C++必会 9 个高频考点

> 我的 b 站主页
>
> 部分基础（如数据类型、运算符等）请结合《C 语言期末必会 10 题 ——10 题搞定 C 语言》来补充复习
> 可以同时结合期末视频讲解食用2020C++试题讲解

初始化列表

作用：构造函数的一种特殊语法，用于在对象创建时直接初始化成员变量，而不是在构造函数体内进行赋值。

为什么需要它?

1. **效率更高**：对于类类型的成员，使用初始化列表是“初始化”，在函数体内赋值是“先默认初始化，再赋值”，前者省去了一次不必要的操作。
2. **必须使用**：当类中有以下成员时，**必须**使用初始化列表：
   - `const` 常量成员变量（因为常量只能在定义时初始化）。
   - 引用类型成员变量（因为引用必须在定义时绑定）。
   - 没有默认构造函数的类类型成员。

注意：初始化列表中的成员变量初始化顺序只与声明顺序有关。

题目：

下面代码有什么问题？如何修正？

```
class MyClass {
    const int id;
    string& name;
public:
    MyClass(int i, string& n) {
        id = i;       // 错误
        name = n;     // 错误
    }
};
```

解析与答案：

- **问题分析**：
  - `id` 是一个 `const` 常量成员，它必须在创建对象时（即构造函数的初始化列表阶段）就被初始化，不能在构造函数体内部被赋值。
  - `name` 是一个引用成员，引用必须在声明时就绑定到一个已存在的对象，这个过程也必须在初始化列表中完成。
- **解决方法**：
  使用初始化列表来初始化 `id` 和 `name`。
- **正确代码**：

```
#include <iostream>
#include <string>

class MyClass {
    const int id;
    std::string& name;
public:
    // 使用初始化列表
    MyClass(int i, std::string& n) : id(i), name(n) {
        // 构造函数体可以为空，或者执行其他操作
    }
    void print() {
        std::cout << "ID: " << id << ", Name: " << name << std::endl;
    }
};

int main() {
    std::string myName = "Alice";
    MyClass obj(101, myName);
    obj.print(); // 输出: ID: 101, Name: Alice

    myName = "Bob"; // 修改原始变量
    obj.print(); // 输出: ID: 101, Name: Bob (因为name是引用)
    return 0;
}
```

拷贝构造函数

23. 拷贝构造函数的作用是什么？何时会调用拷贝构造函数？

- **作用**：
  拷贝构造函数是一个特殊的构造函数，用于**创建一个新对象，并用已有对象的成员变量值初始化新对象**。
  它确保新对象是现有对象的副本，尤其在类中有动态分配资源（如指针）时，需进行**深拷贝**，避免浅拷贝导致的问题（如多个对象共享同一内存）。
  例如：

```
class MyClass {
    int* ptr;
public:
    MyClass(const MyClass& other) {  // 拷贝构造函数
        ptr = new int(*other.ptr);  // 深拷贝
    }
    // ... 其他成员函数和构造函数
};
```

- **调用时机**：
  i. **对象初始化**：用一个对象初始化另一个对象时，例如 `MyClass obj2 = obj1;` 或 `MyClass obj2(obj1);`。
  ii. **函数参数传递**：对象以值传递方式传入函数时，形参创建副本，例如 `void func(MyClass obj); func(obj1);`。
  iii. **函数返回值**：函数返回对象时，返回值创建副本，例如 `MyClass func() { return obj1; }`（除非有返回值优化 RVO）。

深拷贝与浅拷贝

题目代码：

```
#include <cstring>

class A {
    int x, y;
    char *p;
public:
    A(const char *str) {
        x = 0; y = 0;
        p = new char[strlen(str)+1];
        strcpy(p, str);
    }
    ~A() { delete [] p; p = NULL; }
    // ... 缺少拷贝构造函数
};

int main() {
    A a1("abcd");
    A a2(a1); // 这里会出问题
    return 0;
} // 程序结束时，a1和a2的析构函数都会被调用
```

