《逆袭进大厂》第二弹之C++进阶篇59问59答

CPP开发者 今天

以下文章来源于拓跋阿秀 ，作者我是阿秀

拓跋阿秀

双非学历、字节全栈、专业写bug。我踩过的坑不希望你再踩，我走过的路希望你能照...

第一期《逆袭进大厂》之C++篇49问49答

这是第二期。不逼逼了，《逆袭进大厂》系列第二弹 C++ 进阶篇直接发车了。

50、static的用法和作用?

1.先来介绍它的第一条也是最重要的一条：隐藏。（static函数，static变量均可）

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。

2.static的第二个作用是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，说到底static还是用来隐藏的。

3.static的第三个作用是默认初始化为0（static变量）

其实全局变量也具备这一属性，因为全局变量也存储在静态数据区。在静态数据区，内存中所有的字节默认值都是0x00，某些时候这一特点可以减少程序员的工作量。

4.static的第四个作用：C++中的类成员声明static

1） 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；

2） 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；

3） 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；

4） 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；

5） 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。

类内：

6） static类对象必须要在类外进行初始化，static修饰的变量先于对象存在，所以static修饰的变量要在类外初始化；

7） 由于static修饰的类成员属于类，不属于对象，因此static类成员函数是没有this指针的，this指针是指向本对象的指针。正因为没有this指针，所以static类成员函数不能访问非static的类成员，只能访问 static修饰的类成员；

8） static成员函数不能被virtual修饰，static成员不属于任何对象或实例，所以加上virtual没有任何实际意义；静态成员函数没有this指针，虚函数的实现是为每一个对象分配一个vptr指针，而vptr是通过this指针调用的，所以不能为virtual；虚函数的调用关系，this->vptr->ctable->virtual function

51、静态变量什么时候初始化

1） 初始化只有一次，但是可以多次赋值，在主程序之前，编译器已经为其分配好了内存。

2） 静态局部变量和全局变量一样，数据都存放在全局区域，所以在主程序之前，编译器已经为其分配好了内存，但在C和C++中静态局部变量的初始化节点又有点不太一样。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，在程序运行结束，变量所处的全局内存会被全部回收。

3） 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。在程序结束，按照构造顺序反方向进行逐个析构。所以在C++中是可以使用变量对静态局部变量进行初始化的。

52、const关键字?

1） 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；

2） 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；

3） 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；

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”。

6） const成员函数可以访问非const对象的非const数据成员、const数据成员，也可以访问const对象内的所有数据成员；

7） 非const成员函数可以访问非const对象的非const数据成员、const数据成员，但不可以访问const对象的任意数据成员；

8） 一个没有明确声明为const的成员函数被看作是将要修改对象中数据成员的函数，而且编译器不允许它为一个const对象所调用。因此const对象只能调用const成员函数。

9） const类型变量可以通过类型转换符const_cast将const类型转换为非const类型；

10）const类型变量必须定义的时候进行初始化，因此也导致如果类的成员变量有const类型的变量，那么该变量必须在类的初始化列表中进行初始化；

11）对于函数值传递的情况，因为参数传递是通过复制实参创建一个临时变量传递进函数的，函数内只能改变临时变量，但无法改变实参。则这个时候无论加不加const对实参不会产生任何影响。但是在引用或指针传递函数调用中，因为传进去的是一个引用或指针，这样函数内部可以改变引用或指针所指向的变量，这时const 才是实实在在地保护了实参所指向的变量。因为在编译阶段编译器对调用函数的选择是根据实参进行的，所以，只有引用传递和指针传递可以用是否加const来重载。一个拥有顶层const的形参无法和另一个没有顶层const的形参区分开来。

53、指针和const的用法

1） 当const修饰指针时，由于const的位置不同，它的修饰对象会有所不同。

2） int *const p2中const修饰p2的值,所以理解为p2的值不可以改变，即p2只能指向固定的一个变量地址，但可以通过*p2读写这个变量的值。顶层指针表示指针本身是

一个常量

3） int const *p1或者const int *p1两种情况中const修饰*p1，所以理解为*p1的值不可以改变，即不可以给*p1赋值改变p1指向变量的值，但可以通过给p赋值不同的地址改变这个指针指向。

底层指针表示指针所指向的变量是一个常量。

54、形参与实参的区别？

1） 形参变量只有在被调用时才分配内存单元，在调用结束时， 即刻释放所分配的内存单元。因此，形参只有在函数内部有效。函数调用结束返回主调函数后则不能再使用该形参变量。

2） 实参可以是常量、变量、表达式、函数等， 无论实参是何种类型的量，在进行函数调用时，它们都必须具有确定的值， 以便把这些值传送给形参。因此应预先用赋值，输入等办法使实参获得确定值，会产生一个临时变量。

3） 实参和形参在数量上，类型上，顺序上应严格一致， 否则会发生“类型不匹配”的错误。

4） 函数调用中发生的数据传送是单向的。即只能把实参的值传送给形参，而不能把形参的值反向地传送给实参。因此在函数调用过程中，形参的值发生改变，而实参中的值不会变化。

5） 当形参和实参不是指针类型时，在该函数运行时，形参和实参是不同的变量，他们在内存中位于不同的位置，形参将实参的内容复制一份，在该函数运行结束的时候形参被释放，而实参内容不会改变。

55、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。

56、什么是类的继承？

1）类与类之间的关系

has-A包含关系，用以描述一个类由多个部件类构成，实现has-A关系用类的成员属性表示，即一个类的成员属性是另一个已经定义好的类；

use-A，一个类使用另一个类，通过类之间的成员函数相互联系，定义友元或者通过传递参数的方式来实现；

is-A，继承关系，关系具有传递性；

2）继承的相关概念

所谓的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；

3）继承的特点

子类拥有父类的所有属性和方法，子类可以拥有父类没有的属性和方法，子类对象可以当做父类对象使用；

4）继承中的访问控制

public、protected、private

5）继承中的构造和析构函数

6）继承中的兼容性原则

57、什么是内存池，如何实现

> https://www.bilibili.com/video/BV1Kb411B7N8?p=25 C++内存管理：P23-26
> https://www.bilibili.com/video/BV1db411q7B8?p=12 C++STL P11

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。

这里简单描述一下《STL源码剖析》中的内存池实现机制：

allocate包装malloc,deallocate包装free

一般是一次20*2个的申请，先用一半，留着一半，为什么也没个说法，侯捷在STL那边书里说好像是C++委员会成员认为20是个比较好的数字，既不大也不小

1. 首先客户端会调用malloc()配置一定数量的区块（固定大小的内存块，通常为8的倍数），假设40个32bytes的区块，其中20个区块（一半）给程序实际使用，1个区块交出，另外19个处于维护状态。剩余20个（一半）留给内存池，此时一共有（20*32byte）
2. 客户端之后有有内存需求，想申请（20*64bytes）的空间，这时内存池只有（20*32bytes），就先将（10*64bytes)个区块返回，1个区块交出，另外9个处于维护状态，此时内存池空空如也
3. 接下来如果客户端还有内存需求，就必须再调用malloc()配置空间，此时新申请的区块数量会增加一个随着配置次数越来越大的附加量，同样一半提供程序使用，另一半留给内存池。申请内存的时候用永远是先看内存池有无剩余，有的话就用上，然后挂在0-15号某一条链表上，要不然就重新申请。
4. 如果整个堆的空间都不够了，就会在原先已经分配区块中寻找能满足当前需求的区块数量，能满足就返回，不能满足就向客户端报bad_alloc异常

> 《STL源码解析》侯捷 P68

allocator就是用来分配内存的，最重要的两个函数是allocate和deallocate，就是用来申请内存和回收内存的，外部（一般指容器）调用的时候只需要知道这些就够了。内部实现，目前的所有编译器都是直接调用的::operator new()和::operator delete()，说白了就是和直接使用new运算符的效果是一样的，所以老师说它们都没做任何特殊处理。

最开始GC2.9之前：

new和 operator new 的区别：new 是个运算符，编辑器会调用 operator new(0)

operator new()里面有调用malloc的操作，那同样的 operator delete()里面有调用的free的操作

GC2.9的alloc的一个比较好的分配器的实现规则

维护一条0-15号的一共16条链表，其中0表示8 bytes ，1表示 16 bytes,2表示24bytes。。。。而15 表示 16* 8 = 128bytes，如果在申请时并不是8的倍数，那就找刚好能满足内存大小的那个位置。比如想申请 12，那就是找16了，想申请 20 ，那就找 24 了

但是现在GC4.9及其之后 也还有，变成_pool_alloc这个名字了，不再是默认的了，你需要自己去指定它可以自己指定，比如说vector<string,__gnu_cxx::pool_allocvec;这样来使用它，现在用的又回到以前那种对malloc和free的包装形式了

