高程2017卷

程序分析题

第一问

以下是给定程序的语法错误及其原因的详细解释：

```
class A
{
    int m1;
    static int m2;
    const int m3;

public:
    A()
    {
        m1 = 0;
        m3 = 100;  // 错误：不能在构造函数体中初始化const成员
    }
    void f1()
    {
        m1 += m2 + m3;
    }
    static void f2()
    {
        m1++;  // 错误：静态成员函数不能访问非静态成员
        m2++;
    }
    void f3() const
    {
        m1 -= m2 + m3;  // 错误：const成员函数不能修改非静态成员变量
    }
};

int A::m2 = 0;
void func(const A &x)
{
    x.f1();  // 错误：f1不是const成员函数，不能通过const引用调用
    x.f2();  // 错误：静态成员函数通过类名调用
    x.f3();
}

int main()
{
    A a;
    a.f1();
    a.f2();  // 错误：静态成员函数通过类名调用
    a.f3();
    func(a);
    return 0;
}
```

错误及其原因

1. **构造函数初始化const成员的错误**

```
A()
{
    m1 = 0;
    m3 = 100;  // 错误：不能在构造函数体中初始化const成员
}
```

原因：`const`成员变量必须在构造函数的初始化列表中初始化，而不是在构造函数体中。

修改方法：

```
A() : m3(100)
{
    m1 = 0;
}
```

2. **静态成员函数访问非静态成员的错误**

```
static void f2()
{
    m1++;  // 错误：静态成员函数不能访问非静态成员
    m2++;
}
```

原因：静态成员函数没有`this`指针，不能访问非静态成员变量。

修改方法：删除对`m1`的访问或通过对象来访问`m1`。

3. **const成员函数修改非静态成员变量的错误**

```
void f3() const
{
    m1 -= m2 + m3;  // 错误：const成员函数不能修改非静态成员变量
}
```

原因：`const`成员函数不能修改类的非静态数据成员。

修改方法：删除对`m1`的修改，或移除`const`限定符（如果逻辑允许）。

4. **const引用调用非const成员函数的错误**

```
void func(const A &x)
{
    x.f1();  // 错误：f1不是const成员函数，不能通过const引用调用
    x.f2();  // 错误：静态成员函数通过类名调用
    x.f3();
}
```

原因：`x`是`const`引用，不能调用非`const`成员函数`f1`。此外，静态成员函数应通过类名调用。

修改方法：

```
void func(const A &x)
{
    // x.f1();  // 删除或修改为const成员函数
    A::f2();  // 静态成员函数通过类名调用
    x.f3();
}
```

5. **静态成员函数通过类名调用**

```
a.f2();  // 错误：静态成员函数通过类名调用
```

原因：静态成员函数应通过类名调用，而不是对象调用。

修改方法：

```
A::f2();
```

修改后的代码

以下是修改后的代码，修复了所有语法错误：

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

class A
{
    int m1;
    static int m2;
    const int m3;

public:
    A() : m3(100)
    {
        m1 = 0;
    }
    void f1()
    {
        m1 += m2 + m3;
    }
    static void f2()
    {
        // 静态成员函数不能访问非静态成员 m1
        m2++;
    }
    void f3() const
    {
        // const成员函数不能修改非静态成员变量 m1
    }
};

int A::m2 = 0;

void func(const A &x)
{
    // x.f1();  // 非const成员函数，不能通过const引用调用
    A::f2();  // 静态成员函数通过类名调用
```

```
    x.f3();
}

int main()
{
    A a;
    a.f1();
    A::f2();  // 静态成员函数通过类名调用
    a.f3();
    func(a);
    return 0;
}
```

此代码现在应该能够编译并正确运行。

第二问

是的，上述程序存在潜在问题，特别是在赋值操作方面。下面是详细分析和解决方法。

程序问题分析

1. 类 B 的赋值操作问题

当类 `B` 对象 `b1` 被赋值给 `b2` 时，默认的赋值操作不会正确处理指针成员 `p`，这会导致多个 `B` 对象共享同一个 `A` 对象，进而可能引发双重释放或未定义行为。

在当前的实现中，默认的赋值操作是浅拷贝，即直接复制指针的值而不是指针指向的对象。这意味着 `b1` 和 `b2` 在赋值后指向同一个 `A` 对象。

2. 缺乏拷贝构造函数和赋值操作符

为了解决上述问题，我们需要为类 `B` 自定义拷贝构造函数和赋值操作符，以实现深拷贝。

修改代码

为了避免上述问题，需要在类 `B` 中添加拷贝构造函数和赋值操作符的自定义实现。下面是修改后的代码：

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

class A
{
    int i;
    int j;

public:
    A() { i = j = 0; };
    void f()
    {
        i++;
        j++;
    }
};

class B
{
    A *p;
```

```
public:
    B()
    {
        p = new A;
    }

    // 拷贝构造函数
    B(const B &other)
    {
        p = new A(*other.p);
    }

    // 赋值操作符
    B &operator=(const B &other)
    {
        if (this != &other)
        {
            delete p; // 释放已有资源
            p = new A(*other.p);
        }
        return *this;
    }

    ~B()
    {
        delete p;
        p = nullptr;
    }
    void g() { p->f(); }
};

void func(B x)
{
    x.g();
}

int main()
{
    B b1, b2;
    func(b1);
    b2 = b1;
    return 0;
}
```

