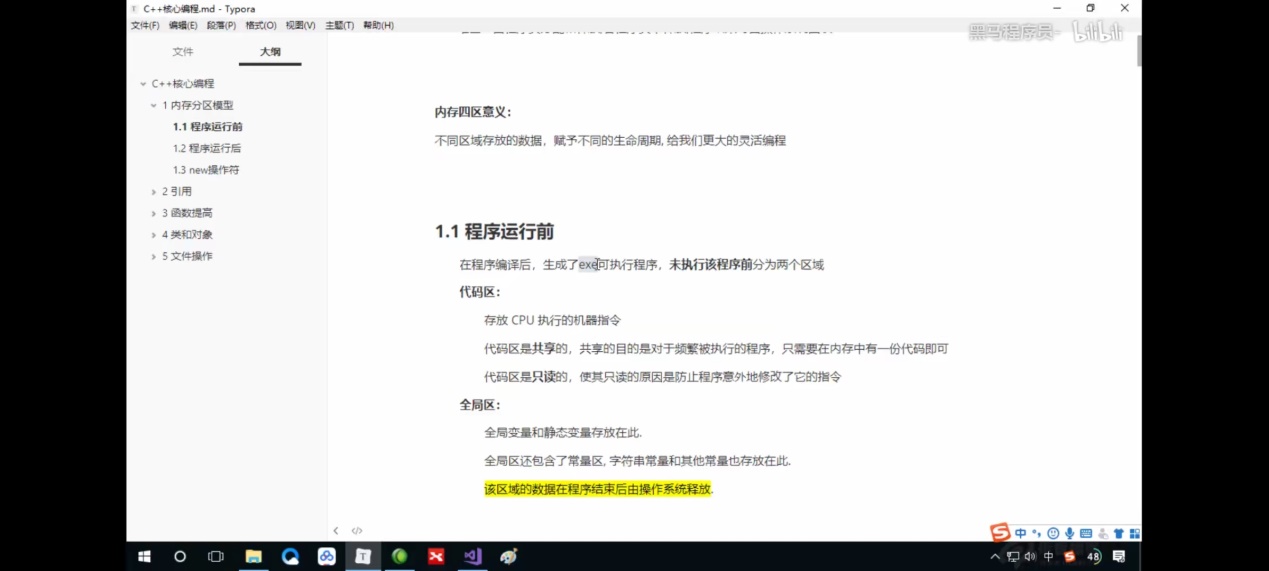
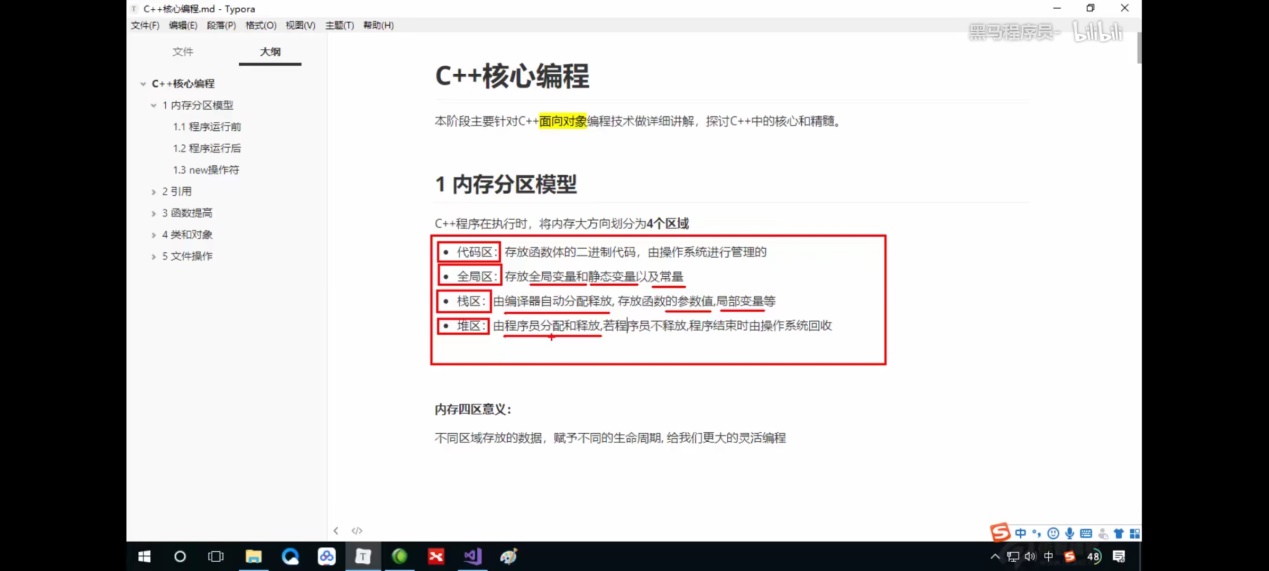
**C++核心编程**



#include<iostream>

#include<string>

using namespace std;

//创建全局变量

int g\_a = 10;

int g\_b = 10;

//const修饰的全局变量

const int c\_g\_a = 10; const int c\_g\_b = 10;

int main()

{

//全局区

//全局变量，静态变量，常量

// 创建普通局部变量

int a = 10;

int b = 10;

cout << "局部变量a的地址： " << (int)&a << endl;

cout << "局部变量b的地址： " << (int)&b << endl;

cout << "全局变量g\_a的地址： " << (int)&g\_a<< endl;

cout << "全局变量g\_b的地址： " << (int)&g\_b << endl;

static int s\_a = 10;

static int s\_b = 10;

cout << "静态变量s\_a的地址： " << (int)&s\_a << endl;

cout << "静态变量s\_b的地址： " << (int)&s\_b << endl;

//常量

//字符串常量

cout << "字符串常量的地址： " << (int)&"hello world" << endl;

//const修饰的变量

// const修饰的全局变量，const修饰的局部变量

cout << "全局常量c\_g\_a的地址的地址： " << (int)&c\_g\_a << endl;

cout << "全局常量c\_g\_b的地址的地址： " << (int)&c\_g\_b << endl;

int c\_l\_a = 10;

int c\_l\_b = 10;

cout << "局部常量c\_l\_b的地址的地址： " << (int)&c\_l\_b << endl;

cout << "全局常量c\_l\_b的地址的地址： " << (int)&c\_l\_b << endl;

system("pause");

return 0;

}

//栈区和数据的注意事项----不要返回局部变量的地址

//栈区的数据由编译器管理开辟和施放

int \*func() //形参数据也会放在栈区

{

int a = 10;//局部变量 存放在栈区，栈区的数据在函数执行完后自动释放

return &a;//返回局部变量的地址

}

int main() {

//接受func函数的返回值

int \*p = func();

cout << \*p << endl;//第一次可以正确打印数字，因为编译器做了保留

cout << \*p << endl;//第二次这个数据就不再保留了

system("pause");

return 0;

}

//&&&&&&&&&&&&& 堆区 &&&&&&&&&&&

int \*func() {

//利用关键字 可以将数据开辟到堆区

int \*p = new int(10);

return p;

}

int main() {

//在堆区开辟数据

int \*p = func();

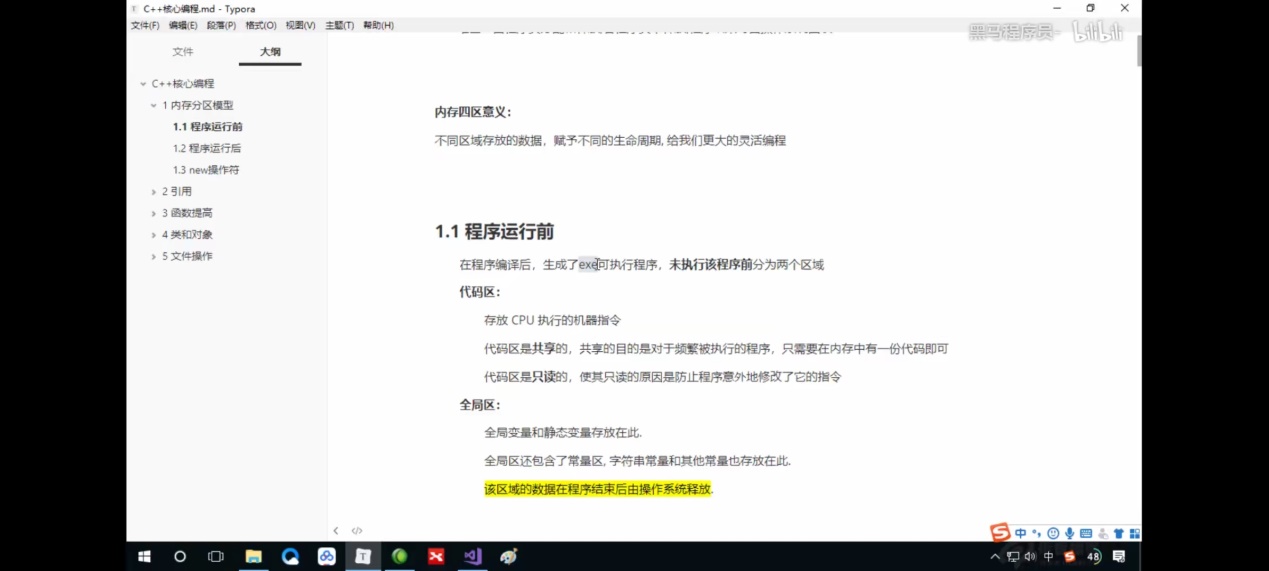
cout << \*p << endl;

system("pause");

return 0;

}

//&&&&&&&&&& new操作符 &&&&&&&&&&&



//&&&&&&&&&& new操作符 &&&&&&&&&&&

//1.new基本语法

int \*func() {

//在堆区创建整型数据

//new返回是 该数据的指针

int \*p = new int(10);

return p;

}

void test01() {

int \*p = func();

cout << \*p << endl;

cout << \*p << endl;

cout << \*p << endl;

//堆区的数据 由程序员管理开辟，程序员管理释放

//如果想释放堆区的数据，利用关键字“delete”

delete p;

}

//2.在堆区利用new开辟数组

void test02()

{

int \*a = new int[10];//10代表数组有10个元素

for (int i = 0; i < 10; i++)

{

a[i] = i + 100;//赋值

}

for (int i = 0; i < 10; i++)

{

cout<<a[i] <<endl;//打印

}

//释放堆区数组

//释放堆区数组时 要加[]才可以

delete [] a;

}

int main() {

test02();

system("pause");

return 0;

