### **1.2 C++内存**

#### **1.2.1 简述一下堆和栈的区别**

****参考回答****

区别：

****堆栈空间分配不同****。栈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；堆一般由程序员分配释放。

****堆栈缓存方式不同****。栈使用的是一级缓存， 它们通常都是被调用时处于存储空间中，调用完毕立即释放；堆则是存放在二级缓存中，速度要慢些。

****堆栈数据结构不同****。堆类似数组结构；栈类似栈结构，先进后出。

#### **1.2.2 简述C++的内存管理**

****参考回答****

****内存分配方式****：

在C++中，内存分成5个区，他们分别是堆、栈、自由存储区、全局/静态存储区和常量存储区。

****栈****，在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。

****堆****，就是那些由new分配的内存块，一般一个new就要对应一个delete。

****自由存储区****，就是那些由malloc等分配的内存块，和堆是十分相似的，不过是用free来结束自己的生命。

****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

****常量存储区****，这是一块比较特殊的存储区，里面存放的是常量，不允许修改。

****常见的内存错误及其对策****：

（1）内存分配未成功，却使用了它。

（2）内存分配虽然成功，但是尚未初始化就引用它。

（3）内存分配成功并且已经初始化，但操作越过了内存的边界。

（4）忘记了释放内存，造成内存泄露。

（5）释放了内存却继续使用它。

对策：

（1）定义指针时，先初始化为NULL。

（2）用malloc或new申请内存之后，应该****立即检查****指针值是否为NULL。防止使用指针值为NULL的内存。

（3）不要忘记为数组和动态内存****赋初值****。防止将未被初始化的内存作为右值使用。

（4）避免数字或指针的下标****越界****，特别要当心发生“多1”或者“少1”操作

（5）动态内存的申请与释放必须配对，防止****内存泄漏****

（6）用free或delete释放了内存之后，立即将指针****设置为NULL****，防止“野指针”

（7）使用智能指针。

****内存泄露及解决办法****：

****什么是内存泄露？****

简单地说就是申请了一块内存空间，使用完毕后没有释放掉。（1）new和malloc申请资源使用后，没有用delete和free释放；（2）子类继承父类时，父类析构函数不是虚函数。（3）Windows句柄资源使用后没有释放。

