目录

[C++类和对象 1](#_Toc37883710)

[继承 1](#_Toc37883711)

[多继承 2](#_Toc37883712)

[接口（抽象类） 5](#_Toc37883713)

[饿汉模式 6](#_Toc37883714)

## C++类和对象

### 继承

#include <iostream>

class Shape {

public:

void SetWidth(int w) {

width = w;

}

void SetHeight(int h) {

height = h;

}

//变量继承来的，不能设置私有

protected:

int width;

int height;

};

class Rectangle : public Shape {

public:

int GetArea(){

return (width \* height);

}

};

using namespace std;

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

{

Rectangle Rect;

Rect.SetWidth(5);

Rect.SetHeight(7);

cout<<"Total area:"<<Rect.GetArea()<<endl;

return 0;

}

### 多继承

#include <iostream>

using namespace std;

// 基类 Shape

class Shape

{

public:

void setWidth(int w)

{

width = w;

}

void setHeight(int h)

{

height = h;

}

protected:

int width;

int height;

};

// 基类 PaintCost

class PaintCost

{

public:

int getCost(int area)

{

return area \* 70;

}

};

// 派生类

class Rectangle: public Shape, public PaintCost

{

public:

int getArea()

{

return (width \* height);

}

};

int main(void)

{

Rectangle Rect;

int area;

Rect.setWidth(5);

Rect.setHeight(7);

area = Rect.getArea();

// 输出对象的面积

cout << "Total area: " << Rect.getArea() << endl;

// 输出总花费

cout << "Total paint cost: $" << Rect.getCost(area) << endl;

return 0;

}

//==================================================

//多态

/\*

导致错误输出的原因是，调用函数 area() 被编译器设置为基类中的版本，

这就是所谓的静态多态，或静态链接 - 函数调用在程序执行前就准备好了。

有时候这也被称为早绑定，因为 area() 函数在程序编译期间就已经设置好了。

但现在，让我们对程序稍作修改，在 Shape 类中，area() 的声明前放置关键字 virtual，

如下所示：此时，编译器看的是指针的内容，而不是它的类型。

因此，由于 tri 和 rec 类的对象的地址存储在 \*shape 中，所以会调用各自的 area() 函数。

正如您所看到的，每个子类都有一个函数 area() 的独立实现。

这就是多态的一般使用方式。有了多态，您可以有多个不同的类，

都带有同一个名称但具有不同实现的函数，函数的参数甚至可以是相同的。\*/

#include <iostream>

using namespace std;

class Shape {

public:

Shape(int a=0,int b=0) {

width = a;

height = b;

}

//int area() {

virtual int *area*() {

cout<<"Parent class area :"<<endl;

return 0;

}

protected:

int width;

int height;

};

class Rectangle : public Shape {

public:

Rectangle( int a=0,int b=0):Shape(a,b){}

int *area*() {

cout<<"Rectangle class area :"<<endl;

return (width \* height);

}

};

class Triange : public Shape {

public:

Triange(int a, int b):Shape(a,b){}

int *area*() {

cout<<"Triange class area :"<<endl;

return(width \* height);

}

};

int main(){

Shape \*shape;

Rectangle rec(10,7);

Triange tri(10,5);

//存储矩形的地址

shape = &rec;

//调用矩形的求面积函数area

shape->*area*();

//存储三角形的地址

shape = &tri;

// 调用三角形的求面积函数 area

int i = shape->*area*();

cout<<i<<endl;

return 0;

}

/\*

Rectangle class area :

Triange class area :

50

\*/

//==========================================================

### 接口（抽象类）

/\*

C++ 接口是使用抽象类来实现的

设计抽象类（通常称为 ABC）的目的，是为了给其他类提供一个可以继承的适当的基类。

抽象类不能被用于实例化对象，它只能作为接口使用。如果试图实例化一个抽象类的对象，会导致编译错误。

\*/

#include <iostream>

using namespace std;

class Shape {

public:

virtual int *GetArea*()=0;

void SetWidth(int w) {

width = w;

}

void SetHeight(int h) {

height = h;

}

protected:

int width;

int height;

};

//派生类

class Rectangle:public Shape {

public:

int *GetArea*(){

return width\*height;

}

};

class Triangle : public Shape {

public:

int *GetArea*(){

return width\*height/2;

}

};

int main(){

Rectangle Rect;

Triangle Tri;

Rect.SetHeight(5);

Rect.SetWidth(7);

cout << "Total Rectangle area: " <<Rect.*GetArea*()<<endl;

Tri.SetWidth(5);

Tri.SetHeight(7);

// 输出对象的面积

cout << "Total Triangle area: " << Tri.*GetArea*() << endl;

return 0;

}

/\*

Total Rectangle area35

Total Triangle area: 17

\*/

### 饿汉模式

#include <iostream>

using namespace std;

class Single {

public:

static Single\* GetInstance() {//使用静态函数时候需要用到作用域

if(instance == NULL) {

instance = new Single();

}

return instance;

}

private:

Single(){};//构造函数这里的构造函数要有括号

static Single \*instance;//在类里面初始化

};

Single\* Single::instance = NULL;//在类外面赋值

int main() {

Single \*p1 = Single::GetInstance();

cout << "p1"<<p1<<endl;

Single \*p2 = Single::GetInstance();

cout << "p2"<<p2<<endl;

Single \*p3 = Single::GetInstance();

cout << "p3"<<p3<<endl;

}

//========================================================

工厂模式