**1、 在main执行之前和之后执行的代码可能是什么？**

**main函数执行之前**，主要就是初始化系统相关资源：

* 设置栈指针
* 初始化静态static变量和global全局变量，即.data段的内容
* 将未初始化部分的全局变量赋初值：数值型short，int，long等为0，bool为FALSE，指针为NULL等等，即.bss段的内容
* 全局对象初始化，在main之前调用构造函数，这是可能会执行前的一些代码
* 将main函数的参数argc，argv等传递给main函数，然后才真正运行main函数
* \_\_attribute\_\_((constructor))

**main函数执行之后**：

* 全局对象的析构函数会在main函数之后执行；
* 可以用 **atexit** 注册一个函数，它会在main 之后执行;
* \_\_attribute\_\_((destructor))

## 2、结构体内存对齐问题？

##### 内存对齐规则

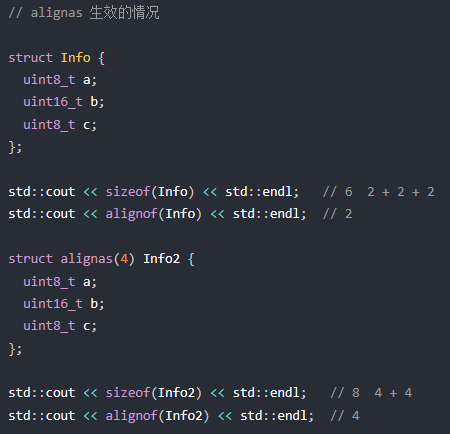
1.第一个成员在与结构体变量偏移量为0的地址处；

2.其它成员变量要对齐到**对齐数**的整数倍地址处。这个对齐数并不是待对齐成员变量的大小，而是**编译器默认对齐数和该成员大小两者中的较小值**。比如V**S中默认的对齐数是8，Linux默认对齐数是4**。

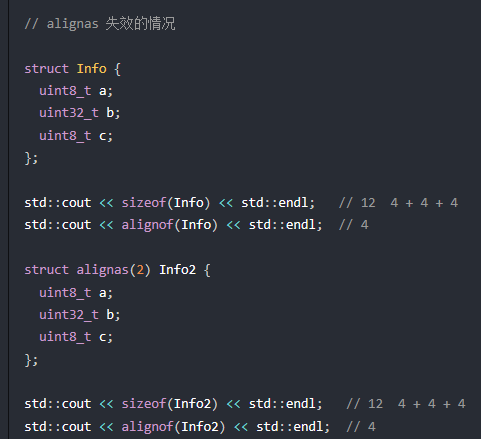
3.结构体整体大小是最大对齐数的整数倍。这个最大对齐数是这样理解的，就是一个结构体里面有多个不同数据类型的成员变量，按照规则2，它们都有一个对齐数，这些对齐数里最大的就是最大对齐数——显然在windows里不会超过8，linux里不会超过4

4.对于结构体里嵌套了结构体的情况，首先里面那个结构体就按照前面三个规则对齐，该内置结构体的首地址就在**自己里面最大对齐数的整数倍处**，比如4的倍数处，对于外面这个结构体，也就是所要求的整个结构体的大小，它就是自己的普通成员变量和嵌套结构体内成员变量中取 最大对齐数 的整数倍。

alignas将内存对齐调整为4个字节。所以sizeof(Info2)的值变为了8：



若alignas小于自然对齐的最小单位，则被忽略：



如果想使用单字节对齐的方式，应该使用#pragma pack(push,1)或者使用\_\_attribute\_\_((packed))

## 3、指针和引用的区别

* 指针是一个变量，存储的是一个地址，引用跟原来的变量实质上是同一个东西，是原变量的别名
* 指针可以有多级，引用只有一级
* 指针可以为空，引用不能为NULL且在定义时必须初始化
* 指针在初始化后可以改变指向，而引用在初始化之后不可再改变
* sizeof指针得到的是本指针的大小，sizeof引用得到的是引用所指向变量的大小
* 当把指针作为参数进行传递时，也是将实参的一个拷贝传递给形参，两者指向的地址相同，但不是同一个变量，在函数中改变这个变量的指向不影响实参，而引用却可以。
* 引用本质是一个指针，同样会占4字节内存；指针是具体变量，需要占用存储空间（32位系统4字节，64位系统8字节）。
* 引用在声明时必须初始化为另一变量，一旦出现必须为typename refname &varname形式；指针声明和定义可以分开，可以先只声明指针变量而不初始化，等用到时再指向具体变量。
* 引用一旦初始化之后就不可以再改变（变量可以被引用为多次，但引用只能作为一个变量引用）；指针变量可以重新指向别的变量。
* 不存在指向空值的引用，必须有具体实体；但是存在指向空值的指针。

