第一部分、 c++ 基础

1、C和C++的区别

C是面向过程的语言，是一个结构化的语言，考虑如何通过一个过程对输入进行处理得到输出；C++是面向对象的语言，主要特征是“封装、继承和多态”。封装隐藏了实现细节，使得代码模块化；派生类可以继承父类的数据和方法，扩展了已经存在的模块，实现了代码重用；多态则是“一个接口，多种实现”，通过派生类重写父类的虚函数，实现了接口的重用。

C和C++动态管理内存的方法不一样，C是使用mallocfree，而C++除此之外还有newdelete关键字。

C++中有引用，C中不存在引用的概念

2、C++中指针和引用的区别

指针有自己的一块空间，而引用只是一个别名；

使用 sizeof 看一个指针的大小为4字节（32位，如果要是64位的话指针为8字节），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；

指针在使用中可以指向其他对象，但是引用只能是一个对象的引用，不能被改变；

指针可以是多级，而引用没有分级；

如果返回动态分配内存的对象或者内存，必须使用指针，引用可能引起内存泄漏。

引用占用内存空间吗？ 对引用取地址，其实是取的引用所对应的内存空间的地址。这个现象让人觉得引用好像并非一个实体。但是引用是占用内存空间的，而且其占用的内存和指针一样，因为引用的内部实现就是通过指针来完成的。

3、结构体struct和共同体union（联合）的区别

结构体：将不同类型的数据组合成一个整体，是自定义类型 共同体：不同类型的几个变量共同占用一段内存

结构体中的每个成员都有自己独立的地址，它们是同时存在的； 共同体中的所有成员占用同一段内存，它们不能同时存在；

sizeof(struct)是内存对齐后所有成员长度的总和，sizeof(union)是内存对齐后最长数据成员的长度。

结构体为什么要内存对齐呢？

平台原因（移植原因）：不是所有的硬件平台都能访问任意地址上的任意数据，某些硬件平台只能在某些地址处取某些特定类型的数据，否则抛出硬件异常

硬件原因：经过内存对齐之后，CPU的内存访问速度大大提升。

4、struct 和 class 的区别？

首先说一下C中的结构体和C++中的结构体的异同：

C++中 struct 与 class 的区别： 内部成员变量及成员函数的默认访问属性：struct 默认防控属性是 public 的，而 class 默认的访问属性是private的 继承关系中默认访问属性的区别：在继承关系，struct 默认是 public 的，而 class 是 private class这个关键字还可用于定义模板参数，就等同于 typename；而strcut不用与定义模板参数

5、#define和const的区别

#define定义的常量没有类型，所给出的是一个立即数；const定义的常量有类型名字，存放在静态区域

处理阶段不同，#define定义的宏变量在预处理时进行替换，可能有多个拷贝，const所定义的变量在编译时确定其值，只有一个拷贝。

#define定义的常量是不可以用指针去指向，const定义的常量可以用指针去指向该常量的地址

#define可以定义简单的函数，const不可以定义函数

6、重载overload，覆盖（重写）override，隐藏（重定义）overwrite，这三者之间的区别

overload，将语义相近的几个函数用同一个名字表示，但是参数列表（参数的类型，个数，顺序不同）不同，这就是函数重载，返回值类型可以不同 特征：相同范围（同一个类中）、函数名字相同、参数不同、virtual关键字可有可无

override，派生类覆盖基类的虚函数，实现接口的重用，返回值类型必须相同 特征：不同范围（基类和派生类）、函数名字相同、参数相同、基类中必须有virtual关键字（必须是虚函数）

overwrite，派生类屏蔽了其同名的基类函数，返回值类型可以不同 特征：不同范围（基类和派生类）、函数名字相同、参数不同或者参数相同且无virtual关键字

7、new 和 delete 是如何实现的，与 malloc 和 free有什么异同？

new操作针对数据类型的处理，分为两种情况：

简单数据类型（包括基本数据类型和不需要构造函数的类型） 简单类型直接调用 operator new 分配内存； 可以通过new\_handler 来处理 new 失败的情况； new 分配失败的时候不像 malloc 那样返回 NULL，它直接抛出异常（bad\_alloc）。要判断是否分配成功应该用异常捕获的机制；

复杂数据类型（需要由构造函数初始化对象） new 复杂数据类型的时候先调用operator new，然后在分配的内存上调用构造函数。

delete也分为两种情况：

简单数据类型（包括基本数据类型和不需要析构函数的类型） delete简单数据类型默认只是调用free函数。

复杂数据类型（需要由析构函数销毁对象） delete复杂数据类型先调用析构函数再调用operator delete。

与 malloc 和 free 的区别：

属性上：new delete 是c++关键字，需要编译器支持。 mallocfree是库函数，需要c的头文件支持。

参数：使用new操作符申请内存分配时无须制定内存块的大小，编译器会根据类型信息自行计算。而mallco则需要显式地指出所需内存的尺寸。

返回类型：new操作符内存分配成功时，返回的是对象类型的指针，类型严格与对象匹配，故new是符合类型安全性的操作符。而malloc内存成功分配返回的是void ，需要通过类型转换将其转换为我们需要的类型。

分配失败时：new内存分配失败时抛出bad\_alloc异常；malloc分配内存失败时返回 NULL。

自定义类型：new会先调用operator new函数，申请足够的内存（通常底层使用malloc实现）。然后调用类型的构造函数，初始化成员变量，最后返回自定义类型指针。delete先调用析构函数，然后调用operator delete函数释放内存（通常底层使用free实现）。 mallocfree是库函数，只能动态的申请和释放内存，无法强制要求其做自定义类型对象构造和析构工作。

重载：C++允许重载 newdelete 操作符。而malloc为库函数不允许重载。

内存区域：new操作符从自由存储区（free store）上为对象动态分配内存空间，而malloc函数从堆上动态分配内存。其中自由存储区为：C++基于new操作符的一个抽象概念，凡是通过new操作符进行内存申请，该内存即为自由存储区。而堆是操作系统中的术语，是操作系统所维护的一块特殊内存，用于程序的内存动态分配，C语言使用malloc从堆上分配内存，使用free释放已分配的对应内存。自由存储区不等于堆，如上所述，布局new就可以不位于堆中。

既然有了mallocfree，C++中为什么还需要newdelete呢？ 运算符是语言自身的特性，有固定的语义，编译器知道意味着什么，由编译器解释语义，生成相应的代码。库函数是依赖于库的，一定程度上独立于语言的。编译器不关心库函数的作用，只保证编译，调用函数参数和返回值符合语法，生成call函数的代码。 对于非内部数据类型而言，光用mallocfree无法满足动态对象都要求。newdelete是运算符，编译器保证调用构造和析构函数对对象进行初始化析构。但是库函数mallocfree是库函数，不会执行构造析构。

8、delete和delete[]的区别

delete只会调用一次析构函数，而delete[]会调用每个成员的析构函数 用new分配的内存用delete释放，用new[]分配的内存用delete[]释放

9、const知道吗？解释一下其作用

const修饰类的成员变量，表示常量不可能被修改 const修饰类的成员函数，表示该函数不会修改类中的数据成员，不会调用其他非const的成员函数 const函数只能调用const函数，非const函数可以调用const函数

10、关键字static的作用

函数体内： static 修饰的局部变量作用范围为该函数体，不同于auto变量，其内存只被分配一次，因此其值在下次调用的时候维持了上次的值

模块内：static修饰全局变量或全局函数，可以被模块内的所有函数访问，但是不能被模块外的其他函数访问，使用范围限制在声明它的模块内

类中：修饰成员变量，表示该变量属于整个类所有，对类的所有对象只有一份拷贝

类中：修饰成员函数，表示该函数属于整个类所有，不接受this指针，只能访问类中的static成员变量 注意和const的区别！！！const强调值不能被修改，而static强调唯一的拷贝，对所有类的对象

11、堆和栈的区别

堆栈空间分配区别： 栈（操作系统）：由操作系统自动分配释放 ，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（操作系统）： 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，分配方式倒是类似于链表。

堆栈的缓存方式区别 栈：是内存中存储值类型的，大小为2M（window，linux下默认为8M，可以更改），超出则会报错，内存溢出； 堆：内存中，存储的是引用数据类型，引用数据类型无法确定大小，堆实际上是一个在内存中使用到内存中零散空间的链表结构的存储空间，堆的大小由引用类型的大小直接决定，引用类型的大小的变化直接影响到堆的变化。

堆栈数据结构上的区别 堆（数据结构）：堆可以被看成是一棵树，如：堆排序； 栈（数据结构）：一种先进后出的数据结构。

C++内存区域中堆和栈的区别： 管理方式不同：栈是由编译器自动管理，无需我们手工控制；对于堆来说，释放由程序员完成，容易产生内存泄漏。 空间大小不同：一般来讲，在32为系统下面，堆内存可达到4G的空间，从这个角度来看堆内存几乎是没有什么限制的。但是对于栈来讲，一般都是有一定空间大小的，例如，在vc6下面，默认的栈大小好像是1M。当然，也可以自己修改：打开工程。 project--setting--link，在category中选中output，然后再reserve中设定堆栈的最大值和 commit。 能否产生碎片：对于堆来讲，频繁的newdelete势必会造成内存空间的不连续，从而造成大量的碎片，使程序效率降低。对于栈来讲，则不会存在这个问题。 生长方向不同：对于堆来讲，生长方向是向上的，也就是向着内存地址增加的方向；对于栈来讲，它的生长方式是向下的，是向着内存地址减小的方向增长。 分配方式不同：堆都是动态分配的；栈静态分配由编译器完成，比如局部变量的分配 分配效率不同：栈是机器系统提供的数据结构，计算机会在底层对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行，这就决定了栈的效率比较高。堆则是cc++库函数提供的，机制很复杂。库函数会按照一定的算法进行分配。显然，堆的效率比栈要低得多。 进程内存中的映像，主要有代码区，堆（动态存储区，newdelete的动态数据），栈，静态存储区

