**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

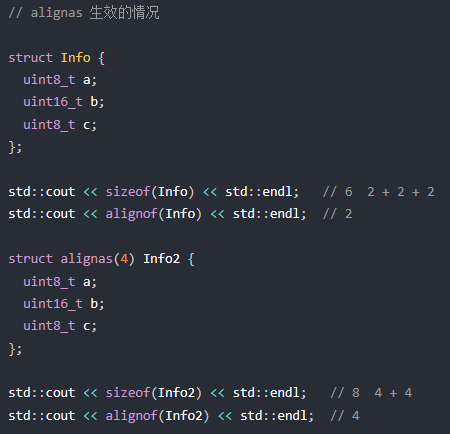
1.第一个成员在与结构体变量偏移量为0的地址处；

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**。

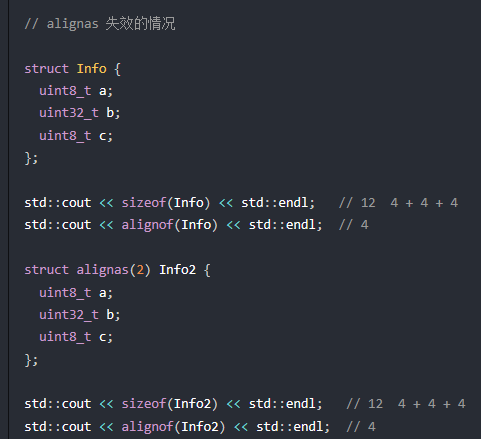
3.结构体整体大小是最大对齐数的整数倍。这个最大对齐数是这样理解的，就是一个结构体里面有多个不同数据类型的成员变量，按照规则2，它们都有一个对齐数，这些对齐数里最大的就是最大对齐数——显然在windows里不会超过8，linux里不会超过4

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



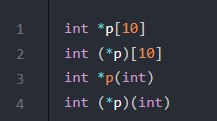
## 6、你觉得堆快一点还是栈快一点？

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。

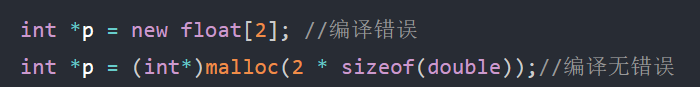
## 8、new / delete 与 malloc / free的异同

**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。
* new内存分配失败时，会抛出bac\_alloc异常。malloc分配内存失败时返回NULL

## 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？

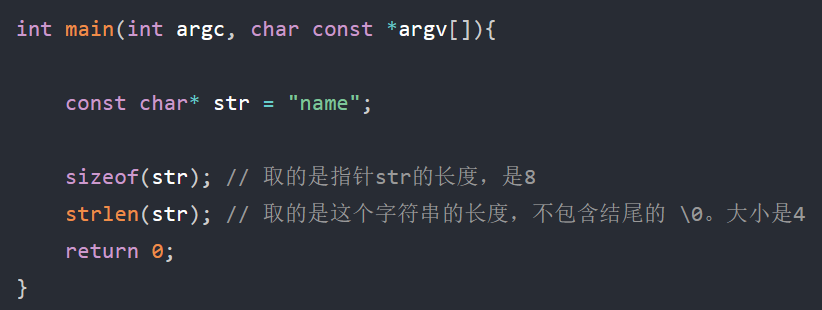
* 宏主要用于定义常量及书写复杂的内容；typedef主要用于定义类型别名。
* 宏替换发生在编译阶段之前，属于文本插入替换；typedef是编译的一部分。
* 宏不检查类型；typedef会检查数据类型。
* 宏不是语句，不在最后加分号；typedef是语句，要加分号标识结束。
* 注意对指针的操作，typedef char \* p\_char和#define p\_char char \*区别巨大。

## 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

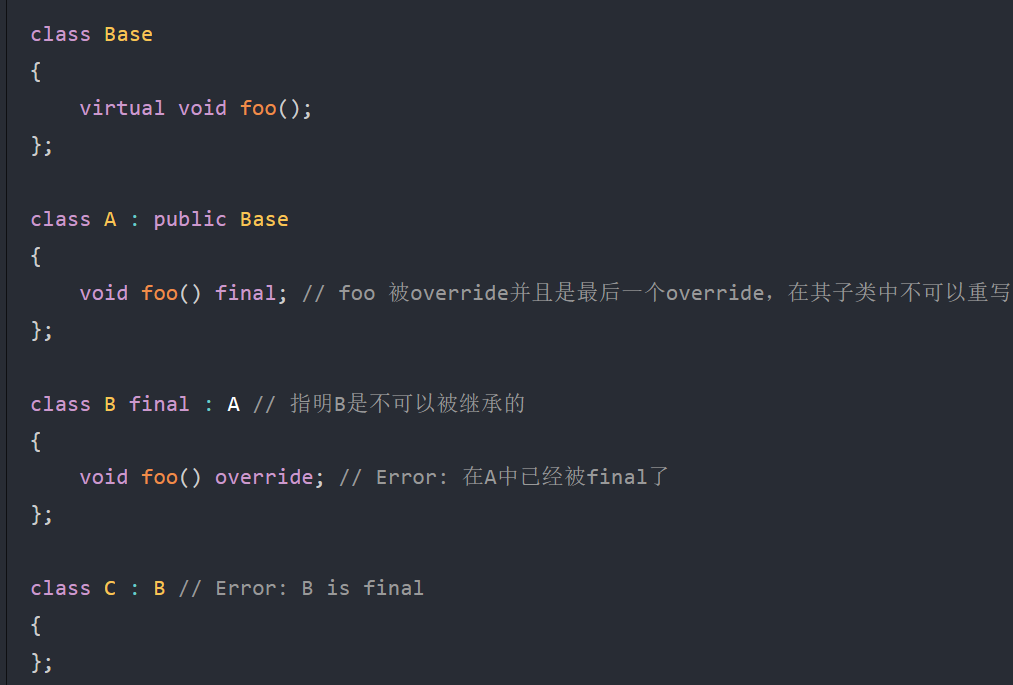
## 26、数组名和指针（这里为指向数组首元素的指针）区别？

* 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。

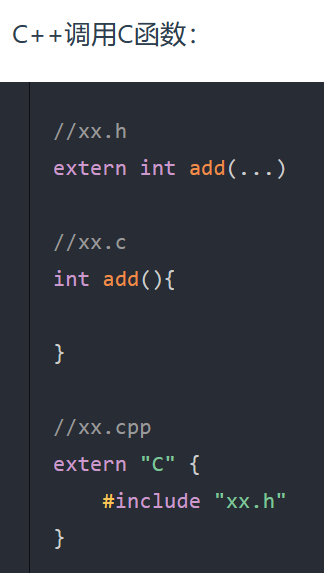
* 使用explicit修饰构造函数时：如果构造函数存在隐式转换，编译时会报错

## 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

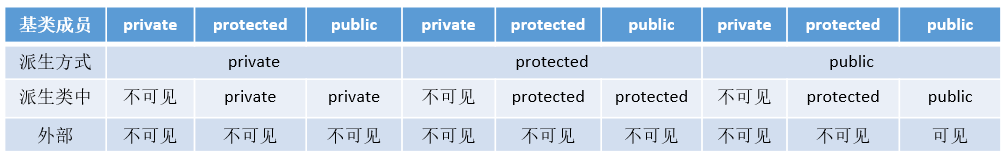
**内联函数适用场景:**

* 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

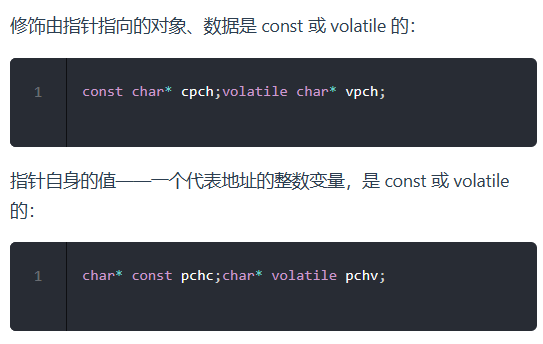
(1) **volatile**

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：



volatile用在如下的几个地方：

1. 中断服务程序中修改的供其它程序检测的变量需要加volatile；
2. 多任务环境下各任务间共享的标志应该加volatile；
3. 存储器映射的硬件寄存器通常也要加volatile说明，因为每次对它的读写都可能由不同意义；

（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**

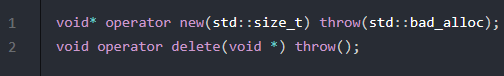
## 40、什么情况下会调用拷贝构造函数

* 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**

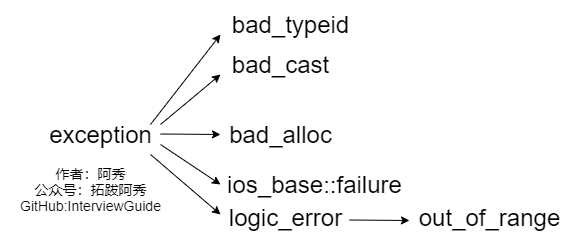
**（2）函数的异常声明列表**

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

在C中，初始化发生在代码执行之前，编译阶段分配好内存之后，就会进行初始化，所以我们看到在C语言中无法使用变量对静态局部变量进行初始化

而在C++中，初始化时在执行相关代码时才会进行初始化，主要是由于C++引入对象后，要进行初始化必须执行相应构造函数和析构函数，在构造函数或析构函数中经常会需要进行某些程序中需要进行的特定操作，并非简单地分配内存。所以C++标准定为全局或静态对象是有首次用到时才会进行构造，并通过atexit()来管理。

## 48、const关键字的作用有哪些?

1. 修饰一个普通的常量
2. 常量指针
3. 函数声明中修饰形参，表明它是一个输入参数，在函数内部不能改变其值；
4. 对于类的成员函数，若指定其为const类型，则表明其是一个常函数，不能修改类的成员变量，类的常对象只能访问类的常成员函数；
5. 对于类的成员函数，有时候必须指定其返回值为const类型，以使得其返回值不为“左值”。

注意：const类型变量必须定义的时候进行初始化

## 49、什么是类的继承？

继承就是一个类继承了另一个类的属性和方法，这个新的类包含了上一个类的属性和方法，被称为子类或者派生类，被继承的类称为父类或者基类；

## 50、从汇编层去解释一下引用



将x的地址存入变量b中。从汇编层次来看，的确引用是通过指针来实现的。

## 53、operator delete、delete、delete []、destory、deallocator都有什么作用？

operator delete只有释放内存

delete将对象析构和内存释放组合在一起的。

delete[]时，数组中的元素按逆序的顺序进行销毁。

deallocate只是释放对象内存（因为是基于operator delete函数封装的），并没有调用对象的析构函数释放对象内的动态内存

destory显式调用一个对象的[析构函数](https://so.csdn.net/so/search?q=%E6%9E%90%E6%9E%84%E5%87%BD%E6%95%B0&spm=1001.2101.3001.7020)

## 54、new和delete的实现原理， delete是如何知道释放内存的大小的？delete和delete[]区别？

1、new

简单类型直接调用operator new分配内存；

复杂类型，先调用operator new分配内存，然后在分配的内存上调用构造函数；

简单类型，new[]计算好大小后调用operator new；

复杂类型，new[]先调用operator new[]分配内存，然后在p的前四个字节写入数组大小n，然后调用n次构造函数

① new表达式调用一个名为operator new(operator new[])函数，分配一块足够大的、原始的、未命名的内存空间；

② 编译器运行相应的构造函数以构造这些对象，并为其传入初始值；

③ 对象被分配了空间并构造完成，返回一个指向该对象的指针。

2、delete

简单类型默认只是调用operator delete函数；

复杂类型先调用析构函数再调用operator delete；

简单类型，或是自定义类型但没有显式定义析构函数，delete和delete[]等同（delete也可以对数组使用！！）。

复杂类型。编译器在分配时，会在数组首地址之前再申请一块空间用于记录数组个数（如果没有析构函数就不需要知道个数，编译器也就不会多申请这块空间），对于x64，这个大小存储在数组首地址的前八个字节，对于x86则是前4个字节。在使用delete去释放一个数组对象时，由于传入的是数组首地址，但是申请的内存应该是数组首地址再往前8个字节的位置，数组首地址操作系统并未记录，所以会出错，同样的，使用delete[]去释放单个对象，由于他会访问前8个字节取得大小从而决定调用多少次析构函数，这时候行为将会不确定，却决于前8个字节会是什么值，可能会调用很多次析构函数，但在最后free时会出错，因为new对象时记录的是首地址，而不是首地址-8。

## 55、malloc申请的存储空间能用delete释放吗?

理论上可以，但最好不要，还是要配对使用

## 56、malloc与free的实现原理？

1. malloc 与 free的原理

malloc 函数利用系统调用来完成对内存的申请分配，系统将可用的内存块连接成为一个长长的列表称为空闲链表，调用malloc时，将遍历链表，找到满足申请空间的空闲链表块，然后将链表块分为两个部分，一部分大小和用户申请的相同，分配给用户使用，另一部分返回给链表。

调用free时，将用户释放的内存块连接到链表中。这样，空闲链表中会存在大量的内存碎片，如果用户调用malloc申请大的内存块时，链表中就可能不存在满足要求的空闲内存块，这时malloc函数请求延时，并对空闲链表中内存块执行整理，将相邻的内存块合并成大的内存块，如果用户的申请的内存大于空闲链表可提供的最大连续内存，将调用系统调用指令mmap向系统申请新的虚拟内存满足用户需要，并将其加入链表管理。

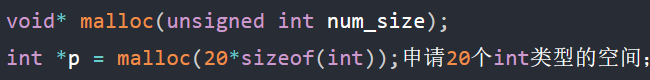
2. malloc 与free的实现

利用空闲链表算法作为堆空间分配算法，在实现上将整个堆空间按照是否被占用分割为若干空闲(Free)块和占用(Used)块，它们之间利用双向链表连接起来。当用户要申请一块内存时，堆分配算法将遍历整个链表，直到找到一块满足要求的空闲块，如果这个空闲块正好和所申请大小相同，就直接标记空闲块为占用块，然后将地址返回给用户。如果空闲块大小大于用户申请大小，就将空闲块分割为两部分，其中一块大小为申请大小，分配给用户，并标记为占用，另外一块仍为空闲块。如果没有找到足够大的空闲块，则利用系统调用(brk, mmap)向系统申请再申请一块空间。