**在编译器看来**, int a = 10; int &b = a; 等价于 int \* const b = &a; 而 b = 20; 等价于 \*b = 20; 自动转换为指针和自动解引用

## 4、在传递函数参数时，什么时候该使用指针，什么时候该使用引用呢？

* 需要返回函数内局部变量的内存的时候用指针。使用指针传参需要开辟内存，用完要记得释放指针，不然会内存泄漏。而返回局部变量的引用是没有意义的
* 对栈空间大小比较敏感（比如递归）的时候使用引用。使用引用传递不需要创建临时变量，开销要更小
* 类对象作为参数传递的时候使用引用，这是C++类对象传递的标准方式

## 5、堆和栈的区别



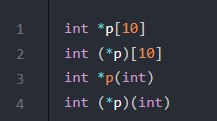
## 6、你觉得堆快一点还是栈快一点？

栈快一点。

因为操作系统会在底层对栈提供支持，会分配专门的寄存器存放栈的地址，栈的入栈出栈操作也十分简单，并且有专门的指令执行，所以栈的效率比较高也比较快。

而堆的操作是由C/C++函数库提供的，在分配堆内存的时候需要一定的算法寻找合适大小的内存。并且获取堆的内容需要两次访问，第一次访问指针，第二次根据指针保存的地址访问内存，因此堆比较慢。

## 7、区别以下指针类型？



* int \*p[10]表示指针数组，强调数组概念，是一个数组变量，数组大小为10，数组内每个元素都是指向int类型的指针变量。
* int (\*p)[10]表示数组指针，强调是指针，只有一个变量，是指针类型，不过指向的是一个int类型的数组，这个数组大小是10。

Int a[10];

int (\*pa)[10] = &a;

Int b[5][10];

int (\*pb)[10] = b;

* int \*p(int)是函数声明，函数名是p，参数是int类型的，返回值是int \*类型的。
* int (\*p)(int)是函数指针，强调是指针，该指针指向的函数具有int类型参数，并且返回值是int类型的。

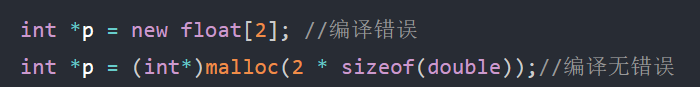
## 8、new / delete 与 malloc / free的异同

**相同点**

* 都可用于内存的动态申请和释放

**不同点**

* 前者是C++**运算符**，后者是C/C++语言**标准库函数**
* new自动计算要分配的空间大小，malloc需要手工计算
* new是类型安全的，malloc不是。例如：



* new调用名为**operator new**的标准库函数分配足够空间并调用相关对象的构造函数，delete对指针所指对象运行适当的析构函数；然后通过调用名为**operator delete**的标准库函数释放该对象所用内存。malloc仅仅分配内存空间，free仅仅回收空间，不具备调用构造函数和析构函数功能。
* 后者需要库文件支持，前者不用
* new是封装了malloc，直接free不会报错，但是这只是释放内存，而不会析构对象
* malloc返回的是void\*类型指针（必须进行类型转换），new返回的是具体类型指针。

## 9、new和delete是如何实现的？

* new的实现过程是：首先调用名为**operator new**的标准库函数，分配足够大的原始为类型化的内存，以保存指定类型的一个对象；接下来运行该类型的一个构造函数，用指定初始化构造对象；最后返回指向新分配并构造后的的对象的指针
* delete的实现过程：对指针指向的对象运行适当的析构函数；然后通过调用名为**operator delete**的标准库函数释放该对象所用内存

## 12、被free回收的内存是立即返还给操作系统吗？

不是的，被free回收的内存会首先被ptmalloc使用双链表保存起来，当用户下一次申请内存的时候，会尝试从这些内存中寻找合适的返回。这样就避免了频繁的系统调用，占用过多的系统资源。同时ptmalloc也会尝试对小块内存进行合并，避免过多的内存碎片。