堆和自由存储区的区别？ 总的来说，堆是C语言和操作系统的术语，是操作系统维护的一块动态分配内存；自由存储是C++中通过new与delete动态分配和释放对象的抽象概念。他们并不是完全一样。 从技术上来说，堆（heap）是C语言和操作系统的术语。堆是操作系统所维护的一块特殊内存，它提供了动态分配的功能，当运行程序调用malloc()时就会从中分配，稍后调用free可把内存交还。而自由存储是C++中通过new和delete动态分配和释放对象的抽象概念，通过new来申请的内存区域可称为自由存储区。基本上，所有的C++编译器默认使用堆来实现自由存储，也即是缺省的全局运算符new和delete也许会按照malloc和free的方式来被实现，这时藉由new运算符分配的对象，说它在堆上也对，说它在自由存储区上也正确。 扩展 堆和栈的区别（转过无数次的文章）

12、#includefile.h #include file.h 的区别

前者是从标准库路径寻找 后者是从当前工作路径

13、什么是内存泄漏？面对内存泄漏和指针越界，你有哪些方法？

动态分配内存所开辟的空间，在使用完毕后未手动释放，导致一直占据该内存，即为内存泄漏。 方法：mallocfree要配套，对指针赋值的时候应该注意被赋值的指针是否需要释放；使用的时候记得指针的长度，防止越界

14、定义和声明的区别

为变量分配地址和存储空间的称为定义，不分配地址的称为声明。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。

15、C++文件编译与执行的四个阶段

预处理：根据文件中的预处理指令来修改源文件的内容

编译：编译成汇编代码

汇编：把汇编代码翻译成目标机器指令

链接：链接目标代码生成可执行程序

16、C++的内存管理

在C++中，内存被分成五个区：栈、堆、自由存储区、静态存储区、常量区 栈：存放函数的参数和局部变量，编译器自动分配和释放 堆：new关键字动态分配的内存，由程序员手动进行释放，否则程序结束后，由操作系统自动进行回收 自由存储区：new所申请的内存则是在自由存储区上，使用delete来释放。

堆是操作系统维护的一块内存，而自由存储是C++中通过new与delete动态分配和释放对象的抽象概念。堆与自由存储区并不等价。

全局静态存储区：存放全局变量和静态变量 常量区：存放常量，不允许被修改

17、 C++的四种强制转换

类型转化机制可以分为隐式类型转换和显示类型转化（强制类型转换） (new-type) expression new-type (expression) 隐式类型转换比较常见，在混合类型表达式中经常发生；四种强制类型转换操作符： static\_cast、dynamic\_cast、const\_cast、reinterpret\_cast

static\_cast ：编译时期的静态类型检查static\_cast静态转换相当于C语言中的强制转换，但不能实现普通指针数据（空指针除外）的强制转换，一般用于父类和子类指针、引用间的相互转换。没有运行时类型检查来保证转换的安全性。

①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。不管是否发生多态，父子之间互转时，编译器都不会报错。

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，由于没有动态类型检查，所以是不安全的，但是编译器不会报错。

②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。 ③把空指针转换成目标类型的空指针。 ④把任何指针类型转换成空指针类型。 ⑤可以对普通数据的const和non\_const进行转换，但不能对普通数据取地址后的指针进行const添加和消去。 ⑥无继承关系的自定义类型，不可转换，不支持类间交叉转换。 另外，static\_cast不能转换掉原有类型的const、volatile、或者 \_\_unaligned属性。(前两种可以使用const\_cast 来去除)。

dynamic\_cast：运行时的检查 动态转换的类型和操作数必须是完整类类型或空指针、空引用，说人话就是说，只能用于类间转换，支持类间交叉转换，不能操作普通数据。 主要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， ①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，允许转换； ②进行下行转换（把基类指针或引用转换成派生类表示）时，由于没有动态类型检查，所以是不安全的，不允许转化，编译器会报错； ③发生多态时，允许互相转换。 ④无继承关系的类之间也可以相互转换，类之间的交叉转换。 ⑤如果dynamic\_cast语句的转换目标是指针类型并且失败了，则结果为0。如果转换目标是引用类型并且失败了，则dynamic\_cast运算符将抛出一个stdbad\_cast异常

const\_cast const\_cast用于强制去掉不能被修改的常数特性，其去除常量性的对象一般为指针或引用。

reinterpret\_cast 在C++语言中，reinterpret\_cast主要有几种强制转换用途：将指针或引用转换为一个足够长度的整型、将整型转换为指针或引用类型。 更详细见 C++的四种强制转换

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。

18、extern“C”作用

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语言的进行编译，而不是C++的。

19、typdef和define区别

#define是预处理命令，在预处理是执行简单的替换，不做正确性的检查 typedef是在编译时处理的，它是在自己的作用域内给已经存在的类型一个别名 typedef (int) pINT; #define pINT2 int 效果相同？实则不同！实践中见差别：pINT a,b;的效果同int a; int b;表示定义了两个整型指针变量。而pINT2 a,b;的效果同int a, b;表示定义了一个整型指针变量a和整型变量b。

20、引用作为函数参数以及返回值的好处

对比值传递，引用传参的好处：

在函数内部可以对此参数进行修改

提高函数调用和运行的效率（所以没有了传值和生成副本的时间和空间消耗） 值传递： 形参是实参的拷贝，改变形参的值并不会影响外部实参的值。从被调用函数的角度来说，值传递是单向的（实参-形参），参数的值只能传入，不能传出。当函数内部需要修改参数，并且不希望这个改变影响调用者时，采用值传递。 指针传递： 形参为指向实参地址的指针，当对形参的指向操作时，就相当于对实参本身进行的操作 引用传递： 形参相当于是实参的“别名”，对形参的操作其实就是对实参的操作，在引用传递过程中，被调函数的形式参数虽然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被处理成间接寻址，即通过栈中存放的地址访问主调函数中的实参变量。正因为如此，被调函数对形参做的任何操作都影响了主调函数中的实参变量。 用引用作为返回值最大的好处就是在内存中不产生被返回值的副本。

但是有以下的限制：

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

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

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。

21、什么是野指针

野指针不是NULL指针，是未初始化或者未清零的指针，它指向的内存地址不是程序员所期望的，可能指向了受限的内存。 成因：

指针变量没有被初始化

指针指向的内存被释放了，但是指针没有置NULL

指针超过了变量了的作用范围，比如b[10]，指针b+11

22、C++中内存泄漏的几种情况

内存泄漏是指动态分配的堆内存由于某种原因程序未释放或无法释放，造成系统内存的浪费，导致程序运行速度减慢甚至系统崩溃等严重后果。

类的构造函数和析构函数中new和delete没有配套

在释放对象数组时没有使用delete[]，使用了delete

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释放，因此造成内存泄露

没有正确的清楚嵌套的对象指针

23、栈溢出的原因以及解决方法

栈溢出是指函数中的局部变量造成的溢出（注：函数中形参和函数中的局部变量存放在栈上） 栈的大小通常是1M-2M,所以栈溢出包含两种情况，一是分配的的大小超过栈的最大值，二是分配的大小没有超过最大值，但是接收的buf比原buf小。

函数调用层次过深,每调用一次,函数的参数、局部变量等信息就压一次栈

局部变量体积太大。

解决办法大致说来也有两种：

增加栈内存的数目；如果是不超过栈大小但是分配值小的，就增大分配的大小

使用堆内存；具体实现由很多种方法可以直接把数组定义改成指针,然后动态申请内存;也可以把局部变量变成全局变量,一个偷懒的办法是直接在定义前边加个static,呵呵,直接变成静态变量(实质就是全局变量)

24、左值和右值

不是很严谨的来说，左值指的是既能够出现在等号左边也能出现在等号右边的变量(或表达式)，右值指的则是只能出现在等号右边的变量(或表达式)。举例来说我们定义的变量 a 就是一个左值，而malloc返回的就是一个右值。或者左值就是在程序中能够寻址的东西，右值就是一个具体的真实的值或者对象，没法取到它的地址的东西(不完全准确)，因此没法对右值进行赋值，但是右值并非是不可修改的，比如自己定义的class, 可以通过它的成员函数来修改右值。 归纳一下就是： 可以取地址的，有名字的，非临时的就是左值 不能取地址的，没有名字的，临时的，通常生命周期就在某个表达式之内的就是右值 但是到了 C++11 之后概念变的略微复杂，引入了 lvalue, glvalue, rvalue, xvalue 和 prvalue。

