Class test

{

Int I; //value

Int \*j; //reference, handle

Public: //构造

Test(int a, int b);

Test(int a); /\*这程序是错的，错在哪里？

~Test(); 者产生fly pointer\*/

Test();

};

Test::Test(int a) //这产生fly pointer

{

This->I = a；

This->j = NULL; 没有用的需要放NULL

}

Test::Test(int a, int b)

{

This->I = a;

This ->j = new int(b)

}

Void main()

{

Test t(1,2);

Test \*p = new Test(1,2);

//t 是局部变量。

}

Test :: test()

{

J = NULL;

Delete j; //delete member j

}

Void Test :: fun()

{

Int\* p = new int;

//….

Delete p;

}

构造参数：

默认有一个，也可以没有

会重载

析构函数：析构唯一的

构造中如果有new肯定有析构

构造中没有new肯定没有析构（destructor）这不对。

没有多个析构的概念。

Delete把指针向的内存删掉。

在类所有的new是在类里面。

//clean 拿来回收

Void main()

{

//reference (引用) ： 是一个安全的指针

Int I = 10;

Int j = 20;

Int \*p;

P = &I;

P = &j; //这指针调用不安全。

//引用的写法

Int& r = i; 引用必须初始化，这说r 是i的引用

R++;

Cout << I << endl;

R = j;

引用解决指针不安全的问题

1. 野指针
2. 指针到处的问题。

}

Void fun(int i)

{

I++;

}

Void fun (int& i){ //pass by value

I++;

}

Int main()

{

Int m = 10;

Fun(m); //转值 pass by value

Fun(&m); n//转地址 pass by address

Cout << m << endl

}

//pointer is ugly but clear

Java一次编译到处执行。

C++一次写到处编译

Pass by value VS pass by address （pointer, reference）

Class Test

{

Int I;

Int \*j;

Public:

Test(int a, int b);

//copy constructor

Bitwise copy vs logical copy

Bitwise copy 是按照位拷贝，不管什么一个比特一个比特拷贝（钱拷贝）

Logical copy 是逻辑拷贝，或者深拷贝

~Test();

Test(Test& t);

};

//逻辑拷贝

Test :: Test(Test& t)

{

This-> I = t.I;

This->j = new int(\*t.j);

}

Test ：： ~Test()

{

Delete j;

}

Test :: Test(int a, int b)

{

I = a;

J = new int (b);

}

Int main()

{

Test t1(1,2);

Test t2(t1); //copy constructor

}

不可以直接写Free, 重复释放不允许应该

If (p != NULL)

Free p

Else

P = NULL

拷贝构造：

/\*指针最常见的错误

1. Fly pointer(飞指针)
2. 重复delete / free
3. 钱拷贝bitwise copy, memcpy 内存拷贝
4. 内存泄漏 Memory leak
5. Return address of local variable 返回局部变量的地址

重定义（redefinition）

C++的super是 Pet::Speak()

Public:

Void speak()//redefinition

{

Pet::Speak();

Cout << “miao” << endl;

}

Void sleep(){

}

Class Car{

Engine e;

Int other;

Public:

Void run(){

e.run();

cout << “other run” endl;

}

每创建父类先调用父类构造函数，先父类，后子类

构造和析构相反

析构先子类，后父类

Cat(int aage, char \*aname, int atype):Pet(aagae,aname){

Cout << “construct cat”<<endl;

}

Class Test

{

Public:

Int i;

Int j;

Public:

Test(int a) :: j(a). i(j){}

//产生垃圾输出

}

C++继承

Class base1{

Public:

Void f(){}

};

Class base2{  
public:

Void h(){}

};

Class derived

{};

Int main(){

Derived d;

//多种继承

d.f();

d.h();

return 0;

}

Binding(绑定)：将函数的一次调用，

前绑定（early binding）:

Later binding, or runtime binding, dynamic binding (后绑定，运行绑定，动态绑定)