

三、C++学习笔记—核心编程

本阶段，将对C++面向对象编程技术做详细学习，深入C++中的核心和精髓

3.1 内存分区模型

C++程序在执行时，将内存大方向划分为**4个区域**

- 代码区：存放函数体的二进制代码，由操作系统进行管理的
- 全局区：存放全局变量和静态变量以及常量
- 栈区：由编译器自动分配释放，存放函数的参数值，局部变量等
- 堆区：由程序员分配和释放，若程序员不释放，程序结束时由操作系统回收

内存四区意义：

不同区域存放的数据，赋予不同的生命周期，给我们更大的灵活编程

3.1.1 程序运行前

在程序编译后，生成了**exe可执行程序**，**未执行该程序前**分为两个区域

代码区：

存放 CPU 执行的机器指令

代码区是**共享的**，共享的目的是对于频繁被执行的程序，只需要在内存中有一份代码即可

代码区是**只读的**，使其只读的原因是防止程序意外地修改了它的指令

全局区：

全局变量和静态变量存放在此。

全局区还包含了**常量区**，**字符串常量**和**其他常量**也存放在此。

该区域的数据在程序结束后由操作系统释放。



```

1  #include<iostream>
2  using namespace std;
3  //全局变量
4  int g_a = 10;
5  int g_b = 10;
6
7  //const修饰的全局变量，全局常量
8  const int c_g_a = 10;
9  const int c_g_b = 10;
10 int main()
11 {
12
13     //全局区
14
15     //全局变量、静态变量、常量
16
17     //创建普通局部变量
18     int a = 10;
19     int b = 10;
20     cout << "局部变量a的地址为: \t" << &a << endl;
21     cout << "局部变量b的地址为: \t" << &b << endl;
22
23     cout << "全局变量g_a的地址为: \t" << &g_a << endl;
24     cout << "全局变量g_b的地址为: \t" << &g_b << endl;
25
26     //静态变量 在普通变量的前面加上static，属于静态变量，也会放在全局区域中
27     static int s_a = 10;
28     cout << "静态变量s_a的地址为: \t" << &s_a << endl;
29
30     //常量
31     //字符串常量
32     cout << "字符串常量的地址为: \t" << &"hell world" << endl;
33
34     //const修饰的变量
35     //const修饰的全局变量，const修饰的局部变量
36     cout << "全局常量c_g_a的地址为: \t" << &c_g_a << endl;
37     cout << "全局常量c_g_b的地址为: \t" << &c_g_b << endl;
38
39     const int c_l_a = 10; //c-const g-global l=local
40     const int c_l_b = 10; //c-const g-global l=local
41     cout << "局部常量c_l_a的地址为: \t" << &c_l_a << endl;
42     cout << "局部常量c_l_b的地址为: \t" << &c_l_b << endl;
43     system("pause");
44     return 0;
45
46 }

```

1 | **总结: **

- C++中在程序运行前分为全局区和代码区
- 代码区特点是共享和只读
- 全局区中存放全局变量、静态变量、常量

- 常量区中存放 const修饰的全局常量 和 字符串常量

3.1.2 程序运行后

栈区：

由编译器自动分配释放, **存放函数的参数值,局部变量等**

注意事项： 不要返回局部变量的地址，栈区开辟的数据由编译器自动释放

示例：

```
1  #include<iostream>
2  using namespace std;
3
4  //栈区注意事项 --- 不要返回局部变量的地址
5  //栈区的数据由编译器管理开辟和释放
6
7  int* func(int b)  //形参数据也会放在栈区
8  {
9      int b = 100;
10     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
11     return &a; //返回局部变量的地址
12 }
13
14 int main()
15 {
16
17     //接收func函数的返回值
18     int * p = func(1);
19     cout << *p << endl; //10,第一次可以打印正确的数字，是因为编译器做了保留
20     cout << *p << endl; // 第二次这个数据就不在保留了
21
22     system("pause");
23     return 0;
24 }
25 }
```

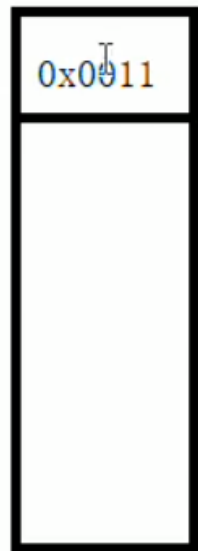
堆区：

由程序员分配释放,若程序员不释放,程序结束时由操作系统回收

在C++中主要利用new在堆区开辟内存

栈

int * p



堆区

0x0011



示例:

```
1  #include<iostream>
2  using namespace std;
3
4  int * func()
5  {
6      //int a = 10;
7      //return &a; //不要返回局部变量的地址，栈区开辟的数据由编译器自动释放
8
9      //利用new关键字， 可以将数据开辟到堆区
10     //指针，本质也是局部变量，放在栈上，指针保存的数据是放在堆区
11     int * p = new int(10);
12     return p;
13 }
14
15 int main()
16 {
17
18     //在堆区开辟数据
19     int *p = func();
20     cout << *p << endl;
21
22     system("pause");
23     return 0;
24 }
25 }
```

总结:

堆区数据由程序员管理开辟和释放

堆区数据利用new关键字进行开辟内存

3.1.3 new操作符

C++中利用new操作符在堆区开辟数据

堆区开辟的数据，由程序员手动开辟，手动释放，释放利用操作符 delete

语法：new 数据类型

利用new创建的数据，会返回该数据对应的类型的指针

示例1：基本语法

```
1  #include<iostream>
2  using namespace std;
3
4  //1、new的基本语法
5  int * func()
6  {
7      //在堆区创建整型数据
8      int *p = new int(10); // new返回的是，该数据类型的指针
9
10     return p;
11 }
12
13 int main()
14 {
15
16     int * p = func();
17     cout << *p << endl; //堆区的数据，由程序员管理开辟，程序员管理释放
18     delete p;
19     //cout << *p << endl; //内存已经被释放，再次访问就是非法操作，会报错
20     system("pause");
21     return 0;
22 }
23 }
```

示例2：开辟数组

```
1  #include<iostream>
2  using namespace std;
3
4  //2、在堆区利用new开辟内存
5  void test02()
6  {
7      //创建10个整型数据的数组，在堆区
8      int * arr = new int[10]; //10代表数组有10个元素
9      for (int i = 0; i < 10; i++)
10     {
11         arr[i] = i + 100; //给10个元素赋值：100-109
12     }
13     for (int i = 0; i < 10; i++)
14     {
15         cout << arr[i] << endl;
```

```

16     }
17     //释放堆区数组
18     delete[] arr; //释放数组的时候，需要加[]才可以
19 }
20
21 int main()
22 {
23     test02();
24     system("pause");
25     return 0;
26 }
27 }

```

3.2 引用

3.2.1 引用的基本使用

作用：给变量起别名

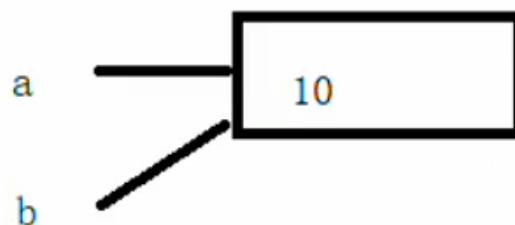
语法：数据类型 &别名 = 原名

```
int a = 10;
```

4个字节

引用：给变量起别名

语法：数据类型 &别名 = 原名



```
int &b = a;
```

```
b = 20;
```

```
cout << a << endl; ??? 也是20
```