## 13、宏定义和函数有何区别？

* 宏在预处理阶段完成替换，之后被替换的文本参与编译，相当于直接插入了代码，运行时不存在函数调用，执行起来更快；函数调用在运行时需要跳转到具体调用函数。
* 宏定义属于在结构中插入代码，没有返回值；函数调用具有返回值。
* 宏定义参数没有类型，不进行类型检查；函数参数具有类型，需要检查类型。
* 宏是不可以调试的，而函数可以进行单步调试
* 宏不支持递归，函数支持递归

## 14、宏定义和typedef区别？

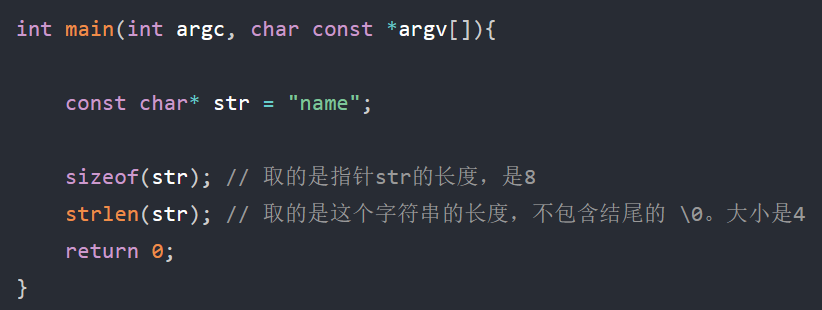
* 宏主要用于定义常量及书写复杂的内容；typedef主要用于定义类型别名。
* 宏替换发生在编译阶段之前，属于文本插入替换；typedef是编译的一部分。
* 宏不检查类型；typedef会检查数据类型。
* 宏不是语句，不在最后加分号；typedef是语句，要加分号标识结束。
* 注意对指针的操作，typedef char \* p\_char和#define p\_char char \*区别巨大。

## 15、变量声明和定义区别？

* 声明仅仅是把变量的声明的位置、名称及类型提供给编译器，并不分配内存空间，不赋初值；定义要在定义的地方为其分配存储空间。
* 相同变量可以在多处声明（外部变量extern），但只能在一处定义。

## 16、strlen和sizeof区别？

* sizeof是运算符，并不是函数，结果在编译时得到而非运行中获得；strlen是字符处理的库函数。
* sizeof参数可以是任何数据的类型或者数据（sizeof参数不退化）；strlen的参数只能是字符指针且结尾是'\0'的字符串。
* 因为sizeof值在编译时确定，所以不能用来得到动态分配（运行时分配）存储空间的大小。



## 16.2、（补充题）一个指针占多少字节？

64位的**编译环境**下，指针的占用大小为8字节；

32位的**编译环境**下，指针的占用大小为4字节。

一个指针占内存的大小跟**编译环境**有关，而与**机器本身**的位数无关。

## 17、常量指针和指针常量区别？

* 指针常量是一个指向只读变量的指针，读成常量的指针，也就是后面所指明的int const 和 const int，都是一个常量，可以写作int const \*p或const int \*p。
* 常量指针是一个不能改变指向的指针。指针是个常量，必须初始化，一旦初始化完成，它的值（也就是存放在指针中的地址）就不能在改变了，即不能中途改变指向，如int \*const p。

## 18、a和&a有什么区别？

int a[10];

int (\*p)[10] = &a;

* a是数组名，是数组首元素地址，+1表示地址值加上一个int类型的大小，如果a的值是0x00000001，加1操作后变为0x00000005。\*(a + 1) = a[1]。
* &a是数组的指针，其类型为int (\*)[10]（就是前面提到的数组指针），其加1时，系统会认为是数组首地址加上整个数组的偏移（10个int型变量），值为数组a尾元素后一个元素的地址。
* 若(int \*)p ，此时输出 \*p时，其值为a[0]的值，因为被转为int \*类型，解引用时按照int类型大小来读取。
* a和&a值是一样的

## 19、C++和Python的区别

* Python是一种脚本语言，是解释执行的，而C++是编译语言，是需要编译后在特定平台运行的。python可以很方便的跨平台，但是效率没有C++高。
* Python使用缩进来区分不同的代码块，C++使用花括号来区分
* C++中需要事先定义变量的类型，而Python不需要，Python的基本数据类型只有数字，布尔值，字符串，列表，元组等等
* Python的库函数比C++的多，调用起来很方便