58、从汇编层去解释一下引用

```
9:      int x = 1;
00401048  mov     dword ptr [ebp-4],1
10:     int &b = x;
0040104F   lea     eax,[ebp-4]
00401052  mov     dword ptr [ebp-8],eax
```

x的地址为ebp-4，b的地址为ebp-8，因为栈内的变量内存是从高往低进行分配的，所以b的地址比x的低。

lea eax,[ebp-4] 这条语句将x的地址ebp-4放入eax寄存器

mov dword ptr [ebp-8],eax 这条语句将eax的值放入b的地址

ebp-8中上面两条汇编的作用即：将x的地址存入变量b中，这不和将某个变量的地址存入指针变量是一样的吗？所以从汇编层次来看，的确引用是通过指针来实现的。

59、深拷贝与浅拷贝是怎么回事？

1） 浅复制 ：只是拷贝了基本类型的数据，而引用类型数据，复制后也是会发生引用，我们把这种拷贝叫做“（浅复制）浅拷贝”，换句话说，浅复制仅仅是指向被复制的内存地址，如果原地址中对象被改变了，那么浅复制出来的对象也会相应改变。

深复制 ：在计算机中开辟了一块新的内存地址用于存放复制的对象。

2） 在某些状况下，类内成员变量需要动态开辟堆内存，如果实行位拷贝，也就是把对象里的值完全复制给另一个对象，如A=B。这时，如果B中有一个成员变量指针已经申

请了内存，那A中的那个成员变量也指向同一块内存。这就出现了问题：当B把内存释放了（如：析构），这时A内的指针就是野指针了，出现运行错误。

60、C++模板是什么，你知道底层怎么实现的？

1） 编译器并不是把函数模板处理成能够处理任意类的函数；编译器从函数模板通过具体类型产生不同的函数；编译器会对函数模板进行两次编译：在声明的地方对模板代码本身进行编译，在调用的地方对参数替换后的代码进行编译。

2） 这是因为函数模板要被实例化后才能成为真正的函数，在使用函数模板的源文件中包含函数模板的头文件，如果该头文件中只有声明，没有定义，那编译器无法实例化该模板，最终导致链接错误。

61、new和malloc的区别？

1、 new/delete是C++关键字，需要编译器支持。malloc/free是库函数，需要头文件支持；

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

3、 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，无须进行类型转换，故new是符合类型安全性的操作符。而malloc内存分配成功则是返回void * ，需要通过强制类型转换将void*指针转换成我们需要的类型。

4、 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。

5、 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）。malloc/free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。

62、delete p、delete [] p、allocator都有什么作用？

1、 动态数组管理new一个数组时，[]中必须是一个整数，但是不一定是常量整数，普通数组必须是一个常量整数；

2、 new动态数组返回的并不是数组类型，而是一个元素类型的指针；

3、 delete[]时，数组中的元素按逆序的顺序进行销毁；

4、 new在内存分配上面有一些局限性，new的机制是将内存分配和对象构造组合在一起，同样的，delete也是将对象析构和内存释放组合在一起的。allocator将这两部分分开进行，allocator申请一部分内存，不进行初始化对象，只有当需要的时候才进行初始化操作。

63、new和delete的实现原理， delete是如何知道释放内存的大小的额?

1、 new简单类型直接调用operator new分配内存；

而对于复杂结构，先调用operator new分配内存，然后在分配的内存上调用构造函数；

对于简单类型，new[]计算好大小后调用operator new；

对于复杂数据结构，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，针对复杂类型，new[]会额外存储数组大小；

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简单数据类型默认只是调用free函数；复杂数据类型先调用析构函数再调用operator delete；针对简单类型，delete和delete[]等同。假设指针p指向new[]分配的内存。因为要4字节存储数组大小，实际分配的内存地址为[p-4]，系统记录的也是这个地址。delete[]实际释放的就是p-4指向的内存。而delete会直接释放p指向的内存，这个内存根本没有被系统记录，所以会崩溃。

3、 需要在 new [] 一个对象数组时，需要保存数组的维度，C++ 的做法是在分配数组空间时多分配了 4 个字节的大小，专门保存数组的大小，在 delete [] 时就可以取出这个保存的数，就知道了需要调用析构函数多少次了。

64、malloc申请的存储空间能用delete释放吗

不能，malloc /free主要为了兼容C，new和delete 完全可以取代malloc /free的。

malloc /free的操作对象都是必须明确大小的，而且不能用在动态类上。

new 和delete会自动进行类型检查和大小，malloc/free不能执行构造函数与析构函数，所以动态对象它是不行的。

当然从理论上说使用malloc申请的内存是可以通过delete释放的。不过一般不这样写的。而且也不能保证每个C++的运行时都能正常。

65、malloc与free的实现原理？

1、 在标准C库中，提供了malloc/free函数分配释放内存，这两个函数底层是由brk、mmap、，munmap这些系统调用实现的;

2、 brk是将数据段(.data)的最高地址指针_edata往高地址推,mmap是在进程的虚拟地址空间中（堆和栈中间，称为文件映射区域的地方）找一块空闲的虚拟内存。这两种方式分配的都是虚拟内存，没有分配物理内存。在第一次访问已分配的虚拟地址空间的时候，发生缺页中断，操作系统负责分配物理内存，然后建立虚拟内存和物理内存之间的映射关系;

3、 malloc小于128k的内存，使用brk分配内存，将_edata往高地址推；malloc大于128k的内存，使用mmap分配内存，在堆和栈之间找一块空闲内存分配；brk分配的内存需要等到高地址内存释放以后才能释放，而mmap分配的内存可以单独释放。当最高地址空间的空闲内存超过128K（可由M_TRIM_THRESHOLD选项调节）时，执行内存紧缩操作（trim）。在上一个步骤free的时候，发现最高地址空闲内存超过128K，于是内存紧缩。

4、 malloc是从堆里面申请内存，也就是说函数返回的指针是指向堆里面的一块内存。操作系统中有一个记录空闲内存地址的链表。当操作系统收到程序的申请时，就会遍历该链表，然后就寻找第一个空间大于所申请空间的堆结点，然后就将该结点从空闲结点链表中删除，并将该结点的空间分配给程序。

66、malloc、realloc、calloc的区别

1) malloc函数

```
void* malloc(unsigned int num_size);

int *p = malloc(20*sizeof(int));申请20个int类型的空间;
```

2) calloc函数

```
void* calloc(size_t n,size_t size);

int *p = calloc(20, sizeof(int));
```

省去了人为空间计算；malloc申请的空间的值是随机初始化的，calloc申请的空间的值是初始化为0的；

3） realloc函数

```
void realloc(void *p, size_t new_size);
```

给动态分配的空间分配额外的空间，用于扩充容量。

67、类成员初始化方式？构造函数的执行顺序 ？为什么用成员初始化列表会快一些？

1） 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。

这两种方式的主要区别在于：

对于在函数体中初始化,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,就是说分配一个数据成员只要冒号后有此数据成员的赋值表达式(此表达式必须是括号赋值表达式),那么分配了内存空间后在进入函数体之前给数据成员赋值，就是说初始化这个数据成员此时函数体还未执行。

2） 一个派生类构造函数的执行顺序如下：

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（按照初始化顺序）

④ 派生类自己的构造函数。

3） 方法一是在构造函数当中做赋值的操作，而方法二是做纯粹的初始化操作。我们都知道，C++的赋值操作是会产生临时对象的。临时对象的出现会降低程序的效率。

68、成员列表初始化?

1） 必须使用成员初始化的四种情况

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；

2） 成员初始化列表做了什么

① 编译器会一一操作初始化列表，以适当的顺序在构造函数之内安插初始化操作，并且在任何显示用户代码之前；

② list中的项目顺序是由类中的成员声明顺序决定的，不是由初始化列表的顺序决定的；

69、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**。堆内存是指程序从堆中分配的，大小任意的(内存块的大小可以在程序运行期决定)内存块，使用完后必须显式释放的内存。应用程序般使用malloc,、realloc、 new等函数从堆中分配到块内存，使用完后，程序必须负责相应的调用free或delete释放该内存块，否则，这块内存就不能被再次使用，我们就说这块内存泄漏了

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**CRT库**

70、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。

在C++中，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：

```
#include <vector>
#include <string>
#include <iostream>
using namespace std;

struct Person
{
    string name;
    int age;
    //初始构造函数
    Person(string p_name, int p_age): name(std::move(p_name)), age(p_age)
    {
         cout << "I have been constructed" <<endl;
    }
     //拷贝构造函数
     Person(const Person& other): name(std::move(other.name)), age(other.age)
    {
         cout << "I have been copy constructed" <<endl;
    }
     //转移构造函数
     Person(Person&& other): name(std::move(other.name)), age(other.age)
    {
         cout << "I have been moved"<<endl;
    }
};
```

```
int main()
{
    vector<Person> e;
    cout << "emplace_back:" <<endl;
    e.emplace_back("Jane", 23); //不用构造类对象

    vector<Person> p;
    cout << "push_back:"<<endl;
    p.push_back(Person("Mike",36));
    return 0;
}
//输出结果:
//emplace_back:
//I have been constructed
//push_back:
//I have been constructed
//I am being moved.
```