}

//\*\*\*\*\*\*\*\*\* 引用 \*\*\*\*\*\*\*\*\*

int main() {

//引用基本语法

//数据类型 &别名 = 原名；

int a = 10;

int &b = a;

cout << "a= " << a << endl;

cout << "b= " << b << endl;

b = 100;

cout << "a= " << a << endl;

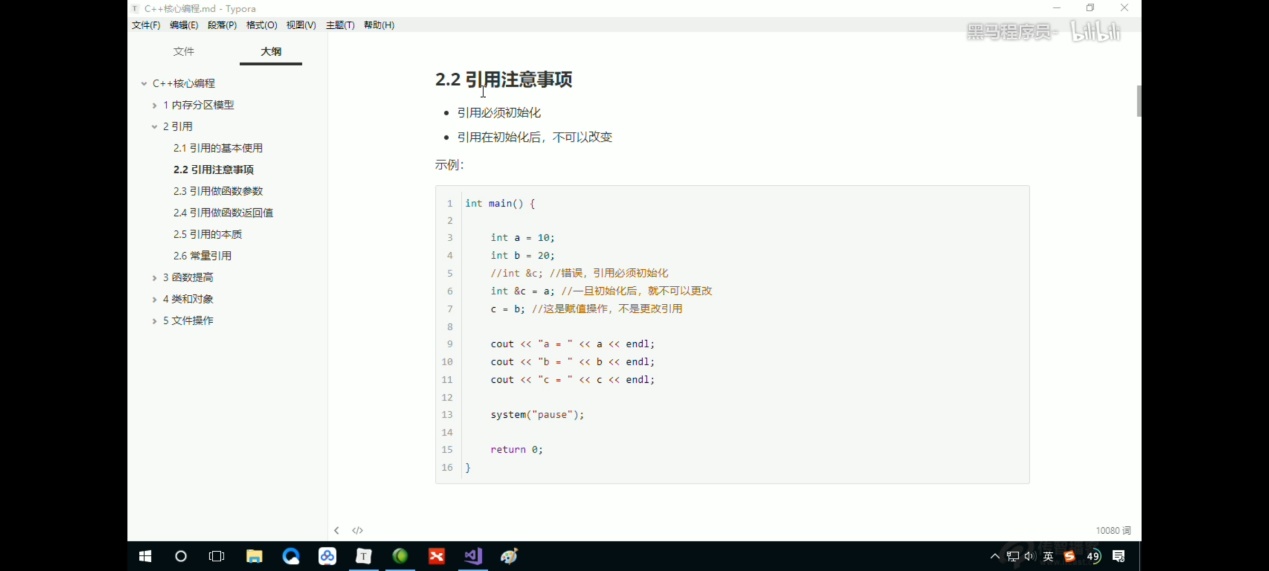
cout << "b= " << b << endl;

system("pause");

return 0;

}

//\*\*\*\*\*\*\*\*\*\*\*\* 引用的注意事项 \*\*\*\*\*\*\*\*\*\*



int main() {

int a = 10;

//1.引用必须初始化

//int &b；//错误，引用必须初始化

int &b = a;

//2.引用在初始化后，不可以改变

int c = 20;

b = c;//赋值操作，而不是更改操作

system("pause");

return 0;

}

//引用做函数参数

//交换函数

void mySwap01(int a, int b) {

int t = a; a = b; b = t;

}

//地址传递

void mySwap02(int \*a, int \*b) {

int t = \*a; \*a = \*b; \*b = t;

}

//引用传递

void mySwap03(int &a, int &b) {

int t = a; a = b; b = t;

}

int main() {

int a = 10; int b = 20;

/\*mySwap01(a, b);//值传递，形参不会修饰实参

cout << "01a=" << a << endl;

cout << "01b=" << b << endl;\*/

/\*mySwap02(&a, &b);//地址传递，形参会修饰实参的

cout << "02a=" << a << endl;

cout << "02b=" << b << endl;\*/

mySwap03(a, b);//引用传递，形参会修饰实参的

cout << "03a=" << a << endl;

cout << "03b=" << b << endl;

system("pause");

return 0;

}

//\*\*\*\*\*\*\*\*\*\* 引用做函数的返回值 \*\*\*\*\*\*\*\*\*\*



//1.不要返回局部变量的引用

int& test01() {

int a = 10; //局部变量存放在四区中的（ 栈区 ）

return a;

}

//2.函数的调用可以作为左值

int& test02() {

static int a = 10;//静态变量 存放在全局区，全局区上的数据在程序结束后系统释放

return a;

}

int main() {

int &ref = test01();

cout << "ref1= " << ref << endl;// 第一次可以正确打印数字，因为编译器做了保留

cout << "ref1= " << ref << endl;//第二次这个数据就不再保留了

int &ref2 = test02();

cout << "ref2= " << ref2 << endl;

cout << "ref2= " << ref2 << endl;

test02() = 1000;//如果函数的返回值是引用，这个函数调用可以作为左值

cout << "ref2= " << ref2 << endl;

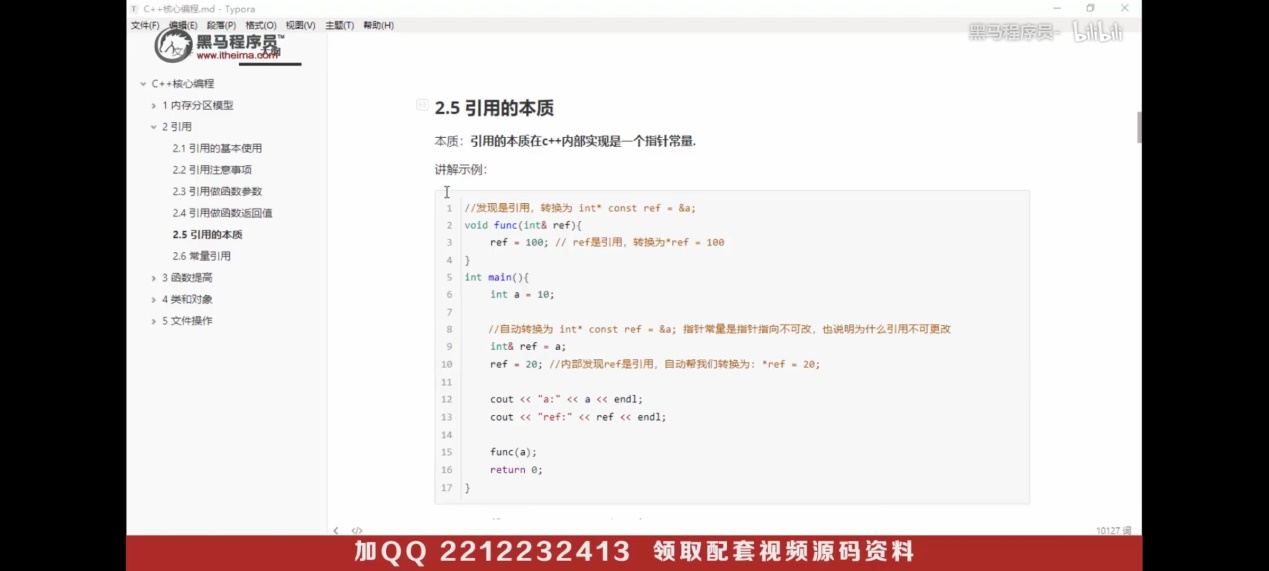
cout << "ref2= " << ref2 << endl;

system("pause");

return 0;

}

\*\*\*\*\*\*\*\*\*\*\*\* 引用的本质 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*



//&&&&&&&&&& 常量引用 &&&&&&&&&&

//打印

void showValue(const int &val) {

// val = 1000;会报错

cout << "val= " << val << endl;

}

int main() {

//常量引用

//使用场景： 用来修饰实参，防止误操作

//int a=10;

//加上const之后，编译器修改 int temp=10;const int & ref = temp;

//const int & ref = 10; //引用必须引一块合法的内存空间

//ref = 20; //加入const之后变为只读，不可以修改

int a = 100;

showValue(a);

cout << "val= " << a << endl;

system("pause");

return 0;

}

//函数默认参数

//如果我们自己传入数据，就用自己的数据，如果没有，就用默认值

//语法： 返回值类型 函数名（形参 = 默认值）

int func(int a, int b = 10, int c = 30) {

return a + b + c;

}

//注意事项

//1.如果某个位置已经有了默认参数，那么从这个位置往后，从左到右都必须有默认值

/\*int func(int a, int b = 10, int c = 30,int d)

{

return a + b + c;

}\*///报错

//2.如果函数的声明有默认参数，函数实现就不能有默认参数

//声明和实现只能有一个有默认参数

//int func2(int a=10, int b=20);// 报错

int func2(int a, int b);

int func2(int a=10, int b=10 ) {

return a + b;

}

int main() {

cout << func(10) << endl;

cout << func2(10) << endl;

system("pause");

return 0;

}

// 占位参数

//语法： 返回值类型 函数名（数据类型）

// 占位参数 可以有默认值

void func(int a, int) {

cout << "this is func " << endl;

}

int main() {

func(10,10);//占位参数必须填补

system("pause");

return 0;

}

// 函数重载



// 函数重载

//函数重载的满足条件

//1.同一个作用域下

//2.函数名称相同

//3.函数参数类型不同，或者个数不同，或者顺序不同

void func1() {

cout << "func的调用" << endl;

}

void func2(int a) {

cout << "func(int a)的调用!" << endl;

}

void func3(double a) {

cout << "func(double a)的调用!" << endl;

}

void func4(int a, double b) {

cout << "func(int a;double b)的调用!" << endl;

}

void func5(double a, int b) {

cout << "func(double a, int b)的调用!" << endl;

}

//注意事项

//函数的返回值不可做为函数重载的条件

//void func5(double a, int b) {

// cout << "func(double a, int b)的调用!" << endl;

//}

int main() {

func1();

func2(10);

func3(10);

func4(10, 3.14);

func5(3.14, 10);

system("pause");

return 0;

}

// 函数重载

//1.引用作为重载的条件

void func1(int &a) {// int &a = 10; 不合法

cout << "func1(int &a)调用" << endl;

}

void func2(const int &a) { //const int &a = 10；合法

cout << "func2(const int &a)调用" << endl;

}

//2.函数重载遇到默认参数

void func3(int a,int b) {

cout << "func3(int a，int b)调用" << endl;

}

void func4(int a) {

cout << "func4(int a)调用" << endl;

}

int main() {

int a = 10;

func1(a);

func2(10);

//func3(10);//当函数重载遇到默认参数，出现二义性，报错，尽量避免这种情况出现

func3(10,20);

