C++高频面试题

- 指针和引用的区别
  - 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
- 堆和栈的区别
  - 1、管理方式不同；
    - 对于栈来讲，是由编译器自动管理，无需我们手工控制；对于堆来说，释放工作由程序员控制，容易产生memory leak。
  - 2、空间大小不同；
    - 一般来讲在32位系统下，堆内存可以达到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，一般都是有一定的空间大小的，例如，在VC6下面，默认的栈空间大小是1M（好像是，记不清楚了）。
  - 3、能否产生碎片不同；
    - 对于堆来讲，频繁的new/delete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，因为栈是先进后出的队列，他们是如此的一一对应，以至于永远都不可能有一个内存块从栈中间弹出，在他弹出之前，在他上面的后进的栈内容已经被弹出，详细的可以参考数据结构，这里我们就不再一一讨论了
  - 4、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，它的生长方向是向下的，是向着内存地址减小的方向增长。
  - 5、分配方式不同；
    - 堆都是动态分配的，没有静态分配的堆。栈有2种分配方式：静态分配和动态分配。静态分配是编译器完成的，比如局部变量的分配。动态分配由alloca函数进行分配，但是栈的动态分配和堆是不同的，他的动态分配是由编译器进行释放，无需我们手工实现。
  - 6、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C/C++函数库提供的，它的机制是很复杂的，例如为了分配一块内存，库函数会按照一定的算法（具体的算法可以参考数据结构/操作系统）在堆内存中搜索可用的足够大小的空间，如果没有足够大小的空间（可能是由于内存碎片太多），就有可能调用系统功能去增加程序数据段的内存空间，这样就有机会分到足够大小的内存，然后进行返回。显然，堆的效率比栈要低得多。
- new和delete是如何实现的，new 与 malloc的异同处
  - 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。

- C和C++的区别
  - 第一点就应该想到C是面向过程的语言，而C++是面向对象的语言，一般简历上第一条都是熟悉C/C++基本语法，了解C++面向对象思想，那么，请问什么是面向对象?
  - C和C++动态管理内存的方法不一样，C是使用malloc/free函数，而C++除此之外还有new/delete关键字；（关于malooc/free与new/delete的不同又可以说一大堆，最后的扩展_1部分列出十大区别）；
  - 接下来就不得不谈到C中的struct和C++的类，C++的类是C所没有的，但是C中的struct是可以在C++中正常使用的，并且C++对struct进行了进一步的扩展，使struct在C++中可以和class一样当做类使用，而唯一和class不同的地方在于struct的成员默认访问修饰符是public,而class默认的是private;
  - C++支持函数重载，而C不支持函数重载，而C++支持重载的依仗就在于C++的名字修饰与C不同，例如在C++中函数int fun(int ,int)经过名字修饰之后变为 _fun_int_int ,而C是
  - _fun，一般是这样的，所以C++才会支持不同的参数调用不同的函数;
  - C++中有引用，而C没有；这样就不得不提一下引用和指针的区别（文后扩展_2）；
  - 当然还有C++全部变量的默认链接属性是外链接，而C是内连接;
  - C 中用const修饰的变量不可以用在定义数组时的大小，但是C++用const修饰的变量可以（如果不进行&,解引用的操作的话，是存放在符号表的，不开辟内存）；
  - 当然还有局部变量的声明规则不同，多态，C++特有输入输出流之类的，很多，下面就不再列出来了;
- C++、Java的联系与区别，包括语言特性、垃圾回收、应用场景等（java的垃圾回收机制）
- Struct和class的区别
  - 最本质的一个区别就是默认的访问控制
  - “class”这个关键字还用于定义模板参数，就像“typename”。但关键字“struct”不用于定义模板参数。这一点在Stanley B.Lippman写的Inside the C++ Object Model有过说明。
- define 和const的区别（编译阶段、安全性、内存占用等）
  - 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
- 在C++中const和static的用法（定义，用途）
  - 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:

-
- const和static在类中使用的注意事项（定义、初始化和使用）
- C++中的const类成员函数（用法和意义）
- C++的STL介绍（这个系列也很重要，建议侯捷老师的这方面的书籍与视频），其中包括内存管理allocator，函数，实现机理，多线程实现等
  - 内存管理allocator
    - STL中的内存分配器实际上是基于空闲列表(free list)的分配策略，最主要的特点是通过组织16个空闲列表，对小对象的分配做了优化。
    - 1）小对象的快速分配和释放。当一次性预先分配好一块固定大小的内存池后，对小于128字节的小块内存分配和释放的操作只是一些基本的指针操作，相比于直接调用malloc/free，开销小。
    - 2）避免内存碎片的产生。零乱的内存碎片不仅会浪费内存空间，而且会给OS的内存管理造成压力。
    - 3）尽可能最大化内存的利用率。当内存池尚有的空闲区域不足以分配所需的大小时，分配算法会将其链入到对应的空闲列表中，然后会尝试从空闲列表中寻找是否有合适大小的区域
      - allocator是一个由两级分配器构成的内存管理器，当申请的内存大小大于128byte时，就启动第一级分配器通过malloc直接向系统的堆空间分配，如果申请的内存大小小于128byte时，就启动第二级分配器，从一个预先分配好的内存池中取一块内存交付给用户，这个内存池由16个不同大小（8的倍数，8~128byte）的空闲列表组成，allocator会根据申请内存的大小（将这个大小round up成8的倍数）从对应的空闲块列表取表头块给用户。
- STL源码中的hash表的实现
  - 哈希表，又名散列表，是根据关键字直接访问内存的数据结构。通过哈希函数，将键值映射转换成数组中的位置，就可以在O（1）的时间内访问到数据。举个例子，比如有一个存储家庭信息的哈希表，通过人名查询他们家的信息，哈希函数为f（），数组info[N]用于存储，那么张三家的信息就在info[f（张三）]上。由此，不需比较便可知道张三家里有几口人，人均几亩地，地里有几头牛。很快对不对，不过有时候会出现f（张三）等于f（李四）的情况，这就叫哈希碰撞。碰撞是由哈希函数造成的，良好的哈希函数只能减少哈希碰撞的概率，而不能完全避免。这就需要处理冲突的方法，一般有两种：
  - 1. 开放定址法
    - 先存储了张三的信息，等到存李四的信息时发现，这位置有记录了，怎么办，假如李四这人不爱跟张三一块凑热闹，就重新找了个位置。这个方法就多了，他可以放在后面一个位置，如果这位置还有的话，就再放后面一个位置，以此类推，这就叫线性探测；他可能嫌一个个位置找太慢了，于是就按照12，22，32的间隔找，这就叫平方探测；或者再调用另外一个哈希函数g（）得到新的位置，这就叫再哈希…
  - 2. 开链法

        - 如果李四这人嫌重新找个坑太麻烦了，愿意和张三放在一起，通过链表连接，这就是开链法。开链法中一个位置可能存放了多个纪录。
        - 一个哈希表中元素的个数与数组的长度的比值称为该哈希表的负载因子。开放定址法的数组空间是固定的，负载因子不会大于1，当负载因子越大时碰撞的概率越大，当负载因子超过0.8时，查询时的缓存命中率会按照指数曲线上升，所以负载因子应该严格控制在0.7-0.8以下，超过时应该扩展数组长度。 开链法的负载因子可以大于1，插入数据的期望时间O（1），查询数据的期望时间是O（1+a）,a是负载因子，a过大时也需要扩展数组长度。
    - stl采用了开链法实现哈希表，其中每个哈希节点有数据和next指针
- STL中unordered_map和map的区别
    - 内部实现机理不同
        - map： map内部实现了一个红黑树（红黑树是非严格平衡二叉搜索树，而AVL是严格平衡二叉搜索树），红黑树具有自动排序的功能，因此map内部的所有元素都是有序的，红黑树的每一个节点都代表着map的一个元素。因此，对于map进行的查找，删除，添加等一系列的操作都相当于是对红黑树进行的操作。map中的元素是按照二叉搜索树（又名二叉查找树、二叉排序树，特点就是左子树上所有节点的键值都小于根节点的键值，右子树所有节点的键值都大于根节点的键值）存储的，使用中序遍历可将键值按照从小到大遍历出来。
        - unordered_map: unordered_map内部实现了一个哈希表（也叫散列表，通过把关键码值映射到Hash表中一个位置来访问记录，查找的时间复杂度可达到O(1)，其在海量数据处理中有着广泛应用）。因此，其元素的排列顺序是无序的。
        - map：
            - 优点：
                - 有序性，这是map结构最大的优点，其元素的有序性在很多应用中都会简化很多的操作
                - 红黑树，内部实现一个红黑书使得map的很多操作在lgn的时间复杂度下就可以实现，因此效率非常的高
            - 缺点： 空间占用率高，因为map内部实现了红黑树，虽然提高了运行效率，但是因为每一个节点都需要额外保存父节点、孩子节点和红/黑性质，使得每一个节点都占用大量的空间
            - 适用处：对于那些有顺序要求的问题，用map会更高效一些
        - unordered_map：
            - 优点： 因为内部实现了哈希表，因此其查找速度非常的快
            - 缺点： 哈希表的建立比较耗费时间
            - 适用处：对于查找问题，unordered_map会更加高效一些，因此遇到查找问题，常会考虑一下用unordered_map
        - 1. 内存占有率的问题就转化成红黑树 VS hash表，还是unorder_map占用的内存要高。
        - 2. 但是unordered_map执行效率要比map高很多
        - 3. 对于unordered_map或unordered_set容器，其遍历顺序与创建该容器时输入的顺序不一定相同，因为遍历是按照哈希表从前往后依次遍历的
- STL中vector的实现
    - vector简单的讲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内存空间，当空间装不下的时候会自动申请一片更大的空间（空间配置器）将原来的数据拷贝到新的空间，然后就会释放旧的空间。当删除的时候空间并不会被释放只是清空了里面的数据。

