程设 2 小班辅导

无 12 李羿璇

目录

1 引言

1.1 程设 2 我们应当掌握什么

- C++的基本语法
  - C++的输入、输出
  - 类的声明、定义与继承，成员函数与成员变量
  - 运算符重载
  - 类的访问权限控制
  - **类对象的构造析构顺序**（重要！八股文）
  - 命名空间的基本概念
  - 模板元编程的基本概念（了解）
  - ModernC++与 STL（了解）
- 面向对象程序设计
  - **面向对象的基本概念**（重要！八股文）
  - 面向对象的思想（了解）

2 背景知识

C++ 是一门支持面向对象特性的语言，支持多种编程范式（面向对象编程、面向过程编程、泛型编程、函数式编程、……）。被成为“C++ 之父”的 Bjarne Stroustrup 于 1979 年开始制作 C with class，后更名为 C++。

自 1998 年第一个 C++标准（C++98）发布后，目前 C++每 3 年发布一次标准。下面列举了现存的 C++标准：

表 1　C++历代标准

发布时间	称呼	备注/锐评
史前时期	C++1.0、C++2.0…	标准混乱、版本繁多，程设比这个好一些
1998 年	C++98	第一个 ISO C++标准，至今依然使用
2003 年	C++03	修复 C++98 的部分 BUG，没有本质区别
2011	C++11	巨大改动，ModernC++一般是从 C++11 开始说
2014	C++14	修复 C++11 的部分设计失误
2017	C++17	较大更新（本来想搞个大的但是拉了，拖到 C++20 了）
2020	C++20	巨大改动（模块、`import` 等）
2023	C++23	修复 C++20 的部分设计失误

问：那咱们的程设是哪一个标准呢?

答：咱们的程设是微软（Microsoft）发布的 VS2008“标准”！（bushi

与之最接近的是 C++98/C++03。

3 知识点复习

3.1 C++基本语法

2022 填空第 10 题

C++中的标识符（例如变量等）的作用域有: 全局作用域、()、()、文件作用域、函数作用域以及复合语句作用域。

答案：类作用域、命名空间作用域、多文件作用域…写前两个是最保险的

备注：有同学可能没听说过甚么是复合语句作用域，举个例子：

```
for (int i = 0; i < 10; i++)
{
    std::cout << i << " ";
}
```

而与之对比，C 语言早期版本（程设版本×）是不允许这么干的，直到 C99 之后才可以. 。

3.1.1 C++的输入、输出

语法上实际上和程设 1 中 C 语言的内容没有本质不同，而在概念上而言，主要要理解的是输入输出流的统一。

- 输出：数据 -> 缓冲区 -> 设备
- 输入：设备 -> 缓冲区 -> 程序

而具体而言，主要考点包括文件输入输出、控制台输入输出、输入输出流的重载。

- 对于控制台输入输出，主要掌握 cin、cout 的基本使用，这在几乎所有题目中都会需要用到。用 iomanip 做输入输出的格式化一般不会重点考察，简单复习即可。
- 对于文件输入输出，和控制台输入输出语法上是类似的（也体现了输入输出流统一的好处）
- 输入输出流的重载，在后面运算符重载部分再具体讲
- 除此之外，还有字符串流等，程设讲过但从来没有考过，同样了解性复习。

2013 年填空第 6 题

用流对象的成员函数控制输出格式时，用于设置字段宽度的流成员函数的名称是（），与之作用相同的控制符名称是（）。

答案：width，setw

备注：死记硬背

表 13.3　输入输出流的控制符

控制符	作　用
dec	设置整数的基数为 10
hex	设置整数的基数为 16
oct	设置整数的基数为 8
setbase(n)	设置整数的基数为 n(n 只能是 8,10,16 三者之一)
setfill(c)	设置填充字符 c,c 可以是字符常量或字符变量
setprecision(n)	设置实数的精度为 n 位。在以一般十进制小数形式输出时 n 代表有效数字。在以 fixed(固定小数位数)形式和 scientific(指数)形式输出时 n 为小数位数
setw(n)	设置字段宽度为 n 位
setiosflags(ios::fixed)	设置浮点数以固定的小数位数显示
setiosflags(ios::scientific)	设置浮点数以科学记数法(即指数形式)显示
setiosflags(ios::left)	输出数据左对齐
setiosflags(ios::right)	输出数据右对齐
setiosflags(ios::skipws)	忽略前导的空格
setiosflags(ios::uppercase)	在以科学记数法输出 E 和以十六进制输出字母 X 时以大写表示
setiosflags(ios::showpos)	输出正数时给出“+”号
resetioflags( )	终止已设置的输出格式状态,在括号中应指定内容

表 13.4　用于控制输出格式的流成员函数

流成员函数	与之作用相同的控制符	作　用
precision(n)	setprecision(n)	设置实数的精度为 n 位
width(n)	setw(n)	设置字段宽度为 n 位
fill(c)	setfill(c)	设置填充字符 c
setf( )	setiosflags( )	设置输出格式状态,括号中应给出格式状态,内容与控制符 setiosflags 括号中的内容相同,如表 13.5 所示
unsetf( )	resetioflags( )	终止已设置的输出格式状态,在括号中应指定内容