71、解释一下什么是trivial destructor

“**trivial destructor**”一般是指用户没有自定义析构函数，而由系统生成的，这种析构函数在《STL源码解析》中成为“无关痛痒”的析构函数。

反之，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**

对于trivial destructor，如果每次都进行调用，显然对效率是一种伤害，如何进行判断呢？《STL源码解析》中给出的说明是：

> 首先利用value_type()获取所指对象的型别，再利用__type_traits判断该型别的析构函数是否trivial，若是(__true_type)，则什么也不做，若为(__false_type)，则去调用destory()函数

也就是说，在实际的应用当中，STL库提供了相关的判断方法**__type_traits**，感兴趣的读者可以自行查阅使用方式。除了trivial destructor，还有trivial construct、trivial copy construct等，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。

> 《C++中的trivial destructor》：https://blog.csdn.net/wudishine/article/details/12307611

72、介绍面向对象的三大特性，并且举例说明

三大特性：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。

封装，也就是**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**（重载实现编译时多态，虚函数实现运行时多态）**。

多态性是允许你将父对象设置成为和一个或更多的他的子对象相等的技术，赋值之后，父对象就可以根据当前赋值给它的子对象的特性以不同的方式运作。**简单一句话：允许将子类类型的指针赋值给父类类型的指针**

实现多态有二种方式：覆盖（override），重载（overload）。覆盖：是指子类重新定义父类的虚函数的做法。重载：是指允许存在多个同名函数，而这些函数的参数表不同（或许参数个数不同，或许参数类型不同，或许两者都不同）。例如：基类是一个抽象对象——人，那教师、运动员也是人，而使用这个抽象对象既可以表示教师、也可以表示运动员。

> 《C++封装继承多态总结》：https://blog.csdn.net/IOT_SHUN/article/details/79674293

73、C++中类的数据成员和成员函数内存分布情况

C++类是由结构体发展得来的，所以他们的成员变量（C语言的结构体只有成员变量）的内存分配机制是一样的。下面我们以类来说明问题，如果类的问题通了，结构体也也就没问题啦。类分为成员变量和成员函数，我们先来讨论成员变量。

一个类对象的地址就是类所包含的这一片内存空间的首地址，这个首地址也就对应具体某一个成员变量的地址。（在定义类对象的同时这些成员变量也就被定义了），举个例子：

```
#include <iostream>
using namespace std;

class Person
{
public:
    Person()
    {
        this->age = 23;
    }
    void printAge()
    {
        cout << this->age <<endl;
    }
    ~Person(){}
public:
    int age;
};

int main()
{
    Person p;
    cout << "对象地址："<< &p <<endl;
    cout << "age地址："<< &(p.age) <<endl;
    cout << "对象大小："<< sizeof(p) <<endl;
    cout << "age大小："<< sizeof(p.age) <<endl;
    return 0;
}
//输出结果
//对象地址：0x7fffec0f15a8
//age地址：0x7fffec0f15a8
//对象大小：4
//age大小：4
```

从代码运行结果来看，对象的大小和对象中数据成员的大小是一致的，也就是说，成员函数不占用对象的内存。这是因为所有的函数都是存放在代码区的，不管是全局函数，还是成员函数。要是成员函数占用类的对象空间，那么将是多么可怕的事情：定义一次

类对象就有成员函数占用一段空间。我们再来补充一下静态成员函数的存放问题：**静态成员函数与一般成员函数的唯一区别就是没有this指针**，因此不能访问非静态数据成员，就像我前面提到的，**所有函数都存放在代码区，静态函数也不例外。所有有人一看到 static 这个单词就主观的认为是存放在全局数据区，那是不对的。**

> 《C++类对象成员变量和函数内存分配的问题》：https://blog.csdn.net/z2664836046/article/details/78967313

74、成员初始化列表的概念，为什么用它会快一些？

成员初始化列表的概念

在类的构造函数中，不在函数体内对成员变量赋值，而是在构造函数的花括号前面使用冒号和初始化列表赋值

效率

用初始化列表会快一些的原因是，对于类型，它少了一次调用构造函数的过程，而在函数体中赋值则会多一次调用。而对于内置数据类型则没有差别。举个例子：

```
#include <iostream>
using namespace std;
class A
{
public:
    A()
    {
        cout << "默认构造函数A()" << endl;
    }
    A(int a)
    {
        value = a;
        cout << "A(int "<<value<<")" << endl;
    }
    A(const A& a)
    {
        value = a.value;
        cout << "拷贝构造函数A(A& a):  "<<value << endl;
    }
    int value;
};

class B
{
public:
    B() : a(1)
    {
        b = A(2);
    }
```

```
    A a;
    A b;
};
int main()
{
    B b;
}

//输出结果:
//A(int 1)
//默认构造函数A()
//A(int 2)
```

从代码运行结果可以看出，在构造函数体内部初始化的对象b多了一次构造函数的调用过程，而对象a则没有。由于对象成员变量的初始化动作发生在进入构造函数之前，对于内置类型没什么影响，但**如果有些成员是类**，那么在进入构造函数之前，会先调用一次默认构造函数，进入构造函数后所做的事其实是一次赋值操作(对象已存在)，所以**如果是在构造函数体内进行赋值的话，等于是一次默认构造加一次赋值，而初始化列表只做一次赋值操作。**

> 《为什么用成员初始化列表会快一些？》：https://blog.csdn.net/JackZhang_123/article/details/82590368

75、(超重要)构造函数为什么不能为虚函数？析构函数为什么要虚函数？

1、 从存储空间角度，虚函数相应一个指向vtable虚函数表的指针，这大家都知道，但是这个指向vtable的指针事实上是存储在对象的内存空间的。

问题出来了，假设构造函数是虚的，就须要通过 vtable来调用，但是对象还没有实例化，也就是内存空间还没有，怎么找vtable呢？所以构造函数不能是虚函数。

2、 从使用角度，虚函数主要用于在信息不全的情况下，能使重载的函数得到相应的调用。

构造函数本身就是要初始化实例，那使用虚函数也没有实际意义呀。

所以构造函数没有必要是虚函数。虚函数的作用在于通过父类的指针或者引用来调用它的时候可以变成调用子类的那个成员函数。而构造函数是在创建对象时自己主动调用的，不可能通过父类的指针或者引用去调用，因此也就规定构造函数不能是虚函数。

3、构造函数不须要是虚函数，也不同意是虚函数，由于创建一个对象时我们总是要明白指定对象的类型，虽然我们可能通过实验室的基类的指针或引用去訪问它但析构却不

一定，我们往往通过基类的指针来销毁对象。这时候假设析构函数不是虚函数，就不能正确识别对象类型从而不能正确调用析构函数。

4、从实现上看，vbtl在构造函数调用后才建立，因而构造函数不可能成为虚函数从实际含义上看，在调用构造函数时还不能确定对象的真实类型（由于子类会调父类的构造函数）；并且构造函数的作用是提供初始化，在对象生命期仅仅运行一次，不是对象的动态行为，也没有必要成为虚函数。

5、当一个构造函数被调用时，它做的首要的事情之中的一个是初始化它的VPTR。

因此，它仅仅能知道它是“当前”类的，而全然忽视这个对象后面是否还有继承者。当编译器为这个构造函数产生代码时，它是为这个类的构造函数产生代码——既不是为基类，也不是为它的派生类（由于类不知道谁继承它）。所以它使用的VPTR必须是对于这个类的VTABLE。

并且，仅仅要它是最后的构造函数调用，那么在这个对象的生命期内，VPTR将保持被初始化为指向这个VTABLE，但假设接着另一个更晚派生的构造函数被调用，这个构造函数又将设置VPTR指向它的 VTABLE，等.直到最后的构造函数结束。

VPTR的状态是由被最后调用的构造函数确定的。这就是为什么构造函数调用是从基类到更加派生类顺序的还有一个理由。可是，当这一系列构造函数调用正发生时，每一个构造函数都已经设置VPTR指向它自己的VTABLE。假设函数调用使用虚机制，它将仅仅产生通过它自己的VTABLE的调用，而不是最后的VTABLE（全部构造函数被调用后才会有最后的VTABLE）。

因为构造函数本来就是为了明确初始化对象成员才产生的，然而virtual function主要是为了再不完全了解细节的情况下也能正确处理对象。另外，virtual函数是在不同类型的对象产生不同的动作，现在对象还没有产生，如何使用virtual函数来完成你想完成的动作。