- **(1) 以上代码存在什么问题？（2 分）**
- **(2) 如何解决？（2 分）**

解析：

- **问题分析**：
  - `A a2(a1)` 调用了编译器自动生成的**默认拷贝构造函数**，该函数执行的是**浅拷贝**。
  - 浅拷贝只是简单地将 `a1.p` 的指针值赋给了 `a2.p`，导致 `a1` 和 `a2` 的成员指针 `p` 指向了**同一块**动态分配的内存。
  - 当 `main` 函数结束时，`a2` 首先被析构，释放了 `p` 指向的内存。随后 `a1` 被析构，试图再次释放已经被释放的同一块内存，这会导致**重复释放（double free）**，引发程序崩溃或未定义行为。
- **解决方法**：
  - 手动定义一个执行**深拷贝**的拷贝构造函数。深拷贝会为新对象 `a2` 的指针 `p` 分配一块全新的内存，并将 `a1.p` 所指向内存中的内容复制过来。这样，两个对象就各自拥有独立的内存资

源。

答案：

- **(1)**：代码没有定义拷贝构造函数，导致编译器使用默认的浅拷贝。这使得 `a1` 和 `a2` 的指针 `p` 指向同一内存地址，在程序结束析构时会造成同一块内存被释放两次，导致程序错误。
- **(2)**：为类 `A` 实现一个自定义的拷贝构造函数，在函数中进行深拷贝，为新对象的指针成员分配新的内存空间。

- **补充代码如下：**

```
// 添加到 class A 的 public 部分
A(const A& other) {
    x = other.x;
    y = other.y;
    // 深拷贝：为新对象分配新内存
    p = new char[strlen(other.p) + 1];
    // 复制内容，而不是指针地址
    strcpy(p, other.p);
}
```

类的继承

虚继承

作用：主要用于解决**菱形继承（Diamond Problem）**问题。

当一个派生类从两个（或更多）基类继承，而这些基类又共同继承自同一个更基层的基类时，就形成了菱形继承。若不使用虚继承，派生类中会包含多份共同基类的成员，导致访问成员时出现**歧义**。虚继承确保共同基类在最终的派生类中只存在一个实例。

题目：

分析以下代码，指出存在的问题并修复它。

```cpp
#include <iostream>
class Animal {
public:
    int age;
};

class Horse : public Animal {}; // 马是动物
class Donkey : public Animal {}; // 驴是动物

class Mule : public Horse, public Donkey { // 骡子是马和驴的后代
public:
    void setAge(int n) {
        // age = n; // 编译错误!
    }
};
```

解析与答案：

- **问题分析**:
  - Mule 类通过 Horse 和 Donkey 分别继承了 Animal 类。
  - 这导致 Mule 类的对象中包含了**两份** Animal 的子对象，即两份 age 成员。
  - 当在 Mule::setAge 中访问 age 时，编译器无法确定是 Horse::age 还是 Donkey::age ，因此产生**二义性（ambiguity）**错误。
- **解决方法**:
  将 Horse 和 Donkey 对 Animal 的继承方式改为**虚继承**（virtual public）。
- **正确代码**:

```cpp
#include <iostream>

class Animal {
public:
    int age;
};

// 使用虚继承
class Horse : virtual public Animal {};
class Donkey : virtual public Animal {};

class Mule : public Horse, public Donkey {
public:
    void setAge(int n) {
        age = n; // 正确：现在只有一个 age 实例
    }
    int getAge() {
        return age; // 正确
    }
};

int main() {
    Mule m;
    m.setAge(10);
    std::cout << "Mule's age is: " << m.getAge() << std::endl; // 输出 10
    return 0;
}
```

不同继承方式成员的访问权限

基类成员权限	public 继承	protected 继承	private 继承
public	public	protected	private
protected	protected	protected	private
private	在派生类中**不可见**	在派生类中**不可见**	在派生类中**不可见**

总结：

1. 基类的 `private` 成员永远不能被派生类访问。

2. 继承方式决定了基类 `public` 和 `protected` 成员在派生类中的**最高访问权限**。权限只会缩小或保持，不会放大。

题目：

对于以下类定义，指出 `main` 函数中哪些访问是合法的，哪些是非法的，并说明原因。

```
class Base {
private:
    int a = 1;
protected:
    int b = 2;
public:
    int c = 3;
};

class Derived : public Base {
public:
    void test() {
        // ... 在这里访问 a, b, c
    }
};

int main() {
    Derived d;
    // d.a = 10; // 访问 1
    // d.b = 20; // 访问 2
    d.c = 30; // 访问 3
    return 0;
}
```