## 20、C++和C语言的区别

* C++中new和delete是对内存分配的运算符，取代了C中的malloc和free。
* 标准C++中的字符串类取代了标准C函数库头文件中的字符数组处理函数（C中没有字符串类型）。
* C++中用来做控制态输入输出的iostream类库替代了标准C中的stdio函数库。
* C++中的try/catch/throw异常处理机制取代了标准C中的setjmp()和longjmp()函数。
* 在C++中，允许有相同的函数名，不过它们的参数类型不能完全相同，这样这些函数就可以相互区别开来。而这在C语言中是不允许的。也就是C++可以重载，C语言不允许。
* C++语言中，允许变量定义语句在程序中的任何地方，只要在是使用它之前就可以；而C语言中，必须要在函数开头部分。
* 在C++中，除了值和指针之外，新增了引用。引用型变量是其他变量的一个别名，我们可以认为他们只是名字不相同，其他都是相同的。
* C++相对与C增加了一些关键字，如：bool、using、dynamic\_cast、namespace等等

## 21、C++与Java的区别

**语言特性**

* Java语言给开发人员提供了更为简洁的语法；完全面向对象，由于JVM可以安装到任何的操作系统上，所以说它的可移植性强
* Java语言中没有指针的概念，引入了真正的数组。不同于C++中利用指针实现的“伪数组”，Java引入了真正的数组，同时将容易造成麻烦的指针从语言中去掉，这将有利于防止在C++程序中常见的因为数组操作越界等指针操作而对系统数据进行非法读写带来的不安全问题
* C++也可以在其他系统运行，但是需要不同的编码（这一点不如Java，只编写一次代码，到处运行），例如对一个数字，在windows下是大端存储，在unix中则为小端存储。Java程序一般都是生成字节码，在JVM里面运行得到结果
* Java用接口(Interface)技术取代C++程序中的抽象类。接口与抽象类有同样的功能，但是省却了在实现和维护上的复杂性

**垃圾回收**

* C++用析构函数回收垃圾，写C和C++程序时一定要注意内存的申请和释放
* Java语言不使用指针，内存的分配和回收都是自动进行的，程序员无须考虑内存碎片的问题

**应用场景**

* Java在桌面程序上不如C++实用，C++可以直接编译成exe文件，指针是c++的优势，可以直接对内存的操作，但同时具有危险性 。（操作内存的确是一项非常危险的事情，一旦指针指向的位置发生错误，或者误删除了内存中某个地址单元存放的重要数据，后果是可想而知的）
* Java在Web 应用上具有C++ 无可比拟的优势，具有丰富多样的框架
* 对于底层程序的编程以及控制方面的编程，C++很灵活，因为有句柄的存在

## 22、C++中struct和class的区别

**相同点**

* 两者都拥有成员函数、公有和私有部分
* 任何可以使用class完成的工作，同样可以使用struct完成

**不同点**

* 两者中如果不对成员不指定公私有，struct默认是公有的，class则默认是私有的
* class默认是private继承， 而struct默认是public继承

## 22.2、C和C++中struct的区别

* C语言中：struct是用户自定义数据类型（UDT）；C++中struct是抽象数据类型（ADT），支持成员函数的定义，（C++中的struct能继承，能实现多态）
* C中struct是没有权限的设置的，且struct中只能是一些变量的集合体，可以封装数据却不可以隐藏数据，而且成员**不可以是函数**
* C++中，struct增加了访问权限，且可以和类一样有成员函数，成员默认访问说明符为public（为了与C兼容）
* struct作为类的一种特例是用来自定义数据结构的。一个结构标记声明后，在C中必须在结构标记前加上struct，才能做结构类型名（除：typedef struct class{};）;C++中结构体标记（结构体名）可以直接作为结构体类型名使用，此外结构体struct在C++中被当作类的一种特例

## 23、define宏定义和const的区别

**编译阶段**

* define是在编译的**预处理**阶段起作用，而const是在**编译、运行**的时候起作用

**安全性**

* define只做替换，不做类型检查和计算
* const常量**有数据类型**，编译器可以对其进行类型安全检查

**内存占用**

* define只是将宏名称进行替换，在内存中会产生多分相同的备份。const在程序运行中只有一份备份，且可以执行常量折叠，能将复杂的的表达式计算出结果放入**常量表**
* 宏定义的数据没有分配内存空间，只是插入替换掉；const定义的变量只是值不能改变，但要分配内存空间。