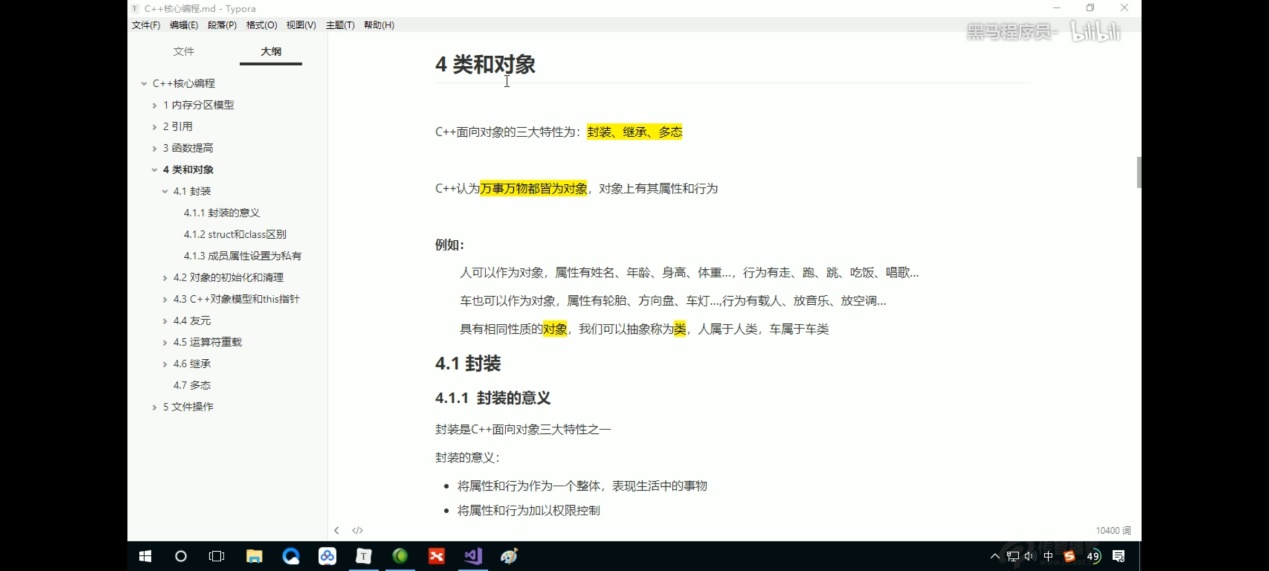
func4(10);

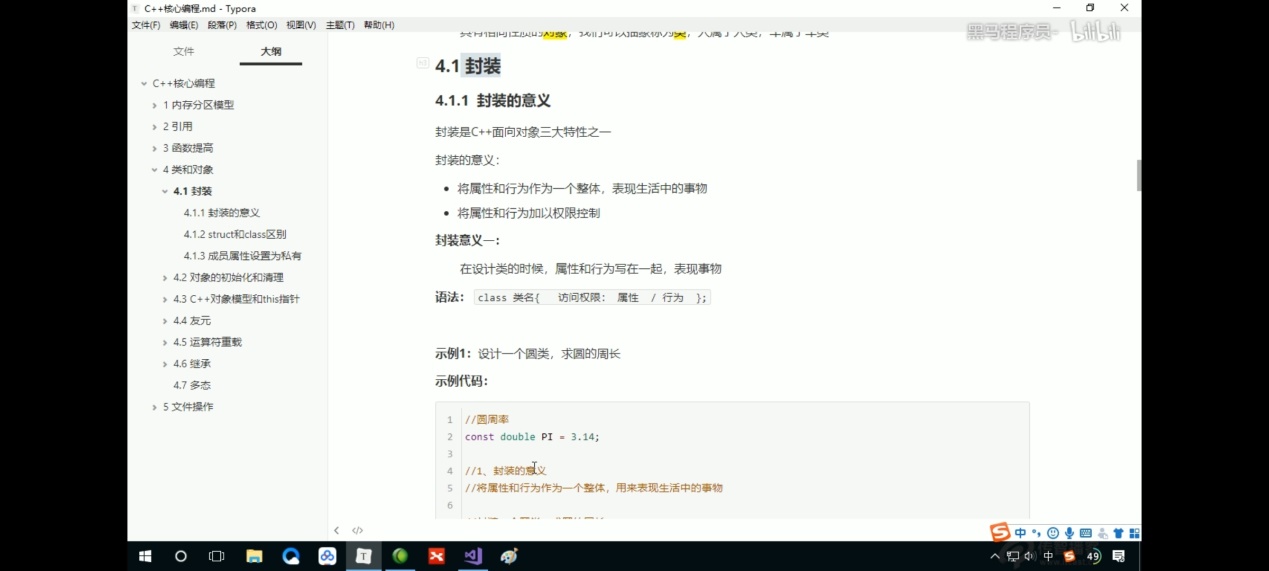
system("pause");

return 0;

}

**类和对象**

****

****

// 类和对象

//设计一个圆类

//圆求圆的周长公式： 2\*PI\*半径

const double PI = 3.14;

//class代表一个类，类后面紧跟的就是类名称

class Circle

{

//访问权限

//公告权限

public:

//属性

//半径

int m\_r;

//行为

//获取圆的周长

double calculateZC() {

return 2 \* PI\*m\_r;

}

};

int main() {

//通过圆类 创建具体的圆（对象）

//实例化 （通过一个类 创建一个对象的过程）

Circle cl;

//给圆对象 的属性进行赋值

cl.m\_r = 10;

//2\*PI\*10;

cout << "圆的周长： " << cl.calculateZC() << endl;

system("pause");

return 0;

}

//设计一个学生类，属性有学生姓名和学号

//可以给学生姓名和学号赋值，可以显示学生的姓名和学号

//设计学生类

class Student

{

public://公共权限

//{类中的属性和行为 我们统称为 成员}

//{属性 成员属性 成员变量}

//{行为 成员函数 成语方法}

//属性

string m\_Name;//姓名

int m\_Id;//学号

//行为

//显示姓名和学号

void showStudent() {

cout << "学生姓名： " << m\_Name << endl;

cout << "学生学号： " << m\_Id << endl;

cout << endl;

}

//2.

//给姓名赋值

void setName(string name) {

m\_Name = name;

}

//给学号赋值

void setId(int Id) {

m\_Id = Id;

}

};

int main() {

//1.

//创建一个具体的学生

//实例化1

Student s1;

//给s1对象赋值

s1.m\_Name = "张三";

s1.m\_Id = 250;

//显示学生信息

s1.showStudent();

Student s2;

//给s2对象赋值

s2.m\_Name = "王五";

s2.m\_Id = 520;

//显示学生信息

s2.showStudent();

//2.

//实例化2

Student s3;

s3.setName("李四");

s3.setId(222);

s3.showStudent();

system("pause");

return 0;

}

封装

封装的意义

****

//访问权限

//三种

//公共权限 public 成员 类内可以访问 类外可以访问

//保护权限 protected 成员 类内可以访问 类外不可以访问 儿子也可以访问父亲的保护内存

//私有权限 private 成员 类内可以访问 类外不可以访问 儿子不可以访问父亲的保护内存

class Person

{

public:

//公共权限

string m\_Name;

protected:

//保护权限

string m\_Car;

private:

//私有权限

int m\_Password;

public:

void func() {

m\_Name = "张三";

m\_Car = "拖拉机";

m\_Password = 123456;

}//类内

};

int main() {

//实例化具体对象

Person p1;

p1.m\_Name = "李四";

//p1.m\_Car = "奔驰";//保护权限内容，在类外访问不到

//p1.m\_Password = 333456;//私有权限内容，在类外访问不到

//类外

system("pause");

return 0;

}

//class和struct的区别

**\**

class C1

{

int m\_A;//默认权限 是私有

};

struct C2

{

int m\_A;//默认权限 是公共

};

int main() {

//stuct 和 class 区别

//struct 默认权限是 公共 publi

//class 默认权限是 私有 private

C1 c1;

//c1.m\_A = 100;//在class里默认权限 私有，因此类外不可访问

C2 c2;

c2.m\_A = 100;//在struct里默认权限 公共，因此类外可以访问

system("pause");

return 0;

}

// 成员属性设置为私有

//1.可以自己控制读写权限

//2.对于写，可以检测数据的有效性

//设计人类

class Person

{

public:

//设置姓名

void setName(string name)

{

m\_Name = name;

}

//获取姓名

string getName() {

return m\_Name;

}

//获取年龄 可读可写 如果想修改（年龄的范围必须是0~150之间）

int getAge() {

m\_Age = 20;//初始化为20岁

return m\_Age;

}

void setAge(int age) {

if (age < 0 || age>150) {

return; //退出函数

}

m\_Age = age;

}

//设置情人 只写

void setLover(string Lover) {

m\_Lover = Lover;

}

private:

//姓名 可读可写

string m\_Name;

//年龄 只读

int m\_Age;

int m\_Age2; //可读可写

//情人 只写

string m\_Lover;

};

int main() {

Person p;

p.setName("张三");

cout << "姓名： " << p.getName() << endl;

//p.m\_setAge(18);

//p.m\_Age = 18; //报错

cout << "年龄1： " << p.getAge() << endl;

p.setAge(25);

cout << "年龄2： " << p.getAge2() << endl;

p.setAge(1000);

//设置情人为 王婆

p.setLover("王婆");

//cout << "情人： " << p.getLover() << endl; //报错，只写是不可以访问的

system("pause");

return 0;

}

//

// 案例： 立方体

/\*

设计立方体类（Cube)

求出立方体的面积和体积

分别用全局函数和成员函数判断两个立方体是否相等

\*/

class Cube {

//属性

private:

int m\_L;

int m\_W;

int m\_H;

public:

//行为

// 设置获取长宽高

//长

void setL(int l) {

m\_L = l;

}

int getL() {

return m\_L;

}

//宽

void setW(int w) {

m\_W = w;

}

int getW() {

return m\_W;

}

//高

void setH(int h) {

m\_H = h;

}

int getH() {

return m\_H;

}

//获取体积

int calculateV() {

return 2\*(m\_L \* m\_W + m\_L \* m\_H + m\_H \* m\_W) ;

}

//获取面积

int calculateS() {

return m\_L \* m\_W \* m\_H ;

}

//利用成员函数判断两个立方体是否相等

bool isSameByClass(Cube &c) {

if (m\_H == c.getH() && m\_L == c.getL() && m\_W == c.getW()) {

return true;

}

else

return false;

}

};

//全局函数判断两个立方体是否相等

bool isSame(Cube& c1, Cube& c2) {

if (c1.getH() == c2.getH() && c1.getL() == c2.getL() && c1.getW() == c2.getW()) {

return true;

}

else

return false;

}

int main() {

//创建立方体对象

Cube c1;

c1.setH(10);

c1.setW(10);

c1.setL(10);

cout << "c1的面积： " << c1.calculateS() << endl;

cout << "c1的体积： " << c1.calculateV() << endl;