修改说明

1. **拷贝构造函数**

   ```
   B(const B &other)
   {
       p = new A(*other.p);
   }
   ```

   当一个 `B` 对象被拷贝构造时，通过 `new` 创建一个新的 `A` 对象，并用 `other.p` 指向的 `A` 对象进行初始化，实现深拷贝。

2. **赋值操作符**

```
B &operator=(const B &other)
{
    if (this != &other)
    {
        delete p; // 释放已有资源
        p = new A(*other.p);
    }
    return *this;
}
```

在赋值操作符中，首先检查自我赋值，如果不是自我赋值，则删除当前对象的 `p` 所指向的 `A` 对象，然后创建一个新的 `A` 对象，并用 `other.p` 指向的 `A` 对象进行初始化，最后返回当前对象。

结论

通过添加拷贝构造函数和赋值操作符的自定义实现，解决了类 `B` 的对象在拷贝和赋值时可能出现的资源管理问题。现在程序能够正确处理 `B` 类对象的赋值操作，避免了潜在的双重释放问题。

第三问

这段代码定义了三个类：`Passenger`、`Car` 和 `Taxi`，其中 `Taxi` 类继承了 `Car` 类，并包含一个 `Passenger` 类的成员变量。以下是代码的详细解释和输出结果。

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

class Passenger {
    int num;
public:
    Passenger() : num(0) {
        cout << "in Passenger's default constructor" << endl;
    }
    Passenger(int i) : num(i) {
        cout << "in Passenger's par constructor" << endl;
    }
    Passenger(const Passenger& psg) : num(psg.num) {
        cout << "in Passenger's copy constructor" << endl;
    }
    ~Passenger() {
        cout << "in Passenger's destructor" << endl;
    }
};

class Car {
    int seated;
public:
    Car() : seated(0) {
        cout << "in Car's default constructor" << endl;
    }
    Car(int i) : seated(i) {
        cout << "in Car's par constructor" << endl;
    }
    Car(const Car& ca) : seated(ca.seated) {
        cout << "in Car's copy constructor" << endl;
```

```
    }
    ~Car() {
        cout << "in Car's destructor" << endl;
    }
};

class Taxi : public Car {
    Passenger pg;
public:
    Taxi() {
        cout << "in Taxi's default constructor" << endl;
    }
    Taxi(int i) : Car(i) {
        cout << "in Taxi's par constructor" << endl;
    }
    Taxi(int i, int j) : Car(i), pg(j) {
        cout << "in Taxi's two par constructor" << endl;
    }
    Taxi(const Taxi& tx) : Car(tx), pg(tx.pg) {
        cout << "in Taxi's copy constructor" << endl;
    }
    ~Taxi() {
        cout << "in Taxi's destructor" << endl;
    }
};

void convey(Taxi x) {
    cout << "In convey" << endl;
}

int main() {
    cout << "Section1" << endl;
    Taxi t1;
    cout << "Section2" << endl;
    Taxi t2(5);
    cout << "Section3" << endl;
    Taxi t3(5, 7);
    cout << "Section4" << endl;
    convey(t1);
    cout << "Section5" << endl;
    return 0;
}
```

代码解释及输出

1. **Section1**:

   ```
   cout << "Section1" << endl;
   Taxi t1;
   ```

   - `Taxi t1;` 调用 `Taxi` 类的默认构造函数。
   - `Taxi` 类的默认构造函数调用 `Car` 类和 `Passenger` 类的默认构造函数。

   输出:

```
Section1
in Car's default constructor
in Passenger's default constructor
in Taxi's default constructor
```

2. **Section2**:

```
cout << "Section2" << endl;
Taxi t2(5);
```

   - `Taxi t2(5);` 调用 `Taxi` 类的单参数构造函数，传递一个 `int` 值给 `Car` 类的单参数构造函数。

   输出：

```
Section2
in Car's par constructor
in Passenger's default constructor
in Taxi's par constructor
```

3. **Section3**:

```
cout << "Section3" << endl;
Taxi t3(5, 7);
```

   - `Taxi t3(5, 7);` 调用 `Taxi` 类的双参数构造函数，传递两个 `int` 值分别给 `Car` 类和 `Passenger` 类的单参数构造函数。

   输出：

```
Section3
in Car's par constructor
in Passenger's par constructor
in Taxi's two par constructor
```

4. **Section4**:

```
cout << "Section4" << endl;
convey(t1);
```

   - `convey(t1);` 将 `t1` 传递给 `convey` 函数，调用 `Taxi` 类的拷贝构造函数。
   - `Taxi` 类的拷贝构造函数调用 `Car` 类和 `Passenger` 类的拷贝构造函数。

   输出：

```
Section4
in Car's copy constructor
in Passenger's copy constructor
in Taxi's copy constructor
In convey
in Taxi's destructor
in Car's destructor
in Passenger's destructor
```

5. **Section5**:

```
cout << "Section5" << endl;
return 0;
```

   - 程序结束时，`main` 函数作用域内的所有对象被销毁，依次调用它们的析构函数。

   输出：

```
Section5
in Taxi's destructor
in Car's destructor
in Passenger's destructor
in Taxi's destructor
in Car's destructor
in Passenger's destructor
in Taxi's destructor
in Car's destructor
in Passenger's destructor
```

总结输出

完整输出如下：

```
Section1
in Car's default constructor
in Passenger's default constructor
in Taxi's default constructor
Section2
in Car's par constructor
in Passenger's default constructor
in Taxi's par constructor
Section3
in Car's par constructor
in Passenger's par constructor
in Taxi's two par constructor
Section4
in Car's copy constructor
in Passenger's copy constructor
in Taxi's copy constructor
In convey
in Taxi's destructor
in Car's destructor
in Passenger's destructor
Section5
in Taxi's destructor
in Car's destructor
in Passenger's destructor
in Taxi's destructor
in Car's destructor
in Passenger's destructor
in Taxi's destructor
in Car's destructor
in Passenger's destructor
```

每个构造函数和析构函数的调用顺序反映了对象创建和销毁的顺序，其中先调用基类的构造函数，再调用成员对象的构造函数，最后调用派生类的构造函数。析构函数调用的顺序则相反。

补充

如果Car类中没有Passenger类的成员变量pg，是不是就不会调用Passenger的构造函数了

是的，如果 `Car` 类中没有 `Passenger` 类的成员变量 `pg`，那么在构造 `Taxi` 对象时就不会调用 `Passenger` 类的构造函数。

让我们详细解释这一点。

原代码分析

当前的 `Taxi` 类包含一个 `Passenger` 类的成员变量 `pg`：

```
class Taxi : public Car {
    Passenger pg;
public:
    Taxi() {
        cout << "in Taxi's default constructor" << endl;
    }
    // 其他构造函数和成员函数
};
```

当我们创建 `Taxi` 对象时，比如 `Taxi t1;`，执行的顺序如下：

1. 调用基类 `Car` 的构造函数。
2. 调用成员变量 `Passenger pg` 的构造函数。
3. 调用 `Taxi` 类的构造函数。

这个顺序保证了所有基类部分和成员变量在派生类部分初始化之前都已经被正确初始化。

修改代码去掉 `Passenger` 成员变量

如果我们去掉 `Taxi` 类中的 `Passenger` 成员变量 `pg`，代码如下：

```
class Taxi : public Car {
public:
    Taxi() {
        cout << "in Taxi's default constructor" << endl;
    }
    Taxi(int i) : Car(i) {
        cout << "in Taxi's par constructor" << endl;
    }
    Taxi(const Taxi& tx) : Car(tx) {
        cout << "in Taxi's copy constructor" << endl;
    }
    ~Taxi() {
        cout << "in Taxi's destructor" << endl;
    }
};
```

修改后代码的输出

让我们假设修改后的代码与之前的 `main` 函数一致：