解析与答案：

- **访问 1 (`d.a = 10;`)：非法**。
  - 原因： `a` 是 `Base` 类的 `private` 成员，它只能被 `Base` 类的成员函数访问，不能被任何外部代码（包括派生类和 `main` 函数）访问。
- **访问 2 (`d.b = 20;`)：非法**。
  - 原因： `b` 是 `Base` 类的 `protected` 成员。在 `public` 继承后，它在 `Derived` 类中仍然是 `protected` 的。 `protected` 成员可以被派生类的成员函数（如 `Derived::test` ）访问，但不能被类外部的代码（如 `main` 函数）访问。
- **访问 3 (`d.c = 30;`)：合法**。

- 原因： `c` 是 `Base` 类的 `public` 成员。在 `public` 继承后，它在 `Derived` 类中仍然是 `public` 的，因此可以被任何外部代码访问。

类的多态

作用：多态允许我们以统一的方式处理不同类型的对象。核心思想是 **"一个接口，多种实现"**。在 C++ 中，运行时多态是通过 **虚函数** 和 **基类指针/引用** 实现的。

多态性的定义和类型：

- **多态性（Polymorphism）**：允许不同类的对象对相同的消息（函数调用）做出不同的响应。
- **静态多态（编译时多态）**：通过函数重载和模板实现，在编译阶段确定调用哪个函数。
- **动态多态（运行时多态）**：通过继承和虚函数实现，在运行时根据对象实际类型确定调用哪个函数。

实现运行时多态的条件：

1. 类之间存在继承关系
2. 基类中声明了虚函数
3. 通过基类指针或引用指向派生类对象，并调用虚函数

题目：

所谓多态性是指：
A. 不同的对象调用不同名称的函数
B. 不同的对象调用相同名称的函数
C. 一个对象调用不同名称的函数
D. 一个对象调用不同名称的对象

解析与答案：

- **答案**: B
- **解释**：
  - **B. 不同的对象调用相同名称的函数：** 正确。这是多态的典型表现。例如，基类指针指向不同的派生类对象，通过该指针调用一个虚函数，会执行各自派生类中该虚函数的版本。
  - 选项 A 描述的是不同对象调用不同函数，这不是多态，只是正常的函数调用。
  - 选项 C 描述的是一个对象调用不同函数，也不是多态，一个对象可以调用它所拥有的多个不同名称的成员函数。
  - 选项 D 的表述不准确，对象之间通常是通过成员函数进行交互。

多态示例：

```
#include <iostream>

class Animal {
public:
    virtual void makeSound() const {
        std::cout << "Some generic animal sound" << std::endl;
    }
};

class Dog : public Animal {
public:
    void makeSound() const override {
        std::cout << "Woof!" << std::endl;
    }
};

class Cat : public Animal {
public:
    void makeSound() const override {
        std::cout << "Meow!" << std::endl;
    }
};

// 通过基类指针调用makeSound函数，实现多态
void animalSound(const Animal* animal) {
    animal->makeSound(); // 根据animal指向的实际对象类型决定调用哪个版本
}

int main() {
    Animal* dog = new Dog();
    Animal* cat = new Cat();
    Animal* animal = new Animal();

    animalSound(dog);    // 输出: Woof!
    animalSound(cat);    // 输出: Meow!
    animalSound(animal); // 输出: Some generic animal sound

    delete dog;
    delete cat;
    delete animal;
```

```
    return 0;
}
```

关键点：

友元函数

作用：友元允许一个类外的函数或另一个类访问该类的 `private` 和 `protected` 成员。它破坏了封装性，但为一些特定场景提供了便利，例如需要访问两个不同类私有成员的全局函数。

特性与规则：

- 友元函数不是类的成员函数
- 友元函数在类中用 `friend` 关键字声明，但通常在类外定义
- 友元函数可以访问该类的所有成员，包括私有和保护成员
- 友元关系不具有传递性（A 是 B 的友元，B 是 C 的友元，A 不自动成为 C 的友元）
- 友元关系不具有继承性（父类的友元不自动成为子类的友元）

题目：

下面对于友元函数描述正确的是：
A. 友元函数的实现必须在类的内部定义
B. 友元函数是类的成员函数
C. 友元函数破坏了类的封装性和隐藏性
D. 友元函数不能访问类的私有成员

