构造/析构 函数

1. 概念

1.1 构造函数

> - **定义：** 与类名相同的特殊成员函数
> - **语法：** ***ClassName();***
> - **作用：** 完成对属性的初始化
> - **特点：** ①. 在定义时可以有参数，也可没有参数　　②. 没有任何返回类型的声明
> - **调用方式：** 一般情况下C++编译器会自动调用构造函数, 在一些情况下则需要手工调用构造函数

```
class MyTest{
public:
    MyTest() {}   // 构造函数：定义对象时，自动调用该函数，完成对属性的初始化
    ~MyTest() {}  // 析构函数：对象声明周期结束时，自动调用该函数，释放对象占用的空间
};
```

1.2 析构函数

> - **定义：** 在构造函数名前加 '~' 的特殊成员函数
> - **语法：** ***~ClassName();***
> - **作用：** 对象销毁时，自动被调用，用来释放对象占用的空间
> - **特点：** ①. 声明的析构函数 没有参数 没有任何返回类型　　②. 在对象销毁时自动被调用
> - **调用方式：** 被C++编译器自动调用

> - **注意（重点!!!）：** 先定义的对象 后析构

---

2. 构造函数分类

> 1. 默认构造函数
> 2. 无参数构造函数
> 3. 带参数构造函数
> 4. 拷贝构造函数

```
class Test{
public:
    Test(){          // 1. 无参数构造函数
        m_a = 0;    m_b = 0;    m_c = 0;
    }
    Test(int a){    // 2. 有参数构造函数 1个参数
        m_a = a;
    }
    Test(int a, int b, int c){  // 2. 有参数构造函数  3个参数
        m_a = a;    m_b = b;    m_c = c
    }

    Test(const Test& obj ){     // 3 . 拷贝构造函数
        /* ... ... */
    }
private:
    int m_a, m_b, m_c;
};
```

2.1 默认构造函数

> - **默认 无参 构造函数：** 当类中没有定义构造函数时，编译器默认提供一个无参构造函数，并且其函数体为空
> - **默认 拷贝 构造函数：** 当类中没有定义拷贝构造函数时，编译器默认提供一个默认拷贝构造函数，简单的进行成员变量的值复制**（浅拷贝）**

2.2 无参构造函数

```
Test t1;   // 直接调用 无参数构造函数  --> 属性初始化值为： m_a = 0;  m_b = 0; m_c = 0;
```

2.3 有参构造函数

> **注意：**对象初始化 和 对象赋值是两个不同的概念！！！

```
/* 1. 括号法 自动调用 */
Test t2(1, 2, 3);    // 直接调用三个参数的构造函数： --> 属性初始化值为： m_a = 1;  m_b = 2;
m_c = 3;

/* 2. 等号法 自动调用 */
Test t3 = (1, 2);    // 直接调用一个参数的构造函数： --> 属性初始化值为： m_a = 2;  注意：C++等号
符功能加强！！！直接取最后一个参数
Test t4 = (1, 2, 3); // 直接调用三个参数的构造函数： --> 属性初始化值为： m_a = 1;  m_b = 2;
m_c = 3;

/* 3. 手动调用 初始化操作 */
Test t5 = Test(1, 2, 3);  // 产生匿名对象， 赋给t5， 但只调用一次构造函数， 原因是把匿名对象那块地
址 直接命名为t5 ！！！
```

2.4 拷贝构造函数

> **作用：**用一个对象 去 初始化另外一个对象

```
class Test{
public:
    Test(int a, int b){     // 1. 有参数构造函数  2个参数
        m_a = a;    m_b = b;
    }

    Test(const Test& obj ){ // 2 . 拷贝构造函数
        m_a = obj.m_a;  m_b = obj.m_b;
    }
private:
    int m_a, m_b;
};

void MyTest_03(Test mp){    // 第三种调用测试
    /* ... ... */
}

Test MyTest_04(){           // 第四种调用测试
    Test tmp(1, 2);
    return tmp;    // *******重点！！！ 返回一个匿名对象 返回时调用 一次拷贝构造函数(构造的是匿名对
象) 在进行一次析构函数(析构的是tmp)
}

void main(){
```

```
    Test a(1, 2);

    /* 第一种调用：直接调用 */
    Test b = a;      // 1. 不会调用普通构造含糊 直接调用拷贝构造函数！！！
    // t1 = t2; --> 不会调用拷贝构造函数 这种写法是操作符重载

    /* 第二种调用：直接调用 */
    Test c(a);       // 2. 与上边一样 只是写法不同

    /* 第三种调用:类的形参初始化 */
    MyTest_03(a);    // 3. 此时会调用拷贝构造函数 该函数执行完毕后 调用形参参数的析构函数

    /* 第四种调用：函数返回类的匿名对象 */
    Test d = MyTest_04();    //  4. 该函数返回了一个匿名对象 使用了匿名对象时 直接把d 的名字给匿名对象（地址不会改变）
    a = MyTest_04();         //     因为 a 已经被定义，此时匿名对象未使用，则直接调用匿名对象的析构函数
}
```