25、左值引用与右值引用

左值引用就是我们通常所说的引用，如下所示。左值引用通常可以看作是变量的别名。

type-id & cast-expression

demo

int a = 10

int &b = a

int &c = 10 错误，无法对一个立即数做引用

const int &d = 10 正确， 常引用引用常数量是ok的，其等价于 const int temp = 10; const int &d = temp

右值引用是 C++11 新增的特性，其形式如下所示。右值引用用来绑定到右值，绑定到右值以后本来会被销毁的右值的生存期会延长至与绑定到它的右值引用的生存期。

右值引用支持移动语义的实现，可以减少拷贝，提升程序的执行效率。

右值引用可以使重载函数变得更加简洁。右值引用可以适用 const T& 和 T& 形式的参数。

type-id && cast-expression

demo

int &&var = 10; ok

int a = 10

int &&b = a 错误， a 为左值

int &&c = var 错误，var 为左值

int &&d = move(a) ok, 通过move得到左值的右值引用

在汇编层面右值引用做的事情和常引用是相同的，即产生临时量来存储常量。但是，唯一 一点的区别是，右值引用可以进行读写操作，而常引用只能进行读操作。

26、头文件中的 ifndefdefineendif 是干什么用的 该用法和 program once 的区别？

相同点： 它们的作用是防止头文件被重复包含。 不同点：

ifndef 由语言本身提供支持，但是 program once 一般由编译器提供支持，也就是说，有可能出现编译器不支持的情况(主要是比较老的编译器)。

通常运行速度上 ifndef 一般慢于 program once，特别是在大型项目上， 区别会比较明显，所以越来越多的编译器开始支持 program once。

ifndef 作用于某一段被包含（define 和 endif 之间）的代码， 而 program once 则是针对包含该语句的文件， 这也是为什么 program once 速度更快的原因。

如果用 ifndef 包含某一段宏定义，当这个宏名字出现“撞车”时，可能会出现这个宏在程序中提示宏未定义的情况（在编写大型程序时特别需要注意，因为有很多程序员在同时写代码）。相反由于program once 针对整个文件， 因此它不存在宏名字“撞车”的情况， 但是如果某个头文件被多次拷贝，program once 无法保证不被多次包含，因为program once 是从物理上判断是不是同一个头文件，而不是从内容上。

27、指针数组和数组指针的区别

数组指针，是指向数组的指针，而指针数组则是指该数组的元素均为指针。

数组指针，是指向数组的指针，其本质为指针，形式如下。如 int (p)[n]，p即为指向数组的指针，()优先级高，首先说明p是一个指针，指向一个整型的一维数组，这个一维数组的长度是n，也可以说是p的步长。也就是说执行p+1时，p要跨过n个整型数据的长度。数组指针是指向数组首元素的地址的指针，其本质为指针，可以看成是二级指针。

类型名 (数组标识符)[数组长度]

指针数组，在C语言和C++中，数组元素全为指针的数组称为指针数组，其中一维指针数组的定义形式如下。指针数组中每一个元素均为指针，其本质为数组。如 int p[n]， []优先级高，先与p结合成为一个数组，再由int说明这是一个整型指针数组，它有n个指针类型的数组元素。这里执行p+1时，则p指向下一个数组元素，这样赋值是错误的：p=a；因为p是个不可知的表示，只存在p[0]、p[1]、p[2]…p[n-1],而且它们分别是指针变量可以用来存放变量地址。但可以这样 p=a; 这里p表示指针数组第一个元素的值，a的首地址的值。

类型名 数组标识符[数组长度]

28、C++是不是类型安全的？

不是。两个不同类型的指针之间可以强制转换

29、main函数执行之前，还会执行什么代码？

全局对象的构造函数会在main函数之前执行。

30、全局变量和局部变量有什么区别？是怎么实现的？操作系统和编译器是怎么知道的？

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程序的各个部分都可以用到；局部变量只能在局部使用；分配在栈区。 内存分配位置不同： 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 。

31、关于sizeof小结的。

答：sizeof计算的是在栈中分配的内存大小。

sizeof不计算static变量占得内存；

32位系统的指针的大小是4个字节，64位系统的指针是8字节，而不用管指针类型；

char型占1个字节，int占4个字节，short int占2个字节 long int占4个字节，float占4字节，double占8字节，string占4字节 一个空类占1个字节，单一继承的空类占1个字节，虚继承涉及到虚指针所以占4个字节

数组的长度： 若指定了数组长度，则不看元素个数，总字节数=数组长度sizeof（元素类型） 若没有指定长度，则按实际元素个数类确定 Ps：若是字符数组，则应考虑末尾的空字符。

结构体对象的长度 在默认情况下，为方便对结构体内元素的访问和管理，当结构体内元素长度小于处理器位数的时候，便以结构体内最长的数据元素的长度为对齐单位，即为其整数倍。若结构体内元素长度大于处理器位数则以处理器位数为单位对齐。

unsigned影响的只是最高位的意义，数据长度不会改变，所以sizeof（unsigned int）=4

自定义类型的sizeof取值等于它的类型原型取sizeof

对函数使用sizeof，在编译阶段会被函数的返回值的类型代替

sizeof后如果是类型名则必须加括号，如果是变量名可以不加括号，这是因为sizeof是运算符

当使用结构类型或者变量时，sizeof返回实际的大小。当使用静态数组时返回数组的全部大小，sizeof不能返回动态数组或者外部数组的尺寸

32、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’ 的字符串做参数。

数组做 siezeof 的参数不退化，传递给 strlen 就退化为指针了。

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。 而且 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

33、说说内联函数

函数调用是有时间和空间开销的。程序在执行一个函数之前需要做一些准备工作，要将实参、局部变量、返回地址以及若干寄存器都压入栈中，然后才能执行函数体中的代码；函数体中的代码执行完毕后还要清理现场，将之前压入栈中的数据都出栈，才能接着执行函数调用位置以后的代码。如果函数体代码比较多，需要较长的执行时间，那么函数调用机制占用的时间可以忽略；如果函数只有一两条语句，那么大部分的时间都会花费在函数调用机制上，这种时间开销就就不容忽视。 为了消除函数调用的时空开销，C++ 提供一种提高效率的方法，即在编译时将函数调用处用函数体替换，类似于C语言中的宏展开。这种在函数调用处直接嵌入函数体的函数称为内联函数（Inline Function），又称内嵌函数或者内置函数。 详见： C++ inline内联函数详解

34、CC++的存储期

C对象有4种存储期：静态存储期、线程存储期、自动存储期、动态分配存储期。 详见： CC++的存储期

35、流操作符重载为什么返回引用

在程序中，流操作符和经常连续使用。因此这两个操作符的返回值应该是一个仍旧支持这两个操作符的流引用。其他的数据类型都无法做到这一点。 注意：除了在赋值操作符和流操作符之外的其他的一些操作符中，如+、-、、等却千万不能返回引用。因为这四个操作符的对象都是右值，因此，它们必须构造一个对象作为返回值。

36、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？

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； 内存中分配在全局数据区

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

操作系统和编译器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 。

第二部分、 c++ 类相关

1、什么是面向对象（OOP）？面向对象的意义？

Object Oriented Programming, 面向对象是一种对现实世界理解和抽象的方法、思想，通过将需求要素转化为对象进行问题处理的一种思想。其核心思想是数据抽象（封装）、继承和动态绑定（多态）。 面向对象的意义在于：将日常生活中习惯的思维方式引入程序设计中；将需求中的概念直观的映射到解决方案中；以模块为中心构建可复用的软件系统；提高软件产品的可维护性和可扩展性。

2、解释下封装、继承和多态？

封装： 封装是实现面向对象程序设计的第一步，封装就是将数据或函数等集合在一个个的单元中（我们称之为类）。 封装的意义在于保护或者防止代码（数据）被我们无意中破坏。 从封装的角度看，public， private 和 protected 属性的特点如下。

不管哪种属性，内类都是可以访问的

public 是一种暴露的手段，比如暴露接口，类的对象可以访问

private 是一种隐藏的手段，类的对象不能访问

protected 成员：

和 public 一样可以被子类继承

和 private 一样不能在类外被直接调用

特例：在衍生类中可以通过衍生类对象访问

继承： 继承主要实现重用代码，节省开发时间。 子类可以继承父类的一些东西。

a.公有继承(public) 公有继承的特点是基类的公有成员和保护成员作为派生类的成员时，它们都保持原有的状态（基类的私有成员仍然是私有的，不能被这个派生类的子类所访问）。

b.私有继承(private) 私有继承的特点是基类的公有成员和保护成员都作为派生类的私有成员（并且不能被这个派生类的子类所访问）。

c.保护继承(protected) 保护继承的特点是基类的所有公有成员和保护成员都成为派生类的保护成员（并且只能被它的派生类成员函数或友元访问，基类的私有成员仍然是私有的）。 这里特别提一下虚继承。虚继承是解决C++多重继承问题（其一，浪费存储空间；第二，存在二义性问题）的一种手段。比如菱形继承，典型的应用就是 iostream, 其继承于 istream 和 ostream，而 istream 和 ostream 又继承于 ios。

多态： 多态是指通过基类的指针或者引用，在运行时动态调用实际绑定对象函数的行为。与之相对应的编译时绑定函数称为静态绑定。多态是设计模式的基础，多态是框架的基础。 C++静态多态与动态多态