解析与答案：

- **答案**: C
- **解释**：
  - **A**：错误。友元函数可以在类内部定义，但通常在类外部定义。
  - **B**：错误。友元函数不是类的成员函数，它没有 `this` 指针。
  - **C**：正确。友元函数可以访问类的私有成员，这在一定程度上确实破坏了封装性。
  - **D**：错误。友元函数的主要目的就是为了能够访问类的私有和保护成员。

友元函数示例：

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

class Box {
private:
    double length;
    double width;
    double height;

public:
    Box(double l = 0, double w = 0, double h = 0) : length(l), width(w), height(h) {}

    // 声明友元函数
    friend double calculateVolume(const Box& box);

    // 重载+运算符的友元函数
    friend Box operator+(const Box& box1, const Box& box2);
};

// 友元函数实现 - 计算体积
double calculateVolume(const Box& box) {
    // 可以直接访问Box的私有成员
    return box.length * box.width * box.height;
}

// 友元函数实现 - 合并两个盒子
Box operator+(const Box& box1, const Box& box2) {
    Box box;
    // 可以直接访问两个Box对象的私有成员
    box.length = box1.length + box2.length;
    box.width = box1.width + box2.width;
    box.height = box1.height + box2.height;
    return box;
}

int main() {
    Box box1(2, 3, 4);
    Box box2(1, 2, 1);

    cout << "Volume of box1: " << calculateVolume(box1) << endl; // 输出: 24
    cout << "Volume of box2: " << calculateVolume(box2) << endl; // 输出: 2
```

```
    Box combinedBox = box1 + box2;
    cout << "Volume of combined box: " << calculateVolume(combinedBox) << endl; // 输出: 126

    return 0;
}
```

题目：

为 `Point` 类设计一个友元函数，用于计算两个 `Point` 对象之间的距离。

```
#include <iostream>
#include <cmath>

class Point {
private:
    double x, y;
public:
    Point(double x_val, double y_val) : x(x_val), y(y_val) {}
    // 在此处声明友元函数
};

// 在此处实现计算距离的函数

int main() {
    Point p1(0, 0);
    Point p2(3, 4);
    // std::cout << "Distance is: " << distance(p1, p2) << std::endl; // 期望输出 5
    return 0;
}
```

解析与答案：

- **思路**:
  - 距离公式需要访问点的 `x` 和 `y` 坐标，而它们是 `private` 的。
  - 将 `distance` 函数声明为 `Point` 类的 `friend`，这样它就可以访问 `p1.x`，`p1.y`，`p2.x`，`p2.y`。
- **完整代码**:

```cpp
#include <iostream>
#include <cmath>

class Point {
private:
    double x, y;
public:
    Point(double x_val, double y_val) : x(x_val), y(y_val) {}

    // 1. 在类定义内声明友元函数
    friend double distance(const Point& p1, const Point& p2);
};

// 2. 在类外实现友元函数（注意：此时它不是成员函数，没有 Point:: 前缀）
double distance(const Point& p1, const Point& p2) {
    //可以直接访问Point类的私有成员
    double dx = p1.x - p2.x;
    double dy = p1.y - p2.y;
    return std::sqrt(dx*dx + dy*dy);
}

int main() {
    Point p1(0, 0);
    Point p2(3, 4);
    std::cout << "Distance is: " << distance(p1, p2) << std::endl; // 输出: Distance is: 5
    return 0;
}
```

运算符重载

作用：允许为自定义类型（类和结构体）的运算赋予特定含义。例如，让 `+` 能够“相加”两个复数对象。

两种方式：

1. **成员函数**：通常用于会改变对象自身状态的运算符（如 `+=`）或需要访问私有成员的二元运算符（如 `+`）。左操作数必须是该类的对象。
2. **全局/友元函数**：通常用于左操作数不是类对象的运算符（如 `cout << obj`），或者希望支持不同类型之间交换运算的场景（如 `2 * complex_obj`）。

28. 编写学生成绩输入输出程序，使用重载操作符>>和<<（10 分）

题目要求：

- 类 StudentScores 定义如下：