示例：

```

1  #include<iostream>
2  using namespace std;
3
4  int main()
5  {

```

```

6 //引用基本语法
7 //数据类型 &别名 = 原名
8
9 int a = 10;
10 //创建引用
11 int &b = a;
12 cout << "a=" << a << endl;
13 cout << "b=" << b << endl;
14
15 b = 100;
16 cout << "a=" << a << endl;
17 cout << "b=" << b << endl;
18
19 system("pause");
20 return 0;
21
22 }

```

3.2.2 引用注意事项

- 引用必须初始化
- 引用在初始化后，不可以改变

```

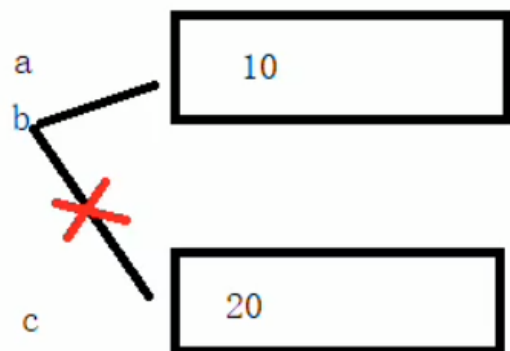
int a = 10;
int &b = a;

```

1、引用必须要初始化

```
int &b; //错误的
```

2、引用一旦初始化后，就不可以更改了



示例：

```

1 #include<iostream>
2 using namespace std;
3
4 int main()
5 {
6
7     int a = 10;
8
9     //1、引用必须初始化
10    //int &b; //错误，必须要初始化
11    int &b = a;
12
13    //2、引用在初始化后，不可以改变

```

```

14     int c = 20;
15     b = c; //赋值操作，而不是更改引用
16     cout << "a=" << a << endl;
17     cout << "b=" << b << endl;
18     cout << "c=" << c << endl;
19
20     system("pause");
21     return 0;
22
23 }

```

3.2.3 引用做函数参数

作用：函数传参时，可以利用引用的技术让形参修饰实参

优点：可以简化指针修改实参

示例：

```

1  #include<iostream>
2  using namespace std;
3
4  //交互函数
5  //1、值传递
6  void mySwap01(int a, int b)
7  {
8      int temp = a;
9      a = b;
10     b = temp;
11     cout << "swap01a=" << a << endl;
12     cout << "swap01b=" << b << endl;
13 }
14
15 //2、地址传递
16 void mySwap02(int *a, int *b)
17 {
18     int temp = *a;
19     *a = *b;
20     *b = temp;
21
22 }
23
24 //3、引用传递
25 void mySwap03(int &a, int &b)
26 {
27     int temp = a;
28     a = b;
29     b = temp;
30 }
31
32 int main()

```



```

33 {
34
35     int a = 10;
36     int b = 20;
37
38     //mySwap01(a, b); //值传递，形参并不能修饰实参，实参并未发生改变
39     cout << "a=" << a << endl;
40     cout << "b=" << b << endl;
41
42     //mySwap02(&a, &b); //地址传递，形参会修饰实参的
43     cout << "02a=" << a << endl;
44     cout << "02b=" << b << endl;
45
46     mySwap03(a,b); //引用传递，形参会修改实参
47     cout << "03a=" << a << endl;
48     cout << "03b=" << b << endl;
49
50     system("pause");
51     return 0;
52
53 }

```

总结：通过引用参数产生的效果同按地址传递是一样的。引用的语法更清楚简单

3.2.4 引用做函数返回值

作用：引用是可以作为函数的返回值存在的

注意：不要返回局部变量引用

用法：函数调用作为左值

示例：

```

1  #include<iostream>
2  using namespace std;
3
4  //引用做函数的返回值
5
6  //1、不要返回局部变量的引用
7  int& test01()
8  {
9      int a = 10; //局部变量存在在四区中的，栈区
10     return a; //引用的反正返回
11 }
12
13 //2、函数的调用可以作为左值
14 int& test02()
15 {
16     static int b = 10; //静态变量，放在在全局区，全局区上的数据在程序结束后系统释放
17     return b;
18 }
19
20 int main()

```

```

21 {
22
23     //int &ref = test01();
24     //cout << "ref=" << ref << endl; //第一次结果正确, 是因为编译器做了保留
25     //cout << "ref=" << ref << endl; //第二次结果错误, 是因为a的内存已经释放
26
27     int &ref2 = test02();
28     cout << "ref2=" << ref2 << endl;
29     cout << "ref2=" << ref2 << endl; //10
30
31     //a=1000的操作, 原名赋值1000, 再使用别名ref2调用 如果函数的返回值是引用, 这个函数调用可以作为左
    值
32     test02() = 1000;
33     cout << "ref2=" << ref2 << endl;
34     cout << "ref2=" << ref2 << endl; //1000
35
36     system("pause");
37     return 0;
38
39 }

```

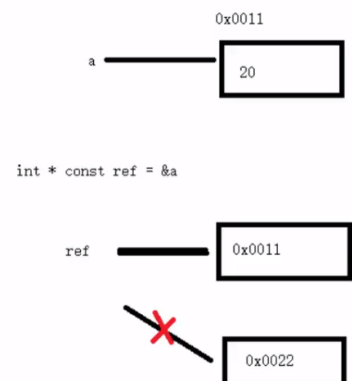
3.2.5 引用的本质

本质: 引用的本质在c++内部实现是一个**指针常量**。

```

1 //发现是引用, 转换为 int* const ref = &a;          引用的本质 就是一个指针常量
2 void func(int& ref){                                引用一旦初始化后, 就不可以发生改变
3     ref = 100; // ref是引用, 转换为*ref = 100
4 }
5 int main(){
6     int a = 10;
7
8     //自动转换为 int* const ref = &a; 指针常量是指针指向不可改, 也说明为什么引用不可更改
9     int& ref = a;
10    ref = 20; //内部发现ref是引用, 自动帮我们转换为: *ref = 20;
11
12    cout << "a:" << a << endl;
13    cout << "ref:" << ref << endl;
14
15    func(a);
16    return 0;
17 }

```



示例:

```

1 #include<iostream>
2 using namespace std;
3
4 //引用的本质
5 void func(int& ref)
6 {
7     ref = 100; // ref是引用, 转换为*ref = 100
8 }
9
10 int main()

```

```

11 {
12     int a = 10;
13
14     //自动转换为 int * const ref = &a; 指针常量是指针指向不可改, 也说明为什么引用不可更改
15     int& ref = a; //给a起了一个别名
16     ref = 20; //内部发现ref是引用, 自动帮我们转换为 *ref = 20;
17
18     cout << "a=" << a << endl;
19     cout << "ref=" << ref << endl;
20
21     func(a);
22     system("pause");
23     return 0;
24
25 }

```

3.2.6 常量引用

作用：常量引用主要用来修饰形参，防止误操作

在函数形参列表中，可以加**const修饰形参**，防止形参改变实参

示例：

```

1  #include<iostream>
2  using namespace std;
3
4  //常量引用
5  //使用场景：用来修饰形参，防止误操作
6
7  //打印数据函数
8  void showValue(const int & val) //使用引用的方式接收
9  {
10     //val = 1000;
11     cout << "val=" << val << endl;
12 }
13
14 int main()
15 {
16     //int a = 10;
17
18     //加上const之后, 编译器将代码修改 int temp = 10; const int & ref = temp;
19     //const int & ref = 10; //引用必须引一块合法的内存空间
20     //ref = 20; //加入const之后变为只读, 不可以修改
21
22     int a = 100;
23     showValue(a);
24     cout << "a=" << a << endl;
25
26     system("pause");
27     return 0;

