# 面向对象程序设计🤣🤣🤣

## 类和对象

### 对象和类

- 类是对象的抽象概念，描述了对象的属性和行为。
- 对象是类的实例，通过实例化类可以创建一个具体的对象。

### 对象的访问控制

public，private，protected

### 对象的初始化和消亡

构造函数，析构函数，拷贝构造和赋值的区别

### 常量对象，静态对象，友元

const，声明常量对象时，必须同时进行初始化，因为常量对象的值在初始化后不可更改，常量对象可以调用常量成员函数（`const`成员函数），但不能调用非`const`成员函数。

static,静态成员变量是在类内声明，在类外定义时加上静态关键字的成员变量。静态成员函数只能访问静态成员变量和其他静态成员函数，不能访问非静态的成员变量和成员函数.

friend，它们可以是全局函数、其它的类或其它类的某些成员函数。友元不是类的成员。单向，不继承，不传递。

## 继承–派生类

### 单继承

### 消息的多态和动态绑定

多态性通过继承和方法重写（覆盖）实现。派生类可以重写基类的方法，从而以自己特定的方式响应相同的消息。在多态性中，消息是根据对象的实际类型来调用相应的方法，而不是变量或引用的声明类型。通过基类指针或引用调用派生类的方法，实现统一的接口。多态性也常用于实现抽象类和纯虚函数（Pure Virtual Function），在基类中声明纯虚函数，要求派生类必须实现该函数。

静态绑定和动态绑定：

- 静态绑定（Static Binding）是在编译时根据变量或引用的声明类型来确定方法的调用。静态绑定使用的是早期绑定（Early Binding）。
- 动态绑定是在运行时根据对象的实际类型来确定方法的调用。动态绑定使用的是晚期绑定。

虚函数和动态绑定：

– 虚函数是在基类中声明为虚函数的成员函数。
– 通过基类的指针或引用调用虚函数时，根据指针或引用所指向的实际对象的类型来选择正确的方法进行调用。
– 动态绑定确保了通过基类指针或引用调用虚函数时，实际被调用的是派生类中重写（覆盖）的虚函数。

### 抽象类，多继承

1. 抽象类的特点：
   – 抽象类无法实例化，只能用作其他类的基类。
   – 抽象类可以包含纯虚函数（Pure Virtual Function）和普通成员函数。
   – 抽象类中的纯虚函数没有实际的实现，需要在派生类中重写（覆盖）。
   – 抽象类可以包含数据成员和其他成员变量，这些成员对于派生类来说是可用的。
2. 纯虚函数：
   – 纯虚函数是在基类中声明为纯虚函数的成员函数，没有具体的实现。
   – 派生类必须重写（覆盖）基类的纯虚函数，才能实例化该派生类。
   – 声明纯虚函数的语法是在函数声明后加上`= 0`，表示没有默认实现。
3. 抽象类的作用：
   – 定义接口和规范：抽象类可以用于定义接口，规定派生类必须实现哪些方法。
   – 实现代码重用：抽象类可以提供一些通用的方法实现，被多个派生类共享。
   – 组织类的层次结构：抽象类可以作为其他类的基类，帮助组织类的继承关系。

C++通过虚继承（Virtual Inheritance）解决了菱形继承问题，确保共享的基类只有一个实例。

## 异常处理

### 异常的异地处理，结构化异常处理，try-catch

发现异常时，往往在异常的发现地（如在被调用的函数中）不知道如何处理这个异常，或者不能很好地处理这个异常。这时，可由程序的其它地方（如函数的调用者）来处理。

异常的异地处理（Exception Handling）是一种在程序中捕获和处理异常的机制，允许在异常发生的地方将异常传递到处理异常的代码块。

一种解决途径：通过函数的返回值，或指针/引用类型的参数，或全局变量把异常情况通知函数的调用者，由调用者处理异常。

另一种try-catch

– 异常处理使用 try-catch 块来捕获和处理异常。try 块用于包含可能抛出异常的代码，catch 块用于捕获并处理异常。

– 语法如下：

```
cppCopy code
try {
    // 可能抛出异常的代码
}
catch (ExceptionType1 e1) {
    // 处理 ExceptionType1 类型的异常
}
catch (ExceptionType2 e2) {
    // 处理 ExceptionType2 类型的异常
}
```