Cube c2;

c2.setH(10);

c2.setW(20);

c2.setL(10);

if (isSame(c1, c2)) {

cout << "c1和c2相等。" << endl;

}

else

cout << "c1和c2不相等。" << endl;

if (c1.isSameByClass(c2)) {

cout << "成员函数的判断：c1和c2相等。"<<endl;

}

else

cout << "成员函数的判断：c1和c2不相等。" << endl;

return 0;

}

// 案例： 点和圆的关系

//点类

//\*\*\*\*\*\*可以分文件编写（头文件(写声明），源文件（写函数体））\*\*\*\*\*\*

#include"point.h"

#include"circle.h"

//class Point {

//private:

// int m\_X;

// int m\_Y;

//public:

// //设置获取X

// void setX(int x) {

// m\_X = x;

// }

// int getX() {

// return m\_X;

// }

// //设置获取Y

// void setY(int y) {

// m\_Y = y;

// }

// int getY() {

// return m\_Y;

// }

//};

//class Circle {

//private:

// int m\_R;

//

// Point m\_Center;//圆心

//

//public:

// //设置获取半径

// void setR(int r) {

// m\_R = r;

// }

// int getR() {

// return m\_R;

// }

// //设置获取圆心

// void setCenter(Point center) {

// m\_Center = center;

// }

// Point getCenter() {

// return m\_Center;

// }

//};

void isInCircle(Circle& c, Point& p) {

//计算两点间距离的平方

int distance =

(c.getCenter().getX() - p.getX()) \* (c.getCenter().getX() - p.getX()) +

(c.getCenter().getY() - p.getY()) \* (c.getCenter().getY() - p.getY());

//计算半径的平方

int rDistance=c.getR()\* c.getR();

//判断位置关系

if (distance == rDistance) {

cout << "点在圆上" << endl;

}

else if (distance > rDistance) {

cout << "点在圆外" << endl;

}

else

cout << "点在圆内" << endl;

}

int main() {

//创建圆

Circle c;

c.setR(10);

Point center;

center.setX(10);

center.setY(0);

c.setCenter(center);

//创建点

Point p;

p.setX(10);

p.setY(10);

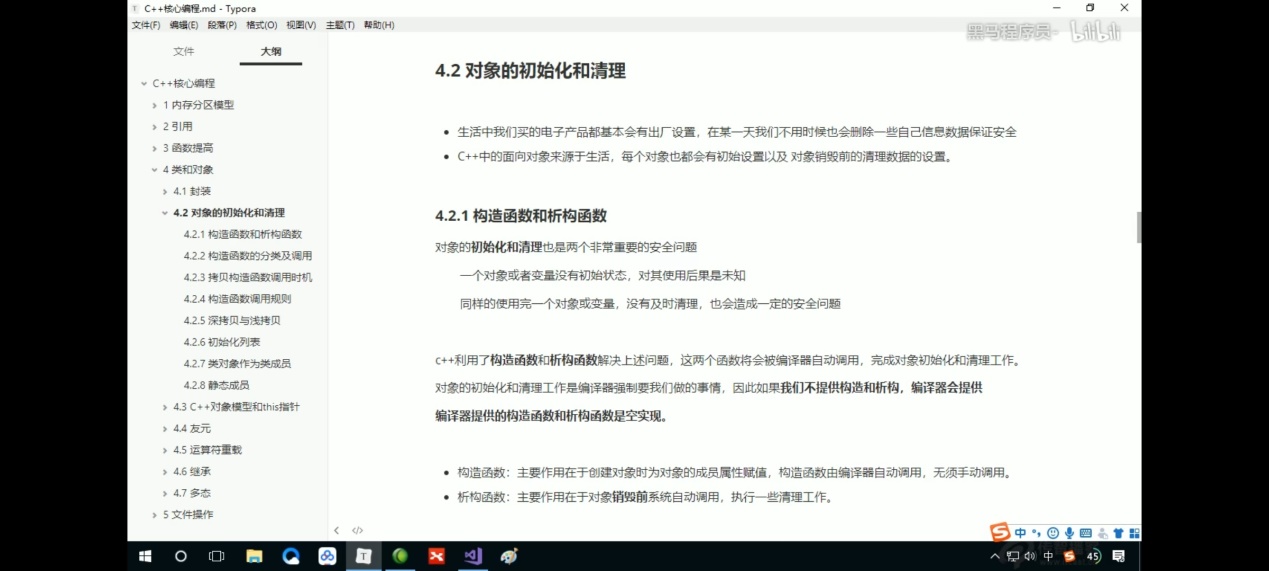
//判断关系

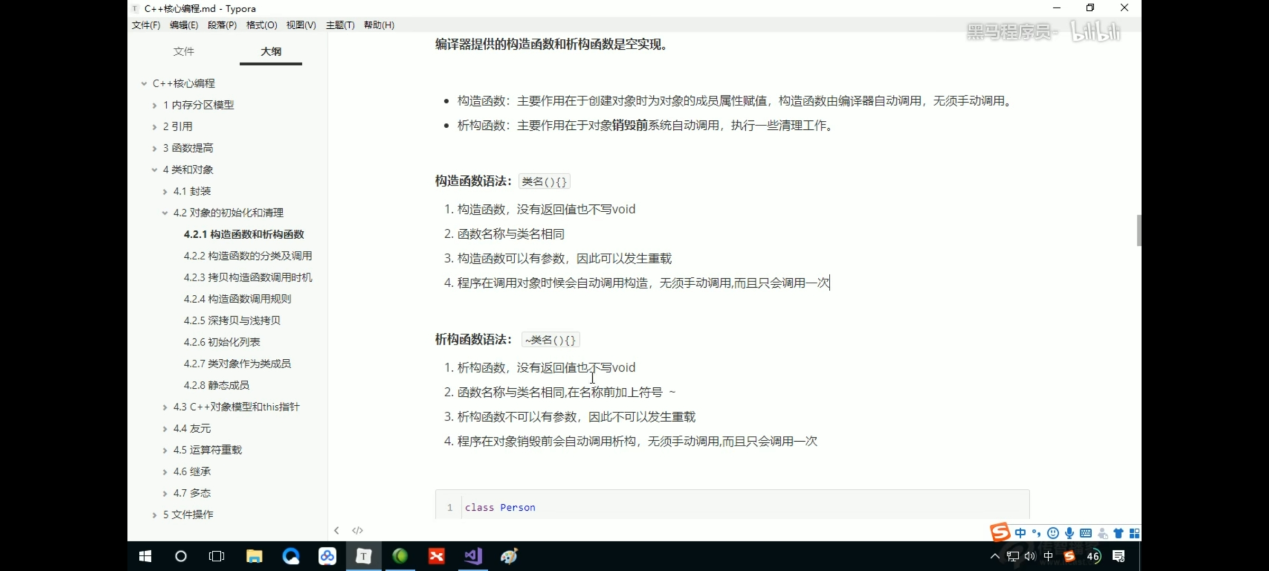
isInCircle(c,p);

return 0;

}

// 对象的初始化和清理

****



/\*

\* 构造函数：

语法： 类名( ) { }

构造函数，没有返回值也不写void

函数名称与类名相同

构造函数可以有参数，因此可以发生重载

程序在调用对象时会自动调用构造，无需手动调用，而且只会调用一次

\* 析构函数：

语法： ~类名 ( ){ }

析构函数，没有返回值也不写void

函数名称与类名相同，在名称前面加上~

析构函数不可以有参数，因此不可以发生重载

程序在对象销毁前会自动调用析构，无需手动调用，而且只调用一次

\*/

class Person {

public:

//构造函数 进行初始化

Person() {

cout << "Person构造函数的调用!" << endl;

}

//析构函数 进行清理操作

~Person() {

cout << "Person析构函数的调用!" << endl;

}

//构造函数和析构函数都是必须有的实现，如果我们不提供，编译器会提供一个空实现的构造和析构

};

void test1() {

Person p;

//栈上的数据，test1执行完毕之后，释放这个对象

}

int main() {

test1();

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

Person p;

system("pause");

return 0;

}

// 构造函数的分类及调用

class Person {

//分类：

// 按照参数分类 无参构造（默认构造） 有参构造

// 按照类型分类 普通构造函数 拷贝构造

public:

int age = 5;

//构造函数

//构造函数 进行初始化

Person() {

cout << "Person无参构造函数的调用!" << endl;

}

Person(int a) {

age = a;

cout << "Person有参构造函数的调用!" << endl;

}

//拷贝构造函数

Person(const Person &p) {

//将传入的对象的所有的属性拷贝到当前对象身上

age = p.age;

cout << "Person拷贝构造函数的调用!" << endl;

}

//析构函数 进行清理操作

~Person() {

cout << "Person析构函数的调用!" << endl;

}

};

//调用

void test1() {

// 括号法

/\*

Person p1; //默认构造

Person p2(10); //有参构造

Person p3(p2); //拷贝构造

cout << "P2的年龄为： " << p2.age << endl;

cout << "P3的年龄为： " << p3.age << endl;

注意事项：

调用默认构造函数的时候，不要加 ( )

因为下面这段代码，编译器会认为是一个函数的声明

\*/

//显示法

/\*

Person p1; //默认构造

Person p2 = Person(10); //有参构造

Person p3 = Person(p2); //拷贝构造

cout << "aaaaaa" << endl;

Person(10); //匿名对象 特点： 当前执行结束后，系统会立即回收掉匿名对象

注意事项：

不要利用拷贝函数构造函数 初始化匿名对象

编译器会认为 Person(p3) ===Person p3;

//Person(p3);

\*/

//隐式转换法

Person p4 = 10; //相当于Person p4 = Person(10);

Person p5 = p4; //相当于Person p5 = Person(p4);

}

int main() {

test1();

system("pause");

return 0;

}

// 拷贝构造函数的调用时机

//将一个实体对象的参数复制给另一个对象

class Person {

public:

Person() {

cout << "Person默认构造函数的调用!" << endl;

}

~Person() {

cout << "Person析构函数的调用!" << endl;

}

int m\_Age;

Person(int age) {

cout << "Person有参构造函数的调用!" << endl;

m\_Age = age;

}

Person(const Person &p) {

cout << "Person拷贝构造函数的调用!" << endl;

m\_Age = p.m\_Age;

}

};

// 1.使用一个已经创建完毕的对象来初始化一个新的对象

void test1() {

Person p1(20);

Person p2(p1);

cout << "p2的年龄为" << p2.m\_Age << endl;

}

// 2.值传递的方式给函数参数传值

void doWork(Person p) {

}

void test2() {

Person p;

doWork(p);