```

3.3 函数提高

3.3.1 函数默认参数

在C++中，函数的形参列表中的形参是可以有默认值的。

语法：返回值类型 函数名 (参数= 默认值) {}

示例：

```
1  #include<iostream>
2  using namespace std;
3
4  //函数默认参数
5  //如果我们自己传入数据吗，就是自己的数据，如果没有，那么就用默认值
6  //语法： 返回值类型 函数名 (形参=默认值) {}
7
8  int func(int a, int b=20, int c=30)
9  {
10     return a + b + c;
11 }
12
13 //注意事项：
14 //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
15 int fun2(int a, int b=10, int c) //c此时也必须得有
16 {
17     return a + b + c;
18 }
19
20 //2、如果函数的声明有了默认参数，函数的实现就不能有默认参数
21 //声明和实现智能有一个有默认参数
22 int func3(int a=10, int b=10); //声明
23 int func3(int a=10, int b=10)
24 {
25     return a + b;
26 }
27
28 int main()
29 {
30
31     int sum = func(10); //
32     cout << "sum=" << sum << endl;
33     cout << func(10, 30)<<endl; //70
34
35     cout<<func3(10, 10)<<endl; //会报错，重定义默认参数
36
37
38     system("pause");
39     return 0;
```

3.3.2 函数占位参数

C++中函数的形参列表里可以有占位参数，用来做占位，调用函数时必须填补该位置

语法： 返回值类型 函数名 (数据类型){}

在现阶段函数的占位参数存在意义不大，但是后面的课程中会用到该技术

示例：

```

1  #include<iostream>
2  using namespace std;
3
4  //占位参数
5  //返回值类型  函数名 (数据类型) {}
6
7  //目前阶段的占位参数，我们还用不到，后面学习中会用到
8  //占位参数 还可以有默认参数
9  void func(int a,int = 10)
10 {
11     cout << "this is func " << endl;
12 }
13
14 int main()
15 {
16
17     func(10);
18     system("pause");
19     return 0;
20
21 }
```

3.3.3 函数重载

3.3.3.1 函数重载概述

作用：函数名可以相同，提高复用性

函数重载满足条件：

- 同一个作用域下
- 函数名称相同
- 函数参数类型不同 或者 个数不同 或者 顺序不同

注意：函数的返回值不可以作为函数重载的条件

示例：

```

1  #include<iostream>
2  using namespace std;
3
4  //函数重载
5  //可以让函数名相同，提高复用性
6
7  //函数重载的满足条件：
8  //1、同一个作用域下
9  //2、函数名称相同
10 //3、函数的参数类型不同，或者个数不同，或者顺序不同
11
12 //1
13 void func()
14 {
15     cout << "func的调用" << endl;
16 }
17 //2
18 void func(int a)
19 {
20     cout << "func(int a)的调用!" << endl;
21 }
22 //3
23 void func(double a)
24 {
25     cout << "func(double a)的调用!" << endl;
26 }
27 //4
28 void func(int a, double b)
29 {
30     cout << "func(int a, double b)的调用!" << endl;
31 }
32 //5
33 void func(double a, int b)
34 {
35     cout << "func(double a, int b)的调用!" << endl;
36 }
37
38 //注意事项：
39 //函数的返回值不可以作为函数重载的条件
40 //就是不能和上面相同的重复
41 //void func(double a, int b)
42 //{
43 //    cout << "func(double a, int b)的调用!" << endl;
44 //}
45
46 int main()
47 {
48     //func();
49     //func(10);
50     //func(3.14);
51     //func(10, 3.14);
52     func(3.13,10);
53

```

```

54     system("pause");
55     return 0;
56
57 }

```

3.3.3.2 函数重载注意事项

- 引用作为重载条件
- 函数重载碰到函数默认参数

示例：

```

1  #include<iostream>
2  using namespace std;
3
4  //函数重载的注意事项
5  //1、引用作为重载的条件
6  void func(int &a) //int &a =10; 不合法
7  {
8      cout << "func(int &a)调用" << endl;
9  }
10
11 void func(const int &a) //只读状态 const int &a =10;合法
12 {
13     cout << "func(const int &a)调用" << endl;
14 }
15
16
17 //2、函数重载碰到默认参数
18
19 void func2(int a,int b=10)
20 {
21     cout << "func2(int a,int b)调用" << endl;
22 }
23
24 void func2(int a)
25 {
26     cout << "func2(int a)调用" << endl;
27 }
28
29
30 int main()
31 {
32     //int a = 10;//变量
33     //func(a);//调用没有加const的。
34
35     //func(10);
36
37     func2(10); //此时，上面都能调，编译器傻了。当函数重载碰到默认参数，出现二义性，报错，尽量避免这种情况
38

```

```
39     system("pause");
40     return 0;
41
42 }
```

3.4 类和对象

C++面向对象的三大特性为：封装、继承、多态

C++认为万事万物都皆为对象，对象上有其属性和行为

例如：

人可以作为对象，属性有姓名、年龄、身高、体重...，行为有走、跑、跳、吃饭、唱歌...

车也可以作为对象，属性有轮胎、方向盘、车灯...，行为有载人、放音乐、放空调...

具有相同性质的对象，我们可以抽象称为类，人属于人类，车属于车类

3.4.1 封装

3.4.1.1 封装的意义

封装是C++面向对象三大特性之一

封装的意义：

- 将属性和行为作为一个整体，表现生活中的事物
- 将属性和行为加以权限控制

封装意义一：

在设计类的时候，属性和行为写在一起，表现事物

语法： `class 类名{ 访问权限: 属性 / 行为 };`

示例1：设计一个圆类，求圆的周长

示例代码：

```
1  #include<iostream>
2  using namespace std;
3
4  //圆周率
5  const double PI = 3.14;
6  //设计一个圆类，求圆的周长
7  //圆求周长的公式： 2 * PI * 半径
8
9  //class 代表设计一个类，类后面紧跟着的就是类名称
10 class Circle
11 {
12     //访问权限
13     //公共权限
14 public:
```



```

15 //属性----变量
16 //半径
17 int m_r;
18
19 //行为----函数, 公式等
20 //获取圆的周长
21 double calculateZC()
22 {
23     return 2 * PI * m_r;
24 }
25
26 };
27
28 int main()
29 {
30     //通过圆类, 创建具体的圆 (对象)
31     //实例化 (通过一个类, 创建一个对象的过程)
32     Circle c1;
33     //给圆对象 的属性进行赋值
34     c1.m_r = 10;
35
36     // 2*PI*10=62.8
37     cout << "圆的周长为: " << c1.calculateZC() << endl;
38
39
40     system("pause");
41     return 0;
42
43 }

```

示例2: 设计一个学生类, 属性有姓名和学号, 可以给姓名和学号赋值, 可以显示学生的姓名和学号

示例2代码:

```

1  #include<iostream>
2  #include<string>
3  using namespace std;
4
5  //学生类
6  //设计一个学生类, 属性有姓名和学号,
7  //可以给学生和学号赋值, 可以显示学生的姓名和学号
8
9  class Student
10 {
11 public://访问权限-公共权限
12
13     // 类中的属性和行为, 我们统一称为 成员
14     // 属性    成员属性    成员变量
15     // 行为    成员函数    成员方法
16
17     //属性
18     string m_Name;
19     int m_Id;

