C++

- 10、浅拷贝与深拷贝
  - 答：浅拷贝：比如拷贝类对象时，对象含有指针成员，只是拷贝指针变量自身，这样新旧对象的指针还是指向同一内存区域；深拷贝：同理，对象含有指针成员，拷贝时不仅拷贝指针变量，还重新在内存中为新对象开辟一块内存区域，将原对象次指针成员所指向的内存数据都拷贝到新开辟的内存区域。
- 静态局部变量VS全局变量
  - 作用域
    - 静态局部变量具有局部作用域。它只被初始化一次，自从第一次初始化直到程序结束都一直存在，他和全局变量的区别在于全局变量对所有的函数都是可见的，而静态局部变量只对定义自己的函数体始终可见。
  - 内存分配
    - 全局变量、静态局部变量、静态全局变量都在静态存储区分配空间，而局部变量在栈分配空间。
    - 全局变量本身就是静态存储方式，静态全局变量当然也是静态存储方式。这两者在存储方式上没有什么不同。区别在于非静态全局变量的作用域是整个源程序，当一个源程序由多个源文件组成时，非静态的全局变量在各个源文件中都是有效的。而静态全局变量则限制了其作用域，即只在定义该变量的源文件内有效，在同一源程序的其他源文件中不能使用它。由于静态全局变量的作用域局限于一个源文件内，只能为该源文件内的函数公用，因此可以避免在其他源文件中引起错误。
    - 1、静态变量会被放在程序的静态数据存储区里，这样可以在下一次调用的时候还可以保持原来的赋值。这一点是他与堆栈变量和堆变量的区别
    - 2、变量用static告知编译器，自己仅仅在变量的作用域范围内可见。这一点是他与全局变量的区别。
    - 从以上分析可以看出，把局部变量改变为静态变量后是改变了他的存储方式，即改变了他的生存期。把全局变量改变为静态变量后是改变了他的作用域，限制了他的使用范围，因此static这个说明符在不同的地方起的作用是不同的。
    - 不同类型的变量在内存中的位置：
    - 1、 已经初始化的全局变量存放与data数据段；未初始化的全局变量存放与bss数据段。
    - 2、 静态的全局变量存放与data数据段
    - 3、 局部变量存放在栈上。
    - 4、 静态局部变量，并不是在调用函数时分配函数返回时释放，而是像全局变量一样静态分配，存放data数据段，但它的作用域在函数中起作用。
- 浅拷贝和深拷贝的区别?
  - 编译系统在我们没有自己定义拷贝构造函数时，会在拷贝对象时调用默认拷贝构造函数，进行的是浅拷贝！即对指针name拷贝后会出现两个指针指向同一个内存空间
  - 浅拷贝只是对指针的拷贝，拷贝后两个指针指向同一个内存空间，深拷贝不但对指针进行拷贝，而且对指针指向的内容进行拷贝，经深拷贝后的指针是指向两个不同地址的指针。
- 复制构造函数和赋值操作符

- 复制构造函数是去完成对未初始化的存储区的初始化，而赋值操作符则是处理一个已经存在的对象。对一个对象赋值，当它一次出现时，它将调用复制构造函数，以后每次出现，都调用赋值操作符。