直接的讲，C++中基类采用virtual虚析构函数是**为了防止内存泄漏。**

具体地说，如果派生类中申请了内存空间，并在其析构函数中对这些内存空间进行释放。假设基类中采用的是非虚析构函数，当删除基类指针指向的派生类对象时就不会触发动态绑定，因而只会调用基类的析构函数，而不会调用派生类的析构函数。那么在这种情况下，派生类中申请的空间就得不到释放从而产生内存泄漏。

所以，为了防止这种情况的发生，C++中基类的析构函数应采用virtual虚析构函数。

76、析构函数的作用，如何起作用？

1） 构造函数只是起初始化值的作用，但实例化一个对象的时候，可以通过实例去传递参数，从主函数传递到其他的函数里面，这样就使其他的函数里面有值了。

规则，只要你一实例化对象，系统自动回调用一个构造函数就是你不写，编译器也自动调用一次。

2） 析构函数与构造函数的作用相反，用于撤销对象的一些特殊任务处理，可以是释放对象分配的内存空间；特点：析构函数与构造函数同名，但该函数前面加~。

析构函数没有参数，也没有返回值，而且不能重载，在一个类中只能有一个析构函数。当撤销对象时，编译器也会自动调用析构函数。

每一个类必须有一个析构函数，用户可以自定义析构函数，也可以是编译器自动生成默认的析构函数。一般析构函数定义为类的公有成员。

77、构造函数和析构函数可以调用虚函数吗，为什么

1）在C++中，提倡不在构造函数和析构函数中调用虚函数；

2）构造函数和析构函数调用虚函数时都不使用动态联编，如果在构造函数或析构函数中调用虚函数，则运行的是为构造函数或析构函数自身类型定义的版本；

3）因为父类对象会在子类之前进行构造，此时子类部分的数据成员还未初始化，因此调用子类的虚函数时不安全的，故而C++不会进行动态联编；

4）析构函数是用来销毁一个对象的，在销毁一个对象时，先调用子类的析构函数，然后再调用基类的析构函数。所以在调用基类的析构函数时，派生类对象的数据成员已经销毁，这个时候再调用子类的虚函数没有任何意义。

78、构造函数、析构函数的执行顺序？构造函数和拷贝构造的内部都干了啥？

1） 构造函数顺序

① 基类构造函数。如果有多个基类，则构造函数的调用顺序是某类在类派生表中出现的顺序，而不是它们在成员初始化表中的顺序。

② 成员类对象构造函数。如果有多个成员类对象则构造函数的调用顺序是对象在类中被声明的顺序，而不是它们出现在成员初始化表中的顺序。

③ 派生类构造函数。

2） 析构函数顺序

① 调用派生类的析构函数；

② 调用成员类对象的析构函数；

③ 调用基类的析构函数。

79、虚析构函数的作用，父类的析构函数是否要设置为虚函数？

1） C++中基类采用virtual虚析构函数是为了防止内存泄漏。

具体地说，如果派生类中申请了内存空间，并在其析构函数中对这些内存空间进行释放。

假设基类中采用的是非虚析构函数，当删除基类指针指向的派生类对象时就不会触发动态绑定，因而只会调用基类的析构函数，而不会调用派生类的析构函数。

那么在这种情况下，派生类中申请的空间就得不到释放从而产生内存泄漏。

所以，为了防止这种情况的发生，C++中基类的析构函数应采用virtual虚析构函数。

2） 纯虚析构函数一定得定义，因为每一个派生类析构函数会被编译器加以扩张，以静态调用的方式调用其每一个虚基类以及上一层基类的析构函数。

因此，缺乏任何一个基类析构函数的定义，就会导致链接失败，最好不要把虚析构函数定义为纯虚析构函数。

80、构造函数析构函数可否抛出异常

1） C++只会析构已经完成的对象，对象只有在其构造函数执行完毕才算是完全构造妥当。在构造函数中发生异常，控制权转出构造函数之外。

因此，在对象b的构造函数中发生异常，对象b的析构函数不会被调用。因此会造成内存泄漏。

2） 用auto_ptr对象来取代指针类成员，便对构造函数做了强化，免除了抛出异常时发生资源泄漏的危机，不再需要在析构函数中手动释放资源；

3） 如果控制权基于异常的因素离开析构函数，而此时正有另一个异常处于作用状态，C++会调用terminate函数让程序结束；

4） 如果异常从析构函数抛出，而且没有在当地进行捕捉，那个析构函数便是执行不全的。如果析构函数执行不全，就是没有完成他应该执行的每一件事情。

81、构造函数一般不定义为虚函数的原因

（1）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动态确定其类型的。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

（2）虚函数的调用需要虚函数表指针vptr，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vtable地址用来调用虚构造函数了

（3）虚函数的作用在于通过父类的指针或者引用调用它的时候能够变成调用子类的那个成员函数。而构造函数是在创建对象时自动调用的，不可能通过父类或者引用去调用，因此就规定构造函数不能是虚函数

（4）析构函数一般都要声明为虚函数，这个应该是老生常谈了，这里不再赘述

> 《为什么C++不能有虚构造函数，却可以有虚析构函数》：https://dwz.cn/lnfW9H6m

82、类什么时候会析构?

1） 对象生命周期结束，被销毁时；

2） delete指向对象的指针时，或delete指向对象的基类类型指针，而其基类虚构函数是虚函数时；

3） 对象i是对象o的成员，o的析构函数被调用时，对象i的析构函数也被调用。

83、构造函数或者析构函数中可以调用虚函数吗

简要结论：

- 从语法上讲，调用完全没有问题。
- 但是从效果上看，往往不能达到需要的目的。

> 《Effective C++》的解释是：
> 派生类对象构造期间进入基类的构造函数时，对象类型变成了基类类型，而不是派生类类型。同样，进入基类析构函数时，对象也是基类类型。

举个例子：

```
#include<iostream>
using namespace std;

class Base
{
public:
    Base()
    {
       Function();
    }

    virtual void Function()
    {
        cout << "Base::Fuction" << endl;
    }
    ~Base()
    {
        Function();
    }
};

class A : public Base
{
public:
    A()
    {
      Function();
    }

    virtual void Function()
    {
        cout << "A::Function" << endl;
    }
    ~A()
    {
        Function();
    }
};

int main()
{
    Base* a = new Base;
    delete a;
    cout << "-------------------------" <<endl;
    Base* b = new A;//语句1
    delete b;
}
//输出结果
//Base::Fuction
//Base::Fuction
//-------------------------
//Base::Fuction
//A::Function
//Base::Fuction
```

语句1讲道理应该体现多态性，执行类A中的构造和析构函数，从实验结果来看，语句1并没有体现，执行流程是先构造基类，所以先调用基类的构造函数，构造完成再执行A自己的构造函数，析构时也是调用基类的析构函数，也就是说构造和析构中调用虚函数并不能达到目的，应该避免

> 《构造函数或者析构函数中调用虚函数会怎么样？》：https://dwz.cn/TaJTJONX

84、智能指针的原理、常用的智能指针及实现

原理

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

实现原理：采用引用计数器的方法，允许多个智能指针指向同一个对象，每当多一个指针指向该对象时，指向该对象的所有智能指针内部的引用计数加1，每当减少一个智能指针指向对象时，引用计数会减1，当计数为0的时候会自动的释放动态分配的资源。

- 智能指针将一个计数器与类指向的对象相关联，引用计数器跟踪共有多少个类对象共享同一指针
- 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）

(2) unique_ptr

unique_ptr采用的是独享所有权语义，一个非空的unique_ptr总是拥有它所指向的资源。转移一个unique_ptr将会把所有权全部从源指针转移给目标指针，源指针被置空；所以unique_ptr不支持普通的拷贝和赋值操作，不能用在STL标准容器中；局部变量的返回值除外（因为编译器知道要返回的对象将要被销毁）；如果你拷贝一个

unique_ptr，那么拷贝结束后，这两个unique_ptr都会指向相同的资源，造成在结束时对同一内存指针多次释放而导致程序崩溃。

(3) weak_ptr

weak_ptr：弱引用。引用计数有一个问题就是互相引用形成环（环形引用），这样两个指针指向的内存都无法释放。需要使用weak_ptr打破环形引用。weak_ptr是一个弱引用，它是为了配合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，也就是说，它只引用，不计数。如果一块内存被shared_ptr和weak_ptr同时引用，当所有shared_ptr析构了之后，不管还有没有weak_ptr引用该内存，内存也会被释放。所以weak_ptr不保证它指向的内存一定是有效的，在使用之前使用函数lock()检查weak_ptr是否为空指针。

