# C 和 C++ 区别 （函数/类/struct/class）

## new delete和malloc free区别：

前者是运算符，后者是库函数，malloc和free只负责分配和释放内存空间，new在分配内存空间后会自动调用对象的构造函数，delete在释放内存空间前会自动调用对象的析构函数。

## explicit

class Point {

public:

int x, y;

explicit Point(int x = 0, int y = 0)

: x(x), y(y) {}

};

void displayPoint(const Point& p)

{

cout << "(" << p.x << ","

<< p.y << ")" << endl;

}

int main()

{

//报错，explicit声名Point对象为显示的，不能隐式调用

displayPoint(1);

Point p(1);

}

# 智能指针

在写程序的时候使用指针容易出现内存泄漏，智能指针超出了自身作用域就会自动调用析构函数，自动释放资源。

## 常用接口

T\* get(); //获取封装在内部的原始指针

T& operator\*(); //等价于指针的\*

T\* operator->(); //等价于指针的->

T& operator=(const T& val);

T\* release(); //释放指针，返回释放前的指针

void reset (T\* ptr = nullptr); //将内部指针置为ptr

## auto\_ptr

c++11已弃用

auto\_ptr<std::string> p1 (new string ("hello"));

auto\_ptr<std::string> p2;

p2 = p1; //auto\_ptr 不会报错

std::cout << \*p1 << std::endl; //报错，p1已经被p2抢占

## unique\_ptr（唯一型）

auto\_ptr<std::string> p1 (new string ("hello"));

auto\_ptr<std::string> p2;

p2 = p1; //报错，unique\_ptr保证同一时间只有一个智能指针指向该对象，//不允许p2抢占p1

## shared\_ptr（共享型，强引用）

shared\_ptr是为了解决auto\_ptr, unique\_ptr在对象所有权上的局限性(auto\_ptr是独占的)，在使用引用计数的机制上提供了可以共享所有权的智能指针。shared\_ptr 实现共享式拥有概念，多个智能指针可以指向相同对象，该对象和其相关资源会在“最后一个引用被销毁”时候释放。它使用计数机制来表明资源被几个指针共享。

可以通过成员函数use\_count()来查看资源的所有者个数，除了可以通过 new 来构造，还可以通过传入auto\_ptr, unique\_ptr, weak\_ptr来构造。当我们调用release()时，当前指针会释放资源所有权，计数减1。当计数等于 0时，资源会被释放。

## weak\_ptr（弱引用）

weak\_ptr是用来解决shared\_ptr循环引用问题的，它指向一个shared\_ptr对象，但它的构造和析构都不会影响计数的大小，可以使用lock函数转化为shared\_ptr。

循环引用的一个例子是有两个类A和B，A类中有一个shared\_ptr<B>类型的数据成员，B类中有一个shared\_ptr<A>类型的数据成员，这两个数据成员相互指向A和B的实例化对象，这样两个shared\_ptr的计数都是2，在程序结束时，两个shared\_ptr的计数都无法降低至0，会导致A和B的资源都不会释放。此时使用weak\_ptr代替其中一个shared\_ptr，这样有一个计数就是1，程序结束时一个计数就会降低至0，释放资源，然后另一个shared\_ptr计数也会降低至0，释放资源。

# 值传递、指针传递和引用传递

## 值传递

当实参以值传递的形式传入函数时，形参是实参的拷贝，所以在函数内部改变形参的值不会影响实参。

## 指针传递

指针传递本质上也是值传递，它所传递的是一个地址，在函数内部通过指针地址修改指针指向的内容是没有问题的，但修改形参地址的指向是不会影响实参的。

## 引用传递

引用是变量的一种别名，当实参以引用传递的形式传入函数时，形参和实参的地址是一样的，修改形参会之间影响实参。

# #define, const和static

## #define

#define是宏替换，是预处理器的一部分，不会进行类型检查。

## const

### 修饰变量

修饰变量，变量不能修改。

const int foo = 1;

foo = 2; // compile time error

const int\* ptr = &foo;

\*ptr = 3 // compile time error

int fuck = 0;

ptr = &fuck; // this is OK

\*ptr = 123; // compile time error

int\* const ptr2 = &foo;

\*ptr2 = 3; // this is OK

ptr2 = &fuck; // compile time error

### 修饰函数

修饰函数，不能修改类数据成员，也不能调用非const函数。

struct FooBar

{

int member;

int MyMethod(int value) const

{

member = value; // compile time error

}

};