3. 构造函数的规则

> - 当类中定义了 有参/无参构造 函数时， C++编译器 不会提供 默认的构造函数
> - 当类中定义了 拷贝构造 函数时，C++编译器 不会提供 默认的构造函数
> - 默认拷贝构造函数成员变量简单赋值 **（浅拷贝）**

---

4. 浅/深 拷贝

> - **浅拷贝：** 默认拷贝构造函数 可以完成 对象的数据成员值 简单地复制
> - **深拷贝：** 对象的属性 有指针 指示的堆时，需要 显式定义 拷贝构造函数（自定义）

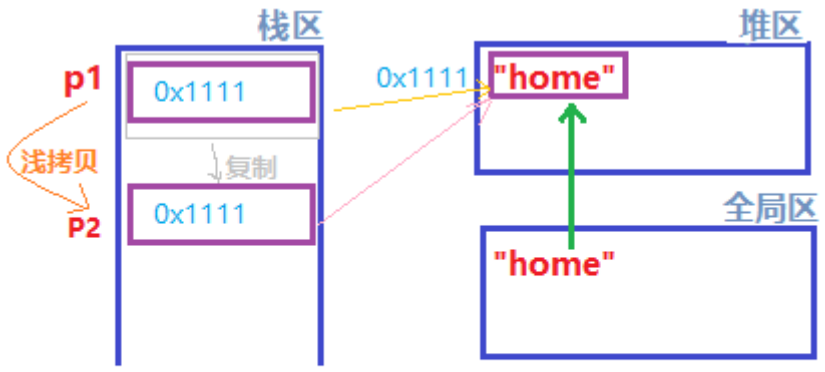
4.1 浅拷贝

```
class MyClass{
public:
    MyClass(const char *sp){
        m_cp = (char *)malloc(strlen(sp) + 1);  // 需要末尾添加 \n ，所以此处 +1
        strcpy(m_cp, sp);
    }

    /* 编译器 自动提供一个默认的拷贝构造函数 */

    ~MyClass(){
        if(m_cp != NULL){
            free(m_cp);
            m_cp = NULL;
        }
    }
private:
    char *m_cp;
}

void Test(){
    MyClass p1("home"), p3;
    MyClass p2 = p1;    // 1. 此处进行默认的拷贝 但是 运行时候会造成 断错误， 原因如下图
    p3 = p1;            // 2. 操作运算符重载 使用默认的运算符重载 也会 存在这样的问题 造成段错误
}
```