- catch语句块要紧接在某个try语句的后面。
- 如果在try语句块的<语句序列>执行中没有抛掷（throw）异常对象，则其后的catch语句不执行，而是继续执行try语句块之后的非catch语句。
- 异常处理可以嵌套

一个例子

```c++
void f()
{ try
  { g();
  }
  catch (int)
  {     ......
  }
  catch (char *)
  { ......
  }
}
void g()
{ try
  { h();
  }
  catch (int)
  { ......
  }
  ......
  ... throw 2; //由f捕获并处理
}
void h()
{       ......
        ... throw 1; //由g捕获并处理
        ......
        ... throw "abcd"; //由f捕获并处理
        ......
}
```

## IO（可不看，记得时候有点不耐烦了）

### 面向控制台的IO

面向控制台的I/O包括从标准输入设备（如：键盘）获得数据和把程序结果从标准输出设备（如：显示器）输出。

头文件#include<iostream>

- cin（istream类的对象）：对应着计算机系统的标准输入设备。（通常为键盘）

- cout（ostream类的对象）：对应着计算机系统的标准输出设备。（通常为显示器）

- cerr和clog（ostream类的对象）：对应着计算机系统用于输出特殊信息（如程序错误信息）的设备。（通常也对应着显示器，但不受输出重定向的影响）。cerr为不带缓冲的，clog为带缓冲的。

```c++
除了用插入操作符（<<）进行基本数据类型数据的输出外，还可以用ostream类的成员函数进行基于字节的输出操作。例如：
//输出一个字节。
ostream& ostream::put(char ch);
cout.put('A');
//输出p所指向的内存空间中count个字节。
ostream& ostream::write(const char *p,int count);
char info[100];
int n;
......
cout.write(info,n);

还可以用istream类的成员函数进行基于字节的输入操作。例如：
//输入一个字节。
istream& istream::get(char &ch);
//输入count个字节至p所指向的内存空间中。
istream& istream::read(char *p,int count);
//输入一个字符串。输入过程直到输入了count-1个字符或
//遇到delim指定的字符为止，并自动加上一个'\0'字符。
istream& istream::getline(char *p, int count, char delim='\n');
```

输出，输入格式可由操作符控制

### 面向文件的IO

面向文件的I/O包括从外存（文件）获得数据和把程序结果保存到外存（文件）中。

对文件数据进行读写的过程：

打开文件：把程序内部的一个表示文件的变量/对象与外部的一个具体文件关联起来，并创建内存缓冲区。

文件读/写：存取文件中的内容。

关闭文件：把暂存在内存缓冲区中的内容写入到文件中，并归还打开文件时申请的内存资源（包括内存缓冲区）

头文件： \#include <iostream> 以及#include <fstream>

打开文件：ofstream out_file(<文件名> [,<打开方式>]);或者ofstream out_file;  out_file.open(<文件名> [,<打开方式>]);

打开文件时，必须要对文件打开操作的成功与否进行判断。

文件输出操作结束时，要使用ofstream的成员函数close关闭文件：
out_file.close();

文件的输入，关闭等

## 操作符重载

### 单目操作符重载

```c++
定义格式
class <类名>
{       ......
        <返回值类型> operator # ();
};
<返回值类型> <类名>::operator # () { ...... }
使用格式
<类名> a;
#a
或，
a.operator#()
重点区分前置++和后置++，后置有个额外的int参数

        Counter& operator ++()  //前置的++重载函数
                {       value++;
                        return *this;
                }
                const Counter operator ++(int)  //后置的++重载函数
                {       Counter temp=*this; //保存原来的对象
                        value++; //写成：++(*this);更好！调用前置的++重载函数
                        return temp; //返回原来的对象
```

```
                }
++(++a);或 (++a)++; //OK，a加2(++x为左值表达式)
++(a++);或 (a++)++; //Error，编译不通过（x++为右值表达式）
```

