C++

1. 引用和指针的区别?

1) 指针是一个实体，需要分配内存空间。引用只是变量的别名，不需要分配内存空间。

2) 引用在定义的时候必须进行初始化，并且不能够改变。指针在定义的时候不一定要初始化，并 0 且指向的空间可变。（注：不能有引用的值不能为 NULL）

3) 有多级指针，但是没有多级引用，只能有一级引用。

4) 指针和引用的自增运算结果不一样。（指针是指向下一个空间，引用时引用的变量值加 1）

5) sizeof 引用得到的是所指向的变量（对象）的大小，而 sizeof 指针得到的是指针本身的大小。

6) 引用访问一个变量是直接访问，而指针访问一个变量是间接访问。

7) 使用指针前最好做类型检查，防止野指针的出现；

8) 引用底层是通过指针实现的；

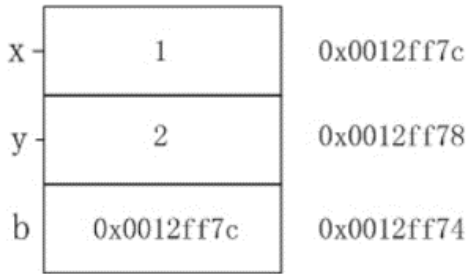
9) 作为参数时也不同，传指针的实质是传值，传递的值是指针的地址；传引用的实质是传地址，传递的是变量的地址。

2. 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。引用，虽然在 C++中底层处理时和指针处理方式相同，不过在用到引用变量的地方，系统会自动对其进行解引用，这一步骤系统默认进行，所以我们找不到引用自身开辟的内存单元，从这里看，引用好像没有开辟自身内存，只是给引用对象起了一个别名。



3. C++中的指针参数传递和引用参数传递

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。

4. 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。

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。

5. static 的用法和作用?

https://blog.csdn.net/wangbeibei23/article/details/89343705###

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。

最后对 static 的上述三条作用做一句话总结。首先 static 的最主要功能是隐藏，其次因为 static 变量存放在静态存储区，所以它具备持久性和默认值 0。

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。(静态数据成员是类的成员，而不是对象的成员)

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；非静态成员函数可以访问静态数据成员

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

6. 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。

7. const?

1) 阻止一个变量被改变，可以使用 const 关键字；
2) const 对象必须初始化
   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；
3) 默认情况下，const 对象只在文件内有效。（如何使用其他文件中的 const？应该在需要被其他文件使用的 const 对象定义成这样：extern const int a = 9;而在别的需要使用这个 const 对象的地方声明：extern const int a;前者是定义，后者是声明。这样就可以使用了。）
4) const 的引用

引用前面带 const 修饰符	被引用前面带 const 修饰符	附注
是	是	对 const 变量的 const 引用

是	否	对 const 的引用可能是引用一个并非 const 的对象。引用认为它引用的变量不可以改变，但实际可以
否	是	试图让一个非常量引用指向一个常量对象。不合法
否	否	普通的引用

5) const 与指针

指针	被指数据	附注
变量	变量	普通指针
变量	常量	不可以
常量	变量	可以。常量指针“自以为是”，认为自己指向了常量，所以自觉的不去改变所指对象的值。实际上，这个值是可以改变的
常量	常量	可以

6) 顶层 const 和底层 const

该对象是 const，说明其为顶层 const；该对象指向或者引用的变量时 const，说明其为底层 const。

7) C++常量折叠

常量折叠表面上的效果和宏替换是一样的，只是，“效果上是一样的”。

区别：①、宏是字符常量，在预编译完宏替换完成后，该宏名字会消失，所有对宏如 PI 的引用已经全部被替换为它所对应的值，编译器当然没有必要再维护这个符号。②而常量折叠发生的情况是，对常量的引用全部替换为该常量如 r 的值，但是，常量名 r 并不会消失，编译器会把他放入到符号表中，同时，会为该变量分配空间，栈空间或者全局空间。

8) const 成员函数

①const 修饰的成员函数不能修改任何的成员变量(mutable 修饰的变量除外)②const 成员函数不能调用非 const 成员函数，因为非 const 成员函数可以会修改成员变量。

const 成员变量

①在类中声明变量为 const 类型，但是不可以初始化；② const 常量的初始化必须在构造函数初始化列表中初始化，而不可以在构造函数函数体内初始化此时的 const 变量属于具体的一个对象。

9) 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。

10) 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；

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。

8. const 成员函数的理解和应用?

