句实现了父指子（父类指针指子类对象）实现动态多态，其 fn1 调用的实际是派生类中覆盖的 fn1 函数（第一行），而 fn2 不是虚函数，故调用的是基类中声明的函数（第二行）。

4. 分析第 8 章 PPT 例 8-5，非多态类型，无虚析构函数，造成内存泄漏的例子，分析出错原因？

结果：

```
xe' '--interpreter=mi'
Base destructor
PS D:\C++ programs>
```

，也就是只进行了基类析构，没有进行派生类析构。

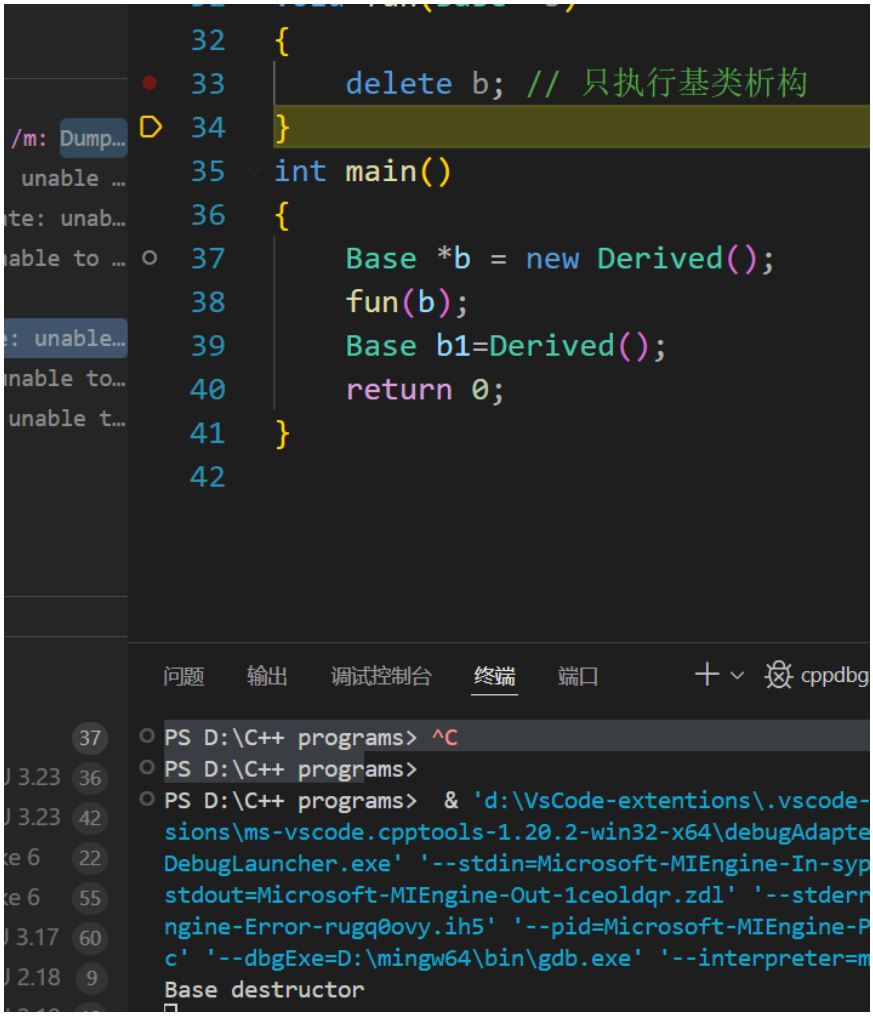
原因在于执行 fun 时，使用基类指针指向派生类对象，析构该派生类对象时，只会调用基类析构函数，不会调用派生类的析构函数。

（a） 在 main()中增加语句 Base b1=Derived(); 请分析输出结果。

结果：

```
xe' '--interpreter=mi'
Base destructor
Derived destructor
Base destructor
Base destructor
```