****怎么检测？****

第一：良好的编码习惯，使用了内存分配的函数，一旦使用完毕,要记得使用其相应的函数释放掉。

第二：将分配的内存的指针以链表的形式自行管理，使用完毕之后从链表中删除，程序结束时可检查改链表。

第三：使用智能指针。

第四：一些常见的工具插件，如ccmalloc、Dmalloc、Leaky、Valgrind等等。

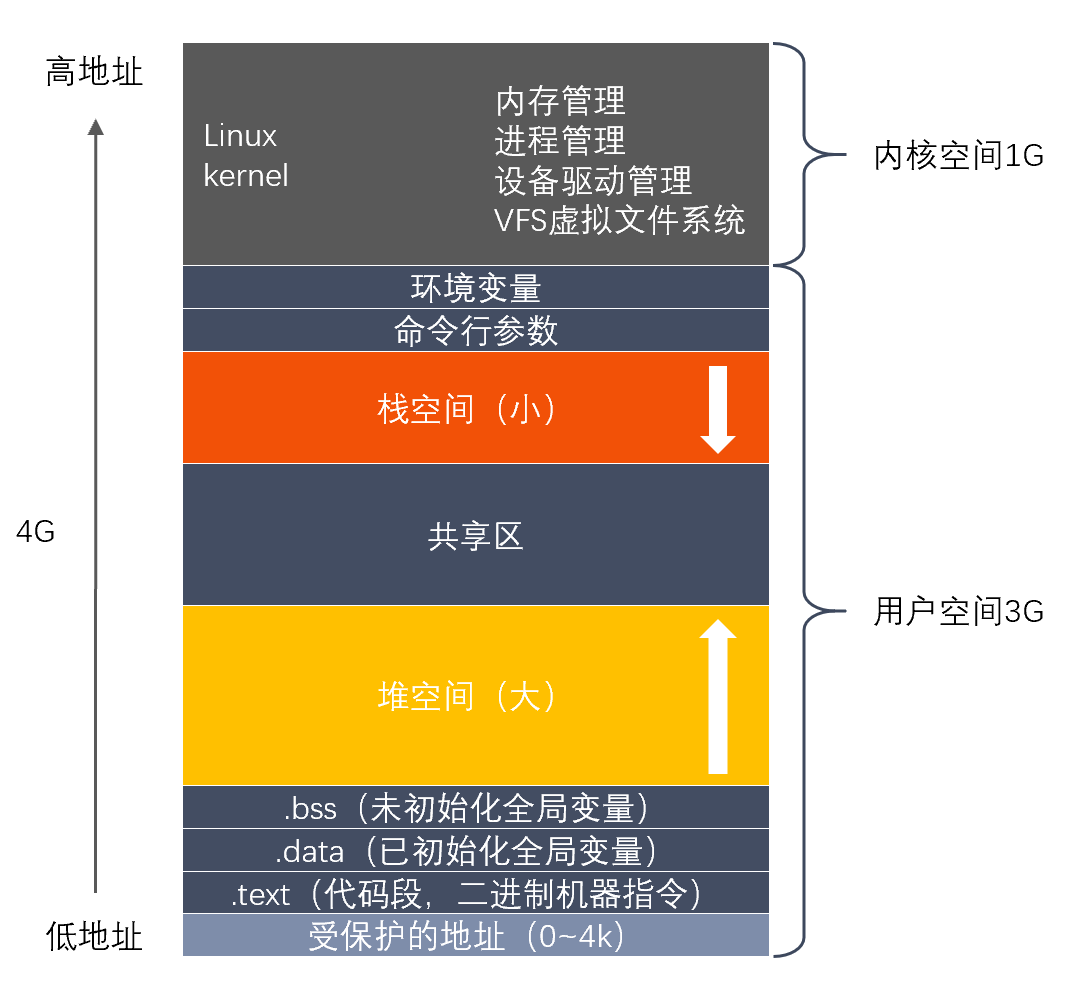
#### **1.2.3 malloc和局部变量分配在堆还是栈？**

****参考回答****

        malloc是在****堆上分配内存****，需要程序员自己回收内存；局部变量是在****栈中分配内存****，超过作用域就自动回收。

#### **1.2.4 程序有哪些section，分别的作用？程序启动的过程？怎么判断数据分配在栈上还是堆上？**

****参考回答****



****一个程序有哪些section：****

如上图，****从低地址到高地址，一个程序由代码段、数据段、BSS段、堆、共享区、栈等****组成。

****数据段：****存放程序中已初始化的全局变量和静态变量的一块内存区域。

****代码段：****存放程序执行代码的一块内存区域。只读，代码段的头部还会包含一些只读的常数变量。

****BSS**** 段：存放程序中未初始化的全局变量和静态变量的一块内存区域。

可执行程序在运行时又会多出两个区域：堆区和栈区。

****堆区：****动态申请内存用。堆从低地址向高地址增长。

****栈区：****存储局部变量、函数参数值。栈从高地址向低地址增长。是一块连续的空间。

最后还有一个****共享区****，位于堆和栈之间。

****程序启动的过程：****

操作系统首先创建相应的进程并分配私有的进程空间，然后操作系统的加载器负责把可执行文件的数据段和代码段映射到进程的虚拟内存空间中。

加载器读入可执行程序的导入符号表，根据这些符号表可以查找出该可执行程序的所有依赖的动态链接库。

加载器针对该程序的每一个动态链接库调用LoadLibrary （1）查找对应的动态库文件，加载器为该动态链接库确定一个合适的基地址。 （2）加载器读取该动态链接库的导入符号表和导出符号表，比较应用程序要求的导入符号是否匹配该库的导出符号。 （3）针对该库的导入符号表，查找对应的依赖的动态链接库，如有跳转，则跳到3 （4）调用该动态链接库的初始化函数