表 13.5　设置格式状态的格式标志

格式标志	作　用
ios::left	输出数据在本域宽范围内向左对齐
ios::right	输出数据在本域宽范围内向右对齐
ios::internal	数值的符号位在域宽内左对齐,数值右对齐,中间由填充字符填充
ios::dec	设置整数的基数为 10
ios::oct	设置整数的基数为 8
ios::hex	设置整数的基数为 16
ios::showbase	强制输出整数的基数(八进制数以 0 打头,十六进制数以 0x 打头)
ios::showpoint	强制输出浮点数的小点和尾数 0
ios::uppercase	在以科学记数法格式 E 和以十六进制输出字母时以大写表示
ios::showpos	对正数显示“+”号
ios::scientific	浮点数以科学记数法格式输出
ios::fixed	浮点数以定点格式(小数形式)输出
ios::unitbuf	每次输出之后刷新所有的流
ios::stdio	每次输出之后清除 stdout,stderr

图 1　iomanip 控制符

2015 年填空第 5 题

在 C++ 程序中 cin 是在（）中定义的（）；cout、cerr、clog 三者的相同点是（）

答案：iostream，输入流对象，向标准设备输出

备注：题目有点问题，cerr 和 clog 其实是向标准错误设备输出，但是考试总不能写都是输出×所以就这么写吧

- cout：标准输出设备，有缓冲区
- cerr：标准错误设备，无缓冲区
- clog：标准错误设备，有缓冲区

2022 年读程序写结果第 2 题

```
#include <iostream>
#include <fstream>
using namespace std;
void main()  // 说明:  在 VS2008 编译系统上,  ios_base::与 ios::功能相同
{
    int dd[] = {66, 86, 898, 939, 2022};
    ofstream tfile("data1.dat", ios_base::binary);
    tfile.write((char*)dd, sizeof dd);
    tfile.close();
    ifstream tfile1("data1.dat", ios_base::binary);
```

```
    tfile1.seekg(sizeof(int) * 2);
    tfile1.read((char*)&dd[1], sizeof(int) * 3);
    tfile1.close();
    for (int k = 0; k < 5; k++)
        cout << dd[k] << ',';
}
```

答案：66,898,939,2022,2022,

备注：本质上和程设 1 考的文件读写没有区别，不要忘记了程设 1 中指针相关的内容。**注意不要漏写最后那个逗号！**

3.1.2 类的声明、定义与继承，成员函数与成员变量

这部分主要考点也比较基础，基本上就是按照 `main` 函数的流程走。做题时，个人建议**优先看 `main`**，不要被淹没在前面类的繁杂定义中，否则容易越做越昏头。

类的继承同样是一大重要考点。

2022 年填空第 6 题

调用复制构造函数的对象隐含复制产生情形有三种，分别是，用类的一个对象去初始化该类的另一个新定义的对象、（）以及（）。

答案：类的对象以值被传入函数，具名对象以值被函数返回

备注：具体这一部分我们后面构造析构顺序部分再说。区分初始化与否，如下面这段代码示例：

```
#include <iostream>
class MyClass
{
public:
    MyClass()
    {
        std::cout << "MyClass constructor called." << std::endl;
    }
    MyClass(const MyClass& other)
    {
        std::cout << "MyClass copy constructor called." << std::endl;
    }
    MyClass& operator=(const MyClass& other)
    {
        std::cout << "MyClass copy assignment operator called." << std::endl;
        return *this;
    }
    ~MyClass()
    {
        std::cout << "MyClass destructor called." << std::endl;
    }
};
void foo(MyClass myClass)
{
}
int main()
{
    MyClass myClass;              // constructor called
    MyClass myClass2(myClass);    // copy constructor called
    myClass2 = myClass;           // copy assignment operator called
    MyClass myClass3 = myClass2;  // copy constructor called
```

```
    foo(myClass);                // copy constructor called
    return 0;
}
```

同时还有一道例题：

2015 年填空第 6 题

有类 A，语句组 `A a(1), b; b = a;` 和语句 `A a(1), b(a);` 在生成对象 b 并用 a 赋初值时的差别是，后者（），而前者（）

答案：调用 b 的复制构造函数，用 a 复制构造 b；先调用 b 的默认构造函数，再使用 a 对 b 赋值。

2022 读程序写结果 10

```
#include <iostream>
#include <cstring>
using namespace std;
class A
{
public:
    A(const char* aa)
    {
        b = strlen(aa);
        a = new char[b + 1];
        strcpy(a, aa);
    }
    ~A()
    {
        delete[] a;
    }
    char* Geta()
    {
        return a;
    }
    int Getb()
    {
        return b;
    }

private:
    char* a;
    int b;
};
int main()
{
    char *p1 = "dongtaiqingling", *p2 = "xianshangkaoshi";
    A x(p1), y(p2);
    cout << strlen(y.Geta()) + x.Getb() << endl;
}
```

答案：30

备注：第一个字符串长度为 15，第二个字符串长度也是 15，答案显然为 15+15=30。不要被 `a = new char[b + 1];`吓到，认为 `strlen(a) = 16`（回忆 C 语言字符串中末尾的`'\0'`），其实是不对的。