3、构造函数和析构函数的执行顺序？

构造函数：

首先调用父类的构造函数；

调用成员变量的构造函数；

调用类自身的构造函数。

析构函数 对于栈对象或者全局对象，调用顺序与构造函数的调用顺序刚好相反，也即后构造的先析构。对于堆对象，析构顺序与delete的顺序相关。

4、虚函数是怎么实现的

每一个含有虚函数的类都至少有有一个与之对应的虚函数表，其中存放着该类所有虚函数对应的函数指针（地址）， 类的示例对象不包含虚函数表，只有虚指针； 派生类会生成一个兼容基类的虚函数表。

5、 构造函数为什么一般不定义为虚函数？而析构函数一般写成虚函数的原因 ？

构造函数不能声明为虚函数 1). 因为创建一个对象时需要确定对象的类型，而虚函数是在运行时确定其类型的。而在构造一个对象时，由于对象还未创建成功，编译器无法知道对象的实际类型，是类本身还是类的派生类等等 2). 虚函数的调用需要虚函数表指针，而该指针存放在对象的内存空间中；若构造函数声明为虚函数，那么由于对象还未创建，还没有内存空间，更没有虚函数表地址用来调用虚函数即构造函数了

析构函数最好声明为虚函数 首先析构函数可以为虚函数，当析构一个指向派生类的基类指针时，最好将基类的析构函数声明为虚函数，否则可以存在内存泄露的问题。 如果析构函数不被声明成虚函数，则编译器实施静态绑定，在删除指向派生类的基类指针时，只会调用基类的析构函数而不调用派生类析构函数，这样就会造成派生类对象析构不完全。 子类析构时，要调用父类的析构函数吗？ 析构函数调用的次序时先派生类后基类的。和构造函数的执行顺序相反。并且析构函数要是virtual的，否则如果用父类的指针指向子类对象的时候，析构函数静态绑定，不会调用子类的析构。 不用显式调用，会自动调用

6、细看拷贝构造函数

对于 class A，它的拷贝构造函数如下：

AA(const A &a){}

为什么必须是当前类的引用呢？ 循环调用。如果拷贝构造函数的参数不是当前类的引用，而是当前类的对象，那么在调用拷贝构造函数时，会将另外一个对象直接传递给形参，这本身就是一次拷贝，会再次调用拷贝构造函数，然后又将一个对象直接传递给了形参，将继续调用拷贝构造函数……这个过程会一直持续下去，没有尽头，陷入死循环。

只有当参数是当前类的引用时，才不会导致再次调用拷贝构造函数，这不仅是逻辑上的要求，也是 C++ 语法的要求。

为什么是 const 引用呢？ 拷贝构造函数的目的是用其它对象的数据来初始化当前对象，并没有期望更改其它对象的数据，添加 const 限制后，这个含义更加明确了。

另外一个原因是，添加 const 限制后，可以将 const 对象和非 const 对象传递给形参了，因为非 const 类型可以转换为 const 类型。如果没有 const 限制，就不能将 const 对象传递给形参，因为 const 类型不能直接转换为非 const 类型，这就意味着，不能使用 const 对象来初始化当前对象了。

7、静态绑定和动态绑定的介绍

静态绑定和动态绑定是C++多态性的一种特性

对象的静态类型和动态类型 由于继承导致对象的指针和引用具有两种不同的类型：静态类型和动态类型。 静态类型：对象在声明时采用的类型，在编译时确定 动态类型：当前对象所指的类型，在运行期决定，对象的动态类型可变，静态类型无法更改

静态绑定和动态绑定 静态绑定：绑定的是对象的静态类型，函数依赖于对象的静态类型，在编译期确定 动态绑定：绑定的是对象的动态类型，函数依赖于对象的动态类型，在运行期确定 只有虚函数才使用的是动态绑定，其他的全部是静态绑定

引用是否能实现动态绑定，为什么引用可以实现 可以。因为引用（或指针）既可以指向基类对象也可以指向派生类对象，这一事实是动态绑定的关键。用引用（或指针）调用的虚函数在运行时确定，被调用的函数是引用（或指针）所指的对象的实际类型所定义的。

8、深拷贝和浅拷贝的区别

深拷贝和浅拷贝可以简单的理解为：如果一个类拥有资源，当这个类的对象发生复制过程的时候，如果资源重新分配了就是深拷贝；反之没有重新分配资源，就是浅拷贝。

9、 什么情况下会调用拷贝构造函数（三种情况）

系统自动生成的构造函数：普通构造函数和拷贝构造函数 （在没有定义对应的构造函数的时候） 生成一个实例化的对象会调用一次普通构造函数，而用一个对象去实例化一个新的对象所调用的就是拷贝构造函数 调用拷贝构造函数的情形：

用类的一个对象去初始化另一个对象的时候

当函数的参数是类的对象时，就是值传递的时候，如果是引用传递则不会调用

当函数的返回值是类的对象或者引用的时候

10、类对象的大小受哪些因素影响？

类的非静态成员变量大小，静态成员不占据类的空间，成员函数也不占据类的空间大小；

内存对齐另外分配的空间大小，类内的数据也是需要进行内存对齐操作的；

虚函数的话，会在类对象插入vptr指针，加上指针大小；

当该该类是某类的派生类，那么派生类继承的基类部分的数据成员也会存在在派生类中的空间中，也会对派生类进行扩展。

11、拷贝构造函数和赋值运算符的理解

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 ：当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。

12、C++空类默认有哪些成员函数？

答：默认构造函数、析构函数、复制构造函数、赋值函数

13、如果虚函数是有效的，那为什么不把所有函数设为虚函数？

答：不行。首先，虚函数是有代价的，由于每个虚函数的对象都要维护一个虚函数表，因此在使用虚函数的时候都会产生一定的系统开销，这是没有必要的。

14、构造函数和虚构函数可以是内联函数？

构造函数、析构函数、虚函数可以声明为内联函数，这在语法上是正确的。 编译器并不真正对声明为inline的构造和析构函数内联，因为编译器会在构造和析构函数中添加额外的操作（申请释放内存，构造析构对象，调用基类成员对象构造函数等），致使构造函数析构函数并不像看上去的那么精简。

15、虚函数声明为inline

inline是编译期决定的，而虚函数是运行期决定的，即在不知道将要调用哪个函数的情况下，如何将函数内联。

16、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。

17、C++中的五种构造函数

默认构造函数

普通构造函数

拷贝构造函数

转换构造函数

移动构造函数 详见： C++中的五种构造函数

第三部分、 c++11c++14c++17

1、C++11 中有哪些智能指针？shared\_ptr 的引用计数是如何实现的？unique\_ptr 的unique 是如何实现的？make\_shared 和 make\_unique 的作用？智能指针使用注意事项？

C++ 11 中的智能指针有：shared\_ptr, unique\_ptr 和 weak\_ptr。 shared\_ptr 的引用计数是存放在堆上的，多个 shared\_ptr 的对象的引用计数都指向同一个堆地址。 unique\_ptr 中拷贝构造函数和赋值操作符都声明为delete或private。 优先使用 make\_shared 和 make\_unique 的原因是为了避免内存泄露。参考 C++11 中的 Smart Pointer（shared\_ptrweak\_ptrunique\_ptr） 总结 智能指针使用注意事项： 不使用相同的内置指针值初始化，或reset多个智能指针 不delete get()返回的指针 不使用get()初始化或reset另一个智能指针 get()返回的智能指针可能变成dangling pointer 如果智能指针管理的内存不是new出来的，需要提供删除器 拓展问题 shared\_ptr 是否线程安全？ 侵入式智能指针？ 从源码理解智能指针（二）—— shared\_ptr、weak\_ptr

2、智能指针weak\_ptr 能够破坏环型引用的原理（引用计数的原理）

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。 引用计数原理 shared\_ptr的实现是这样的 shared\_ptr模板类有一个\_\_shared\_count类型的成员\_M\_refcount来处理引用计数的问题。\_\_shared\_count也是一个模板类，它的内部有一个指向Sp\_counted\_base\_impl类型的指针\_M\_pi。所有引用同一个对象的shared\_ptr都共用一个\_M\_pi指针。 当我们谈论shared\_ptr的线程安全性时，我们在谈论什么

3、C++ 的闭包

闭包：与函数A调用函数B相比较，闭包中函数A调用函数B，可以不通过函数A给函数B传递函数参数，而使函数B可以访问函数A的上下文环境才可见(函数A可直接访问到)的变量； c++11提供的闭包方式：lambda、function、bind

引用百度上对闭包的定义：闭包是指可以包含自由（未绑定到特定对象）变量的代码块；这些变量不是在这个代码块内或者任何全局上下文中定义的，而是在定义代码块的环境中定义（局部变量）。“闭包” 一词来源于以下两者的结合：要执行的代码块（由于自由变量被包含在代码块中，这些自由变量以及它们引用的对象没有被释放）和为自由变量提供绑定的计算环境（作用域）。

详见： c++11的闭包(lambda、function、bind)

4、lambda 表达式、怎么捕获外部变量

值捕获 值捕获和参数传递中的值传递类似，被捕获的变量的值在Lambda表达式创建时通过值拷贝的方式传入，因此随后对该变量的修改不会影响影响Lambda表达式中的值。

