C++面试八股文快问快答の基础篇

文章目录

- 基础篇
  - 变量的声明和定义有什么区别
  - 简述#ifdef、#else、#endif和#ifndef的作用
  - 写出int 、bool、 float 、指针变量与 “零值”比较的if 语句
  - 结构体可以直接赋值吗
  - sizeof 和strlen 的区别
  - sizeof求类型大小
  - C 语言的关键字 static 和 C++ 的关键字 static 有什么区别
  - C 语言的 malloc 和 C + + 中的 new 有什么区别
  - 写一个 “标准”宏MIN
  - ++i和i++的区别
  - volatile有什么作用
  - C++中volatile的作用
  - 一个参数可以既是const又是volatile吗
  - a 和&a 有什么区别
  - 用C 编写一个死循环程序
  - 结构体内存对齐问题
  - 全局变量和局部变量有什么区别？实怎么实现的？操作系统和编译器是怎么知道的？
  - 简述C、C++程序编译的内存分配情况
  - 简述strcpy、sprintf 与memcpy 的区别
  - 请解析((void ()( ))0)( )的含义
  - typedef 和define 有什么区别
  - 指针常量与常量指针区别
  - 简述队列和栈的异同
  - 设置地址为0x67a9 的整型变量的值为0xaa66
  - 编码实现字符串转化为数字
  - C语言的结构体和C++的有什么区别
  - 简述指针常量与常量指针的区别
  - 如何避免“野指针”
  - 句柄和指针的区别和联系是什么?
  - new/delete与malloc/free的区别是什么
  - 说一说extern“C”
  - 请你来说一下C++中struct和class的区别
  - C++类内可以定义引用数据成员吗?
  - C++中类成员的访问权限
  - 什么是右值引用，跟左值又有什么区别?
  - 面向对象的三大特征
  - 说一说c++中四种cast转换
    - 1、const_cast
    - 2、static_cast
    - 3、dynamic_cast*
    - 4、reinterpret_cast
    - 5、为什么不使用C的强制转换?
  - C++的空类有哪些成员函数
  - 对c++中的smart pointer四个智能指针：shared_ptr,unique_ptr,weak_ptr,auto_ptr的理解
  - 说说强制类型转换运算符
  - 谈谈你对拷贝构造函数和赋值运算符的认识
  - 在C++中，使用malloc申请的内存能否通过delete释放？使用new申请的内存能否用free?
  - 用C++设计一个不能被继承的类
  - C++自己实现一个String类
  - 访问基类的私有虚函数

对虚函数和多态的理解
简述类成员函数的重写、重载和隐藏的区别
（1）重写和重载主要有以下几点不同。
（2）隐藏和重写、重载有以下几点不同。
链表和数组有什么区别
存储形式：
数据查找：
越界问题：
注意：
用两个栈实现一个队列的功能
共享数据的保护
常引用：
常对象：
常成员函数：
extern int a:使其他文件也能访问该变量
程序内存分配方式以及它们的区别
栈区（stack）
堆区（heap）
全局区（静态区）（static）
常量存储区
程序代码区
explicit
mutable关键字
用const修饰函数的返回值
宏、const和enum
static的生存期
全局变量和static变量的区别
为什么栈要比堆速度要快
c++ 析构函数调用时间
静态绑定 动态绑定 （也叫动态连编，静态连编）
C语言的指针和c++的引用有什么区别?
请你说说C语言是怎么进行函数调用的
C++中拷贝赋值函数的形参能否进行值传递?
include头文件的顺序以及双引号""和尖括号<>的区别
一个C++源文件从文本到可执行文件经历的过程
内存泄漏原因和判断方法
段错误的产生原因
段错误是什么
段错误产生的原因
C++ 函数调用过程
如何调试c++ 多线程程序?
面向对象和面向过程的区别
（过程）优点：
（对象）优点：
关于引用赋值的多态：
模板的声明和实现不能分开的原因
C++类中引用成员和常量成员的初始化（初始化列表）
memset为int型数组初始化问题
编译器对 inline 函数的处理步骤
优缺点
优点
缺点
虚函数（virtual）可以是内联函数（inline）吗?
静态库和动态库比较
静态库
动态库（共享库）
区别
虚函数、虚函数表，虚指针
C/C++如何判断两个小数是否相等

C++空类的大小
c++ 空类，含有虚函数的类的大小(此问题都是在32位机器上而言)
32位机与64位机指针占用空间不同
引经据典

基础篇

变量的声明和定义有什么区别

变量的定义为变量分配地址和存储空间， 变量的声明不分配地址。一个变量可以在多个地方声明， 但是只在一个地方定义。 加入 extern 修饰的是变量的声明，说明此变量将在文件以外或在文件后面部分定义。
说明：很多时候一个变量，只是声明不分配内存空间，直到具体使用时才初始化，分配内存空间， 如外部变量。

```
int main()
{
    extern int A;
    //这是个声明而不是定义，声明A是一个已经定义了的外部变量
    //注意：声明外部变量时可以把变量类型去掉如：extern A;
    dosth(); //执行函数
}
int A; //是定义，定义了A为整型的外部变量
```

简述#ifdef、#else、#endif和#ifndef的作用

利用#ifdef、#endif将某程序功能模块包括进去，以向特定用户提供该功能。在不需要时用户可轻易将其屏蔽。

```
#ifdef MATH
    #include “math.c”
#endif

//在子程序前加上标记，以便于追踪和调试。

#ifdef DEBUG
    printf (“Indebugging…!”);
#endif
```

应对硬件的限制。由于一些具体应用环境的硬件不一样，限于条件，本地缺乏这种设备，只能绕过硬件，直接写出预期结果。
注意：虽然不用条件编译命令而直接用if语句也能达到要求，但那样做目标程序长（因为所有语句都编译），运行时间长（因为在程序运行时间对if语句进行测试）。而采用条件编译，可以减少被编译的语句，从而减少目标程序的长度，减少运行时间。

写出int 、bool、 float 、指针变量与 “零值”比较的if 语句

```
//int与零值比较
if ( n == 0 )
if ( n != 0 )

//bool与零值比较
if (flag) // 表示flag为真
if (!flag) // 表示flag为假

//float与零值比较
const float EPSINON = 0.00001;
if ((x >= - EPSINON) && (x <= EPSINON) //其中EPSINON是允许的误差（即精度）。

//指针变量与零值比较
if (p == NULL)
if (p != NULL)
```

结构体可以直接赋值吗

声明时可以直接初始化，同一结构体的不同对象之间也可以直接赋值，但是当结构体中含有指针“成员”时一定要小心。
注意：当有多个指针指向同一段内存时，某个指针释放这段内存可能会导致其他指针的非法操作。因此在释放前一定要确保其他指针不再使用这段内存空间。

sizeof 和strlen 的区别

sizeof是一个**操作符**，strlen是库函数。
sizeof的参数可以是数据的类型，也可以是变量，而strlen只能以结尾为'\0'的字符串作参数。
编译器在编译时就计算出了sizeof的结果，而 strlen函数 必须在运行时才能计算出来。并且sizeof计算的是数据类型占内存的大小，而strlen计算的是字符串实际的长度。
数组做sizeof的参数不退化，传递给strlen就退化为指针了

sizeof求类型大小

类的大小为类的非静态成员数据的类型大小之和，也就是说 静态成员 数据不作考虑。
普通成员函数与sizeof无关。
虚函数由于要维护在虚函数表，所以要占据一个指针大小，也就是4字节。
类的总大小也遵守类似class字节对齐的，调整规则。
例如有如下结构体：

```
struct Stu
{
    int id;
    char sex;
    float hight;
};
```

那么一个这样的结构体变量占多大内存呢？也就是
cout<<sizeof(Stu)<<endl; 会输出什么？
在了解字节对齐方式之前想当然的会以为：sizeof(Stu) = sizeof(int)+sizeof(char)+sizeof(float) = 9.
然而事实并非如此！

字节对齐原则:在系统默认的对齐方式下：每个成员相对于这个结构体变量地址的偏移量正好是该成员类型所占字节的整数倍，且最终占用字节数为成员类型中最大占用字节数的整数倍。

在这个例子中，id的偏移量为0（0=40），sex的偏移量为4（4=14），hight的偏移量为8（8=24），此时占用12字节，也同时满足12=34.所以sizeof(Stu)=12.

```
struct A {
    char y;
    char z;
    long long x;
};    16字节
struct A {
    char y;
    char z;
    int x;
};  8字节

struct A {
    char y;
    char* z;
    int x;
};12字节
struct A {
    char y;
};  1字节
```

我的总结：

最终大小一定是最大数据类型的整数倍；
静态变量不占空间
每种类型的偏移量为自身的n倍；
详细请查阅：struct/class等内存字节对齐问题详解

C 语言的关键字 static 和 C++ 的关键字 static 有什么区别

在 C 中 static 用来修饰局部静态变量和外部静态变量、函数。而 C++中除了上述功能外，还用来定义类的成员变量和函数。即静态成员和静态成员函数。
注意：编程时 static 的记忆性，和全局性的特点可以让在不同时期调用的函数进行通信，传递信息，而 C++的静态成员则可以在多个对象实例间进行通信，传递信息。