### 双目操作符重载

略

### 特殊操作符重载

=，->,(),[],new,delete

操作符new有两个功能：

- 为动态对象分配空间

- 调用对象类的构造函数

操作符delete也有两个功能：

- 调用对象类的析构函数

- 释放动态对象的空间

重载操作符new和delete时，重载的是它们的分配空间和释放空间的功能，不影响对构造函数和析构函数的调用

智能指针

## 泛型,STL

### 函数模板

1. 函数模板的定义：

   - 函数模板使用 `template` 关键字定义，后跟模板参数列表和函数原型。

   - 模板参数列表中可以包含类型参数、非类型参数和模板参数的默认值。

   - 语法如下：

     ```
     cppCopy codetemplate <typename T>
     returnType functionName(T parameter1, T parameter2, ...) {
         // 函数实现
     }
     ```

2. 模板参数：

   - 函数模板使用模板参数来表示可以在函数中使用的类型或值。
   - 类型参数可以用 `typename` 或 `class` 关键字定义，并在函数中用于声明类型。
   - 非类型参数可以是整数、指针、引用等，并在函数中用于声明常量或确定数组大小等。

3. 实例化函数模板：

   - 当调用函数模板时，编译器会根据实际参数的类型来推导模板参数，并生成对应的函数实例。
   - 函数模板可以自动实例化为多个函数，以适应不同类型的参数。

4. 显式实例化和特化：

   - 显式实例化是指在代码中明确指定模板参数，强制编译器生成特定类型的函数实例。
max<int>(x,m);
   - 特化是指对特定的模板参数提供特定的实现，以覆盖通用模板的行为。

### 类模板

1. 类模板的定义：

   - 类模板使用 `template` 关键字定义，后跟模板参数列表和类定义。

   - 模板参数列表中可以包含类型参数、非类型参数和模板参数的默认值。

   - 语法如下：

     ```
     cppCopy codetemplate <typename T>
     class ClassName {
         // 类成员和方法定义
     };
     ```

2. 模板参数：

   - 类模板使用模板参数来表示可以在类中使用的类型或值。
   - 类型参数可以用 `typename` 或 `class` 关键字定义，并在类中用于声明成员变量、成员函数等。
   - 非类型参数可以是整数、指针、引用等，并在类中用于声明常量或确定数组大小等。

3. 实例化类模板：

   - 当使用类模板创建对象时，编译器会根据实际类型推导模板参数，并生成对应的类实例。

- 类模板可以自动实例化为多个类，以适应不同类型的对象。

4. 显式实例化和特化：

   - 显式实例化是指在代码中明确指定模板参数，强制编译器生成特定类型的类实例。
   - 特化是指对特定的模板参数提供特定的实现，以覆盖通用模板的行为。

   一个类模板的不同实例之间不共享该类模板中的静态成员

   模板是基于源代码的复用

### STL中的容器类模板，算法模板，迭代器

容器：vector，list，deque，stack，queue，priority_queue,map

如果容器的元素类型是一个类，则针对该类可能需要：自定义拷贝构造函数和赋值操作符重载函数,对容器进行的一些操作中可能会创建新的元素（对象的拷贝构造）或进行元素间的赋值（对象赋值）。

重载小于操作符（<）,对容器进行的一些操作中可能要对元素进行“小于”比较运算。

```c++
//利用容器vector来实现序列元素的存储与处理
   #include <iostream>
#include <vector>
using namespace std;
int main()
{ vector<int> v; //创建一个vector类容器对象v
   int x;
   cin >> x;
   while (x > 0) //循环往容器v中增加正的int型元素
   {  v.push_back(x); //往容器v的尾部增加一个元素
      cin >> x;
   }
   //求容器中所有元素的和以及最大与最小元素
   int sum=0,max,min;
   max = min = v[0]; //vector重载了操作符“[]”
   for (int i=0; i<v.size(); i++) //遍历容器元素
   { sum += v[i];
      if (v[i] < min) min = v[i];
      else if (v[i] > max) max = v[i];
   }
   cout << "sum=" << sum << ",max="
          << max << ",min="  << min << endl;
   return 0;
}
//求容器中所有元素的和以及最大与最小元素
   int sum=0,max,min;
   max = min = v[0];
   for (int n: v) //遍历容器元素（C++11）
```

```
   { sum += n;
      if (n < min) min = n;
      else if (n > max) max = n;
   }
   cout << "sum=" << sum << ",max="
          << max << ",min="  << min << endl;
   return 0;
}
```