(4) auto_ptr

主要是为了解决“有异常抛出时发生内存泄漏”的问题 。因为发生异常而无法正常释放内存。

auto_ptr有拷贝语义，拷贝后源对象变得无效，这可能引发很严重的问题；而unique_ptr则无拷贝语义，但提供了移动语义，这样的错误不再可能发生，因为很明显必须使用std::move()进行转移。

auto_ptr不支持拷贝和赋值操作，不能用在STL标准容器中。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。

智能指针shared_ptr代码实现：

```
template<typename T>
class SharedPtr
{
public:
    SharedPtr(T* ptr = NULL):_ptr(ptr), _pcount(new int(1))
    {}

    SharedPtr(const SharedPtr& s):_ptr(s._ptr), _pcount(s._pcount){
        *(_pcount)++;
    }

    SharedPtr<T>& operator=(const SharedPtr& s){
        if (this != &s)
        {
            if (--(*(this->_pcount)) == 0)
            {
                delete this->_ptr;
                delete this->_pcount;
            }
             ptr = s. ptr:
```

```
            _pcount = s._pcount;
            *(_pcount)++;
        }
        return *this;
    }
    T& operator*()
    {
        return *(this->_ptr);
    }
    T* operator->()
    {
        return this->_ptr;
    }
    ~SharedPtr()
    {
        --(*(this->_pcount));
        if (this->_pcount == 0)
        {
            delete _ptr;
            _ptr = NULL;
            delete _pcount;
            _pcount = NULL;
        }
    }
private:
    T* _ptr;
    int* _pcount;//指向引用计数的指针
};
```

> 《智能指针的原理及实现》：https://blog.csdn.net/lizhentao0707/article/details/81156384

85、构造函数的几种关键字

default

default关键字可以显式要求编译器生成合成构造函数，防止在调用时相关构造函数类型没有定义而报错

```
#include <iostream>
using namespace std;

class CString
{
public:
    CString() = default; //语句1
    //构造函数
    CString(const char* pstr) : _str(pstr){}
    void* operator new() = delete;//这样不允许使用new关键字
    //析构函数
    ~CString(){}
public:
     string  str;
```

```
};


int main()
{
   auto a = new CString(); //语句2
   cout << "Hello World" <<endl;
   return 0;
}
//运行结果
//Hello World
```

如果没有加语句1，语句2会报错，表示找不到参数为空的构造函数，将其设置为default可以解决这个问题

delete

delete关键字可以删除构造函数、赋值运算符函数等，这样在使用的时候会得到友善的提示

```
#include <iostream>
using namespace std;

class CString
{
public:
    void* operator new() = delete;//这样不允许使用new关键字
    //析构函数
    ~CString(){}
};


int main()
{
   auto a = new CString(); //语句1
   cout << "Hello World" <<endl;
   return 0;
}
```

在执行语句1时，会提示new方法已经被删除，如果将new设置为私有方法，则会报惨不忍睹的错误，因此使用delete关键字可以更加人性化的删除一些默认方法

=0

将虚函数定义为纯虚函数（纯虚函数无需定义，= 0只能出现在类内部虚函数的声明语句处；当然，也可以为纯虚函数提供定义，不过函数体必须定义在类的外部）

> 《C++构造函数的default和delete》：https://blog.csdn.net/u010591680/article/details/71101737

86、C++的四种强制转换reinterpret_cast/const_cast/static_cast/dynamic_cast

reinterpret_cast

reinterpret_cast(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(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

static_cast < type-id > (expression)

该运算符把expression转换为type-id类型，但没有运行时类型检查来保证转换的安全性。它主要有如下几种用法：

- 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，由于没有动态类型检查，所以是不安全的
- 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

注意：static_cast不能转换掉expression的const、volatile、或者__unaligned属性。

dynamic_cast

有类型检查，基类向派生类转换比较安全，但是派生类向基类转换则不太安全

dynamic_cast(expression)

该运算符把expression转换成type-id类型的对象。type-id 必须是类的指针、类的引用或者void*

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

dynamic_cast运算符可以在执行期决定真正的类型，也就是说expression必须是多态类型。如果下行转换是安全的（也就说，如果基类指针或者引用确实指向一个派生类对象）这个运算符会传回适当转型过的指针。如果 如果下行转换不安全，这个运算符会传回空指针（也就是说，基类指针或者引用没有指向一个派生类对象）

dynamic_cast主要用于类层次间的上行转换和下行转换，还可以用于类之间的交叉转换

在类层次间进行上行转换时，dynamic_cast和static_cast的效果是一样的

在进行下行转换时，dynamic_cast具有类型检查的功能，比static_cast更安全

举个例子：

```
#include <bits/stdc++.h>
using namespace std;

class Base
{
public:
    Base() :b(1) {}
    virtual void fun() {};
    int b;
};

class Son : public Base
```

```
{
public:
    Son() :d(2) {}
    int d;
};

int main()
{
    int n = 97;

    //reinterpret_cast
    int *p = &n;
    //以下两者效果相同
    char *c = reinterpret_cast<char*> (p);
    char *c2 =  (char*)(p);
    cout << "reinterpret_cast输出: "<< *c2 << endl;
    //const_cast
    const int *p2 = &n;
    int *p3 = const_cast<int*>(p2);
    *p3 = 100;
    cout << "const_cast输出: " << *p3 << endl;

    Base* b1 = new Son;
    Base* b2 = new Base;

    //static_cast
    Son* s1 = static_cast<Son*>(b1); //同类型转换
    Son* s2 = static_cast<Son*>(b2); //下行转换，不安全
    cout << "static_cast输出: "<< endl;
    cout << s1->d << endl;
    cout << s2->d << endl; //下行转换，原先父对象没有d成员，输出垃圾值

    //dynamic_cast
    Son* s3 = dynamic_cast<Son*>(b1); //同类型转换
    Son* s4 = dynamic_cast<Son*>(b2); //下行转换，安全
    cout << "dynamic_cast输出: " << endl;
    cout << s3->d << endl;
    if(s4 == nullptr)
        cout << "s4指针为nullptr" << endl;
    else
        cout << s4->d << endl;


    return 0;
}
//输出结果
//reinterpret_cast输出: a
//const_cast输出: 100
//static_cast输出:
//2
//-33686019
//dynamic_cast输出:
//2
//s4指针为nullptr
```

从输出结果可以看出，在进行下行转换时，dynamic_cast安全的，如果下行转换不安全的话其会返回空指针，这样在进行操作的时候可以预先判断。而使用static_cast下行转换存在不安全的情况也可以转换成功，但是直接使用转换后的对象进行操作容易造成错误。

87、C++函数调用的压栈过程

从代码入手，解释这个过程：

```
#include <iostream>
using namespace std;

int f(int n)
{
    cout << n << endl;
    return n;
}

void func(int param1, int param2)
{
    int var1 = param1;
    int var2 = param2;
    printf("var1=%d,var2=%d", f(var1), f(var2));
}

int main(int argc, char* argv[])
{
    func(1, 2);
    return 0;
}
//输出结果
//2
//1
//var1=1,var2=2
```

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。

> 《C/C++函数调用过程分析》：https://www.cnblogs.com/biyeymyhjob/archive/2012/07/20/2601204.html
> 《C/C++函数调用的压栈模型》：https://blog.csdn.net/m0_37717595/article/details/80368411

88、说说移动构造函数

1） 我们用对象a初始化对象b，后对象a我们就不在使用了，但是对象a的空间还在呀（在析构之前），既然拷贝构造函数，实际上就是把a对象的内容复制一份到b中，那么为什么我们不能直接使用a的空间呢？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；

2） 拷贝构造函数中，对于指针，我们一定要采用深层复制，而移动构造函数中，对于指针，我们采用浅层复制。浅层复制之所以危险，是因为两个指针共同指向一片内存空间，若第一个指针将其释放，另一个指针的指向就不合法了。

所以我们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；

3） 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初值是一个右值引用。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，只用用一个右值，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。而那个move语句，就是将一个左值变成一个将亡值。

89、C++中将临时变量作为返回值时的处理过程

首先需要明白一件事情，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

C语言里规定：16bit程序中，返回值保存在ax寄存器中，32bit程序中，返回值保持在eax寄存器中，如果是64bit返回值，edx寄存器保存高32bit，eax寄存器保存低32bit

由此可见，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

如果我们需要返回值，一般使用赋值语句就可以了

> 《【C++】临时变量不能作为函数的返回值？》：https://www.wandouip.com/t5i204349/

（栈上的内存分配、拷贝过程）

90、关于this指针你知道什么？全说出来

- this指针是类的指针，指向对象的首地址。
- this指针只能在成员函数中使用，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。

this指针的用处

一个对象的this指针并不是对象本身的一部分，不会影响sizeof(对象)的结果。this作用域是在类内部，当在类的**非静态成员函数**中访问类的**非静态成员**的时候（全局函数，静态函数中不能使用this指针），编译器会自动将对象本身的地址作为一个隐含参数传递给函数。也就是说，即使你没有写上this指针，编译器在编译的时候也是加上this的，它作为非静态成员函数的隐含形参，对各成员的访问均通过this进行

this指针的使用

一种情况就是，在类的非静态成员函数中返回类对象本身的时候，直接使用 return *this;