C 语言的 malloc 和 C + + 中的 new 有什么区别

new 、delete 是操作符，可以重载，只能在C++ 中使用。
malloc、free 是函数，可以覆盖，C、C++ 中都可以使用。
new 可以调用对象的构造函数，对应的delete 调用相应的析构函数。
malloc 仅仅分配内存，free 仅仅回收内存，并不执行构造和析构函数
new 、delete 返回的是某种数据类型指针，malloc、free 返回的是void 指针。
注意：malloc 申请的内存空间要用free 释放，而new 申请的内存空间要用delete 释放，不要混用。

写一个 “标准”宏MIN

```
#define min(a,b) ((a)<=(b)?(a):(b))
```

++i和i++的区别

++i先自增1，再返回，i++先返回i,再自增1

volatile有什么作用

状态寄存器一类的并行设备硬件寄存器。
一个中断服务子程序会访问到的非自动变量。
多线程间被几个任务共享的变量。
注意：虽然volatile在嵌入式方面应用比较多，但是在PC软件的多线程中，volatile修饰的临界变量也是非常实用的。

C++中volatile的作用

总结：建议编译器不要对该变量进行优化

volatile是“易变的”、“不稳定”的意思。volatile是C的一个较为少用的关键字，它用来解决变量在“共享”环境下容易出现读取错误的问题。

定义为volatile的变量是说这变量可能会被意想不到地改变，即在你程序运行过程中一直会变，你希望这个值被正确的处理，每次从内存中去读这个值，而不是因编译器优化从缓存的地方读取，比如读取缓存在寄存器中的数值，从而保证volatile变量被正确的读取。

在单任务的环境中，一个函数体内部，如果在两次读取变量的值之间的语句没有对变量的值进行修改，那么编译器就会设法对可执行代码进行优化。由于访问寄存器的速度要快过RAM（从RAM中读取变量的值到寄存器），以后只要变量的值没有改变，就一直从寄存器中读取变量的值，而不对RAM进行访问。

而在多任务环境中，虽然在一个函数体内部，在两次读取变量之间没有对变量的值进行修改，但是该变量仍然有可能被其他的程序（如中断程序、另外的线程等）所修改。如果这时还是从寄存器而不是从RAM中读取，就会出现被修改了的变量值不能得到及时反应的问题。如下程序对这一现象进行了模拟。

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

int main(int argc,char* argv[])
{
    int i=10;
    int a=i;
    cout<<a<<endl;
    _asm
    {
        mov dword ptr [ebp-4],80
    }
    int b=i;
    cout<<b<<endl;
}
/*
程序在VS2012环境下生成Release版本，输出结果是：
10
10
*/
```

阅读以上程序，注意以下几个要点：

以上代码必须在Release模式下考查，因为只有Release模式下才会对程序代码进行优化，而这种优化在变量共享的环境下容易引发问题。

在语句b=i；之前，已经通 过内联汇编代码修改了i的值，但是i的变化却没有反映到b中，如果i是一个被多个任务共享的变量，这种优化带来的错误很可能是致命的。
汇编代码[ebp-4]表示变量i的存储单元，因为ebp是扩展基址指针寄存器，存放函数所属栈的栈底地址，先入栈，占用4个字节。随着函数内申明的局部变量的增多，esp（栈顶指针寄存器）就会相应的减小，因为栈的生长方向由高地址向低地址生长。i为第一个变量，栈空间已被ebp入栈占用了4个字节，所以i的地址为ebp-i，[ebp-i]则表示变量i的存储单元。

一个参数可以既是const又是volatile吗

可以，用const和volatile同时修饰变量，表示这个变量在程序内部是只读的，不能改变的，只在程序外部条件变化下改变，并且编译器不会优化这个变量。每次使用这个变量时，都要小心地去内存读取这个变量的值，而不是去寄存器读取它的备份。
注意：在此一定要注意const的意思，const只是不允许程序中的代码改变某一变量，其在编译期发挥作用，它并没有实际地禁止某段内存的读写特性。

a 和&a 有什么区别

&a：其含义就是“变量a的地址”。
*a：用在不同的地方，含义也不一样。
在声明语句中，*a只说明a是一个指针变量，如int *a；
在其他语句中，*a前面没有操作数且a是一个指针时，*a代表指针a指向的地址内存放的数据，如b=*a；
*a前面有操作数且a是一个普通变量时，a代表乘以a，如c=ba。

用C 编写一个死循环程序

```
while(1)
{
}
```

注意：很多种途径都可实现同一种功能，但是不同的方法时间和空间占用度不同，特别是对于嵌入 式软件，处理器速度比较慢，存储空间较小，所以时间和空间优势是选择各种方法的首要考虑条件。

结构体内存对齐问题

请写出以下代码的输出结果：

```
#include <stdio.h>

using namespace std;
/*************************************************************
*       结构体内存对齐问题
*   从偏移为0的位置开始存储；
*   如果没有定义 #pragma pack(n)
*   sizeof 的最终结果必然是结构内部最大成员的整数倍，不够补齐；
*   结构内部各个成员的首地址必然是自身大小的整数倍；
* 对齐数== min(编译器默认的一个对齐数,该成员大小的较小值 ************************************************************
struct S1
{
    int i ;  //起始偏移0，sizeof(i)=4; 地址0、1、2、3分配给成员i
    char j ; //起始偏移4，sizeof(j)=1;
    int a ;  //sizeof(a)=4,内存对齐到8个字节，从偏移量为8处存放a;
    double b;//sizeof(b)=8,内存对齐到16个字节，再存放b,结构体总大小24;
};
//结构体成员的首地址必须是自身大小的整数倍
struct S2
{
    char j;//起始偏移0，sizeof(j)=1;
    float i;//sizeof(i)=4，内存对齐到4，起始偏移量为4,再存放i
    double b;//当前地址为8，是b大小的整数倍，无需对齐，直接存放成员b 8个字节
    int a;//sizeof(a)=4,内存对齐到16，再存放a,大小为20，但是又考虑double的整数倍，所以总大小24字节；
};
int main()
{
    printf("%d\n", sizeof(S1));
    printf("%d\n", sizeof(S2));

    return 0;
}

```

```
24
24
```

说明：结构体作为一种复合数据类型，其构成元素既可以是基本数据类型的变量，也可以是一些复合型类型数据。对此，编译器会自动进行成员变量的对齐以提高运算效率。对齐数为编译器默认的一个对齐数 与 该成员大小的较小值。默认情况下，按自然对齐条件分配空间。各个成员按照它们被声明的顺序在内存中顺序存储，第一个成员的地址和整个结构的地址相同，向结构体成员中size最大的成员对齐。
许多实际的计算机系统对基本类型数据在内存中存放的位置有限制，它们会要求这些数据的首地址的值是某个数k（通常它为4或8）的倍数，而这个k则被称为该数据类型的对齐模数。

全局变量和局部变量有什么区别？实怎么实现的？操作系统和编译器是怎么知道的？

全局变量是整个程序都可访问的变量，谁都可以访问，生存期在整个程序从运行到结束（在程序结束时所占内存释放）；
而局部变量存在于模块（子程序，函数）中，只有所在模块可以访问，其他模块不可直接访问，模块结束（函数调用完毕），局部变量消失，所占据的内存释放。
操作系统和编译器，可能是通过内存分配的位置来知道的，全局变量分配在全局数据段并且在程序开始运行的时候被加载.局部变量则分配在堆栈里面。

简述C、C++程序编译的内存分配情况

从静态存储区域分配：
内存在程序编译时就已经分配好，这块内存在程序的整个运行期间都存在。速度快、不容易出错， 因为有系统会善后。例如全局变量，static 变量，常量字符串等。

在栈上分配：
在执行函数时，函数内局部变量的存储单元都在栈上创建，函数执行结束时这些存储单元自动被释 放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。大小为2M。

从堆上分配：
即动态内存分配。程序在运行的时候用 malloc 或new 申请任意大小的内存，程序员自己负责在何 时用free 或delete 释放内存。动态内存的生存期由程序员决定，使用非常灵活。如果在堆上分配了空间，就有责任回收它，否则运行的程序会出现内存泄漏，另外频繁地分配和释放不同大小的堆空间将会产生 堆内碎块。

一个C、C++程序编译时内存分为5 大存储区：堆区、栈区、全局区、文字常量区、程序代码区。

简述strcpy、sprintf 与memcpy 的区别

操作对象不同，strcpy 的两个操作对象均为字符串，sprintf 的操作源对象可以是多种数据类型， 目的操作对象是字符串，memcpy 的两个对象就是两个任意可操作的内存地址，并不限于何种数据类型。
执行效率不同，memcpy 最高，strcpy 次之，sprintf 的效率最低。
实现功能不同，strcpy 主要实现字符串变量间的拷贝，sprintf 主要实现其他数据类型格式到字 符串的转化，memcpy 主要是内存块间的拷贝。
注意：strcpy、sprintf 与memcpy 都可以实现拷贝的功能，但是针对的对象不同，根据实际需求，来 选择合适的函数实现拷贝功能。

请解析((void ()( ))0)( )的含义

void (0)( )：是一个返回值为void，参数为空的函数指针0。
(void ()( ))0：把0转变成一个返回值为void，参数为空的函数指针。
(void ()( ))0：在上句的基础上加表示整个是一个返回值为void，无参数，并且起始地址为0的函数的名字。
((void (*)( ))0)( )：这就是上句的函数名所对应的函数的调用。

typedef 和define 有什么区别

用法不同：

typedef 用来定义一种数据类型的别名，增强程序的可读性。define 主要用来定义 常量，以及书写复杂使用频繁的宏。
执行时间不同：

typedef 是编译过程的一部分，有类型检查的功能。define 是宏定义，是预编译的部分，其发生在编译之前，只是简单的进行字符串的替换，不进行类型的检查。
作用域不同：

typedef 有作用域限定。define 不受作用域约束，只要是在define 声明后的引用 都是正确的。
对指针的操作不同：

typedef 和define 定义的指针时有很大的区别。
注意：typedef 定义是语句，因为句尾要加上分号。而define 不是语句，千万不能在句尾加分号。

指针常量与常量指针区别

指针常量是指定义了一个指针，这个指针的值只能在定义时初始化，其他地方不能改变。常量指针 是指定义了一个指针，这个指针指向一个只读的对象，不能通过常量指针来改变这个对象的值。 指针常量强调的是指针的不可改变性，而常量指针强调的是指针对其所指对象的不可改变性。

注意：无论是指针常量还是常量指针，其最大的用途就是作为函数的形式参数，保证实参在被调用 函数中的不可改变特性。

简述队列和栈的异同

队列和栈都是线性存储结构，但是两者的插入和删除数据的操作不同，队列是“先进先出”，栈是 “后进先出”。

注意：区别栈区和堆区。堆区的存取是“顺序随意”，而栈区是“后进先出”。栈由编译器自动分配释放 ，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。堆一般由程序员分配释放， 若程序员不释放，程序结束时可能由OS 回收。分配方式类似于链表。 它与本题中的堆和栈是两回事。堆栈只是一种数据结构，而堆区和栈区是程序的不同内存存储区域。

设置地址为0x67a9 的整型变量的值为0xaa66