- 智能指针的设计和实现
  - 下面是一个简单智能指针的demo。智能指针类将一个计数器与类指向的对象相关联，引用计数跟踪该类有多少个对象共享同一指针。每次创建类的新对象时，初始化指针并将引用计数置为1；当对象作为另一对象的副本而创建时，拷贝构造函数拷贝指针并增加与之相应的引用计数；对一个对象进行赋值时，赋值操作符减少左操作数所指对象的引用计数（如果引用计数为减至0，则删除对象），并增加右操作数所指对象的引用计数；调用析构函数时，构造函数减少引用计数（如果引用计数减至0，则删除基础对象）。智能指针就是模拟指针动作的类。所有的智能指针都会重载 -> 和 * 操作符。智能指针还有许多其他功能，比较有用的是自动销毁。这主要是利用栈对象的有限作用域以及临时对象（有限作用域实现）析构函数释放内存。
  - shared_ptr
    - 多个指针指向相同的对象。shared_ptr使用引用计数，每一个shared_ptr的拷贝都指向相同的内存。每使用他一次，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，自动删除所指向的堆内存。shared_ptr内部的引用计数是线程安全的，但是对象的读取需要加锁。
    - 初始化。智能指针是个模板类，可以指定类型，传入指针通过构造函数初始化。也可以使用make_shared函数初始化。不能将指针直接赋值给一个智能指针，一个是类，一个是指针。例如std::shared_ptr<int> p4 = new int(1);的写法是错误的
    - 拷贝和赋值。拷贝使得对象的引用计数增加1，赋值使得原对象引用计数减1，当计数为0时，自动释放内存。后来指向的对象引用计数加1，指向后来的对象。
    - get函数获取原始指针
    - 注意不要用一个原始指针初始化多个shared_ptr，否则会造成二次释放同一内存
    - 注意避免循环引用，shared_ptr的一个最大的陷阱是循环引用，循环，循环引用会导致堆内存无法正确释放，导致内存泄漏。循环引用在weak_ptr中介绍。
  - unique_ptr的使用
    - unique_ptr“唯一”拥有其所指对象，同一时刻只能有一个unique_ptr指向给定对象（通过禁止拷贝语义、只有移动语义来实现）。相比与原始指针unique_ptr用于其RAII的特性，使得在出现异常的情况下，动态资源能得到释放。unique_ptr指针本身的生命周期：从unique_ptr指针创建时开始，直到离开作用域。离开作用域时，若其指向对象，则将其所指对象销毁(默认使用delete操作符，用户可指定其他操作)。unique_ptr指针与其所指对象的关系：在智能指针生命周期内，可以改变智能指针所指对象，如创建智能指针时通过构造函数指定、通过reset方法重新指定、通过release方法释放所有权、通过移动语义转移所有权。
  - weak_ptr的使用
    - weak_ptr是为了配合shared_ptr而引入的一种智能指针，因为它不具有普通指针的行为，没有重载operator*和->,它的最大作用在于协助shared_ptr工作，像旁观者那样观测资源的使用情况。weak_ptr可以从一个shared_ptr或者另一个weak_ptr对象构造，获得资源的观测权。但weak_ptr没有共享资源，它的构造不会引起指针引用计数的增加。使用weak_ptr的成员函数use_count()可以观测资源的引用计数，另一个成员函数expired()的功能等价于use_count()==0,但更快，表示被观测的资源(也就是shared_ptr的管理的资源)已经不复存在。weak_ptr可以使用一个非常重要的成员函数lock()从被观测的shared_ptr获得一个可用的shared_ptr对象， 从而操作资源。但当expired()==true的时候，lock()函数将返回一个存储空指针的shared_ptr。

- 请写一个C函数，若处理器是Big_endian的，则返回0；若是Little_endian的，则返回1

```
int checkCPU()
{
 {
 union w
 {
 int a;
 char b;
 } c;
 c.a = 1;
 return (c.b == 1);
 }
}
```

  - union 联合体是共用内存区域，也就是说int 和 char一起公用4byte.并且union一定是从低地址开始存放，所以char b对应最低内存区域。如果是大端存储，int的1存在最高位，其他全为0，小端存储时1存在最低位，所以只要判断b是否为0即可