当用户释放一块内存时，堆分配算法会判别被释放的内存块钱后两个块是否为空闲块，如果是，将它们合并为一个大的空闲块，标记为空闲块，不是，直接标记为空闲块。

## 57、malloc、realloc、calloc的区别

1. malloc函数

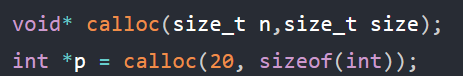


malloc带一个参数，用于指定期望分配的内存大小按字节计算。

malloc分配内存后不对其进行初始化

运行效率高

2. calloc函数



calloc带两个参数，一个是对象数量，另一个是单个对象大小

calloc申请的空间的值是初始化为0的

运行效率低

3. realloc函数



给已经动态分配的空间重新分配空间，增加和减少都可以，一般用于增加。

扩容时：

**如果原有空间之后有足够多的空间时，直接追加**

**如果原有空间之后没有足够多的空间时，会在堆空间上另找一个合适大小的连续空间来使用。**

## 58.1、类成员初始化方式？

赋值初始化，通过在函数体内进行赋值初始化

列表初始化，在冒号后使用初始化列表进行初始化。

区别：

赋值初始化是在所有的数据成员被分配内存空间后才进行的。

列表初始化是给数据成员分配内存空间时就进行初始化。就是说初始化这个数据成员时函数体还未执行。

## 58.2、为什么用成员初始化列表会快一些？

赋值初始化是在构造函数当中做赋值的操作，而列表初始化是做纯粹的初始化操作。C++的赋值操作是会产生临时对象的，会降低程序的效率。

## 58.3、构造函数的执行顺序？

① 虚拟基类的构造函数（多个虚拟基类则按照继承的顺序执行构造函数）。

② 基类的构造函数（多个普通基类也按照继承的顺序执行构造函数）。

③ 类类型的成员对象的构造函数（按照成员对象在类中的定义顺序）

④ 派生类自己的构造函数。

## 59、有哪些情况必须用到成员列表初始化？作用是什么？

① 当初始化一个引用成员时；

② 当初始化一个常量成员时；

③ 当调用一个基类的构造函数，而它拥有一组参数时；

④ 当调用一个成员类的构造函数，而它拥有一组参数时；

list中的项目初始化顺序是由类中的成员声明顺序决定的（基类构造函数在其他前面），不是初始化列表中的排列顺序决定的

## 60、C++中新增了string，它与C语言中的 char \*有什么区别吗？它是如何实现的？

string继承自basic\_string,其实是对char\*进行了封装，封装的string包含了char\*数组，容量，长度等等属性。

string可以进行动态扩展，在每次扩展的时候另外申请一块原空间大小两倍的空间（2\*n），然后将原字符串拷贝过去，并加上新增的内容。

## 61、什么是内存泄露，如何检测与避免

**内存泄露**

一般我们常说的内存泄漏是指**堆内存的泄漏**。应用程序使用malloc,、realloc、 new等函数从堆中分配到块内存，使用完后程序没有调用相应的free或delete释放该内存块，就会导致内存泄漏。

**避免内存泄露的几种方式**

* 计数法：使用new或者malloc时，让该数+1，delete或free时，该数-1，程序执行完打印这个计数，如果不为0则表示存在内存泄露
* 一定要将基类的析构函数声明为**虚函数**
* 对象**数组**的释放一定要用**delete []**
* 有new就有delete，有malloc就有free，保证它们一定成对出现

**检测工具**

* Linux下可以使用**Valgrind工具**
* Windows下可以使用**BoundsChecker、CRT库**

## 62、对象复用的了解，零拷贝的了解

**对象复用**

对象复用其本质是一种设计模式：Flyweight享元模式。通过将对象存储到“对象池”中实现对象的重复利用，这样可以**避免多次创建重复对象**的开销，节约系统资源。

**零拷贝**

零拷贝就是一种避免 CPU 将数据从一块存储拷贝到另外一块存储的技术。

零拷贝技术可以减少数据拷贝和共享总线操作的次数。

例如：vector的一个成员函数emplace\_back()很好地体现了零拷贝技术，它跟push\_back()函数一样可以将一个元素插入容器尾部，区别在于：**使用push\_back()函数需要调用拷贝构造函数和转移构造函数，而使用emplace\_back()插入的元素原地构造，不需要触发拷贝构造和转移构造**，效率更高。举个例子：



## 63、介绍面向对象的三大特性，并且举例说明

继承、封装和多态

**（1）继承**

**让某种类型对象获得另一个类型对象的属性和方法。**

它可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展

常见的继承有三种方式：

1. **实现继承**：指使用基类的属性和方法而无需额外编码的能力
2. **接口继承**：指仅使用属性和方法的名称、但是子类必须提供实现的能力
3. 可视继承：指子窗体（类）使用基窗体（类）的外观和实现代码的能力（C++里好像不怎么用）

例如，将人定义为一个抽象类，拥有姓名、性别、年龄等公共属性，吃饭、睡觉、走路等公共方法，在定义一个具体的人时，就可以继承这个抽象类，既保留了公共属性和方法，也可以在此基础上扩展跳舞、唱歌等特有方法

**（2）封装**

数据和代码捆绑在一起，避免外界干扰和不确定性访问。**把客观事物封装成抽象的类**，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏。

例如：将公共的数据或方法使用public修饰，而不希望被访问的数据或方法采用private修饰。

**（3）多态**

同一事物表现出不同事物的能力，即向不同对象发送同一消息，不同的对象在接收时会产生不同的行为

**允许将子类类型的指针赋值给父类类型的指针**

编译时多态：重载

运行时多态：虚函数

## 65、C++的四种强制转换reinterpret\_cast、const\_cast、static\_cast 、dynamic\_cast

**reinterpret\_cast**

reinterpret\_cast<type-id> (expression)

type-id 必须是一个指针、引用、算术类型、函数指针或者成员指针。它可以用于类型之间进行强制转换。

**const\_cast**

const\_cast<type\_id> (expression)

该运算符用来修改类型的const或volatile属性。除了const 或volatile修饰之外， type\_id和expression的类型是一样的。用法如下：

* 常量指针被转化成非常量的指针，并且仍然指向原来的对象
* 常量引用被转换成非常量的引用，并且仍然指向原来的对象
* const\_cast一般用于修改底指针。如const char \*p形式

**static\_cast**

没有运行时类型检查来保证转换的安全性

* 用于类层次结构中基类（父类）和派生类（子类）之间指针或引用的转换
  + 进行上行转换（把派生类的指针或引用转换成基类表示）是安全的
  + 进行下行转换（把基类指针或引用转换成派生类表示）时是不安全的
* 用于基本数据类型之间的转换，如把int转换成char，把int转换成enum。这种转换的安全性也要开发人员来保证。
* 把空指针转换成目标类型的空指针
* 把任何类型的表达式转换成void类型

**dynamic\_cast**

用于类层次间的上行转换和下行转换，有类型检查

如果 type-id 是类指针类型，那么expression也必须是一个指针，如果 type-id 是一个引用，那么 expression 也必须是一个引用

如果下行转换不安全，这个运算符会传回空指针

## 66、C++函数调用的压栈过程



当函数从入口函数main函数开始执行时，编译器会将我们操作系统的运行状态，main函数的返回地址、main的参数、mian函数中的变量进行依次压栈；

当main函数开始调用func()函数时，编译器此时会将main函数的运行状态进行压栈，再将func()函数的返回地址、func()函数的参数从右到左、func()定义变量依次压栈；

当func()调用f()的时候，编译器此时会将func()函数的运行状态进行压栈，再将的返回地址、f()函数的参数从右到左、f()定义变量依次压栈

从代码的输出结果可以看出，函数f(var1)、f(var2)依次入栈，而后先执行f(var2)，再执行f(var1)，最后打印整个字符串，将栈中的变量依次弹出，最后主函数返回。

## 67、写C++代码时有一类错误是 coredump ，很常见，你遇到过吗？怎么调试这个错误？

coredump是程序由于异常或者bug在运行时异常退出或者终止，在一定的条件下生成的一个叫做core的文件，这个core文件会记录程序在运行时的内存，寄存器状态，内存指针和函数堆栈信息等等。对这个文件进行分析可以定位到程序异常的时候对应的堆栈调用信息。

## 68、说说移动构造函数

用对象a初始化对象b后，对象a不再使用了，此时可以直接使用a的空间，这样就避免了新的空间的分配，大大降低了构造的成本。这就是移动构造函数设计的初衷

移动构造函数中，对于指针，我们采用浅拷贝。但是这个浅拷贝是安全的，只要避免第一个指针释放空间就可以了。避免的方法就是将第一个指针（比如a->value）置为NULL，这样在调用析构函数的时候，由于有判断是否为NULL的语句，所以析构a的时候并不会回收a->value指向的空间；

移动构造函数的参数和拷贝构造函数不同，拷贝构造函数的参数是一个左值引用，但是移动构造函数的参数是一个右值引用。意味着，移动构造函数的参数是一个右值或者将亡值的引用。也就是说，只有用一个右值或者将亡值初始化另一个对象的时候，才会调用移动构造函数。move语句，可以将一个左值强制变成一个将亡值。

## 69、C++中将临时变量作为返回值时的处理过程

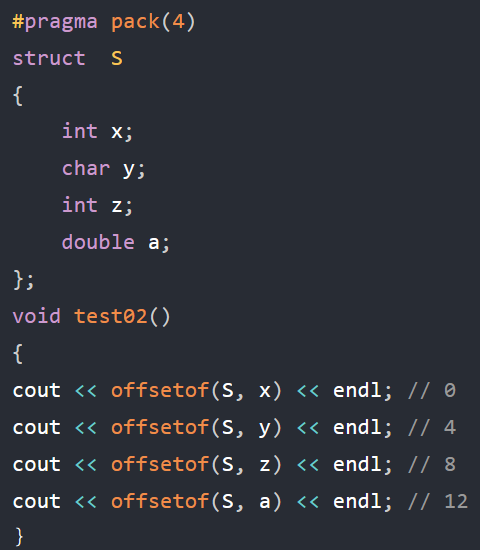
临时变量，在函数调用过程中是被压到程序进程的栈中的，当函数退出时，临时变量出栈，即临时变量已经被销毁，临时变量占用的内存空间没有被清空，但是可以被分配给其他变量，所以有可能在函数退出时，该内存已经被修改了，对于临时变量来说已经是没有意义的值了

函数调用结束后，返回值被临时存储到寄存器中，并没有放到堆或栈中，也就是说与内存没有关系了。当退出函数的时候，临时变量可能被销毁，但是返回值却被放到寄存器中与临时变量的生命周期没有关系

总结：临时变量被销毁；返回值保存下来，如果需要直接赋值即可

## 70、如何获得结构成员相对于结构开头的字节偏移量

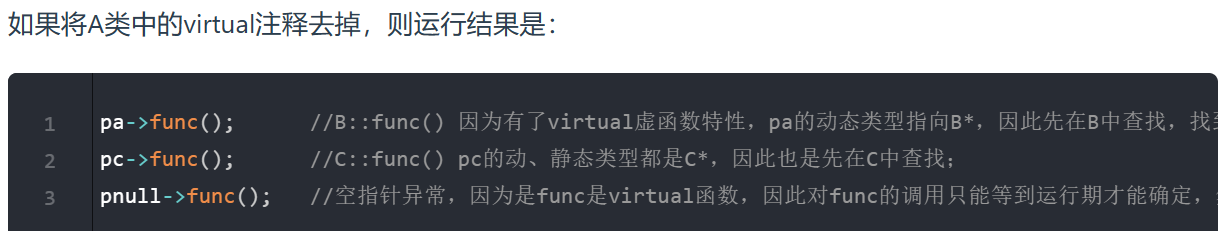
使用<stddef.h>头文件中的offsetof宏



## 71、静态类型和动态类型，静态绑定和动态绑定的介绍

* 静态类型：对象在声明时采用的类型，在编译期既已确定；
* 动态类型：通常是指一个指针或引用目前所指对象的类型，是在运行期决定的；
* 静态绑定：绑定的是静态类型，所对应的函数或属性依赖于对象的静态类型，发生在编译期；
* 动态绑定：绑定的是动态类型，所对应的函数或属性依赖于对象的动态类型，发生在运行期；





总结：要想实现多态，必须使用动态绑定。在继承体系中只有虚函数使用的是动态绑定，其他的全部是静态绑定；

## 72、引用是否能实现动态绑定，为什么可以实现？

可以

在访问虚函数时，编译器会根据其所绑定的对象类型（动态类型）决定要调用哪个函数。注意只能调用虚函数。

## 73、全局变量和局部变量有什么区别？

生命周期不同：全局变量随主程序创建和创建，随主程序销毁而销毁；局部变量在局部函数内部，甚至局部循环体等内部存在，退出就不存在；

使用方式不同：通过声明后全局变量在程序的各个部分都可以用到；局部变量分配在堆栈区，只能在局部使用。

操作系统和编译器通过内存分配的位置可以区分两者，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 。

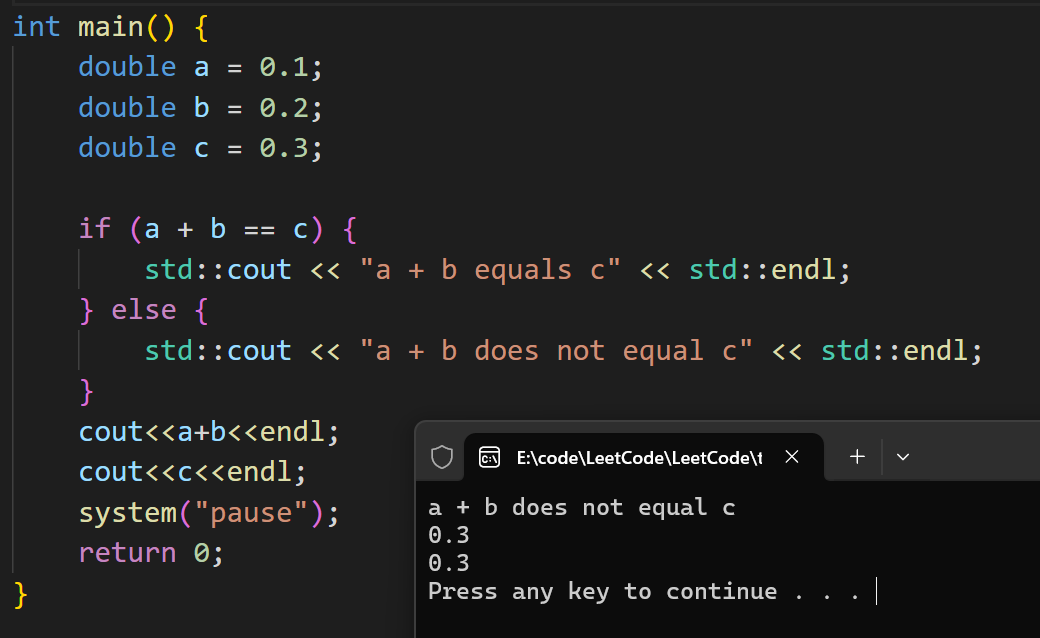
## 74、指针加减计算要注意什么？

指针加减本质是对其所指地址的移动，移动的步长跟指针的类型是有关系的

**实际跨越的内存间隔是指针类型的长度，计算时建议都转成10进制计算，计算结果除以类型长度取得结果**

## 75、 怎样判断两个浮点数是否相等？

不能直接用==来判断



对于两个浮点数比较只能通过相减并与预先设定的精度比较，记得要取绝对值！浮点数与0的比较也应该注意。与浮点数的表示方式有关。



## 77、C++中的指针参数传递和引用参数传递有什么区别？底层原理你知道吗？

**1)** 指针参数传递本质上是值传递，它所传递的是一个地址值，会在栈中开辟内存空间以存放由主调函数传递进来的实参值。形参指针变了，实参指针不会变

**2)** 引用参数传递过程中，被调函数的形式参数也作为局部变量在栈中开辟了内存空间，但是这时存放的是由主调函数放进来的实参变量的地址。被调函数对形参的任何操作都会影响主调函数中的实参变量。

## 78、类如何实现只能静态分配和只能动态分配

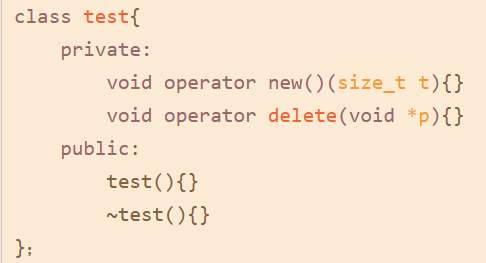
建立类的对象有两种方式：

① 静态建立，静态建立一个类对象，就是由编译器为对象在栈空间中分配内存；

② 动态建立，A \*p = new A();动态建立一个类对象，就是使用new运算符为对象在堆空间中分配内存。这个过程分为两步，第一步执行operator new()函数，在堆中搜索一块内存并进行分配；第二步调用类构造函数构造对象；