## 24、C++中const和static的作用

**static**

* 不考虑类的情况
  + 隐藏。所有不加static的全局变量和函数具有全局可见性，可以在其他文件中使用，加了之后只能在该文件所在的编译模块中使用
  + 默认初始化为0，包括未初始化的全局静态变量与局部静态变量，都存在全局未初始化区
  + 静态变量在函数内定义，始终存在，且只进行一次初始化，具有记忆性，其作用范围与局部变量相同，函数退出后仍然存在，但不能使用
* 考虑类的情况
  + static成员变量：只与类关联，不与类的对象关联。定义时要分配空间，不能在类声明中初始化，必须在类定义体外部初始化，初始化时不需要标示为static；可以被非static成员函数任意访问。
  + static成员函数：不具有this指针，无法访问类对象的非static成员变量和非static成员函数；**不能被声明为const、虚函数和volatile**；可以被非static成员函数任意访问

**const**

* 不考虑类的情况
  + const常量在定义时必须初始化，之后无法更改
  + const形参可以接收const和非const类型的实参，例如// i 可以是 int 型或者 const int 型void fun(const int& i){ //...}
  + const修饰变量是也与static有一样的隐藏作用。只能在该文件中使用，其他文件不可以引用声明使用。 因此在**头文件中声明const变量是没问题的**，因为即使被多个文件包含，链接性都是内部的，不会出现符号冲突。
* 考虑类的情况
  + const成员变量：不能在类定义外部初始化，只能通过构造函数初始化列表进行初始化，并且必须有构造函数
  + const成员函数：const对象不可以调用非const成员函数；非const对象都可以调用；不可以改变非mutable（用该关键字声明的变量可以在const成员函数中被修改）数据的值

## 25、C++的顶层const和底层const

* **顶层**const：指的是const修饰的变量**本身**是一个常量（常量指针，或普通常量），无法修改，指的是指针，就是 \* 号的右边
* **底层**const：指的是const修饰的变量**所指向的对象**是一个常量（指针常量），指的是所指变量，就是 \* 号的左边

使用命名的强制类型转换函数const\_cast时，只能改变运算对象的底层const ？？？试了一下好像都可以

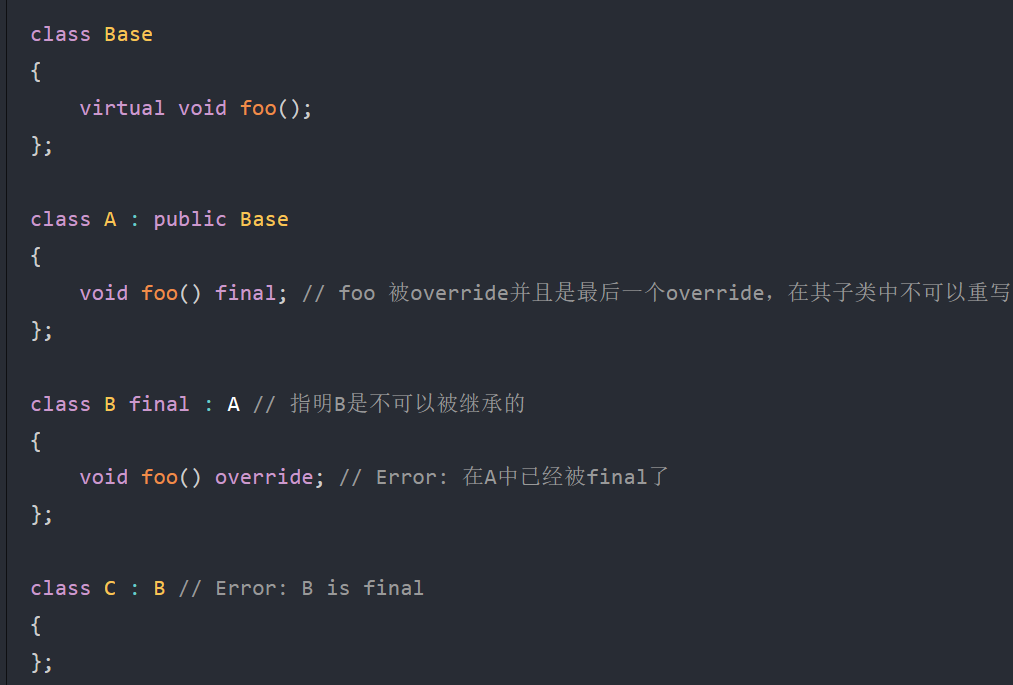
## 26、数组名和指针（这里为指向数组首元素的指针）区别？

* 二者均可通过增减偏移量来访问数组中的元素。
* 数组名不是真正意义上的指针，可以理解为常指针，所以数组名没有自增、自减等操作。
* **当数组名当做形参传递给调用函数后，就失去了原有特性，退化成一般指针，多了自增、自减操作，但sizeof运算符不能再得到原数组的大小了。**