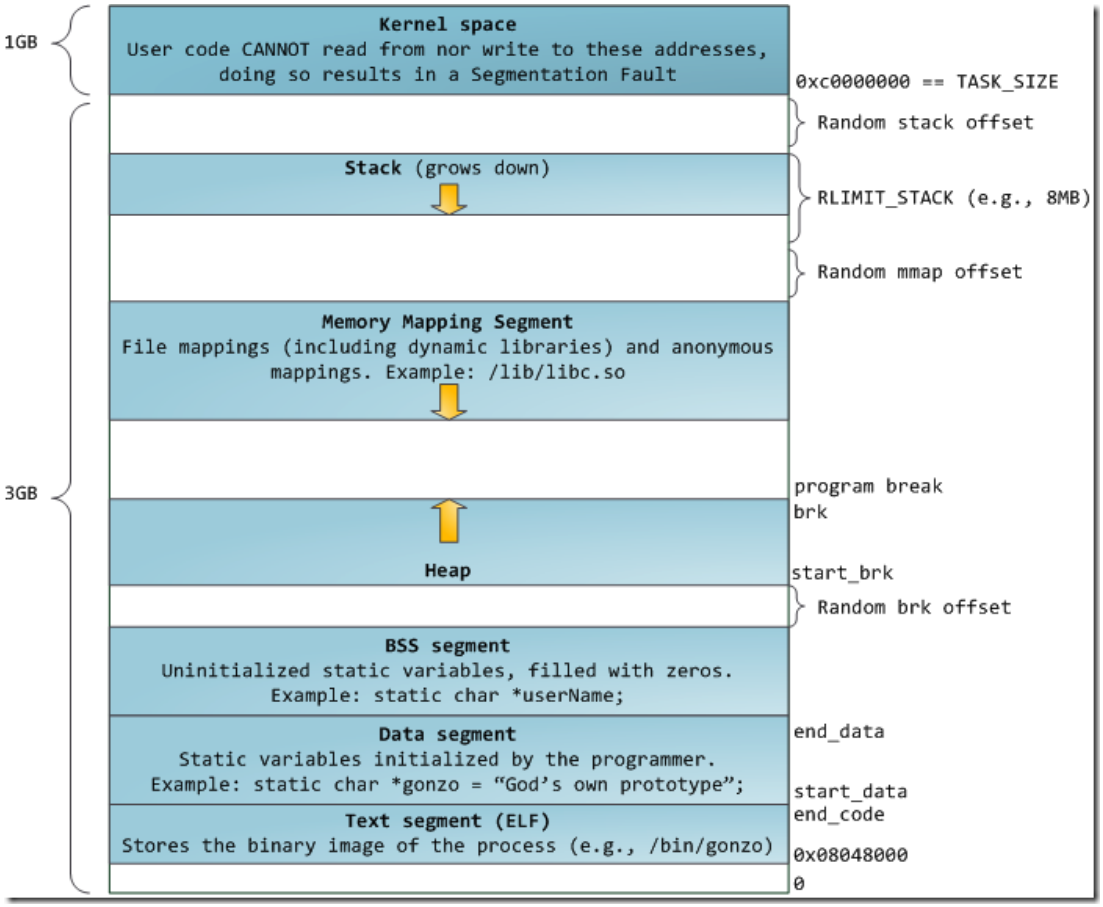
- static关键字至少有下列n个作用:
  - 函数体内static变量的作用范围为该函数体，不同于auto变量，该变量的内存只被分配一次，因此其值在下次调用时仍维 持上次的值;
  - 在模块内的static全局变量可以被模块内所用函数访问，但不能被模块外其它函数访问;
  - 在模块内的static函数只可被这一模块内的其它函数调用，这个函数的使用范围被限制在声明它的模块内;
  - 在类中的static成员变量属于整个类所拥有，对类的所有对象只有一份拷贝;
  - 在类中的static成员函数属于整个类所拥有，这个函数不接收this指针，因而只能访问类的static成员变量。
- const关键字至少有下列n个作用:
  - 欲阻止一个变量被改变，可以使用const关键字。在定义该const变量时，通常需要对它进行初始化，因为以后就没有机 会再去改变它了;
  - 对指针来说，可以指定指针本身为const，也可以指定指针所指的数据为const，或二者同时指定为const;
  - 在一个函数声明中，const可以修饰形参，表明它是一个输入参数，在函数内部不能改变其值;
  - 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;
  - 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”。例如: const classA operator*(const classA& a1,const classA& a2); operator*的返回结果必须是一个const对象。如果不是，这样的变态代码也不会编译出错:
- classA a, b, c; (a * b) = c; // 对a*b的结果赋值 操作(a * b) = c显然不符合编程者的初衷，也没有任何意义。
- 虚函数
  - 每一个类都有虚表
  - 虚表可以继承，如果子子类没有重写虚函数，那么子类虚表中仍然会有该函数的地址，只不过这个地址指向的是基类的虚函数实现，如果基类有3个虚函数，那么基类的虚表中就有三项(虚函数地址)，派生类也会虚表，至少有三项，如果重写了了相应的虚函数，那么虚表中的地址就会改变，指向自身的虚函数实现，如果派生类有自己的虚函数，那么虚表中就会添加该项。
  - 这就是c++中的多态性，当c++编译器在编译的时候，发现Father类的Say()函数是虚函数，这个时候c++就会采用晚绑定技术，也就是编译时并不确定具体调用的函数，