}

// 3.值方式返回局部对象

Person doWork2() {

Person p1;

cout << (int\*)&p1 << endl;

return p1;

}

void test3() {

Person p = doWork2();

cout << (int\*)&p << endl;

}

int main() {

test1();

cout << "~~~~~~~~~~~~~~~~~~~~" << endl;

test2();

cout << "~~~~~~~~~~~~~~~~~~~~" << endl;

test3();

return 0;

}

// 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） （空实现）

2.默认析构函数（无参，函数体为空） （空实现）

3.默认拷贝构造函数，对数学进行值拷贝 （值拷贝）

\*

\* 构造函数调用规则如下：

1. 如果用户定义有函数，c++不再提供默认无参构造，但会提供默认拷贝构造

2. 如果用户定义拷贝构造函数，c++不会提供其他构造函数

\*/

class Person {

public:

//无参（默认）构造函数

Person() {

cout << "无参构造函数！" << endl;

}

Person(int age) {

m\_Age = age;

cout << "有参构造函数！" << endl;

}

Person(const Person& p) {

m\_Age = p.m\_Age;

cout << "拷贝构造函数！" << endl;

}

~Person() {

cout << "无参析构函数！" << endl;

}

int m\_Age = 0;

};

void test1() {

Person p;

p.m\_Age = 18;

Person p2(p);

cout << "p2的年龄： " << p2.m\_Age << endl;

}

int main(){

test1();

return 0;

}

// 深拷贝与浅拷贝

//浅拷贝：由编译器编写,是一个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

//如果属性有在堆区开辟的，一定要自己提供构造函数，防止浅拷贝带来的问题

class Person{

public:

Person(){

cout << "有参默认构造函数" << endl;

}

Person(int age,int height) {

m\_Age = age;

m\_Height = new int(height);

cout << "Person的有参构造函数" << endl;

}

//自己实现拷贝构造函数 解决拷贝带来的问题

Person(const Person& p) {

cout << "Person拷贝构造函数的调用" << endl;

m\_Age = p.m\_Age;

m\_Height = p.m\_Height; //编译器默认实现这行代码

//深拷贝操作

m\_Height = new int(\*p.m\_Height);

}

~Person(){

cout << "Person的析构造函数" << endl;

if (m\_Height != NULL) {

delete m\_Height;

m\_Height = NULL;

}

}

int m\_Age;

int\* m\_Height;

};

void test1() {

Person p1(18,166);

cout << "p1的年龄： " << p1.m\_Age << endl;

cout << "p1的身高： " << \*p1.m\_Height << endl;

Person p2(p1);

cout << "p2的年龄： " << p2.m\_Age << endl;

cout << "p2的身高： " << \*p2.m\_Height << endl;

}

int main(){

test1();

return 0;

}

// 初始化列表

//作用： C++提供了初始化列表语法，用来自初始化属性

//语法： 构造函数( ):属性1(值1),属性2(值2),...{ }

class Person {

public:

//传统初始化

/\*Person(int a, int b, int c) {

m\_A = a; m\_B = b; m\_C = c;

}\*/

//初始化列表初始化

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

};

int m\_A;

int m\_B;

int m\_C;

};

void test1() {

Person p(10, 20, 30);

cout << "m\_A = " << p.m\_A << endl;

cout << "m\_B = " << p.m\_B << endl;

cout << "m\_C = " << p.m\_C << endl;

}

int main(){

test1();

return 0;

}

// 类对象作为类成员

class Phone {

public:

string m\_PName; //手机品牌

Phone(string pName) {

m\_PName = pName;

cout << "Phone构造函数调用！" << endl;

}

~Phone() {

cout << "Phone析构函数调用！" << endl;

}

};

class Person {

public:

string m\_Name;

Phone m\_Phone;

//Phone m\_Phone = pName; 隐式转换法

Person(string name, string pName):

m\_Name(name),m\_Phone(pName)

{

cout << "Person构造函数调用！" << endl;

}

~Person() {

cout << "Person析构函数调用！" << endl;

}

};

//当其他类作为本类成员时，构造时候先构造类对象，再构造类自身

//析构函数的调用顺序搞好相反

void test1(){

Person p("张三","华为Mate40");

cout << p.m\_Name << "拿着" << p.m\_Phone.m\_PName << endl;

}

int main() {

test1();

return 0;

}

// 静态成员函数

//所有对象共享一个函数

//静态成员函数只能访问静态成员变量

class Person

{

public:

//静态成员函数

static void func() {

m\_A = 100; //静态成员函数可以访问 静态成员变量

//m\_B = 200; //静态成员函数 不可访问 非静态成员变量 无法区分到底是哪个对象的成员函数

cout << "static void func调用 " << endl;

}

static int m\_A; //静态成员变量

int m\_B;

//静态成员函数也是有访问权限的

private:

static void func2() {

cout << "static void func2调用 " << endl;

}

};

int Person::m\_A = 10;

void test01() {

//两种方式

//1,通过对象访问

Person p;

p. func();

//2.通过类名访问

Person::func();

//Person::func2(); //类外不可访问

}

int main() {

test01();

return 0;

}

// C++对象模型和this指针

// 成员变量 和 成员函数分开存储

class Person {

int m\_A; //非静态成员变量 属于类的对象上

static int m\_B; //静态成员变量 不属于类对象上

void func(){}//非静态成员函数 不属于类对象上

static void func2() {}//静态成员函数 不属于类对象上

};

int Person::m\_B = 10;

void test1() {

Person p;

//空对象占用内存空间为：1

//C++编译器会给每个空对象也分配一个字节空间(大小为4)，是为了区分对象占用的内存空间的位置

//每个空对象也应该有一个独一无二的内存地址

cout << "size of p = " << sizeof(p) << endl;

}

void test2() {

Person p;

cout << "size of p = " << sizeof(p) << endl;

}

int main() {

test1();

test2();

return 0;

}

// this指针

//this指针指向被调用的成员函数所属的对象

//this指针是隐含在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：

\*当形参和成员变量同名时，可用this指针来区分

\* 在类的非静态成员函数中返回对象本身，可使用return \*this

\*/

/\*

\*this指针->类 对象不占用类的大小

\*

\* this指针->地址 地址里存放的首地址

\*

\* 静态成员函数先于对象存在 故没有this指针

\*/

class Person {

public:

Person(int age) {

//this指针 指向被调用的成员函数 所属的对象

this->age = age;

}

Person &PersonAddAge(Person& p) {

this->age += p.age;

//this指向p2的指针 而\*this指向的就是p2这个对象本体

return \*this;

}

int age;

};

//1.解决名称冲突

void test1() {

Person p1(18);

cout << "p1的年龄为： " << p1.age << endl;

}

//2.返回对象本身 用\*this

void test2() {

Person p1(10);

Person p2(5);

//链式编程思想

p2.PersonAddAge(p1).PersonAddAge(p1).PersonAddAge(p1);

cout << "p2的年龄为： " << p2.age << endl;

}

int main() {

test1();

test2();

return 0;

}

//空指针调用成员函数

class Person {

public:

void showClassName() {

cout << "this is Person class " << endl;

}

void showPersonAge() {

//传入的指针为空，找不到实体对象，会报错

if (this == NULL)

return; //保持程序的健壮性

cout << "age = " << m\_Age << endl;

}

int m\_Age;

};

//

void test1() {

Person\* p = NULL;

p->showClassName();

p->showPersonAge();

}

int main() {

test1();

return 0;

}

// const修饰成员函数

//常函数

class Person {

public:

//this指针的本质 时指针常量 指针的指向是不可修改的

// 在成员函数后面加const的本质：修饰的是this指向，让指针指向的值也不可修改

//Person \* const this ——》 const Person \* const this;

void showPerson()const {

// m\_A = 100;

//this = NULL; //this指针不可修改指针的指向的

this->m\_B = 100;

}

void func() {

}

int m\_A;

mutable int m\_B; //特殊变量（加上mutable）， 即使在常函数中，也可修改这个值

};

void test1() {

Person p;

p.showPerson();

}

//常对象

void test2() {

const Person p; //在对象前加const,变为常对象

//p.m\_A = 100;

p.m\_B = 100; //m\_B是特殊值，在常对象下也可修改

//常对象只能调用常函数

p.showPerson();

//p.func(); //常对象 不可以调用普通成员函数，因为普通成员函数可以修改属性

}

int main() {

test1();

test2();

}

// 友元

// 关键字 friend

//全局函数做友元

class Building {

friend void goodFriend(Building\* building);

//goodFriend全局函数是Building的友元，可以访问Building中的私有成员

public:

Building() {

m\_SittingRoom = "客厅";

m\_BedRoom = "卧室";

}

public:

string m\_SittingRoom; // 客厅

private:

string m\_BedRoom; //卧室

};

//全局函数

void goodFriend(Building\* building) {

cout << "好朋友全局函数 正在访问 ： " << building->m\_SittingRoom << endl;

cout << "好朋友全局函数 正在访问 ： " << building->m\_BedRoom << endl;

}

void test1(){

Building building;

goodFriend(&building);

}

int main() {

test1();

return 0;

}

//类做友元

class Building;

class GoodFriend {

public:

GoodFriend();

void visit(); //参观函数 访问Building中的属性

Building\* building;

};

class Building {

friend class GoodFriend;

public:

Building();

public:

string m\_SittingRoom; //客厅

private:

string m\_BedRoom; // 卧室

};

//类外写成员函数

Building::Building() {

m\_SittingRoom = "客厅";

m\_BedRoom = "卧室";

}

GoodFriend::GoodFriend() {

//创建一个建筑物

building = new Building;

}

void GoodFriend::visit() {

cout << "好朋友正在访问： " << building->m\_SittingRoom << endl;

cout << "好朋友正在访问： " << building->m\_BedRoom << endl;

}

void test1() {

GoodFriend g;

g.visit();

}

int main() {

test1();

return 0;

}

// 成员函数做友元

class Building;

class GoodFriend {

public:

GoodFriend();

void visit(); //让visit函数作友元 可以访问Building中私有成员

void visit2(); //让visit2函数不可以访问Building中的私有成员

Building\* building;

};

class Building {

//告诉编译器 GoodFriend类下的私有成员作为本类的友元，可以访问私有成员

friend void GoodFriend::visit();

public:

Building();

public:

string m\_SittingRoom; //客厅

private:

string m\_BedRoom; //卧室

};

//类外实现成员函数

Building::Building() {

m\_SittingRoom = "客厅";

m\_BedRoom = "卧室";

}

GoodFriend::GoodFriend() {

building = new Building;