    - vector的数据安排以及操作方式与array非常相似，两者的唯一区别在于空间运用的灵活性，array是静态空间一旦配置了就不能改变大小，如果要扩大或缩小容量的话，就要把数据搬到新大小的数组里面，然后再把原来的空间释放还给系统。vector是动态空间是随着元素的加入，它的内部机制会自动的扩充空间来容纳新的元素。因此，vector的运用对于内存的合理利用与运用的灵活性有很大的帮助，我们不必因为害怕空间不足而一开始就开辟一块很大的内存。
    - vector的实现技术，关键在于其对大小的控制以及重新配置时的数据移动效率。一旦vector的旧有空间满载，如果客户端每新增一个元素，vector的内部只是扩充一个元素的空间，实为不智。因为所谓的扩充空间（无论多大），过程是配置新空间–数据移动–释还旧空间的成本很高。vector维护的是一个连续线性空间，所以vector支持随机访问。 在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。因此，对vector的任何操作一旦引起了空间的重新配置，指向原vector的所有迭代器就会都失效了这是程序员易犯的一个错误。
- 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。
- C++中的重载和重写的区别：
    - 重载：是指同一可访问区内被声明的几个具有不同参数列（参数的类型，个数，顺序不同）的同名函数，根据参数列表确定调用哪个函数，重载不关心函数返回类型。
    - 隐藏：是指派生类的函数屏蔽了与其同名的基类函数，注意只要同名函数，不管参数列表是否相同，基类函数都会被隐藏。
    - 重写(覆盖)：是指派生类中存在重新定义的函数。其函数名，参数列表，返回值类型，所有都必须同基类中被重写的函数一致。只有函数体不同（花括号内），派生类调用时会调用派生类的重写函数，不会调用被重写函数。重写的基类中被重写的函数必须有virtual修饰。
- C ++内存管理（热门问题）
    - 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。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。
    - 堆，就是那些由new分配的内存块，他们的释放编译器不去管，由我们的应用程序去控制，一般一个new就要对应一个delete。如果程序员没有释放掉，那么在程序结束后，操作系统会自动回收。
    - 自由存储区，就是那些由malloc等分配的内存块，他和堆是十分相似的，不过它是用free来结束自己的生命的。
    - 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，全局变量又分为初始化的和未初始化的，在C++里面没有这个区分了，他们共同占用同一块内存区。
    - 常量存储区，这是一块比较特殊的存储区，他们里面存放的是常量，不允许修改。
- 介绍面向对象的三大特性，并且举例说明每一个。
    - 封装：
        - 封装是指将数据与具体操作的实现代码放在某个对象内部，使这些代码的实现细节不被外界发现，外界只能通过接口使用该对象，而不能通过任何形式修改对象内部实现，正是由于封装机制，程序在使用某一对象时不需要关心该对象的数据结构细节及实现操作的方法。使用封装能隐藏对象实现细节，使代码更易维护，同时因为不能直接调用、修改对象内部的私有信息，在一定程度上保证了系统安全性。