## 27、override和final关键字

override的作用是指定了所修饰的函数是对父类的虚函数的重写，如果父类中没有就会报错。如果不加override，若函数名打错，会认为是一个新的函数，并不会报错

当不希望某个类被继承，或不希望某个虚函数被重写，可以在类名和虚函数后添加final关键字，添加final关键字后被继承或重写，编译器会报错。



## 28、拷贝初始化和直接初始化

直接初始化直接调用**与实参匹配的构造函数**，拷贝初始化总是调用**拷贝构造函数**。拷贝初始化首先使用指定构造函数创建一个临时对象，然后用拷贝构造函数将那个临时对象拷贝到正在创建的对象。

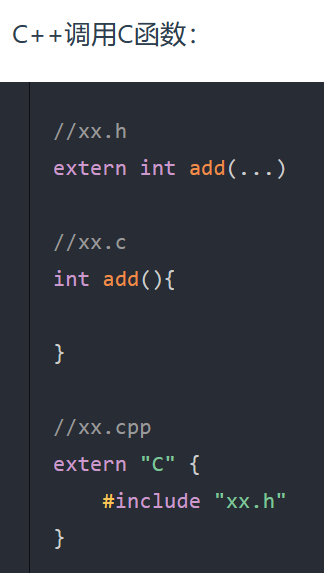
* 使用explicit修饰构造函数时：如果构造函数存在隐式转换，编译时会报错

## 29、初始化和赋值的区别

* 对于简单类型来说，初始化和赋值没什么区别
* 对于类和复杂数据类型来说，初始化调用构造函数/拷贝构造函数，赋值调用赋值运算符重载函数

## 30、extern"C"的用法

为了能够**正确的在C++代码中调用C语言**的代码：在程序中加上extern "C"后，相当于告诉编译器这部分代码是C语言写的，因此要按照C语言进行编译



## 31、野指针和悬空指针

野指针：指针变量未及时初始化。 解决方法：定义指针变量及时初始化，要么置空。

悬空指针：指针free或delete之后没有及时置空。 解决方法：释放操作后立即置空。

## 32、C和C++的类型安全

**什么是类型安全？**

类型安全很大程度上可以等价于内存安全，类型安全的代码不会试图访问自己没被授权的内存区域。类型安全的编程语言与类型安全的程序之间，**没有**必然联系。

**（1）C的类型安全**

C中相当多的操作是不安全的。比如使用void\*进行类型转换会导致转换后错误。

C中malloc函数可以随意写类型而不会报错

**（2）C++的类型安全**

C++比C更有类型安全性，提供了一些新的机制保障类型安全：

* 操作符new返回的指针类型严格与对象匹配，而不是void\*
* C中很多以void\*为参数的函数可以改写为C++模板函数，而模板是支持类型检查的；
* 引入const关键字代替#define constants，它是有类型、有作用域的，而#define constants只是简单的文本替换
* 一些#define宏可被改写为inline函数（宏在预处理阶段替换，inline在编译阶段展开），结合函数的重载，可在类型安全的前提下支持多种类型，当然改写为模板也能保证类型安全
* C++提供了**dynamic\_cast**关键字，使得转换过程更加安全，因为dynamic\_cast比static\_cast涉及更多具体的类型检查。

## 33、C++中的重载、重写（覆盖）和隐藏的区别

（1）重载（overload）

重载是指在同一范围定义中的同名成员函数才存在重载关系。主要特点是函数名相同，参数类型和数目有所不同，不能出现参数个数和类型均相同，**仅仅依靠返回值**不同来区分的函数。