```
int main() {
    cout << "Section1" << endl;
    Taxi t1;
    cout << "Section2" << endl;
    Taxi t2(5);
    cout << "Section3" << endl;
    Taxi t3(5, 7);
    cout << "Section4" << endl;
    convey(t1);
    cout << "Section5" << endl;
    return 0;
}
```

运行输出

1. **Section1**:

   ```
   cout << "Section1" << endl;
   Taxi t1;
   ```

   - `Taxi t1;` 调用 `Taxi` 类的默认构造函数。
   - `Taxi` 类的默认构造函数调用 `Car` 类的默认构造函数。

   输出：

   ```
   Section1
   in Car's default constructor
   in Taxi's default constructor
   ```

2. **Section2**:

   ```
   cout << "Section2" << endl;
   Taxi t2(5);
   ```

   - `Taxi t2(5);` 调用 `Taxi` 类的单参数构造函数，传递一个 `int` 值给 `Car` 类的单参数构造函数。

   输出：

   ```
   Section2
   in Car's par constructor
   in Taxi's par constructor
   ```

3. **Section3**:

   ```
   cout << "Section3" << endl;
   Taxi t3(5, 7);
   ```

   - `Taxi t3(5, 7);` 调用 `Taxi` 类的单参数构造函数，传递一个 `int` 值给 `Car` 类的单参数构造函数。注意这里已经没有第二个参数用于 `Passenger` 的初始化，因为我们移除了

`Passenger` 成员变量。

输出：

```
Section3
in Car's par constructor
in Taxi's par constructor
```

4. **Section4**：

```
cout << "Section4" << endl;
convey(t1);
```

- `convey(t1);` 将 `t1` 传递给 `convey` 函数，调用 `Taxi` 类的拷贝构造函数。
- `Taxi` 类的拷贝构造函数调用 `Car` 类的拷贝构造函数。

输出：

```
Section4
in Car's copy constructor
in Taxi's copy constructor
In convey
in Taxi's destructor
in Car's destructor
```

5. **Section5**：

```
cout << "Section5" << endl;
return 0;
```

- 程序结束时，`main` 函数作用域内的所有对象被销毁，依次调用它们的析构函数。

输出：

```
Section5
in Taxi's destructor
in Car's destructor
in Taxi's destructor
in Car's destructor
in Taxi's destructor
in Car's destructor
```

总结输出

修改后的完整输出如下：

```
Section1
in Car's default constructor
in Taxi's default constructor
Section2
in Car's par constructor
in Taxi's par constructor
Section3
in Car's par constructor
in Taxi's par constructor
```

```
Section4
in Car's copy constructor
in Taxi's copy constructor
In convey
in Taxi's destructor
in Car's destructor
Section5
in Taxi's destructor
in Car's destructor
in Taxi's destructor
in Car's destructor
in Taxi's destructor
in Car's destructor
```

所以，如果 `Car` 类中没有 `Passenger` 类的成员变量 `pg`，那么在构造 `Taxi` 对象时确实不会调用 `Passenger` 的构造函数。构造和析构过程将会如上所示，只涉及 `Car` 和 `Taxi` 类的构造函数和析构函数。

总结

构造函数的调用顺序总是：

1. 基类的构造函数
2. 成员变量的构造函数（按它们在类中声明的顺序）
3. 派生类的构造函数

对于析构函数，调用顺序则相反：首先调用派生类的析构函数，然后依次调用成员变量的析构函数，最后调用基类的析构函数。这确保了派生类和成员变量在销毁前有机会清理自己依赖的资源。

第四问

这段代码展示了C++中的类继承、多态、虚函数以及构造函数和析构函数的调用顺序。我们将逐个部分分析代码，并解释每个部分的输出。

代码分析

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

class A {
    int x, y;
public:
    A() { cout << "in A's default constructor\n"; }
    ~A() { cout << "in A's destructor\n"; }
    virtual void f() { cout << "in A's f\n"; }
    void g() { cout << "in A's g\n"; }
};

class B: public A {
    int z;
public:
    B() { cout << "in B's default constructor\n"; }
    ~B() { cout << "in B's destructor\n"; }
    void f() { cout << "in B's f\n"; }
    void g() { cout << "in B's g\n"; }
};
```

```
void func(A *pa) {
    pa->f();
    pa->g();
}

int main() {
    cout << "-----Section 1-----\n";
    A a;
    func(&a);

    cout << "-----Section 2-----\n";
    B b;
    func(&b);

    cout << "-----Section 3-----\n";
    A *p = new B;
    func(p);

    cout << "-----Section 4-----\n";
    delete p;

    cout << "-----Section 5-----\n";
    return 0;
}
```