先析构P2后，堆区的“home”这块内存空间就释放了

接着在析构P1, 所以会造成断错误

> **注意（重点!!!）：** 操作运算符重载 使用 默认的运算符重载 (浅拷贝) 也会段错误

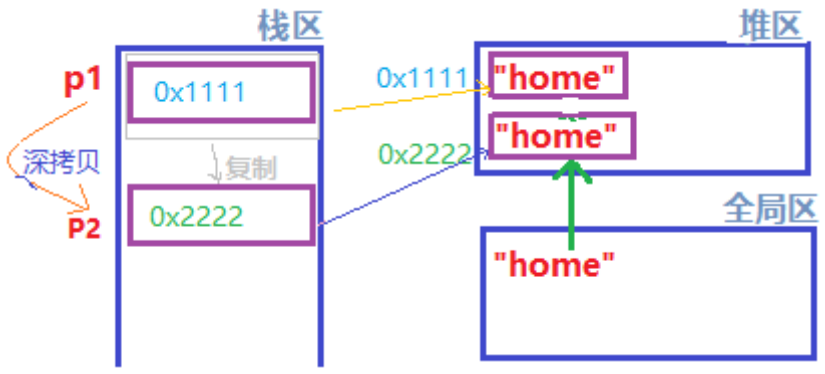
4.2 深拷贝

```
class MyClass{
public:
    MyClass(const char *sp){
        m_cp = (char *)malloc(strlen(sp) + 1);  // 需要末尾添加 \n ，所以此处 +1
        strcpy(m_cp, sp);
    }

    MyClass(const Name& obj1){                  // 自定义拷贝构造函数完成深拷贝
        m_cp = (char *)malloc(strlen(bj1.m_cp) + 1);
        strcpy(m_cp, obj1.m_cp);
    }

    ~MyClass(){
        if(m_cp != NULL){
            free(m_cp);
            m_cp = NULL;
        }
    }
private:
    char *m_cp;
}

void Test(){
    MyClass p1("home"), p3;
    MyClass p2 = p1;         // 这样就解决了这个问题
}
```



---

5. 构造 初始化列表

5.1 语法规则

> 构造函数初始化列表以一个冒号开始, 以逗号分隔的数据成员列表, 每个数据成员后面跟一个放在括号中的初始化式(所赋的参数)

```
class CExample {
public:
    CExample(): a(0),b(8.8) {} /* 构造函数初始化列表 */
    CExample(){ a=0; b=8.8; }  /* 构造函数内部赋值 */
private:
    int a; float b;
};
```

5.2 必须 使用初始化列表 情景

> ☑
>
> **情景1：** 成员类型是 **没有默认构造函数的类**，若没有提供显示初始化式，则编译器隐式使用成员类型的默认构造函数，若类没有默认构造函数，则编译器尝试使用默认构造函数将会失败。

```
class A {
public:
    A(int a) {m_a = a}
private:
    int m_a;
};

class B {
public:     /* 编译器提供默认构造函数 */
private:
    A m_b;
};

void Test(){
    B obj;       // 编译出错！！！ 原因：没有机会初始化 B 类
}
```

> ☑
>
> **情景2：** **const 成员** 或 **引用类型** 的成员。因为 const 对象或引用类型只能初始化，不能对他们赋值。

```
class CExample {
public:
    CExample(): a(0),b(8.8) {}
private:
    int m_a; const float m_fb;
};
```

5.3 成员变量的初始化顺序

> - 成员变量的初始化顺序 与 **声明的顺序** 相关，与在 初始化列表中的顺序 无关（重点!!!）

```
class CExample {
public:
    CExample(): a(0),b(8.8) {} /* 构造函数初始化列表 与初始化列表顺序无关 */
private:
    float b; int a;     /* 先初始化b 再初始化a 与声明顺序有关 */
};
```

---

6. 构造函数中调用构造函数

> **说明：** 构造函数 中 调用构造函数 是 ***不可取(危险)*** 的！！！

```
class Test{
public:
    Test(int a, int b, int c){
        this->a = a;    this->b = b;    this->c = c;
    }
    Test(int a, int b){
        this->a = a;    this->b = b;
        Test(a, b, 100);  // 产生新的匿名对象，  对原有的 属性并无影响，并且执行完此函数后 匿名对象被析构
    }
private:
    int a, b, c;
}

void MyTest(){
    Test x(1, 2);
    cout << "c的值为？" << endl;         // C的值是 随机值！！
}
```

> - **运行结果：** C的值是 随机值 (没有赋初值)
> - **原因：** 构造 调用 构造， 产生了新的匿名对象， 对原有的 属性并无影响

---

7. 匿名对象的生命周期

```
class Test{
public:
    Test(int a, int b, int c){
        this->a = a;    this->b = b;    this->c = c;
    }
    ~Test(){}
private:
    int a, b, c;
}

void MyTest(){
    Test(1, 2, 3);              // 直产生匿名对象  --> 先调用一次 构造函数   紧接着在调用析构函数
    Test aa = Test(1, 2, 3);    // 先调用一次 构造函数 然后将aa名字赋给匿名对象地址(编译器自动优化) aa生命周期结束后 在调用析构
}
```