```

```

20
21 //行为
22 //显示姓名和学号
23 void showStudent()
24 {
25     cout << "姓名: " << m_Name << endl;
26     cout << "学号: " << m_Id << endl;
27
28 }
29
30 //给姓名赋值
31 void setName(string name)
32 {
33     m_Name = name;
34 }
35 //给学号赋值
36 void set_m_Id(int Id)
37 {
38     m_Id = Id;
39 }
40 };
41
42
43 int main()
44 {
45
46     //创建一个具体的学生（对象）实例化
47     Student s1;
48
49     //给s1对象，进行属性赋值操作
50     //s1.m_Name = "贾继康";
51     s1.setName("张三");
52     //s1.m_Id = 123;
53     s1.set_m_Id(1);
54
55     //显示学生信息
56     s1.showStudent();
57
58     system("pause");
59     return 0;
60
61 }

```

封装意义二:

类在设计时，可以把属性和行为放在不同的权限下，加以控制

访问权限有三种:

1. public 公共权限
2. protected 保护权限
3. private 私有权限

示例:

```

1  #include<iostream>
2  #include<string>
3  using namespace std;
4
5  //访问权限
6  //三种
7  //公共权限  public          成员 类内可以访问, 类外可以访问
8  //保护权限  protected      成员 类内可以访问, 类外不可以访问 儿子可以访问父亲中的保护内容
9  //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私有内容
10
11 class Person
12 {
13 public:
14     //公共权限
15     string m_Name;
16 protected:
17     //保护权限
18     string c_Car; //汽车
19
20 private:
21     // 私有权限
22     int m_Password; // 银行卡密码
23 public: //private, 或者protected在类内都可以访问的
24     void func()
25     {
26         m_Name = "张三";
27         c_Car = "拖拉机";
28         m_Password = 123456;
29     }
30
31 };
32
33 int main()
34 {
35     Person p1; // 实例化具体对象
36     p1.m_Name = "李四";
37     p1.c_Car = "拖拉机"; //保护权限内容, 在类外访问不到
38     p1.m_Password = 12345; // 私有权限内容, 类外访问不到
39
40
41     system("pause");
42     return 0;
43
44 }

```

3.4.1.2 struct和class区别

在C++中 struct和class唯一的区别就在于 默认访问权限不同

区别:

- struct 默认权限为公共
- class 默认权限为私有

```
1  #include<iostream>
2  #include<string>
3  using namespace std;
4
5  //struct 和 class 区别
6  //struct 默认权限是 公共 public
7  //class 默认权限是 私有 private
8
9  class C1
10 {
11     int m_A; // 默认权限 是私有
12 };
13 struct C2
14 {
15     int m_A; //默认权限 是公共
16 };
17
18 int main()
19 {
20
21     C1 c1; //实例化对象
22     //c1.m_A=100; // 在class里默认权限,私有,此处报错
23
24     C2 c2;
25     c2.m_A = 100; //struct默认权限公共的,因此可以访问
26
27     system("pause");
28     return 0;
29 }
30 }
```

3.4.1.3 成员属性设置为私有

优点1: 将所有成员属性设置为私有,可以自己控制读写权限

优点2: 对于写权限,我们可以检测数据的有效性

示例:

```
1  #include<iostream>
2  #include<string>
3  using namespace std;
4
5
6  // 成员属性设置为私有
7  // 1、可以自己控制读写权限
8  // 2、对于写可以检测数据的有效性
9
```

```

10 //设计人类
11 class Person
12 {
13 public:
14
15     //姓名--可读可写
16     //设置姓名
17     void setName(string name)
18     {
19         m_Name = name;
20     }
21     //获取姓名
22     string getName()
23     {
24         return m_Name;
25     }
26
27     //年龄--只读  --改成可读可写（如果想修改，年龄的范围必须是0-150范围）
28     //获取年龄
29     int getAge()
30     {
31         //m_Age = 0; // 初始化为0
32         return m_Age;
33     }
34     //设置年龄
35     void setAge(int age)
36     {
37         if (age<0||age>150)
38         {
39             m_Age = 0;
40             cout << "您这个老妖精" << endl;
41             return;
42         }
43         m_Age = age;
44     }
45
46     //情人--只写
47     void setLover(string lover)
48     {
49         m_Lover = lover;
50     }
51
52 private: //设置为私有化
53     //姓名  可读可写
54     string m_Name;
55     //年龄  只读
56     int m_Age;
57     // 情人  只读
58     string m_Lover;
59
60 };
61
62 int main()

```

```

63 {
64     //实例化对象
65     Person p;
66     p.setName("贾继康");
67     cout << "姓名为: " << p.getName() << endl;
68
69     p.setAge(1000);
70     cout << "年龄为: " << p.getAge() << endl;
71
72     p.setLover("苍井空");
73     //cout << "情人为: " << p.m_Lover << endl; //只设置, 所以不可以访问的
74
75     system("pause");
76     return 0;
77
78 }

```

练习案例1: 设计立方体类

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。

```

1  #include<iostream>
2  #include<string>
3  using namespace std;
4
5  //设计一个立方体
6  //1、创建立方体类
7  //2、设计属性
8  //3、设计行为: 获取立方体面积和体积
9  //4、分别利用全局函数和成员函数 判断两个立方体是否相等
10
11
12 class Cube
13 {
14
15 public:
16
17     //设置长
18     void setL(int l)
19     {
20         m_L = l;
21
22     }
23     //获取长
24     int getL()
25     {
26         return m_L;
27
28     }
29

```

```

30 //设置宽
31 void setW(int w)
32 {
33     m_W = w;
34
35 }
36
37 //获取宽
38 int getW()
39 {
40     return m_W;
41
42 }
43 //设置高
44 void setH(int h)
45 {
46     m_H = h;
47
48 }
49 //获取高
50 int getH()
51 {
52     return m_W;
53
54 }
55
56 //获取立方体的面积
57 int calculates()
58 {
59     return 2 * m_L*m_W + 2 * m_W*m_H + 2 * m_L*m_H;
60 }
61 //获取立方体的体积
62 int calculateV()
63 {
64     return m_L * m_H*m_W;
65 }
66
67 //利用成员函数判断两个立方体是否相等
68 bool isSameByClass(Cube &c)
69 {
70     if (m_L == c.getL() && m_W == c.getW() && m_H == c.getH())
71     {
72         return true;
73     }
74     return false;
75
76 }
77
78 private: //属性
79
80 int m_L; //长
81 int m_W; //宽
82 int m_H; //高