① const Stock & Stock::topval (②const Stock & s) ③const

① 处 const：确保返回的 Stock 对象在以后的使用中不能被修改

② 处 const：确保此方法不修改传递的参数 S

③ 处 const：保证此方法不修改调用它的对象，const 对象只能调用 const 成员函数,不能调用非 const 成员函数

9. 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。

4) int const *const p;

10. mutable

1) 如果需要在 const 成员方法中修改一个成员变量的值，那么需要将这个成员变量修饰为 mutable。即用 mutable 修饰的成员变量不受 const 成员方法的限制;

2) 可以认为 mutable 的变量是类的辅助状态，但是只是起到类的一些方面表述的功能，修改他的内容我们可以认为对象的状态本身并没有改变的。实际上由于 const_cast 的存在，这个概念很多时候用处不是很到了。

11. extern 用法?

1) extern 修饰变量的声明

如果文件 a.c 需要引用 b.c 中变量 int v，就可以在 a.c 中声明 extern int v，然后就可以引用变量 v。

2) extern 修饰函数的声明

如果文件 a.c 需要引用 b.c 中的函数，比如在 b.c 中原型是 int fun(int mu)，那么就可以在 a.c 中声明 extern int fun（int mu），然后就能使用 fun 来做任何事情。就像变量的声明一样，extern int fun（int mu）可以放在 a.c 中任何地方，而不一定非要放在 a.c 的文件作用域的范围中。

3) extern 修饰符可用于指示 C 或者 C + + 函数的调用规范。

比如在 C + + 中调用 C 库函数，就需要在 C + + 程序中用 extern “C”声明要引用的函数。这是给链接器用的，告诉链接器在链接的时候用 C 函数规范来链接。主要原因是 C + + 和 C 程序编译完成后在目标代码中命名规则不同。

12. int 转字符串，字符串转 int? strcat, strcpy, strncpy, memset, memcpy 的内部实现?

c++11 标准增加了全局函数 std::to_string

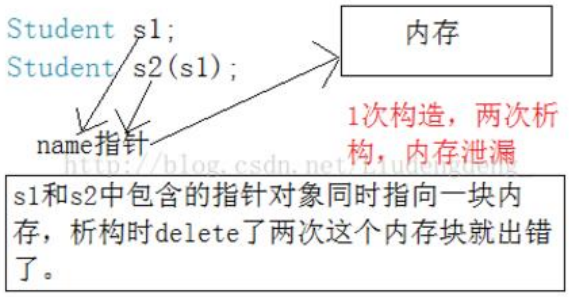
可以使用 std::stoi/stol/stoll 等等函数

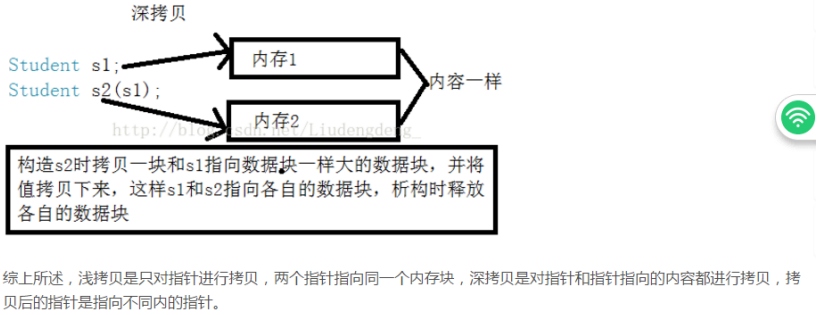
strcpy 拥有返回值，有时候函数原本不需要返回值，但为了增加灵活性如支持链式表达，

13. 深拷贝与浅拷贝?

1) 浅拷贝——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。

深拷贝——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。





综上所述，浅拷贝是只对指针进行拷贝，两个指针指向同一个内存块，深拷贝是对指针和指针指向的内容都进行拷贝，拷贝后的指针是指向不同内的指针。

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。

14. C++模板是什么，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。

15. C 语言 struct 和 C++struct 区别

1) C 语言中：struct 是用户自定义数据类型（UDT）；C++中 struct 是抽象数据类型（ADT），支持成员函数的定义，（C++中的 struct 能继承，能实现多态）。
2) C 中 struct 是没有权限的设置的，且 struct 中只能是一些变量的集合体，可以封装数据却不可以隐藏数据，而且成员不可以是函数。
3) C++中，struct 的成员默认访问说明符为 public（为了与 C 兼容），class 中的默认访问限定符为 private，struct 增加了访问权限，且可以和类一样有成员函数。

4) struct 作为类的一种特例是用来自定义数据结构的。一个结构标记声明后，在 C 中必须在结构标记前加上 struct，才能做结构类型名

16. 虚函数可以声明为 inline 吗?

