错题（期末考复习）

类和对象

内存分配

运行时系统为每个对象的数据成员分配内存

第20题（类与对象的内存分配）

题目： 下列关于类和对象的说法中，正确的是（）。
选项：
A. 运行时系统为每个对象的数据成员分配内存
B. 类的对象具有成员函数的副本
C. 类的成员函数由类来调用
D. 编译器为每个类和类的对象分配内存

正确答案：A
解析：

1. **选项A** 正确：
   - 对象实例化时，系统为其数据成员分配独立内存（不同对象的数据成员互不干扰）。
2. 其他选项错误：
   - B：成员函数代码在内存中只有一份，所有对象共享。
   - C：成员函数通过对象调用（如 `obj.func()` ），而非类名直接调用（静态函数除外）。
   - D：编译器不为类分配内存，仅为实例化的对象分配内存。

关键点：

- 类的成员函数存储在代码区，数据成员存储在对象实例的内存中。

构造函数 **没有返回值**（连 `void` 也不行），但可以有参数。构造函数的作用是初始化对象，无返回值，名称与类名相同。

友元函数 是独立函数，通过 `friend` 声明可访问类私有成员，但 **不属于成员函数。**

静态函数无 `this` 指针。

友元函数独立于类存在，无 `this` 指针，但能突破访问权限限制。

构造函数调用次数

第16题（构造函数调用次数）

题目： `AB` 是一个类，执行语句 `AB a(4), b[3], *p;` 调用了（）次构造函数。

选项：

A. 3

B. 4

C. 5

D. 6

正确答案：B

解析：

1. ** `AB a(4)` ：**调用** 1次** 带参数的构造函数。
2. ** `AB b[3]` ：**调用** 3次** 默认构造函数（数组每个元素初始化一次）。
3. ** `AB *p` ：**声明指针** 不调用** 构造函数（未实际创建对象）。
4. **总计**：1（ `a` ）+ 3（ `b[3]` ）= **4次。**

关键点：

- 指针声明（ `*p` ）不触发构造函数，只有对象实例化时才会调用。

常函数

第二题（常成员函数定义）

题目： 若有以下类 `W` 说明，则函数 `fConst` 的正确定义是（）。

类定义：

```cpp
class W {
    int a;
public:
    void fConst(int&) const;  // 常成员函数声明
};
```

选项：

A. `void W::fConst(int& k) const { k = a; }`

B. `void W::fConst(int& k) const { cin >> a; }`

C. `void W::fConst(int& k) const { a = k; }`

D. `void W::fConst(int& k) const { k = a++; }`

正确答案：A

解析：

1. **常成员函数（ `const` ）不能修改类的非静态成员变量（ `a` ），但可以：**
   - 读取成员变量（如 `k = a` ）。
   - 修改传入的引用参数（如 `k = a` ）。

静态成员属于类，必须通过类名和作用域运算符 `::` 访问（`Y::n`）。

Y::n = 1;

运算符函数是友元和成员函数

运算符函数是一种特殊的（ 成员 ）或友元函数

隐式/显式转换

具有一个非默认参数的构造函数一般用于实现从（ ）的转换

具有一个非默认参数的构造函数一般用于实现从**该参数类型**到**当前类类型**的转换（即**隐式类型转换**或**用户定义的类型转换**）。

cpp 复制

```cpp
class MyString {
public:
    MyString(int size) {  // 非默认构造函数，接受一个 int 参数
        data = new char[size];
        length = size;
    }
    // ... 其他成员函数 ...
private:
    char* data;
    int length;
};
```