```
const int MAX_NUM_OF_COURSES = 30;
const int MAX_ID_LEN = 10;
const int MAX_NAME_LEN = 8;
class StudentScores {
public:
    StudentScores() { initialized = false; }
    bool data_is_ok() const { return initialized; }
private:
    int scores[MAX_NUM_OF_COURSES], num_of_courses;
    char id[MAX_ID_LEN + 1], name[MAX_NAME_LEN + 1];
    bool initialized;
    friend istream &operator>>(istream &in, StudentScores &x);
    friend ostream &operator<<(ostream &out, const StudentScores &x);
};
```

- 从键盘输入学生信息（学号、姓名、选课门数、成绩），保存到文件。
- 使用重载 >> 和 << 实现输入输出。

解答：

实现代码：

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

const int MAX_NUM_OF_COURSES = 30;
const int MAX_ID_LEN = 10;
const int MAX_NAME_LEN = 8;

class StudentScores {
public:
    StudentScores() { initialized = false; }
    bool data_is_ok() const { return initialized; }
private:
    int scores[MAX_NUM_OF_COURSES], num_of_courses;
    char id[MAX_ID_LEN + 1], name[MAX_NAME_LEN + 1];
    bool initialized;
    friend istream &operator>>(istream &in, StudentScores &x);
    friend ostream &operator<<(ostream &out, const StudentScores &x);
};

// 重载输入操作符
istream &operator>>(istream &in, StudentScores &x) {
    in >> x.id >> x.name >> x.num_of_courses;
    for (int i = 0; i < x.num_of_courses; i++) {
        in >> x.scores[i];
    }
    x.initialized = true;  // 数据有效
    return in;
}

// 重载输出操作符
ostream &operator<<(ostream &out, const StudentScores &x) {
    if (!x.initialized) return out;  // 未初始化则不输出
    out << x.id << " " << x.name << " " << x.num_of_courses;
    for (int i = 0; i < x.num_of_courses; i++) {
        out << " " << x.scores[i];
    }
    return out;
}
```

```
// main函数
int main() {
    StudentScores student;
    cout << "Enter student data (id name courses scores): ";
    cin >> student;  // 从键盘输入

    // 保存到文件
    ofstream out("students.txt");
    if (!out) {
        cerr << "Failed to open file!" << endl;
        return 1;
    }
    out << student;
    out.close();

    // 从文件读取并显示
    ifstream in("students.txt");
    if (!in) {
        cerr << "Failed to open file!" << endl;
        return 1;
    }
    StudentScores student2;
    in >> student2;
    cout << "Read from file: " << student2 << endl;
    in.close();

    return 0;
}
```

题目：

为 Complex （复数）类重载 + 运算符以实现复数加法，并重载 << 运算符以方便地用 cout 输出复数。

```
#include <iostream>

class Complex {
private:
    double real, imag;
public:
    Complex(double r = 0, double i = 0) : real(r), imag(i) {}
    // 在这里添加运算符重载
};

int main() {
    Complex c1(2, 3);
    Complex c2(4, -1);
    Complex c3 = c1 + c2; // 使用 + 运算符
    // std::cout << c3 << std::endl; // 期望输出 6+2i
    return 0;
}
```

解析与答案：

- **`+` 运算符重载**：可以作为成员函数。它接受一个 `Complex` 对象作为右操作数，返回一个新的 `Complex` 对象作为结果。
- **`<<` 运算符重载**：**必须作为全局/友元函数**。因为 `cout << c3` 的左操作数是 `ostream` 类型的 `cout`，而不是 `Complex` 对象。为了访问 `Complex` 的私有成员 `real` 和 `imag`，需要将其声明为友元。
- **完整代码**:

```cpp
#include <iostream>

class Complex {
private:
    double real, imag;
public:
    Complex(double r = 0, double i = 0) : real(r), imag(i) {}

    // 1. 重载 + 运算符（作为成员函数）
    Complex operator+(const Complex& other) const {
        return Complex(real + other.real, imag + other.imag);
    }

    // 2. 声明 << 运算符重载为友元
    friend std::ostream& operator<<(std::ostream& os, const Complex& c);
};

// 3. 实现 << 运算符重载（作为全局友元函数）
std::ostream& operator<<(std::ostream& os, const Complex& c) {
    os << c.real;
    if (c.imag >= 0) {
        os << "+";
    }
    os << c.imag << "i";
    return os; // 返回 ostream 引用以支持链式调用 (cout << c1 << c2)
}

int main() {
    Complex c1(2, 3);
    Complex c2(4, -1);
    Complex c3 = c1 + c2; // 调用 c1.operator+(c2)
    std::cout << "c1: " << c1 << std::endl; // 输出 c1: 2+3i
    std::cout << "c2: " << c2 << std::endl; // 输出 c2: 4-1i
    std::cout << "c3: " << c3 << std::endl; // 输出 c3: 6+2i
    return 0;
}
```