迭代器（iterator）实现了抽象的指针（智能指针）功能，它们指向容器中的元素，用于对容器中的元素进行访问和遍历。

算法

## 消息（事件）驱动的程序设计

### 消息驱动的程序结构

消息队列：

- 消息队列是用于存储待处理消息的缓冲区。
- 消息队列可以实现消息的异步处理和解耦。

消息处理：

- 组件接收到消息后，根据消息的内容执行相应的操作。
- 消息处理可以触发其他组件的行为，产生新的消息，并可能修改组件的状态。

每个应用程序都有一个消息队列。统会把属于各个应用程序的消息放入各自的消息队列。应用程序不断地从自己的消息队列中获取消息并处理获得的消息。

"取消息–处理消息”的过程称为消息循环。

### 过程式windows应用程序设计

每个Windows应用程序都必须提供一个主函数WinMain，其主要功能是：注册窗口类（定义程序中要创建的窗口类型）：窗口的基本风格、消息处理函数、图标、光标、背景颜色以及菜单等。每类窗口（不是每个窗口）都需要注册.

### 面向对象的windows应用程序设计

## 函数式，逻辑式编程

1. 窗口对象

- 显示程序的处理数据。

– 处理Windows的消息、实现与用户的交互。

– 窗口对象类之间可以存在继承和组合关系。

2. 文档对象

– 管理在各个窗口中显示和处理的数据。

– 文档对象与窗口对象之间可以存在关联关系。

– 应用程序对象

3. 管理属于它的窗口对象和文档对象。

– 实现消息循环。

– 它与窗口对象及文档对象之间构成了组合关系。

### 程序设计范式简介

命令式程序设计范式是指：针对一个目标，需要给出达到目标的操作步骤和状态变化，即要对“如何做”进行详细描述。它们与冯诺依曼体系结构一致，是使用较广泛的程序设计范式，适合于解决大部分的实际应用问题。

命令式程序设计范式的典型代表：过程式程序设计，面向对象程序设计。

声明式程序设计范式是指：只需要给出目标的定义，不需要对如何达到目标（包括操作步骤和状态变化）进行描述，即只需要对“做什么”进行描述。有良好的数学理论支持，易于保证程序的正确性，并且，设计出的程序比较精炼和具有潜在的并行性。

声明式程序设计范式的典型代表：函数式程序设计，逻辑式程序设计。

```c++
计算:
a * b + c / d
命令式编程:
        t1 = a * b;
        t2 = c / d;
        r = t1 + t2;
函数式编程:
        ... add(multiply(a,b),divide(c,d)) ...
```

### 函数式程序设计

函数式程序设计（functional programming）是指把程序组织成一组数学函数，计算过程体现为基于一系列函数应用（把函数作用于数据）的表达式求值。函数也被作为值（数据）来看待，即函数的参数和返回值也可以是函数。

基本特征

1．纯”函数：以相同的参数调用一个函数总得到相同的值。（引用透明），除了产生计算结果，不会改变其他任何东西。（无副作用）

2．没有状态：计算体现为数据之间的映射，它不改变已有数据，而是产生新的数据。（无赋值操作）

3．函数也是值：函数的参数和返回值都可以是函数，可由已有函数生成新的函数。（高阶函数）

4．递归是主要的控制结构：重复操作采用函数的递归调用来实现，而不采用迭代（循环）
5．表达式的惰性（延迟）求值（Lazy evaluation）：一个表达式只有需要用到它的值的时候才会去计算它。
6．潜在的并行性：由于程序没有状态以及函数的引用透明和无副作用等特点，因此一些操作可以并行执行

过滤操作，映射操作，规约操作，部分函数应用，柯里化操作。

### 逻辑式程序设计

程序由一组事实和一组推理规则构成，在事实基础上运用推理规则来实施计算。基于的理论是谓词演算。

特征:数据（事实和规则）就是程序。计算（匹配、搜索、回溯）由实现系统自动完成。

# 一.简述题（欢迎补充）

## 1．为什么说封装与继承存在矛盾?

继承是指一个类（子类）继承另一个类（父类）的属性和方法，并且可以在此基础上添加、修改或扩展功能。封装是将数据和操作数据的方法封装在一个单元（类）中，通过对外提供公共接口（方法）来控制对数据的访问。当一个类继承自另一个类时，它会继承父类的属性和方法，包括那些可能是私有或受限制的。这就破坏了封装的原则，因为子类可以直接访问父类的私有成员，绕过了封装的控制。