```
int *ptr;
ptr = (int *)0x67a9;
*ptr = 0xaa66;
```

注意：这道题就是强制类型转换的典型例子，无论在什么平台地址长度和整型数据的长度是一样的， 即一个整型数据可以强制转换成地址指针类型，只要有意义即可。

编码实现字符串转化为数字

编码实现函数atoi()，设计一个程序，把一个字符串转化为一个整型数值。例如数字：“5486321 ”， 转化成字符：5486321。

```
/*****************************************************************************

Welcome to GDB Online.
  GDB online is an online compiler and debugger tool for C, C++, Python, PHP, Ruby,
  C#, OCaml, VB, Perl, Swift, Prolog, Javascript, Pascal, COBOL, HTML, CSS, JS
  Code, Compile, Run and Debug online from anywhere in world.

*****************************************************************************/
#include <stdio.h>
#include <math.h>

int myAtoi(const char * str)
{
   int num = 0; //保存转换后的数值
   int isNegative = 0; //记录字符串中是否有负号

   int n =0;
   const char *p = str;
   if(p == NULL) //判断指针的合法性
   {
      return -1;
   }
   while(*p++ != '\0') //计算数字符串度
   {
      n++;
   }
   p = str;
   if(p[0] == '-') //判断数组是否有负号
   {
      isNegative = 1;
   }

   char temp = '0';
   for(int i = 0 ; i < n; i++)
   {
      char temp = *p++;
       if(temp > '9' ||temp < '0') //滤除非数字字符

```

```
      {
         continue;
      }
      if(num !=0 || temp != '0') //滤除字符串开始的0 字符
      {
         temp -= 0x30; //将数字字符转换为数值
          num += temp *int( pow(10 , n - 1 -i) );
       }
   }
   if(isNegative) //如果字符串中有负号，将数值取反
   {
       return (0 - num);
   }
   else
   {
      return num; //返回转换后的数值
   }
}


int main()
{
   // printf("Hello World");

    char test[] = "12345";

    printf("%d\n", myAtoi(test));
    return 0;
}
```

C语言的结构体和C++的有什么区别

C语言的结构体是不能有函数成员的，而C++的类可以有。
C语言的结构体中数据成员是没有private、public和protected访问限定的。而C++的类的成员有这些访问限定。
C语言的结构体是没有继承关系的，而C++的类却有丰富的继承关系。
注意：虽然C的结构体和C++的类有很大的相似度，但是类是实现面向对象的基础。而结构体只可以简单地理解为类的前身。

简述指针常量与常量指针的区别

指针常量是指定义了一个指针，这个指针的值只能在定义时初始化，其他地方不能改变。常量指针是指定义了一个指针，这个指针指向一个只读的对象，不能通过常量指针来改变这个对象的值。
指针常量强调的是指针的不可改变性，而常量指针强调的是指针对其所指对象的不可改变性。
注意：无论是指针常量还是常量指针，其最大的用途就是作为函数的形式参数，保证实参在被调用函数中的不可改变特性。

如何避免“野指针”

指针变量声明时没有被初始化。解决办法：指针声明时初始化，可以是具体的地址值，也可让它指向NULL。
指针p被free或者delete之后，没有置为NULL。解决办法：指针指向的内存空间被释放后指针应该指向NULL。
指针操作超越了变量的作用范围。解决办法：在变量的作用域结束前释放掉变量的地址空间并且让指针指向NULL。

句柄和指针的区别和联系是什么?

句柄和指针其实是两个截然不同的概念。Windows系统用句柄标记系统资源，隐藏系统的信息。你只要知道有这个东西，然后去调用就行了，它是个32it的uint。指针则标记某个物理内存地址，两者是不同的概念。

new/delete与malloc/free的区别是什么

new能自动计算需要分配的内存空间，而malloc需要手工计算字节数。