（2）重写（覆盖）（override）

重写指的是在派生类中覆盖基类中的同名函数，**重写就是重写函数体**，**要求基类函数必须是虚函数**且：

* 与基类的虚函数有相同的参数个数
* 与基类的虚函数有相同的参数类型
* 与基类的虚函数有相同的返回值类型

除了函数体不同其他都相同

重载与重写的区别：重写关系中，调用方法根据对象类型决定，重载根据调用时实参表与形参表的对应关系来选择函数体

（3）隐藏（hide）

派生类中的函数屏蔽了基类中的同名函数

隐藏与重写的区别：基类函数是否是虚函数。此外**两个函数参数不同，无论基类函数是不是虚函数，都会被隐藏。**

## 34、C++有哪几种的构造函数

* 默认构造函数
* 初始化构造函数（有参数）
* 拷贝构造函数
* 转换构造函数（用于将其他类型的变量，隐式转换为本类对象）
* 移动构造函数（move和右值引用）
* 委托构造函数（委托构造函数会将控制权交给目标构造函数，在目标构造函数执行完之后，再执行委托构造函数的主体）

## 35、浅拷贝和深拷贝的区别

**浅拷贝**

浅拷贝只是拷贝一个指针，并没有新开辟一个地址，拷贝的指针和原来的指针指向同一块地址，如果原来的指针所指向的资源释放了，那么再释放浅拷贝的指针的资源就会出现错误。

**深拷贝**

深拷贝不仅拷贝值，还开辟出一块新的空间用来存放新的值，即使原先的对象被析构掉，释放内存了也不会影响到深拷贝得到的值。在自己实现拷贝赋值的时候，如果有指针变量的话是需要自己实现深拷贝的。

## 36、内联函数和宏定义的区别

* 在使用时，宏只做简单字符串替换（预处理阶段）。而内联函数可以进行参数类型检查（编译阶段），且具有返回值（**是一个真的函数**）。
* 内联函数在编译时直接将函数代码嵌入到目标代码中，省去函数调用的开销来提高执行效率，并且进行参数类型检查，具有返回值，可以实现重载。
* 宏定义时要注意书写（参数要括起来）否则容易出现歧义，内联函数不会产生歧义
* 内联函数有类型检测、语法判断等功能，而宏没有

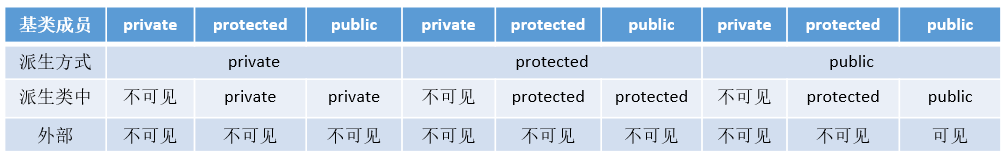
**内联函数适用场景:**

* 使用宏定义的地方都可以使用 inline 函数。
* 作为类成员接口函数来读写类的私有成员或者保护成员，会提高效率。

## 37、public，protected和private访问和继承权限/public/protected/private的区别？

* public的变量和函数在类的内部外部都可以访问。
* protected的变量和函数只能在类的内部和其派生类中访问。
* private修饰的元素只能在类内访问。

访问权限和派生类访问权限：



## 38、如何用代码判断大小端存储？

大端存储：字数据的高字节存储在低地址中

小端存储：字数据的低字节存储在低地址中

**方式一：使用强制类型转换**

**方式二：巧用union联合体**

## 39、volatile、mutable和explicit关键字的用法

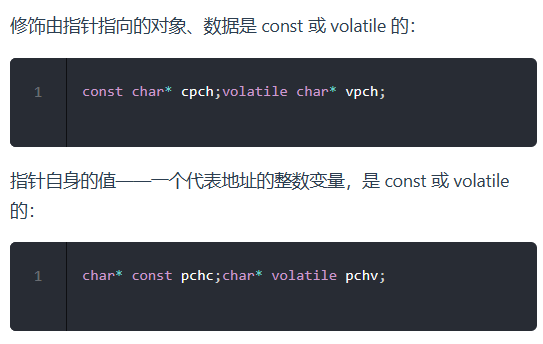
(1) **volatile**

volatile 关键字是一种类型修饰符，**用它声明的类型变量表示可以被某些编译器未知的因素更改**，比如：操作系统、硬件或者其它线程等。遇到这个关键字声明的变量，编译器对访问该变量的代码就不再进行优化，从而可以提供对特殊地址的稳定访问。