引用捕获 使用引用捕获一个外部变量，只需要在捕获列表变量前面加上一个引用说明符&。

隐式捕获 还可以让编译器根据函数体中的代码来推断需要捕获哪些变量，这种方式称之为隐式捕获

混合方式捕获 即同时使用显式捕获和隐式捕获。混合捕获时，捕获列表中的第一个元素必须是 = 或 &，此符号指定了默认捕获的方式是值捕获或引用捕获。

C++11捕获外部变量总结

捕获形式 说明

[] 不捕获任何外部变量

[变量名, …] 默认以值得形式捕获指定的多个外部变量（用逗号分隔），如果引用捕获，需要显示声明（使用&说明符）

[this] 以值的形式捕获this指针

[=] 以值的形式捕获所有外部变量

[&] 以引用形式捕获所有外部变量

[=, &x] 变量x以引用形式捕获，其余变量以传值形式捕获

[&, x] 变量x以值的形式捕获，其余变量以引用形式捕获

5、C++ 的垃圾回收机制

很多语言都有对内存自动管理，即所谓的垃圾回收机制。所谓垃圾，指的是那些不再使用或者没有任何指针指向的内存空间，而“回收”则指的是将这些“垃圾”收集起来以便再次利用。C++采用 unique\_ptr、shared\_ptr 以及 weak\_ptr 这 3 个智能指针来实现堆内存的自动回收。来实现自动释放分配的内存。 C++之智能指针（C++的垃圾回收机制）

6、vector的clear()的时间复杂度是多少？

C++标准明确指出不管是序列容器（⽐如vector）还是关联容器（⽐如unordered\_map)其 clear()成员函数都是线性时间复杂度O(n)的。因为只要执⾏了clear()就需要对其存储的元素调⽤析构函数 ，这个析构操作显然是逐个析构的。因⽽时间复杂度是O(n)。 当然在实践中，也有个例。⽐如当vector存储基本数据类型或POD类型（⽐如基本数据类型构成的struct）的时候，由于其元素类型没有析构函数（也不需要析构函数），加之vector内部连续存储的特性，编译器的实现是可以在常量时间完成clear()的。 仅限于vector存储基本数据类型和POD类型的时候，编译器可能有此优化。如果vector存储的 是其他类型的对象，或者是其他容器（⽐如list、map、unordered\_map）都是没办法做这个优化的！ 详见： C++和STL中有哪些副作用或者稍不注意会产生性能开销的地方？

7、C++11为什么引入enum class？

C++98 的 enum 是“非域化的”；而 C++11 的 enum class 是“域化的”，限制了枚举成员只在域内可见

enum class 的缺省潜在类型 (underlying type) 是 int 型，而 enum 没有缺省潜在类型

enum class 一般总是前置声明，而 enum 只有在指定了潜在类型时才可以是前置声明

详见： C++11 之 enum class

8、stdthread使用lambda做回调，有没有什么注意事项？

使用到外部引用要小心谨慎，避免悬空引用。 若需要用到的外部局部变量，需以值传递的方式捕获而非引用捕获（若是外部指针变量则需深拷贝）。

谨慎使用或者不用外部指针。 如果你用值捕获了个指针，你在lambda创建的闭包中持有这个指针的拷贝，但你不能阻止lambda外面的代码删除指针指向的内容，从而导致你拷贝的指针空悬。

注意引用捕获陷阱：引用捕获[&]不要使用外部局部变量。

注意this陷阱：lambda里避免有全局变量或静态变量或者比当前类生命周期更长的变量。Effective Modern C++ 条款31 对于lambda表达式，避免使用默认捕获模式。

避免使用默认捕获模式(（即“[=]”或“[&]”,它可能导致你看不出悬空引用问题)。 默认值捕获就意外地捕获了this指针，而不是你以为的外部变量。

例子参考： c++的lambda使用注意事项,可能导致的崩溃问题分析

9、C++11的thread\_local有没有使用过？

有且只有thread\_local关键字修饰的变量具有线程周期(thread duration)，这些变量(或者说对象）在线程开始的时候被生成(allocated)，在线程结束的时候被销毁(deallocated)。并且每 一个线程都拥有一个独立的变量实例(Each thread has its own instance of the object)。thread\_local 可以和static 与 extern关键字联合使用，这将影响变量的链接属性(to adjust linkage)。 那么，哪些变量可以被声明为thread\_local？以下3类都是ok的

命名空间下的全局变量

类的static成员变量

本地变量

详见： C++11中thread\_local的使用

10、谈一谈你对zero overhead（零开销原则）的理解

不需要为没有使用到的语言特性付出代价。

使用某种语言特性，不会带来运行时的代价。

详见： 跟我学c++中级篇——zero overhead abstraction

11、了解移动语义和完美转发吗？

移动语义：可以理解为转移所有权，拷贝是对于别人的资源，自己重新分配一块内存存储复制过来的资源，而对于移动语义，类似于转让或者资源窃取的意思，对于那块资源，转为自己所拥有，别人不再拥有也不会再使用，通过C++11新增的移动语义可以省去很多拷贝负担，如何利用移动语义，主要通过移动构造函数。

完美转发：指可以写一个接受任意实参的函数模板，并转发到其它函数，目标函数会收到与转发函数完全相同的实参。转发函数实参是左值那目标函数实参也是左值，转发函数实参是右值那目标函数也是右值。 参考： C++11之右值引用：移动语义和完美转发(带你了解移动构造函数、纯右值、将亡值、右值引用、stdmove、forward等新概念)

12、了解列表初始化吗？

列表初始化：可以直接在变量名后面加上初始化列表来进行对象的初始化。 参考： C++11之初始化列表

第四部分、 多线程（多进程） 相关

1、线程与进程的区别

线程与进程的比较如下：

进程是资源（包括内存、打开的文件等）分配的单位，线程是 CPU 调度的单位；

进程拥有一个完整的资源平台，而线程只独享必不可少的资源，如寄存器和栈；

线程同样具有就绪、阻塞、执行三种基本状态，同样具有状态之间的转换关系；

线程能减少并发执行的时间和空间开销；

对于，线程相比进程能减少开销，体现在：

线程的创建时间比进程快，因为进程在创建的过程中，还需要资源管理信息，比如内存管理信息、文件管理信息，而线程在创建的过程中，不会涉及这些资源管理信息，而是共享它们；

线程的终止时间比进程快，因为线程释放的资源相比进程少很多；

同一个进程内的线程切换比进程切换快，因为线程具有相同的地址空间（虚拟内存共享），这意味着同一个进程的线程都具有同一个页表，那么在切换的时候不需要切换页表。而对于进程之间的切换，切换的时候要把页表给切换掉，而页表的切换过程开销是比较大的；

由于同一进程的各线程间共享内存和文件资源，那么在线程之间数据传递的时候，就不需要经过内核了，这就使得线程之间的数据交互效率更高了； 所以，线程比进程不管是时间效率，还是空间效率都要高。

进程之间私有和共享的资源 私有：地址空间、堆、全局变量、栈、寄存器 共享：代码段，公共数据，进程目录，进程 ID 线程之间私有和共享的资源 私有：线程栈，寄存器，程序计数器 共享：堆，地址空间，全局变量，静态变量

2、什么是线程不安全

在随机调度之下,线程执行有多种可能,其中某些可能会导致代码出bug就称为线程不安全.

3、造成线程不安全的原因

操作系统的随机调度抢占式执行(万恶之源)--无法改变

多个线程修改同一个变量(一个字都不能少)---尽量避免

有些修改操作,不是原子的!(不可拆分的最小单位,就叫原子 即对应一条机器指令)---通过加锁操作,把指令打包

内存可见性问题(内存改了,但是在优化的背景下,读不到,看不见) 如线程1一直在读取硬盘上的资源再判断,在多次读取硬盘并获得相同的结果后编译器会认为这样的做法过于低效,然后就会省略读取的过程,一直判断,若此时有线程2需要这个判断结果(读取到数据后的判断结果)就会出现问题.