}

void GoodFriend::visit() {

cout << "visit函数正在访问：" << building->m\_SittingRoom << endl;

cout << "visit函数正在访问：" << building->m\_BedRoom << endl;

}

void GoodFriend::visit2() {

cout << "visit2函数正在访问：" << building->m\_SittingRoom << endl;

//cout << "visit2函数正在访问：" << building->m\_BedRoom << endl; //非友元的存在，不可访问

}

void test1() {

GoodFriend gf;

gf.visit();

gf.visit2();

}

int main() {

test1();

return 0;

}

// 运算符重载

//概念：对已有的运算符重新进行定义，赋予其另外一种功能，以适应不同的数据类型

//重载运算符 加号（ + ）

class Person {

public:

//1. 成员函数重载+号

/\*Person operator+(Person& p) {

Person temp;

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

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

return temp;

}\*/

int m\_A;

int m\_B;

};

//2. 全局函数重载+号

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

Person temp;

temp.m\_A = p1.m\_A + p2.m\_A;

temp.m\_B = p1.m\_B + p2.m\_B;

return temp;

}

//函数重载

Person operator+(Person& p1,int num) {

Person temp;

temp.m\_A = p1.m\_A + num;

temp.m\_B = p1.m\_B + num;

return temp;

}

void test1() {

Person p1;

p1.m\_A = 10;

p1.m\_B = 10;

Person p2;

p2.m\_A = 10;

p2.m\_B = 10;

Person p3 = p1 + p2;

cout << "P3.m\_A = " << p3.m\_A << endl;

cout << "P3.m\_B = " << p3.m\_B << endl;

//成员函数重载 本质上为 Person p3 = p1.operator+(p2);

//全局函数重载 本质上为 Person p3 = operator+(p1，p2);

//运算符重载 也可以发生函数重载

Person p4 = p1 + 10; //相当于Person p3 = operator+(p1，10);

cout << "P4.m\_A = " << p4.m\_A << endl;

cout << "P4.m\_A = " << p4.m\_A << endl;

}

//对于内置的数据类型的表达式的运算符是不可更改的 如：1+1=2——》1+1=0

//不要滥用运算符重载

int main() {

test1();

return 0;

}

//左移运算符

class Person {

friend ostream& operator<<(ostream& out, Person& p);

public:

Person(int a, int b) {

m\_A = a;

m\_B = b;

}

private:

////1. 成员函数重载 左移运算符

//void operator<<(cout) {

//}

//一般不会利用成员函数重载<<运算符，因为无法实现cout在左边

int m\_A;

int m\_B;

};

//2. 只能利用全局函数重载 左移运算符

ostream & operator<<(ostream &out, Person &p) //本质 operator<< (cout , p) 简化 cout << p

{

out << "m\_A = " << p.m\_A << " m\_B = " << p.m\_B << endl;

return out;

}

void test1() {

Person p(10,10);

cout << p << endl << "hello world" << endl;

}

int main() {

test1();

return 0;

}

//运算符重载 递增

//自定义整型

class MyInteger {

friend ostream& operator<<(ostream& cout, MyInteger myint);

public:

MyInteger() {

m\_Num = 0;

}

//重载前置++运算符 返回引用是为了一直对一个数据进行递增操作

MyInteger& operator++() {

//先进行++运算

m\_Num++;

//再将自身做返回

return \*this;

}

//重载后置++运算符 返回值，防止运行后被释放

MyInteger operator++(int) { // int表示占位参数，可以用于区分前置和后置递增

//先 记录当时结果

MyInteger temp = \*this;

//后 递增

m\_Num++;

//最后将记录结果做返回

return temp;

}

private:

int m\_Num;

};

ostream& operator<<(ostream& cout, MyInteger myint) {

cout << myint.m\_Num;

return cout;

}

void test1() {

MyInteger myint;

cout << "前置递增： " << endl;

cout << ++myint << endl;

cout << myint << endl;

};

void test2() {

MyInteger myint;

cout << "后置递增： " << endl;

cout << myint++ << endl;

cout << myint << endl;

}

int main() {

test1();

test2();

return 0;

}

// 赋值运算符重载

/\*

类包括：

默认构造函数（无参，函数体为空）

默认析构函数（无参，函数体为空）

默认拷贝构造函数，对属性进行值拷贝

赋值运算符 operator=,对属性进行值拷贝

\*\*/

class Person {

public:

Person(int age) {

m\_Age = new int(age);

}

~Person() {

if (m\_Age != NULL) {

delete m\_Age;

m\_Age = NULL;

}

}

//浅拷贝 调用析构函时重复释放堆区数据，程序崩溃

//重载 赋值运算符

Person& operator=(Person& p) {

//编译器是提供浅拷贝

//m\_Age = p.m\_Age;

//应该先判断是否有属性在堆区，如果有，先释放干净，然后再深拷贝

if (m\_Age != NULL) {

delete m\_Age;

m\_Age = NULL;

}

//深拷贝 解决浅拷贝问题

m\_Age = new int(\*p.m\_Age);

return \*this;

}

int \*m\_Age;

};

void test1() {

Person p1(18);

cout << "p1的年龄为： " << \*p1.m\_Age << endl;

Person p2(12);

cout << "p2的年龄为： " << \*p2.m\_Age << endl;

p2 = p1; //赋值操作

cout << "p2的年龄为： " << \*p2.m\_Age << endl;

Person p3(20);

cout << "p3的年龄为： " << \*p3.m\_Age << endl;

p3 = p2 = p1;

cout << "p1的年龄为： " << \*p1.m\_Age << endl;

cout << "p2的年龄为： " << \*p2.m\_Age << endl;

cout << "p3的年龄为： " << \*p3.m\_Age << endl;

}

int main() {

test1();

return 0;

}

//运算符重载 关系运算符

class Person {

public:

Person(string name, int age) {

m\_Name = name;

m\_Age = age;

}

//重载关系运算符

bool operator==(Person& p) {

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

return true;

return false;

}

bool operator!=(Person& p) {

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

return true;

return false;

}

string m\_Name;

int m\_Age;

};

void test() {

Person p1("Tom", 18);

Person p2("Tom", 18);

if (p1 == p2)

cout << "p1 = p2 " << endl;

else

cout << "p1 != P2 " << endl;

Person p3("Jerry", 20);

if (p1 != p3)

cout << "p1 != p3 " << endl;

else

cout << "p1 = P3 " << endl;

}

int main() {

test();

return 0;

}

//重载 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的像函数的调用，因此称为仿函数

仿函数没有固定的写法，非常的灵活

\*/

class MyPrint {

public:

//重载函数调用运算符

void operator()(string test) {

cout << test << endl;

}

};

void MyPrint2(string test) {

cout << test << endl;

}

void test() {

MyPrint myPrint;

myPrint("hello world");

MyPrint2("hello world");

}

//仿函数非常灵活，没有固定的写法

//加法类

class MyAdd {

public:

int operator()(int num1, int num2) {

return num1 + num2;

}

};

void test2() {

MyAdd myAdd;

int ret = myAdd(100, 100);

cout << "ret = " << ret << endl;

//匿名函数对象

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

}

int main() {

test();

test2();

return 0;

}

// 继承

//普通实现

/\*

Java页面

class Java {

public:

void header() {

cout << "首页、公共课、登录注册、、、（公共头部)" << endl;

}

void footer() {

cout << "帮助中心，交流合作、站内地图、、、（公共底部）" << endl;

}

void left() {

cout << "Java、Python、C++、、、(公共课分类列表）" << endl;

}

void content(){

cout << "Java课视频"<<endl;

}

};

Python页面

class Python {

public:

void header() {

cout << "首页、公共课、登录注册、、、（公共头部)" << endl;

}

void footer() {

cout << "帮助中心，交流合作、站内地图、、、（公共底部）" << endl;

}

void left() {

cout << "Java、Python、C++、、、(公共课分类列表）" << endl;

}

void content() {

cout << "Python课视频" << endl;

}

};

C++页面

class Cpp {

public:

void header() {

cout << "首页、公共课、登录注册、、、（公共头部)" << endl;

}

void footer() {

cout << "帮助中心，交流合作、站内地图、、、（公共底部）" << endl;

}

void left() {

cout << "Java、Python、C++、、、(公共课分类列表）" << endl;

}

void content() {

cout << "C++课视频" << endl;

}

};

\*/

//继承实现

//公共页面

class BasePage {

public:

void header() {

cout << "首页、公共课、登录注册、、、（公共头部)" << endl;

}

void footer() {

cout << "帮助中心，交流合作、站内地图、、、（公共底部）" << endl;

}

void left() {

cout << "Java、Python、C++、、、(公共课分类列表）" << endl;

}

};

//Java页面

class Java :public BasePage {

public:

void content() {

cout << "Java课视频" << endl;

}

};

//Python页面

class Python :public BasePage {

public:

void content() {

cout << "Python课视频" << endl;

}

};

//C++页面

class Cpp :public BasePage {

public:

void content() {

cout << "C++课视频" << endl;

}

};

void test1() {

cout << "Java下载视频如下： "<<endl;

Java ja;

ja.header();

ja.footer();

ja.left();

ja.content();

cout << endl;

cout << "——————————————" << endl;

cout << "Python下载视频如下： " << endl;

Python Py;

Py.header();

Py.footer();

Py.left();

Py.content();

cout << endl;

cout << "——————————————" << endl;

cout << "C++下载视频如下： " << endl;

Cpp Cpp;

Cpp.header();

Cpp.footer();

Cpp.left();

Cpp.content();

}

// 继承的好处：减少重复的代码

//语法： class 子类: 继承方式(public，protect，private) 父类

//子类 也称为 派生类

//父类 也称为 基类

//从基类继承过来的表现其共性，而新增成员体现其个性

int main() {

test1();

return 0;

}

// 继承方式

/\*

有三种

公共继承

保护继承

私有继承

\*/

//公共继承

class Base1 {

public:

int m\_A;

protected:

int m\_B;

private:

int m\_C;

};

class Son1 :public Base1 {

public:

void func() {

m\_A = 10; //父类中的公共权限成员 到子类中依然是公共权限

m\_B = 10; //父类中的保护权限成员 到子类中依然是保护权限

//m\_C = 10; //父类中的私有权限成员 子类访问不到

}

};

//保护继承

class Base2 {

public:

int m\_A;

protected:

int m\_B;

private:

int m\_C;

};

class Son2 :protected Base2 {

public:

void func() {

m\_A = 20; //父类中的公共权限成员 到子类中变为保护权限成员

m\_B = 20; //父类中的保护权限成员 到子类中依然是保护权限成员

//m\_C = 20; //父类中的私有权限成员 子类访问不到

}

};