第一行中的 base destructor 是执行 fun（b）时，delete b 产生的：



由于

```
Base b1=Derived();
```

是基类对象代表派生类对象，但只能使用从基类继承来的成员需要析构掉派生类对象保证不会内存泄漏；其中析构函数执行是派生类函数的析构，这样有了 2，3 行：

```
Derived destructor
Base destructor
```

（动态绑定）

然后最后 return 0 时析构 b1，即第四行的 base destructor。

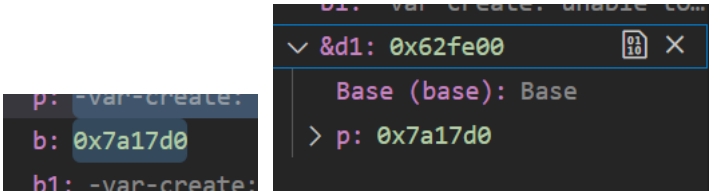
（b）在 main()中增加语句 Derived d1; fun(&d1); 程序会出问题吗？

会，在这里：

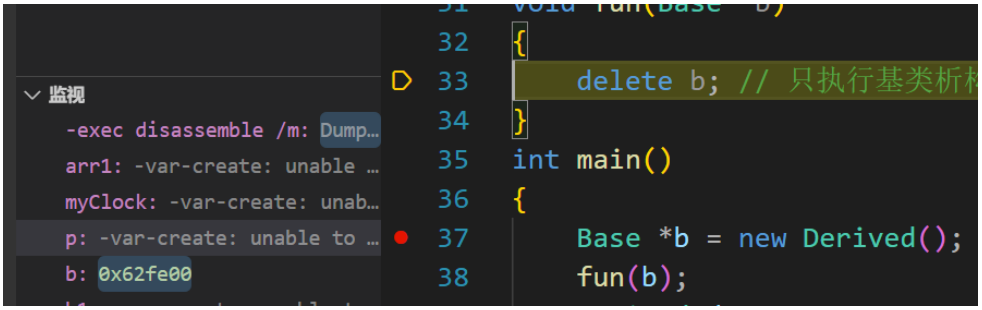
```
30  }
31  void fun(Base *b)
32  {
33      delete b; // 只执行基类析构
34  }
35  int main()
36  {
37      Base *b = new Derived();
38      fun(b);
39      Derived d1;
40      fun(&d1);
41      return 0;
42  }
43  //b 0xfc17d0 p 0xfc1810
```

```
问题  输出  调试控制台  终端  端口
PS D:\C++ programs> ^C
PS D:\C++ programs>  & 'd:\VsCode-extentions\.vsc
sions\ms-vscode.cpptools-1.20.2-win32-x64\debugAd
DebugLauncher.exe' '--stdin=Microsoft-MIEngine-In
stdout=Microsoft-MIEngine-Out-vika5ftx.22o' '--st
ngine-Error-hjrzzxm2.4f5' '--pid=Microsoft-MIEngi
v' '--dbgExe=D:\mingw64\bin\gdb.exe' '--interpret
Base destructor
Base destructor
```