①const Stock & Stock::topval (②const Stock & s) ③const

①处const：确保返回的Stock对象在以后的使用中不能被修改。

②处const：确保此方法不修改传递的参数 S。

③处const：保证此方法不修改调用它的对象，const对象只能调用const成员函数,不能调用非const函数。

## static

### 修饰全局变量

用在全局变量，表明这个变量在每个编译单元有独自的实例：

// foo.h

static int a = 123;

// foo.cpp

#include "foo.h"

int foo\_func() { return a++; }

// bar.cpp

#include "foo.h"

int bar\_func() { return a++; }

### 修饰函数中的局部变量

用在函数里的局部变量，表明它的生存周期其实是全局变量，但仅在函数内可见。访问这个函数的时候，static变量只有在第一次访问时初始化。

int get\_global\_id()

{

static int seed = 0;

return seed++;

}

### 修饰类中的函数

用在类成员，表明成员或者方法是类的，而不是对象实例的。

每个Foo实例都含有一个int a。bbb方法通过Foo::bbb()调用。

struct Foo

{

int a = 0;

static int aaa = 0;

static int bbb() { return 123456; }

};

# C与C++的区别

* C没有class，是面向过程的，C++有class，是面向对象的。
* C与C++本质区别是C++有多态特性，C没有。如C++有函数重载，C没有，C++可以通过继承与虚函数实现多态。
* C++新增了STL标准库，以及语法和关键字，如new、delete、auto、explicit。

# Java和C++的区别

## 指针

Java 语言让程序员没法找到指针来直接访问内存，没有指针的概念，并有内存的自动管理功能，从而有效的防止了C++语言中的指针操作失误的影响。但并非Java 中没有指针，Java 虚拟机内部中还是用了指针，保证了Java 程序的安全。

## 多重继承

C++支持多重继承但 Java 不支持，但支持一个类继承多个接口，实现C++中多重继承的功能，又避免了C++的多重继承带来的不便。

数据类型和类：Java 是完全面向对象的语言，所有的函数和变量必须是类的一部分。除了基本数据类型之外，其余的都作为类对象，对象将数据和方法结合起来，把它们封装在类中，这样每个对象都可以实现自己的特点和行为。Java没有C++中的struct和union 。

## 自动内存管理

Java程序中所有对象都是用new操作符建立在内存堆栈上，Java自动进行无用内存回收操作，不需要程序员进行手动删除。而C++中必须由程序员释放内存资源。Java 中当一个对象超出作用域时，利用空闲时间工作来删除。

## 操作符重载

Java不支持操作符重载。操作符重载被认为是C++的突出特性。

## 预处理

Java不支持预处理功能。C++ 在编译过程中都有一个预编译阶段，Java没有预处理器，但它提供了import与C++预处理器具有类似功能。

## 类型转换

C++中有数据类型隐含转换的机制，Java中需要强制类型转换。

# 重载、重写和重定义

## 重载（overload）

重载是指在同一个类（或者作用域）中函数名相同但是参数个数或者参数类型不同，一般重载不关心返回类型。

## 重写（override）

重写是指子类重新实现父类的虚函数。在父类指针指向子类对象时，指针调用父类被重写的函数会直接调用子类重写后的函数，这是多态的一种实现方式。

## 重定义

重定义是指子类重写实现父类的非虚函数，此时在子类中父类的方法被隐藏。在父类指针指向子类对象时，指针调用父类被重写的函数会调用父类的该函数。

class Animal {

public:

void func1(int tmp) {

cout << "animal func1-" << tmp << endl;

}

void func1(const char\* s) {

cout << "animal func1-" << s << endl;

}

virtual void func2(int tmp) {

cout << "animal virtual func2-" << tmp << endl;

}

void func3(int tmp) {

cout << "animal func3-" << tmp << endl;

}

};

class Fish :public Animal {

public:

//函数的重定义 会隐藏父类同名方法

void func1() {

cout << "fish func1" << endl;

}

//函数的重写， 覆盖父类的方法 override

void func2(int tmp) {

cout << "fish func2-" << tmp << endl;

}

//函数的重定义 会隐藏父类同名方法

void func3(int tmp) {

cout << "fish func3-" << tmp << endl;

}

};

int main() {

Fish fi;

Animal an;

fi.func1();

//fish func1

// 由于是重定义 父类的方法已经被隐藏

// 需要显示声明，重载不能跨作用域

fi.Animal::func1(1);

//animal func1-1

dynamic\_cast<Animal\*>(&fi)->func1(11); // 强转之后即可调用到父类被隐藏的方法

//animal func1-11

dynamic\_cast<Animal\*>(&fi)->func1("hello world"); // 强转之后即可调用到父类被隐藏的方法

//animal func1-hello world

fi.func2(2); // 调用子类

//fish func2-2

dynamic\_cast<Animal\*>(&fi)->func2(22); // 调用"子类方法"(因为是虚函数，会被子类覆盖)