1) 虚函数用于实现运行时的多态，或者称为晚绑定或动态绑定。而内联函数用于提高效率。内联函数的原理是，在编译期间，对调用内联函数的地方的代码替换成函数代码。内联函数对于程序中需要频繁使用和调用的小函数非常有用。
2) 虚函数要求在运行时进行类型确定，而内联函数要求在编译期完成相关的函数替换
3) #define 和 inline 的区别：①define，定义预编译时处理的宏，只进行简单的字符替换，无类型检测；②typedef，定义类型别名，用于处理复杂类型；③inline，内联函数对编译器提出建议，是否进行宏替换，由编译器决定，编译器有权拒绝。

17. 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。

18. 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；

19. 构造函数为什么不能为虚函数？析构函数为什么要虚函数?

总结：对象未实例化，无 vtable；构造函数只执行一次，无动态行为；虚函数是用于信息不全的情况下，而构造函数是必要要明确指定对象类型。

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. 构造函数不需要是虚函数，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。

20. 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。

21. 构造函数和析构函数可以调用虚函数吗，为什么

1．从语法上讲，调用完全没有问题。

2．但是从效果上看，往往不能达到需要的目的。

Effective 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。

22. 构造函数的执行顺序？析构函数的执行顺序？构造函数内部干了啥？拷贝构造干了啥？

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。

23. 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) （析构函数可以是纯虚的，但纯虚析构函数必须有定义体，因为析构函数的调用是在子类中隐含的）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。因此，最好不要把虚析构函数定义为纯虚析构函数。

24. 构造函数析构函数可以调用虚函数吗?

1) 在构造函数和析构函数中最好不要调用虚函数;
2) 构造函数或者析构函数调用虚函数并不会发挥虚函数动态绑定的特性，跟普通函数没区别;
3) 即使构造函数或者析构函数如果能成功调用虚函数， 程序的运行结果也是不可控的。

25. 构造函数和析构函数可否抛出异常

1) 答案：只有构造函数可以
2) ①不要在析构函数中抛出异常！虽然 C++并不禁止析构函数抛出异常，但这样会导致程序过早结束或出现不明确的行为。②如果某个操作可能会抛出异常，class 应提供一个普通函数（而非析构函数），来执行该操作。目的是给客户一个处理错误的机会。③如果析构函数中异常非抛不可，那就用 try catch 来将异常吞下，但这样方法并不好，我们提倡有错早些报出来。
3) ①构造函数中抛出异常，会导致析构函数不能被调用，但对象本身已申请到的内存资源会被系统释放（已申请到资源的内部成员变量会被系统依次逆序调用其析构函数）。②因为析构函数不能被调用，所以可能会造成内存泄露或系统资源未被释放。③构造函数中可以抛出异常，但必须保证在构造函数抛出异常之前，把系统资源释放掉，防止内存泄露。（如何保证？？？使用 auto_ptr）。

26. 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。
2) 后者有两种方式：①把**析构函数设置为 private**，因为编译器在为类对象分配栈空间时，会先检查类的析构函数的访问性。如果析构函数在类外部无法访问，则编译器拒绝在栈空间上为类对象分配内存。但是**必须提供一个 destroy 函数来实现内存空间释放。缺点：**无法实现继承②将**析构函数设置为 protected**，类外无法访问 protected 成员，但子类可以访问。③**不能将构造函数设为 private。**原因：new 运算符分为两步，C++提供 new 运算符重载，其实只允许 operator new()函数，而 operator()函数用于内存分配，无法提供构造函数功能，因此不可行

3) 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；

4) 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。

27. 如果想将某个类用作基类，为什么该类必须定义而非声明?

1) 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。

28. 什么情况会自动生成默认构造函数?

1) 带有默认构造函数的类成员对象，如果一个类没有任何构造函数，但它含有一个成员对象，而后者有默认构造函数，那么编译器就为该类合成出一个默认构造函数。不过这个合成操作只有在构造函数真正被需要的时候才会发生；如果一个类 A 含有多个成员类对象的话，那么类 A 的每一个构造函数必须调用每一个成员对象的默认构造函数而且必须按照类对象在类 A 中的声明顺序进行；

2) 带有默认构造函数的基类，如果一个没有任务构造函数的派生类派生自一个带有默认构造函数基类，那么该派生类会合成一个构造函数调用上一层基类的默认构造函数；

3) 带有一个虚函数的类

4) 带有一个虚基类的类

5) 合成的默认构造函数中，只有基类子对象和成员类对象会被初始化。所有其他的非静态数据成员都不会被初始化。

29. 什么是类的继承?