实现只能静态分配：

把new、delete运算符重载为private属性



实现只能动态分配：

把构造、析构函数设为protected属性，再用子类或提供一个public的构造和析构接口来动态创建

## 79、如果想将某个类用作基类，为什么该类必须定义而非声明？

派生类中包含并且可以使用它从基类继承而来的成员，为了让派生类能够使用这些成员，派生类必须知道他们是什么。所以必须定义而非声明。

## 80、 继承机制中对象之间如何转换？指针和引用之间如何转换？

① 向上类型转换

将派生类指针或引用转换为基类的指针或引用被称为向上类型转换，向上类型转换会自动进行，而且向上类型转换是安全的。

② 向下类型转换

将基类指针或引用转换为派生类指针或引用被称为向下类型转换，是不安全的。用dynamic\_cast进行向下类型转换。

## 81、知道C++中的组合吗？它与继承相比有什么优缺点吗？

**组合**

组合也就是设计类的时候把要组合的类的对象加入到该类中作为自己的成员变量。

优点：

①：当前对象只能通过所包含的那个对象去调用其方法，所以所包含的对象的内部细节对当前对象时不可见的。

②：当前对象与包含的对象是一个低耦合关系，如果修改包含对象的类中代码不需要修改当前对象类的代码。

③：当前对象可以在运行时动态的绑定所包含的对象。可以通过set方法给所包含对象赋值。

缺点：

①：容易产生过多的对象。

②：为了能组合多个对象，必须仔细对接口进行定义。

**继承**

优点：子类可以重写父类的方法来方便地实现对父类的扩展

缺点：

①：父类的内部细节对子类是可见的。

②：子类从父类继承的方法在编译时就确定下来了，所以无法在运行期间改变从父类继承的方法的行为。

③：如果对父类的方法做了修改的话（比如增加了一个参数），则子类的方法必须做出相应的修改。所以说子类与父类是一种高耦合，违背了面向对象思想。

## 82、函数指针？

**1) 什么是函数指针?**

函数指针指向的是特殊的数据类型（函数），函数的类型是由其返回的数据类型和其参数列表共同决定的。

**2) 函数指针的声明方法**

int (\*pf)(const int&, const int&);

如果\*pf外面没加括号就变成一个函数的声明了

**3) 为什么有函数指针**

可以把函数也当作一个参数输入到另一个函数中。我们希望在同一个函数中通过使用相同的形参在不同的时间使用产生不同的效果。

**4) 一个函数名就是一个指针，它指向函数的代码。**

**5) 两种方法赋值：**

指针名 = 函数名； 指针名 = &函数名

## 83、说一说你理解的内存对齐以及原因

1、 分配内存的顺序是按照声明的顺序。

2、 每个变量相对于起始位置的偏移量必须是该变量类型大小的整数倍，不是整数倍空出内存，直到偏移量是整数倍为止。

3、 最后整个结构体的大小必须是里面变量类型最大值的整数倍。

添加了#pragma pack(n)后规则就变成了下面这样：

1、 偏移量要是n和当前变量大小中较小值的整数倍

2、 整体大小要是n和最大变量大小中较小值的整数倍

3、 n值必须为1,2,4,8…，为其他值时就按照默认的分配规则

## 84、结构体变量比较是否相等

需要重载”==”运算符。对于结构体中的成员变量，如果是元素需要一个个比较（可能要进一步调用其他结构体的==运算符），如果是指针直接比较。

## 85、函数调用过程栈的变化，返回地址和参数变量哪个先入栈？

① 将函数的参数压入栈中。参数的压入顺序与调用约定有关，可以是从右到左，也可以是从左到右。

② 将当前函数的返回地址压入栈中。返回地址是指函数调用结束后回到调用点的地址。

③ 将被调用函数的栈帧（Stack Frame）压入栈中。栈帧包括**本地变量、临时变量、函数的返回值**等信息。栈帧的大小取决于函数中定义的变量和数据类型。

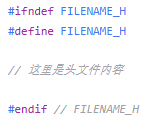
④ 跳转到函数的入口点开始执行函数代码。

⑤ 函数执行完毕后，将返回值存放在寄存器中（或者放在栈内存中），然后将栈帧弹出，恢复返回地址，跳转回调用点。

## 86、define、const、typedef、inline的使用方法？他们之间有什么区别？

一、#define与const的区别：

1. const定义的常量是变量，带类型，而#define定义的只是个常数，不带类型；
2. define只在预处理阶段起作用，简单的文本替换，而const在编译、链接过程中起作用；
3. define只是简单的字符串替换没有类型检查。而const是有数据类型的，是要进行类型检查的，可以避免一些低级错误；
4. define预处理后，占用代码段空间，const占用数据段空间；
5. const不能重定义，而define可以通过#undef取消某个符号的定义，进行重定义；
6. define独特功能，比如可以用来防止文件重复引用。



二、#define和别名typedef的区别

1. 执行时间不同，typedef在编译阶段有效，typedef有类型检查的功能；#define是宏定义，发生在预处理阶段，不进行类型检查；
2. 功能差异，typedef用来定义类型的别名，定义与平台无关的数据类型，与struct的结合使用等。#define不只是可以为类型取别名，还可以定义常量、变量、编译开关等。
3. 作用域不同，#define没有作用域的限制，只要是之前预定义过的宏，在以后的程序中都可以使用。而typedef有自己的作用域。

三、 define与inline的区别

1. #define是关键字，inline是函数；
2. 宏定义在预处理阶段进行文本替换，inline函数在编译阶段进行替换；
3. inline函数有类型检查，相比宏定义比较安全；

## 87、你知道printf函数的实现原理是什么吗？

printf的声明：   int \_cdecl printf(const char\* format, ...);

调用规则遵循\_cdedl调用规则。 在\_cdedl规则中：

1.  参数从右到左依次入栈  
2.  调用者负责清理堆栈

3.  参数的数量类型不会导致编译阶段的错误

对于printf函数来说，调用者通过第一个fmt参数中的%+格式字符的方式通知了被调用者（printf的实现者）

### 格式解析

扫描format参数里的字符，如果是普通字符就打印输出，如果是%，就说明后面有可能是格式字符，需要进行检测，然后从栈低（其实是第一个参数的位置）弹出指定类型的数据，按照指定格式（十进制、十六进制、指定宽度、指定精度等等）进行输出

## 88、为什么模板类的声明和实现一般都是放在同一个.h文件中

C++程序**预编译过程中，所有头文件都会展开。头文件声明的作用，就是让编译器知道，这个函数的定义应该在其他文件中，就不会因为暂时找不到函数的定义而报错。至于找到对应的定义，就是链接器需要干的事情。**

如果函数模板和类模板的声明和定义分散在.h文件和.cpp文件中，运行会报一个链接错误，提示找不到函数的具体定义。是因为函数模板实际上并不是一个真正的函数。它仅仅是编译器用来生成函数或类的一张“图纸”。模板不会占用内存，最终生成的函数才会占用内存。函数模板只是用来告诉编译器，在遇到某个函数调用的时候需要生成什么样子的函数。在编译过程中，编译器会参考函数模板，然后生成对应的真正的函数。

模板仅在需要的时候才会实例化出来

当编译器只看到模板的声明时，它不能实例化该模板，只能创建一个具有外部连接的符号并期待连接器能够将符号的地址决议出来。当实现该模板的.cpp文件中没有用到模板的实例时，编译器就不会实例化，于是连接器也黔驴技穷了。

## 90、cout和printf有什么区别？

cout是类std::ostream的全局对象。cout<<后可以跟不同的类型是因为<<运算符已存在针对各种类型数据进行了重载。安全。

printf是一个变参函数。没有类型检查，不安全。printf碰到不认识的类型就没办法了，而cout可以自己重载进行扩展。

## 91、重载运算符？

1、我们只能重载已有的运算符，而无权发明新的运算符；对于一个重载的运算符，其优先级和结合律与内置类型一致才可以；不能改变运算符操作数个数；

2、两种重载方式：成员运算符和非成员运算符，成员运算符比非成员运算符少一个参数；下标运算符、箭头运算符必须是成员运算符；

3、引入运算符重载，是为了实现类的多态性；

4、当重载的运算符是成员函数时，this绑定到左侧运算符对象。成员运算符函数的参数数量比运算符对象的数量少一个；至少含有一个类类型的参数；

5、从参数的个数推断到底定义的是哪种运算符，当运算符既是一元运算符又是二元运算符（+，-，\*，&）；

6、下标运算符必须是成员函数，下标运算符通常以所访问元素的引用作为返回值，同时最好定义下标运算符的常量版本和非常量版本；

7、箭头运算符必须是类的成员，返回对象的指针；解引用通常也是类的成员，返回对象的引用

## 92、当程序中有函数重载时，函数的匹配原则和顺序是什么？

1.精确匹配，包括以下情况(它们具有相同且最高的匹配度)：

实参类型和形参类型相同。

实参从数组类型或函数类型转换成对应的指针类型

向实参添加顶层const或者从实参中删除顶层const（形参的顶层cosnt是可以直接忽略掉的，顶层const不能作为重载条件）

2.通过const转化实现匹配(底层const的转换)

3.通过类型提升实现的匹配

4.通过算术类型转换实现的匹配

5.通过类类型转换实现的匹配

## 93、定义和声明的区别

**如果是指变量的声明和定义：** 从编译原理上来说，声明是仅仅告诉编译器，有个某类型的变量会被使用，但是编译器并不会为它分配任何内存。而定义就是分配了内存。

**如果是指函数的声明和定义：** 声明：一般在头文件里，对编译器说：这里我有一个函数叫function() 让编译器知道这个函数的存在。 定义：一般在源文件里，具体就是函数的实现过程 写明函数体。

## 94、全局变量和static变量的区别

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。

## 95、 静态成员与普通成员的区别是什么？

1. 生命周期

静态成员变量从类被加载开始到类被卸载，一直存在；

普通成员变量只有在类创建对象后才开始存在，对象结束，它的生命期结束；

1. 共享方式

静态成员变量是全类共享；普通成员变量是每个对象单独享用的；

1. 定义位置

普通成员变量存储在栈或堆中，而静态成员变量存储在静态全局区；

1. 初始化位置

普通成员变量在类中初始化；静态成员变量在类外初始化；

1. 默认实参

可以使用静态成员变量作为默认实参

## 96、说一下你理解的 ifdef endif代表着什么？

有时希望对其中一部分内容只在满足一定条件才进行编译，也就是对一部分内容指定编译的条件，这就是“条件编译”。

在一个大的软件工程里面，可能会有多个文件同时包含一个头文件，导致“重定义”错误。在头文件中使用#define、#ifndef、#ifdef、#endif能避免头文件重定义。

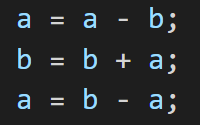
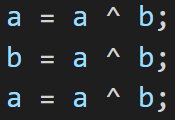
## 97、隐式转换，如何消除隐式转换？

C++中提供了explicit关键字，在构造函数声明的时候加上explicit关键字，能够禁止隐式转换。如果构造函数只接受一个参数（或有多个参数，但只有第一个没有缺省值），则它实际上定义了转换为此类类型的隐式转换机制。可以通过将构造函数声明为explicit加以制止隐式类型转换

## 98、C++如何处理多个异常的？

 C++异常处理机制由3个模块组成：try(检查)、throw(抛出)、catch(捕获) 抛出异常的语句格式为：throw 表达式；如果try块中程序段发现了异常则抛出异常。抛出不同类型的异常以处理多种异常。

## 99、如何在不使用额外空间的情况下，交换两个数？

①  ② 

注意异或存在的问题：若a和b是同一个变量的引用，则交换完之后变量会变成0

## 100、你知道strcpy和memcpy的区别是什么吗？

1、复制的内容不同。strcpy只能复制字符串，而memcpy可以复制任意内容，例如字符数组、整型、结构体、类等。

2、复制的方法不同。strcpy不需要指定长度，它遇到被复制字符的串结束符"\0"才结束，所以容易溢出。memcpy则是根据其第3个参数决定复制的长度。

头文件都是<string.h>



## 101、程序在执行int main(int argc, char \*argv[])时参数的含义？

参数的含义是程序在命令行下运行的时候，需要输入argc 个参数，每个参数是以char 类型输入的，依次存在数组里面，数组是 argv[]，所有的参数在指针

char \* 指向的内存中，数组的中元素的个数为 argc 个，第一个参数为程序的名称。

## 103、如果有一个空类，它会默认添加哪些函数？

C++中**空类默认会产生以下6个函数：**

**默认构造函数**

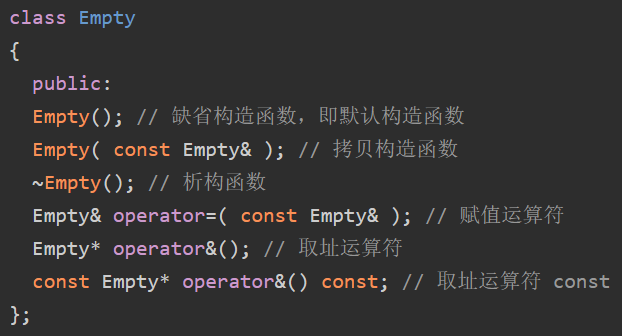
**拷贝构造函数**

**析构函数**

**赋值运算符重载函数**

**取址运算符重载函数**

**const取址运算符重载函数等。**



## 104、C++中标准库是什么？

1. C++ 标准库可以分为两部分：

标准函数库： 这个库是由通用的、独立的、不属于任何类的函数组成的。函数库继承自 C 语言。

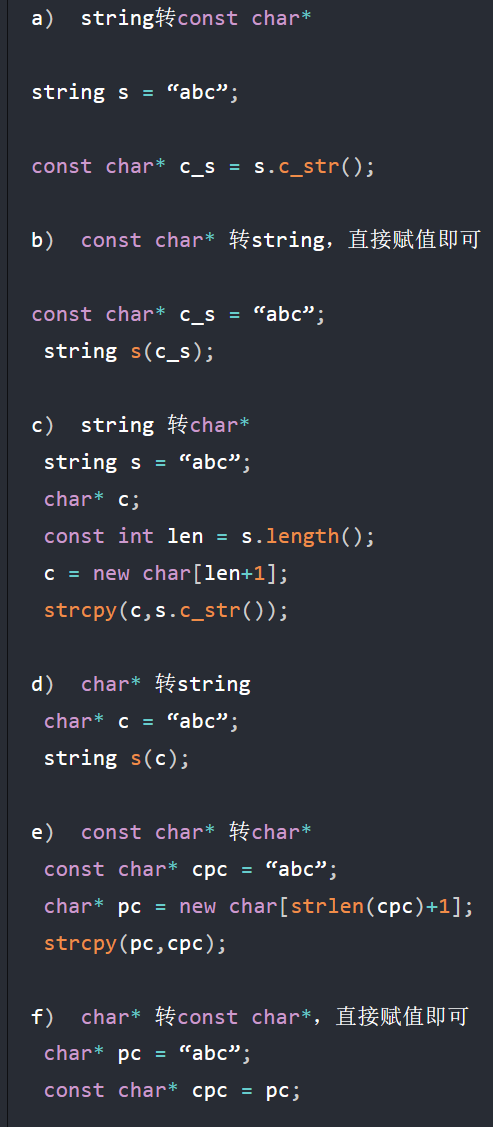
面向对象类库： 这个库是类及其相关函数的集合。

1. 输入/输出 I/O、字符串和字符处理、数学、时间、日期和本地化、动态分配、其他、宽字符函数
2. 标准的 C++ I/O 类、String 类、数值类、STL 容器类、STL 算法、STL 函数对象、STL 迭代器、STL 分配器、本地化库、异常处理类、杂项支持库