指令重排序(也是编译器,操作系统等的优化,调整了代码的执行顺序)

最常见的就是对象new的问题： 分为三步1.创建内存空间 2.往内存空间上构造对象 3.把这个内存的引用赋值给你要创建的变量 有时候编译器会认为先执行3或者先执行2最高效,就会出现指令顺序倒转的问题,这时另一个线程尝试读取变量的引用就会出现问题

4、C++线程中的几类锁

线程之间的锁有： 互斥锁、条件锁、自旋锁、读写锁、递归锁。一般而言，锁的功能与性能成反比（详见C++11线程中的几种锁、如何理解互斥锁、条件锁、读写锁以及自旋锁？）

5、什么是死锁，解决方式

死锁：死锁是指两个或两个以上的进程（线程）在运行过程中因争夺资源而造成的一种僵局，若无外力作用，这些进程（线程）都将无法向前推进。 详见： 死锁四个必要条件及死锁的预防、检测、避免、解除

6、进程之间的通信方式

管道（PIPE） （1）有名管道：一种半双工的通信方式，它允许无亲缘关系进程间的通信 ①优点：可以实现任意关系的进程间的通信 ②缺点：a、长期存于系统中，使用不当容易出错；b、缓冲区有限 （2）无名管道：一种半双工的通信方式，只能在具有亲缘关系的进程间使用（父子进程） ①优点：简单方便 ②缺点：a、局限于单向通信；b、只能创建在它的进程以及其有亲缘关系的进程之间；c、缓冲区有限

信号量（Semaphore）：一个计数器，可以用来控制多个线程对共享资源的访问 ①优点：可以同步进程 ②缺点：信号量有限

信号（Signal）：一种比较复杂的通信方式，用于通知接收进程某个事件已经发生

消息队列（Message Queue）：是消息的链表，存放在内核中并由消息队列标识符标识 ①优点：可以实现任意进程间的通信，并通过系统调用函数来实现消息发送和接收之间的同步，无需考虑同步问题，方便 ②缺点：信息的复制需要额外消耗 CPU 的时间，不适宜于信息量大或操作频繁的场合

共享内存（Shared Memory）：映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问 ①优点：无须复制，快捷，信息量大 ②缺点：a、通信是通过将共享空间缓冲区直接附加到进程的虚拟地址空间中来实现的，因此进程间的读写操作的同步问题；b、利用内存缓冲区直接交换信息，内存的实体存在于计算机中，只能同一个计算机系统中的诸多进程共享，不方便网络通信

套接字（Socket）：可用于不同计算机间的进程通信 ①优点： a、传输数据为字节级，传输数据可自定义，数据量小效率高； b、传输数据时间短，性能高； c、适合于客户端和服务器端之间信息实时交互； d、可以加密,数据安全性强； ②缺点：需对传输的数据进行解析，转化成应用级的数据。 更多详见 C++基础语法梳理：进程与线程！知识点详细梳理

7、线程之间的通信方式

锁机制 包括互斥锁量（mutex）、读写锁（reader-writer lock）、自旋锁（spin lock）、条件变量（condition）

互斥锁量（mutex）：提供了以排他方式防止数据结构被并发修改的方法。

读写锁（reader-writer lock）：允许多个线程同时读共享数据，而对写操作是互斥的。

自旋锁（spin lock）与互斥锁类似，都是为了保护共享资源。互斥锁是当资源被占用，申请者进入睡眠状态；而自旋锁则循环检测保持者是否已经释放锁。

条件变量（condition）：可以以原子的方式阻塞进程，直到某个特定条件为真为止。对条件的测试是在互斥锁的保护下进行的。条件变量始终与互斥锁一起使用。

信号量机制(Semaphore)

无名线程信号量

命名线程信号量 信号机制(Signal)：类似进程间的信号处理 屏障（barrier）：屏障允许每个线程等待，直到所有的合作线程都达到某一点，然后从该点继续执行。 线程间的通信目的主要是用于线程同步，所以线程没有像进程通信中的用于数据交换的通信机制

第五部分、 STL

1、STL 六大组件

STL 六大组件：容器（Container）、算法（Algorithm）、迭代器（Iterator）、仿函数（Function object）、适配器（Adaptor）和 空间配置器（allocator）。

2、stack 中有 pop() 和 top() 方法，为什么不直接用 pop() 实现弹出和取值的功能？

把pop()和top()的功能放在一个函数中，如果 stack 中存放的是较大是内容时，比如 vector 类型，取值的时候就会发生拷贝，如果拷贝失败，要弹出的数据将会丢失；就很可能出现下述的bug，导致原始数据丢失。 假设有一个stack，vector是一个动态容器，当你拷贝一个vector时，标准库会从堆上分配很多内存来完成这次拷贝。当这个系统处在重度负荷，或有严重的资源限制的情况下，这种内存分配就会失败，所以vector的拷贝构造函数可能会抛出一个stdbad\_alloc异常。当vector中存有大量元素时，这种情况发生的可能性更大。当pop()函数返回“弹出值”时(也就是从栈中将这个值移除)，会有一个潜在的问题：这个值被返回到调用函数的时候，栈才被改变；但当拷贝数据的时候，调用函数抛出一个异常会怎么样？如果事情真的发生了，要弹出的数据将会丢失；它的确从栈上移出了，但是拷贝失败了！stdstack的设计人员将这个操作分为两个部分：先获取顶部元素(top())，然后从栈中移除元素(pop())。这样，在不能安全的将元素拷贝出去的情况下，栈中的这个数据还依旧存在，没有丢失。当问题是堆空间不足时，应用可能会释放一些内存，然后再进行尝试。 参考： 为什么适配器stack中成员函数top()和pop()需要分离实现

3、 STL库用过吗？常见的STL容器有哪些？算法用过几个？

STL包括两部分内容：容器和算法 容器即存放数据的地方，比如array, vector，分为两类，序列式容器和关联式容器 序列式容器，其中的元素不一定有序，但是都可以被排序，比如vector,list,queue,stack，heap, priority-queue, slist 关联式容器，内部结构是一个平衡二叉树，每个元素都有一个键值和一个实值，比如map, set, hashtable, hash\_set 算法有排序，复制等，以及各个容器特定的算法 迭代器是STL的精髓，迭代器提供了一种方法，使得它能够按照顺序访问某个容器所含的各个元素，但无需暴露该容器的内部结构，它将容器和算法分开，让二者独立设计。 Vector是顺序容器，是一个动态数组，支持随机存取、插入、删除、查找等操作，在内存中是一块连续的空间。在原有空间不够情况下自动分配空间，增加为原来的两倍。vector随机存取效率高，但是在vector插入元素，需要移动的数目多，效率低下。 注意：vector动态增加大小时，并不是在原空间之后持续新空间（因为无法保证原空间之后尚有可供配置的空间），而是以原大小的两倍另外配置一块较大的空间，然后将原内容拷贝过来，然后才开始在原内容之后构造新元素，并释放原空间。因此，对vector的任何操作，一旦引起空间重新配置，指向原vector的所有迭代器就都失效了。

4、STL中map 、set、multiset、multimap的底层原理（关联式容器）

map 、set 、multiset 、multimap 的底层实现都是红黑树。 红 黑 树 的 特 性 ：

每个结点或是红色或是黑色；

根结点是黑色；

每个叶结点是黑的；

如果一个结点是红的，则它的两个儿子均是黑色；

每个结点到其子孙结点的所有路径上包含相同数目的黑色结点。

5、map 、set、multiset、multimap的特点

set和multiset会根据特定的排序准则自动将元素排序，set中元素不允许重复，multiset可以重复。

map和multimap将key和value组成的pair作为元素，根据key的排序准则自动将元素排序（因为红黑树也是二叉搜索树，所以map默认是按key排序的），map中元素的key不允许重复，multimap可以重复。

map和set的增删改查速度为都是logn，是比较高效的。

6、hash\_map与map的区别？什么时候用hash\_map，什么时候用map？

构造函数：hash\_map需要hash function和等于函数，而map需要比较函数（大于或小于）。 存储结构：hash\_map以hashtable为底层，而map以RB-TREE为底层。 总的说来，hash\_map查找速度比map快，而且查找速度基本和数据量大小无关，属于常数级别。而map的查找速度是logn级别。但不一定常数就比log小，而且hash\_map还有hash function耗时。 如果考虑效率，特别当元素达到一定数量级时，用hash\_map。 考虑内存，或者元素数量较少时，用map。

7、STL中unordered\_map和map的区别

map是STL中的一个关联容器，提供键值对的数据管理。底层通过红黑树来实现，实际上是二叉排序树和非严格意义上的二叉平衡树。所以在map内部所有的数据都是有序的，且map的查询、插入、删除操作的时间复杂度都是O(logN)。 unordered\_map和map类似，都是存储key-value对，可以通过key快速索引到value，不同的是unordered\_map不会根据key进行排序。unordered\_map底层是一个防冗余的哈希表，存储时根据key的hash值判断元素是否相同，即unoredered\_map内部是无序的。

8、STL中的vector的实现，是怎么扩容的？

vector使用的注意点及其原因，频繁对vector调用push\_back()对性能的影响和原因。 vector就是一个动态增长的数组，里面有一个指针指向一片连续的空间，当空间装不下的时候，会申请一片更大的空间，将原来的数据拷贝过去，并释放原来的旧空间。当删除的时候空间并不会被释放，只是清空了里面的数据。对比array是静态空间一旦配置了就不能改变大小。 vector的动态增加大小的时候，并不是在原有的空间上持续新的空间（无法保证原空间的后面还有可供配置的空间），而是以原大小的两倍另外配置一块较大的空间，然后将原内容拷贝过来，并释放原空间。在VS下是1.5倍扩容，在GCC下是2倍扩容。

9、C++中vector和list的区别

vector和数组类似，拥有一段连续的内存空间。vector申请的是一段连续的内存，当插入新的元素内存不够时，通常以2倍重新申请更大的一块内存，将原来的元素拷贝过去，释放旧空间。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，时间复杂度为o(n)。 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。 vector拥有一段连续的内存空间，能很好的支持随机存取，因此vectoriterator支持“+”，“+=”，“”等操作符。 list的内存空间可以是不连续，它不支持随机访问，因此listiterator则不支持“+”、“+=”、“”等 vectoriterator和listiterator都重载了“++”运算符。 总之，如果需要高效的随机存取，而不在乎插入和删除的效率，使用vector; 如果需要大量的插入和删除，而不关心随机存取，则应使用list。