另外一种情况是当形参数与成员变量名相同时用于区分，如this->n = n （不能写成n = n）

类的this指针有以下特点

(1) **this**只能在成员函数中使用，全局函数、静态函数都不能使用this。实际上，**成员函数默认第一个参数**为**T * const this**

如：

```
class A{
public:
    int func(int p){}
};
```

其中，**func**的原型在编译器看来应该是：

```
int func(A * const this,int p);
```

（2）由此可见，**this**在成员函数的开始前构造，在成员函数的结束后清除。这个生命周期同任何一个函数的参数是一样的，没有任何区别。当调用一个类的成员函数时，编译器将类的指针作为函数的this参数传递进去。如：

```
A a;
a.func(10);
//此处，编译器将会编译成：
A::func(&a,10);
```

看起来和静态函数没差别，对吗？不过，区别还是有的。编译器通常会对this指针做一些优化，因此，this指针的传递效率比较高，例如VC通常是通过ecx（计数寄存器）传递this参数的。

91、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。

但是如果class或者struct里面没有方法的话，它们是没有构造函数的，只能当做C的struct使用。采用TYPE xx的方式定义的话，在栈里分配内存，这时候this指针的值就是这块内存的地址。采用new的方式创建对象的话，在堆里分配内存，new操作符通过eax（累加寄存器）返回分配的地址，然后设置给指针变量。之后去调用构造函数（如果有构造函数的话），这时将这个内存块的地址传给ecx，之后构造函数里面怎么处理请看上面的回答

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。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。

C. this指针是如何传递类中的函数的？绑定？还是在函数参数的首参数就是this指针？那么，this指针又是如何找到“类实例后函数的”？

大多数编译器通过ecx（寄数寄存器）寄存器传递this指针。事实上，这也是一个潜规则。一般来说，不同编译器都会遵从一致的传参规则，否则不同编译器产生的obj就无法匹配了。

在call之前，编译器会把对应的对象地址放到eax中。this是通过函数参数的首参来传递的。this指针在调用之前生成，至于“类实例后函数”，没有这个说法。类在实例化时，只分配类中的变量空间，并没有为函数分配空间。自从类的函数定义完成后，它就在那儿，不会跑的

D. this指针是如何访问类中的变量的?

如果不是类，而是结构体的话，那么，如何通过结构指针来访问结构中的变量呢？如果你明白这一点的话，就很容易理解这个问题了。

在C++中，类和结构是只有一个区别的：类的成员默认是private，而结构是public。

this是类的指针，如果换成结构体，那this就是结构的指针了。

E.我们只有获得一个对象后，才能通过对象使用this指针。如果我们知道一个对象this指针的位置，可以直接使用吗?

this指针只有在成员函数中才有定义。因此，你获得一个对象后，也不能通过对象使用this指针。所以，我们无法知道一个对象的this指针的位置（只有在成员函数里才有this指针的位置）。当然，在成员函数里，你是可以知道this指针的位置的（可以通过&this获得），也可以直接使用它。

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?

普通的类函数（不论是成员函数，还是静态函数）都不会创建一个函数表来保存函数指针。只有虚函数才会被放到函数表中。但是，即使是虚函数，如果编译期就能明确知道调用的是哪个函数，编译器就不会通过函数表中的指针来间接调用，而是会直接调用该函数。正是由于this指针的存在，用来指向不同的对象，从而确保不同对象之间调用相同的函数可以互不干扰

> 《C++ 中 this 指针的用法详解》 http://blog.chinaunix.net/uid-21411227-id-1826942.html

92、构造函数、拷贝构造函数和赋值操作符的区别

构造函数

对象不存在，没用别的对象初始化，在创建一个新的对象时调用构造函数

拷贝构造函数

对象不存在，但是使用别的已经存在的对象来进行初始化

赋值运算符

对象存在，用别的对象给它赋值，这属于重载“=”号运算符的范畴，“=”号两侧的对象都是已存在的

举个例子：

```
#include <iostream>
using namespace std;

class A
{
public:
    A()
    {
        cout << "我是构造函数" << endl;
    }
    A(const A& a)
    {
        cout << "我是拷贝构造函数" << endl;
    }
    A& operator = (A& a)
    {
        cout << "我是赋值操作符" << endl;
        return *this;
    }
    ~A() {};
};

int main()
{
    A a1; //调用构造函数
    A a2 = a1; //调用拷贝构造函数
    a2 = a1; //调用赋值操作符
    return 0;
}
//输出结果
//我是构造函数
//我是拷贝构造函数
//我是赋值操作符
```

93、拷贝构造函数和赋值运算符重载的区别？

- 拷贝构造函数是函数，赋值运算符是运算符重载。
- 拷贝构造函数会生成新的类对象，赋值运算符不能。

- 拷贝构造函数是直接构造一个新的类对象，所以在初始化对象前不需要检查源对象和新建对象是否相同；赋值运算符需要上述操作并提供两套不同的复制策略，另外赋值运算符中如果原来的对象有内存分配则需要先把内存释放掉。

- 形参传递是调用拷贝构造函数（调用的被赋值对象的拷贝构造函数），但并不是所有出现"="的地方都是使用赋值运算符，如下：

```
Student s;
 Student s1 = s;     // 调用拷贝构造函数
 Student s2;
 s2 = s;     // 赋值运算符操作
```

注：类中有指针变量时要重写析构函数、拷贝构造函数和赋值运算符

94、智能指针的作用；

1） C++11中引入了智能指针的概念，方便管理堆内存。使用普通指针，容易造成堆内存泄露（忘记释放），二次释放，程序发生异常时内存泄露等问题等，使用智能指针能更好的管理堆内存。

2） 智能指针在C++11版本之后提供，包含在头文件中，shared_ptr、unique_ptr、weak_ptr。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。

3） 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_ptrp4 = 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

4） 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。

5） 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）。

6） weak_ptr 是一种不控制对象生命周期的智能指针，它指向一个 shared_ptr 管理的对象. 进行该对象的内存管理的是那个强引用的 shared_ptr. 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.

95、说说你了解的auto_ptr作用

1） auto_ptr的出现，主要是为了解决“有异常抛出时发生内存泄漏”的问题；抛出异常，将导致指针p所指向的空间得不到释放而导致内存泄漏；

2） auto_ptr构造时取得某个对象的控制权，在析构时释放该对象。我们实际上是创建一个auto_ptr类型的局部对象，该局部对象析构时，会将自身所拥有的指针空间释放，所以不会有内存泄漏；

3） auto_ptr的构造函数是explicit，阻止了一般指针隐式转换为 auto_ptr的构造，所以不能直接将一般类型的指针赋值给auto_ptr类型的对象，必须用auto_ptr的构造函数创建对象；

4） 由于auto_ptr对象析构时会删除它所拥有的指针，所以使用时避免多个auto_ptr对象管理同一个指针；

5） Auto_ptr内部实现，析构函数中删除对象用的是delete而不是delete[]，所以auto_ptr不能管理数组；

6） auto_ptr支持所拥有的指针类型之间的隐式类型转换。

7） 可以通过*和->运算符对auto_ptr所有用的指针进行提领操作；

8） T* get(),获得auto_ptr所拥有的指针；T* release()，释放auto_ptr的所有权，并将所有用的指针返回。

96、智能指针的循环引用

循环引用是指使用多个智能指针share_ptr时，出现了指针之间相互指向，从而形成环的情况，有点类似于死锁的情况，这种情况下，智能指针往往不能正常调用对象的析构函数，从而造成内存泄漏。举个例子：

```
#include <iostream>
using namespace std;

template <typename T>
class Node
{
public:
    Node(const T& value)
        :_pPre(NULL)
        , _pNext(NULL)
        , _value(value)
    {
        cout << "Node()" << endl;
    }
    ~Node()
    {
        cout << "~Node()" << endl;
        cout << "this:" << this << endl;
    }

    shared_ptr<Node<T>> _pPre;
    shared_ptr<Node<T>> _pNext;
    T _value;
};

void Funtest()
{
    shared_ptr<Node<int>> sp1(new Node<int>(1));
    shared_ptr<Node<int>> sp2(new Node<int>(2));

    cout << "sp1.use_count:" << sp1.use_count() << endl;
    cout << "sp2.use_count:" << sp2.use_count() << endl;

    sp1->_pNext = sp2; //sp1的引用+1
    sp2->_pPre = sp1; //sp2的引用+1

    cout << "sp1.use_count:" << sp1.use_count() << endl;
    cout << "sp2.use_count:" << sp2.use_count() << endl;
}
int main()
{
    Funtest();
    system("pause");
    return 0;
}
//输出结果
//Node()
//Node()
//sp1.use_count:1
//sp2.use_count:1
//sp1.use_count:2
//sp2.use_count:2
```