当要求使用 volatile 声明的变量的值的时候，**系统总是重新从它所在的内存读取数据**，即使它前面的指令刚刚从该处读取过数据。

**volatile定义变量的值是易变的，每次用到这个变量的值的时候都要去重新读取这个变量的值，而不是读寄存器内的备份。多线程中被几个任务共享的变量需要定义为volatile类型。**

volatile 指针和 const 修饰词类似，const 有常量指针和指针常量的说法，volatile 也有相应的概念：



（2）**mutable**

被mutable修饰的变量，将永远处于可变的状态，即使在一个const函数或一个const对象中也可以修改这个变量。

（3）**explicit**

explicit关键字用来修饰类的构造函数，被修饰的构造函数的类，不能发生相应的隐式类型转换，只能以**显式的方式进行类型转换**

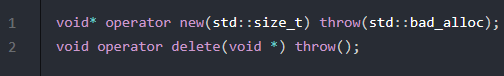
## 40、什么情况下会调用拷贝构造函数

* 用类的一个实例化对象去初始化另一个对象的时候
* 函数的参数是类的对象时（非引用传递）
* 函数的返回值是函数体内局部对象的类的对象时（此时会发生（Named return Value优化）NRV优化，不同的编译器结果不同）

## 41、C++中有几种类型的new

（1）**plain new**

普通的new。



在空间分配失败的情况下，抛出异常**std::bad\_alloc**而不是返回NULL

（2）**nothrow new**

在空间分配失败的情况下是不抛出异常，而是返回NULL

（3）**placement new**

允许在一块已经分配成功的内存上重新构造对象或对象数组。placement new不用担心内存分配失败，因为它根本不分配内存，它做的唯一一件事情就是调用对象的构造函数。

* placement new构造起来的对象数组，要显式的调用他们的析构函数来销毁（析构函数并不释放对象的内存，以便其他对象的构造），千万不要使用delete，这是因为placement new构造起来的对象或数组大小并不一定等于原来分配的内存大小，使用delete会造成内存泄漏或者之后释放内存时出现运行时错误。

## 42、C++的异常处理的方法

**（1）try、throw和catch关键字**

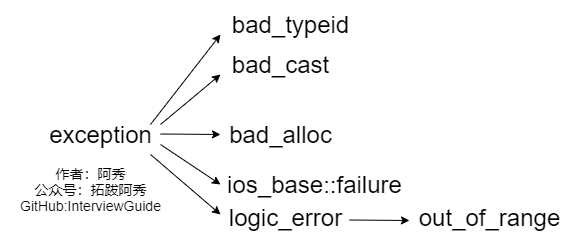
**（2）函数的异常声明列表**

可以在函数声明和定义时，指出所能抛出异常的列表：



表明函数可能会抛出int,double型或者A、B、C三种类型的异常，如果throw中为空，表明不会抛出任何异常（C++11 已经不用这种写法，换成了noexcept关键字），如果没有throw则可能抛出任何异常

**（3）C++标准异常类 exception**



* bad\_typeid：使用typeid运算符，如果其操作数是一个多态类的指针，而该指针的值为 NULL，则会拋出此异常
* bad\_cast：在用 dynamic\_cast 进行从多态基类对象（或引用）到派生类的引用的强制类型转换时，如果转换是不安全的或非法的，则会拋出此异常
* bad\_alloc：在用 new 运算符进行动态内存分配时，如果没有足够的内存，则会引发此异常
* out\_of\_range:用 vector 或 string的at 成员函数根据下标访问元素时，如果下标越界，则会拋出此异常

## 43、static的用法和作用？

* 隐藏。当同时编译多个文件时，所有未加static前缀的全局变量和函数都具有全局可见性，加了static只有当前文件可见。
* 保持变量内容的持久。（static变量中的记忆功能和全局生存期）存储在静态数据区的变量会在程序刚开始运行时就完成初始化，也是唯一的一次初始化。共有两种变量存储在静态存储区：全局变量和static变量，只不过和全局变量比起来，static可以控制变量的可见范围。
* static变量默认初始化为0
* 类成员声明static

## 46、值传递、指针传递、引用传递的区别和效率

1. 值传递：有一个形参向函数所属的栈拷贝数据的过程，如果值传递的对象是类对象 或是大的结构体对象，将耗费一定的时间和空间。（传值）
2. 指针传递：同样有一个形参向函数所属的栈拷贝数据的过程，但拷贝的数据是一个固定为4字节（指针的大小）的地址。（传值，传递的是地址值）
3. 引用传递：同样有上述的数据拷贝过程，但其是针对地址的，相当于为该数据所在的地址起了一个别名。（传地址）
4. 效率上讲，指针传递和引用传递比值传递效率高。一般主张使用引用传递，代码逻辑上更加紧凑、清晰。

## 47、静态变量什么时候初始化