## 105、你知道const char\* 与string之间的关系是什么吗？

string 是c++标准库里面其中一个，封装了对字符串的操作，实际操作过程我们可以用const char\*给string类初始化

string、const char\*、char\* 三者转化关系：



## 106、你什么情况用指针当参数，什么时候用引用，为什么？

对于使用值而不做修改的函数：

如果数据对象很小，如内置数据类型或者小型结构，则按照值传递；

如果数据对象是数组，则使用指针，并且使用指针常量；

如果数据对象是较大的结构，则使用const指针或者引用，已提高程序的效率。这样可以节省结构所需的时间和空间；

如果数据对象是类对象，则使用const引用（传递类对象参数的标准方式是按照引用传递）；

对于修改函数中数据的函数：

如果数据是内置数据类型，则使用指针

如果数据对象是结构，则使用引用或者指针

如果数据是类对象，则使用引用

## 108、如何设计一个计算仅单个子类的对象个数？

1、为类设计一个static静态变量count作为计数器；

2、类定义结束后初始化count;

3、在构造函数中对count进行+1;

4、设计拷贝构造函数，在进行拷贝构造函数中进行count +1，操作；

5、设计赋值构造函数，在进行赋值函数中对count+1操作；

6、在析构函数中对count进行-1；

## 109、怎么快速定位错误出现的地方?

1、如果是简单的错误，可以直接双击错误列表里的错误项或者生成输出的错误信息中带行号的地方就可以让编辑窗口定位到错误的位置上。

2、对于复杂的模板错误，最好使用生成输出窗口。

## 111、在进行返回值传递时，使用引用传递的注意事项？

1）不能返回局部变量的引用。因为函数返回以后局部变量就会被销毁

2）不能返回函数内部new分配的内存的引用。虽然不存在局部变量的被动销毁问题，可对于这种情况（返回函数内部new分配内存的引用），又面临其它尴尬局面。例如，被函数返回的引用只是作为一个临时变量出现，而没有被赋予一个实际的变量，那么这个引用所指向的空间（由new分配）就无法释放，造成memory leak

3）可以返回类成员的引用，但是最好是const。因为如果其他对象可以获得该属性的非常量的引用，那么对该属性的单纯赋值就会破坏业务规则的完整性。

## 112、说一说strcpy、sprintf与memcpy这三个函数的不同之处

1. 操作对象不同

① strcpy的两个操作对象均为字符串

② 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

③ 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。

1. 执行效率不同

memcpy最高，strcpy次之，sprintf的效率最低。

1. 实现功能不同

① strcpy主要实现字符串变量间的拷贝

② sprintf主要实现其他数据类型格式到字符串的转化

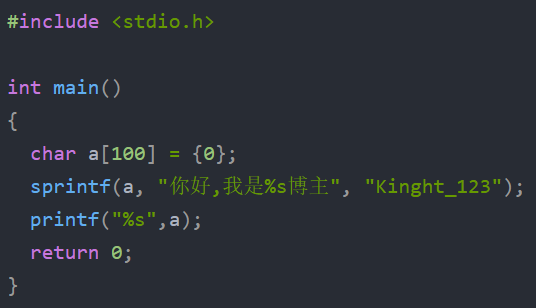
③ memcpy主要是内存块间的拷贝。

4. 头文件不同

strcpy和memcpy在<string.h>头文件中

sprintf在<stdio.h>头文件中





## 115、如何阻止一个类被实例化？有哪些方法？

1. 将类定义为抽象基类
2. 将构造函数声明为private

## 116、如何禁止程序自动生成拷贝构造函数？

可以定义一个base类，在base类中将拷贝构造函数和拷贝赋值函数设置成private,那么派生类中编译器将不会自动生成这两个函数，且由于base类中该函数是私有的，因此，派生类将阻止编译器执行相关的操作。

## 116.2、如何禁止程序使用拷贝构造函数？

① 手动重写，并设置为private

② 手动重写，并设置为delete

③ 定义一个base类，在base类中将拷贝构造函数和拷贝赋值函数设置成private，然后private继承这个base类（效果和delete一样）

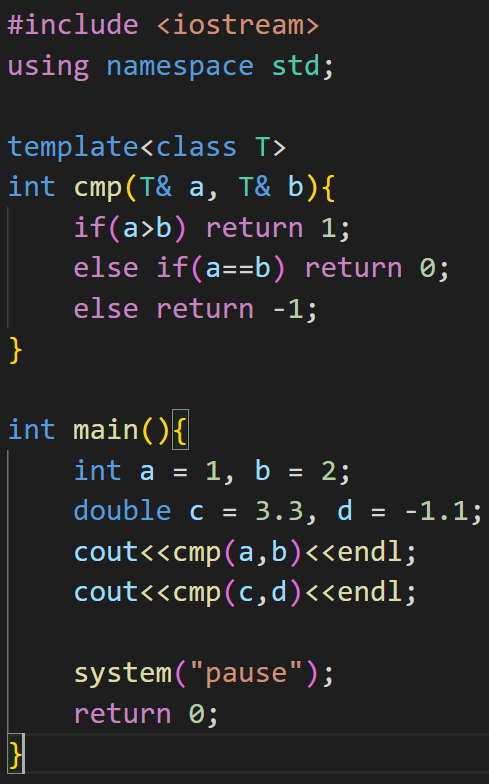
## 117、你知道Debug和Release的区别是什么吗？

1. Debug是调试版本，包含调试信息，所以容量比Release大很多，并且不进行任何优化（优化会使调试复杂化，因为源代码和生成的指令间关系会更复杂），便于程序员调试。Debug模式下生成两个文件，除了.exe或.dll文件外，还有一个.pdb文件，该文件记录了代码中断点等调试信息；
2. Realease是发布版本，不对源代码进行调试，编译时对应用程序的速度进行优化，使得程序在代码大小和运行速度上都是最优的。（调试信息可在单独的PDB文件中生成）。Release模式下生成一个文件.exe或.dll文件。
3. 实际上，Debug 和 Release 并没有本质的界限，他们只是一组编译选项的集合，编译器只是按照预定的选项行动。事实上，我们甚至可以修改这些选项，从而得到优化过的调试版本或是带跟踪语句的发布版本

## 118、main函数的返回值有什么值得考究之处吗？

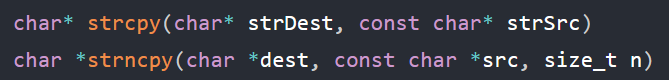
main（）函数返回值类型必须是int，这样返回值才能传递给程序激活者（如操作系统）表示程序正常退出。

## 119、模板会写吗？写一个比较大小的模板函数

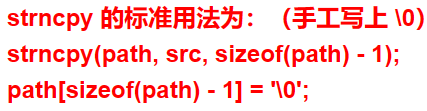


a和b是自定义数据类型的时候要重载比较运算符

## 120、strcpy函数和strncpy函数的区别？哪个函数更安全？



* strcpy函数: 将源[字符串复制](https://so.csdn.net/so/search?q=%E5%AD%97%E7%AC%A6%E4%B8%B2%E5%A4%8D%E5%88%B6&spm=1001.2101.3001.7020" \t "_blank)到目标字符串中，并在目标字符串的末尾添加一个'\0'。如果参数 dest 所指的内存空间不够大，可能会造成缓冲溢出。
* strncpy函数：用来复制源字符串的前n个字符，src 和 dest 所指的内存区域不能重叠，且 dest 必须有足够的空间放置n个字符。



* 如果目标长>指定长>源长，则将源长全部拷贝到目标长，自动补上’\0’到指定长度
* 如果指定长<源长，则将源长中按指定长度拷贝到目标字符串，不包括’\0’

strcpy函数和strncpy函数都不安全，可以使用strncpy\_s()、strcpy\_s()

## 121、static\_cast比C语言中的转换强在哪里？

1. 在编译时进行类型检查，如果类型转换不合法，编译器会发出警告或错误。更加安全；
2. 更直接明显，能够一眼看出是什么类型转换为什么类型，容易找出程序中的错误；可清楚地辨别代码中每个显式的强制转换；可读性更好，能体现程序员的意图

## 122、成员函数里memset(this,0,sizeof(\*this))会发生什么

1. 有时候类里面定义了很多int,char,struct等c语言里的那些类型的变量，我习惯在构造函数中将它们初始化为0，但是一句句的写太麻烦，所以直接就memset(this, 0, sizeof \*this);将整个对象的内存全部置为0。对于这种情形可以很好的工作
2. 类含有虚函数表：这么做会破坏虚函数表，后续对虚函数的调用都将出现异常；
3. 类中含有C++类型的对象：例如，类中定义了一个list的对象，由于在构造函数体的代码执行之前就对list对象完成了列表初始化，假设list在它的构造函数里分配了内存，那么我们这么一做就破坏了list对象的内存

## 123、回调函数的作用？

1. 当发生某种事件时，系统或其他函数将会自动调用你定义的一段函数；
2. 回调函数就相当于一个中断处理函数，由系统在符合你设定的条件时自动调用。为此，你需要做三件事：1，声明；2，定义；3，设置触发条件，就是在你的函数中把你的回调函数名称转化为地址作为一个参数，以便于系统调用；
3. 回调函数就是一个通过函数指针调用的函数。如果你把函数的指针（地址）作为参数传递给另一个函数，当这个指针被用为调用它所指向的函数时，我们就说这是回调函数；
4. 因为可以把调用者与被调用者分开。调用者不关心谁是被调用者，所有它需知道的，只是存在一个具有某种特定原型、某些限制条件（如返回值为int）的被调用函数。

## 124、什么是一致性哈希？

在分布式系统中，常常需要将数据分布到不同的节点上，而且这些节点也可能会动态地加入或离开集群。一致性哈希是一种哈希算法，可以解决节点的动态变化和负载均衡的问题。核心思想是**在移除或者增加一个结点时，能够尽可能小的改变已存在key的映射关系；同时利用虚拟节点进行负载均衡**

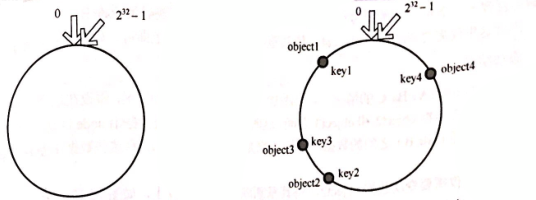
考虑这么一种场景：

我们有三台缓存服务器编号node0、node1、node2，现在有 3000 万个key，希望可以将这些个 key 均匀的缓存到三台机器上。普通的方案是：取模算法hash（key）% N，即：对 key 进行 hash 运算后取模，N 是机器的数量。这样，对 key 进行 hash 后的结果对 3 取模，得到的结果一定是 0、1 或者 2，正好对应服务器node0、node1、node2，存取数据直接找对应的服务器即可，简单粗暴，完全可以解决上述的问题；

这种方法有一定的局限性：**因为在生产环境中根据业务量的大小，调整服务器数量是常有的事，而服务器数量 N 发生变化后hash（key）% N计算的结果也会随之变化，那么之前缓存 key 的数据也会失去作用与意义**

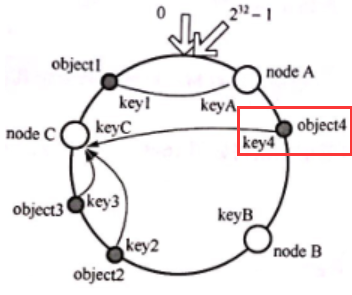
**一致性哈希算法**

将整个哈希值空间组**织成一个虚拟的圆环。**现在有一批的数据object1-object4需要存在服务器上，则可以使用相同的哈希算法对数据进行哈希，其结果必然也在环上，可以沿着顺时针方向寻找，找到一个结点（服务器）则将数据存在这个结点上，这样数据和结点就产生了一对一的关联



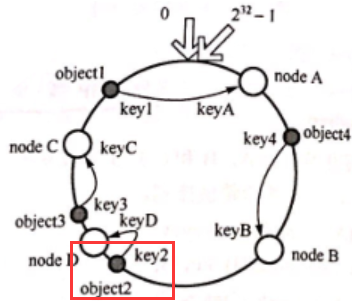
① **移除结点**

如果一台服务器出现问题，如下图中的nodeB，则受影响的是其逆时针方向至下一个结点之间的数据，只需将这些数据映射到它顺时针方向的第一个结点上即可



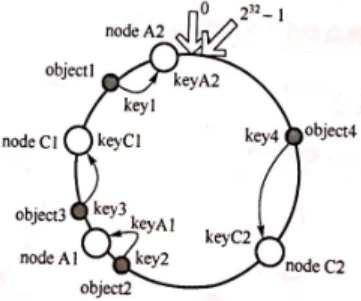
② **添加结点**

如果新增一台服务器nodeD，受影响的是其逆时针方向至下一个结点之间的数据，将这些数据映射到nodeD上即可



③ **虚拟结点**

假设仅有2台服务器：nodeA和nodeC，nodeA映射了1条数据，nodeC映射了3条，这样数据分布是不平衡的。引入虚拟结点，假设结点复制个数为2，则nodeA变成：nodeA1和nodeA2，nodeC变成：nodeC1和nodeC2，映射情况变成如下：



## 125、C++从代码到可执行程序经历了什么？

**（1）预编译（生成.i/.ii文件）**

主要处理源代码文件中的以“#”开头的预编译指令。处理规则见下：

1. 删除所有的#define，展开所有的宏定义。
2. 处理所有的条件预编译指令，如“#if”、“#endif”、“#ifdef”、“#elif”和“#else”。
3. 处理“#include”预编译指令，将文件内容替换到它的位置，这个过程是递归进行的，文件中包含其他文件。
4. 删除所有的注释，“//”和“/\*\*/”。
5. 保留所有的#pragma编译器指令，编译器需要用到他们，如：#pragma once 是为了防止有文件被重 复引用。
6. 添加行号和文件标识，便于编译时编译器产生调试用的行号信息，和编译时产生编译错误或警告是 能够显示行号。