输出和解释

Section 1

```
cout << "-----Section 1-----\n";
A a;
func(&a);
```

1. `A a;` 创建了一个 `A` 类的对象。
    - 调用 `A` 的默认构造函数，输出 `in A's default constructor\n`。
2. `func(&a);` 调用 `func` 函数，传递 `a` 的地址。
    - `func` 函数调用 `pa->f();`，由于 `a` 是 `A` 类对象，所以调用 `A` 的 `f` 函数，输出 `in A's f\n`。
    - `func` 函数调用 `pa->g();`，由于 `g` 函数不是虚函数，直接调用 `A` 的 `g` 函数，输出 `in A's g\n`。

```
-----Section 1-----
in A's default constructor
in A's f
in A's g
```

Section 2

```
cout << "-----Section 2-----\n";
B b;
func(&b);
```

1. `B b;` 创建了一个 `B` 类的对象。

- 调用 `A` 的默认构造函数，输出 `in A's default constructor\n`。
- 调用 `B` 的默认构造函数，输出 `in B's default constructor\n`。

2. `func(&b);` 调用 `func` 函数，传递 `b` 的地址。
   - `func` 函数调用 `pa->f();`，由于 `f` 是虚函数，调用 `B` 的 `f` 函数，输出 `in B's f\n`。
   - `func` 函数调用 `pa->g();`，由于 `g` 不是虚函数，调用 `A` 的 `g` 函数，输出 `in A's g\n`。

```
-----Section 2-----
in A's default constructor
in B's default constructor
in B's f
in A's g
```

Section 3

```
cout << "-----Section 3-----\n";
A *p = new B;
func(p);
```

1. `A *p = new B;` 动态创建一个 `B` 类的对象，并将其地址赋值给 `A` 类的指针 `p`。
   - 调用 `A` 的默认构造函数，输出 `in A's default constructor\n`。
   - 调用 `B` 的默认构造函数，输出 `in B's default constructor\n`。
2. `func(p);` 调用 `func` 函数，传递 `p` 的地址。
   - `func` 函数调用 `pa->f();`，由于 `f` 是虚函数，调用 `B` 的 `f` 函数，输出 `in B's f\n`。
   - `func` 函数调用 `pa->g();`，由于 `g` 不是虚函数，调用 `A` 的 `g` 函数，输出 `in A's g\n`。

```
-----Section 3-----
in A's default constructor
in B's default constructor
in B's f
in A's g
```

Section 4

```
cout << "-----Section 4-----\n";
delete p;
```

1. 由于A的析构函数不是虚函数，`delete p`只调用A的析构函数，而不会调用B的析构函数。这就是为什么你只看到`in A's destructor`。

```
-----Section 4-----
in A's destructor
```

Section 5

```
cout << "-----Section 5-----\n";
return 0;
```

1. 输出 `-----Section 5-----\n`
2. `b` 离开作用域，调用 `B` 的析构函数，输出 `in B's destructor\n`。

接着调用 `A` 的析构函数，输出 `in A's destructor\n`。

3. `a` 离开作用域，调用 `A` 的析构函数，输出 `in A's destructor\n`。

```
-----Section 5-----
in B's destructor
in A's destructor
in A's destructor
```

总结

完整的输出结果如下：

```
-----Section 1-----
in A's default constructor
in A's f
in A's g
-----Section 2-----
in A's default constructor
in B's default constructor
in B's f
in A's g
-----Section 3-----
in A's default constructor
in B's default constructor
in B's f
in A's g
-----Section 4-----
in A's destructor
-----Section 5-----
in B's destructor
in A's destructor
in A's destructor
```

第五问

这段代码定义了一个名为 `A` 的类，并在 `main` 函数中创建了两个该类的对象（`a1` 和 `a2`），并对它们调用了一些成员函数。我们先来看代码，然后解释其输出。

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

class A {
    int m, n;
public:
    A() { m = n = 0; }
    A inc1() {
        m++;
        n++;
        return *this;
    }
    A& inc2() {
        m++;
        n++;
        return *this;
    }
```