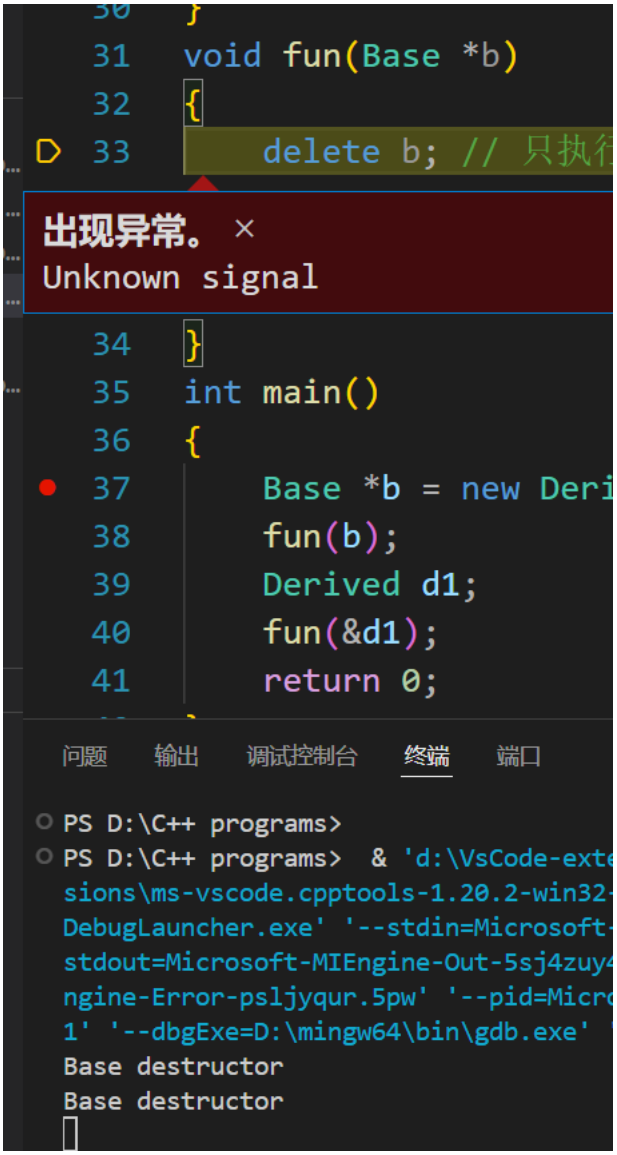
问题在于：



，d1 中 p 和 b 的地址相同。

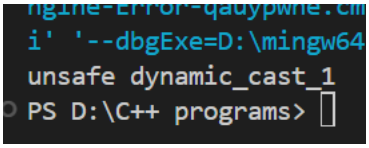


，也就是执行 fun（&d1）时，把 derived 型指针当成 base 型，因为 d1 中 p 和 b 的地址相同导致 delete 同一处地址，uke。



5. 阅读学习 cast 中几种不同的 cast 类型和用法，理解并分析其中 p3.cpp 的运行结果。

结果：



分析：

运行警告里也有提示：

```
    Derived * pd;
dynamic_cast of 'Base b' to 'class Derived*' can never succeed gcc
Base b
查看问题 (Alt+F8)    没有可用的快速修复
    pd = dynamic_cast<Derived*> ( &b);
    if( pd == NULL)
        cout << "unsafe dynamic_cast_1" << enc
```

dynamic_cast 此时执行的是指针类型的转换，会返回空指针。

6. 第 8 章 PPT，P69 例 8-9 dynamic_cast 练习，并做如下修改，分析运行结果。

（a）给 void fun(Base *b) 函数中的 if 语言增加如下语句，

```
else
cout<<b<<"  "<<d<<endl;
```

改前&改后：

```
ngine-Error-223vygmb.lq4' '--pid=
y' '--dbgExe=D:\mingw64\bin\gdb.e
Base::fun1()
Derived1::fun1()
Derived1::fun2()
Derived2::fun1()
Derived2::fun2()
PS D:\C++ programs> ^C
PS D:\C++ programs>
PS D:\C++ programs>  & 'd:\VsCode
sions\ms-vscode.cpptools-1.20.2-w
DebugLauncher.exe' '--stdin=Micro
stdout=Microsoft-MIEngine-Out-oys
ngine-Error-nunmdwfo.tw3' '--pid=
e' '--dbgExe=D:\mingw64\bin\gdb.e
Base::fun1()
0x62fe08  0
Derived1::fun1()
Derived1::fun2()
Derived2::fun1()
Derived2::fun2()
PS D:\C++ programs>
```