**（2）编译（生成.s文件）**

把预编译之后生成的xxx.i或xxx.ii文件，进行一系列词法分析、语法分析、语义分析及优化后，生成相应 的汇编代码文件。

1. 词法分析：利用类似于“有限状态机”的算法，将源代码程序输入到扫描机中，将其中的字符序列分 割成一系列的记号。
2. 语法分析：语法分析器对由扫描器产生的记号，进行语法分析，产生语法树。由语法分析器输出的 语法树是一种以表达式为节点的树。
3. 语义分析：语法分析器只是完成了对表达式语法层面的分析，语义分析器则对表达式是否有意义进 行判断，其分析的语义是静态语义——在编译期能分期的语义，相对应的动态语义是在运行期才能确定 的语义。
4. 优化：源代码级别的一个优化过程。
5. 目标代码生成：由代码生成器将中间代码转换成目标机器代码，生成一系列的代码序列——汇编语言 表示。
6. 目标代码优化：目标代码优化器对上述的目标机器代码进行优化：寻找合适的寻址方式、使用位移 来替代乘法运算、删除多余的指令等。

**（3）汇编（生成.o/.obj文件）**

将汇编代码转变成机器可以执行的指令(机器码文件)。 汇编器的汇编过程相对于编译器来说更简单，没有复杂的语法，也没有语义，更不需要做指令优化，只是根据汇编指令和机器指令的对照表一一翻译过 来，汇编过程有汇编器as完成。经汇编之后，产生目标文件(与可执行文件格式几乎一样)xxx.o(Linux 下)、xxx.obj(Window下)。

**（4）链接（生成.exe/.out文件）**

将不同的源文件产生的目标文件进行链接，从而形成一个可以执行的程序。链接分为静态链接和动态链接：

**静态链接（linux .a / Windows .lib）：**

函数和数据被编译进一个二进制文件。在使用静态库的情况下，在编译链接可执行文件时，链接器从库中复制这些函数和数据并把它们和应用程序的其它模块组合起来创建最终的可执行文件。

空间浪费：因为每个可执行程序中对所有需要的目标文件都要有一份副本，所以如果多个程序对同一个目标文件都有依赖，会出现同一个目标文件都在内存存在多个副本；

更新困难：每当库函数的代码修改了，这个时候就需要重新进行编译链接形成可执行程序。

运行速度快：但是静态链接的优点就是，在可执行程序中已经具备了所有执行程序所需要的任何东西，在执行的时候运行速度快。

**动态链接(linux .so / Windows .dll)：**

动态链接的基本思想是把程序按照模块拆分成各个相对独立部分，在程序运行时才将它们链接在一起形 成一个完整的程序，而不是像静态链接一样把所有程序模块都链接成一个单独的可执行文件。

共享库：就是即使需要每个程序都依赖同一个库，但是该库不会像静态链接那样在内存中存在多份副本，而是这多个程序在执行时共享同一份副本；

更新方便：更新时只需要替换原来的目标文件，而无需将所有的程序再重新链接一遍。当程序下一次运行时，新版本的目标文件会被自动加载到内存并且链接起来，程序就完成了升级的目标。

性能损耗：因为把链接推迟到了程序运行时，所以每次执行程序都需要进行链接，所以性能会有一定损失。

## 127、友元函数和友元类的基本情况

友元提供了不同类的成员函数之间、类的成员函数和一般函数之间进行数据共享的机制。通过友元，一个不同函数或者另一个类中的成员函数可以访问类中的私有成员和保护成员。友元的正确使用能提高程序的运行效率，但同时也破坏了类的封装性和数据的隐藏性，导致程序可维护性变差。

注意：

(1) 友元关系不能被继承。

(2) 友元关系是单向的，不具有交换性。若类B是类A的友元，类A不一定是类B的友元，要看在类中是否有相应的声明。

(3) 友元关系不具有传递性。若类B是类A的友元，类C是B的友元，类C不一定是类A的友元，同样要看类中是否有相应的申明

## 128、用C语言实现C++的继承



## 129、静态编译与动态编译

**静态编译**

  静态编译是将程序代码和库函数一起编译成一个可执行文件的过程。在静态编译过程中，程序代码和库函数的代码被组合在一起，形成一个独立的可执行文件，该文件可以在任何系统上运行，因为它包含了所有所需的代码和库函数。

优点：

程序在运行时不需要依赖任何外部库函数，因此可以保证程序的稳定性和可移植性。

程序的性能通常比动态编译的程序要高，因为它不需要在运行时加载库函数。

缺点：

程序的体积通常比较大，因为所有的库函数都被编译进了可执行文件中。

如果库函数发生变化，需要重新编译整个程序。

**动态编译**

  动态编译是将程序代码编译成一个中间文件，该文件在程序运行时动态加载所需的库函数。在动态编译的过程中，程序代码被编译成一个中间文件，而不包括库函数的代码。在程序运行时，操作系统动态加载所需的库函数，将它们链接到程序中。

优点：

可执行文件的体积比静态编译的程序要小。

如果库函数发生变化，只需要重新编译库函数，而不需要重新编译整个程序。

缺点：

程序在运行时需要依赖外部库函数，如果某个库函数无法找到或发生错误，程序就会崩溃。

动态编译的程序在运行时需要加载库函数，因此性能可能比静态编译的程序略低。

## 131、介绍一下几种典型的锁

**互斥锁**

一次只能一个线程拥有互斥锁，其他线程只有等待

互斥锁是在抢锁失败的情况下主动放弃CPU进入睡眠状态直到锁的状态改变时再唤醒，而操作系统负责线程调度，为了实现锁的状态发生改变时唤醒阻塞的线程或者进程，需要把锁交给操作系统管理，所以互斥锁在加锁操作时涉及上下文的切换。互斥锁实际的效率还是可以让人接受的，加锁的时间大概100ns左右，而实际上互斥锁的一种可能的实现是先自旋一段时间，当自旋的时间超过阀值之后再将线程投入睡眠中，因此在并发运算中使用互斥锁（每次占用锁的时间很短）的效果可能不亚于使用自旋锁

**条件变量**

互斥锁一个明显的缺点是他只有两种状态：锁定和非锁定。而条件变量通过允许线程阻塞和等待另一个线程发送信号的方法弥补了互斥锁的不足，他常和互斥锁一起使用，以免出现竞态条件。当条件不满足时，线程往往解开相应的互斥锁并阻塞线程然后等待条件发生变化。一旦其他的某个线程改变了条件变量，他将通知相应的条件变量唤醒一个或多个正被此条件变量阻塞的线程。总的来说互斥锁是线程间互斥的机制，条件变量则是同步机制。

**读写锁**

* 多个读者可以同时进行读
* 写者必须互斥（只允许一个写者写，也不能读者写者同时进行）
* 写者优先于读者（一旦有写者，则后续读者必须等待，唤醒时优先考虑写者）

**自旋锁**

如果进线程无法取得锁，进线程不会立刻放弃CPU时间片，而是一直循环尝试获取锁，直到获取为止。如果别的线程长时期占有锁那么自旋就是在浪费CPU做无用功，但是自旋锁一般应用于加锁时间很短的场景，这个时候效率比较高

## 133、为什么不能把所有的函数写成内联函数?

内联函数以代码复杂为代价，它以省去函数调用的开销来提高执行效率。所以一方面如果内联函数体内代码执行时间相比函数调用开销较大，则没有太大的意义；另一方面每一处内联函数的调用都要复制代码，消耗更多的内存空间，因此以下情况不宜使用内联函数：

* 函数体内的代码比较长，将导致内存消耗代价
* 函数体内有循环，函数执行时间要比函数调用开销大
* 递归函数

实际中，编译器不一定会内联所有声明为内联的函数。编译器会根据一些条件，如函数的复杂度、调用频率等来决定是否进行内联。

## 134、为什么C++没有垃圾回收机制？这点跟Java不太一样。

* 首先，实现一个垃圾回收器会带来额外的空间和时间开销。你需要开辟一定的空间保存指针的引用计数和对他们进行标记mark。然后需要单独开辟一个线程在空闲的时候进行free操作。
* 垃圾回收会使得C++不适合进行很多底层的操作。

**内存管理**

## 1、类的对象存储空间？

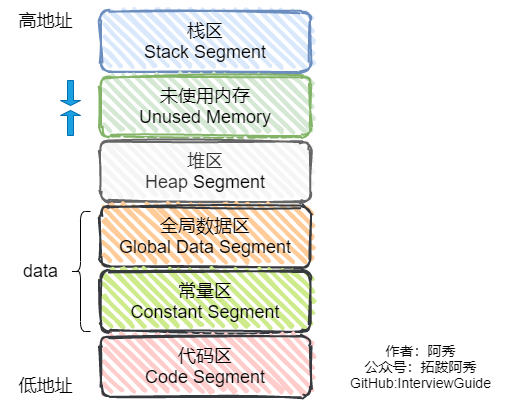
* 非静态成员的数据类型大小之和。
* 编译器加入的额外成员变量（如指向虚函数表的指针）。
* 为了边缘对齐优化加入的padding。

空类(无非静态数据成员)的对象的size为1, 当作为基类时, size为0.

（这里的size为0不是说当一个类是另一个类的基类时，对他使用sizeof会得到0，而是指的是他的size不会加到子类的size中去）

带有虚函数的类的对象中还包含一个指向虚函数表的指针，大小为8或4个字节

## 2、简要说明C++的内存分区



**全局/静态存储区**：全局变量和静态变量被分配到同一块内存中，在以前的C语言中，全局变量和静态变量又分为初始化的和未初始化的，在C++里面没有这个区分了，它们共同占用同一块内存区，在该区定义的变量若没有初始化，则会被自动初始化，例如int型变量自动初始为0

## 3、什么是内存池，如何实现

内存池（Memory Pool） 是一种**内存分配**方式。通常我们习惯直接使用new、malloc 等申请内存，这样做的缺点在于：由于所申请内存块的大小不定，当频繁使用时会造成大量的内存碎片并进而降低性能。内存池则是在真正使用内存之前，先申请分配一定数量的、大小相等(一般情况下)的内存块留作备用。当有新的内存需求时，就从内存池中分出一部分内存块， 若内存块不够再继续申请新的内存。这样做的一个显著优点是尽量避免了内存碎片，使得内存分配效率得到提升

## 5、C++中类的数据成员和成员函数内存分布情况

① 静态数据成员在全局数据区

② 非静态数据成员在和对象存放的位置一致（堆/栈）

③ 无论是否静态，成员函数都存放在代码区

## 6、关于this指针你知道什么？

* this指针是类的指针，指向对象的首地址。
* this指针只能在非静态成员函数中使用，在全局函数、静态成员函数中都不能用this。
* this指针只有在成员函数中才有定义，且存储位置会因编译器不同有不同存储位置。

func的原型在编译器看来应该是：

int func(A \* const this, …);

## 7、几个this指针的易混问题

**A. this指针是什么时候创建的？**

this在成员函数的开始执行前构造，在成员的执行结束后清除

### B. this指针存放在何处？堆、栈、全局变量，还是其他？

this指针会因编译器不同而有不同的放置位置。可能是栈，也可能是寄存器，甚至全局变量。大多数编译器通过ecx（寄数寄存器）寄存器传递this指针。在汇编级别里面，一个值只会以3种形式出现：立即数、寄存器值和内存变量值。不是存放在寄存器就是存放在内存中，它们并不是和高级语言变量对应的。

this为右值，无法通过&符号获取地址

### F.每个类编译后，是否创建一个类中函数表保存函数指针，以便用来调用函数？

不会。函数都放在代码区，只是用this指针指向不同的对象，从而确保不同对象之间调用相同的函数可以互不干扰

## 8、 内存泄漏的后果？如何监测？解决方法？

**1) 内存泄漏**

内存泄漏是指由于疏忽或错误造成了程序未能释放掉不再使用的内存的情况。内存泄漏并非指内存在物理上消失，而是应用程序分配某段内存后，由于设计错误，失去了对该段内存的控制；

**2) 后果**

只发生一次小的内存泄漏可能不被注意，但泄漏大量内存的程序将会出现各种证照：性能下降到内存逐渐用完，导致另一个程序失败；

**3) 如何排除**

**①** Windows使用工具软件BoundsChecker，BoundsChecker是一个运行时错误检测工具，它主要定位程序运行时期发生的各种错误；

② 调试运行DEBUG版程序，运用以下技术：CRT(C run-time libraries)、运行时函数调用堆栈、内存泄漏时提示的内存分配序号(集成开发环境OUTPUT窗口)，综合分析内存泄漏的原因，排除内存泄漏。

Linux使用Valgrind

**4) 解决方法**

智能指针。

**5) 使用CRT库检查、定位内存泄漏**

头文件#include <crtdbg.h>

检查方法：在main函数最后面一行，加上一句\_CrtDumpMemoryLeaks()。调试程序，自然关闭程序让其退出，查看输出：

输出这样的格式{453}normal block at 0x02432CA8,868 bytes long

被{}包围的453就是我们需要的内存泄漏定位值，868 bytes long就是说这个地方有868字节内存没有释放。

定位代码位置

在main函数第一行加上\_CrtSetBreakAlloc(453);意思就是在申请453这块内存的位置中断。然后调试程序，程序中断了，查看调用堆栈。

## 9、在成员函数中调用delete this会出现什么问题？对象还可以使用吗？

delete this是未定义的行为

如果是在栈上声明的对象，有些编译器会直接down掉。

如果是在堆上声明的对象，可以正常编译，但之后不能再使用涉及this的内容

在类对象的内存空间中，只有数据成员和虚函数表指针，并不包含代码内容，类的成员函数单独放在代码段中。在调用成员函数时，隐含传递一个this指针，让成员函数知道当前是哪个对象在调用它。当调用delete this时，类对象的内存空间被释放。在delete this之后进行的其他任何函数调用，只要不涉及到this指针的内容，都能够正常运行。一旦涉及到this指针，如操作数据成员，调用虚函数等，就会出现不可预期的问题。

## 10、为什么是不可预期的问题？

这个问题牵涉到操作系统的内存管理策略。delete this释放了类对象的内存空间，但是内存空间却并不是马上被回收到系统中，可能是缓冲或者其他什么原因，导致这段内存空间暂时并没有被系统收回。此时这段内存是可以访问的，你可以加上100，加上200，但是其中的值却是不确定的。当你获取数据成员，可能得到的是一串很长的未初始化的随机数；访问虚函数表，指针无效的可能性非常高，造成系统崩溃。

## 11、 如果在类的析构函数中调用delete this，会发生什么？

会导致堆栈溢出。原因很简单，delete的本质是“为将被释放的内存调用一个或多个析构函数，然后释放内存”。显然，delete this会去调用本对象的析构函数，而析构函数中又调用delete this，形成无限递归，造成堆栈溢出，系统崩溃。