```
    void dec() {
        m--;
        n--;
    }
    void show() {
        cout << m << "," << n << endl;
    }
};

int main() {
    A a1;
    a1.inc1().dec();
    a1.show();
    A a2;
    a2.inc2().dec();
    a2.show();
    return 0;
}
```

分析

1. **Class A**:
   - **成员变量**: `int m, n` 用于存储两个整数。
   - **构造函数**: `A()` 初始化成员变量 `m` 和 `n` 为 0。
   - **成员函数**:
     - `A inc1()`: 自增 `m` 和 `n` 的值，并返回当前对象的一个拷贝。
     - `A& inc2()`: 自增 `m` 和 `n` 的值，并返回当前对象的引用。
     - `void dec()`: 自减 `m` 和 `n` 的值。
     - `void show()`: 输出 `m` 和 `n` 的值。
2. **main 函数**:
   - **a1 对象**:

     ```
     A a1;
     a1.inc1().dec();
     a1.show();
     ```

     - `A a1;` 创建一个 `A` 类的对象 `a1`，初始化 `m` 和 `n` 为 0。
     - `a1.inc1()` 调用 `inc1` 成员函数，这会将 `m` 和 `n` 自增 1，变为 1。但 `inc1` 返回的是当前对象的拷贝（而不是引用）。
     - `.dec()` 调用的是拷贝对象的 `dec` 函数，这会将拷贝对象的 `m` 和 `n` 自减 1，但不会影响原对象 `a1`。
     - `a1.show()` 输出 `a1` 的 `m` 和 `n` 的值，仍然是 1。
   - **a2 对象:**

     ```
     A a2;
     a2.inc2().dec();
     a2.show();
     ```

     - `A a2;` 创建一个 `A` 类的对象 `a2`，初始化 `m` 和 `n` 为 0。
     - `a2.inc2()` 调用 `inc2` 成员函数，这会将 `m` 和 `n` 自增 1，变为 1。由于 `inc2` 返回的是当前对象的引用，所以后续的 `.dec()` 操作直接作用于 `a2` 本身。

- `.dec()` 将 `a2` 的 `m` 和 `n` 自减 1，变回 0。
- `a2.show()` 输出 `a2` 的 `m` 和 `n` 的值，为 0。

输出结果

```
1,1
0,0
```

1. `a1.inc1().dec(); a1.show();` 的输出是 `1,1`。
2. `a2.inc2().dec(); a2.show();` 的输出是 `0,0`。

实现下面的程序

第一问

编写一个程序，把文本文件abc.txt中所有的行加上行号，然后写到文件abc2.txt中去。例如：

(abc.txt)	(abc2.txt)
first line	1 first line
second line	2 second line
third line	3 third line
.....	.....

为了实现这个程序，可以使用 C++ 文件输入输出流库 `<fstream>`。这个程序将打开一个名为 `abc.txt` 的文件，读取其中的每一行，并将行号添加到行的开头，然后将这些带行号的行写入一个名为 `abc2.txt` 的新文件中。

以下是实现这个功能的 C++ 代码示例：

```cpp
#include <iostream>
#include <fstream>
#include <string>

int main() {
    // 打开输入文件 abc.txt
    std::ifstream inputFile("abc.txt");
    if (!inputFile.is_open()) {
        std::cerr << "Failed to open input file." << std::endl;
        return 1;
    }

    // 打开输出文件 abc2.txt
    std::ofstream outputFile("abc2.txt");
    if (!outputFile.is_open()) {
        std::cerr << "Failed to open output file." << std::endl;
        return 1;
    }

    std::string line;
    int lineNumber = 1;

    // 逐行读取输入文件并写入输出文件
    while (std::getline(inputFile, line)) {
        outputFile << lineNumber << " " << line << std::endl;
```

```
        lineNumber++;
    }

    // 关闭文件
    inputFile.close();
    outputFile.close();

    std::cout << "Lines have been numbered and written to abc2.txt." <<
std::endl;

    return 0;
}
```

代码说明

1. **打开输入文件**:
   - 使用 `std::ifstream` 打开名为 `abc.txt` 的文件。
   - 如果文件打开失败，输出错误消息并返回。
2. **打开输出文件**:
   - 使用 `std::ofstream` 打开名为 `abc2.txt` 的文件。
   - 如果文件打开失败，输出错误消息并返回。
3. **逐行读取和处理**:
   - 使用 `std::getline` 从输入文件中逐行读取内容。
   - 将行号添加到每一行的开头，并写入到输出文件中。
   - 行号从 1 开始，每处理一行，行号递增。
4. **关闭文件**:
   - 处理完毕后，关闭输入和输出文件。

运行示例

假设 `abc.txt` 文件内容如下：

```
first line
second line
third line
```

运行上述程序后，生成的 `abc2.txt` 文件内容将会是：

```
1 first line
2 second line
3 third line
```

该程序正确地为每一行添加了行号，并将结果写入到新的文件中。

对于将文本1写到文本2的代码的详解

1. **声明变量**