```
int *p = new int[2];
int *q = (int )malloc(2sizeof(int));
```

new与delete直接带具体类型的指针，malloc和free返回void类型的指针。
new类型是安全的，而malloc不是。例如int *p = new float[2];就会报错；而int p = malloc(2sizeof(int))编译时编译器就无法指出错误来。
new一般分为两步：new操作和构造。new操作对应与malloc，但new操作可以重载，可以自定义内存分配策略，不做内存分配，甚至分配到非内存设备上，而malloc不行。

new调用构造函数，malloc不能；delete调用析构函数，而free不能。
malloc/free需要库文件stdlib.h的支持，new/delete则不需要！
注意：delete和free被调用后，内存不会立即回收，指针也不会指向空，delete或free仅仅是告诉操作系统，这一块内存被释放了，可以用作其他用途。但是由于没有重新对这块内存进行写操作，所以内存中的变量数值并没有发生变化，出现野指针的情况。因此，释放完内存后，应该讲该指针指向NULL。

说一说extern“C”

extern "C"的主要作用就是为了**能够正确实现C++代码调用其他C语言代码。**加上extern "C"后，会指示**编译器这部分代码按C语言（而不是C++）的方式进行编译。**由于C++支持函数重载，因此编译器编译函数的过程中会将函数的参数类型也加到编译后的代码中，而不仅仅是函数名；而C语言并不支持函数重载，因此编译C语言代码的函数时不会带上函数的参数类型，一般只包括函数名。

这个功能十分有用处，因为在C++出现以前，很多代码都是C语言写的，而且很底层的库也是C语言写的，**为了更好的支持原来的C代码和已经写好的C语言库，需要在C++中尽可能的支持C**，而extern "C"就是其中的一个策略。

C++代码调用C语言代码在C++的头文件中使用在多个人协同开发时，可能有的人比较擅长C语言，而有的人擅长C++，这样的情况下也会有用到。

请你来说一下C++中struct和class的区别

在C++中，class和struct做类型定义是只有两点区别：

默认继承权限不同，class继承默认是private继承，而struct默认是public继承
class还可用于定义模板参数，像typename，但是关键字struct不能同于定义模板参数 C++保留struct关键字，原因
保证与C语言的向下兼容性，C++必须提供一个struct
C++中的struct定义必须百分百地保证与C语言中的struct的向下兼容性，把C++中的最基本的对象单元规定为class而不是struct，就是为了避免各种兼容性要求的限制
对struct定义的扩展使C语言的代码能够更容易的被移植到C++中

C++类内可以定义引用数据成员吗?

可以，必须通过成员函数初始化列表初始化。

C++中类成员的访问权限

C++通过 public、protected、private 三个关键字来控制成员变量和成员函数的访问权限，它们分别表示公有的、受保护的、私有的，被称为成员访问限定符。在类的内部（定义类的代码内部），无论成员被声明为 public、protected 还是 private，都是可以互相访问的，没有访问权限的限制。在类的外部（定义类的代码之外），只能通过对象访问成员，并且通过对象只能访问 public 属性的成员，不能访问 private、protected 属性的成员

什么是右值引用，跟左值又有什么区别?

左值和右值的概念：

左值：能取地址，或者具名对象，表达式结束后依然存在的持久对象；
右值：不能取地址，匿名对象，表达式结束后就不再存在的临时对象； 区别：
左值能寻址，右值不能；
左值能赋值，右值不能；
左值可变，右值不能（仅对基础类型适用，用户自定义类型右值引用可以通过成员函数改变）；

面向对象的三大特征

封装性：将客观事物抽象成类，每个类对自身的数据和方法实行 protection （private， protected， public ）。
继承性：广义的继承有三种实现形式：实现继承（使用基类的属性和方法而无需额外编码的能力)、可 视继承(子窗体使用父窗体的外观和实现代码)、接口继承(仅使用属性和方法,实现滞后到子类实现)。
多态性：是将父类对象设置成为和一个或更多它的子对象相等的技术。用子类对象给父类对象赋值 之后，父类对象就可以根据当前赋值给它的子对象的特性以不同的方式运作。

说一说c++中四种cast转换

C++中四种类型转换是：static_cast, dynamic_cast, const_cast, reinterpret_cast

1、const_cast

用于将const变量转为非const

2、static_cast

用于各种隐式转换，比如非const转const，void*转指针等, static_cast能用于多态向上转化，如果向下转能成功但是不安全，结果未知；

3、dynamic_cast*

用于动态类型转换。只能用于含有虚函数的类，用于类层次间的向上和向下转化。只能转指针或引用。向下转化时，如果是非法的***对于指针返回NULL，对于引用抛异常***。要深入了解内部转换的原理。

向上转换：指的是子类向基类的转换
向下转换：指的是基类向子类的转换
它通过判断在执行到该语句的时候变量的运行时类型和要转换的类型是否相同来判断是否能够进行向下转换。

4、reinterpret_cast

几乎什么都可以转，比如将int转指针，可能会出问题，尽量少用；

5、为什么不使用C的强制转换?

C的强制转换表面上看起来功能强大什么都能转，但是转化不够明确，不能进行错误检查，容易出错。

C++的空类有哪些成员函数

缺省构造函数。
缺省拷贝构造函数。
缺省析构函数。
缺省赋值运算符。
缺省取址运算符。
缺省取址运算符 const 。
注意：有些书上只是简单的介绍了前四个函数。没有提及后面这两个函数。但后面这两个函数也是 空类的默认函数。另外需要注意的是，只有当实际使用这些函数的时候，编译器才会去定义它们。

对c++中的smart pointer四个智能指针：shared_ptr,unique_ptr,weak_ptr,auto_ptr的理解

C++里面的四个智能指针: auto_ptr, shared_ptr, weak_ptr, unique_ptr 其中后三个是c++11支持，并且第一个已经被11弃用。

智能指针的作用是管理一个指针，因为存在以下这种情况：申请的空间在函数结束时忘记释放，造成内存泄漏。使用智能指针可以很大程度上的避免这个问题，因为智能指针就是一个类，当超出了类的作用域是，类会自动调用析构函数，析构函数会自动释放资源。所以智能指针的作用原理就是在函数结束时自动释放内存空间，不需要手动释放内存空间。

auto_ptr（c++98的方案，cpp11已经抛弃）
采用所有权模式。

```
auto_ptr< string> p1 (new string ("I reigned lonely as a cloud.”));
auto_ptr p2;
p2 = p1; //auto_ptr不会报错.
```

此时不会报错，p2剥夺了p1的所有权，但是当程序运行时访问p1将会报错。所以auto_ptr的缺点是：存在潜在的内存崩溃问题！

unique_ptr（替换auto_ptr）
unique_ptr实现独占式拥有或严格拥有概念，保证同一时间内只有一个智能指针可以指向该对象。它对于避免资源泄露(例如“以new创建对象后因为发生异常而忘记调用delete”)特别有用。

采用所有权模式。

```
unique_ptr p3 (new string (“auto”)); //#4
unique_ptr p4;  //#5
p4 = p3;//此时会报错！！
```

编译器认为p4=p3非法，避免了p3不再指向有效数据的问题。因此，unique_ptr比auto_ptr更安全。

另外unique_ptr还有更聪明的地方：当程序试图将一个 unique_ptr 赋值给另一个时，如果源 unique_ptr 是个临时右值，编译器允许这么做；如果源 unique_ptr 将存在一段时间，编译器将禁止这么做，比如：

```
unique_ptr pu1(new string (“hello world”));
unique_ptr pu2;
pu2 = pu1; // #1 not allowed


```

```
unique_ptr pu3;
pu3 = unique_ptr(new string (“You”)); // #2 allowed
```

其中#1留下悬挂的unique_ptr(pu1)，这可能导致危害。而#2不会留下悬挂的unique_ptr，因为它调用 unique_ptr 的构造函数，该构造函数创建的临时对象在其所有权让给 pu3 后就会被销毁。这种随情况而已的行为表明，unique_ptr 优于允许两种赋值的auto_ptr 。

注：如果确实想执行类似与#1的操作，要安全的重用这种指针，可给它赋新值。C++有一个标准库函数std::move()，让你能够将一个unique_ptr赋给另一个。例如：

```
unique_ptr ps1, ps2;
ps1 = demo(“hello”);
ps2 = move(ps1);
ps1 = demo(“alexia”);
cout << *ps2 << *ps1 << endl;
shared_ptr
```

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。从名字share就可以看出了资源可以被多个指针共享，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，计数减一。当计数等于0时，资源会被释放。

shared_ptr 是为了解决 auto_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。

成员函数：

use_count 返回引用计数的个数

unique 返回是否是独占所有权(use_count 为 1)

swap 交换两个 shared_ptr 对象(即交换所拥有的对象)

reset 放弃内部对象的所有权或拥有对象的变更, 会引起原有对象的引用计数的减少

get 返回内部对象(指针), 由于已经重载了()方法, 因此和直接使用对象是一样的.如 shared_ptrsp(new int(1)); sp 与 sp.get()是等价的

weak_ptr
weak_ptr 是一种不控制对象生命周期的智能指针, 它指向一个 shared_ptr 管理的对象. 进行该对象的内存管理的是那个强引用的 shared_ptr. weak_ptr只是提供了对管理对象的一个访问手段。weak_ptr 设计的目的是为配合 shared_ptr 而引入的一种智能指针来协助 shared_ptr 工作, 它只可以从一个 shared_ptr 或另一个 weak_ptr 对象构造, 它的构造和析构不会引起引用记数的增加或减少。weak_ptr是用来解决shared_ptr相互引用时的死锁问题,如果说两个shared_ptr相互引用,那么这两个指针的引用计数永远不可能下降为0,资源永远不会释放。它是对对象的一种弱引用，不会增加对象的引用计数，和shared_ptr之间可以相互转化，shared_ptr可以直接赋值给它，它可以通过调用lock函数来获得shared_ptr。

```
class B;
class A
{
public:
shared_ptr<B> pb_;
~A()
{
     cout<<"A delete
";
}
};
class B
{
public:
shared_ptr<A> pa_;
~B()
{
    cout<<"B delete
";
}
};
void fun()
{

```

```
    shared_ptr<B> pb(new B());
    shared_ptr<A> pa(new A());
    pb->pa_ = pa;
    pa->pb_ = pb;
    cout<<pb.use_count()<<endl;
    cout<<pa.use_count()<<endl;
}
int main()
{
    fun();
    return 0;
}
```

可以看到fun函数中pa ，pb之间互相引用，两个资源的引用计数为2，当要跳出函数时，智能指针pa，pb析构时两个资源引用计数会减一，但是两者引用计数还是为1，导致跳出函数时资源没有被释放（A B的析构函数没有被调用），如果把其中一个改为weak_ptr就可以了，我们把类A里面的shared_ptr pb_; 改为weak_ptr pb_; 运行结果如下，这样的话，资源B的引用开始就只有1，当pb析构时，B的计数变为0，B得到释放，B释放的同时也会使A的计数减一，同时pa析构时使A的计数减一，那么A的计数为0，A得到释放。

注意：不能通过weak_ptr直接访问对象的方法，比如B对象中有一个方法print(),我们不能这样访问，pa->pb_->print(); 英文pb_是一个weak_ptr，应该先把它转化为shared_ptr,如：shared_ptr p = pa->pb_.lock(); p->print();

说说强制类型转换运算符

static_cast
用于非多态类型的转换
不执行运行时类型检查（转换安全性不如 dynamic_cast）
通常用于转换数值数据类型（如 float -> int）
可以在整个类层次结构中移动指针，子类转化为父类安全（向上转换），父类转化为子类不安全（因为子类可能有不在父类的字段或方法）
dynamic_cast
用于多态类型的转换
执行行运行时类型检查
只适用于指针或引用
对不明确的指针的转换将失败（返回 nullptr），但不引发异常
可以在整个类层次结构中移动指针，包括向上转换、向下转换
const_cast
用于删除 const、volatile 和 __unaligned 特性（如将 const int 类型转换为 int 类型 ） reinterpret_cast
用于位的简单重新解释
滥用 reinterpret_cast 运算符可能很容易带来风险。除非所需转换本身是低级别的，否则应- 使用其他强制转换运算符之一。
允许将任何指针转换为任何其他指针类型（如 char* 到 int* 或 One_class* 到 Unrelated_class* 之类的转换，但其本身并不安全）
也允许将任何整数类型转换为任何指针类型以及反向转换。
reinterpret_cast 运算符不能丢掉 const、volatile 或 __unaligned 特性。
reinterpret_cast 的一个实际用途是在哈希函数中，即，通过让两个不同的值几乎不以相同的索引结尾的方式将值映射到索引。
bad_cast
由于强制转换为引用类型失败，dynamic_cast 运算符引发 bad_cast 异常。
bad_cast 使用