此外，这里还考了一处 `new` 和 `delete` 的用法，此处做一个辨析：

- Q：`new` 与 `delete` 和 C 语言中的 `malloc` 和 `free` 有什么区别？

- A：new 失败的情况（例如系统内存占满了，下学期 OJ 会经常遇到这种情况×）会直接抛异常，而 malloc 失败的情况下会返回 NULL，因此 malloc 之后要判空，而 new 则没有必要（如果失败了的话程序就直接挂了）。同时，new 与 delete 会调用类的构造析构函数，而 malloc 与 free 不会。很多情况下，这两者其实不会导致本质性的区别，但是有一种例外：使用了虚函数重写的情况。

```
#include <iostream>
#include <cstdlib>
using namespace std;
class Base
{
public:
    virtual void print()
    {
        cout << "Base::print()" << endl;
    }
};
class Derived : public Base
{
public:
    void print() override
    {
        cout << "Derived::print()" << endl;
    }
};
int main()
{
    Base* b = new Derived();
    b->print();  // 没有问题，Derived::print()
    delete b;

    Base* b2 = (Base*)malloc(sizeof(Derived)); // 无论 Derived* 还是 Base* 都会 G
    ((Derived*)b2)->print();  // G 了！直接 Segmentation Fault!
    free(b2);
}
```

由此可见，malloc 做的事情是单纯的分配内存，**而没有正确地初始化**。而当使用虚函数的时候，虚函数表会在调用构造函数时被正确初始化，这时使用 malloc 大概率会出现问题。**类似的问题去年很多同学在程设大作业里都遇到过**，衷心希望大家在写今年大作业时**不要再用 malloc 管理内存**。

- Q：new[]、delete[] 和 new、delete 有什么区别？
- A：new[]时包含了数组长度这样一个信息，因此应当配对地使用 delete[]，从而正确地释放整个数组的内存。否则，可能会导致未定义行为，例如以下这个例子：

```
#include <iostream>
#include <cstdlib>
using namespace std;
class MyClass
{
public:
    MyClass()
    {
        y = x;
        cout << x++ << endl;
    }
    ~MyClass()
```

```
    {
        cout << "Destruct " << y << endl;
    }

    static int x;

private:
    int y;
};
int MyClass::x = 0;
int main()
{
    MyClass* classes = new MyClass[3];
    delete[] classes;
    MyClass* classes2 = new MyClass[3];
    delete classes2;
    return 0;
}
```

这个例子中，我们期望输出 012、Destruct210、345、Destruct543 或者 Destruct5/4/3 中的一个。但是实际上，这样做会直接导致代码崩掉：

```
munmap_chunk(): invalid pointer
Aborted
```

2015 年填空第 10 题

用 new 申请某一个类的动态对象数组时，在该类中必须能够匹配到（）的构造函数，否则应用程序会产生一个编译错误。

答案：没有形参/缺省参数

备注：new[]的时候不能给初值，C++11 之后可以给初值，例如下面这样，但是没什么用（×

```
int *p = new int[5]{1, 2, 3, 4, 5};
```

2022 填空第 7 题

虚函数通常在重名的成员函数前加上 () 关键词声明，此类函数的调用延迟到程序运行时再进行动态绑定,没有声明为 virtual 的成员函数则在（）时进行静态绑定。

答案：virtual，编译

备注：virtual 是 C++中实现多态的重要工具。同时，用 virtual 继承重写的话，最好在子类加上 override。否则可能会在写代码时出错。

```
#include <iostream>
using namespace std;
class Base
{
public:
    virtual void print()
    {
        cout << "Base::print()" << endl;
    }
    virtual void print2()
    {
        cout << "Base::print2()" << endl;
    }
};
```

```
class Derived : public Base
{
public:
    void print(int aa) // 编译器不能正常检查出错误
    {
        cout << "Derived::print()" << endl;
    }
    void print2(int aa) override // 编译器能正常检查出错误
    {
        cout << "Derived::print2()" << endl;
    }
};
```

2022 年填空第 8 题

仅包含纯虚函数的类称为 (), 某类中包含一个纯虚函数 `float fun()`, 则该成员函数应声明为 ()。

答案：抽象类；`virtual float fun() = 0;`

备注：类似的写法在正统的 OOP 语言（C#、Java 等）中又称为接口

2022 年读程序写结果第 1 题

```
#include <iostream>
using namespace std;
class A
{
public:
    A()
    {
        a = 5;
        b = 5;
    }
    A(int i)
    {
        a = i;
        b = 7;
    }
    A(int i, int j)
    {
        a = i;
        b = j;
    }
    void display()
    {
        cout << "a=" << a << " b=" << b;
    }

private:
    int a;
    int b;
};
class AB : public A
{
public:
    AB()
    {
        X = 2022;
    }
```

```
    AB(int i) :
        A(i)
    {
        X = 0;
    }
    AB(int i, int j) :
        A(i, j + 9)
    {
        X = 4;
    }
    AB(int i, int j, int k) :
        A(i, j + 9)
    {
        X = k;
    }
    void display1()
    {
        display();
        cout << " X=" << X << endl;
    }

private:
    int X;
};
int main()
{
    AB b1;
    AB b2(39);
    AB b3(3, 3);
    AB b4(6, 6, 6);
    b1.display1();
    b2.display1();
    b3.display1();
    b4.display1();
    return 0;
}
```

答案:

```
a=5 b=5 X=2022
a=39 b=7 X=0
a=3 b=12 X=4
a=6 b=15 X=6
```

备注：题目本身没有难度，主要是要**正确匹配具体调了哪个构造函数**。同时，**要注意父类构造函数参数的各种奇怪加减的情况**，做类似题目一定要**慢下来**仔细思考。