```
std::string line;
int lineNumber = 1;
```

- `std::string line;`：声明一个字符串变量 `line`，用于存储每次读取的一行内容。
- `int lineNumber = 1;`：声明并初始化一个整数变量 `lineNumber` 为 1，用于记录行号。

1. **逐行读取输入文件并写入输出文件**

```
while (std::getline(inputFile, line)) {
    outputFile << lineNumber << " " << line << std::endl;
    lineNumber++;
}
```

- `while (std::getline(inputFile, line))`:
  - `std::getline` 是一个标准库函数，用于从输入流中读取一行文本。
  - `inputFile` 是输入文件流，`line` 是用来存储读取行内容的字符串。
  - 该函数读取一行内容（直到遇到换行符 `\n`），并将其存储在 `line` 中。
  - 如果读取成功（即没有到达文件末尾），`std::getline` 返回 `true`，否则返回 `false`。因此，这个 `while` 循环在文件中还有行可读取时继续执行。
- `outputFile << lineNumber << " " << line << std::endl;`:
  - `outputFile` 是输出文件流。
  - `<<` 操作符用于将数据写入到文件。
  - `lineNumber` 是当前行的行号。
  - `" "` 是一个空格，用于分隔行号和行内容。
  - `line` 是当前读取的行内容。
  - `std::endl` 用于在输出文件中插入一个换行符。
- `lineNumber++`:
  - 这是后缀递增运算符，将 `lineNumber` 的值加 1。下一次循环时，行号会增加，确保每一行都有正确的行号。

整体流程

1. 代码首先声明一个字符串变量 `line` 来存储每次读取的一行内容，声明一个整型变量 `lineNumber` 初始化为 1 用于记录行号。
2. 使用 `std::getline` 函数从输入文件流 `inputFile` 中逐行读取内容，存储到 `line` 中。
3. 读取成功后，将当前行号 `lineNumber`、一个空格和读取的行内容 `line` 写入输出文件流 `outputFile`。
4. 使用 `lineNumber++` 将行号递增，以便下一行记录正确的行号。
5. 循环继续，直到 `std::getline` 不能再读取新行（即文件结束），结束循环。

`std::getline` 函数

- `std::getline` 是一个非常有用的函数，可以读取输入流中的一行直到换行符（`'\n'`）为止。
- 使用 `std::getline(inputFile, line)` 可以简化从文件中逐行读取的操作，不需要手动处理换行符。

`std::cerr` 函数

`std::cerr` 是 C++ 标准库中用于输出错误信息的流。相对于 `std::cout`，它有一些独特的特性和用途。以下是详细解释为什么使用 `std::cerr` 以及它与 `std::cout` 的区别：

`std::cerr` 的特点和用途

1. 错误输出：
   - `std::cerr` 专门用于输出错误消息和诊断信息。这使得程序的错误输出与正常输出分开，便于用户区分和处理。
2. 无缓冲：

- `std::cerr` 默认是无缓冲的，这意味着数据在输出时立即被发送到输出设备（通常是控制台）。这对及时显示错误消息很有帮助，尤其是在程序崩溃或出现严重错误时。
- 反之，`std::cout` 默认是有缓冲的，只有在缓冲区满或遇到换行符（`'\n'`）、显式刷新缓冲区（如调用 `std::flush`）、或程序正常结束时，数据才会输出到设备。

第二问

设计一个数组类模板Array(不允许用C++的STL),使其能用于下面的程序：

```
#include <iostream>
using namespace std;
int main(){
    int m,n;
    cin>>m;
    Array<int> a(m);
    cin>>a;
    cout a.max();
    cout a.sum();
    a.sort();
    cout<<a;
    cin>>n;
    a.resize(n);
    cout<<a;
    return 0;
}
```

要设计一个数组类模板 `Array<T>`，需要实现以下功能：

1. 动态分配内存以存储数组元素。
2. 输入和输出操作。
3. 查找数组的最大值和总和。
4. 对数组进行排序。
5. 调整数组大小。

以下是实现这个模板类的详细代码：

```
#include <iostream>
#include <algorithm>

template <class T>
class Array {
private:
    T* data;
    int size;

public:
    // 构造函数
    Array(int s) : size(s) {
        data = new T[size];
    }

    // 拷贝构造函数
    Array(const Array<T>& other) : size(other.size) {
        data = new T[size];
        for (int i = 0; i < size; ++i) {
            data[i] = other.data[i];
```