问　　题　　出　　现　　在

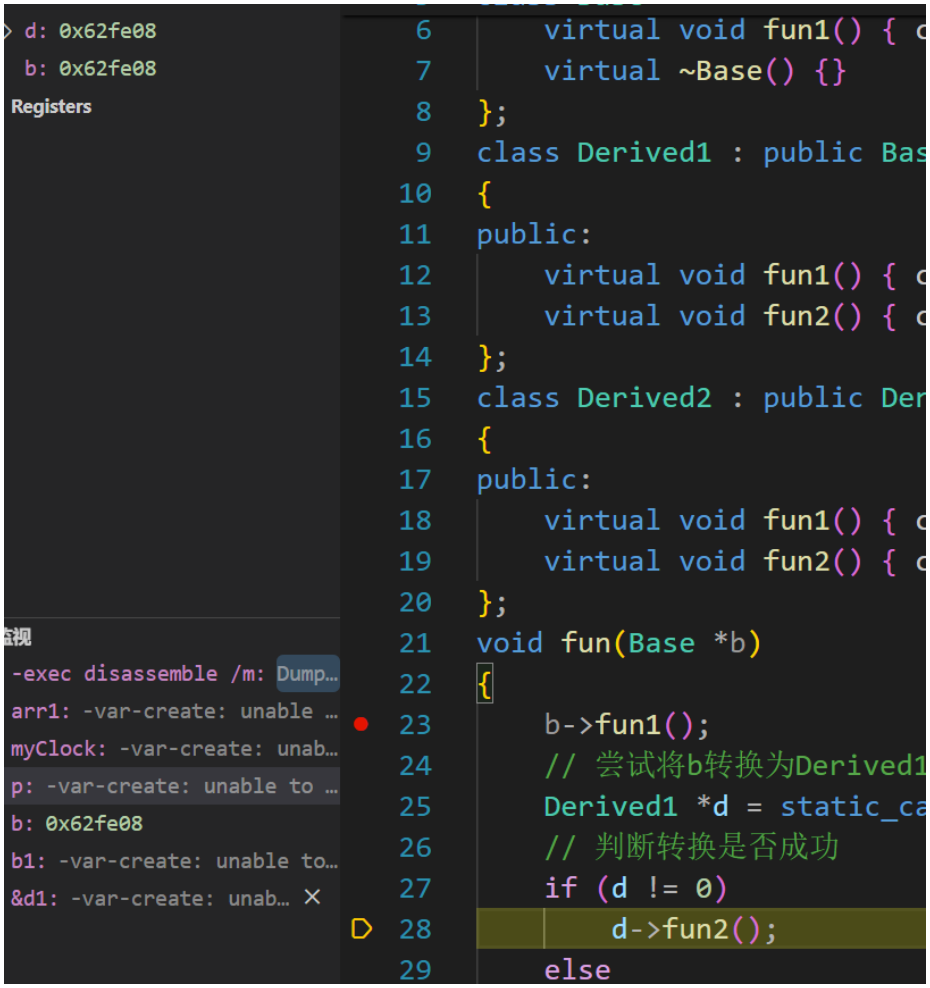
```
void fun(Base *b)
{
    b->fun1();
    // 尝试将b转换为Derived1指针
    Derived1 *d = dynamic_cast<Derived1 *>(b);
    // 判断转换是否成功
```

，此时进行了指针类型的转换，返回了空指针（d=0）。
而执行

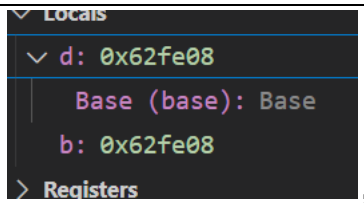
```
    fun(&d1);
```

语句时，&d1 本来就是 derived 型，可以使用 dynamic_cast.

（b） 试将其中的 dynamic_cast 改为 static_cast：



可是此时问题在于 d 是 base 型的，



，没有 fun2，错误，程序卡死。

结论分析与体会：
非常，非常好的实验，使我对于 dynamic_cast 有了深入的了解。