//私有继承

class Base3 {

public:

int m\_A;

protected:

int m\_B;

private:

int m\_C;

};

class Son3 :private Base3 {

public:

void func() {

m\_A = 20; //父类中的公共权限成员 到子类中变为私有权限成员

m\_B = 20; //父类中的保护权限成员 到子类中变为私有权限成员

//m\_C = 20; //父类中的私有权限成员 子类访问不到

}

};

class GrandSon3 :public Son3 {

public:

void func() {

//m\_A = 400; //在Son3中 m\_A变为私有，即使是儿子，也是访问不到

//m\_B = 400; //在Son3中 m\_A变为私有，即使是儿子，也是访问不到

}

};

void test1() {

Son1 s1;

s1.m\_A = 100;

//s1.m\_B = 100; //在Son1中，m\_B是保护权限 类外访问不到

}

void test2() {

Son2 s2;

//s2.m\_A = 200; //在Son2中，m\_A变为保护权限 类外访问不到

//s2.m\_B = 200; //在Son2中，m\_B是保护权限 类外访问不到

}

void test3() {

Son3 s3;

//s3.m\_A = 300; //在Son3中，m\_A变为私有权限 类外访问不到

//s3.m\_B = 300;

}

int main(){

test1();

test2();

test3();

return 0;

}

// 继承中的对象模型

class Base {

public:

int m\_A;

protected:

int m\_B;

private:

int m\_C;

};

class Son :public Base {

public:

int m\_D;

};

void test1() {

cout << "size of Son = " << sizeof(Son) << endl;

// 输出结果：16

//父类中的所有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 是被编译器隐藏了，因此访问不到，但确实被继承了

/\*

\* 利用VS 2019的开发人员命令提示符中查看对象模型

\* 步骤： 跳转至该文件目录下

\* 输入： cl /dl reportSingleClassLayout类名 （所属文件）"该cpp文件名" ——》回车

\*/

}

int main() {

test1();

return 0;

}

// 继承中的构造和析构顺序

class Base {

public:

Base() {

cout << "Base构造函数！" << endl;

}

~Base() {

cout <<"Base析构函数！"<<endl;

}

};

class Son:public Base {

public:

Son() {

cout << "Son构造函数！" << endl;

}

~Son() {

cout << "Son析构函数！" << endl;

}

};

void test() {

Base B1;

cout << "~~~~~~~~~~~~~~~~~~~~~~~~" << endl;

Son S1;

}

//继承中的构造和析构的顺序如下：

//先构造父类，再构造子类，析构的顺序正好与构造的顺序相反

int main() {

test();

return 0;

}

// 继承中同名的成员处理方式

class Base {

public:

Base() {

m\_A = 100;

}

void func() {

cout << "Base - func()调用"<<endl;

}

void func(int a) {

cout << "Base - func(int a)调用" << endl;

}

int m\_A;

};

class Son:public Base {

public:

Son() {

m\_A = 100;

}

void func() {

cout << "Son - func()调用" << endl;

}

int m\_A;

};

//同名成员属性处理方式

void test1() {

Son s;

cout << "Son 中 m\_A = " << s.m\_A << endl;

//如果通过子类对象 访问到父类中的同名成员，需要加作用域

cout << "Base 中 m\_A = " << s.Base::m\_A << endl;

}

//同名函数处理方式

void test2() {

Son s;

//如果通过子类成员函数 访问到父类中的同名成员函数，需要加作用域

s.Base::func();

//s.func(100);

//如果子类中出现和父类同名的成员函数，子类的同名成员会隐藏掉父类中的所有成员函数

//如果想访问到父类中被隐藏的同名成员函数，需要加作用域

s.Base::func(100);

}

int main() {

test1();

test2();

return 0;

}

// 继承同名静态成员处理方式

class Base {

public:

static int m\_A;

static void func() {

cout << "Base - static void func()调用" << endl;

}

};

int Base::m\_A = 100;

class Son :public Base {

public:

static int m\_A;

static void func() {

cout << "Son - static void func()调用" << endl;

}

};

int Son::m\_A = 200;

//同名静态成员属性

void test1(){

//通过对象访问

Son s;

cout << "对象访问" << endl;

cout << "Son m\_A = " << s.m\_A << endl;

cout << "Base m\_A = " << s.Base::m\_A << endl;

//通过类名访问

cout << "类名访问" << endl;

cout << "Son m\_A = " << Son::m\_A << endl;

//第一个::表示类名访问，第二个::表示父类的作用域

cout << "Base m\_A = " << Son::Base::m\_A << endl;

}

//同名静态成员函数

void test2() {

//通过对象访问

Son s;

cout << "对象访问" << endl;

s.func();

s.Base::func();

//通过类名访问

cout << "类名访问" << endl;

Son::func();

Son::Base::func();

//如果子类中出现和父类同名的静态成员函数，子类的同名成员隐会藏掉父类中的所有成员函数

//如果想访问到父类中被隐藏的同名成员函数，需要加作用域

}

int main() {

test1();

cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << endl;

test2();

return 0;

}

// 多继承语法

//语法： class 子类: 继承方式 父类1, 继承方式 父类2 ...{ }

class Base2 {

public:

Base2() {

m\_A = 100;

}

int m\_A;

};

class Base1 {

public:

Base1() {

m\_A = 200;

}

int m\_A;

};

//子类

class Son :public Base1, public Base2 {

public:

Son() {

m\_C = 300;

m\_D = 400;

}

int m\_C;

int m\_D;

};

void test1() {

Son s;

cout << "size of Son = " << sizeof(s) << endl;

//输出结果：16

//当父类中出现同名成员，需要加作用域区分

cout << "Base1 m\_A = " << s.Base1::m\_A<< endl;

cout << "Base2 m\_A = " << s.Base2::m\_A << endl;

}

int main() {

test1();

return 0;

}

// 菱形继承

//动物类

class Animal {

public:

int m\_Age;

};

//利用虚继承 解决菱形继承的问题

// 在继承之前 加上关键字 virtual 变为虚继承

// Animal称为虚基类（最大的虚继承）

//羊类

class Sheep:virtual public Animal{};

//驼类

class Tuo:virtual public Animal{};

//羊驼类

class SheepTuo:public Sheep,public Tuo{};

void test() {

SheepTuo st1;

st1.Sheep::m\_Age = 18;

st1.Tuo::m\_Age = 20;

//当菱形继承时，两个父类拥有相同的数据，需要加以作用域用以区分

cout << "st1.Sheepm\_Age = " << st1.Sheep::m\_Age << endl;

cout << "st1.Sheepm\_Age = " << st1.Tuo::m\_Age << endl;

//据常识 该数据只需一份，而菱形继承有两份，浪费资源且形成了错误

//虚继承之后 指向的是同一块数据的地址 故只会保存最后一次修改后的数据

SheepTuo st2;

st2.Sheep::m\_Age = 21;

st2.Tuo::m\_Age = 28;

cout << "st2.Sheepm\_Age = " << st2.Sheep::m\_Age << endl;

cout << "st2.Sheepm\_Age = " << st2.Tuo::m\_Age << endl;

cout << "st2.m\_Age = " << st2.m\_Age << endl;

}

int main() {

test();

return 0;

}

// 多态

/\*

\* 多态：

\* 静态多态：运算符重载 和 函数重载属于静态多态，复用函数名

\* 动态多态：派生类和虚函数实现要运行时多态

\*

\* 动态多态和静态多态的区别：

\* 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

\*/

class Animal {

public:

virtual void speak() {

cout << "动物在说话" << endl;

}

};

//猫类

class Cat :public Animal {

public:

//函数重写： 函数返回值类型，函数名，参数列表，完全相同

void speak() {

cout << "小猫在说话！" << endl;

}

};

//狗类

class Dog :public Animal {

public:

void speak() {

cout << "小狗在说话！" << endl;

}

};

//地址早绑定 在编译阶段确定函数地址——》动物在说话

//地址晚绑定 在运行阶段确定函数地址——》小猫在说话

/\*

动态多态满足的条件

1.有继承关系

2.子类重写父类的虚函数

\*/

/\*

动态多态使用

父类的指针或者引用 执行于类对象

\*/

void doSpeak(Animal& animal) { //Animal& animal = cat;

animal.speak();

}

void test1() {

Cat cat;

doSpeak(cat);

Dog dog;

doSpeak(dog);

}

int main() {

test1();

return 0;

}

//写虚函数时，类内部发生变化，产生一个虚函数指针，故产生了多态

// 当子类重写父类的虚函数

//子类中的虚函数表 内部 会被替换成 子类的虚函数地址

// 多态案例 计算器类

//普通实现

class Calculator {

public:

int getResult(string oper){

if (oper == "+")

return m\_Num1 + m\_Num2;

else if (oper == "-")

return m\_Num1 - m\_Num2;

else if (oper == "\*")

return m\_Num1 \* m\_Num2;

}

//开闭原则： 对扩展进行开发，对修改进行封闭

int m\_Num1;

int m\_Num2;

};

//利用多态实现计算器

//实现计算器抽象类

class AbstractCalculator {

public:

virtual int getResult() {

return 0;

}

int m\_Num1;

int m\_Num2;

};

//加法计算器类

class AddCalculator :public AbstractCalculator {

public:

int getResult() {

return m\_Num1 + m\_Num2;

}

};

//减法计算类

class SubCalculator :public AbstractCalculator {

public:

int getResult() {

return m\_Num1 - m\_Num2;

}

};

//乘法计算类

class MulCalculator :public AbstractCalculator {

public:

int getResult() {

return m\_Num1 \* m\_Num2;

}

};

void test1() {

//创建对象

Calculator c1;

c1.m\_Num1 = 10;

c1.m\_Num2 = 10;

cout << c1.m\_Num1 << " + " << c1.m\_Num2 << " = " << c1.getResult("+") << endl;

cout << c1.m\_Num1 << " - " << c1.m\_Num2 << " = " << c1.getResult("-") << endl;

cout << c1.m\_Num1 << " - " << c1.m\_Num2 << " = " << c1.getResult("\*") << endl;

}

void test2() {

//多态使用条件

//父类指针或者引用指向子类对象

//加法运算

AbstractCalculator\* abc = new AddCalculator;

abc->m\_Num1 = 10;

abc->m\_Num2 = 10;

cout << abc->m\_Num1 << " + " << abc->m\_Num2 << " = " << abc->getResult() << endl;

delete abc; //new的堆区数据，使用完成后记得销毁

//减法运算

abc = new SubCalculator;

abc->m\_Num1 = 10;

abc->m\_Num2 = 10;

cout << abc->m\_Num1 << " - " << abc->m\_Num2 << " = " << abc->getResult() << endl;

delete abc;