- 继承：
  - 继承来源于现实世界，一个最简单的例子就是孩子会具有父母的一些特征，即每个孩子都会继承父亲或者母亲的某些特征，当然这只是最基本的继承关系，现实世界中还存在着更复杂的继承，面向对象之所以使用继承机制主要是用于实现代码的复用多个类所公用的代码部分可以只在一个类中提供，而其他类只需要继承即可。
- 多态：
  - 多态与继承纤细紧密，是面向对象编程中另一个突出的特征，所谓的多态是指在继承体系中，所有派生类都从基类继承接口，但由于每个派生类都是独立的实体，因此在接收同一消息的时候，可能会生成不同的响应。多态的作用作为隐藏代码实现细节，使得代码能够模块化;扩展代码模块，实现接口重用。简单来说：一种行为产生多种效果。
- 总的来说：封装可以隐藏实现细节同时包含私有成员，使得代码模块化并增加安全指数；基础可以扩展已存在的模块，目的是为了代码重用；多态则是为了保证：类在继承和派生的时候，保证家谱中任何类的实例被正确调用，实现了接口重用。

- 多态的实现（和下个问题一起回答）
- C++虚函数相关（虚函数表，虚函数指针），虚函数的实现原理（热门，重要）
- 实现编译器处理虚函数表应该如何处理
- 析构函数一般写成虚函数的原因
  - 析构函数执行时先调用派生类的析构函数，其次才调用基类的析构函数。如果析构函数不是虚函数，而程序执行时又要通过基类的指针去销毁派生类的动态对象，那么用delete销毁对象时，只调用了基类的析构函数，未调用派生类的析构函数。这样会造成销毁对象不完全。
- 构造函数为什么一般不定义为虚函数
  - 1.虚函数的作用是什么？是实现部分或默认的功能，而且该功能可以被子类所修改。如果父类的构造函数设置成虚函数，那么子类的构造函数会直接覆盖掉父类的构造函数。而父类的构造函数就失去了一些初始化的功能。这与子类的构造需要先完成父类的构造的流程相违背了。而这个后果会相当严重。
  - 2.虚函数的调用是需要通过“虚函数表”来进行的，而虚函数表也需要在对象实例化之后才能够进行调用。在构造对象的过程中，还没有为“虚函数表”分配内存。所以，这个调用也是违背先实例化后调用的准则。
  - 3.虚函数的调用是由父类指针进行完成的，而对象的构造则是由编译器完成的，由于在创建一个对象的过程中，涉及到资源的创建，类型的确定，而这些是无法在运行过程中确定的，需要在编译的过程中就确定下来。而多态是在运行过程中体现出来的，所以是不能够通过虚函数来创建构造函数的，与实例化的次序不同也有关系。
- 构造函数或者析构函数中调用虚函数会怎样
  - 铁律一：最好不要在构造函数和析构函数中调用虚函数！
  - 构造派生类对象时，首先调用基类构造函数初始化对象的基类部分。在执行基类构造函数时，对象的派生类部分是未初始化的。实际上，此时的对象还不是一个派生类对象。
  - 析构派生类对象时，首先撤销/析构他的派生类部分，然后按照与构造顺序的逆序撤销他的基类部分。
  - 因此，在运行构造函数或者析构函数时，对象都是不完整的。为了适应这种不完整，编译器将对象的类型视为在调用构造/析构函数时发生了变换，即：视对象的类型为当前构造函数/析构函数所在的类的类类型。由此造成的结果是：在基类构造函数或者析构函数中，会将派生类对象当做基类类型对象对待。

    - 而这样一个结果，会对构造函数、析构函数调用期间调用的虚函数类型的动态绑定对象产生影响，最终的结果是：如果在构造函数或者析构函数中调用虚函数，运行的都将是为构造函数或者析构函数自身类类型定义的虚函数版本。 无论有构造函数、析构函数直接还是间接调用虚函数。
- 纯虚函数
    - 虚函数与纯虚函数的区别在于：纯虚函数是虚函数的一个子集，用于抽象类，含有纯虚函数的类就是抽象类，它不能生成对象。
- 静态绑定和动态绑定的介绍
    - 对于非虚成员函数， C ++是静态绑定的，而虚函数都是动态绑定，如此才可实现多态性。这也是C ++语言和其它语言Java, Python的一个显著区别。
    - 几个名词定义：
    - - 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，表现一个对象将会有什么行为，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；
- 引用是否能实现动态绑定，为什么引用可以实现
    - 因为对象的类型是确定的，在编译期就确定了
    - 指针或引用是在运行期根据他们绑定的具体对象确定。
- 深拷贝和浅拷贝的区别（举例说明深拷贝的安全性）
    - 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。
- 对象复用的了解，零拷贝的了解
    - 对象池通过对象复用的方式来避免重复创建对象，它会事先创建一定数量的对象放到池中，当用户需要创建对象的时候，直接从对象池中获取即可，用完对象之后再放回到对象池中，以便复用。
    - 适用性：类的实例可重用。类的实例化过程开销较大。类的实例化的频率较高。
    - 零拷贝：emplace_back
- 介绍C++所有的构造函数
    -
- 什么情况下会调用拷贝构造函数（三种情况）
    - （1）用类的一个对象去初始化另一个对象时
    - （2）当函数的形参是类的对象时（也就是值传递时），如果是引用传递则不会调用
    - （3）当函数的返回值是类的对象或引用时
- 结构体内存对齐方式和为什么要进行内存对齐?
    - CPU效率
- 内存泄露的定义，如何检测与避免?
    - 申请的堆没有被释放，该内存不能在被利用。
    - glvinf
- 手写实现智能指针类（34-37我没遇见过）