初始化应用程序的全局变量，对于全局对象自动调用构造函数。

进入应用程序入口点函数开始执行。

****怎么判断数据分配在栈上还是堆上：****首先局部变量分配在栈上；而通过malloc和new申请的空间是在堆上。

#### **1.2.5 初始化为0的全局变量在bss还是data**

****参考回答****

BSS段通常是指用来存放程序中未初始化的或者初始化为0的全局变量和静态变量的一块内存区域。特点是可读写的，在程序执行之前BSS段会自动清0。

#### **1.2.6 什么是内存泄露，内存泄露怎么检测？**

****参考回答****

****什么是内存泄露？****

简单地说就是申请了一块内存空间，使用完毕后没有释放掉。（1）new和malloc申请资源使用后，没有用delete和free释放；（2）子类继承父类时，父类析构函数不是虚函数。（3）Windows句柄资源使用后没有释放。

****怎么检测？****

第一：良好的编码习惯，使用了内存分配的函数，一旦使用完毕,要记得使用其相应的函数释放掉。

第二：将分配的内存的指针以链表的形式自行管理，使用完毕之后从链表中删除，程序结束时可检查改链表。

第三：使用智能指针。

第四：一些常见的工具插件，如ccmalloc、Dmalloc、Leaky、Valgrind等等。

#### **1.2.7 请简述一下atomoic内存顺序。**

****参考回答****

有六个内存顺序选项可应用于对原子类型的操作：

memory\_order\_relaxed：在原子类型上的操作以自由序列执行，没有任何同步关系，仅对此操作要求原子性。

memory\_order\_consume：memory\_order\_consume只会对其标识的对象保证该对象存储先行于那些需要加载该对象的操作。

memory\_order\_acquire：使用memory\_order\_acquire的原子操作，当前线程的读写操作都不能重排到此操作之前。

memory\_order\_release：使用memory\_order\_release的原子操作，当前线程的读写操作都不能重排到此操作之后。

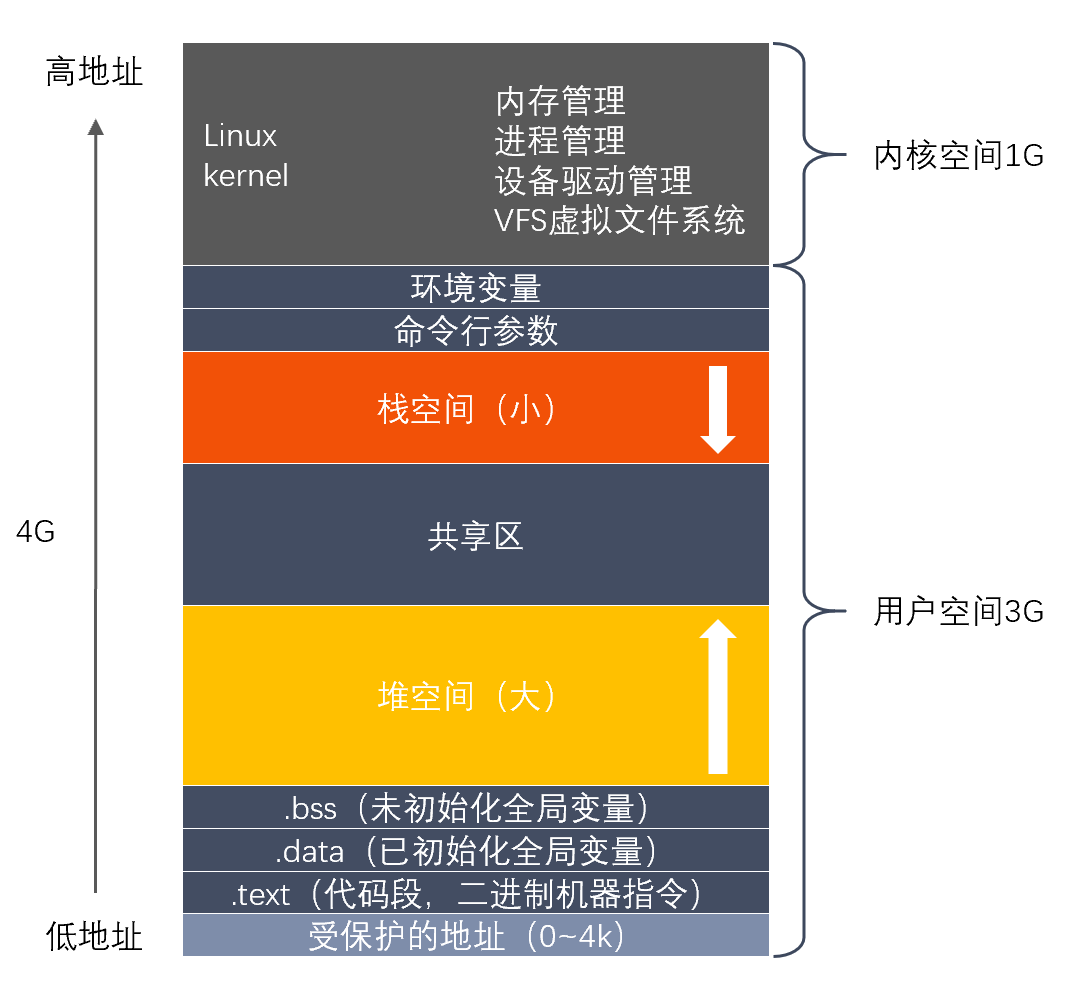
memory\_order\_acq\_rel：memory\_order\_acq\_rel在此内存顺序的读-改-写操作既是获得加载又是释放操作。没有操作能够从此操作之后被重排到此操作之前，也没有操作能够从此操作之前被重排到此操作之后。