## 14、this指针调用成员变量时，堆栈会发生什么变化？

调用非静态成员函数时，编译器会自动将对象的地址传给作为隐含参数传递给函数，这个隐含参数就是this指针。This指针首先入栈，然后成员函数的参数从右向左进行入栈，最后函数返回地址入栈。

# C++11新标准

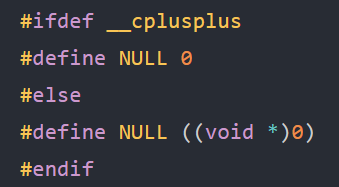
## 1、C++ 11有哪些新特性？

* nullptr替代 NULL
* 引入了 auto 和 decltype 这两个关键字实现了类型推导
* 基于范围的 for 循环for(auto& i : res){}
* 类和结构体的中初始化列表
* Lambda 表达式（匿名函数）
* std::forward\_list（单向链表）
* 右值引用和move语义
* 智能指针
* ……

## 2、C++中NULL和nullptr区别

NULL来自C语言，一般由宏定义实现，而 nullptr 则是C++11的新增关键字，是一个nullptr\_t类型的对象。

在C语言中，NULL被定义为(void\*)0。而在C++中，NULL则被定义为整数0。编译器一般对其实际定义如下：



**NULL无法与整数的0区分。当有函数重载，在传入NULL参数时，会把NULL当做整数0来看，不能成功调用参数是指针的函数。nullptr在C++11被引入用于解决这一问题，nullptr可以明确区分整型和指针类型，能够根据环境自动转换成相应的指针类型，但不会被转换为任何整型，所以不会造成参数传递错误。**

nullptr的缺点：如果存在对不同指针类型的函数重载，此时如果传入nullptr指针则仍然存在无法区分应实际调用哪个函数，这种情况下必须显示的指明参数类型。

## 3、auto、decltype和decltype(auto)的用法

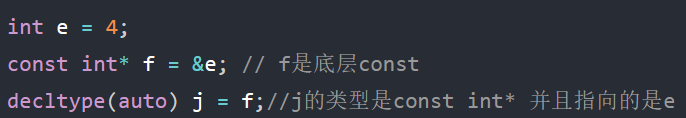
**（1）auto**

**auto 让编译器通过初始值来进行类型推演，从而获得定义变量的类型，所以说 auto 定义的变量必须有初始值。**

**（2）decltype**

如果希望从表达式或函数的返回值中推断出要定义变量的类型，但不想通过初始化一个变量的方式，可以使用decltype。**它的作用是选择并返回操作数的数据类型。在此过程中，编译器只是分析表达式并得到它的类型，却不进行实际的计算表达式的值。**

**（3）decltype(auto) （C++14）**



使用时，会将“=”号右边的表达式替换掉auto，再根据decltype的语法规则来确定类型。

## 4、lambda函数

Lambda函数，亦称为Lambda表达式、匿名函数等，是一种函数对象，Lambda函数可以让函数像普通变量一样进行赋值、传递、函数返回等操作。C++中的Lambda函数经常用来解决如下问题：

① 使得程序更加简洁，尤其对于一次性使用的函数。

② 使得函数可以自由流动，就像变量一样，这给函数式编程模式奠定了语法基础。

## ****语法逻辑****



**captrue list**: 外部变量列表

**params list**: 形参列表

**mutable**: 是否可以修改外部变量。

默认情况下，Lambda函数总是一个const函数，mutable可以取消其常量性。

在使用mutable时，形参列表不可省略

**exception**: 异常设定

**return type**: 返回类型

可以不需要声明返回值，此时返回类型相当于使用decltype根据返回值推断得到

**function body**: 函数体



在上述代码中，从左到右逐个解释:

a和b是从外部可见作用域中捕获的变量，默认它们在Lambda函数内部是只读的。

()圆括号内是Lambda函数的形参列表，此处为空，但不可省略。

Lambda函数的形参列表一般用于跟容器类结合的时候。

->int 是Lambda函数的返回值类型，可以省略。

{}内部就是Lambda函数的具体代码实现。

请注意，最右边的圆括号 () 是对Lambda函数调用的符号，不是Lamdba函数本身

外部变量列表的特殊语法：

① 以传值模式（只读）捕获所有外部变量：

[=]()->int{...};

② 以引用模式（读写）捕获所有外部变量

[&]()->int{...};

## 5、右值引用和move语义

**5.1. 引言**

C++11中引入了右值引用和移动语义，可以避免无谓的复制，提高了程序性能

① **提高程序性能**：传统的拷贝构造函数会对对象进行深拷贝，这会导致内存的频繁分配和释放，降低程序的性能。而移动构造函数利用右值引用，可以将对象的资源所有权转移，避免了不必要的内存分配和释放，从而提高程序性能。

② **支持移动语义**：移动语义是指将一个对象的资源所有权转移给另一个对象，而不是进行拷贝。移动语义在实现容器类时特别有用，可以避免不必要的拷贝和内存分配，提高程序的效率。

③ **支持完美转发**：右值引用还可以用于完美转发，即将一个函数的参数转发给另一个函数，同时保留参数的值类别（左值或右值）。这在模板编程中特别有用，可以避免不必要的拷贝和内存分配，提高程序的效率。

④ **支持移动语义的STL容器**：C++11中的STL容器（例如vector、list等）都支持移动语义，可以提高程序的效率。

左值是表达式结束后仍然存在的持久对象，右值是指表达式结束时就不存在的临时对象。区分左值和右值的便捷方法是看能不能对表达式取地址，如果能则为左值，否则为右值

将亡值是C++11新增的、与右值引用相关的表达式，比如：

右值引用（例如：int&& x = 42;）、返回右值引用的函数返回值（普通函数、类型转换函数）

std::move返回的右值引用（例如：int x = 42; int&& y = std::move(x);）

### 5.2. 不同值的辨析

C++11中的所有的值必将属于左值、将亡值、纯右值三者之一，将亡值和纯右值都属于右值。

如何区分将亡值和纯右值：

① 将亡值是指一个对象即将被移动，但它的生命周期仍然可以延长的值。而纯右值是指一个临时对象，它的生命周期只在当前表达式中存在，不能被延长。

② 在 C++11 中，可以通过std::move函数将一个对象转换为将亡值。在某些情况下，编译器也会自动将一个对象识别为将亡值，**例如返回一个std::vector对象时**。而纯右值则通常是字面量、临时对象或者返回值表达式。

③ 区分将亡值和纯右值的关键在于它们的生命周期。如果一个对象的生命周期可以被延长，那么它就是将亡值；如果一个对象的生命周期只在当前表达式中存在，那么它就是纯右值。

区分表达式的左右值属性：**如果可对表达式用&符取址，则为左值，否则为右值**

左值 lvalue 是有标识符、可以取地址的表达式，最常见的情况有：

① 变量、函数或数据成员的名字

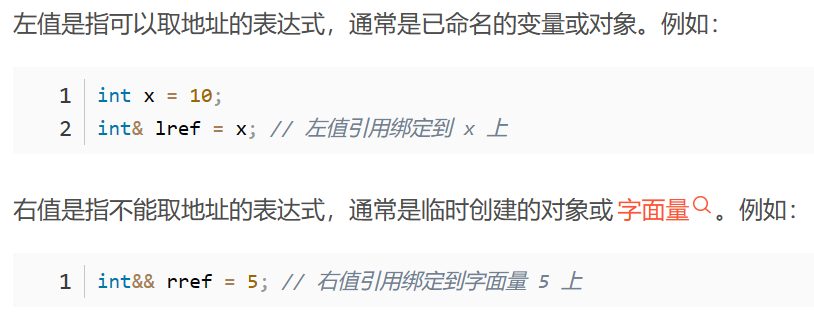
② 返回左值引用的表达式，如 ++x、x = 1、cout << ’ ’

纯右值 prvalue 是没有标识符、不可以取地址的表达式，一般也称之为“临时对象”，最常见的情况有：

① 返回非引用类型的表达式，如 x++、x + 1、std::make\_shared(42)

② 字符串字面量和除字符串字面量之外的字面量，如 42、true

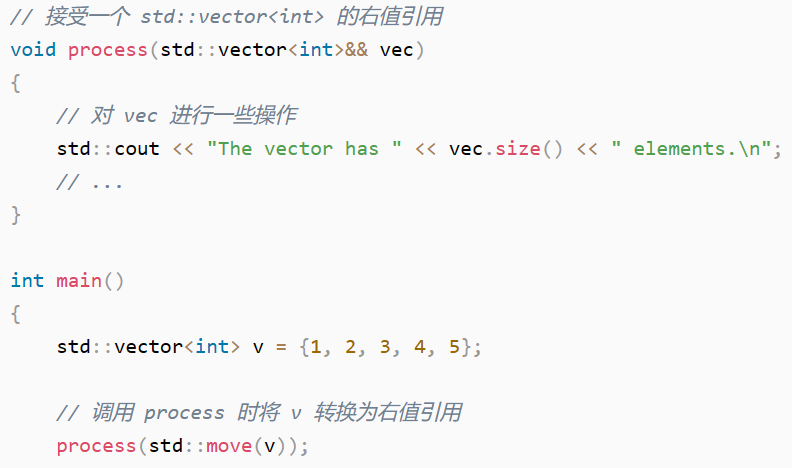
### 5.4. 左值引用和右值引用



左值引用和右值引用的另一个区别是它们对于移动语义和完美转发的支持。右值引用通常用于实现移动语义和完美转发，而左值引用则用于传递和修改**已命名**的对象。



### 5.5. 右值引用优化性能



process函数接受一个std::vector<int>的右值引用，这意味着传递给函数的参数是一个临时对象或者可以被移动的对象，而不是一个具有所有权的对象。在函数内部，我们可以对这个对象进行一些操作，而无需进行深拷贝。此外，我们使用了 std::move函数将 v 转换为右值引用，这意味着 v 的所有权被移动到了process函数内部

### 5.8. forward完美转发

forward是一个模板函数，主要用于将参数进行完美转发，即将参数以原本的类型和引用类型传递给另一个函数，避免因为多次拷贝而引起的性能问题。

forward完美转发实现了**参数在传递过程中保持其值属性的功能**，即左值传递之后仍然是左值，右值传递之后仍然是右值。



### 5.9. emplace\_back

与push\_back不同的是，emplace\_back可以直接在vector中构造一个新元素，而不需要先创建一个对象再插入。这个新元素是通过参数列表中的参数来构造的，而不是通过复制或移动现有对象来构造的。因此，emplace\_back比push\_back更高效，因为它可以避免不必要的对象构造和复制。

## 6、智能指针的原理、常用的智能指针及实现

智能指针是一个类，用来存储指向动态分配对象的指针，负责自动释放动态分配的对象，防止堆内存泄漏。动态分配的资源，交给一个类对象去管理，当类对象声明周期结束时，自动调用析构函数释放资源

**常用的智能指针**

**(1) shared\_ptr**

shared\_ptr使用**引用计数，每一个shared\_ptr的拷贝都指向相同的内存。每使用它一次，内部的引用计数加1，每析构一次，内部的引用计数减1，减为0时，释放所指向的堆内存。shared\_ptr内部的引用计数是安全的，但是对象的读取需要加锁**。

**shared\_ptr支持拷贝构造和拷贝赋值。**

**在管理动态数组的时候，shared\_ptr需要指定删除器。删除器不是shared\_ptr类型的组成部分**。也就是说，两个shared\_ptr，就算sp1和sp2有着不同的删除器，但只要两者的类型是一致的，都可以被放入vector<shared\_ptr >类型的同一容器里

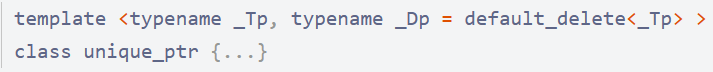
可以使用make\_shared创建，C++11后就支持。

成员函数：

* s.get()：返回shared\_ptr中保存的裸指针；
* s.reset(…)：重置shared\_ptr；
* s.use\_count()：返回shared\_ptr的**强引用计数**；
* s.unique()：若use\_count()为1，返回true，否则返回false。

**(2) unique\_ptr**

unique\_ptr对象**始终是关联的原始指针的唯一所有者，实现了独享所有权的语义**。一个非空的unique\_ptr总是拥有它所指向的资源。**转移一个unique\_ptr将会把所有权也从源指针转移给目标指针(源指针被置空)。拷贝一个unique\_ptr将不被允许**



删除器的作用就是规定：**当unique\_ptr对象被销毁时，在其析构函数中释放关联的原始指针的方式**。**删除器是unique\_ptr类型的组成部分，可以是普通函数指针或lambda表达式。注意，当指定删除器时需要同时指定其类型，即\_Dp不可省略，可通过decltype获得**。

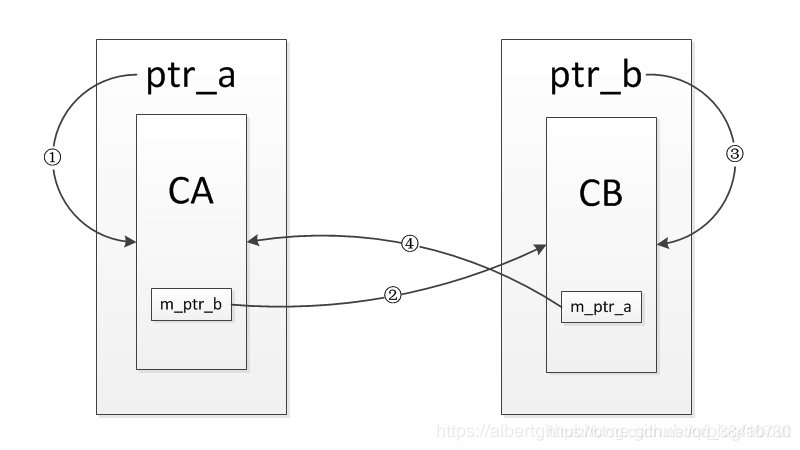
可以使用make\_unique创建，C++14后才支持。

**(3) weak\_ptr**

weak\_ptr 是一种智能指针，通常不单独使用，只能和 shared\_ptr 类型指针搭配使用，可以视为 shared\_ptr 指针的一种辅助工具。借助 weak\_ptr 类型指针可以获取 shared\_ptr 指针的一些状态信息，比如有多少指向相同的 shared\_ptr 指针、通过expired()判断shared\_ptr 指针指向的堆内存是否已经被释放等等，还可以解决shared\_ptr 循环引用的问题。

### 循环引用

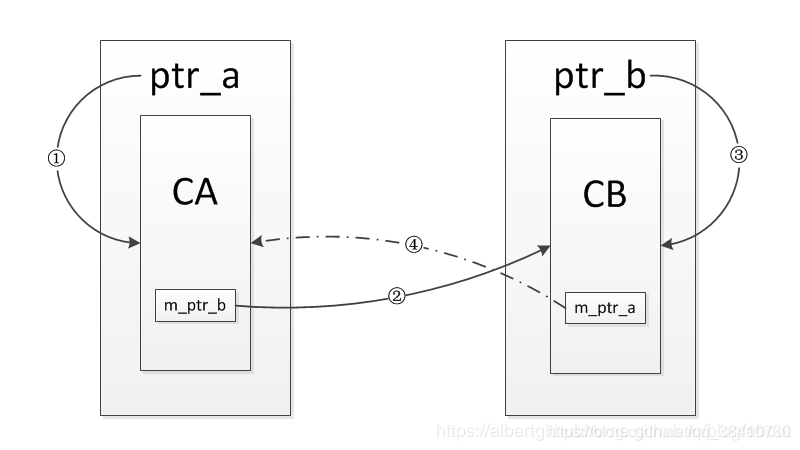
在[shared\_ptr](https://so.csdn.net/so/search?q=shared_ptr&spm=1001.2101.3001.7020)的使用过程中，当强引用计数为0是，就会释放所指向的堆内存。如果两个shared\_ptr互相引用，那么它们就永远无法被释放了。