- 调试程序的方法
  -
- 遇到coredump要怎么调试
- 内存检查工具的了解
- 模板的用法与适用场景
  - Template <class或者也可以用typename T>
  - 1.数据类型与算法相分离的泛型编程
    - 在模板编程中，最常见的就是此类用法，将数据类型与算法相分离，实现泛型编程。
    - STL本身实现了数据容器与算法的分离，而STL中大量的模板应用，则实现了数据类型与容器算法的分离，它是泛型编程的一个典范。
  - 2.类型适配Traits
  - 3.函数转发
    - 模板类的很多应用在于它能针对不同的模板参数生成不同的类。这使得我们可以通过模板类将函数指针以及它的参数类型记录下来，在需要的时候再对函数进行调用。
  - 4.元编程
  -
- 成员初始化列表的概念，为什么用成员初始化列表会快一些（性能优势）？
  - 成员初始化列表和构造函数体的区别
    - 成员初始化列表和构造函数的函数体都可以为我们的类数据成员指定一些初值，但是两者在给成员指定初值的方式上是不同的。成员初始化列表使用初始化的方式来为数据成员指定初值，而构造函数的函数体是通过赋值的方式来给数据成员指定初值。也就是说，成员初始化列表是在数据成员定义的同时赋初值，但是构造函的函数体是采用先定义后赋值的方式来做。这样的区别就造成了，在有些场景下，是必须要使用成员初始化列表的。
  - 功能上面的区别
    - 初始化一个引用成员变量
    - 初始化一个const变量
    - 当我们在初始化一个子类对象的时候，而这个子类对象的父类有一个显示的带有参数的构造函数
    - 当调用一个类类型成员的构造函数，而它拥有一组参数的时候
  - 如果使用构造函数体来对类类型付初值的时候，平白无故多了很多步骤和花销。
  - 一次默认构造函数的调用
  - 一个临时对象temp的创建
  - 一次拷贝赋值运算符函数的调用
  - 一次析构函数的调用
  - 成员初始化列表是可以初始化类的数据成员，那么他是如何操作的呢？是通过一系列的函数调用么，不是的。成员初始化列表是按照数据成员的声明顺序，将初始化操作安排在构造函数所有usercode的前面。成员初始化列表的初始化顺序是按照类成员的声明顺序来的，所以在初始化的时候，尽量不要用次序较后的成员来初始化次序较前的成员，这样就会出问题，这也是成员初始化列表的一个弊端。
- 用过C11吗，知道C11新特性吗？（有面试官建议熟悉C11）
- C++的调用惯例（简单一点C++函数调用的压栈过程）

- C++的四种强制转换
- 类成员函数的压栈

  - 下面对上面的汇编代码中的重点行进行分析：
  - 1、将ecx寄存器中的值压栈，也就是把this指针压栈。
  - 2、ecx寄存器出栈，也就是this指针出栈。
  - 3、将ecx的值放到指定的地方，也就是this指针放到[ebp-8]内。
  - 4、取this指针的值放入eax寄存器内。此时，this指针指向test对象，test对象只有两个int型的成员变量，在test对象内存中连续存放，也就是说this指针目前指向m_iValue1。
  - 5、给寄存器eax指向的地址赋值0Dh（十六进制的13）。其实就是给成员变量m_iValue1赋值13。
  - 6、同4。
  - 7、给寄存器eax指向的地址加4的地址赋值。在4中已经说明，eax寄存器内存放的是this指针，而this指针指向连续存放的int型的成员变量m_iValue1。this指针加4（sizeof(int)）也就是成员变量m_iValue2的地址。因此这一行就是给成员变量m_iValue2赋值。
  - 通过上面的分析，我们可以从底层了解了C++中this指针的实现方法。虽然不同的编译器会使用不同的处理方法，但是C++编译器必须遵守C++标准，因此对于this指针的实现应该都是差不多的。