memory\_order\_seq\_cst：memory\_order\_seq\_cst比std::memory\_order\_acq\_rel更为严格。memory\_order\_seq\_cst不仅是一个"获取释放"内存顺序，它还会对所有拥有此标签的内存操作建立一个单独全序。

除非你为特定的操作指定一个顺序选项，否则内存顺序选项对于所有原子类型默认都是memory\_order\_seq\_cst。

#### **1.2.8 内存模型，堆栈，常量区。**

****参考回答****



****内存模型（内存布局）：****

如上图，****从低地址到高地址，一个程序由代码段、数据段、BSS段、堆、共享区、栈等****组成。

****数据段：****存放程序中已初始化的全局变量和静态变量的一块内存区域。

****代码段：****存放程序执行代码的一块内存区域。只读，代码段的头部还会包含一些只读的常数变量。

****BSS**** 段：存放程序中未初始化的全局变量和静态变量的一块内存区域。

可执行程序在运行时又会多出两个区域：堆区和栈区。

****堆区：****动态申请内存用。堆从低地址向高地址增长。

****栈区：****存储局部变量、函数参数值。栈从高地址向低地址增长。是一块连续的空间。

最后还有一个****共享区****，位于堆和栈之间。

****堆 heap**** ：由new分配的内存块，其释放由程序员控制（一个new对应一个delete）

****栈 stack**** ：是那些编译器在需要时分配，在不需要时自动清除的存储区。存放局部变量、函数参数。

****常量存储区**** ：存放常量，不允许修改。

#### **1.2.9 简述C++中内存对齐的使用场景**

****参考回答****

内存对齐应用于三种数据类型中：****struct/class/union****

struct/class/union内存对齐原则有四个：

数据成员对齐规则：结构(struct)或联合(union)的数据成员，第一个数据成员放在offset为0的地方，以后每个数据成员存储的起始位置要从该成员大小或者成员的子成员大小的整数倍开始。

结构体作为成员:如果一个结构里有某些结构体成员,则结构体成员要从其内部"最宽基本类型成员"的整数倍地址开始存储。(struct a里存有struct b,b里有char,int ,double等元素,那b应该从8的整数倍开始存储)。