模板类

作用：创建“参数化的类”，即定义一个通用的类蓝图，其中的数据类型可以作为参数在创建对象时指定。这极大地提高了代码的**复用性**。例如，可以写一个 `Stack` 模板，用它来创建 `Stack<int>` , `Stack<double>` , `Stack<string>` 等。

题目：

实现一个简单的栈（Stack）模板类，支持 `push` （入栈）、 `pop` （出栈）、 `top` （查看栈顶元素）和 `isEmpty` （判空）操作。并实例化一个整数栈和一个字符串栈进行测试。

解析与答案：

- **思路**:
  - 使用 `template <typename T>` 来声明这是一个模板类， `T` 是一个占位符，代表任意数据类型。
  - 在类内部，使用 `T` 来定义数据成员（如存储元素的数组）和函数的参数/返回值类型。
  - 可以使用 `std::vector` 作为底层容器来简化实现。
- **完整代码**:

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <stdexcept>

template <typename T>
class Stack {
private:
    std::vector<T> elems; // 使用 vector 作为底层容器

public:
    void push(const T& elem) {
        elems.push_back(elem);
    }

    void pop() {
        if (isEmpty()) {
            throw std::out_of_range("Stack<T>::pop(): empty stack");
        }
        elems.pop_back();
    }

    T& top() {
        if (isEmpty()) {
            throw std::out_of_range("Stack<T>::top(): empty stack");
        }
        return elems.back();
    }

    bool isEmpty() const {
        return elems.empty();
    }
};

int main() {
    // --- 实例化一个整数栈 ---
    Stack<int> intStack;
    intStack.push(10);
    intStack.push(20);
    std::cout << "Top of intStack: " << intStack.top() << std::endl; // 输出 20
    intStack.pop();
    std::cout << "Top of intStack after pop: " << intStack.top() << std::endl; // 输出 10
```

```
    std::cout << "--------------------" << std::endl;

    // --- 实例化一个字符串栈 ---
    Stack<std::string> stringStack;
    stringStack.push("Hello");
    stringStack.push("World");
    std::cout << "Top of stringStack: " << stringStack.top() << std::endl; // 输出 World

    return 0;
}
```

流

作用：C++ 的流库提供了一种统一的、类型安全的方式来进行输入/输出操作。最常用的是标准输入输出流（ `cin` ， `cout` ）和**文件流**。

C++标准数据流对象

C++ 标准库在 `<iostream>` 头文件中预定义了四个用于处理标准数据流的对象：

- `cin` : 标准输入流对象，通常关联到键盘。
- `cout` : 标准输出流对象，通常关联到屏幕。
- `cerr` : 标准错误流对象，非缓冲，通常关联到屏幕，用于输出错误信息。
- `clog` : 标准错误流对象，缓冲，通常关联到屏幕，也用于输出错误信息。

题目：

下列选项中不属于 C++系统预定义的标准数据流对象的是：
A. `cout` B. `cin` C. `cerr` D. `cset`

解析与答案：

- **答案**: D. `cset`
- **解释**:
  - `cset` 并非 C++标准库中的预定义流对象。
  - `cerr` 和 `clog` 的区别在于， `cerr` 是非缓冲的，输出会立即显示；而 `clog` 是缓冲的，输出内容可能先存储在缓冲区中，待缓冲区满或特定条件下再实际输出。

文件流关键类：

- `ifstream` (input file stream): 从文件读取。
- `ofstream` (output file stream): 向文件写入。
- `fstream` (file stream): 可读可写。

20. 填写程序，完成文件输入（4 分）

题目代码：