只有①③被释放，②④没有，导致内部对象一直不能析构。

#### 解决循环引用

**将两个类中的一个成员变量改为weak\_ptr对象**



④这条引用是通过weak\_ptr建立的，并不会增加引用计数。也就是说，CA的对象只有一个引用计数，而CB的对象只有2个引用计数，当main函数返回时，对象ptr\_a和ptr\_b被销毁，也就是①、③两条引用会被断开，此时CA对象的引用计数会减为0，对象被销毁，其内部的m\_ptr\_b成员变量也会被析构，导致CB对象的引用计数会减为0，对象被销毁，进而解决了引用成环的问题。

#### weak\_ptr的原理

weak\_ptr是为了配合shared\_ptr而引入的一种智能指针，它指向一个由shared\_ptr管理的对象而不影响所指对象的生命周期，也就是，将一个weak\_ptr绑定到一个shared\_ptr不会改变shared\_ptr的引用计数。不论是否有weak\_ptr指向，一旦最后一个指向对象的shared\_ptr被销毁，对象就会被释放。从这个角度看，weak\_ptr更像是shared\_ptr的一个助手而不是智能指针。

#### weak\_ptr的使用

1. 通过shared\_ptr直接初始化，也可以通过隐式转换来构造；
2. 允许移动构造，也允许拷贝构造。

user\_count()：返回weak\_ptr的强引用计数；

reset(…)：重置weak\_ptr。

expired()：判断所指对象是否已被销毁

lock()：返回一个指向共享对象的shared\_ptr。**weak\_ptr并没有重载operator ->和operator \*操作符，因此必须要通过lock才能访问原始对象。**

**(4) auto\_ptr**

成员函数：

get：返回所指的对象的内存地址

reset：重新设置auto\_ptr指向的对象，类似于赋值操作，但是赋值操作不允许将一个普通指针直接赋值给auto\_ptr，而reset允许

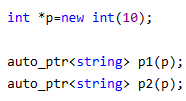
release：返回auto\_ptr指向的那个对象的内存地址，并且释放这个对象的所有权

## auto\_ptr的缺陷

创建两个auto\_ptr对象p1、p2，并将p2指向p1的同一块内存空间，会导致函数结束时释放这块空间两次。为了解决这个问题，auto\_ptr类模板制定了一个”所有权(ownership)“的概念。对于一个特定的对象，只允许有一个auto\_ptr拥有它。在拷贝构造和赋值的时候，发生所有权的转移，将对象的所有权从旧的auto\_ptr转移给新的auto\_ptr（此时旧的auto\_ptr无法再访问这个内存空间），并将就得auto\_ptr置为nullptr，这样在释放内存空间时，不会出现好几个auto\_ptr试图释放同一块内存空间得情况。

但是还有一些问题：

① **两个auto\_ptr对象拥有同一个对象**



可以用release来解决

② **auto\_ptr对象不要放在容器中**

STL容器中的元素经常要支持拷贝、赋值操作，在这过程中auto\_ptr会传递所有权，所以不能在STL中使用。

③ **警惕auto\_ptr作为参数**

按值传递时，函数的调用过程中在函数的作用域中会产生一个局部对象来接收传入的auto\_ptr(拷贝构造),这样传入的实参得到了原对象的所有权，而该对象在函数退出时会被局部的auto\_ptr删除

参数为引用或指针,虽然不会存在上述的拷贝过程，但是我们并不知道函数对传入的auto\_ptr做了什么，如果其中的某种操作使其失去了对象的所有权，那么还是会导致致命的执行错误

# STL模板库

## 1、什么是STL？

STL(Standard Template Library),即标准模板库。

C++ STL从广义来讲包括了三类：算法，容器和迭代器。

* 算法包括排序，复制等常用算法，以及不同容器特定的算法。
* 容器就是数据的存放形式，包括序列式容器和关联式容器，序列式容器就是list，vector等，关联式容器就是set，map等。
* 迭代器就是在不暴露容器内部结构的情况下对容器的遍历。

## 2、解释一下什么是trivial destructor

一般是指用户没有自定义析构函数，而由系统生成的 “无关痛痒”的析构函数。

用户自定义了析构函数，则称之为“non-trivial destructor”，这种析构函数**如果申请了新的空间一定要显式的释放，否则会造成内存泄露**

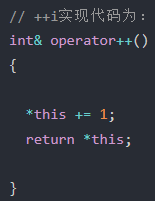
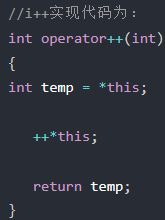
如果能够对是否trivial进行区分，可以采用内存处理函数memcpy()、malloc()等更加高效的完成相关操作，提升效率。

## 3、使用智能指针管理内存资源，RAII是怎么回事？

RAII全称是“Resource Acquisition is Initialization”，直译过来是“资源获取即初始化”，也就是说在构造函数中申请分配资源，在析构函数中释放资源。

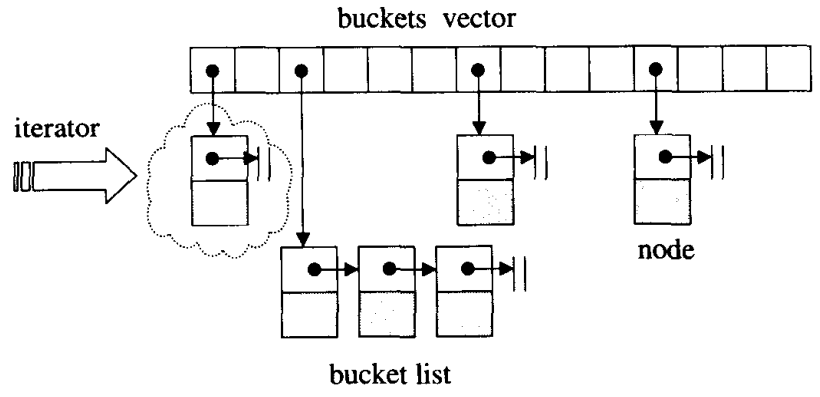
## 4、迭代器：++it、it++哪个好，为什么

前置++返回一个引用，后置++返回一个对象。前置不会产生临时对象，后置必须产生临时对象，临时对象会导致效率降低，所以一般是++it比较好。

## 6、STL中hashtable的实现？

STL中的hashtable使用的是**开链法**解决hash冲突问题



Hashtable的bucket聚合体本身使用vector进行存储

在hashtable设计bucket的数量上，其内置了28个质数[53, 97, 193,...,429496729]，在创建hashtable时，会根据存入的元素个数选择大于等于元素个数的质数作为hashtable的容量（vector的长度），其中每个bucket所维护的linked-list长度也等于hashtable的容量。如果插入hashtable的元素个数超过了bucket的容量，就要进行重建table操作，即**找出下一个质数，创建新的buckets vector**，重新计算元素在新hashtable的位置。

hashtable的迭代器只提供前进操作，不提供后退操作

## 7、简单说一下traits技法

**iterator\_traits**

**关注迭代器的特性，能够方便的让外界获取以下5种型别：**

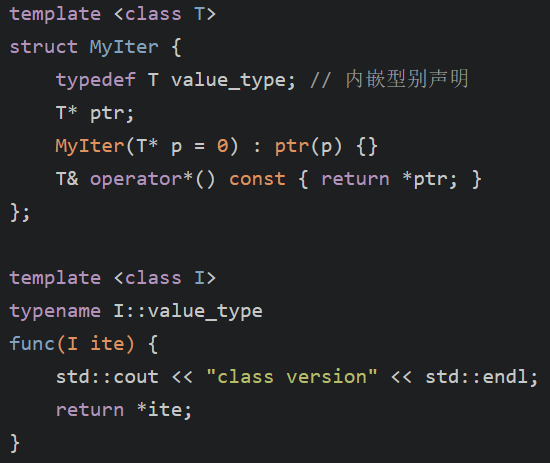
* **value\_type：迭代器所指对象的型别**
* **difference\_type：两个迭代器之间的距离**
* **pointer：迭代器所指向的对象**
* **reference：迭代器所指对象的类型的引用**
* **iterator\_category：标识迭代器的移动特性和可以对迭代器执行的操作**

**以value\_type为例：**

[**STL**](https://so.csdn.net/so/search?q=STL&spm=1001.2101.3001.7020)编程中，容器和算法是独立设计的，即数据结构和算法是独立设计的，连接容器和算法的桥梁就是迭代器。

**迭代器能够顺序访问容器中每个元素的方法，使用该方法不能暴露容器内部的表达方式。而类型萃取技术就是为了要解决和 iterator 有关的问题的。**

算法中使用迭代器时，需要知道[迭代器](https://so.csdn.net/so/search?q=%E8%BF%AD%E4%BB%A3%E5%99%A8&spm=1001.2101.3001.7020)所指元素的类型是什么，可以通过**template 参数推导**和**声明内嵌型别**来实现：



但是还存在问题：无法接收原生指针。因为原生指针根本就没有 value\_type 这个内嵌类型。

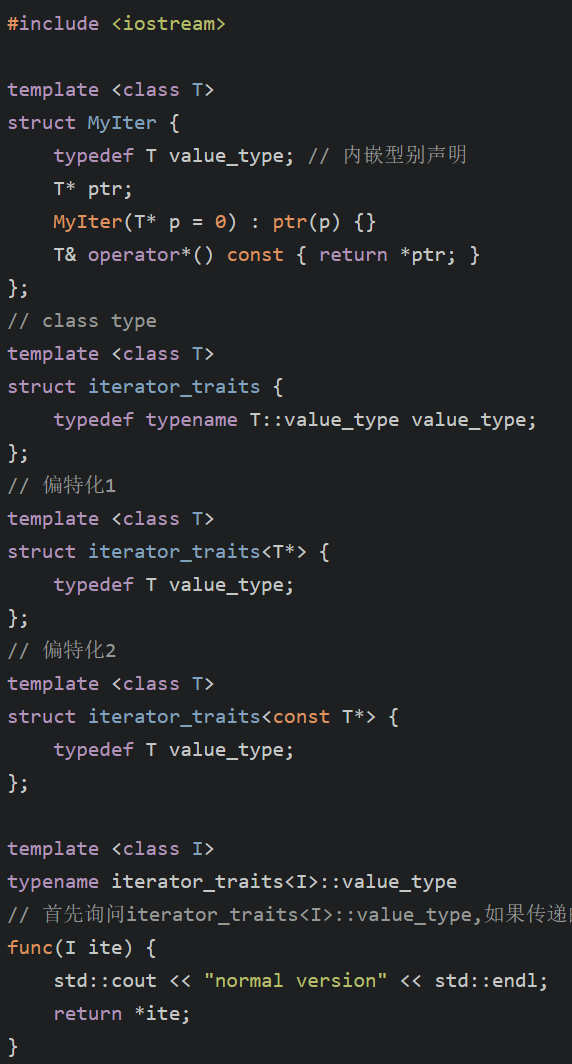
解决方法：

① 函数偏特化

需要对每个可调用函数都进行重载，重复冗长

② **加入中间层iterator\_traits**

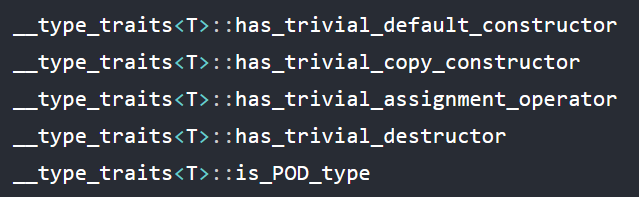
利用一个中间层iterator\_traits固定了func的形式，使得重复的代码大量减少，唯一要做的就是稍稍特化一下 iterator\_tartis 使其支持 pointer 和 const pointer



可以理解为利用中间层查询传入的迭代器对象类型是自定义类型指针还是原生指针

**type\_traits**

**关注类型的特性**。支持以下5种类型的判断：



判断是否有各种构造函数。如果没有，可以采取直接操作内存的方式提高效率

## 8、STL的两级空间配置器

① 为什么需要二级空间配置器？

频繁的在堆开辟释放内存，则就会**在堆上造成很多外部碎片**，浪费了内存空间

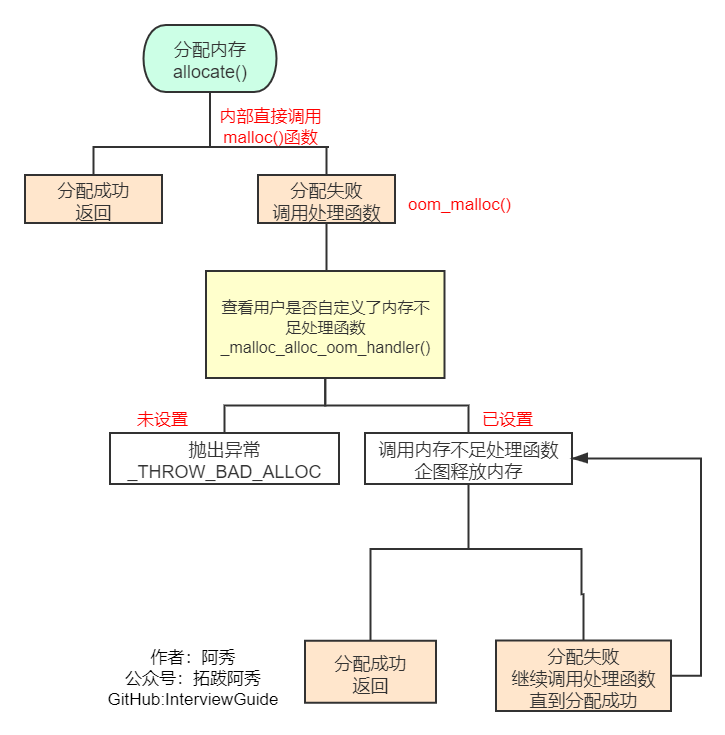
**二级空间配置器**：

**当开辟内存<=128bytes时，即视为开辟小块内存，则调用二级空间配置器。**

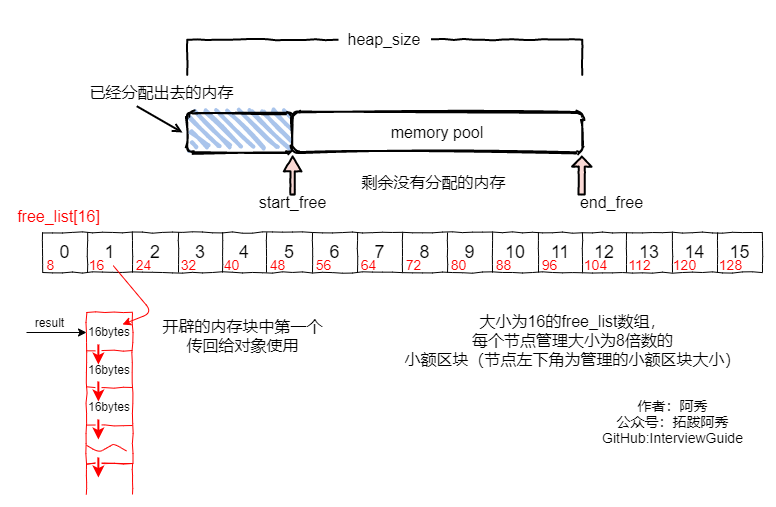
**如果大于128字节再转去一级配置器**

### 一级配置器

其中重要的函数就是allocate、deallocate、reallocate 。 一级空间配置器是以malloc()，free()，realloc()等C函数执行实际的内存配置 。