2022 年读程序写结果第 9 题

```
#include <iostream>
#include <cstring>
using namespace std;
class A
{
public:
    A(const char* name = "Huang")
    {
        strcpy(this->name, name);
    }
```

```
    const char* getName() const
    {
        return name;
    }
    virtual char* getAddress() const
    {
        return "THU";
    }

private:
    char name[20];
};
class AB : public A
{
public:
    AB(const char* name) :
        A(name)
    {
    }
    virtual char* getAddress() const
    {
        return "PKU";
    }
};
int main()
{
    A* gs = new AB("Yang");
    cout << gs->getName() << " STAY IN " << gs->getAddress() << endl;
    delete gs;
    gs = new A;
    cout << gs->getName() << " STAY IN " << gs->getAddress() << endl;
    delete gs;
}
```

答案:

```
Yang STAY IN PKU
Huang STAY IN THU
```

备注：题目本身无难度，但本题是 C++最常见的多态用法。同样的 A 类指针 `gs`，指向 `AB` 对象或 `A` 对象，得到结果是不一样的。

2022 年读程序写结果第 8 题

```
#include <iostream>
using namespace std;
class A
{
public:
    A(int x = 0) :
        r1(x)
    {
    }
    void print()
    {
        cout << 'E' << r1 << '-';
    }
    void print() const
    {
```

```
        cout << 'C' << r1 * r1 << '-';
    }
    void print(int x)
    {
        cout << 'P' << r1 * r1 * r1 << '-';
    }
    void print(int x) const
    {
        cout << 'T' << r1 * r1 * r1 * r1 << '-';
    }

private:
    int r1;
};
int main()
{
    A a1, a2(2);
    const A a3(3), a4(4);
    a1.print(5);
    a2.print();
    a3.print(6);
    a4.print();
    return 0;
}
```

答案：P0-E2-T81-C16-

备注：这里考察了知识点，const 的类对象会优先调用 const 的方法，而非 const 的类对象则会优先调用非 const 的方法。此外，我们一般说参数量个数、类型都相同是不能重载的，但在本题中可以看到，对于 C++而言，考虑 const 与否的情况下是可以重载的。

2022 年读程序写结果第 7 题

```
#include <iostream>
using namespace std;
class A
{
public:
    A(int xx = 0) :
        x(xx)
    {
        ++count;
    }
    virtual void show() const
    {
        cout << count << " " << x << endl;
    }
protected:
    static int count;
private:
    int x;
};
class B : public A
{
public:
    B(int xx, int yy) :
        A(xx),
        y(yy)
```

```
    {
        y += 27;
    }
    virtual void show() const
    {
        cout << count << " " << y << endl;
    }
private:
    int y;
};
int A::count = 0;
int main()
{
    A* ptr[] = {new B(30, 24), new A};
    ptr[0]->show();
    ptr[0]->A::show();
    ptr[1]->show();
    delete ptr[0];
    delete ptr[1];
}
```

答案:

```
2 51
2 30
2 0
```

备注：主要考察了 static 的用法，static 是静态的，所有对象里的 static 都是统一的。在 new B 和 new A 两次操作中各给 count+1。剩下的部分体现了 C++的多态，B->show()和 B->A::show()是有区别的。

3.1.3 运算符重载

主要分为几大考点:

- 重载输入输出流
- 重载算术运算符
- 重载类型转换运算符
- 一些存在隐式类型转换的情况

2015 年填空第 8 题

重载的运算符保持其原有的（）、优先级和结合性不变

答案：操作数个数

备注：三个都有可能挖空，不要只背一个。

2015 年读程序写结果第 4 题

```
#include <iostream>
using namespace std;
class matrix
{
public:
    matrix(double a1 = 1, double a2 = 2, double a3 = 3, double a4 = 4)
    {
        a[0][0] = a1;
        a[0][1] = a2;
```

```
        a[1][0] = a3;
        a[1][1] = a4;
    }
    matrix operator+(matrix& m);
    friend ostream& operator<<(ostream&, matrix&);
private:
    double a[2][2];
};
matrix matrix::operator+(matrix& m)
{
    int i, j;
    matrix c(0, 0, 0, 0);
    for (i = 0; i < 2; i++)
        for (j = 0; j < 2; j++)
            c.a[i][j] = a[i][j] + m.a[i][j];
    return c;
}
ostream& operator<<(ostream& out, matrix& b)
{
    int i, j;
    for (i = 0; i < 2; i++)
    {
        for (j = 0; j < 2; j++)
            out << b.a[i][j] << " ";
        out << endl;
    }
    return out;
}
int main()
{
    matrix a(1.1, 1.2, 1.3, 1.4), b(2.1, 2.2, 2.3, 2.4), c;
    c = a + b;
    cout << "矩阵 a+b 等于" << endl;
    cout << c;
    return 0;
}
```

答案:

```
矩阵 a+b 等于
3.2 3.4
3.6 3.8
```

备注：是一道比较常规的题目，ostream 的使用基本上也就只会考到这个程度，同时本题中涉及的算术运算符重载也比较简单。但是值得注意的是，本题传参用的是**引用 `matrix&`**，但是很多同学对这个概念不是很理解。事实上，程设这种写法并不是很好。例如，下面这段代码：

```
matrix m1;
matrix m2 = m1 + matrix(1, 2, 3, 4);
```