//fish func2-22

dynamic\_cast<Animal\*>(&fi)->func3(222); // 调用父类

//animal func3-222

fi.func3(2222); // 调用子类

//fish func3-2222

an.func1(1);

//animal func1-1

an.func1("I'm an animal");

//animal func1-I'm an animal

an.func2(1);

//animal virtual func2-1

return 0;

}

# 构造函数

类的对象被创建时，编译系统为对象分配内存空间，并自动调用构造函数，由构造函数完成成员的初始化工作。即构造函数的作用：初始化对象的数据成员。

## 无参数构造函数

即默认构造函数，如果没有明确写出无参数构造函数，编译器会自动生成默认的无参数构造函数，函数为空，什么也不做，如果不想使用自动生成的无参构造函数，必需要自己显示写出一个无参构造函数。

## 一般构造函数

也称重载构造函数，一般构造函数可以有各种参数形式，一个类可以有多个一般构造函数，前提是参数的个数或者类型不同，创建对象时根据传入参数不同调用不同的构造函数。

## 拷贝构造函数

拷贝构造函数的函数参数为对象本身的引用，用于根据一个已存在的对象复制出一个新的该类的对象，一般在函数中会将已存在的对象的数据成员的值一一复制到新创建的对象中。如果没有显示的写拷贝构造函数，则系统会默认创建一个拷贝构造函数，但当类中有指针成员时，最好不要使用编译器提供的默认的拷贝构造函数，最好自己定义并且在函数中执行深拷贝。

## 类型转换构造函数

根据一个指定类型的对象创建一个本类的对象，也可以算是一般构造函数的一种，这里提出来，是想说有的时候不允许默认转换的话，要记得将其声明为 explict 的，来阻止一些隐式转换的发生。

## 赋值运算符的重载

注意，这个类似拷贝构造函数，将＝右边的本类对象的值复制给＝左边的对象，它不属于构造函数，＝左右两边的对象必需已经被创建。如果没有显示的写赋值运算符的重载，系统也会生成默认的赋值运算符，做一些基本的拷贝工作。

A a1, A a2; a1 = a2;//调用赋值运算符

A a3 = a1;//调用拷贝构造函数，因为进行的是初始化工作，a3 并未存在

# C++的四种强制转换

## static\_cast

明确指出类型转换，一般建议将隐式转换都替换成显示转换，因为没有动态类型检查，上行转换（派生类->基类）安全，下行转换（基类->派生类）不安全，所以主要执行非多态的强制转换操作。

## dynamic\_cast

专门用于派生类之间的转换，type-id必须是类指针，类引用或void\*，对于下行转换是安全的，当类型不一致时，转换过来的是空指针，而static\_cast，当类型不一致时，转换过来的事错误意义的指针，可能造成非法访问等问题。

Fish fi;

Animal\* an = dynamic\_cast<Animal\*>(&fi);

Fish new\_fi = dynamic\_cast<Fish\*>(&an);

## const\_cast

专门用于const属性的转换，去除const性质，或增加const性质，是四个转换符中唯一一个可以操作常量的转换符。一般只能操作指针和引用，不能直接将常量对象直接转化为非常量对象。

const int ica = 100;

int \* ia = const\_cast<int \*>(&ica);

\*ia = 200;

cout << \*ia << ica << endl; //200 100

const int i = 100;

int j = const\_cast<int>(i); //不允许

## reinterpret\_cast

不到万不得已，不要使用这个转换符，高危操作。使用特点：从底层对数据进行重新解释，依赖具体的平台，可移植性差；可以将整形转换为指针，也可以把指针转换为数组；可以在指针和引用之间进行肆无忌惮的转换。当无法使用static\_cast进行强制转换时，又真的确保转换是可以安全转换时，可以使用reinterpret\_cast，用法和static\_cast一样。

class int\_type {

public:

int value;

};

int main(void) {

int\_type t;

t.value = 233;

int\* p = reinterpret\_cast<int\*>(&t);

\*p = 666;

std::cout << t.value << std::endl;

return 0;

}

上面代码中将int\_type类型的对象t的地址强制从int\_type \*转换成int \*。类int\_type只有一个成员变量并且是int类型的，占用的大小就是int的大小。所以使用int的指针操作int\_type的地址是没有问题的，不会操作到无关的内存。如果使用static\_cast，编译器检测这两个类型不是同一种类型，所以会报错。

注意：如果上面使用long long \*p来操作int\_type，由于long long会操作8个字节的内存，而int\_type只有4字节，这样将会导致内存中的数据错误。所以使用reinterpret\_cast的时候要非常小心。