```
try {
    Circle& ref_circle = dynamic_cast<Circle&>(ref_shape);
}
catch (bad_cast b) {
    cout << "Caught: " << b.what();
}
```

谈谈你对拷贝构造函数和赋值运算符的认识

拷贝构造函数和赋值运算符重载有以下两个不同之处：

拷贝构造函数生成新的类对象，而赋值运算符不能。
由于拷贝构造函数是直接构造一个新的类对象，所以在初始化这个对象之前不用检验源对象 是否和新建对象相同。而赋值运算符则需要这个操作，另外赋值运算中如果原来的对象中有内存分配要先把内存释放掉。
注意：当有类中有指针类型的成员变量时，一定要重写拷贝构造函数和赋值运算符，不要使用默认 的。

在C++中，使用malloc申请的内存能否通过delete释放？使用new申请的内存能否用free？

不能，malloc /free主要为了兼容C，new和delete 完全可以取代malloc /free的。malloc /free的操作对象都是必须明确大小的。而且不能用在动态类上。new 和delete会自动进行类型检查和大小，malloc/free不能执行构造函数与析构函数，所以动态对象它是不行的。当然从理论上说使用malloc申请的内存是可以通过delete释放的。不过一般不这样写的。而且也不能保证每个C++的运行时都能正常。

用C++设计一个不能被继承的类

```
template <typename T> class A
{
   friend T;
    private:
     A() {}
    ~A() {}
};
class B : virtual public A<B>
{
   public:
    B() {}
   ~B() {}
};
class C : virtual public B
{
   public:
     C() {}
    ~C() {}
};
void main( void )
{
    B b;
    //C c;
    return;
}
```

注意：构造函数是继承实现的关键，每次子类对象构造时，首先调用的是父类的构造函数，然后才 是自己的。

C++自己实现一个String类

```
- #include <iostream>
  #include <cstring>

using namespace std;

class String{
public:
    // 默认构造函数
    String(const char *str = nullptr);
    // 拷贝构造函数
    String(const String &str);
    // 析构函数
    ~String();
    // 字符串赋值函数
    String& operator=(const String &str);

private:
    char *m_data;
    int m_size;
};

// 构造函数
String::String(const char *str)
{
    if(str == nullptr)  // 加分点：对m_data加NULL 判断
    {
        m_data = new char[1];   // 得分点：对空字符串自动申请存放结束标志'\0'的
        m_data[0] = '\0';
        m_size = 0;
    }
    else
    {

```

```
        m_size = strlen(str);
        m_data = new char[m_size + 1];
        strcpy(m_data, str);
    }
}

// 拷贝构造函数
String::String(const String &str)   // 得分点：输入参数为const型
{
    m_size = str.m_size;
    m_data = new char[m_size + 1];  //加分点：对m_data加NULL 判断
    strcpy(m_data, str.m_data);
}

// 析构函数
String::~String()
{
    delete[] m_data;
}

// 字符串赋值函数
String& String::operator=(const String &str)  // 得分点：输入参数为const
{
    if(this == &str)    //得分点：检查自赋值
        return *this;

delete[] m_data;    //得分点：释放原有的内存资源
m_size = strlen(str.m_data);
m_data = new char[m_size + 1];  //加分点：对m_data加NULL 判断
strcpy(m_data, str.m_data);
return *this;       //得分点：返回本对象的引用

}
```

访问基类的私有虚函数

写出以下程序的输出结果：

```
#include <iostream.h>
class A
{
   virtual void g()
   {
      cout << "A::g" << endl;
   }
  private:
   virtual void f()
   {
      cout << "A::f" << endl;
   }
};
class B : public A
{
   void g()
   {
      cout << "B::g" << endl;
   }
   virtual void h()
   {
      cout << "B::h" << endl;
   }
};
typedef void( *Fun )( void );
void main()
{
   B b;
   Fun pFun;
   for(int i = 0 ; i < 3; i++)
   {
      pFun = ( Fun )*( ( int* ) * ( int* )( &b ) + i );
      pFun();
```

```
    }
}
```

输出结果：

```
B::g
A::f
B::h
```

注意：考察了面试者对虚函数的理解程度。一个对虚函数不了解的人很难正确的做出本题。 在学习面向对象的多态性时一定要深刻理解虚函数表的工作原理。

对虚函数和多态的理解

多态的实现主要分为静态多态和动态多态，静态多态主要是重载，在编译的时候就已经确定；动态多态是用虚函数机制实现的，在运行期间动态绑定。举个例子：一个父类类型的指针指向一个子类对象时候，使用父类的指针去调用子类中重写了的父类中的虚函数的时候，会调用子类重写过后的函数，在父类中声明为加了virtual关键字的函数，在子类中重写时候不需要加virtual也是虚函数。

虚函数的实现：在有虚函数的类中，类的最开始部分是一个虚函数表的指针，这个指针指向一个虚函数表，表中放了虚函数的地址，实际的虚函数在代码段(.text)中。当子类继承了父类的时候也会继承其虚函数表，当子类重写父类中虚函数时候，会将其继承到的虚函数表中的地址替换为重新写的函数地址。使用了虚函数，会增加访问内存开销，降低效率。

简述类成员函数的重写、重载和隐藏的区别

（1）重写和重载主要有以下几点不同。

范围的区别：被重写的和重写的函数在两个类中，而重载和被重载的函数在同一个类中。
参数的区别：被重写函数和重写函数的参数列表一定相同，而被重载函数和重载函数的参数列表一 定不同。
virtual 的区别：重写的基类中被重写的函数必须要有virtual 修饰，而重载函数和被重载函数可以被 virtual 修饰，也可以没有。

（2）隐藏和重写、重载有以下几点不同。

与重载的范围不同：和重写一样，隐藏函数和被隐藏函数不在同一个类中。
参数的区别：隐藏函数和被隐藏的函数的参数列表可以相同，也可不同，但是函数名肯定要相同。 当参数不相同时，无论基类中的参数是否被virtual 修饰，基类的函数都是被隐藏，而不是被重写。
注意：虽然重载和覆盖都是实现多态的基础，但是两者实现的技术完全不相同，达到的目的也是完 全不同的，覆盖是动态态绑定的多态，而重载是静态绑定的多态。

链表和数组有什么区别

存储形式：

数组是一块连续的空间，声明时就要确定长度。链表是一块可不连续的动态空间， 长度可变，每个结点要保存相邻结点指针。

数据查找：

数组的线性查找速度快，查找操作直接使用偏移地址。链表需要按顺序检索结点， 效率低。
数据插入或删除：

链表可以快速插入和删除结点，而数组则可能需要大量数据移动。

越界问题：

链表不存在越界问题，数组有越界问题。

注意：

在选择数组或链表数据结构时，一定要根据实际需要进行选择。数组便于查询，链表便于插 入删除。数组节省空间但是长度固定，链表虽然变长但是占了更多的存储空间。

用两个栈实现一个队列的功能

```
typedef struct node
{
   int data;
   node *next;
}node,*LinkStack;

```

```
//创建空栈：
LinkStack CreateNULLStack( LinkStack &S)
{
 S = (LinkStack)malloc( sizeof( node ) ); // 申请新结点
 if( NULL == S)
 {
  printf("Fail to malloc a new node.\n");

  return NULL;
 }
 S->data = 0; //初始化新结点
 S->next = NULL;

 return S;
}

//栈的插入函数：
LinkStack Push( LinkStack &S, int data)
{
 if( NULL == S) //检验栈
 {
  printf("There no node in stack!");
  return NULL;
 }