这段代码是不能通过编译的！这涉及到了左值、左值引用右值等各种复杂概念，**此处不过多深究**，感兴趣的同学可以查看 \xfgg/关于 const 的讲义、\xfgg/关于左右值的讲义深入了解。

简单地说，`matrix(1, 2, 3, 4)`是一个“匿名对象”，是一个所谓的“右值”，是不能被绑定到“左值引用”`matrix&` 上的，原因可以理解为我们**不可能对一个匿名的、临时的 `matrix(1, 2, 3, 4)`里的数值做任何修改**，然而使用 `matrix&` 却默认可以做这样的修改。两者发生冲突，进而编译报错。加上 `const` 之后，则避免了这个问题。大家在写程设大作业时，**同样一定一定要注意避免这个问题！**

2022 年填空第 3 题

C++中认为（）是给对象或变量取一个别名。使用这种机制和（）一样，可以在函数间双向传递参数的变量值。

答案：引用，指针

备注：八股文（其实这题放这里不是很合适×）

2022 年补充程序第 2 题

```
#include <iostream>
using namespace std;
class CInstan
{
public:
    CInstan(int, int);
    CInstan operator++();
    CInstan operator++(int);
    CInstan& operator=(const CInstan&);
    void Display() const;
private:
    int t1, t2;
};
// 类成员函数的实现
CInstan::CInstan(int a, int b)
{
    t1 = a;
    t2 = b;
}
CInstan CInstan::operator++()
{
    cout << "++test" << endl;
    t1++;
    t2++;
    ____________________________;
}
CInstan& CInstan::operator=(const CInstan& c)
{
    if (this == &c)
        return *this;
    t1 = c.t1;
    t2 = c.t2;
    ____________________________;
}
CInstan CInstan::operator++(int)
{
    cout << "test++" << endl;
    int tmp1 = t1, tmp2 = t2;
    t1++;
    t2++;
    ____________________________;
}
void CInstan::Display() _______
{
    cout << "t1 = " << t1 << ", t2 = " << t2 << endl;
}
int main()
```

```
{
    CInstan b(3, 4), a(0, 0);
    b.Display();
    cout << endl;
    ++b;
    b.Display();
    cout << endl;
    a = b;
    return 0;
}
```

答案：

```
return *this;
return *this;
return CInstan(tmp1, tmp2);
const
```

解析：注意区分两种自增的区别，前置作为一元运算符（`operator++()`），后置则作为二元运算符（`operator++(int)`）。同时，原则上来说，前置的需要返回引用（要求返回的是一个左值），后置的则需要返回值（虽然这题没管×），这个主要是为了照顾如下写法：

```
int a = 5;
(a++) = 114514; // 错误，编译直接报错
(++a) = 114514; // 正确
```

这个设计有一说一比较愚蠢，大家看看有个大概印象就好）

2022 年读程序写结果第 5 题

```
#include <iostream>
using namespace std;
class Complex
{
public:
    Complex()
    {
        real = 0;
        imag = 0;
    }
    Complex(double r)
    {
        real = r;
        imag = 0;
    }
    Complex(double r, double i)
    {
        real = r;
        imag = i;
    }
    operator double()
    {
        return real;
    }  // 强制转换 Complex 对象为其实部 real 并返回
    void display();

private:
    double real;
    double imag;
```

```
};
void Complex::display()
{
    cout << "(" << real << ", " << imag << ")" << endl;
}
int main()
{
    Complex R1(5, 6.3), RR;
    int D;
    double E = 1.9;
    D = 15 + R1;
    cout << "D=" << D << endl;
    RR = Complex(D);
    RR = E + 5 + RR;
    cout << "RR=";
    RR.display();
    return 0;
}
```

答案:

```
D=20
RR=(26.9, 0)
```

备注: 运算符重载题目的集大成者，写出正确结果并不难，理解最重要。大概而言，按照 `main` 的顺序，这个题干了这么几件事:

- `Complex R1(5, 6.3), RR;`，此时 `R1=5+6.3i`，`RR=0+0i`，分别调用了两个构造函数
- `D = 15 + R1;`，这里首先把 `R1` 转成了 `double`（`Complex` 没有和 `double` 做加法的重载，因此只能先转成 `double`），得到 5，再加上 15，得到 20，赋值给 `D`，`D=20`
- `cout << "D=" << D << endl;`，`D` 是个 `int`，直接输出 `D` 的值
- `RR = Complex(D);`先把 `D` 转成 `double`，再调用 `Complex` 的构造函数，最后再赋值，得到 `RR=20+0i`
- `RR = E + 5 + RR;`，和上述所有操作类似，`RR->double`，然后相加得到 `20+5+1.9=26.9`，然后再调用一个参数的构造函数，因此 `RR=26.9+0i`

上面这道题里存在大量的隐式类型转换，但这样做实际上是**很危险的**，例如下面这个行为:

```
cout << cin << endl;
```

这个操作一看起来就很奇怪！在 C++11 之后的版本这个操作通过不了编译，也是我们一般希望看到的。但是在 C++98 下，输出是这样的:

```
0x56490636e170
```

为何会这样？起因是希望我们希望 `iostream` 有一种渠道，来确认能否进行输入/输出。一个直接的想法是重载一个 `operator bool`，例如类似下面这样的实现:

```
struct OStream
{
  operator bool() { return true; }
  OStream& operator<<(int x) { cout << x; return *this; }
};
OStream out;
int main()
{
  if (out) { out << 111; }
}
```