//乘法运算

abc = new MulCalculator;

abc->m\_Num1 = 10;

abc->m\_Num2 = 10;

cout << abc->m\_Num1 << " \* " << abc->m\_Num2 << " = " << abc->getResult() << endl;

delete abc;

}

int main() {

test1();

test2();

return 0;

}

/\*

多态的好处：

1.组织结构清晰

2.可读性强

\*/

// 纯虚函数和抽象类

//纯虚函数语法：virtual 返回值类型 函数名 （参数列表） = 0；

class Base {

public:

//纯虚函数

/\*

\* 当类中有了纯虚函数，这个类也称为抽象类

抽象类的特点：

无法实例化对象

抽象类的子类必须重写抽象类中的纯虚函数，否则也属于抽象类

\*/

virtual void func() = 0;

};

class Son :public Base {

public:

virtual void func() { };

};

void test1(){

//Base b; //抽象类无法实例化对象

Son s; // 抽象类的子类必须重写抽象类中的纯虚函数，否则也属于抽象类

Base\* base = new Son;

base->func(); //new的新的指针，调用的时Son的实例化对象，故可以实现

}

int main() {

test1();

return 0;

}

// 案例 制作饮品

class AbstractDrinking {

public:

//煮水

virtual void Boil() = 0;

//冲泡

virtual void Brew() = 0;

//倒入杯中

virtual void PourInCup() = 0;

//加入辅料

virtual void PutSomething() = 0;

//制作饮品

void makeDrink() {

Boil();

Brew();

PourInCup();

PutSomething();

}

};

//制作咖啡

class Coffee:public AbstractDrinking {

public:

//煮水

virtual void Boil() {

cout << "煮农夫山泉" << endl;

};

//冲泡

virtual void Brew() {

cout << "冲泡咖啡" << endl;

};

//倒入杯中

virtual void PourInCup() {

cout << "倒入杯中" << endl;

};

//加入辅料

virtual void PutSomething() {

cout << "加入糖和牛奶" << endl;

};

};

//制作绿茶

class Tea :public AbstractDrinking {

public:

//煮水

virtual void Boil() {

cout << "煮恒大冰泉" << endl;

};

//冲泡

virtual void Brew() {

cout << "冲泡茶叶" << endl;

};

//倒入杯中

virtual void PourInCup() {

cout << "倒入杯中" << endl;

};

//加入辅料

virtual void PutSomething() {

cout << "加入糖和枸杞" << endl;

};

};

void doWork(AbstractDrinking\*abs) {

abs->makeDrink();

delete abs;

}

void test1() {

//制作咖啡

doWork(new Coffee);

cout << "^^^^^^^^^^^^^^^^" << endl;

doWork(new Tea);

}

int main() {

test1();

return 0;

}

// 虚析构和纯虚析构

/\*

\* 多态使用时，如果子类有属性开辟到堆区，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

\*

\* 虚析构和纯虚析构共性

\* 可以解决父类指针释放子类的现象

\* 都需要有具体的函数实现

\* 区别：

\* 如果是纯虚析构，该类属于抽象类，无法实例化对象

\*

\* 语法：

\* 虚析构语法：

\* virtual ~类名(){ }

\* 纯虚析构语法：

\* 类内声明：virtual ~类名()= 0;

\* 类外实现：类名::~类名(){ }

\*/

class Animal {

public:

Animal() {

cout << "Animal构造函数调用！" << endl;

}

////利用虚析构可以解决 父类指针释放子类对象时不干净的问题

//virtual ~Animal() {

// cout << "Animal虚析构函数调用！" << endl;

//}

//纯虚析构 需要声明，也需要实现

//有了纯虚析构 之后，这个类也属于抽象类，无法实例化对象

virtual ~Animal() = 0;

//纯虚函数

virtual void speak() = 0;

};

Animal:: ~Animal() {

cout << "Animal纯虚析构函数调用！" << endl;

};

class Cat :public Animal {

public:

Cat(string name) {

cout << "Cat构造函数调用！" << endl;

m\_Name = new string(name);

}

virtual void speak() {

cout << \*m\_Name << "猫在说话！" << endl;

}

~Cat(){

if (m\_Name != NULL) {

cout << "Cat析构函数调用!" << endl;

delete m\_Name;

m\_Name = NULL;

}

}

string \*m\_Name;

};

void test1() {

Animal\* animal = new Cat("Tom");

animal->speak();

//父类的指针在析构的时候 不会调用子类中析构函数，导致子类如果有堆区数据，出现内存泄漏

delete animal;

}

int main() {

test1();

return 0;

}

// 案例： 电脑组装

//抽象不同零件类

//抽象CPU类

class CPU {

public:

//抽象的计算函数

virtual void calculate() = 0;

};

//抽象显卡类

class VideoCard {

public:

//抽象的显示函数

virtual void display() = 0;

};

//抽象内存条类

class Memory {

public:

//抽象的存储函数类

virtual void storage() = 0;

};

//电脑类

class Computer {

public:

Computer(CPU\* cpu, VideoCard\* vc, Memory\* mem) {

m\_cpu = cpu;

m\_vc = vc;

m\_mem = mem;

};

//提供工作函数

void work() {

//让零件工作起来，调用接口

m\_cpu->calculate();

m\_vc->display();

m\_mem->storage();

}

//提供析构函数 释放3个电脑零件

~Computer() {

//释放CPU

if (m\_cpu != NULL) {

delete m\_cpu;

m\_cpu = NULL;

}

//释放VideoCard

if (m\_vc != NULL) {

delete m\_vc;

m\_vc = NULL;

}

//释放Memory

if (m\_mem != NULL) {

delete m\_mem;

m\_mem = NULL;

}

}

private:

CPU\* m\_cpu; //CPU零件指针

VideoCard\* m\_vc; //显卡零件指针

Memory\* m\_mem; //内存条零件指针

};

//具体厂商

//Intel厂商

class IntelCPU :public CPU {

public:

virtual void calculate() {

cout << "Intel的CPU开始工作了" << endl;

}

};

class IntelVideoCard :public VideoCard {

public:

virtual void display() {

cout << "Intel的VideoCard开始工作了" << endl;

}

};

class IntelMemory :public Memory {

public:

virtual void storage() {

cout << "Intel的CPU开始工作了" << endl;

}

};

//Lenovo厂商

class LenovoCPU :public CPU {

public:

virtual void calculate() {

cout << "Lenovo的CPU开始工作了" << endl;

}

};

class LenovoVideoCard :public VideoCard {

public:

virtual void display() {

cout << "Lenovo的VideoCard开始工作了" << endl;

}

};

class LenovoMemory :public Memory {

public:

virtual void storage() {

cout << "Lenovo的CPU开始工作了" << endl;

}

};

void test1() {

//电脑零件

CPU\* intelCpu = new IntelCPU;

VideoCard\* intelCard = new IntelVideoCard;

Memory\* intelMem = new IntelMemory;

//创建第一台电脑

cout << "第一台电脑： " << endl;

Computer\* computer1 = new Computer(intelCpu, intelCard, intelMem);

computer1->work();

delete computer1;

//创建第二台电脑

cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << endl;

cout << "第二台电脑： " << endl;

Computer\* computer2 = new Computer(new LenovoCPU, new LenovoVideoCard, new LenovoMemory);

computer2->work();

delete computer2;

}

int main() {

test1();

return 0;

}

// 文件操作

//通过文件操作可以将数据持久化

//C++中对文件操作需要包含头文件<fstream>

/\*

\* 文件分为两种：

\* 文本文件 ——文件以文本的ASII码形式储存在计算机中

\* 二进制文件——文件以文本形式的二进制储存在计算机中，用户一般不能直接读懂他们

\*

\* 操作文件的三大类：

\* 1. ofstream: 写操作

\* 2. ifstream: 读操作

\* 3. fstream: 读写操作

\*/

// 文本文件

//1.写文件

/\*

\* 步骤：

\* 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();

\*/

/\*

\* 打开方式：

\* 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，在创建

\* ios::binary 二进制方式

\*

\* 注意： 文件打开方式可以配合使用 | 操作符

\* 例如：二进制文件方式写文件 ios::binary || ios::out

\*/

void test1() {

//包含头文件 fstream

//创建流对象

ofstream ofs;

//指定打开方式

ofs.open("test1.txt", ios::out);

//写内容

ofs << "姓名：张三" << endl;

ofs << "性别：男" << endl;

ofs << "年龄：18" << endl;

//关闭文件

ofs.close();

}

int main() {

test1();

return 0;

}

//2.读文件

/\* 步骤：

\* 1.包含头文件 #include<fstream>

\* 2.创建流对象 ifstream ifs;

\* 3.打开文件 ifs.open("文件路径",打开方式);

\* 4.四种方式

\* 5.关闭文件 ifs.close();

\*/

void test1() {

//包含头文件，创建流对象

ifstream ifs;

//打开文件 并判断是否打开成功

ifs.open("test1.txt", ios::in);

if (!ifs.is\_open()) {

cout << "文件打开失败" << endl;

return;

}

//读数据

//第一种

/\*char buf1[1024] = { 0 };

while (ifs >> buf1) {

cout << buf1 << endl;

}\*/

//第二种

/\*char buf2[1024] = { 0 };

while (ifs.getline(buf2, sizeof(buf2))) {

cout << buf2 << endl;

}\*/

//第三种

/\*string buf3;

while (getline(ifs, buf3)) {

cout << buf3 << endl;

}\*/

//第四种

//char buf4;

//while ((buf4 = ifs.get()) != EOF) { //EOF end of file

// cout << buf4;

//}

//关闭文件

ifs.close();

}

int main() {

test1();

return 0;

}

// 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

//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： ofstream& write(const char \* buffer, int len);

\* 参数介绍： 字符指针buffer指向内存中的一段存储空间，len是读写的字节数

\*/

class Person {

public:

char m\_Name[64];//姓名

int m\_Age;

};

void test3() {

//包含头文件，创建流对象

ofstream ofs;

//打开文件,写文件

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

//写文件

Person p = { "张三",18 };

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

//关闭文件

ofs.close();

}

int main() {

test3();

return 0;

}

//读文件

//二进制方式读文件主要是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指向内存中的一段存储空间，len是读写的字节数

class Person {

public:

char m\_Name[64];//姓名

int m\_Age;

};

void test4() {

//包含头文件，创建流对象

ifstream ifs;

//打开文件 判断文件是否打开成功

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

if (!ifs.is\_open()) {

cout << "文件打开击败" << endl;

return;

}

//读文件

Person p;

ifs.read((char\*)&p, sizeof(Person));

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

//关闭文件

ifs. close();

}

int main() {

test4();

return 0;

}