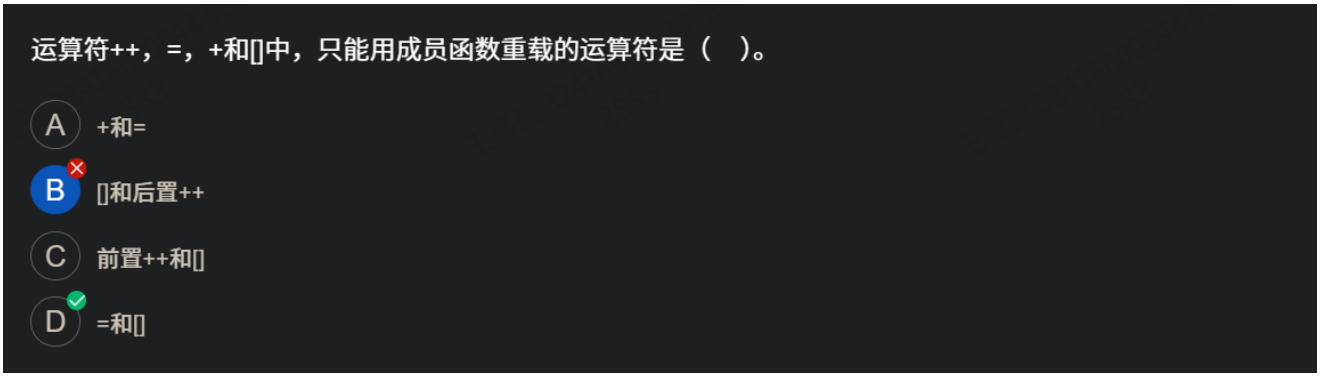
不要返回临时变量

```
Matrix operator*(int n) const

{

  Matrix result(rows);

  for (int i = 0; i < rows; i++)

  {

    for (int j = 0; j < cols; j++)

    {

      result.data[i][j] = data[i][j] * n;

    }

  }

  return result;
```

```
  }
注意不是 Matrix & operator()
friend 也没有
除了几个特殊的都没有
```

重载运算符

什么可以重载运算符



在C++ 中，运算符重载分为成员函数重载和友元函数重载。不同的运算符在选择重载方式上有一定限制：

- 赋值运算符 `=`、下标运算符 `[]`、函数调用运算符 `()`、成员访问运算符 `->` 等通常只能作为成员函数重载。

- 双目运算符（如 `+`）既可以作为成员函数重载，也可以作为友元函数重载。

- 前置单目运算符（如前置 `++`）既可以作为成员函数重载，也可以作为友元函数重载。
- 后置单目运算符（如后置 `++`）只能作为成员函数重载。
- **构造函数**不能重载运算符，因为其作用是初始化对象，而非运算符行为。

- **构造函数的作用**：将参数类型隐式转换为该类类型。

例如：`MyClass(int x)` 允许 `int` 隐式转换为 `MyClass`。

- 其他选项均可重载运算符：
- **B**（友元函数）：如 `friend MyClass operator+(...)`。

- **C**（普通函数）：全局函数可重载运算符（如 `operator<<(ostream&, ...)`）。
- **D**（成员函数）：如 `MyClass operator+(...)`。

题目：如果希望运算符的操作数（尤其是第一个操作数）有隐式转换，则重载运算符时必须用（）。
正确答案：**A（友元函数）**
解析：

- **成员函数重载**时，第一个操作数（左操作数）必须是当前类的对象，**无法隐式转换**。
- **友元函数重载**时，两个操作数均可参与隐式转换（如 `operator+(int, MyClass)` 允许 `int` 转 `MyClass`）。
- **选项B/D**（析构函数、构造函数）与运算符重载无关。

- **不可重载的运算符**：`sizeof`、`::`（作用域解析）、`?:`（三元条件）、`.`（成员访问）、`.*`（成员指针访问）。
- **可重载的运算符**：
  - **A/B**（`new`/`delete`）：可重载内存管理行为。
  - **D**（`!`）：可重载为逻辑非。

重载函数

类型转换函数只能定义为一个类的（）。正确答案：A（成员函数）

左右元

题目：当一元运算符的操作数，或者二元运算符的左操作数是该类的一个对象时，重载运算符函数一般定义为（）。正确答案：D（成员函数）

定义

无论是**一元运算符**还是**二元运算符**，都可以在**类外定义**（通过友元函数或普通函数），但需满足以下条件：

1. **二元运算符**：
   - 如果左操作数**不是**当前类的对象（如 `5 + obj`），则**必须**在类外定义（通常为友元函数）。
   - 示例：

```cpp
friend Complex operator+(int a, const Complex& obj); // 类外定义
```

2. **一元运算符**：
   - 也可在类外定义，但需通过**友元函数**（因为一元运算符的操作数通常是类对象）。
   - 示例：

```cpp
friend Complex operator-(const Complex& obj); // 类外定义一元负号
```

记忆：
左翼，相信二元论，是异教徒，必须类外定义
一元（一神论）无所谓，是朋友

编译器怎么解释前置--

关键点总结

特性	前置 --	后置 --
重载形式	成员函数或友元函数 `operator--(A& obj)`	**必须**是友元函数 `operator--(A& obj, int)`
编译器解释	`operator--(Aobject)`	`operator--(Aobject, 0)`
返回值语义	返回自减后的对象（引用）	返回自减前的对象（副本）