# 指针和引用的区别

* 指针存储的是地址，指向内存的一个存储单元，引用是原变量的一个别名，和原变量实质上是相同的。
* 指针可以有多级，引用只能有一级。
* 指针在定义时可以不初始化，引用在定义时必须初始化。
* 指针可以指向NULL，引用不行。
* 指针初始化后可以再改变，引用不行。

# 野(wild)指针与悬空(dangling)指针有什么区别？如何避免？

* 野指针是没有初始化的指针，避免野指针需要保证使用指针前初始化或者使用智能指针。
* 悬空指针是已经被释放的指针，避免悬空指针可以在释放指针后将其赋值为nullptr或者使用智能指针。、

# 函数指针

用途：回调函数

char \* fun(char \* p) {…} // 函数fun

char \* (\*pf)(char \* p); // 函数指针pf

pf = fun; // 函数指针pf指向函数fun

pf(p); // 通过函数指针pf调用函数fun

# volatile关键字

volatile修饰的变量就是告诉编译器这个变量可能会出现突发情况，不要对其过度优化，每次使用该变量都需要从内存中读取。

# extern关键字

## extern+变量/函数

声明此变量/函数是在别处定义的，需要在此处引用。

## extern “C”

在C++中调用C库函数，可以使用extern “C”修饰函数，告诉链接器用C函数规范来链接。

# 面向对象的三大特性

## 封装

就是把客观事物封装成抽象的类，将某些数据和函数设置为私有，对外部隐藏，将某些数据和函数设置为共有，供外部调用。

## 继承

A继承B，A是子类（派生类），B是父类（基类），A和B的关系是A是B。例如A是Dog，B是Animal。子类拥有父类的数据成员和成员函数，并在父类的基础上进行拓展。

## 多态

多态简单来说就是一个接口，可以实现多种方法，比如静态多态：重载。比如动态多态：父类的指针指向子类的对象，使用父类的指针调用虚函数时，会得到不同子类重写的不同结果，也就是通过继承加虚函数来实现的多态。

# 析构函数一般写成虚函数的原因

当一个基类的指针指向一个派生类的对象并且需要销毁时，如果基类的析构函数没有设置成虚函数，那么不会调用派生类的析构函数，只会调用基类的析构函数，可能会出现内存泄漏；如果设置成虚函数，销毁时由于多态会先调用派生类的析构函数，再调用基类的析构函数。

# 构造函数为什么不能是虚函数

虚函数的调用只需要知道基类的信息，不需要知道派生类的具体信息，但是创建对象，调用构造函数时是需要知道对象的具体信息的。虚函数的意义在于基类指针调用派生类对象的成员函数。构造函数时在创建对象时主动调用的，不能通过基类的指针来调用，所以构造函数设置为虚函数是没有意义的。

# 构造函数和析构函数是否应该调用虚函数？

不应该，虚函数的意义在于基类的指针调用派生类对象的成员函数，在创建基类对象时只应该对基类的数据成员作初始化，不应该涉及派生类的方法。析构函数也是一样，基类的善后操作不应该涉及派生类的数据成员。

# 构造函数和析构函数的执行顺序

## 构造函数顺序

* 基类构造函数。如果有多个基类，则构造函数的调用顺序是某类在类派生表中出现的顺序，而不是它们在成员初始化表中的顺序。
* 成员类对象构造函数。如果有多个成员类对象则构造函数的调用顺序是对象在类中被声明的顺序，而不是它们出现在成员初始化表中的顺序。
* 派生类构造函数。

## 析构函数顺序

* 派生类的析构函数。
* 成员类对象的析构函数。
* 基类的析构函数。

# 纯虚函数

拥有纯虚函数的类是抽象类，不能实例化。继承抽象类的派生类必须重写纯虚函数，否则该类也是抽象类，不能实例化。纯虚函数的意义在于提供一个派生类必须实现的接口。

# 深拷贝和浅拷贝

默认的类的拷贝构造函数是浅拷贝，如果类的数据成员中没有指针，那么浅拷贝是可行的，如果有指针，复制后两个类中的两个指针会指向同一个地址，调用析构函数会释放指针两次，导致出错。而深拷贝则会在堆中申请空间来存储指针指向的数据。

# 什么情况会调用拷贝构造函数？

* 一个对象通过另一个对象进行初始化
* 当一个对象以值传递的方式作为形参传入函数
* 当一个对象以值传递的方式作为返回值从函数返回

# 拷贝构造函数可以使用值传递吗？

不能，只能使用引用传递，如果使用值传递，作为形参输入函数时会调用拷贝构造函数，这样会循环调用，导致堆栈溢出。

# 如何监测内存泄漏？

valgrind