看起来非常不错。但是如果我们写错了呢？例如，

```
out >> 111;
```

这个竟然也能过编译！这就非常奇怪了。实际上这个操作做了两件事情：

- 首先把 `out` 转为 `bool`（true）
- 然后再把 `bool` 转为 `int`（1）
- 最后执行 `1 >> 111` 这样一个右移的操作

这个显然很愚蠢，应该直接报错才对。因此，实际 C++98 标准库中不是这么设计的，一般是重载 `operator void*`，当不能输出时返回空指针，否则是返回正常的自身指针地址。

```
struct OStream
{
  operator void*() { return this; }
  OStream& operator<<(int x) { cout << x; return *this; }
};
OStream out;
int main()
{
  if (out) { out << 111; }
}
```

看起来非常好！这样就不可能调用 `out >> 111` 这样的操作了（`void*` 不能隐式转化到整形，也就不能这样右移）。但是这样就构成了一些不是很好的后果，例如我们上面举的例子。

那么为什么 C++11 之后就修好了这个问题呢？原因是 C++11 引入了 `explicit` 关键字，要求只能显式调用类型转换，特别的，对于 `explicit operator bool()`，规定在 `if`、`while` 等操作中也可以隐式转换为 `bool`，解决了以上的所有问题。

```
struct OStream
{
    explicit operator bool()
    {
        return true;
    }
    OStream& operator<<(int x)
    {
        cout << x;
        return *this;
    }
};
OStream out;
int main()
{
    // out >> 1; Compile Error!
    if (out)
    {
        out << 2;
    }
}
```

3.1.4 类的访问权限控制

C++主要提供了三种访问权限控制：

- `public`：公共访问权限，可以被任何函数访问

- `private`：私有访问权限，只能被本类的成员函数访问
- `protected`：受保护访问权限，只能被本类和派生类的成员函数访问

`public` 继承后，派生类里的所有权限不变；`private` 继承后，派生类的全部权限都变成 `private`；`protected` 继承后，派生类里的 `public` 权限变成 `protected`。

此外，`class` 的默认访问权限是 `private`，而 `struct` 的默认访问权限是 `public`，除此之外两者**没有任何区别**。

2022 年填空第 1 题

C++ 类的派生类对基类的成员有 3 种，分别是公有、（）和（）。

答案：保护、私有

2022 补充程序 1

```
#include <iostream>
using namespace std;
class TS
{
public:
    TS(int n = 0, int d = 0);
    void Print();
    ____________________

private:
    int N, D;
};
TS Div(TS& r1, TS& r2)
{
    return TS(r1.N / r2.D + r1.D / r2.N, r1.D / r2.D);
}
_____ TS(int n, int d)
{
    N = n;
    D = d;
}
void TS::Print()
{
    cout << N << "/" << D << endl;
}
int main()
{
    TS a(1, 2), b(3, 4), c;
    c = Div(a, b);
    c.Print();
}
```

答案：`friend TS Div(TS& r1, TS& r2); TS::`

备注：通过 `friend` 来使外部函数（`TS Div`）访问内部的私有成员。

3.1.5 类对象的构造析构顺序

这一部分是程设最重要的考点之一，主要是考察一段程序里构造了多少个对象，以及它们的构造顺序。具体细分的话又有两类，一类是**调用函数时的构造析构顺序**（重点考察对象），还有一类是继承、成员的构造顺序（不作为重点考察）。

一般来讲，学完程设后，我们会产生三大核心想法：

- 先构造的后析构
- 根据参数数量匹配对应的构造函数
- 传值/返回值则复制

但是实际情况远没有那么简单，类似的题目答案都存在争议，在不同的编译器下可能会导致不同的结果。为了确保知识覆盖的完整性，我们改一道题，完整考察构造析构顺序。

2022 年读程序写结果第 3 题（改编）

```
#include <iostream>
using namespace std;
class complex
{        // 复数类声明
public:  // 外部接口
    complex(double r = 0, double i = 0)
    {
        this->r = r;
        this->i = i;
        cout << "Constructor! ";
        display();
    }
    complex(complex& c) :
        r(c.r),
        i(c.i)
    {
        cout << "Copy Constructor! ";
        display();
    }
    ~complex()
    {
        cout << "Destructor! ";
        display();
    }
    complex operator-(complex c2);
    complex operator+(const complex& c2);
    void display();  // 输出复数

private:
    double r;  // 复数实部
    double i;  // 复数虚部
};
complex complex::operator-(complex c2)
{
    return complex(r - c2.r, i - c2.i);
}
complex complex::operator+(const complex& c2)
{
    complex c;
    c.r = r + c2.r;
    c.i = i + c2.i;
    return c;
}
void complex::display()
{
    cout << "(" << r << "," << i << ")" << endl;
```

```
}
int main()  // 主函数
{
    complex c1(3, 4), c2(8, 9), c3;
    cout << "c1=";
    c1.display();
    cout << "c2=";
    c2.display();
    c3 = c1 - c2;
    cout << "c3=c1-c2=";
    c3.display();
    c3 = c1 + c2;
    cout << "c3=c1+c2=";
    c3.display();
}
```