而是在运行时，依据对象的类型来确认调用的是哪一个函数，这种能力就叫做c++的多态性，我们没有在Say()函数前加virtual关键字时，c++编译器就确定了哪个函数被调用，这叫做早期绑定。

- c++的多态性用一句话概括就是:在基类的函数前加上virtual关键字，在派生类中重写该函数，运行时将会根据对象的实际类型来调用用相应的函数，如果对象类型是派生生类，就调用派生类的函数，如果对象类型是基类，就调用用基类的函数。
- 派生类的虚表中虚地址的排列顺序和基类的虚表中虚函数地址排列顺序相同。

- malloc和new有什么区别
  - malloc与free是C++/C语言的标准库函数，new/delete是C++的运算符。它们都可用于申请动态内存和释放内存。
  - 对于非内部数据类型的对象而言，光用malloc/free无无法满足足动态对象的要求。对象在创建的同时要自动执行构造函数，对象在消亡之前要自动执行析构函数。由于malloc/free是库函数而而不不是运算符，不不在编译器器控制权限之内，不不能够把执行行行构造函数和析构函数的任务强加于malloc/free。
  - 因此C++语言言需要一个能完成动态内存分配和初始化工作的运算符new，以一个能完成清理与释放内存工作的运算符delete。 注意new/delete不不是库函数。 C++程序经常要调用C函数，而C程序只能用malloc/free管理动态内存。
  - new可以认为是malloc加构造函数的执行。new出来的指针是直接带类型信息的。而malloc返回的都是void指针。
- 请简述智能指针原理，并实现一个简单的智能指针。
  - 智能指针是一种资源管理类，通过对原始指针进行封装，在资源管理对象进行析构时对指针指向的内存进行释放；通常使用引用计数方式进行管理。
- 引用和指针有什么区别
  - 本质：引用是别名，指针是地址，具体的：
  - 指针可以在运行时改变其所指向的值，引用一旦和某个对象绑定就不再改变
  - 从内存上看，指针会分配内存区域，而引用不会，它仅仅是一个别名
  - 在参数传递时，引用用会做类型检查，而指针不会
  - 引用不能为空，指针可以为空
- const和define有什么区别
  - 本质：define只是字符串替换，const参与编译运行，具体的：
  - define不会做类型检查，const拥有类型，会执行相应的类型检查
  - define仅仅是宏替换，不占用内存，而const会占用内存
  - const内存效率更高，编译器通常将const变量保存在符号表中，而不会分配存储空间，这使得它成 为一个编译期间的常量，没有存储和读取的操作
- C++中包含哪几种强制类型转换？他们有什么区别和联系？
  - reinterpret_cast:
    - 转换一个指针为其它类型的指针。它也允许从一个指针转换为整数类型,反之亦 然. 这个操作符能够在非相关的类型之间转换. 操作结果只是简单的从一个指针到别的指针的值的 二进制拷贝. 在类型之间指向的内容不做任何类型的检查和转换?
      - class A{};
      - class B{};
      - A* a = new A;
      - B* b = reinterpret_cast(a);
  - static_cast:

        - 允许执行任意的隐式转换和相反转换动作（即使它是不允许隐式的）,例如：应用到类 的指针上, 意思是说它允许子类类型的指针转换为父类类型的指针(这是一个有效的隐式转换), 同 时, 也能够执行相反动作: 转换父类为它的子类
            - class Base {};
            - class Derive:public Base{};
            - Base* a = new Base;
            - Derive *b = static_cast(a);
    - dynamic_cast:
        - 只用于对象的指针和引用. 当用于多态类型时，它允许任意的隐式类型转换以及相 反过程. 不过，与static_cast不同，在后一种情况里（注：即隐式转换的相反过程）,dynamic_cast 会检查操作是否有效. 也就是说, 它会检查转换是否会返回一个被请求的有效的完整对象。检测在 运行时进行. 如果被转换的指针不是一个被请求的有效完整的对象指针，返回值为NULL. 对于引用 类型，会抛出bad_cast异常
    - const_cast:
        - 这个转换类型操纵传递对象的const属性，或者是设置或者是移除,例如：
            - class C{};
            - const C* a = new C; C *b = const_cast(a);
- 简述C++中虚继承的作用及底层实现原理?
    - 虚继承用于解决多继承条件下的菱形继承问题，底层实现原理与编译器相关，一般通过虚基类 指针实现，即各对象中只保存一份父类的对象，多继承时通过虚基类指针引用该公共对象，从而避 免菱形继承中的二义性问题。
- 同样可以实现互斥，互斥锁和信号量有什么区别?
    - 信号量是一种同步机制，可以当作锁来用，但也可以当做进程/线程之间通信使用，作为通信使用时不一定有锁的概念；互斥锁是为了锁住一些资源，是为了对临界区做保护
- 简述Linux进程内存空间分为哪几个段？作用分别是什么?
-



- 1.Text：存放可执行的指令操作，其只读不能写。
- 2.Bss：存放未初始化的全局变量和静态变量。
- 3.Data：存放初始化的全局变量和静态变量。
- 4.Stack：存放临时变量，函数参数等。
- 5.Heap：存放New/Malloc等动态申请的变量，用户必须手动进行Delete/Free操作。
- 其中Stack和Heap的内存增长方向是相反的。

- 简述Malloc实现原理
  - 可以基于伙伴系统实现，也可以使用基于链表的实现
  - 将所有空闲内存块连成链表，每个节点记录空闲内存块的地址、大小等信息
  - 分配内存时，找到大小合适的块，切成两份，一分给用户，一份放回空闲链表
  - free时，直接把内存块返回链表
  - 解决外部碎片：将能够合并的内存块进行合并
- 使用mmap读写文件为什么比普通读写函数要快?
  - mmap函数：可以将文件映射到内存中的一段区域，普通函数读写文件：用户空间buffer内核空间buffer磁盘；mmap映射之后：用户空间buffer进程内存空间，省掉了拷贝到内核空间的时间?
- 设计并实现一个LRU Cache
  - 解题思路：题目让设计一个LRU　Cache，即根据LRU算法设计一个缓存。在这之前需要弄清楚LRU算法的核心思想，LRU全称是Least
  - Recently Used，即最近最久未使用的意思。在操作系统的内存管理中，有一类很重要的算法就是内存页面置换算法（包括FIFO，LRU,LFU等几种常见页面置换算法）。事实上，Cache算法和内存页面置换算法的核心思想是一样的：都是在给定一个限定大小的空间的前提下，设计一个原则如何来更新和访问其中的元素。下面说一下LRU算法的核心思想，LRU算法的设计原则是：　如果一个数据在最近一段时间没有被访问