从上面shared_ptr的实现中我们知道了只有当引用计数减减之后等于0，析构时才会释放对象，而上述情况造成了一个僵局，那就是析构对象时先析构sp2,可是由于sp2的空间sp1还在使用中，所以sp2.use_count减减之后为1，不释放，sp1也是相同的道理，由于sp1的空间sp2还在使用中，所以sp1.use_count减减之后为1，也不释放。sp1等着sp2先释放，sp2等着sp1先释放,二者互不相让，导致最终都没能释放，内存泄漏。

在实际编程过程中，应该尽量避免出现智能指针之间相互指向的情况，如果不可避免，可以使用弱指针—weak_ptr，它不增加引用计数，只要出了作用域就会自动析构。

> 《C++智能指针（及循环引用问题）》：https://blog.csdn.net/m0_37968340/article/details/76737395

97、什么是虚拟继承

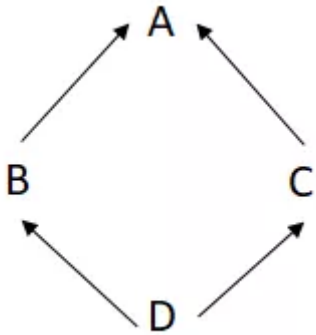
由于C++支持多继承，除了public、protected和private三种继承方式外，还支持虚拟（virtual）继承，举个例子：

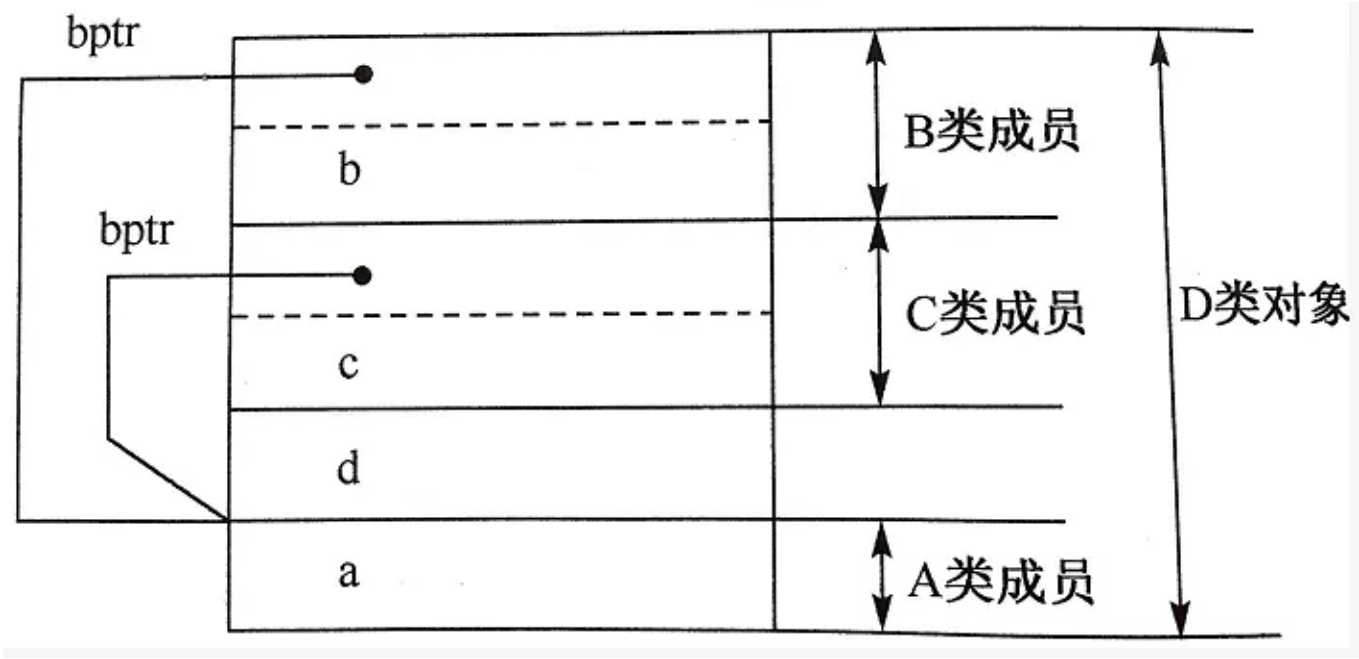
```
#include <iostream>
using namespace std;

class A{}
class B : virtual public A{};
class C : virtual public A{};
class D : public B, public C{};

int main()
{
    cout << "sizeof(A): " << sizeof A <<endl; // 1，空对象，只有一个占位
    cout << "sizeof(B): " << sizeof B <<endl; // 4，一个bptr指针，省去占位,不需要对齐
    cout << "sizeof(C): " << sizeof C <<endl; // 4，一个bptr指针，省去占位,不需要对齐
    cout << "sizeof(D): " << sizeof D <<endl; // 8，两个bptr，省去占位,不需要对齐
}
```

上述代码所体现的关系是，B和C虚拟继承A，D又公有继承B和C，这种方式是一种**菱形继承或者钻石继承**，可以用如下图来表示





虚拟继承的情况下，无论基类被继承多少次，只会存在一个实体。虚拟继承基类的子类中，子类会增加某种形式的指针，或者指向虚基类子对象，或者指向一个相关的表格；表格中存放的不是虚基类子对象的地址，就是其偏移量，此类指针被称为bptr，如上图所示。如果既存在vptr又存在bptr，某些编译器会将其优化，合并为一个指针

98、如何获得结构成员相对于结构开头的字节偏移量

使用**offsetof()**函数

举个例子：

```
#include <iostream>
#include <stddef.h>
using namespace std;

struct  S
{
    int x;
    char y;
    int z;
    double a;
};
```

```
int main()
{
    cout << offsetof(S, x) << endl; // 0
    cout << offsetof(S, y) << endl; // 4
    cout << offsetof(S, z) << endl; // 8
    cout << offsetof(S, a) << endl; // 12
    return 0;
}

在VS2019 + win下 并不是这样的

    cout << offsetof(S, x) << endl; // 0
    cout << offsetof(S, y) << endl; // 4
    cout << offsetof(S, z) << endl; // 8
    cout << offsetof(S, a) << endl; // 16 这里是 16的位置，因为 double是8字节，需要找一个8的
当然了，如果加上  #pragma pack(4)指定 4字节对齐就可以了
#pragma pack(4)
struct  S
{
    int x;
    char y;
    int z;
    double a;
};
void test02()
{

    cout << offsetof(S, x) << endl; // 0
    cout << offsetof(S, y) << endl; // 4
    cout << offsetof(S, z) << endl; // 8
    cout << offsetof(S, a) << endl; // 12
    }
```

S结构体中各个数据成员的内存空间划分如下所示，需要注意内存对齐

99、静态类型和动态类型以及静态绑定和动态绑定的总结

- 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；

从上面的定义也可以看出，非虚函数一般都是静态绑定，而虚函数都是动态绑定（如此才可实现多态性）。
举个例子：

```
#include <iostream>
using namespace std;

class A
{
public:
    /*virtual*/ void func() { std::cout << "A::func()\n"; }
};
class B : public A
{
public:
    void func() { std::cout << "B::func()\n"; }
};
class C : public A
{
public:
    void func() { std::cout << "C::func()\n"; }
};
int main()
{
    C* pc = new C(); //pc的静态类型是它声明的类型C*，动态类型也是C*；
    B* pb = new B(); //pb的静态类型和动态类型也都是B*；
    A* pa = pc;      //pa的静态类型是它声明的类型A*，动态类型是pa所指向的对象pc的类型C*；
    pa = pb;         //pa的动态类型可以更改，现在它的动态类型是B*，但其静态类型仍是声明时候的A
    C *pnull = NULL; //pnull的静态类型是它声明的类型C*,没有动态类型，因为它指向了NULL；

    pa->func();      //A::func() pa的静态类型永远都是A*，不管其指向的是哪个子类，都是直接调用
    pc->func();      //C::func() pc的动、静态类型都是C*，因此调用C::func()；
    pnull->func();   //C::func() 不用奇怪为什么空指针也可以调用函数，因为这在编译期就确定了，
    return 0;
}
```

如果将A类中的virtual注释去掉，则运行结果是：

```
pa->func();      //B::func() 因为有了virtual虚函数特性，pa的动态类型指向B*，因此先在B中查找，
pc->func();      //C::func() pc的动、静态类型都是C*，因此也是先在C中查找；
pnull->func();   //空指针异常，因为是func是virtual函数，因此对func的调用只能等到运行期才能确
```

