C++ 类和对象

一、理解类和对象

类的存在

用C风格：

```
int age1, age2;
string name1, name2;
```

← 数据是散的，行为也散

用类：

```
class Student {
public:
    string name;
    int age;
};
```

← 现实世界的“一个东西”
→ 程序里的“一个类”

类不是代码结构
类是现实事物的抽象

什么是对象

```
Student s1;
```

类：学生的“设计图”
对象：真正的一个学生
👉 只有对象才是真实存在的

二、构造函数　对象“出生”的那一刻

构造函数存在的作用

对象一出生，立刻被初始化　　✓ 分配内存
✓ 调用构造函数

构造函数 = 出生时设定身份信息

```
class Student {
public:
    int age;
    Student(int a) {
        age = a;
    }
};
```

```
Student s(18);
```

对象一出生，直接写好年龄

构造函数的知识点

构造函数的定义

构造函数是一种特殊的成员函数，核心作用是对对象分配空间并完成初始化，基础属性包括：

- 名字必须与类名相同，不能由用户任意命名；
- 可包含任意类型的参数，但无返回值类型（甚至不能用 `void` 声明）；
- 无需用户手动调用，在建立对象时自动执行。

构造函数的特性

1. **命名要求**：名字必须与类名一致，否则会被编译器当作普通成员函数处理；
2. **返回值限制**：无返回值，定义时不能声明任何类型（包括 `void`）；
3. **定义位置**：函数体可写在类内，也可写在类外；
4. **功能规范**：核心用于对象初始化（可对数据成员赋值、包含其他语句），建议不加入与初始化无关的内容以保持功能清晰；
5. **调用规则**：一般声明为公有成员，无需 / 不能被显式调用，在对象定义时自动调用且仅执行一次；
6. **默认生成**：若未自定义构造函数，编译系统会自动生成一个默认构造函数。

三、初始化列表　　不是“赋值”，是“直接给”

```
Student(int a) {
    age = a;
}
```

先构造，再改

```
Student(int a) : age(a) {}
```

构造时就定好

✓ const 成员 只能用初始化列表
✓ 引用成员 只能用初始化列表

四、析构函数　　对象“死亡”的那一刻

析构函数什么时候执行？

```
{
    Student s;
}
```

出了作用域 → 对象死亡 → 析构

析构函数理解

程序里：

- delete 内存
- 关文件
- 释放资源

析构函数的知识点

析构函数的定义

析构函数是一种特殊的成员函数，其执行操作与构造函数相反，核心作用是完成清理任务（如释放对象所占用的内存空间）。

析构函数的特点

1. **命名规则**：函数名需与类名相同，但名称前必须添加波浪号（~）；
2. **返回值限制**：无返回值，定义时不能声明任何类型（包括`void`类型）；
3. **参数与重载**：没有参数，因此无法被重载（一个类只能有一个析构函数，而构造函数可存在多个）；
4. **调用时机**：当对象被撤销（销毁）时，由编译系统自动调用。

五、拷贝构造函数

不是“复制人”
是“搬家”

拷贝构造执行时间

```
Student s1;
Student s2 = s1;
```

👉 s2 是新对象
👉 s1 不会消失

默认拷贝构造

```
class Student {
public:
    int* age;
    Student(int a) {
        age = new int(a);
    }
};
```

两个对象指向同一块内存

```
s1.age  ──┐
          ├── 同一块内存
s2.age  ──┘
```

拷贝构造的意义

不是复制指针

是复制“资源”

拷贝构造函数的知识点

1. **定义**：特殊构造函数，用同类已有对象初始化新对象。
2. **核心特点**：
   - 名与类名一致；
   - 参数为“本类 const 对象引用”；
   - 无返回值，不可重载。
3. **调用时机**：
   - 用已有对象初始化新对象；
   - 类对象按值传参；
   - 函数返回类对象。
4. **默认行为**：未自定义则编译器生成默认版（执行浅拷贝）；对象含动态内存时，需自定义实现深拷贝避免错误。