到，那么在将来它被访问的可能性也很小 。也就是说，当限定的空间已存满数据时，应当把最久没有被访问到的数据淘汰。

- 而用什么数据结构来实现LRU算法呢？可能大多数人都会想到：用一个数组来存储数据，给每一个数据项标记一个访问时间戳，每次插入新数据项的时候，先把数组中存在的数据项的时间戳自增，并将新数据项的时间戳置为0并插入到数组中。每次访问数组中的数据项的时候，将被访问的数据项的时间戳置为0。当数组空间已满时，将时间戳最大的数据项淘汰。
- 这种实现思路很简单，但是有什么缺陷呢？需要不停地维护数据项的访问时间戳，另外，在插入数据、删除数据以及访问数据时，时间复杂度都是O(n)。
- 那么有没有更好的实现办法呢？
- 那就是利用链表和hashmap。当需要插入新的数据项的时候，如果新数据项在链表中存在（一般称为命中），则把该节点移到链表头部，如果不存在，则新建一个节点，放到链表头部，若缓存满了，则把链表最后一个节点删除即可。在访问数据的时候，如果数据项在链表中存在，则把该节点移到链表头部，否则返回-1。这样一来在链表尾部的节点就是最近最久未访问的数据项。
- 总结一下：根据题目的要求，LRU Cache具备的操作：
- 1）set(key,value)：如果key在hashmap中存在，则先重置对应的value值，然后获取对应的节点cur，将cur节点从链表删除，并移动到链表的头部；若果key在hashmap不存在，则新建一个节点，并将节点放到链表的头部。当Cache存满的时候，将链表最后一个节点删除即可。
- 2）get(key)：如果key在hashmap中存在，则把对应的节点放到链表头部，并返回对应的value值；如果不存在，则返回-1。

- 垃圾回收机制【JAVA】
  - Mark - Sweep（标记 - 清除）算法
  - Mark - Compact（标记 - 整理）算法
  - Copying（复制）算法
  - Generational（分代）算法
- volatile是干啥的
  - 访问寄存器要比访问内存要块，因此CPU会优先访问该数据在寄存器中的存储结果，但是内存中的数据可能已经发生了改变，而寄存器中还保留着原来的结果。为了避免这种情况的发生将该变量声明为volatile，告诉CPU每次都从内存去读取数据。
  - 一个参数可以即是const又是volatile的吗？可以，一个例子是只读状态寄存器，是volatile是因为它可能被意想不到的被改变，是const告诉程序不应该试图去修改他。
- new与malloc区别
  - new分配内存按照数据类型进行分配，malloc分配内存按照大小分配；
  - new不仅分配一段内存，而且会调用构造函数，但是malloc则不会。new的实现原理？但是还需要注意的是，之前看到过一个题说int* p = new int与int* p = new int()的区别，因为int属于C++内置对象，不会默认初始化，必须显示调用默认构造函数，但是对于自定义对象都会默认调用构造函数初始化。翻阅资料后，在C++11中两者没有区别了，自己测试的结构也都是为0；
  - new返回的是指定对象的指针，而malloc返回的是void*，因此malloc的返回值一般都需要进行类型转化；
  - new是一个操作符可以重载，malloc是一个库函数；
  - new分配的内存要用delete销毁，malloc要用free来销毁；delete销毁的时候会调用对象的析构函数，而free则不会；
  - malloc分配的内存不够的时候，可以用realloc扩容。扩容的原理？new没用这样操作；