```

```
83
84 };
85
86 //全局函数
87 //利用全局函数判断 两个立方体是否相等
88 bool isSame(Cube &c1, Cube &c2)// 引用方式
89 {
90     if (c1.getL() == c2.getL() && c1.getW()==c2.getW() && c1.getH()==c2.getH())
91     {
92         return true;
93     }
94     return false;
95 }
96
97 int main()
98 {
99
100     //创建立方体对象
101     Cube c1;
102     c1.setL(10);
103     c1.setW(10);
104     c1.setH(10);
105
106     cout << "c1的面积为: " << c1.calculateS() << endl;
107     cout << "c1的体积为: " << c1.calculateV() << endl;
108
109     //创建第二个立方体对象
110
111     Cube c2;
112     c2.setL(10);
113     c2.setW(10);
114     c2.setH(10);
115
116     bool ret = isSame(c1, c2);
117     if (ret)
118     {
119         cout << "c1和c2是相等的" << endl;
120     }
121     else
122     {
123         cout << "c1和c2是不相等的" << endl;
124     }
125
126
127     ret = c1.isSameByClass(c2);
128     if (ret)
129     {
130         cout << "这是成员函数判断的结果: c1和c2相等" << endl;
131     }
132     else
133     {
134         cout << "这是成员函数判断的结果: c1和c2不相等" << endl;
135     }
```



```

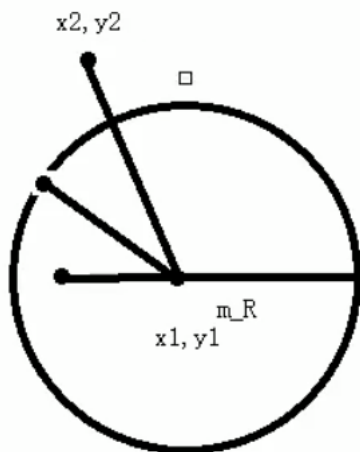
136
137     system("pause");
138     return 0;
139 }

```

练习案例2：点和圆的关系

设计一个圆形类（Circle）， 和一个点类（Point）， 计算点和圆的关系。

注意：点到圆心的距离—欧氏距离



点和圆关系判断

点到圆心的距离 == 半径 点在圆上

点到圆心的距离 > 半径 点在圆外

点到圆心的距离 < 半径 点在圆内

点到圆心的距离 ???

$$(x1 - x2)^2 + (y1 - y2)^2$$

和 m_R^2 对比

程序布局第一种方式：

```

1  #include<iostream>
2  #include<string>
3  using namespace std;
4
5  #include "circle.h"
6  #include "point.h"
7
8  //点和圆关系案例
9
10 //点类
11 //class Point
12 //{
13 //public:
14 //
15 //    //设置x
16 //    void setX(int x)
17 //    {
18 //        m_X = x;
19 //    }
20 //
21 //    //获取x
22 //    int getX()
23 //    {
24 //        return m_X;
25 //    }
26 //

```

```
27 // //设置Y
28 // void setY(int y)
29 // {
30 //     m_Y = y;
31 // }
32 // //获取Y
33 // int getY()
34 // {
35 //     return m_Y;
36 // }
37 //
38 //
39 //private: //属性
40 // int m_X;
41 // int m_Y;
42 //
43 //};
44
45
46 ////圆类
47 //class Circle
48 //{
49 //public:
50 //
51 // //设置半径
52 // void setR(int r)
53 // {
54 //     m_R = r;
55 // }
56 // //获取半径
57 // int getR()
58 // {
59 //     return m_R;
60 // }
61 // //设置圆心
62 // void setCenter(Point center)
63 // {
64 //     m_Center = center;
65 // }
66 // //获取圆心
67 // Point getCenter()
68 // {
69 //     return m_Center;
70 // }
71 //
72 //private://属性
73 // int m_R;//半径
74 //
75 // //核心内容1: 在类中, 可以让另一个类 作为本类中的成员
76 // Point m_Center; // 圆心
77 //
78 //};
79
```

```

80 //判断点和圆的关系
81 void isInCircle(Circle &c, Point &p)
82 {
83     //计算两点之间的距离 平方
84     int distance =
85         (c.getCenter().getX() - p.getX())*(c.getCenter().getX() - p.getX()) +
86         (c.getCenter().getY() - p.getY())*(c.getCenter().getY() - p.getY());
87
88     //计算半径的平方
89     int rDistance = c.getR()*c.getR();
90
91     //判断
92     if (distance==rDistance)
93     {
94         cout << "点在圆上" << endl;
95     }
96     else if (distance>rDistance)
97     {
98         cout << "点在圆外" << endl;
99     }
100    else
101    {
102        cout << "点在圆内" << endl;
103    }
104 }
105
106 int main()
107 {
108     //实例化对象
109     //创建圆
110     Circle c;
111     c.setR(10);
112     Point center;
113     center.setX(10);
114     center.setY(0);
115     c.setCenter(center);
116
117     //创建点
118     Point p;
119     p.setX(10);
120     p.setY(10);
121
122     //判断关系
123     isInCircle(c, p);
124
125     system("pause");
126     return 0;
127 }

```

程序布局第二种方式:

```

1 //1、point.h
2 #pragma once

```

```

3  #include<iostream>
4  using namespace std;
5
6  //函数的声明和变量的声明
7  //点类
8  class Point
9  {
10 public:
11
12     //设置x
13     void setX(int x);
14     //获取x
15     int getX();
16     //设置y
17     void setY(int y);
18     //获取y
19     int getY();
20 private: //属性
21     int m_X;
22     int m_Y;
23
24 };

```

```

1  //point.cpp
2  #include "point.h"
3
4  //点类
5  //只需要留住函数所有的实现
6
7  //设置x
8  void Point:: setX(int x) //需要告诉Point作用域下的成员函数
9  {
10     m_X = x;
11 }
12
13 //获取x
14 int Point::getX()
15 {
16     return m_X;
17 }
18
19 //设置y
20 void Point::setY(int y)
21 {
22     m_Y = y;
23 }
24 //获取y
25 int Point:: getY()
26 {
27     return m_Y;
28 }

```

```

1 //3、circle.h
2 #pragma once
3 #include<iostream>
4 using namespace std;
5
6 #include "point.h"
7
8 //圆类
9 class Circle
10 {
11 public:
12
13     //设置半径
14     void setR(int r); //函数的声明
15
16     //获取半径
17     int getR();
18
19     //设置圆心
20     void setCenter(Point center);
21
22     //获取圆心
23     Point getCenter();
24
25 private://属性
26     int m_R;//半径
27
28     //核心内容1: 在类中, 可以让另一个类 作为本类中的成员
29     Point m_Center; // 圆心
30
31 };

```

```

1 //4、circle.cpp
2
3
4 #include "circle.h"
5 //圆类实现
6     //设置半径
7     void Circle::setR(int r) // 这是一个Circle的成员函数
8     {
9         m_R = r;
10    }
11    //获取半径
12    int Circle::getR()
13    {
14        return m_R;
15    }
16    //设置圆心
17    void Circle::setCenter(Point center)
18    {
19        m_Center = center;
20    }
21    //获取圆心

```

```
22     Point Circle:: getCenter()
23     {
24         return m_Center;
25     }
```

```
1  //5、主程序
2  #include<iostream>
3  #include<string>
4  using namespace std;
5
6  #include "circle.h"
7  #include "point.h"
8  //判断点和圆的关系
9  void isInCircle(Circle &c, Point &p)
10 {
11     //计算两点之间的距离 平方
12     int distance =
13         (c.getCenter().getX() - p.getX())*(c.getCenter().getX() - p.getX()) +
14         (c.getCenter().getY() - p.getY())*(c.getCenter().getY() - p.getY());
15
16     //计算半径的平方
17     int rDistance = c.getR()*c.getR();
18
19     //判断
20     if (distance==rDistance)
21     {
22         cout << "点在圆上" << endl;
23     }
24     else if (distance>rDistance)
25     {
26         cout << "点在圆外" << endl;
27     }
28     else
29     {
30         cout << "点在圆内" << endl;
31     }
32 }
33
34 int main()
35 {
36     //实例化对象
37     //创建圆
38     Circle c;
39     c.setR(10);
40     Point center;
41     center.setX(10);
42     center.setY(0);
43     c.setCenter(center);
44
45     //创建点
46     Point p;
47     p.setX(10);
48     p.setY(10);
```

```

49
50     //判断关系
51     isInCircle(c, p);
52
53     system("pause");
54     return 0;
55 }