 LinkStack p = NULL;
 p = (LinkStack)malloc( sizeof( node ) ); // 申请新结点

 if( NULL == p)
 {
  printf("Fail to malloc a new node.\n");
  return S;
 }
 if( NULL == S->next)
 {
  p->next = NULL;
 }
 else
 {
  p->next = S->next;
 }
 p->data = data; //初始化新结点
 S->next = p; //插入新结点
 return S;
}

//出栈函数：
node Pop( LinkStack &S)
{
 node temp;
 temp.data = 0;
 temp.next = NULL;

 if( NULL == S) //检验栈
 {
  printf("There no node in stack!");
  return temp;
 }
 temp = *S;

 if( S->next == NULL )
 {
  printf("The stack is NULL,can't pop!\n");
  return temp;
 }
 LinkStack p = S ->next; //节点出栈

 S->next = S->next->next;
 temp = *p;
 free( p );
 p = NULL;
```

```
 return temp;
}

//双栈实现队列的入队函数：
LinkStack StackToQueuPush( LinkStack &S, int data)
{
 node n;
 LinkStack S1 = NULL;
 CreateNULLStack( S1 ); //创建空栈

 while( NULL != S->next ) //S 出栈入S1
 {
  n = Pop( S );
  Push( S1, n.data );
 }
 Push( S1, data ); //新结点入栈

 while( NULL != S1->next ) //S1 出栈入S
 {
  n = Pop( S1 );
  Push( S, n.data );
 }
 return S;
}
```

注意：用两个栈能够实现一个队列的功能，那用两个队列能否实现一个队列的功能呢？结果是否定 的，因为栈是先进后出，将两个栈连在一起，就是先进先出。而队列是现先进先出，无论多少个连在一 起都是先进先出，而无法实现先进后出。

共享数据的保护

常引用：

使所引用的形参不能被更新
void display(const double& a);

常对象：

在生存期内不能被更新，但必须被初始化
A const a(3,4);

常成员函数：

不能修改对象中数据成员，也不能调用类中没有被const 修饰的成员函数（常对象唯一的对外接口）.如果声明了一个常对象，则该对象只能调用他的常函数！->可以用于对重载函数的区分;
void print();
void print() const;

extern int a:使其他文件也能访问该变量

声明一个函数或定义函数时，冠以static的话，函数的作用域就被限制在了当前编译单元，当前编译单元内也必须包含函数的定义，也只在其编译单元可见，其他单元不能调用这个函数(每一个cpp 文件就是一个编译单元)。

程序内存分配方式以及它们的区别

内存分配大致上可以分成5块：

栈区（stack）

栈，就是那些由编译器在需要时分配，在不需要的时候自动清除的变量的存储区。里面的变量通常是局部变量、函数参数等。（由编译器管理）

堆区（heap）

一般由程序员分配、释放，若程序员不是放，程序结束时可能由系统回收。注意，它与数据结构中的堆是两回事，分配方式类似于链表。

全局区（静态区）（static）

全局变量和静态变量被分配到同
一块内存中。程序结束后由系统释放。

常量存储区

常量字符串就是放在这里的，不允许修改，程序结束后由系统释放。

程序代码区

存放函数体的二进制代码。

explicit

函数声明时加上explicit可以阻止函数参数被隐式转换。

```
Class A
{
   explicit A(int a);
}

Void main()
{
   A a1=12;   //不加explicit时会被隐式转换位 A a1=A(12);加了此时编译器会报错。
}
```

被声明为explicit的构造函数通常比non-explicit 函数更受欢迎。

mutable关键字

mutalbe的中文意思是“可变的，易变的”，跟constant（既C++中的const）是反义词。在C++中，mutable也是为了突破const的限制而设置的。被mutable修饰的变量(mutable只能由于修饰类的非静态数据成员)，将永远处于可变的状态，即使在一个const函数中。

我们知道，假如类的成员函数不会改变对象的状态，那么这个成员函数一般会声明为const。但是，有些时候，我们需要在const的函数里面修改一些跟类状态无关的数据成员，那么这个数据成员就应该被mutalbe来修饰。（使用mutable修饰的数据成员可以被const成员函数修改）。

用const修饰函数的返回值

如果给以“指针传递”方式的函数返回值加const修饰，那么函数返回值（即指针）的内容不能被修改，该返回值只能被赋给加const修饰的同类型指针。例如函数