    - new如果分配失败了会抛出bad_malloc的异常，而malloc失败了会返回NULL。因此对于new，正确的姿势是采用try...catch语法，而malloc则应该判断指针的返回值。为了兼容很多c程序员的习惯，C++也可以采用new nothrow的方法禁止抛出异常而返回NULL；
    - new和new[]的区别，new[]一次分配所有内存，多次调用构造函数，分别搭配使用delete和delete[]，同理，delete[]多次调用析构函数，销毁数组中的每个对象。而malloc则只能sizeof(int) * n；
    - 如果不够可以继续谈new和malloc的实现，空闲链表，分配方法(首次适配原则，最佳适配原则，最差适配原则，快速适配原则)。delete和free的实现原理，free为什么直到销毁多大的空间?
- 析构函数能抛出异常吗
    - 如果析构函数抛出异常，则异常点之后的程序不会执行，如果析构函数在异常点之后执行了某些必要的动作比如释放某些资源，则这些动作不会执行，会造成诸如资源泄漏的问题。
    - 通常异常发生时，c++的机制会调用已经构造对象的析构函数来释放资源，此时若析构函数本身也抛出异常，则前一个异常尚未处理，又有新的异常，会造成程序崩溃的问题。
- 智能指针是怎么实现的？什么时候改变引用计数?
    - 构造函数中计数初始化为1；
    - 拷贝构造函数中计数值加1；
    - 赋值运算符中，左边的对象引用计数减一，右边的对象引用计数加一；
    - 析构函数中引用计数减一；
    - 在赋值运算符和析构函数中，如果减一后为0，则调用delete释放对象。
    - share_prt与weak_ptr的区别?
        - share_ptr可能出现循环引用，从而导致内存泄露
        - weak_ptr是一种弱引用指针，其存在不会影响引用计数，从而解决循环引用的问题
- static_cast, dynamic_cast, const_cast, reinterpret_cast
    - const_cast用于将const变量转为非const
    - static_cast用的最多，对于各种隐式转换，非const转const，void*转指针等, static_cast能用于多态想上转化，如果向下转能成功但是不安全，结果未知；
    - dynamic_cast用于动态类型转换。只能用于含有虚函数的类，用于类层次间的向上和向下转化。只能转指针或引用。向下转化时，如果是非法的对于指针返回NULL，对于引用抛异常。要深入了解内部转换的原理。
    - reinterpret_cast几乎什么都可以转，比如将int转指针，可能会出问题，尽量少用；
    - 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。
- 内存对齐的原则
    - 从0位置开始存储；
    - 变量存储的起始位置是该变量大小的整数倍；
    - 结构体总的大小是其最大元素的整数倍，不足的后面要补齐；
    - 结构体中包含结构体，从结构体中最大元素的整数倍开始存；
    - 如果加入pragma pack(n)，取n和变量自身大小较小的一个。
- 内联函数有什么优点？内联函数与宏定义的区别?
    - 宏定义在预编译的时候就会进行宏替换；