```

3.4.2 对象的初始化和清理

- 生活中我们买的电子产品都基本会有出厂设置，在某一天我们不用时候也会删除一些自己信息数据保证安全
- C++中的面向对象来源于生活，每个对象也都会有初始设置以及对象销毁前的清理数据的设置。

3.4.2.1 构造函数和析构函数

对象的**初始化和清理**也是两个非常重要的安全问题

一个对象或者变量没有初始状态，对其使用后果是未知

同样的使用完一个对象或变量，没有及时清理，也会造成一定的安全问题

c++利用了**构造函数**和**析构函数**解决上述问题，这两个函数将会被编译器自动调用，完成对象初始化和清理工作。

对象的初始化和清理工作是编译器强制要我们做的事情，因此如果**我们不提供构造和析构，编译器会提供但是，编译器提供的构造函数和析构函数是空实现。**

- 构造函数：主要作用在于创建对象时为对象的成员属性赋值，构造函数由编译器自动调用，无须手动调用。
- 析构函数：主要作用在于对象**销毁前**系统自动调用，执行一些清理工作。

构造函数语法： 类名(){}

1. 构造函数，没有返回值也不写void
2. 函数名称与类名相同
3. 构造函数可以有参数，因此可以发生重载
4. 程序在调用对象时候会自动调用构造，无须手动调用,而且只会调用一次

析构函数语法： ~类名(){}

1. 析构函数，没有返回值也不写void
2. 函数名称与类名相同,在名称前加上符号 ~
3. 析构函数不可以有参数，因此不可以发生重载
4. 程序在对象销毁前会自动调用析构，无须手动调用,而且只会调用一次

```

1  #include<iostream>
2  #include<string>
3  using namespace std;
4
5  //对象的初始化和清理
6  class Person

```

```

7  {
8  public:
9
10     ///1、构造函数 进行初始化操作
11     //没法返回值 不用写void
12     //函数名 与类名相同
13     //构造函数可以有参数， 可以发生重载
14     //创建对象的时候，构造函数会自动调用，而且只调用一次
15     Person()
16     {
17         cout << "构造函数的调用" << endl;
18     }
19
20     //2、析构函数 进行清理的操作
21     //没有返回值 不写void
22     //函数名和类名相同 在名称前面加~
23     //析构函数不可以有参数的，不可以发生重载
24     //对象在销毁前，会自动调用析构函数，而且只会调用一次
25     ~Person()
26     {
27         cout << "这是Person的析构函数调用" << endl;
28     }
29
30 };
31
32 //构造和析构函数都是必须有的实现，如果我们自己不提供，编译器会提供一个空实现的构造和析构
33 void test01()
34 {
35     Person p; //创建一个对象 这是在栈上的数据，test01执行完毕后，释放这个对象
36 }
37
38 int main()
39 {
40
41     //test01();
42     //Person p; //都执行完毕之后，才会执行析构函数的代码；对象执行完了，销毁了才会调用析构函数
43     system("pause");
44     return 0;
45 }

```

3.4.2.2 构造函数的分类及调用

两种分类方式：

按参数分为：有参构造和无参构造

按类型分为：普通构造和拷贝构造

三种调用方式：

括号法

显示法

隐式转换法

示例：

```
1  #include<iostream>
2  #include<string>
3  using namespace std;
4
5  //构造函数的分类和调用
6  //分类
7  // 按照参数分类    无参构造（默认构造） 和 有参构造
8  // 按照类型分类    普通构造和拷贝构造
9  class Person
10 {
11 public:
12     //无参构造
13     Person()
14     {
15         cout << "Person无参构造函数的调用" << endl;
16     }
17     //有参构造
18     Person(int a)
19     {
20         age = a;
21         cout << "Person有参构造函数的调用" << endl;
22     }
23
24     //拷贝构造函数
25     Person(const Person &p) //引用方式
26     {
27         //将传入的人身上的所有的属性拷贝到当前对象身上
28         cout << "Person拷贝构造函数的调用" << endl;
29         age = p.age;
30     }
31
32
33
34     //析构函数
35     ~Person()
36     {
37         cout << "Person的析构函数调用" << endl;
38     }
39
40     int age;
41 };
42
43 //调用
44 void test01()
45 {
46     //1、括号法
47     Person p1; //默认构造函数调用
48     Person p2(10); //有参构造函数
49     Person p3(p2); //拷贝构造函数
```

```

50
51 //注意事项1:
52 //调用默认构造函数时候, 不要加() Person p1()。因为, 编译器会认为是一个函数的声明, 不会认为在创建
对象
53
54 cout << "p2的年龄为: " << p2.age << endl;
55 cout << "p3的年龄为: " << p3.age << endl;
56
57 //2、显示法
58 Person p4; //无参构造
59 Person p5 = Person(10); //调用有参构造
60 Person p6 = Person(p5); // 拷贝构造
61
62 //Person(10); //匿名对象, , 起个名字, 放在右侧; 特点: 当前行执行结束后, 系统会立即回收掉匿名对象
63 //cout << "aaaa" << endl;
64
65 //注意事项2
66 //不要利用拷贝构造函数, 初始化匿名对象; 编译器会认为Person(p6) == Person P6; 对象声明
67 //Person(p6);
68
69
70 //3、隐式转换法
71 Person p7 = 10; //相当于 写了 Person p7 = Person(10);
72 Person p8 = p7; //拷贝构造
73
74 }
75
76 int main()
77 {
78
79     test01();
80
81     system("pause");
82     return 0;
83 }

```

3.4.2.3 拷贝构造函数调用时机

C++中拷贝构造函数调用时机通常有三种情况

- 使用一个已经创建完毕的对象来初始化一个新对象
- 值传递的方式给函数参数传值
- 以值方式返回局部对象

示例:

```

1 #include<iostream>
2 #include<string>
3 using namespace std;
4
5 //拷贝构造函数调用时机

```

```

6 //1、 使用一个已经创建完毕的对象来初始化一个新对象
7 //2、 值传递的方式给函数参数传值
8 //3、 值方式返回局部对象
9
10 class Person
11 {
12 public:
13     Person()
14     {
15         cout << "Person默认构造函数调用" << endl;
16         m_Age = 0;
17     }
18     Person(int age)
19     {
20         cout << "Person有参构造函数调用" << endl;
21         m_Age = age;
22     }
23
24     //拷贝构造函数
25     Person(const Person &p)
26     {
27         cout << "Person拷贝构造函数调用" << endl;
28         m_Age = p.m_Age;
29     }
30
31     //析构函数
32     ~Person()
33     {
34         cout << "Person析构函数调用" << endl;
35     }
36
37 public:
38     int m_Age;
39 };
40
41
42 //1、 使用一个已经创建完毕的对象来初始化一个新对象
43 void test01()
44 {
45     Person p1(20);
46     Person p2(p1);
47     cout << "p2的年龄为: " << p2.m_Age << endl;
48 }
49
50 //2、 值传递的方式给函数参数传值
51 void dowork(Person p)
52 {
53 }
54
55
56 void test02()
57 {
58     Person p; //默认构造