10、STL内存优化？

STL内存管理使用二级内存配置器。

第一级配置器： 第一级配置器以malloc()，free()，realloc()等C函数执行实际的内存配置、释放、重新配置等操作，并且能在内存需求不被满足的时候，调用一个指定的函数。一级空间配置器分配的是大于128字节的空间，如果分配不成功，调用句柄释放一部分内存，如果还不能分配成功，抛出异常。 第一级配置器只是对malloc函数和free函数的简单封装，在allocate内调用malloc，在deallocate内调用free。同时第一级配置器的oom\_malloc函数，用来处理malloc失败的情况。

第二级配置器： 第一级配置器直接调用malloc和free带来了几个问题： 内存分配释放的效率低 当配置大量的小内存块时，会导致内存碎片比较严重 配置内存时，需要额外的部分空间存储内存块信息，所以配置大量的小内存块时，还会导致额外内存负担 如果分配的区块小于128bytes，则以内存池管理，第二级配置器维护了一个自由链表数组，每次需要分配内存时，直接从相应的链表上取出一个内存节点就完成工作，效率很高。 自由链表数组：自由链表数组其实就是个指针数组，数组中的每个指针元素指向一个链表的起始节点。数组大小为16，即维护了16个链表，链表的每个节点就是实际的内存块，相同链表上的内存块大小都相同，不同链表的内存块大小不同，从8一直到128。如下所示，obj为链表上的节点，free\_list就是链表数组。 内存分配：allocate函数内先判断要分配的内存大小，若大于128字节，直接调用第一级配置器，否则根据要分配的内存大小从16个链表中选出一个链表，取出该链表的第一个节点。若相应的链表为空，则调用refill函数填充该链表。默认是取出20个数据块。 填充链表 refill：若allocate函数内要取出节点的链表为空，则会调用refill函数填充该链表。refill函数内会先调用chunk\_alloc函数从内存池分配一大块内存，该内存大小默认为20个链表节点大小，当内存池的内存也不足时，返回的内存块节点数目会不足20个。接着refill的工作就是将这一大块内存分成20份相同大小的内存块，并将各内存块连接起来形成一个链表。 内存池：chunk\_alloc函数内管理了一块内存池，当refill函数要填充链表时，就会调用chunk\_alloc函数，从内存池取出相应的内存。 在chunk\_alloc函数内首先判断内存池大小是否足够填充一个有20个节点的链表，若内存池足够大，则直接返回20个内存节点大小的内存块给refill； 若内存池大小无法满足20个内存节点的大小，但至少满足1个内存节点，则直接返回相应的内存节点大小的内存块给refill； 若内存池连1个内存节点大小的内存块都无法提供，则chunk\_alloc函数会将内存池中那一点点的内存大小分配给其他合适的链表，然后去调用malloc函数分配的内存大小为所需的两倍。若malloc成功，则返回相应的内存大小给refill；若malloc失败，会先搜寻其他链表的可用的内存块，添加到内存池，然后递归调用chunk\_alloc函数来分配内存，若其他链表也无内存块可用，则只能调用第一级空间配置器。

11、正确释放vector的内存(clear(), swap(), shrink\_to\_fit())

vec.clear()：清空内容，但是不释放内存。 vector().swap(vec)：清空内容，且释放内存，想得到一个全新的vector。 vec.shrink\_to\_fit()：请求容器降低其capacity和size匹配。 vec.clear();vec.shrink\_to\_fit();：清空内容，且释放内存。

12、什么情况下用vector，什么情况下用list，什么情况下用deque

vector可以随机存储元素（即可以通过公式直接计算出元素地址，而不需要挨个查找），但在非尾部插入删除数据时，效率很低，适合对象简单，对象数量变化不大，随机访问频繁。除非必要，我们尽可能选择使用vector而非deque，因为deque的迭代器比vector迭代器复杂很多。

list不支持随机存储，适用于对象大，对象数量变化频繁，插入和删除频繁，比如写多读少的场景。

需要从首尾两端进行插入或删除操作的时候需要选择deque。

13、priority\_queue的底层原理

priority\_queue：优先队列，其底层是用堆来实现的。在优先队列中，队首元素一定是当前队列中优先级最高的那一个

14、 STL线程不安全的情况

在对同一个容器进行多线程的读写、写操作时；

在每次调用容器的成员函数期间都要锁定该容器；

在每个容器返回的迭代器（例如通过调用begin或end）的生存期之内都要锁定该容器；

在每个在容器上调用的算法执行期间锁定该容器。

推荐阅读： C++ STL容器如何解决线程安全的问题？

15、vector 中迭代器失效的情况

当插入（push\_back）一个元素后，end操作返回的迭代器肯定失效。

当插入(push\_back)一个元素后，capacity返回值与没有插入元素之前相比有改变，则需要重新加载整个容器，此时first和end操作返回的迭代器都会失效。

当进行删除操作（erase，pop\_back）后，指向删除点的迭代器全部失效；指向删除点后面的元素的迭代器也将全部失效。

16、迭代器的几种失效的情况

vector、deque，该数据结构分配在连续的内存中，当删除一个元素后，内存中的数据会发生移动以保证数据的紧凑。所以删除一个元素后，其他数据的地址发生了变化，之前获取的迭代器根据原有信息就访问不到正确的数据 解决方法：erase返回下一个有效迭代器的值

map、set、multiset、map、multimap，树形数据结构。以红黑树或者平衡二叉树组织数据，虽然删除了一个元素，整棵树也会调整，以符合红黑树或者二叉树的规范，但是单个节点在内存中的地址没有变化，变化的是各节点之间的指向关系。删除一个结点不会对其他结点造成影响。 erase迭代器只是被删元素的迭代器失效。

list，链表型数据结构。双向链表，使用不连续分配的内存，删除运算使指向删除位置的迭代器失效，不会使其他迭代器失效。

17、STL 是复制性还是侵入性

STL实现的容器都是非侵入式容器，通过模板类可以放入任何类型的结构，与浸入式容器的最大不同是，C++的非侵入式容器必须存储用户数据的拷贝 参考： STL容器概述

18、vector使用注意事项

在push\_back()、resize()、insert()后有可能引起重新分配内存，此时原始的迭代器失效，需要重新生成一次迭代器

为了防止多次扩容造成性能低，可以先使用reserve()，预留足够的空间，最后在使用 shrink\_to\_fit()收缩空间

vector不能用来存储bool类型的元素，存储机理不同，bool类型的元素存储到vector中，会转化为1个bit，不是1个byte，可以用deque或者是bitset存储bool类型的

vector的扩容：VS是1.5倍、GCC是2倍

第六部分、 网络编程

1、三次握手和四次挥手

参考 详解 TCP 连接的“三次握手”与“四次挥手”

2、 TCP 和 UDP 的区别

TCP面向连接（如打电话要先拨号建立连接）;UDP是无连接的，即发送数据之前不需要建立连接

TCP提供可靠的服务。也就是说，通过TCP连接传送的数据，无差错，不丢失，不重复，且按序到达;UDP尽最大努力交付，即不保 证可靠交付

TCP面向字节流，实际上是TCP把数据看成一连串无结构的字节流;UDP是面向报文的 UDP没有拥塞控制，因此网络出现拥塞不会使源主机的发送速率降低（对实时应用很有用，如IP电话，实时视频会议等）

每一条TCP连接只能是点到点的;UDP支持一对一，一对多，多对一和多对多的交互通信

TCP首部开销20字节;UDP的首部开销小，只有8个字节

TCP的逻辑通信信道是全双工的可靠信道，UDP则是不可靠信道 详见： TCP和UDP的最完整的区别

3、TCP 粘包 、拆包

如果应用一次请求发送的数据量比较小，没达到缓冲区大小，TCP则会将多个请求合并为同一个请求进行发送，站在业务上来看这就是「粘包」; 如果应用一次请求发送的数据量比较大，超过了缓冲区大小，TCP就会将其拆分为多次发送，这就是「拆包」，也就是将一个大的包拆分为多个小包进行发送。 常见的解决思路如下：

发送端给每个数据包添加包首部。

发送端将每个数据包封装为固定长度。

可以在数据包之间设置边界。 详见： 一分钟看懂TCP粘包拆包

4、TCP 丢包

服务器给客户端发大量数据，Send的频率很高，那么就有可能在Send时发生错误（原因可能是又多种，可能是程序处理逻辑问题，多线程同步问题，缓冲区溢出问题等等），如果没有对Send失败做处理重发数据，那么客户端收到的数据就会比理论应该收到的少，就会造成丢数据，丢包的现象。 这种现象，其实本质上来说不是丢包，也不是丢数据，只是因为程序处理有错误，导致有些数据没有成功地被socket发送出去。 常用的解决方法如下： 拆包、加包头、发送，组合包，如果客户端、服务端掉线，常采用心跳测试。 详见： TCP通信丢包原因总结

5、为什么“握手”是三次，“挥手”却要四次？

TCP建立连接时之所以只需要三次握手，是因为在第二次握手过程中，服务器端发送给客户端的TCP报文是以SYN与ACK作为标志位的。SYN是请求连接标志，表示服务器端同意建立连接；ACK是确认报文，表示告诉客户端，服务器端收到了它的请求报文。 即SYN建立连接报文与ACK确认接收报文是在同一次握手当中传输的，所以三次握手不多也不少，正好让双方明确彼此信息互通。 TCP释放连接时之所以需要“四次挥手”,是因为FIN释放连接报文与ACK确认接收报文是分别由第二次和第三次握手传输的。为何建立连接时一起传输，释放连接时却要分开传输？