在上面的例子中，

- 如果基类A中的func不是virtual函数，那么不论pa、pb、pc指向哪个子类对象，对func的调用都是在定义pa、pb、pc时的静态类型决定，早已在编译期确定了。

- 同样的空指针也能够直接调用no-virtual函数而不报错（这也说明一定要做空指针检查啊！），因此静态绑定不能实现多态；

- 如果func是虚函数，那所有的调用都要等到运行时根据其指向对象的类型才能确定，比起静态绑定自然是要有性能损失的，但是却能实现多态特性；

  本文代码里都是针对指针的情况来分析的，对于引用的情况也同样适用。

至此总结一下静态绑定和动态绑定的区别：

- 静态绑定发生在编译期，动态绑定发生在运行期；

- 对象的动态类型可以更改，但是静态类型无法更改；

- 要想实现动态，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；

  建议：

绝对不要重新定义继承而来的非虚(non-virtual)函数（《Effective C++ 第三版》条款36），因为这样导致函数调用由对象声明时的静态类型确定了，而和对象本身脱离了关系，没有多态，也这将给程序留下不可预知的隐患和莫名其妙的BUG；另外，在动态绑定也即在virtual函数中，要注意默认参数的使用。当缺省参数和virtual函数一起使用的时候一定要谨慎，不然出了问题怕是很难排查。
看下面的代码：

```
#include <iostream>
using namespace std;

class E
{
public:
    virtual void func(int i = 0)
    {
        std::cout << "E::func()\t" << i << "\n";
    }
};
class F : public E
{
public:
    virtual void func(int i = 1)
    {
```

```
        std::cout << "F::func()\t" << i << "\n";
    }
};

void test2()
{
    F* pf = new F();
    E* pe = pf;
    pf->func(); //F::func() 1  正常，就该如此；
    pe->func(); //F::func() 0  哇哦，这是什么情况，调用了子类的函数，却使用了基类中参数的默认
}
int main()
{
    test2();
    return 0;
}
```

> 《C++中的静态绑定和动态绑定》：https://www.cnblogs.com/lizhenghn/p/3657717.html

100、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
- ...

101、引用是否能实现动态绑定，为什么可以实现？

可以。

引用在创建的时候必须初始化，在访问虚函数时，编译器会根据其所绑定的对象类型决定要调用哪个函数。注意只能调用虚函数。

举个例子：

```
#include <iostream>
using namespace std;

class Base
{
public:
    virtual void  fun()
    {
        cout << "base :: fun()" << endl;
    }
};

class Son : public Base
{
public:
    virtual void  fun()
    {
        cout << "son :: fun()" << endl;
    }
    void func()
    {
        cout << "son :: not virtual function" <<endl;
    }
};

int main()
{
    Son s;
    Base& b = s; // 基类类型引用绑定已经存在的Son对象，引用必须初始化
    s.fun(); //son::fun()
    b.fun(); //son :: fun()
    return 0;
}
```

需要说明的是虚函数才具有动态绑定，上面代码中，Son类中还有一个非虚函数func()，这在b对象中是无法调用的，如果使用基类指针来指向子类也是一样的。

102、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 。

> 《C++经典面试题》：https://www.cnblogs.com/yjd_hycf_space/p/7495640.html

103、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，因此在涉及到指针加减运算需要十分小心，加多或者减多都会导致指针指向一块未知的内存地址，如果再进行操作就会很危险。

举个例子：

```
#include <iostream>
using namespace std;

int main()
{
    int *a, *b, c;
    a = (int*)0x500;
    b = (int*)0x520;
    c = b - a;
    printf("%d\n", c); // 8
    a += 0x020;
    c = b - a;
    printf("%d\n", c); // -24
    return 0;
}
```

首先变量a和b都是以16进制的形式初始化，将它们转成10进制分别是1280（5*16\^2=1280）和1312（5*16\^2+2*16=1312），那么它们的差值为32，也就是说a和b所指向的地址之间间隔32个位，但是考虑到是int类型占4位，所以c的值为32/4=8

a自增16进制0x20之后，其实际地址变为1280 + 2*16*4 = 1408，（因为一个int占4位，所以要乘4），这样它们的差值就变成了1312 - 1280 = -96，所以c的值就变成了-96/4 = -24

遇到指针的计算，**需要明确的是指针每移动一位，它实际跨越的内存间隔是指针类型的长度，建议都转成10进制计算，计算结果除以类型长度取得结果**

104、 怎样判断两个浮点数是否相等？

对两个浮点数判断大小和是否相等不能直接用==来判断，会出错！明明相等的两个数比较反而是不相等！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。

105、方法调用的原理（栈、汇编）

1） 机器用栈来传递过程参数、存储返回信息、保存寄存器用于以后恢复，以及本地存储。而为单个过程分配的那部分栈称为帧栈；帧栈可以认为是程序栈的一段，它有两个端点，一个标识起始地址，一个标识着结束地址，两个指针结束地址指针esp，开始地址指针ebp;

2） 由一系列栈帧构成，这些栈帧对应一个过程，而且每一个栈指针+4的位置存储函数返回地址；每一个栈帧都建立在调用者的下方，当被调用者执行完毕时，这一段栈帧会被释放。由于栈帧是向地址递减的方向延伸，因此如果我们将栈指针减去一定的值，就相当于给栈帧分配了一定空间的内存。如果将栈指针加上一定的值，也就是向上移动，那么就相当于压缩了栈帧的长度，也就是说内存被释放了。

3） 过程实现

① 备份原来的帧指针，调整当前的栈帧指针到栈指针位置;

② 建立起来的栈帧就是为被调用者准备的，当被调用者使用栈帧时，需要给临时变量分配预留内存;

③ 使用建立好的栈帧，比如读取和写入，一般使用mov，push以及pop指令等等。

④ 恢复被调用者寄存器当中的值，这一过程其实是从栈帧中将备份的值再恢复到寄存器，不过此时这些值可能已经不在栈顶了

⑤ 恢复被调用者寄存器当中的值，这一过程其实是从栈帧中将备份的值再恢复到寄存器，不过此时这些值可能已经不在栈顶了。

⑥ 释放被调用者的栈帧，释放就意味着将栈指针加大，而具体的做法一般是直接将栈指针指向帧指针，因此会采用类似下面的汇编代码处理。

⑦ 恢复调用者的栈帧，恢复其实就是调整栈帧两端，使得当前栈帧的区域又回到了原始的位置。

⑧ 弹出返回地址，跳出当前过程，继续执行调用者的代码。

4） 过程调用和返回指令

① call指令

② leave指令

③ ret指令

106、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。

值传递过程中，被调函数的形式参数作为被调函数的局部变量处理，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，从而形成了实参的一个副本（替身）。

值传递的特点是，被调函数对形式参数的任何操作都是作为局部变量进行的，不会影响主调函数的实参变量的值（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（本体）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（根据别名找到主调函数中的本体）。

因此，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。

3）引用传递和指针传递是不同的，虽然他们都是在被调函数栈空间上的一个局部变量，但是任何对于引用参数的处理都会通过一个间接寻址的方式操作到主调函数中的相关变量。

而对于指针传递的参数，如果改变被调函数中的指针地址，它将应用不到主调函数的相关变量。如果想通过指针参数传递来改变主调函数中的相关变量（地址），那就得使用指向指针的指针或者指针引用。

4）从编译的角度来讲，程序在编译时分别将指针和引用添加到符号表上，符号表中记录的是变量名及变量所对应地址。

指针变量在符号表上对应的地址值为指针变量的地址值，而引用在符号表上对应的地址值为引用对象的地址值（与实参名字不同，地址相同）。

符号表生成之后就不会再改，因此指针可以改变其指向的对象（指针变量中的值可以改），而引用对象则不能修改。

107、类如何实现只能静态分配和只能动态分配

1） 前者是把new、delete运算符重载为private属性。后者是把构造、析构函数设为protected属性，再用子类来动态创建

2） 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；

3） 只有使用new运算符，对象才会被建立在堆上，因此只要限制new运算符就可以实现类对象只能建立在栈上，可以将new运算符设为私有。

108、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，为了使用这些成员，派生类必须知道他们是什么。

结语

好家伙，终于完了。你要是能看到这里，那真是个狠人。

- EOF -

推荐阅读 点击标题可跳转

1、C++模版的本质

2、使用C++20实现轻量级AOP库

3、C++内存管理全景指南

点赞和在看就是最大的支持♡

喜欢此内容的人还喜欢

面试没过，被现代C++问懵了！

C语言与CPP编程

双非渣硕的秋招之路总结（已拿抖音研发岗SP）

拓跋阿秀

【嵌入式C】放弃printf,选择了精简snprintf

最后一个bug