！ 核心内容辨析

析构顺序！

构造顺序：先成员，再对象本身

析构顺序：先对象本身，再成员（反过来）

```
/* ---------- 基础类 ---------- */
class A {
public:
    A()  { cout << "A 构造\n"; }
    ~A() { cout << "A 析构\n"; }
};
class B {
public:
    B()  { cout << "B 构造\n"; }
    ~B() { cout << "B 析构\n"; }
};
/* ---------- 含成员对象 ---------- */
class C {
    A a;    // 成员1 (先声明)
    B b;    // 成员2 (后声明)
public:
    C()  { cout << "C 构造\n"; }
    ~C() { cout << "C 析构\n"; }
};
/* ---------- 继承关系 ---------- */
class Base {
public:
    Base()  { cout << "Base 构造\n"; }
    ~Base() { cout << "Base 析构\n"; }
};

class Derived : public Base {
    A a;    // 派生类成员
public:
    Derived()  { cout << "Derived 构造\n"; }
    ~Derived() { cout << "Derived 析构\n"; }
};
```

```
===== 1. 普通对象 =====
A 构造

===== 2. 对象数组 =====
A 构造
A 构造

===== 3. 含成员对象 =====
A 构造
B 构造
C 构造

===== 4. 继承关系 =====
Base 构造
A 构造
Derived 构造
```

```
Derived 析构
A 析构
Base 析构
C 析构
B 析构
A 析构
A 析构
A 析构
A 析构
```

栈对象：后构造，先析构

对象数组：从后往前析构

成员对象：构造正序，析构反序（按声明顺序）

继承体系：先析构派生类，再析构基类

深拷贝&浅拷贝

浅拷贝 / 深拷贝，是在解决：

“对象里有资源（尤其是指针）时，复制对象该怎么复制才安全”

浅拷贝的本质

浅拷贝 = 只拷贝成员的“值”本身

浅拷贝不是复制资源，而是“共享同一份资源”

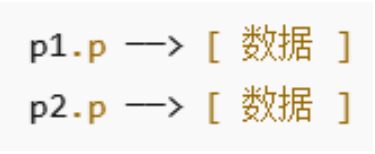
✓ 两个对象

✕ 一份资源

深拷贝的核心思想

不共享资源，每个对象都有“自己的一份”

是重新申请地址 + 复制内容



✓ 两个对象

✓ 两份资源

浅拷贝 / 深拷贝 ->拷贝构造函数行为的描述

👉 默认拷贝构造 = 浅拷贝

👉 自定义拷贝构造 = 可以实现深拷贝

```cpp
#include <iostream>
using namespace std;

class Sample {
public:
    int* data;

    Sample(int value) {
        data = new int(value);
        cout << "Constructor: " << *data << endl;
    }

    // Shallow copy: copy address only
    Sample(const Sample& other) {
        data = other.data;
        cout << "Shallow copy constructor" << endl;
    }

    ~Sample() {
        cout << "Destructor: delete " << *data << endl;
        delete data;
    }
};

int main() {
    Sample a(10);
    Sample b = a;   // shallow copy
    return 0;
}
```

浅拷贝只复制指针，不复制内存内容

两个对象共享同一块堆内存

析构顺序不同，可能导致重复 delete → 程序崩溃或未定义行为

解决方法：自己实现深拷贝构造函数，重新分配内存并拷贝数据

```cpp
#include <iostream>
using namespace std;

class Sample {
public:
    int* data;

    Sample(int value) {
        data = new int(value);
        cout << "Constructor: " << *data << endl;
    }

    // Deep copy: copy value, not address
    Sample(const Sample& other) {
        data = new int(*other.data);
        cout << "Deep copy constructor" << endl;
    }

    ~Sample() {
        cout << "Destructor: delete " << *data << endl;
        delete data;
    }
};

int main() {
    Sample a(20);
    Sample b = a;    // deep copy
    return 0;
}
```

```
C:\WINDOWS\system32\cmd.
Constructor: 20
Deep copy constructor
Destructor: delete 20
Destructor: delete 20

请按任意键继续. . .
```

深拷贝为每个对象分配独立内存
对象互不影响，修改一个对象不会影响另一个对象
析构安全，每个对象 delete 自己的内存
解决了浅拷贝的 double delete 问题

Conclusion:

1. 浅拷贝：
   - 拷贝指针地址 → 共享内存
   - 析构时可能重复释放 → 未定义行为
   - 默认拷贝构造就是浅拷贝
2. 深拷贝：
   - 重新申请内存，拷贝数据 → 独立内存
   - 析构安全
   - 需要手动实现拷贝构造函数
3. 关键一句话：

“只要类中有指针成员，并且析构函数会 delete 该指针，就必须实现深拷贝拷贝构造函数，否则会发生重复释放。”