```
#include __(1)___
#include <iostream>
using namespace std;
struct Student {
    int no;
    char name[10];
    int scores[5];
} s1;
void main() {
    // 以二进制方式输入数据
    ifstream in_file("d:\\students.dat", ____(2)____);
    if(_____(3)____) {
        cerr << "Fail to open file" << endl;
        exit(1);
    }
    in_file.read(_____(4)_____, sizeof(s1));
    in_file.close();
}
```

解析：

- **目标**：从文件读取二进制数据到结构体 `s1`。
- **填空**：
  - **(1)**：文件流需包含 `<fstream>`。
  - **(2)**：二进制模式为 `ios::binary`。
  - **(3)**：检查文件打开失败，`!in_file.is_open()`。
  - **(4)**：`read` 需 `char*` 类型指针，转换 `s1` 地址为 `char* (&s1)`。

题目：

编写一个程序，完成以下两个任务：

1. 将字符串 "Hello, C++ File IO!" 写入到名为 `data.txt` 的文件中。

2. 从 `data.txt` 文件中读取内容，并打印到控制台上。

解析与答案：

- **思路**:
  - **写入**：创建一个 `ofstream` 对象并关联到 `data.txt` 文件。使用 `<<` 运算符将字符串写入流。完成后关闭流（或利用析构函数自动关闭）。
  - **读取**：创建一个 `ifstream` 对象并关联到 `data.txt` 文件。使用 `getline` 函数或 `>>` 运算符从流中读取数据。打印到控制台。
- **完整代码**:

```cpp
#include <iostream>
#include <fstream> // 文件流头文件
#include <string>

int main() {
    std::string filename = "data.txt";

    // --- 1. 写入文件 ---
    { // 使用代码块确保 ofs 在块结束时被析构，从而自动关闭文件
        std::ofstream ofs(filename); // 创建输出文件流对象并打开文件
        if (!ofs.is_open()) { // 检查文件是否成功打开
            std::cerr << "Error: Could not open " << filename << " for writing." << std::en
            return 1;
        }

        std::cout << "Writing to file..." << std::endl;
        ofs << "Hello, C++ File IO!" << std::endl;
        ofs << "This is a new line." << std::endl;

        std::cout << "Finished writing." << std::endl;
    } // ofs 在此被销毁，文件自动关闭

    std::cout << "--------------------" << std::endl;

    // --- 2. 读取文件 ---
    {
        std::ifstream ifs(filename); // 创建输入文件流对象并打开文件
        if (!ifs.is_open()) {
            std::cerr << "Error: Could not open " << filename << " for reading." << std::en
            return 1;
        }

        std::cout << "Reading from file:" << std::endl;
        std::string line;
        // 逐行读取文件内容直到文件末尾
        while (std::getline(ifs, line)) {
            std::cout << line << std::endl;
        }
    } // ifs 在此被销毁，文件自动关闭

    return 0;
}
```

构造函数与析构函数

析构函数

析构函数的特征：

- 名称为 `~类名` 。
- 没有返回类型（连 `void` 都不写）。
- 不能有参数。
- 不能被重载（因此一个类只有一个）。
- 通常声明为 `public` 。
- 当对象的生命周期结束时（例如，局部对象离开其作用域， `delete` 指向动态对象的指针），析构函数会被自动调用。

题目：

下列对析构函数的描述中，正确的是：
A. 一个类中只能定义一个析构函数
B. 析构函数名与类名不同
C. 析构函数的定义只能在类体内
D. 析构函数可以有一个或多个参数

解析与答案：

- **答案**: A
- **解释**:
  - **A. 一个类中只能定义一个析构函数：** 正确。析构函数不能重载，因此一个类最多只能有一个析构函数。
  - **B. 析构函数名与类名不同：** 错误。析构函数的名称是在类名前加上波浪号 `~` 。
  - **C. 析构函数的定义只能在类体内：** 错误。析构函数可以在类体内定义，也可以在类体外定义（类内声明，类外通过作用域解析运算符 `::` 实现）。
  - **D. 析构函数可以有一个或多个参数：** 错误。析构函数不能有任何参数。

用途示例：

析构函数特别适合用来释放类对象中指针成员所指向的动态存储空间：

```
class DynamicArray {
private:
    int* data;
    int size;
public:
    DynamicArray(int s) : size(s) {
        data = new int[size]; // 在构造函数中分配动态内存
    }

    ~DynamicArray() {
        delete[] data; // 在析构函数中释放动态内存
        data = nullptr;
    }

