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**

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

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

示例如下：

```
//1、构造函数分类
```

```cpp
// 按照参数分类分为 有参和无参构造    无参又称为默认构造函数
// 按照类型分类分为 普通构造和拷贝构造
class Person {
public:
    //无参（默认）构造函数
    Person() {
        cout << "无参构造函数!" << endl;
    }
    //有参构造函数
    Person(int a) {
        age = a;
        cout << "有参构造函数!" << endl;
    }
    //拷贝构造函数
    Person(const Person& p) {
        age = p.age;
        cout << "拷贝构造函数!" << endl;
    }
    //析构函数
    ~Person() {
        cout << "析构函数!" << endl;
    }
public:
    int age;
};

//2、构造函数的调用
//调用无参构造函数
void test01() {
    Person p; //调用无参构造函数
}

//调用有参的构造函数
void test02() {

    //2.1  括号法，常用
    Person p1(10);
    //注意1：调用无参构造函数不能加括号，如果加了编译器认为这是一个函数声明
    //Person p2();

    //2.2 显式法
    Person p2 = Person(10);
    Person p3 = Person(p2);
    //Person(10)单独写就是匿名对象  当前行结束之后，马上析构

    //2.3 隐式转换法
    Person p4 = 10; // Person p4 = Person(10);
    Person p5 = p4; // Person p5 = Person(p4);

    //注意2：不能利用 拷贝构造函数 初始化匿名对象 编译器认为是对象声明
    //Person p5(p4);
}

int main() {
    test01();
    //test02();
    system("pause");
    return 0;
}
```

构造函数的调用规则

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

拷贝构造的书写方式：

类名 （const 类名 &对象名）

{

```
代码段；
```

}

深拷贝与浅拷贝

```
class Person
{
  person()
  {
    cout<<"Person的默认构造调用"<<endl;
  }
  person(int age,int height)
  {
    m_Age=age;
    m_Height= new int(height);
    cout<<"Person的有参构造调用"<<endl;
  }
  ~person()
  {
    //析构代码：将堆区开辟的数据释放掉
    if(m_Height!=NULL)
    {
      delete m_Height;
      m_Height=NULL;
    }
    cout<<"Person的析构函数调用"<<endl;
  }
  int m_Age;
  int *m_Height;
```

```
}
int main()
{
  Person p1(18,180);
  Person p2(p1);
  cout << "p1的年龄：  " << p1.m_age << " 身高：  " << *p1.m_height << endl;
    cout << "p2的年龄：  " << p2.m_age << " 身高：  " << *p2.m_height << endl;
  system("pause");
  return 0;
}
```

上述代码在运行时会崩溃，为什么呢?

原因是，我们的m_Height这个变量开辟在堆区，而编译器提供给我们的拷贝构造函数仅仅只是简单的值传递（浅拷贝），因此，在Person p2(p1);执行完之后，p1和p2的m_Height属性指向的是一块相同的内存空间。也正是因为指向的是同一块内存空间，因此，在p1的析构函数调用时，执行delete m_Height;该区域已经被释放。而当p2的析构函数调用时，执行delete m_Height；此时原有的堆区内存已经被释放，再次释放属于非法操作，于是程序崩溃。

那么怎么解决这个问题呢？

我们需要做深拷贝的操作，也就是重写拷贝构造函数，代码如下：

```
Peron(const Person &p)
{
  m_Age=p.m_Age;
  m_Height= new int(*p.m_Height);
}
```

可以看到，我们在堆区重新申请了空间，那么调用拷贝构造之后，就不会指向同一块内存，继而也不会造成同一块内存重复释放的问题了，我我们完成了深拷贝。