```
        }
    }

    // 赋值操作符
    Array& operator=(const Array<T>& other) {
        if (this != &other) {
            delete[] data;
            size = other.size;
            data = new T[size];
            for (int i = 0; i < size; ++i) {
                data[i] = other.data[i];
            }
        }
        return *this;
    }

    // 析构函数
    ~Array() {
        delete[] data;
    }

    // 输入操作符
    friend std::istream& operator>>(std::istream& is, Array<T>& arr) {
        for (int i = 0; i < arr.size; ++i) {
            is >> arr.data[i];
        }
        return is;
    }

    // 输出操作符
    friend std::ostream& operator<<(std::ostream& os, const Array<T>& arr) {
        for (int i = 0; i < arr.size; ++i) {
            os << arr.data[i] << " ";
        }
        return os;
    }

    // 查找最大值
    T max() const {
        T maxVal = data[0];
        for (int i = 1; i < size; ++i) {
            if (data[i] > maxVal) {
                maxVal = data[i];
            }
        }
        return maxVal;
    }

    // 计算总和
    T sum() const {
        T total = 0;
        for (int i = 0; i < size; ++i) {
            total += data[i];
        }
        return total;
    }

    // 排序
```

```
    void sort() {
        std::sort(data, data + size);
    }

    // 调整大小
    void resize(int newSize) {
        T* newData = new T[newSize];
        for (int i = 0; i < std::min(size, newSize); ++i) {
            newData[i] = data[i];
        }
        delete[] data;
        data = newData;
        size = newSize;
    }
};

int main() {
    int m, n;
    std::cin >> m;
    Array<int> a(m);
    std::cin >> a;
    std::cout << a.max() << std::endl;
    std::cout << a.sum() << std::endl;
    a.sort();
    std::cout << a << std::endl;
    std::cin >> n;
    a.resize(n);
    std::cout << a << std::endl;
    return 0;
}
```

功能说明

1. **构造函数**：初始化数组大小并分配内存。
2. **拷贝构造函数**：深拷贝另一个 `Array` 对象。
3. **赋值操作符**：深拷贝另一个 `Array` 对象，同时避免自赋值。
4. **析构函数**：释放动态分配的内存。
5. **输入操作符**：从输入流读取数组元素。
6. **输出操作符**：将数组元素输出到输出流。
7. `max` **函数**：返回数组中的最大值。
8. `sum` **函数**：返回数组元素的总和。
9. `sort` **函数**：对数组进行排序。
10. `resize` **函数**：调整数组大小并保留原有元素。

这个 `Array` 类模板实现了动态数组的基本操作，可以满足题目中的需求。

第三问

把下面的递归函数转化成功能等价的尾递归函数

```
int recsum(int n){
    if(n==1){
        return 1;
    }
    else{
        return n+recsum(n-1);
    }
}
```

将普通递归函数转换为尾递归函数的一般方法是引入一个辅助参数来累积结果，从而使得递归调用是最后一个操作。对于 `recsum` 函数，我们可以通过引入一个累积参数 `acc` 来实现这一点。

下面是将 `recsum` 转换为等价的尾递归函数：

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

// 辅助函数，实现尾递归
int recsum_tail(int n, int acc) {
    if (n == 1) {
        return acc + 1;
    } else {
        return recsum_tail(n - 1, acc + n);
    }
}

// 包装函数，提供简洁的接口
int recsum(int n) {
    return recsum_tail(n, 0);
}

int main() {
    int n = 5;
    cout << "The sum of first " << n << " natural numbers is: " << recsum(n) <<
endl;
    return 0;
}
```

解释

1. **辅助函数** `recsum_tail`：
    - 这个函数接收两个参数：当前要处理的数字 `n` 和累积和 `acc`。
    - 如果 `n` 等于 1，则返回 `acc + 1`。
    - 否则，调用 `recsum_tail(n - 1, acc + n)`，其中 `acc` 累积当前的 `n`。
2. **包装函数** `recsum`：
    - 提供一个简单的接口来调用尾递归函数。
    - 初始调用时，将 `acc` 初始化为 0。

调用示例

```
int main() {
    int n = 5;
    cout << "The sum of first " << n << " natural numbers is: " << recsum(n) <<
endl;
    return 0;
}
```

这个代码输出：

```
The sum of first 5 natural numbers is: 15
```

这个尾递归函数的优点是，它可以被编译器优化成迭代形式，从而避免了普通递归可能导致的栈溢出问题。