因此，`Aobject--` 会被编译器解释为调用友元函数 `operator--(Aobject, 0)`。

虚基类

怎么理解虚基类和二义性

1. 先看问题：多重继承的“菱形问题”

想象你要设计一个游戏角色系统：

- **战士（Warrior）** 和 **法师（Mage）** 都继承自 **人类（Human）**（因为战士和法师都是人）。
- 现在想设计一个 **圣骑士（Paladin）**，它既是战士又是法师（即同时继承 Warrior 和 Mage）。

这时就会出现 **“菱形继承”**：

复制

```
      Human
     /     \
Warrior    Mage
     \     /
     Paladin
```

问题来了：

Paladin 会 **间接继承两次 Human**（通过 Warrior 和 Mage），导致：

- Paladin 内部会有 **两份 Human 的成员**（比如 name、age 等），浪费内存！
- 如果访问 Human 的成员（比如 name），编译器不知道该用哪一份（二义性）！

特点

3. 虚基类的核心特点

特点	说明
解决菱形继承问题	确保最终派生类（如 Paladin）只继承一份虚基类（如 Human）的成员。
共享基类成员	所有派生类（Warrior、Mage）共享同一份虚基类的成员。
构造函数特殊规则	虚基类的构造函数由 **最终派生类（如 Paladin）直接调用**，而不是中间类（Warrior/Mage）。

虚函数，多态

顺序！

在创建派生类对象时，类层次中构造函数的执行顺序是由（　）。 系统规定的

7. 当一个派生类私有继承一个基类时，基类中的所有公有成员和保护成员成为派生类的（b　　）。

（A）public 成员　　（B）private 成员　　（C）protected 成员　　（D）友元

8. 下列关于类层次中重名成员的描述，错误的是（　　c）。

（A）C++允许派生类的成员与基类成员重名

（B）在派生类中访问重名成员时，屏蔽基类的同名成员

（C）在派生类中不能访问基类的同名成员

（D）如果要在派生类中访问基类的同名成员，可以显式地使用作用域符指定

9. 在创建派生类对象时，构造函数的执行顺序是（D　　）。

（A）对象成员构造函数—基类构造函数—派生类本身的构造函数

（B）派生类本身的构造函数—基类构造函数—对象成员构造函数

（C）基类构造函数—派生类本身的构造函数—对象成员构造函数

（D）基类构造函数—对象成员构造函数—派生类本身的构造函数

在具有继承关系的类层次体系中，析构函数执行的顺序是（　　）。

派生类本身的析构函数—对象成员析构函数—基类析构函数

指针到底怎么用

- **基类指针（`Base*`）**：存储基类对象或派生类对象的内存地址（需满足类型兼容）。
- **派生类指针（`Derived*`）**：只能存储派生类对象的内存地址（直接指向基类对象会导致“部分内存”问题）。

1. **基类引用/指针指向派生类对象**：　- 如果基类方法不是虚函数 → 调用基类版本。　- 如果基类方法是虚函数 → 调用派生类版本。
2. **派生类指针/引用**：
   - 直接调用派生类版本（如果重写了基类方法）。
3. **成员访问**：
   - 基类指针/引用只能访问基类成员（除非是虚函数）。
   - 派生类指针/引用可以访问派生类特有成员。

当基类指针指向派生类对象时，只能调用基类自己定义的成员函数

当派生类指针指向基类对象时( ) A 必须强制将派生类指针转换成基类指针才能调用基类的成员函数

```
#include <iostream>
using namespace std;
class Bclass
{  public:
        Bclass( int i, int j ) { x = i; y = j; }
        int fun() { return 0; }
   protected:          int x, y;
};
class Iclass: public Bclass
{      public :
         Iclass(int i, int j, int k):Bclass(i, j) { z = k; }
         int fun() { return ( x + y + z ) / 3; }
       private :   int z;
};
int main()
{      Iclass obj( 2, 4, 10 );
       Bclass & p2 = obj;
       cout << p2.fun() << endl;
       cout << obj.fun() << endl;
       Iclass *p3 = &obj;
       cout << p3-> fun() << endl;
} 类似这种题我不会做
```

```
0
5
5
```

基类析构函数必须为虚函数

若基类指针指向派生类对象，用 `delete` 释放时：

- 若基类析构函数**非虚**，仅调用基类析构函数，派生类析构函数不会被调用，导致资源泄漏（如派生类动态分配的内存未释放）。 - 若基类析构函数**为虚**，会先调用派生类析构函数，再调用基类析构函数（保证资源完全释放）。
-