### 二级配置器



使用了**内存池**：先申请一块比较大的内存当做备用，当需要内存时，直接去内存池中拿内存就好了，当内存池中的空间不够时，再向内存中取，当用户不用时，直接还回内存池即可。避免了频繁的向系统申请小块内存所造成的效率低，内存碎片以及额外浪费的问题。

维护16条链表，分别是0-15号链表，最小8字节，以8字节逐渐递增，最大128字节，你传入一个字节参数，表示你需要多大的内存，会自动帮你校对到第几号链表（如需要13bytes空间，我们会给它分配16bytes大小），在找到第n个链表后查看链表是否为空：

① 如果不为空

直接从对应的free\_list中拔出以供使用，将已经拨出的指针向后移动一位。

② 如果为空

先看其内存池是不是空时，如果内存池不为空：（1）先检验它剩余空间是否够20个节点大小（即所需内存大小(提升后) \* 20），若足够则直接从内存池中拿出20个节点大小空间，将其中一个分配给用户使用，另外19个当作自由链表中的区块挂在相应的free\_list下，这样下次再有相同大小的内存需求时，可直接拨出。 （2）如果不够20个节点大小，则看它是否能满足1个节点大小，如果够的话则直接拿出一个分配给用户，然后从剩余的空间中分配**尽可能多**的节点挂在相应的free\_list中。 （3）如果连一个节点内存都不能满足的话，则将内存池中剩余的空间挂在相应的free\_list中（找到相应的free\_list），然后再给内存池申请内存，转到3。

3、内存池为空，申请内存 此时二级空间配置器会使用malloc()从heap上申请内存，（一次所申请的内存大小为2 \* 所需节点内存大小（提升后）\* 20 + 一段额外空间），申请40块，一半拿来用，一半放内存池中。 4、malloc没有成功 在第三种情况下，如果malloc()失败了，说明heap上没有足够空间分配给我们了，这时，二级空间配置器会从比所需节点空间大的free\_list中一一搜索，从比它所需节点空间大的free\_list中拔除一个节点来使用。如果这也没找到，说明比其大的free\_list中都没有自由区块了，那就要调用一级适配器了

## 9、 vector与list的区别与应用？怎么找某vector或者list的倒数第二个元素

### vector

拥有一段连续的内存空间，并且起始地址不变。因此能高效的进行随机存取，时间复杂度为o(1);但因为内存空间是连续的，所以在进行插入和删除操作时，会造成内存块的拷贝，平均时间复杂度为o(n)。当数组中内存空间不够时，会重新申请一块内存空间并进行内存拷贝。在对vector进行插入和删除后，会导致迭代器失效。

### list

 list是由双向链表实现的，因此内存空间是不连续的。只能通过指针访问数据，所以list的随机存取非常没有效率，时间复杂度为o(n)。但由于链表的特点，能高效地进行插入和删除，时间复杂度为o(1)。在对list进行插入和删除后，迭代器还可以继续使用。

list不提供随机访问，所以不能用下标直接访问到某个位置的元素。

找倒数第二个元素：

vector：v[v.size()-2];

list: auto it = l.rbegin(); \*(++it);

## 10、vector的扩容

1、空的vector对象，size()和capacity()都为0

2、当空间大小不足时，新分配的空间大小为原空间大小的2倍/1.5倍。

3、使用reserve()预先分配一块内存后，在空间未满的情况下，不会引起重新分配，从而提升了效率。

4、当reserve()分配的空间比原空间小时，是不会引起重新分配的。

5、resize()函数只改变容器的元素数目，未改变容器大小。

6、用reserve(size\_type)只是扩大capacity值，这些内存空间可能还是“野”的，如果此时使用“[ ]”来访问，则可能会越界。而resize(size\_type new\_size)会真正使容器具有new\_size个对象。

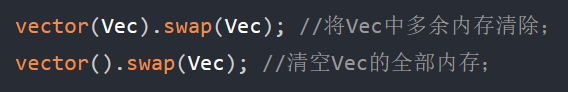
不同的编译器，vector有不同的扩容大小。在vs下是1.5倍，在GCC下是2倍；

采用采用成倍方式扩容，可以保证常数的时间复杂度，而增加指定大小的容量只能达到O(n)的时间复杂度，因此，使用成倍的方式扩容。

## 11、Vector如何释放空间?

vector的内存占用空间只增不减，比如你首先分配了10,000个字节，然后erase掉后面9,999个，留下一个有效元素，但是内存占用仍为10,000个。所有内存空间是在vector析构时候才能被系统回收。

可以用swap()来帮助你释放多余内存或者清空全部内存：



## 12、容器内部删除一个元素

1. 序列式容器，比如vector、deque

erase迭代器不仅使所指向被删除的迭代器失效，而且使被删元素之后的所有迭代器失效(list除外)，所以不能使用erase(it++)的方式，但是erase的返回值是下一个有效迭代器；

It = c.erase(it);

1. 关联式容器，比如map、set、multimap、multiset等

erase迭代器只是被删除元素的迭代器失效，但是返回值是void，所以要采用erase(it++)的方式删除迭代器；

c.erase(it++)

## 13、STL迭代器如何实现

迭代器可以在不了解容器内部原理的情况下遍历容器。迭代器一个最重要的作用就是作为容器与算法的粘合剂。

迭代器内部必须保存一个与容器相关联的指针，然后重载各种运算操作来遍历，其中最重要的是\*运算符与->运算符，以及++、--等可能需要重载的运算符重载。这和C++中的智能指针很像，智能指针也是将一个指针封装，然后通过引用计数或是其他方法完成自动释放内存的功能。

## 14、map、set是怎么实现的，红黑树是怎么能够同时实现这两种容器？ 为什么使用红黑树？

底层都是以红黑树的结构实现，因此插入删除等操作都在O(logn)时间内完成

实现map的红黑树的节点数据类型是key+value，而实现set的节点数据类型是value

map和set要求是自动排序的，红黑树能够实现这一功能，而且时间复杂度比较低

## 15、如何在共享内存上使用STL标准库？

假设进程A在共享内存中放入了数个容器，进程B如何找到这些容器呢？

一个方法就是进程A把容器放在共享内存中的确定地址上（fixed offsets），则进程B可以从该已知地址上获取容器。

另外一个改进点的办法是，进程A先在共享内存某块确定地址上放置一个map容器，然后进程A再创建其他容器，然后给其取个名字和地址一并保存到这个map容器里。进程B知道如何获取该保存了地址映射的map容器，然后同样再根据名字取得其他容器的地址。

## 16、map插入方式有哪几种？

① 用insert函数插入pair数据，

mapStudent.insert(pair<int, string>(1, "student\_one"));

② 用insert函数插入value\_type数据

mapStudent.insert(map<int, string>::value\_type (1, "student\_one"));

③ 在insert函数中使用make\_pair()函数

mapStudent.insert(make\_pair(1, "student\_one"));

④ 用数组方式插入数据

mapStudent[1] = "student\_one";

## 17、STL中unordered\_map(hash\_map)和map的区别，hash\_map如何解决冲突以及扩容

1. unordered\_map和map类似，都是存储的key-value的值，可以通过key快速索引到value。不同的是unordered\_map不会根据key的大小进行排序，
2. 存储时是根据key的hash值判断元素是否相同，即unordered\_map内部元素是无序的，而map中的元素是按照二叉搜索树存储，进行中序遍历会得到有序遍历。
3. 使用时map的key需要定义operator<。而unordered\_map需要定义hash\_value函数并且重载operator==。但是很多系统内置的数据类型都自带这些
4. hash\_map底层使用的是hash\_table，而hash\_table使用的开链法进行冲突避免，所有hash\_map采用开链法进行冲突解决。
5. 什么时候扩容：当向容器添加元素的时候，会判断当前容器的元素个数，如果大于等于阈值---即当前数组的长度乘以加载因子的值的时候，自动扩容。
6. 扩容(resize)就是重新计算容量，向HashMap对象里不停的添加元素，而HashMap对象内部的数组无法装载更多的元素时，对象就需要扩大数组的长度，以便能装入更多的元素。

## 18、vector越界访问下标，map越界访问下标？vector删除元素时会不会释放空间？

1. 通过下标访问vector中的元素时会做边界检查，但该处的实现方式要看具体IDE，不同IDE的实现方式不一样，确保不可访问越界地址。
2. map的下标运算符[]的作用是：将key作为下标去执行查找，并返回相应的值；如果不存在这个key，就将一个具有该key和value的默认值插入这个map。
3. erase()函数，只能删除内容，不能改变容量大小;

erase成员函数，它删除了itVect迭代器指向的元素，并且返回要被删除的itVect之后的迭代器，迭代器相当于一个智能指针;clear()函数，只能清空内容，不能改变容量大小;如果要想在删除内容的同时释放内存，那么你可以选择deque容器。

## 19、map中[]与find的区别？

1. map的下标运算符[]的作用是：将关键码作为下标去执行查找，并返回对应的值；如果不存在这个关键码，就将一个具有该关键码和值类型的默认值的项插入这个map。
2. map的find函数：用关键码执行查找，找到了返回该位置的迭代器；如果不存在这个关键码，就返回尾迭代器。

## 20、 STL中list与deque之间的区别

1. list不再能够像vector一样以普通指针作为迭代器，因为其节点不保证在存储空间中连续存在；
2. list不仅是一个双向链表，而且还是一个**环状**双向链表，所以它只需要一个指针；
3. list不像vector那样有可能在空间不足时做重新配置、数据移动的操作，所以插入前的所有迭代器在插入操作之后都仍然有效；
4. deque是一种双向开口的连续线性空间，所谓双向开口，意思是可以在头尾两端分别做元素的插入和删除操作；
5. deque和vector最大的差异，一在于deque允许常数时间内对起头端进行元素的插入或移除操作；二在于deque没有所谓容量概念，因为它是动态地以分段连续空间组合而成，随时可以增加一段新的空间并链接起来，deque没有所谓的空间保留功能。

## 21、STL中的allocator、deallocator

1. 第一级配置器直接使用malloc()、free()和relloc()，第二级配置器视情况采用不同的策略：当配置区块超过128bytes时，视之为足够大，便调用第一级配置器；当配置器区块小于128bytes时，为了降低额外负担，使用复杂的内存池整理方式，而不再用一级配置器；
2. 第二级配置器主动将任何小额区块的内存需求量上调至8的倍数，并维护16个free-list，各自管理大小为8~128bytes的小额区块；
3. 空间配置函数allocate()，首先判断区块大小，大于128就直接调用第一级配置器，小于128时就检查对应的free-list。如果free-list之内有可用区块，就直接拿来用，如果没有可用区块，就将区块大小调整至8的倍数，然后调用refill()，为free-list重新分配空间；
4. 空间释放函数deallocate()，该函数首先判断区块大小，大于128bytes时，直接调用一级配置器，小于128bytes就找到对应的free-list然后释放内存。

## 22、STL中hash table扩容发生什么？

当哈希表的元素数量增长到某个阈值时，就需要进行扩容。这个阈值通常是哈希表容量（bucket count）和装载因子（load factor）的乘积。装载因子是一个浮点数，它决定了哈希表元素数与容量之间的比例。默认装载因子通常是1.0。

当需要扩容时，STL会执行以下步骤：

1.创建一个新的、更大的哈希表。新的大小通常是原来大小的两倍。

2.遍历原有哈希表中的所有元素，并重新计算它们在新哈希表中的位置（这个位置由元素的哈希值和新的哈希表大小决定）。

3.将每个元素从旧的哈希表移动到新的哈希表的对应位置。

这个过程被称为 rehash。这是一个相当消耗资源的操作，因为它涉及到重新计算每个元素的哈希值，并可能涉及到大量的内存操作。但是，通过这种方式，可以保证哈希表的性能，因为它能保证元素在哈希表中的分布更均匀，减少哈希冲突，提高查找效率。

注意，虽然扩容可以提高哈希表的性能，但是频繁的扩容操作会消耗大量的资源，影响程序的性能。因此，如果你预先知道哈希表需要存储大量的元素，可以通过 rehash 或 reserve 方法预先分配足够的空间，避免频繁的扩容操作。

## 23、常见容器性质总结？

1.vector 底层数据结构为数组，支持快速随机访问

2.list 底层数据结构为环状双向链表，支持快速增删

3.deque 底层数据结构为一个中央控制器和多个缓冲区，支持首尾（中间不能）快速增删，也支持随机访问。没有reserve功能

deque是一个双端队列(double-ended queue)，也是在堆中保存内容的.它的保存形式如下:

[堆1] --> [堆2] -->[堆3] --> ...

每个堆保存好几个元素,然后堆和堆之间有指针指向,看起来像是list和vector的结合品.

4.stack 底层一般用list或deque实现，封闭头部即可，不用vector的原因应该是容量大小有限制，扩容耗时

5.queue 底层一般用list或deque实现，封闭头部即可，不用vector的原因应该是容量大小有限制，扩容耗时（stack和queue其实是适配器,而不叫容器，因为是对容器的再封装）

6.priority\_queue 的底层数据结构一般为vector为底层容器，堆heap为处理规则来管理底层容器实现

7.set 底层数据结构为红黑树，有序，不重复

8.multiset 底层数据结构为红黑树，有序，可重复

9.map 底层数据结构为红黑树，有序，不重复

10.multimap 底层数据结构为红黑树，有序，可重复

11.unordered\_set 底层数据结构为hash表，无序，不重复

12.unordered\_multiset 底层数据结构为hash表，无序，可重复

13.unordered\_map 底层数据结构为hash表，无序，不重复

14.unordered\_multimap 底层数据结构为hash表，无序，可重复

## 25、说一下STL每种容器对应的迭代器



## 28、STL中slist（forward\_list）的实现

list是双向链表，而slist（single linked list）是单向链表，它们的主要区别在于：前者的迭代器是双向的Bidirectional iterator，后者的迭代器属于单向的Forward iterator。虽然slist的很多功能不如list灵活，但是其所耗用的空间更小，操作更快。

由于forward\_list的迭代器不能回头，只能往后走，因此在slist的其他位置插入或者移除元素是十分不明智的，slist特别提供了insert\_after()和erase\_after供灵活应用（在某迭代器后插入和删除）。考虑到效率问题，slist只提供push\_front()操作。

forward\_list没有size方法