    // 其他成员函数...
};
```

this 指针

作用：在类的非静态成员函数内部， `this` 是一个指向调用该函数的对象的指针。它的主要用途包括：

1. 区分同名的成员变量和局部变量/参数
2. 在成员函数中返回对象自身的引用或指针
3. 处理对象的自引用

特性：

- `this` 是一个隐含的参数，不需要在函数定义中声明
- 它的类型是 `类类型* const` （一个指向类类型的常量指针）
- 静态成员函数没有 `this` 指针，因为静态成员函数不与任何特定对象关联

题目：

关于 this 指针使用说法正确的是：
A. 保证每个对象拥有自己的数据成员,但共享处理这些数据的代码
B. 保证基类私有成员在子类中可以被访问。
C. 保证基类保护成员在子类中可以被访问。
D. 保证基类公有成员在子类中可以被访问。

解析与答案：

- **答案**: A
- **解释**:
  - **A. 保证每个对象拥有自己的数据成员,但共享处理这些数据的代码：** 正确。 `this` 指针使得同一个类的不同对象可以共享成员函数代码，但同时每个对象都有其独立的数据成员。当类的成员函数被调用时， `this` 指针指向调用该函数的对象。
  - **B、C、D 都是错误的。** `this` 指针与继承层次中的访问控制无关。基类成员在子类中的可访问性由访问修饰符（ `private` 、 `protected` 、 `public` ）决定。

例子：

```
class Counter {
private:
    int count;
public:
    Counter() : count(0) {}

    // 使用this指针返回对象自身的引用，实现链式调用
    Counter& increment() {
        this->count++; // 或简写为 count++
        return *this;  // 返回当前对象的引用
    }

    // 解决名称冲突
    void setValue(int count) {
        this->count = count; // 区分成员变量和参数
    }

    int getCount() const {
        return count;
    }
};

int main() {
    Counter c;
    c.increment().increment().increment(); // 链式调用
    std::cout << c.getCount() << std::endl; // 输出: 3
    return 0;
}
```

继承中的构造与析构顺序

构造函数执行顺序：当创建派生类对象时，构造函数的调用顺序如下：

1. **基类构造函数**：先执行基类的构造函数
2. **成员对象构造函数**：按照成员对象在类中声明的顺序
3. **派生类自身构造函数体**：最后执行派生类构造函数的代码

析构函数执行顺序：与构造函数顺序相反

1. **派生类析构函数体**：先执行
2. **成员对象析构函数**：按照成员对象在类中声明的逆序
3. **基类析构函数**：最后执行

题目：

派生类构造函数的执行顺序是先执行**____** 的构造函数，然后执行成员对象的构造函数，最后执行____ 的构造函数。

解析与答案：

- **答案**：基类；派生类自身

示例：

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

class Base {
public:
    Base() {
        cout << "Base constructor" << endl;
    }
    ~Base() {
        cout << "Base destructor" << endl;
    }
};

class Member {
public:
    Member() {
        cout << "Member constructor" << endl;
    }
    ~Member() {
        cout << "Member destructor" << endl;
    }
};

class Derived : public Base {
private:
    Member m; // 成员对象
public:
    Derived() {
        cout << "Derived constructor" << endl;
    }
    ~Derived() {
        cout << "Derived destructor" << endl;
    }
};

int main() {
    {
        cout << "Creating Derived object:" << endl;
        Derived d;
        cout << "Derived object created." << endl;
    } // 作用域结束，d的生命周期结束
```

```
    return 0;
}
/*
输出：
Creating Derived object:
Base constructor
Member constructor
Derived constructor
Derived object created.
Derived destructor
Member destructor
Base destructor
*/
```

重点注意：

- 如果存在多重继承，基类构造函数的调用顺序取决于它们在派生类继承列表中的声明顺序
- 如果存在虚继承，虚基类的构造函数会优先被调用，并且只调用一次

引用成员和类的构造

引用成员

特性：

- 引用成员必须在构造函数的初始化列表中初始化
- 引用一旦初始化，不能再改变其绑定的对象
- 不能有引用数组
- 引用不能为 `NULL` ，必须始终引用到一个有效的对象

题目 1：

对类中引用成员的初始化只能通过在构造函数中给出的**_____**来实现。