```
Const char * GetString(void);
// 如下语句将出现编译错误：
char*str = GetString();
// 正确的用法是
Const char *str =GetString();
```

宏、const和enum

1. #define 不被视为语言的一部分。对于单纯常量，最好用const对象或者enum替换#define。
2. 对于类似函数的宏，尽量使用内联函数替换掉#define。

static的生存期

C++中的static对象是指存储区不属于stack和heap、"寿命"从被构造出来直至程序结束为止的对象。这些对象包括全局对象，定义于namespace作用域的对象，在class、function以及file作用域中被声明为static的对象。其中，函数内的static对象称为local static对象，而其它static对象称为non-local static对象。

这两者在何时被初始化(构造)这个问题上存在细微的差别：

对于local static对象，在其所属的函数被调用之前，该对象并不存在，即只有在第一次调用对应函数时，local static对象才被构造出来。
而对于non-local static对象，在main()函数开始前就已经被构造出来，并在main()函数结束后被析构。

建议：

1.对内置对象进行手工初始化，因为C++不保证初始化它们。
2.构造函数最好使用成员初值列，而不要在构造函数本体中使用赋值操作。初值列中列出的成员变量，其排序次序应该和它们在

class中的声明次序相同(初始化顺序与声明变量顺序一致)。
3.为免除“跨编译单元的初始化次序问题”，尽量以local static对象替换non-local static对象。

全局变量和static变量的区别

全局变量本身就是静态存储方式， 静态全局变量当然也是静态存储方式。 这两者在存储方式上并无不同。这两者的区别在于非静态全局变量的作用域是整个源程序， 当一个源程序由多个源文件组成时，非静态的全局变量在各个源文件中都是有效的。 而静态全局变量则限制了其作用域， 即只在定义该变量的源文件内有效， 在同一源程序的其它源文件中不能使用它。由于静态全局变量的作用域局限于一个源文件内，只能为该源文件内的函数公用， 因此可以避免在其它源文件中引起错误。

为什么栈要比堆速度要快

首先, 栈是本着LIFO原则的存储机制, 对栈数据的定位相对比较快速, 而堆则是随机分配的空间, 处理的数据比较多, 无论如何, 至少要两次定位.
其次, 栈是由CPU提供指令支持的, 在指令的处理速度上, 对栈数据进行处理的速度自然要优于由操作系统支持的堆数据.
再者, 栈是在一级缓存中做缓存的, 而堆则是在二级缓存中, 两者在硬件性能上差异巨大.
最后, 各语言对栈的优化支持要优于对堆的支持, 比如swift语言中, 三个字及以内的struct结构, 可以在栈中内联, 从而达到更快的处理速度.

c++ 析构函数调用时间

1. 对象生命周期结束，被销毁时
2. delete指向对象的指针时，或delete指向对象的基类类型指针，而其基类虚构函数是虚函数时
3. 对象i是对象o的成员，o的析构函数被调用时，对象i的析构函数也被调用

静态绑定 动态绑定 （也叫动态连编，静态连编）

如果父类中存在有虚函数，那么编译器便会为之生成虚表（属于类）与虚指针（属于某个对象），在程序运行时，根据虚指针的指向，来决定调用哪个虚函数，这称之与动态绑定，与之相对的是静态绑定，静态绑定在编译期就决定了。

1. class和template都支持接口与多态；
2. 对classes而言，接口是显式的，以函数签名为中心。多态则是通过virtual函数发生于运行期；
3. 对template参数而言，接口是隐式的，奠基于有效表达式。多态则是通过template具现化和函数重载解析发生于编译期。
   泛型
4. 泛型是通过参数化类型来实现在同一份代码上操作多种数据类型。利用“参数化类型”将类型抽象化，从而实现灵活的复用。

C语言的指针和c++的引用有什么区别?

- 指针有自己的一块空间，指针是一个变量，只不过这个变量存储的是一个地址，指向内存的一个存储单元，即指针是一个实体。而引用只是一个别名；
- 使用sizeof看一个指针的大小是4，而引用则是被引用对象的大小；
- 指针可以被初始化为NULL，而引用必须被初始化且必须是一个已有对象 的引用；
- 作为参数传递时，指针需要被解引用才可以对对象进行操作，而直接对引 用的修改都会改变引用所指向的对象；
- 可以有const指针，但是没有const引用；
- 指针在使用中可以指向其它对象，但是引用只能是一个对象的引用，不能 被改变；
- 指针可以有多级指针（**p），而引用至于一级；
- 指针和引用使用++运算符的意义不一样；
- 如果返回动态内存分配的对象或者内存，必须使用指针，引用可能引起内存泄露。

请你说说C语言是怎么进行函数调用的

每一个函数调用都会分配函数栈，在栈内进行函数执行过程。调用前，先把返回地址压栈，然后把当前函数的esp指针压栈。（ESP（Extended Stack Pointer）为扩展栈指针寄存器，是指针寄存器的一种，用于存放函数栈顶指针）

C语言参数压栈顺序？：从右到左

C++中拷贝赋值函数的形参能否进行值传递?

不能。如果是这种情况下，调用拷贝构造函数的时候，首先要将实参传递给形参，这个传递的时候又要调用拷贝构造函数(aa = ex.aa; //此处调用拷贝构造函数)。。如此循环，无法完成拷贝，栈也会满。

include头文件的顺序以及双引号”“和尖括号<>的区别

编译器预处理阶段查找头文件的路径不一样

- 使用双引号包含的头文件，查找头文件路径的顺序为：
  当前头文件目录
  编译器设置的头文件路径（编译器可使用-I显式指定搜索路径）
  系统变量CPLUS_INCLUDE_PATH/C_INCLUDE_PATH指定的头文件路径
- 对于使用尖括号包含的头文件，查找头文件的路径顺序为：
  编译器设置的头文件路径（编译器可使用-I显式指定搜索路径）
  系统变量CPLUS_INCLUDE_PATH/C_INCLUDE_PATH指定的头文件路径

一个C++源文件从文本到可执行文件经历的过程

对于C/C++编写的程序，从源代码到可执行文件，一般经过下面四个步骤：

- 预编译，预编译的时候做一些简单的文本替换，比如宏替换，而不进行语法的检查；
- 编译，在编译阶段，编译器将检查一些语法错误，但是，如果使用的函数事先没有定义这种情况，不再这一阶段检查，编译后，得到.s文件
- 汇编，将C/C++代码变为汇编代码，得到.o或者.obj文件
- 链接，将所用到的外部文件链接在一起，在这一阶段，就会检查使用的函数有没有定义
- 链接过后，形成可执行文件.exe
  详细请参阅：一个C++源文件从文本到可执行文件经历的过程

内存泄漏原因和判断方法

内存泄漏通常是因为调用了malloc/new等内存申请操作，但是缺少了对应的free/delete。
为了判断内存是否泄漏，我们一方面可以使用Linux环境下的内存泄漏检查工具Valgrind，另一方面我们写代码的时候，可以添加内存申请和释放的统计功能，统计当前申请和释放的内存是否一致，以此来判断内存是否有泄漏。

内存泄漏分类：

- 堆内存泄漏（heap leak）。堆内存值得是程序运行过程中根据需要分配通过malloc\realloc\new等从堆中分配的一块内存，再完成之后必须要通过调用对应的free或者delete删除。
  如果程序的设计的错误导致这部分内存没有被释放，那么此后这块内存将不会被使用，就会产生Heap Leak。
- 系统资源泄露（Resource Leak）。主要指程序使用系统分配的资源比如 Bitmap，handle，SOCKET等没有使用相应的函数释放掉，导致系统资源的浪费，严重可导致系统效能降低，系统运行不稳定。
- 没有将基类的析构函数定义为虚函数。当基类指针指向子类对象时，如果基类的析构函数不是virtual，那么子类的析构函数将不会被调用，子类的资源没有正确的释放，从而造成内存泄漏。

段错误的产生原因

段错误是什么

一句话来说，段错误是指访问的内存超出了系统给这个程序所设定的内存空间，例如访问了不存在的内存地址、访问了系统保护的内存地址、访问了只读的内存地址等等情况。这里贴一个对于“段错误”的准确定义。

段错误产生的原因

1. 访问不存在的内存地址
2. 访问系统保护的内存地址
3. 访问只读的内存地址
4. 栈溢出

   详细请参阅：Linux环境下段错误的产生原因及调试方法小结

C++ 函数调用过程

总结起来整个过程就三步：
1）根据调用的函数名找到函数入口；

2）在栈中审请调用函数中的参数及函数体内定义的变量的内存空间
3）函数执行完后，释放函数在栈中的审请的参数和变量的空间，最后返回值（如果有的话）
详细请查阅：[函数调用过程 / C/C++函数调用过程分析(https://www.cnblogs.com/biyeymyhjob/archive/2012/07/20/2601204.html)

如何调试c++ 多线程程序?

1. 打印日志，日志中加上线程ID；（简单粗暴）
   gdb有thread相关命令，如infothread（简写成infoth）显示线程消息，bxxthreadyy可以
2. 对某个thread设置断点，threadxx（简写成thrxx）切换到某个thread。再配合frame（简写f）相关的命令（比如up，down在不同frame间跳转），基本可以处理若干个不同的线程间的debug……
   详细请查阅：C++(vs)多线程调试 （转）

面向对象和面向过程的区别

- 面向对象方法中，把数据和数据操作放在一起，组成对象；对同类的对象抽象出其共性组成类；类通过简单的接口与外界发生联系，对象和对象之间通过消息进行通信。**面向对象的三大特性是"封装、“多态”、“继承”，五大原则是"单一职责原则"、“开放封闭原则”、“里氏替换原则”、“依赖倒置原则”、“接口分离原则”。**
- 而面向过程方法是**以过程为中心的开发方法**，它自顶向下顺序进行， **程序结构按照功能划分成若干个基本模块，这些模块形成树状结构。**

  （过程）优点：

  性能比面向对象高，因为类调用时需要实例化，开销比较大，比较消耗源;比如嵌入式开发、Linux/Unix等一般采用面向过程开发，性能是最重要的因素。缺点：没有面向对象易维护、易复用、易扩展。

  （对象）优点：

  易维护、易复用、易扩展，由于面向对象有封装、继承、多态性的特性，可以设计出低耦合的系统。缺点：性能比面向过程低。

关于引用赋值的多态：

```
Class B;
Class D : public B;

B& b;
D& d;
B& b1 = d ;  //父类可以作为子类的引用，此时b1表现和指针形式一致（会调用B的非虚函数）
D& d1 = b;  //错误，不能将子类作为父类的引用

```

模板的声明和实现不能分开的原因

- 链接的时候，需要实例化模板，这时候就需要找模板的具体实现了。假设在main函数中调用了一个模板函数，这时候就需要去实例化该类型的模板。注意main函数里面只包含了.h文件，也就是只有模板的声明，没有具体实现。就会报错。
- 而模板的实现.cpp里面，虽然有模板的具体实现，但是没有谁在该.cpp里面使用一个模板函数，就不会生成一个具体化的实例
  详细请参阅：C++ 模板类的声明与实现分离问题 / C++ 模板类的声明与实现分离问题（模板实例化）

C++类中引用成员和常量成员的初始化（初始化列表）

如果一个类是这样定义的：

```
Class A
{
     public:
          A(int pram1, int pram2, int pram3);
     privite:
          int a;
          int &b;
          const int c;
}
```

假如在构造函数中对三个私有变量进行赋值则通常会这样写：

```
A::A(int pram1, int pram2, int pram3)
{
     a=pram1;
     b=pram2;
     c=pram3;
}
```

但是，这样是编译不过的。因为常量和引用初始化必须赋值。所以上面的构造函数的写法只是简单的赋值，并不是初始化。

正确写法应该是：

```
A::A(int pram1, int pram2, int pram3):b(pram2),c(pram3)
{
     a=pram1;
}
```

采用初始化列表实现了对常量和引用的初始化。采用括号赋值的方法，括号赋值只能用在变量的初始化而不能用在定义之后的赋值。

凡是有引用类型的成员变量或者常量类型的变量的类，不能有缺省构造函数。默认构造函数没有对引用成员提供默认的初始化机制，也因此造成引用未初始化的编译错误。并且必须使用初始化列表进行初始化const对象、引用对象。

memset为int型数组初始化问题

头文件：#include <string.h>
memset() 函数用来将指定内存的前n个字节设置为特定的值，其原型为：

```
void * memset( void * ptr, int value, size_t num );
```

参数说明：

- ptr 为要操作的内存的指针。
- value 为要设置的值。你既可以向 value 传递 int 类型的值，也可以传递 char 类型的值，int 和 char 可以根据 ASCII 码相互转换。
- num 为 ptr 的前 num 个字节，size_t 就是unsigned int。

> 【函数说明】memset() 会将 ptr 所指的内存区域的前 num 个字节的值都设置为 value，然后返回指向 ptr 的指针。

无法下面这样初始化，这样的结果是a被赋值成168430090，168430090.。。。。。。。。。

```
int a[10];
memset(a, 1, sizeof(a));
```

这是因为int由4个字节(说)表示，并且不能得到数组a中整数的期望值。
但我经常看到程序员使用memset将int数组元素设置为0或-1。其他值不行！