    - 内联函数在编译阶段，在调用内联函数的地方进行替换，减少了函数的调用过程，但是使得编译文件变大。因此，内联函数适合简单函数，对于复杂函数，即使定义了内联编译器可能也不会按照内联的方式进行编译。
    - 内联函数相比宏定义更安全，内联函数可以检查参数，而宏定义只是简单的文本替换。因此推荐使用内联函数，而不是宏定义。
    - 使用宏定义函数要特别注意给所有单元都加上括号，#define MUL(a, b) a * b，这很危险，正确写法：#define MUL(a, b) ((a) * (b))
- C++内存管理
    - C++内存分为那几块？（堆区，栈区，常量区，静态和全局区）
    - 每块存储哪些变量？
    - 学会迁移，可以说到malloc，从malloc说到操作系统的内存管理，说道内核态和用户态，然后就什么高端内存，slab层，伙伴算法，VMA可以巴拉巴拉了，接着可以迁移到fork()。
- STL里的内存池实现
    - STL内存分配分为一级分配器和二级分配器，一级分配器就是采用malloc分配内存，二级分配器采用内存池。
        - 二级分配器设计的非常巧妙，分别给8k，16k,..., 128k等比较小的内存片都维持一个空闲链表，每个链表的头节点由一个数组来维护。需要分配内存时从合适大小的链表中取一块下来。假设需要分配一块10K的内存，那么就找到最小的大于等于10k的块，也就是16K，从16K的空闲链表里取出一个用于分配。释放该块内存时，将内存节点归还给链表。
        - 如果要分配的内存大于128K则直接调用一级分配器。
        - 为了节省维持链表的开销，采用了一个union结构体，分配器使用union里的next指针来指向下一个节点，而用户则使用union的空指针来表示该节点的地址。
- STL里set和map是基于什么实现的。红黑树的特点？
    - set和map都是基于红黑树实现的。
    - 红黑树是一种平衡二叉查找树，与AVL树的区别是什么？AVL树是完全平衡的，红黑树基本上是平衡的。
    - 为什么选用红黑数呢？因为红黑数是平衡二叉树，其插入和删除的效率都是N(logN)，与AVL相比红黑数插入和删除最多只需要3次旋转，而AVL树为了维持其完全平衡性，在坏的情况下要旋转的次数太多。
    - 红黑树的定义：
    - (1) 节点是红色或者黑色；
    - (2) 父节点是红色的话，子节点就不能为红色；
    - (3) 从根节点到每个页子节点路径上黑色节点的数量相同；
    - (4) 根是黑色的，NULL节点被认为是黑色的。
- 必须在构造函数初始化式里进行初始化的数据成员有哪些
    - (1) 常量成员，因为常量只能初始化不能赋值，所以必须放在初始化列表里面
    - (2) 引用类型，引用必须在定义的时候初始化，并且不能重新赋值，所以也要写在初始化列表里面
    - (3) 没有默认构造函数的类类型，因为使用初始化列表可以不必调用默认构造函数来初始化，而是直接调用拷贝构造函数初始化
- 模板特化
    - 模板特化分为全特化和偏特化，模板特化的目的就是对于某一种变量类型具有不同的实现，因此需要特化版本。例如，在STL里迭代器为了适应原生指针就将原生指针进行

特化。

- 定位内存泄露
  - (1)在windows平台下通过CRT中的库函数进行检测；
  - (2)在可能泄漏的调用前后生成块的快照，比较前后的状态，定位泄漏的位置
  - (3)Linux下通过工具valgrind检测
- strcpy
  -

```
char* strcpy(char* dst, const char* src)
{
    assert((dst != NULL) && (src != NULL));
    char* ret = dst;
    int size = strlen(src) + 1;
    if(dst > src || dst < src + len)
    {
        dst = dst + size - 1;
        src = src + size - 1;
        while(size--)
        {
            *dst-- = *src--;
        }
    }
    else
    {
        while(size--)
        {
            *dst++ = *src++;
        }
    }
    return ret;
}
```

- memcpy
  -

```
void* memcpy(void* dst, const void* src, size_t size)
{
    if(dst == NULL || src == NULL)
    {
        return NULL;
    }
    void* res = dst;
    char* pdst = (char*)dst;
    char* psrc = (char*)src;

    if(pdst > psrc && pdst < psrc + size) //重叠
    {
        pdst = pdst + size - 1;
        psrc = pdst + size - 1;
        while(size--)
        {
            *pdst-- = *psrc--;
        }
    }
    else //无重叠
    {
        while(size--)
        {
            *dst++ = *src++;
        }
    }
    return ret;
}
```

- strcat
  -
```
char* strcat(char* dst, const char* src)
{
    char* ret = dst;

    while(*dst != '\0')
        ++dst;

    while((*dst++ = *src) != '\0');
    return ret;
}
```
- strcmp
  -
```
int strcmp(const char* str1, const char* str2)
{

    while(*str1 == *str2 && *str1 != '\0')
    {
        ++str1;
        ++str2;
    }
    return *str1 - *str2;
}
```
-