错误示例（非虚析构函数）：

cpp 复制

```cpp
class Base {
public:
    ~Base() { std::cout << "Base::~Base()" << std::endl; } // 非虚析构函数
};

class Derived : public Base {
public:
    ~Derived() { std::cout << "Derived::~Derived()" << std::endl; } // 派生类析构函数
};

int main() {
    Base* ptr = new Derived();
    delete ptr; // 仅调用 Base::~Base()，Derived::~Derived() 不会被调用！
    return 0;
}
```

正确做法：

基类析构函数声明为 `virtual`：

cpp 复制

```cpp
class Base {
public:
    virtual ~Base() { ... } // 虚析构函数
};
```

重名?

当基类指针指向派生类对象时,利用基类指针调用派生类中与基类同名但被派生类重写后的成员函数时,调用的是(　　)

A
先调用基类的,再调用派生类的

B
基类的成员函数

C
派生类的成员函数

D
不确定

在C++中，当基类指针指向派生类对象时，利用基类指针调用派生类中与基类同名但被派生类重写后的成员函数时，**调用的是哪个版本的函数取决于该成员函数是否被声明为虚函数**。具体分析如下：

关键点：是否使用虚函数（virtual）

1. **如果成员函数是虚函数（virtual）：**
   - **调用派生类的版本**（动态绑定，运行时多态）。
   - 基类指针会根据实际指向的对象类型（派生类）决定调用哪个函数。
2. **如果成员函数不是虚函数（没有virtual）：**
   - **调用基类的版本**（静态绑定，编译时决定）。
   - 基类指针的类型决定了调用的函数，与实际对象类型无关。

D

虚析构函数?

A. 对虚析构函数的调用可以采用动态联编

- 正确。当基类析构函数声明为虚函数时，通过基类指针删除派生类对象时，会调用派生类的析构函数，这是多态（动态联编）的体现。

B. 构造函数可以声明为虚函数

- **错误**。这是本题的正确答案。构造函数不能声明为虚函数，原因包括：
  1. 对象构造时，虚函数表(vtable)尚未建立，无法实现多态
  2. 构造函数的作用是创建对象，此时对象类型已经确定，不需要多态

C. 当基类的析构函数是虚函数时,其派生类的析构函数也一定是虚函数

- 正确。虚函数特性会被继承，派生类析构函数自动成为虚函数(即使不显式使用virtual关键字)

D. 析构函数调用采用静态联编

- 错误表述。实际上：
  - 当基类析构函数是虚函数时，通过基类指针删除派生类对象会调用派生类析构函数(动态联编)
  - 只有非虚析构函数才会采用静态联编
- 因此这个选项本身表述不严谨，但相比B选项，B的错误更明确

最明显的错误是B选项，因为：

1. C++标准明确规定构造函数不能是虚函数
2. 其他选项要么正确(A、C)，要么表述不够严谨但不是完全错误(D)

MFC

15. 应用程序类 CWinApp 完成以下（ A ）工作。

（A）程序的初始化、运行和结束工作　　（B）消息的发送和执行

（C）程序的框架和窗口　　（D）事件的发生与处理

填空题

二、填空题（本题共16分，每小题各2分）

1. 设 op 表示要重载的运算符，那么重载运算符的函数名是（operator op ）。
2. 在 Windows 应用程序设计中，WM_LBUTTONUP 表示(释放鼠标左键)消息。
3. 在文档（CDocument）/视图(CView)程序结构中，当视图窗口需要重画时，视图类函数（ OnDraw 函数 ）被调用。
4. MFC 应用程序设计中，CEdit 是（编辑 ）控件
5. 要进行文件的输出，除了包含头文件 iostream 外，还要包含头文件（fstream 或 ofstream ）。
6. 在文件操作中，表示以追加方式打开文件的模式是（ ios::app ）。
7. 类的非静态成员函数调用都会传递一个指向对象的指针，这就是称为“隐藏”参数的（this 指针 ）指针。
8. 为解决多继承中因公共基类而产生的二义性问题，C++语言提供了（虚继承 ）机制

三.1 a:3 b:8 a:5 b:13

a=3 b=8 a=5 b=13

2. v1=1,2 v2=3,4 v3=v1+v2=5,6
3. 1 2

5 6

6 9

4/ z=10

0 5 5