```
int a[10];
int b[10];
memset(a, 0, sizeof(a));
memset(b, -1, sizeof(b));
//假设a为int型数组：
memset(a,0x7f,sizeof(a));
//a数组每个空间将被初始化为0x7f7f7f7f,原因是C函数传参过程中的指针降级，导致sizeof(a)，返回的是一个 something*指针类型大
memset(a,0xaf,sizeof(a));
//a数组每个空间将被初始化为0xafafafaf
```

编译器对 inline 函数的处理步骤

- 将 inline 函数体复制到 inline 函数调用点处；
- 为所用 inline 函数中的局部变量分配内存空间；
- 将 inline 函数的的输入参数和返回值映射到调用方法的局部变量空间中；
- 如果 inline 函数有多个返回点，将其转变为 inline 函数代码块末尾的分支（使用 GOTO）

优缺点

优点

- 内联函数同宏函数一样将在被调用处进行代码展开，省去了参数压栈、栈帧开辟与回收，结果返回等，从而提高程序运行速度。
- 内联函数相比宏函数来说，在代码展开时，会做安全检查或自动类型转换（同普通函数），而宏定义则不会。
- 在类中声明同时定义的成员函数，自动转化为内联函数，因此内联函数可以访问类的成员变量，宏定义则不能。
- 内联函数在运行时可调试，而宏定义不可以。

缺点

- 代码膨胀。内联是以代码膨胀（复制）为代价，消除函数调用带来的开销。如果执行函数体内代码的时间，相比于函数调用的开销较大，那么效率的收获会很少。另一方面，每一处内联函数的调用都要复制代码，将使程序的总代码量增大，消耗更多的内存空间。
- inline 函数无法随着函数库升级而升级。inline函数的改变需要重新编译，不像 non-inline 可以直接链接。
- 是否内联，程序员不可控。内联函数只是对编译器的建议，是否对函数内联，决定权在于编译器。

虚函数（virtual）可以是内联函数（inline）吗?

- 虚函数可以是内联函数，内联是可以修饰虚函数的，但是当虚函数表现多态性的时候不能内联。
- 内联是在编译器建议编译器内联，而虚函数的多态性在运行期，编译器无法知道运行期调用哪个代码，因此虚函数表现为多态性时（运行期）不可以内联。
- inline virtual 唯一可以内联的时候是：编译器知道所调用的对象是哪个类（如 Base::who()），这只有在编译器具有实际对象而不是对象的指针或引用时才会发生;

静态库和动态库比较

静态库

将静态库的内容添加到程序中区，此时程序的空间，变成了源程序空间大小+静态库空间大小。

动态库（共享库）

常驻内存，当程序需要调用相关函数时，会从内存调用。

区别

静态库：对空间要求较低，而时间要求较高的核心程序中。
动态库：对时间要求较低，对空间要求较高。

虚函数、虚函数表，虚指针

在C++的标准规格说明书中说到，编译器必需要保证虚函数表的指针存在于对象实例中最前面的位置（这是为了保证正确取到虚函数的偏移量）。这意味着我们通过对象实例的地址得到这张虚函数表，然后就可以遍历其中函数指针，并调用相应的函数。

虚继承的作用是减少了对基类的重复，代价是增加了虚表指针的负担（更多的虚表指针）。详细请查阅：虚指针、虚函数原理

下面总结一下（当基类有虚函数时）：

每个类都有虚指针和虚表;
如果不是虚继承，那么子类将父类的虚指针继承下来，并指向自身的虚表（发生在对象构造时）。有多少个虚函数，虚表里面的项就会有多少。多重继承时，可能存在多个的基类虚表与虚指针;
如果是虚继承，那么子类会有两份虚指针，一份指向自己的虚表，另一份指向虚基表，多重继承时虚基表与虚基表指针有且只有一份。

C/C++如何判断两个小数是否相等

不能使用等号

```
const double EPSINON = 0.00000001; ///< double数精度设置为%.8lf即可
bool equal(double a, double b)
{
    if((a-b)>-EPSINON && (a-b)<EPSINON)
    {

```

```
        return true;
    }
    else
    {
        return false;
    }
}
```

C++空类的大小

本文中所说是C++的空类是指这个类不带任何数据，即类中没有非静态(non-static)数据成员变量，没有虚函数(virtual function)，也没有虚基类(virtual base class)。
直观地看，空类对象不使用任何空间，因为没有任何隶属对象的数据需要存储。然而，C++标准规定，凡是一个独立的(非附属)对象都必须具有非零大小。换句话说，

C++空类的大小不为0

为了验证这个结论，可以先来看测试程序的输出。

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

class NoMembers
{
};

int main()
{
    NoMembers n;  // Object of type NoMembers.
    cout << "The size of an object of empty class is: "
         << sizeof(n) << endl;
}
```

输出:

```
The size of an object of empty class is: 1
```

C++标准指出，不允许一个对象（当然包括类对象）的大小为0，不同的对象不能具有相同的地址。这是由于：

new需要分配不同的内存地址，不能分配内存大小为0的空间
避免除以 sizeof(T)时得到除以0错误
故使用一个字节来区分空类。

c++ 空类，含有虚函数的类的大小(此问题都是在32位机器上而言)

1、为何空类的大小不是0呢?

为了确保两个不同对象的地址不同，必须如此。

类的实例化是在内存中分配一块地址，每个实例在内存中都有独一无二的二地址。同样，空类也会实例化，所以编译器会给空类隐含的添加一个字节，这样空类实例化后就有独一无二的地址了。所以，空类的sizeof为1，而不是0.

2、请看下面的类:

class A{ virtual void f(){} };

class A{

virtual void f(){}

virtual void f1(){}

};

class B:public A{}

此时，类A和类B都不是空类，其sizeof都是4，因为它们都具有**虚函数表**的地址(**32位系统指针4字节，64位系统指针8字节**)。多个虚函数，也是对应一个虚函数表，所以都是对应为**一个指针大小。**

3、请看:

class A{};

class B:public virtual A{};

此时，A是空类，其大小为1；B不是空类，其大小为4.因为含有指向虚基类的指针。

4、多重继承的空类的大小也是1.

class Father1{}; class Father2{};

class Child:Father1, Father2{};

它们的sizeof都是1.

5、何时共享虚函数地址表：

如果派生类继承的第一个是基类，且该基类定义了虚函数地址表，则派生类就共享该表首址占用的存储单元。对于除前述情形以外的其他任何情形，派生类在处理完所有基类或虚基类后，根据派生类是否建立了虚函数地址表，确定是否为该表首址分配存储单元。

测试：运行下面的代码，输出是什么?

```
class A
{
};

class B
{
public:
        B() {}
~B() {}

};


class C
{
public:
        C() {}

        virtual ~C() {}
};

int _tmain(int argc, _TCHAR* argv[])
{
        printf("%d, %d, %d\n", sizeof(A), sizeof(B), sizeof(C));
        return 0;
}
```

答案是1, 1, 4。class A是一个空类型，它的实例不包含任何信息，本来求sizeof应该是0。但当我们声明该类型的实例的时候，它必须在内存中占有一定的空间，否则无法使用这些实例。至于占用多少内存，由编译器决定。Visual Studio 2008中每个空类型的实例占用一个byte的空间。

class B在class A的基础上添加了构造函数和析构函数。由于构造函数和析构函数的调用与类型的实例无关（调用它们只需要知道函数地址即可），在它的实例中不需要增加任何信息。所以sizeof(B)和sizeof(A)一样，在Visual Studio 2008中都是1。

class C在class B的基础上把析构函数标注为虚拟函数。C++的编译器一旦发现一个类型中有虚拟函数，就会为该类型生成虚函数表，并在该类型的每一个实例中添加一个指向虚函数表的指针。在32位的机器上，一个指针占4个字节的空间，因此sizeof©是4。

32位机与64位机指针占用空间不同

```
#include<iostream>
using namespace std;
int main()
{
cout << "sizeof(int*)="<<sizeof(int*) << endl;
cout << "sizeof(int)=" << sizeof(int) << endl;
}
```

`32位机  sizeof(int*)=4,  sizeof(int)=4`

`64位机  sizeof(int*)=8,  sizeof(int)=4`

sizeof(int)指的int占用的字节数，字节数为4.

sizeof(int*) 指的是指针变量占用的字节数

32为机上:sizeof(char*)=sizeof(int*)=sizeof(short*)=sizeof(long*)=4
64位机：sizeof(char*)=sizeof(int*)=sizeof(shor*)=sizeof(long*)=8

引经据典

感谢以下博主的文章，如有遗漏，请联系我添加，谢谢!

> https://blog.csdn.net/weixin_43519366/article/details/118634870
> https://blog.csdn.net/qq_31349683/article/details/112381183
> https://blog.csdn.net/lihao21/article/details/47973609
> https://blog.csdn.net/yhc166188/article/details/81159415
> https://www.cnblogs.com/yuanshijie/p/12884288.html
> https://blog.csdn.net/qq_43686329/article/details/119811453?utm_medium=distribute.pc_feed_404.none-task-blog-2$_{default}$BlogCommendFromBaidu$_{Rate-3.pc_404_mixedpudn\&depth_1-utm_source=distribute.pc_feed_404.none-task-blog-2default}$$_{BlogCommendFromBaidu}$Rate-3.pc_404_mixedpud