## 2．深拷贝和浅拷贝的区别是什么？浅拷贝可能会引发哪些问题?

浅拷贝是创建一个新对象，将原始对象的引用复制到新对象中。这意味着新对象与原始对象共享同一块内存，任何对其中一个对象的修改都会影响另一个对象。简而言之，浅拷贝只复制了对象的引用，而不是对象本身的内容。

深拷贝则是创建一个新对象，并递归地复制原始对象及其所有嵌套对象的内容。这意味着新对象与原始对象是完全独立的，对其中一个对象的修改不会影响另一个对象。换句话说，深拷贝会复制对象的所有内容，包括嵌套的对象。

指针指向同一空间导致该空间的值更改受两个指针的影响  两次归还内存  使用已归还的空

间。

## 3. 构造函数以及析构函数调用顺序?

创建：分配内存（对象）–> 父类构造–> 成员构造–> 自己构造
父类构造：按照继承表从左到右依次构造。
成员构造：按照声明顺序从上至下依次构造。
释放：自己析构–> 成员析构–> 父类析构–> 释放内存（对象）
成员析构：按照声明顺序从下到上依次构造。
父类析构：按照继承表从右到左依次构造。

在虚拟继承的情况下，虚基类的构造函数会在派生类的构造函数之前调用，而析构函数的调用顺序与一般继承相同，派生类的析构函数先于虚基类的析构函数调用。

## 4. 初始化=和赋值=有什么区别?

初始化调用拷贝构造函数，赋值调用=重载函数。

# 二.程序分析题

# 三.程序题

## 将递归函数改为尾递归函数

尾递归：由于函数递归调用效率低、递归调用深度受栈空间的限制，因此，函数式编程常采用尾递归，即递归调用是函数的最后一步操作。

常规递归和尾递归的区别在于函数的调用和返回顺序以及对栈空间的利用方式不同。

具体来说，常规递归的过程是：每次函数调用时需要记录当前函数上下文的信息，包括传入参数、局部变量值等，并将这些信息保存在栈空间（也称调用栈或运行时栈）中。在递归过程中，每一次递归调用都会新建一个对应的栈帧（栈的一层）并保存上述信息。
而尾递归则是指递归过程中，只有一个栈帧被不断的更新。即，函数执行完成后直接返回结果，不再进行任何其他操作。这样一来，由于只占用一个栈帧，程序在空间上会得到很大优化，极大地降低了内存负担。同时，尾递归也可以被优化为迭代的形式，进一步减少时间负担。

尾递归的**本质**，其实是将递归方法中的需要的“所有状态”通过方法的参数传入下一次调用中。

```c++
int fib(int n)
{       if (n == 1 || n == 2)  return 1;
        else  return fib(n-2)+fib(n-1);
}
cout << fib(10) << endl; //求第10个fibonacci数
```

求第n个fibonacci数的函数式方案2（尾递归）

```
int fib(int n, int a, int b)
{       if (n == 1)  return a;
        else if (n == 2) return b;
        else  return fib(n-1,b,a+b);
}
cout << fib(10,1,1) << endl; //求第10个fibonacci数
```

一般形式的尾递归：

```
T f(T1 x1, T2 x2, ...)
{  ......
    ... return f(m1,m2,...);
    ......
    ... return f(n1,n2,...);
    ......
}
```

转换成等价的迭代：

```
T f(T1 x1, T2 x2, ...)
{  while (true)
    { ......
       ... { T1 t1=m1; T2 t2=m2; ...
              x1 = t1; x2 = t2; ... continue;}
       ......
       ... { T1 t1=n1; T2 t2=n2; ...
              x1 = t1; x2 = t2; ... continue;}
       ......
    }
}
```

## 请根据下面的使用示例和相应的输出，设计相应的类和全局函数。