建立连接时，被动方服务器端结束CLOSED阶段进入“握手”阶段并不需要任何准备，可以直接返回SYN和ACK报文，开始建立连接。

释放连接时，被动方服务器，突然收到主动方客户端释放连接的请求时并不能立即释放连接，因为还有必要的数据需要处理，所以服务器先返回ACK确认收到报文，经过CLOSE-WAIT阶段准备好释放连接之后，才能返回FIN释放连接报文。

所以是“三次握手”，“四次挥手”。

6、为什么“握手”是三次，“挥手”却要四次？

TCP建立连接时之所以只需要三次握手，是因为在第二次握手过程中，服务器端发送给客户端的TCP报文是以SYN与ACK作为标志位的。SYN是请求连接标志，表示服务器端同意建立连接；ACK是确认报文，表示告诉客户端，服务器端收到了它的请求报文。 即SYN建立连接报文与ACK确认接收报文是在同一次握手当中传输的，所以三次握手不多也不少，正好让双方明确彼此信息互通。

TCP释放连接时之所以需要“四次挥手”,是因为FIN释放连接报文与ACK确认接收报文是分别由第二次和第三次握手传输的。为何建立连接时一起传输，释放连接时却要分开传输？ 建立连接时，被动方服务器端结束CLOSED阶段进入“握手”阶段并不需要任何准备，可以直接返回SYN和ACK报文，开始建立连接。

释放连接时，被动方服务器，突然收到主动方客户端释放连接的请求时并不能立即释放连接，因为还有必要的数据需要处理，所以服务器先返回ACK确认收到报文，经过CLOSE-WAIT阶段准备好释放连接之后，才能返回FIN释放连接报文。

所以是“三次握手”，“四次挥手”。

7、网络的七层模型，作用、传输单位分别是什么

整个模型分为七层，物理层，数据链路层，网络层，传输层，会话层，表示层，应用层。

详见： OSI七层模型及各层作用

第七部分、 设计模式

1、设计模式

创建型模式 对象实例化的模式，创建型模式用于解耦对象的实例化过程。

单例模式：某个类智能有一个实例，提供一个全局的访问点。

工厂模式：一个工厂类根据传入的参量决定创建出哪一种产品类的实例。

抽象工厂模式：创建相关或依赖对象的家族，而无需明确指定具体类。

建造者模式：封装一个复杂对象的创建过程，并可以按步骤构造。

原型模式：通过复制现有的实例来创建新的实例。

结构型模式 把类或对象结合在一起形成一个更大的结构。

装饰器模式：动态的给对象添加新的功能。

代理模式：为其它对象提供一个代理以便控制这个对象的访问。

桥接模式：将抽象部分和它的实现部分分离，使它们都可以独立的变化。

适配器模式：将一个类的方法接口转换成客户希望的另一个接口。

组合模式：将对象组合成树形结构以表示“部分-整体”的层次结构。

外观模式：对外提供一个统一的方法，来访问子系统中的一群接口。

享元模式：通过共享技术来有效的支持大量细粒度的对象。

行为型模式 类和对象如何交互，及划分责任和算法。

策略模式：定义一系列算法，把他们封装起来，并且使它们可以相互替换。

模板模式：定义一个算法结构，而将一些步骤延迟到子类实现。

命令模式：将命令请求封装为一个对象，使得可以用不同的请求来进行参数化。

迭代器模式：一种遍历访问聚合对象中各个元素的方法，不暴露该对象的内部结构。

观察者模式：对象间的一对多的依赖关系。

仲裁者模式：用一个中介对象来封装一系列的对象交互。

备忘录模式：在不破坏封装的前提下，保持对象的内部状态。

解释器模式：给定一个语言，定义它的文法的一种表示，并定义一个解释器。

状态模式：允许一个对象在其对象内部状态改变时改变它的行为。

责任链模式：将请求的发送者和接收者解耦，使的多个对象都有处理这个请求的机会。

访问者模式：不改变数据结构的前提下，增加作用于一组对象元素的新功能。

2、单例模式的线程安全问题

饿汉模式的单例模式本身就是现场安全的。懒汉模式的线程安全见 C++单例模式与线程安全 推荐阅读： c++的单例模式为什么不直接全部使用static，而是非要实例化一个对象？

3、工厂方法模式

简单工厂模式 简单工厂模式属于类的创建型模式,又叫做静态工厂方法模式。通过专门定义一个类来负责创建其他类的实例，被创建的实例通常都具有共同的父类。简单工厂模式可以减少客户程序对类创建过程的依赖。 优点：

帮助封装 实现组件封装，面向接口编程

解耦合客户端和具体实现类的解耦合

缺点：

可能增加客户端的复杂度

不方便扩展子工厂

工厂模式 工厂方法模式同样属于类的创建型模式又被称为多态工厂模式 。工厂方法模式的意义是 定义一个创建产品对象的工厂接口，将实际创建工作推迟到子类当中。 核心工厂类不再负责产品的创建，这样核心类成为一个抽象工厂角色，仅负责具体工厂子类 必须实现的接口，这样进一步抽象化的好处是使得工厂方法模式可以使系统在不修改具体工厂角色的情况下引进新的产品 优点：

需求改变时改动最小

具体的创建实例过程与客户端分离

缺点：

新增功能时，工程量稍大

抽象工厂模式 抽象工厂模式是所有形态的工厂模式中最为抽象和最一般性的。抽象工厂模式可以向客户端提供一个接口，使得客户端在不必指定产品的具体类型的情况下，能够创建多个产品族的产品对象。

抽象工厂方法是针对与一个产品族，使得易于交换产品系列，只需改变具体的工厂就可以使用不同的产品配置。当一个族中的产品对象被设计成一起工作且一个应用只适用同一族的对象，例如设计系统生成不同风格的UI界面，按钮，边框等UI元素在一起使用，并且只能同属于一种风格，这很容易使用抽象工厂实现。 优点：

抽象工厂封装了变化，封装了对象创建的具体细节

增加新的产品族很方便，无须修改已有系统

针对接口进行编程而不是针对具体进行编程

缺点：

增加新的产品等级结构需对原系统做较大修改(违背开放封闭)

详见： C++工厂模式（简单工厂、工厂方法、抽象工厂） 详见： C++设计模式：三种工厂模式详解（简单工厂，工厂模式，抽象工厂）

第八部分、 数据结构

1、双链表和单链表的优缺点

单向链表：单向链表包含两个域，一个是信息域，一个是指针域。也就是单向链表的节点被分成两部分，一部分是保存或显示关于节点的信息，第二部分存储下一个节点的地址，而最后一个节点则指向一个空值。 优点：单向链表增加删除节点简单。遍历时候不会死循环。（双向也不会死循环，循环链表忘了进行控制的话很容易进入死循环）；缺点：只能从头到尾遍历。只能找到后继，无法找到前驱，也就是只能前进。

双向链表：每个节点有2个链接，一个是指向前一个节点（当此链接为第一个链接时，指向的是空值或空列表），另一个则指向后一个节点（当此链接为最后一个链接时，指向的是空值或空列表）。意思就是说双向链表有2个指针，一个是指向前一个节点的指针，另一个则指向后一个节点的指针。 优点：可以找到前驱和后继，可进可退；缺点：增加删除节点复杂。

2、红黑树比AVL的优势，为何用红黑树

红黑树的查询性能略微逊色于AVL树，因为他比avl树会稍微不平衡最多一层，也就是说红黑树的查询性能只比相同内容的avl树最多多一次比较，但是，红黑树在插入和删除上完爆avl树， avl树每次插入删除会进行大量的平衡度计算，而红黑树为了维持红黑性质所做的红黑变换和旋转的开销，相较于avl树为了维持平衡的开销要小得多

3、红黑树的高度

从任何一个节点到其最远的后代叶子的节点数不超过到最近的后代叶子的节点数的两倍。 每个具有 n 个节点的红黑树的高度 = 2Log 2 (n+1) 详见 红黑树详解 详见 红黑树 (Red-Black Tree) – 介绍

4、判断链表是不是环形链表

基于循环次数判断的暴力方法

如果一个链表不成环，那么它就是可以穷尽的，我们假定你的链表的节点的个数小于10^4个，如果比这个还多的话，也许你需要用到树型结构了，不然你的速度会比乌龟还慢。因此我们可以一直循环，循环的时候，要么是因为到达了指针的尾部NULL，要么是因为循环次数太多超越了节点的个数，循环结束时判断循环次数就可以了。

借助Hash表

每次访问节点的时候都判断是否在hash表中，如果没有就加入，如果存在就说明有环，如果没有环，一个节点只可能被访问一次,这和我们企图修改链表的数据结构去维护访问次数是等价的，只是以一种不影响节点内容的方式去完成。

快慢指针

定义两种类型的指针，slow慢指针前进步长为1，快指针前进步长为2（在第一次前进后要判断前进后是否到达了NULL）。这样，如果在循环的过程中，快指针追上了慢指针，那么说明成了环。

5、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