答案：

```
Constructor! (3,4)
Constructor! (8,9)
Constructor! (0,0)
c1=(3,4)
c2=(8,9)
Copy Constructor! (8,9)
Constructor! (-5,-5)
Destructor! (8,9)
Destructor! (-5,-5)
c3=c1-c2=(-5,-5)
Constructor! (0,0)
Copy Constructor! (11,13)
Destructor! (11,13)
Destructor! (11,13)
c3=c1+c2=(11,13)
Destructor! (11,13)
Destructor! (8,9)
Destructor! (3,4)
```

备注：这道题有如下几个坑点：

- 我们一般说“先构造后析构”，一般情况下的确如此，但是在函数的复制中例外
  - 先构造复制过去的传参，再构造临时变量（如果有），最后析构返回值
  - 先析构参数，再析构临时变量，最后构造返回值
- 到底要不要复制？
  - 传参的情况下，如果是传值则复制，如果是传引用/常引用则不复制
  - 返回值的情况下，情况比较特殊。如果是返回一个具名对象（上面的加法例子），则要复制；如果是返回一个临时的匿名对象（上面的减法例子），则不需要复制
- **上述说法均只在【程设的】特定【VS】版本中生效，学这个只为了期末考试拿分，没有任何本质的意义。大家背下来上面这些原则即可，考完试就可以扔了（bushi）**

如果对这部分有更深兴趣、想了解背后原理的同学，可以去参考\xfgg/返回值优化讲义

2022 年读程序写结果第 6 题

```
#include <iostream>
using namespace std;
class B
{
```

```
public:
    B(int x = 0) :
        ValB(x)
    {
        cout << ValB;
    }
    ~B()
    {
        cout << ValB;
    }
private:
    int ValB;
};
class D : public B
{
public:
    D(int x = 0, int y = 0) :
        B(x),
        c(x),
        ValD(y)
    {
        cout << ValD;
    }
    ~D()
    {
        cout << ValD;
    }
private:
    B c;
    int ValD;
};
int main()
{
    D Obj1(5, 6), Obj2(7, 8);
    return 0;
}
```

答案：556778877655

备注：继承、成员这类题目的考法大概就是这样，可以看到显然没什么难度，记住顺序即可：

- 先构造父类（非虚继承）
- 再构造父类（虚继承）
- 再构造成员变量
- 最后构造自己
- 析构顺序与构造顺序相反

3.1.6 命名空间的基本概念

程设中所学的命名空间非常浅，此处只涉及一道简单例题。

2015 年读程序写结果第 1 题

```
#include <iostream>
using namespace std;
int Num = 1;
namespace Name1
{
```

```
    int Num = 2;
    int Add(int Num)
    {
        Num = ::Num + Num;
        return Num;
    }
}
namespace Name2
{
    int Num = 3;
    int Add(int Num)
    {
        Num = ::Num + Num;
        return Num;
    }
}
int main()
{
    cout << Name1::Add(4) + ::Num << endl;
    cout << Name2::Add(5) + Num << endl;
    namespace N2 = Name1;
    cout << N2::Add(6) + N2::Num << endl;
    using namespace Name2;
    cout << Add(7) + Name2::Num << endl;
}
```

答案:

```
6
7
9
11
```

备注: 体现了命名空间的作用，可以防止这些 Num 冲突。记住单写一个 Num 是优先调用自身所在命名空间的 Num，而::Num 则是全局的 Num 即可。

- Name1::Add(4): Num 是传进来的形参 4、::Num 是全局变量 Num=1，故这里得到 5，再加上::Num=1 得到 6
- Name2::Add(5): Num 是传进来的形参 5、::Num 是全局变量 Num=1，故这里得到 6，再加上::Num=1 得到 7
- namespace N2=Name1: 为 Name1 起一个别名 N2，此后 N2 和 Name1 就没有任何区别了
- N2::Add(6): Num 是传进来的形参 6、::Num 是全局变量 Num=1，故这里得到 7，再加上 N2::Num=2 得到 9
- using namespace Name2: 将 Name2 命名空间的全部内容导入到当前命名空间，可以省略 Name2:: 前缀
- Add(7): Num 是传进来的形参 7、::Num 是全局变量 Num=1，故这里得到 8，再加上 Name2::Num=3 得到 11

3.2 面向对象程序设计

3.2.1 面向对象的基本概念

对于程设考试而言，这部分就是纯纯八股文，背住课件即可。

2013 年填空第 1 题

面向对象程序设计有四个主要特点，即抽象、封装、() 和 ()。

答案：继承、多态

备注：如果是三个特点，就不写抽象。

2022 年填空第 2 题

C++中类是对象的 ()，而 () 是类的抽象。

答案：抽象；类模板

备注：反过来说的话就是对象是类的实例。

2015 年填空第 1 题

C++ 引入了函数重载和虚函数。其用途分别用一句话概括：函数重载是 ()；虚函数是 ()

答案：通过静态束定，在编译时确定调用同名函数中的哪一个；通过动态束定，在程序运行时确定调用同名函数中的哪一个。

备注：死记硬背，理解记忆

4 写在最后

4.1 C++究竟有什么用

可能很多同学在程设、数算等课程结束后不会再写 C++了，但是还是希望大家能在程设课之后，学会基本的程序设计思想，有所收获。同时，也应当大致记住 C++的一些深入操作系统的特性，在未来需要时使用。