收尾工作:结构体的总大小，也就是sizeof的结果，必须是其内部最大成员的"最宽基本类型成员"的整数倍。不足的要补齐。(基本类型不包括struct/class/uinon)。

sizeof(union)，以结构里面size最大元素为union的size，因为在某一时刻，union只有一个成员真正存储于该地址。

****答案解析****

****什么是内存对齐？****

那么什么是字节对齐？在C语言中，结构体是一种复合数据类型，其构成元素既可以是基本数据类型（如int、long、float等）的变量，也可以是一些复合数据类型（如数组、结构体、联合体等）的数据单元。在结构体中，****编译器为结构体的每个成员按其自然边界（alignment）分配空间。****各个成员按照它们被声明的顺序在内存中顺序存储，第一个成员的地址和整个结构体的地址相同。

为了使CPU能够对变量进行快速的访问，变量的起始地址应该具有某些特性，****即所谓的“对齐”，比如4字节的int型，其起始地址应该位于4字节的边界上，即起始地址能够被4整除****，也即“对齐”跟数据在内存中的位置有关。如果一个变量的内存地址正好位于它长度的整数倍，他就被称做自然对齐。

比如在32位cpu下，假设一个整型变量的地址为0x00000004(为4的倍数)，那它就是自然对齐的，而如果其地址为0x00000002（非4的倍数）则是非对齐的。现代计算机中内存空间都是按照byte划分的，从理论上讲似乎对任何类型的变量的访问可以从任何地址开始，但实际情况是在访问特定类型变量的时候经常在特定的内存地址访问，这就需要各种类型数据按照一定的规则在空间上排列，而不是顺序的一个接一个的排放，这就是对齐。

****为什么要字节对齐？****

需要字节对齐的根本原因在于CPU访问数据的效率问题。假设上面整型变量的地址不是自然对齐，比如为0x00000002，则CPU如果取它的值的话需要访问两次内存，第一次取从0x00000002-0x00000003的一个short，第二次取从0x00000004-0x00000005的一个short然后组合得到所要的数据，如果变量在0x00000003地址上的话则要访问三次内存，第一次为char，第二次为short，第三次为char，然后组合得到整型数据。

而如果变量在自然对齐位置上，则只要一次就可以取出数据。一些系统对对齐要求非常严格，比如sparc系统，如果取未对齐的数据会发生错误，而在x86上就不会出现错误，只是效率下降。

各个硬件平台对存储空间的处理上有很大的不同。一些平台对某些特定类型的数据只能从某些特定地址开始存取。比如有些平台每次读都是从偶地址开始，如果一个int型（假设为32位系统）如果存放在偶地址开始的地方，那么一个读周期就可以读出这32bit，而如果存放在奇地址开始的地方，就需要2个读周期，并对两次读出的结果的高低字节进行拼凑才能得到该32bit数据。显然在读取效率上下降很多。

****字节对齐实例****

union example {   int a[5];   char b;   double c;   };   int result = sizeof(example);   /\* 如果以最长20字节为准，内部double占8字节，这段内存的地址0x00000020并不是double的整数倍，只有当最小为0x00000024时可以满足整除double（8Byte）同时又可以容纳int a[5]的大小，所以正确的结果应该是result=24 \*/ struct example {   int a[5];   char b;   double c;   }test\_struct; int result = sizeof(test\_struct);   /\* 如果我们不考虑字节对齐，那么内存地址0x0021不是double（8Byte）的整数倍，所以需要字节对齐，那么此时满足是double（8Byte）的整数倍的最小整数是0x0024，说明此时char b对齐int扩充了三个字节。所以最后的结果是result=32 \*/ struct example {   char b;   double c;   int a;   }test\_struct;   int result = sizeof(test\_struct);   /\* 字节对齐除了内存起始地址要是数据类型的整数倍以外，还要满足一个条件，那就是占用的内存空间大小需要是结构体中占用最大内存空间的类型的整数倍，所以20不是double（8Byte）的整数倍，我们还要扩充四个字节，最后的结果是result=24 \*/