```

```

59     dowork(p); //实参传递给形参
60 }
61
62 //3、 值方式返回局部对象
63 Person dowork2()
64 {
65     Person p1;
66     cout << &p1 << endl;
67     return p1;
68 }
69 void test03()
70 {
71     Person p = dowork2();
72     cout << &p << endl;
73 }
74
75
76
77 int main()
78 {
79
80     //test01();
81     //test02();
82     test03();
83     system("pause");
84     return 0;
85 }

```

3.4.2.4 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)
2. 默认析构函数(无参，函数体为空)
3. 默认拷贝构造函数，对属性进行值拷贝

构造函数调用规则如下：

- 如果用户定义有参构造函数，c++不在提供默认无参构造，但是会提供默认拷贝构造
- 如果用户定义拷贝构造函数，c++不会再提供其他构造函数

```

//Person()
//{
//  cout << "Person的默认构造函数调用" << endl;
//}

//Person(int age)
//{
//  cout << "Person的有参构造函数调用" << endl;
//  m_Age = age;
//}

Person(const Person & p)
{
  cout << "Person的拷贝构造函数调用" << endl;
  m_Age = (const char [26])"Person的拷贝构造函数调用"
}

```

示例:

```

1  #include<iostream>
2  #include<string>
3  using namespace std;
4
5  //构造函数的调用规则
6  //1、创建一个类，c++编译器好 给每个类都添加至少3个函数
7  // 默认构造    (空实现)
8  // 析构函数    (空实现)
9  // 拷贝构造    (值拷贝)
10
11
12 //2、
13 // 如果我们写了有参构造函数，编译器就不在提供默认构造，依然提供拷贝构造
14 // 如果我们写了拷贝构造函数，编译器就不在提供其他普通构造函数了
15
16 class Person
17 {
18 public:
19     //Person()
20     //{
21     //  cout << "Person的默认构造函数调用" << endl;
22     //}
23
24     //有参构造
25     //Person(int age)
26     //{
27     //  cout << "Person的有参构造函数调用" << endl;
28     //  m_Age = age;
29     //}
30
31     Person(const Person &p)

```

```

32     {
33         cout << "Person的拷贝构造函数调用" << endl;
34         m_Age = p.m_Age;
35     }
36
37     ~Person()
38     {
39         cout << "Person的析构函数调用" << endl;
40     }
41
42
43 public:
44     int m_Age;
45 };
46
47 //void test01()
48 //{
49 //    Person p; // 默认构造函数
50 //    p.m_Age = 18;
51 //
52 //    Person p2(p); //拷贝构造函数
53 //    cout << "p2的年龄: " << p2.m_Age << endl;
54 //}
55
56 void test02()
57 {
58     //Person p; // 无参构造——注释掉自己提供的无参构造，编译器也不再提供啦，因为我提供了有参构造
59     //Person p(10);
60     //Person p2(p);
61     //cout << "p2的年龄: " << p2.m_Age << endl;
62
63 }
64
65
66 int main()
67 {
68
69     //test01();
70     test02();
71     system("pause");
72     return 0;
73 }

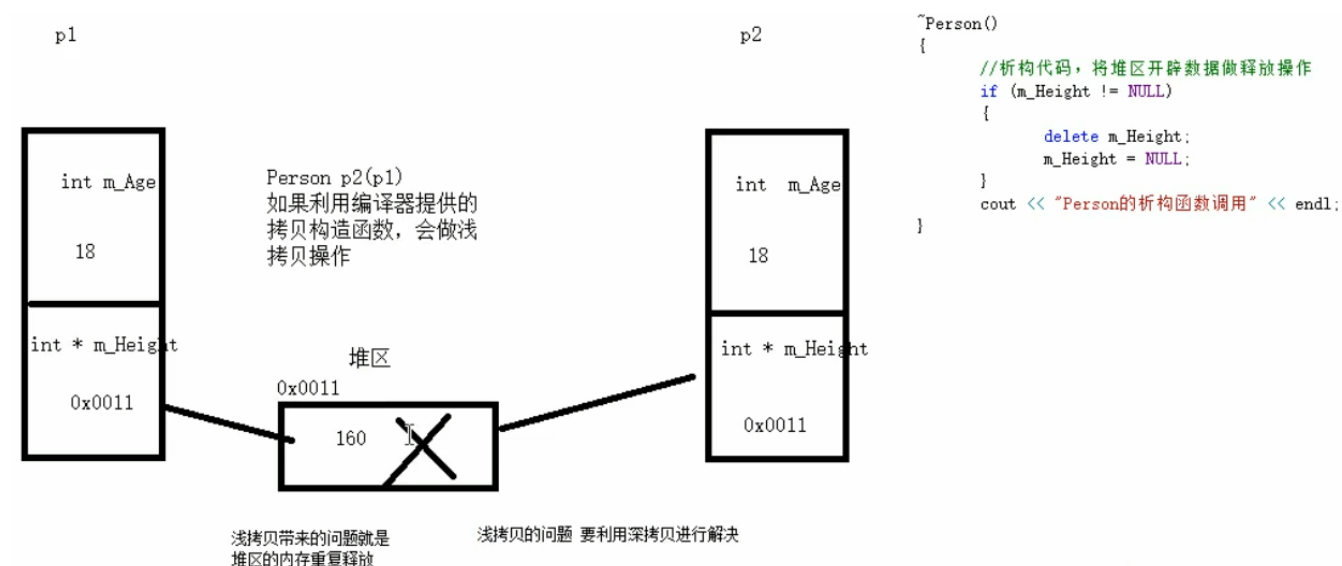
```

3.4.2.5 深拷贝与浅拷贝

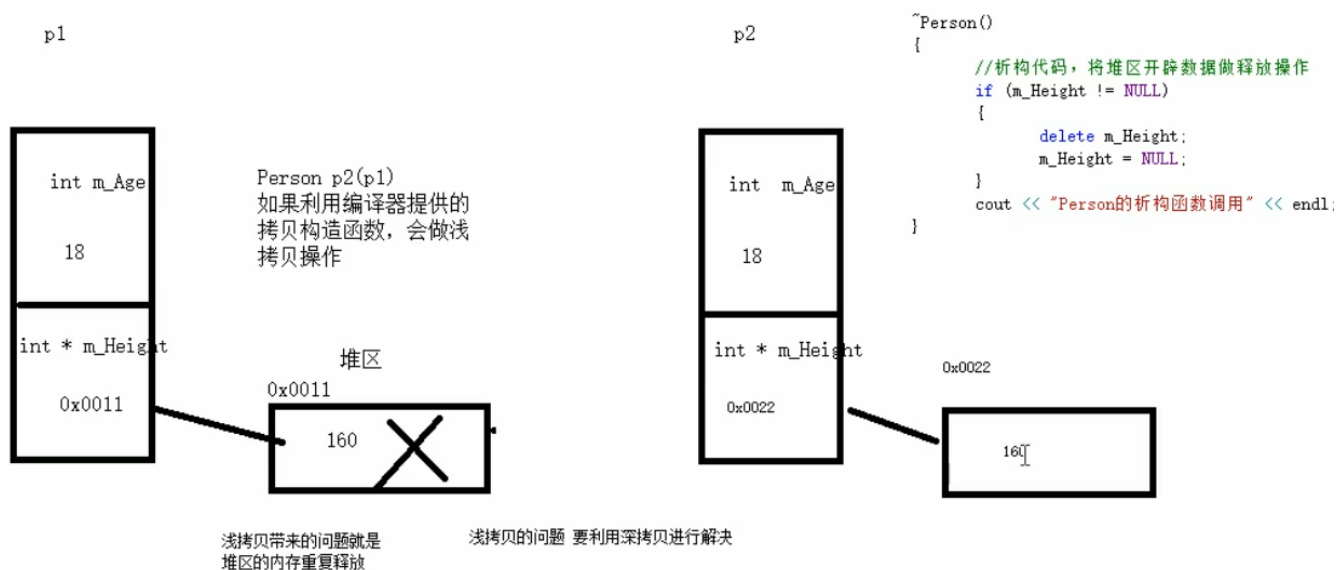
深浅拷贝是面试经典问题，也是常见的一个坑

浅拷贝：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



解决:



示例:

```

1  #include<iostream>
2  #include<string>
3  using namespace std;
4
5  // 深拷贝和浅拷贝
6  class Person
7  {
8  public:
9      Person()
10     {
11         cout << "Person默认构造函数" << endl;
12     }
13
14     Person(int age, int height)
15     {
16
17         m_Age = age;

```

```

18     m_Height = new int(height); //在堆区开辟了一块内存
19     cout << "Person有参构造函数" << endl;
20 }
21
22 //自己实现拷贝构造函数，解决浅拷贝带来的问题
23 //拷贝构造函数
24 Person(const Person &p)
25 {
26     cout << "Person拷贝构造函数!" << endl;
27     //如果不利用深拷贝在堆区创建新内存，会导致浅拷贝带来的重复释放堆区问题
28     m_Age = p.m_Age;
29     //m_Height = p.m_Height; //编译器默认实现就是这行代码
30     //深拷贝操作：
31     m_Height = new int(*p.m_Height);
32
33 }
34
35 ~Person()
36 {
37     //析构代码，将堆区开辟的数据做一个释放作用
38     if (m_Height != NULL)
39     {
40         delete m_Height;
41         m_Height = NULL; //防止野指针出现，做一个制空操作
42     }
43     cout << "Person析构构造函数" << endl;
44 }
45
46 public:
47     int m_Age;
48     int *m_Height; // 升高，指针 要开辟到堆区
49 };
50
51 void test01()
52 {
53     Person p1(18,160);
54     cout << "p1的年龄为: " << p1.m_Age << "升高为: " << *p1.m_Height << endl;
55
56     Person p2(p1);
57     cout << "p1的年龄为: " << p2.m_Age << "升高为: " << *p2.m_Height << endl;
58
59 }
60
61 int main()
62 {
63
64     test01();
65
66     system("pause");
67     return 0;
68 }

```


总结：如果属性有在堆区开辟的，一定要自己提供拷贝构造函数，防止浅拷贝带来的问题

3.4.2.6 初始化列表

作用：

C++提供了初始化列表语法，用来初始化属性

语法：构造函数()：属性1(值1),属性2 (值2) ... {}

示例：

```
1  #include<iostream>
2  #include<string>
3  using namespace std;
4
5  //初始化列表
6  class Person
7  {
8
9      //传统初始化操作
10 public:
11
12     //有参构造
13     //Person(int a, int b, int c)
14     //{
15     //    m_A = a;
16     //    m_B = b;
17     //    m_C = c;
18     //}
19
20     //初始化列表初始化属性
21     Person(int a, int b, int c) :m_A(a), m_B(b), m_C(c) //Person() :m_A(10), m_B(20),
m_C(30
22     {
23
24     }
25
26 public:
27     int m_A;
28     int m_B;
29     int m_C;
30 };
31
32 void test01()
33 {
34     //Person p(10, 20, 30);
35
36     Person p(10,20,34);
37     cout << "m_A=" << p.m_A << endl;
38     cout << "m_B=" << p.m_B << endl;
39     cout << "m_C=" << p.m_C << endl;
40 }
41
```

```
42 int main()
43 {
44
45     test01();
46
47     system("pause");
48     return 0;
49 }
```

3.4.2.7 类对象作为类成员

C++类中的成员可以是另一个类的对象，我们称该成员为 对象成员

例如：

B类中有对象A作为成员，A为对象成员

那么当创建B对象时，A与B的构造和析构的顺序是谁先谁后？

示例：

3.4.2.8 静态成员

静态成员就是在成员变量和成员函数前加上关键字static，称为静态成员

静态成员分为：

- 静态成员变量
 - 所有对象共享同一份数据
 - 在编译阶段分配内存
 - 类内声明，类外初始化
- 静态成员函数
 - 所有对象共享同一个函数
 - 静态成员函数只能访问静态成员变量

示例1：静态成员变量

3.4.3 C++对象模型和this指针

3.4.3.1 成员变量和成员函数分开存储

在C++中，类内的成员变量和成员函数分开存储

只有非静态成员变量才属于类的对象上

3.4.3.2 this指针概念

通过4.3.1我们知道在C++中成员变量和成员函数是分开存储的

每一个非静态成员函数只会诞生一份函数实例，也就是说多个同类型的对象会共用一块代码

那么问题是：这一块代码是如何区分那个对象调用自己的呢？

c++通过提供特殊的对象指针，this指针，解决上述问题。**this指针指向被调用的成员函数所属的对象**

this指针是隐含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

3.4.3.3 空指针访问成员函数

C++中空指针也是可以调用成员函数的，但是也要注意有没有用到this指针

如果用到this指针，需要加以判断保证代码的健壮性

示例：

3.4.3.4 const修饰成员函数

常函数：

- 成员函数后加const后我们称为这个函数为**常函数**
- 常函数内不可以修改成员属性
- 成员属性声明时加关键字mutable后，在常函数中依然可以修改

常对象：

- 声明对象前加const称该对象为常对象
- 常对象只能调用常函数

示例：

3.4.4 友元

生活中你的家有客厅(Public)，有你的卧室(Private)

客厅所有来的客人都可以进去，但是你的卧室是私有的，也就是说只有你能进去

但是呢，你也可以允许你的好闺蜜好基友进去。

在程序里，有些私有属性 也想让类外特殊的一些函数或者类进行访问，就需要用到友元的技术

友元的目的就是让一个函数或者类 访问另一个类中私有成员

友元的关键字为 `friend`

友元的三种实现

- 全局函数做友元
- 类做友元
- 成员函数做友元

3.4.4.1 全局函数做友元

3.4.4.2 类做友元

3.4.4.3 成员函数做友元

3.4.5 运算符重载

运算符重载概念：对已有的运算符重新进行定义，赋予其另一种功能，以适应不同的数据类型

3.4.5.1 加号运算符重载

作用：实现两个自定义数据类型相加的运算

总结1：对于内置的数据类型的表达式的运算符是不可能改变的

总结2：不要滥用运算符重载

3.4.5.2 左移运算符重载

作用：可以输出自定义数据类型

总结：重载左移运算符配合友元可以实现输出自定义数据类型

3.4.5.3 递增运算符重载

作用：通过重载递增运算符，实现自己的整型数据

总结：前置递增返回引用，后置递增返回值

3.4.5.4 赋值运算符重载

c++编译器至少给一个类添加4个函数

1. 默认构造函数(无参，函数体为空)
2. 默认析构函数(无参，函数体为空)
3. 默认拷贝构造函数，对属性进行值拷贝
4. 赋值运算符 operator=, 对属性进行值拷贝

如果类中有属性指向堆区，做赋值操作时也会出现深浅拷贝问题

示例：

3.4.5.5 关系运算符重载

作用：重载关系运算符，可以让两个自定义类型对象进行对比操作

示例：

3.4.5.6 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像函数的调用，因此称为仿函数
- 仿函数没有固定写法，非常灵活

示例：