在生产环境下，C++实际上还是有不少应用的。例如高性能计算的情景下，C++是不可或缺的。下面是我本学期选修的《高性能计算导论》课程的一个大作业片段，可以看到全是 C++：

```
__global__ void ThirdStageBatch(int n, int p, int *graph)
{
          __shared__     int     horizontalblocks[BATCH_SIZE_STAGE3][BLOCK_SIZE][BLOCK_SIZE],
verticalblocks[BATCH_SIZE_STAGE3][BLOCK_SIZE][BLOCK_SIZE];
  int dist = 1000000;
  int centerPos = p * BLOCK_SIZE;
  for (int i = 0; i < BATCH_SIZE_STAGE3; i++)
          horizontalblocks[i][threadIdx.y][threadIdx.x]   =   (threadIdx.y   +   (blockIdx.x   *
BATCH_SIZE_STAGE3 + i) * BLOCK_SIZE < n && centerPos + threadIdx.x < n) ? graph[(threadIdx.y +
(blockIdx.x * BATCH_SIZE_STAGE3 + i) * BLOCK_SIZE) * n + centerPos + threadIdx.x] : 1000000;
  for (int i = 0; i < BATCH_SIZE_STAGE3; i++)
    verticalblocks[i][threadIdx.y][threadIdx.x] = (centerPos + threadIdx.y < n && threadIdx.x
+ (blockIdx.y * BATCH_SIZE_STAGE3 + i) * BLOCK_SIZE < n) ? graph[(centerPos + threadIdx.y) * n
+ threadIdx.x + (blockIdx.y * BATCH_SIZE_STAGE3 + i) * BLOCK_SIZE] : 1000000;
  // divide into 2 loops for locality
  __syncthreads();
  int posX, posY;
  for (int i = 0; i < BATCH_SIZE_STAGE3; i++)
  {
    for (int j = 0; j < BATCH_SIZE_STAGE3; j++)
    {
      posX = threadIdx.x + (blockIdx.y * BATCH_SIZE_STAGE3 + j) * BLOCK_SIZE;
      posY = threadIdx.y + (blockIdx.x * BATCH_SIZE_STAGE3 + i) * BLOCK_SIZE;
      if (posX < n && posY < n)
      {
```

```
        dist = graph[posY * n + posX];
        for (int k = 0; k < BLOCK_SIZE; k++)
        {
                dist = min(dist, horizontalblocks[i][threadIdx.y][k] + verticalblocks[j][k]
[threadIdx.x]);
        }
        graph[posY * n + posX] = dist;
      }
    }
  }
}
```

4.2 程设之外

程设知识只是入门，还有很多内容完全没有涉及，是难以让大家产生更好理解的。下面讲一些可能有用的东西的引子：

4.2.1 设计模式

面向对象程序设计原则（注意此处说的是面向对象，而非指 C++ 面向对象的语言特性）：

- 单一原则
- 开闭原则
- 里氏替换原则（LSP）
- 接口隔离原则
- 依赖反转原则
- ……

下面着重说一下里氏替换原则。

里氏替换原则：派生类可以替换掉基类，派生类在语义上“完全是”一个基类。

程设课上的一些反例：

`Square` 继承 `Rectangle`。

```
class Rectangle {
public:
    Rectangle(int width, int height) : width(width), height(height) {}
    int getWidth() const { return width; }
    int getHeight() const { return height; }
    void setWidth(int newWidth) { width = newWidth; }
    void setHeight(int newHeight) { height = newHeight; }
private:
    int width;
    int height;
};
class Square : public Rectangle {
public:
    Square(int edge) : Rectangle(edge, edge) {}
    int getEdge() const { return Rectangle::getWidth(); }
    void setEdge(int newEdge) {
        Rectangle::setWidth(newEdge);
        Rectangle::setHeight(newEdge);
    }
};
```

确实我们在日常生活中都说“正方形是一个长方形”，但是在本代码中并不是这样——因为我们对 `Square` 暴露了公有接口 `setWidth` 和 `setHeight`。如果我们执行了：`Square sq(10); sq.setWidth(88);`，那么就出现了问题——这个正方形长等于宽的性质被打破。也就是说，正方形并不能完全替代长方形，并不满足长方形所能进行的全部操作！所以这种设计思路是不符合里氏替换原则的。

更多的关于设计模式内容，可以参考科协网站。

4.2.2 STL

STL 提供了很多好用的容器与接口，在后续的《数据与算法》课程 OJ 中极其有用。例如，动态数组 `std::vector`、队列 `std::queue`、`std::map`、压位神器 `std::bitset`、自动计算哈希值的 `std::hash` 等等，同时还有 `std::sort`,`std::find` 等等各种好用的算法。

关于 STL 不可能在这里大书特书，但是**强烈建议大家在小学期的大作业里多用 STL**，为明年写 OJ 做好充足的准备。可以通过科协网站查阅相关资料。

4.2.3 Modern C++

Modern C++实际上与我们所学的程设已经完全是两种语言了。以最新的 C++23 为例：

```
import std;

auto main() -> int {
    std::println("Hello, world!");
}
```

可以看出与我们之前所学的 C++大相径庭。

在改变语法，减少了过去的各种历史遗留问题的同时，Modern C++还增加了智能指针、类型推导（`auto`）、多线程支持等更多重要功能。有兴趣的同学可以参考科协网站来获取更多信息。

5 反馈

求一个好评

图 2　反馈二维码