1) 类的继承的重点：代码复用

2) 3 种继承方式下基类成员在派生类中的访问属性

基类私有成员	继承方式	基类成员在派生类中的访问属性
Private	Public	不可直接访问
Private	Protected	不可直接访问
Private	Private	不可直接访问
Protected	Public	Protected

Protected	Protected	Protected
Protected	Private	Private
Public	Public	Public
Public	Protected	Protected
Public	Private	Private

3) 类成员访问属性及作用

访问属性	作用
Private	只允许类的成员函数及友元函数访问，不能被其他函数访问
Protected	既允许该类的成员函数及友元函数访问，也允许其他派生类的成员函数访问
Public	既允许该类的成员函数访问，也允许类外部的其他函数访问

4) 类与类之间的关系

has-A 包含、组合关系，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；如：。如：车包含方向盘、轮胎、发动机

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、属于关系，关系具有传递性；

5) 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；

6) 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；

7) 继承中的访问控制

public、protected、private

8) 继承中的构造和析构函数

9) 继承中的兼容性原则

30. 什么是组合?

1) 一个类里面的数据成员是另一个类的对象，即内嵌其他类的对象作为自己的成员；创建组合类的对象：首先创建各个内嵌对象，难点在于构造函数的设计。创建对象时既要对基本类型的成员进行初始化，又要对内嵌对象进行初始化。

2) 创建组合类对象，构造函数的执行顺序：先调用内嵌对象的构造函数，然后按照内嵌对象成员在组合类中的定义顺序，与组合类构造函数的初始化列表顺序无关。然后执行组合类构造函数的函数体，析构函数调用顺序相反。

31. 抽象基类为什么不能创建对象？

抽象类是一种特殊的类，它是为了抽象和设计的目的为建立的，它处于继承层次结构的较上层。

（1） 抽象基类的定义：称带有纯虚函数的类为抽象基类。

（2） 抽象类的作用：

抽象类的主要作用是将有关的操作作为结果接口组织在一个继承层次结构中，由它来为派生类提供一个公共的根，派生类将具体实现在其基类中作为接口的操作。所以派生类实际上刻画了一组子类的操作接口的通用语义，这些语义也传给子类，子类可以具体实现这些语义，也可以再将这些语义传给自己的子类。

（3）使用抽象类时注意：

抽象类只能作为基类来使用，其纯虚函数的实现由派生类给出。如果派生类中没有重新定义纯虚函数，而只是继承基类的纯虚函数，则这个派生类仍然还是一个抽象类。如果派生类中给出了基类纯虚函数的实现，则该派生类就不再是抽象类了，它是一个可以建立对象的具体的类。

抽象类是不能定义对象的。一个纯虚函数不需要（但是可以）被定义。

一、纯虚函数定义

纯虚函数是一种特殊的虚函数，它的一般格式如下：

```
class <类名>
{
virtual <类型><函数名>(<参数表>)=0;
…
};
```

在许多情况下，在基类中不能对虚函数给出有意义的实现，而把它声明为纯虚函数，它的实现留给该基类的派生类去做。这就是纯虚函数的作用。

纯虚函数可以让类先具有一个操作名称，而没有操作内容，让派生类在继承时再去具体地给出定义。凡是含有纯虚函数的类叫做抽象类。这种类不能声明对象，只是作为基类为派生类服务。除非在派生类中完全实现基类中所有的的纯虚函数，否则，派生类也变成了抽象类，不能实例化对象。

二、纯虚函数引入原因

1、为了方便使用多态特性，我们常常需要在基类中定义虚拟函数。

2、在很多情况下，基类本身生成对象是不合情理的。例如，动物作为一个基类可以派生出老虎、孔 雀等子类，但动物本身生成对象明显不合常理。

为了解决上述问题，引入了纯虚函数的概念，将函数定义为纯虚函数（方法：virtual ReturnType Function()= 0;）。若要使派生类为非抽象类，则编译器要求在派生类中，必须对纯虚函数予以重载以实现多态性。同时含有纯虚函数的类称为抽象类，它不能生成对象。这样就很好地解决了上述两个问题。

例如，绘画程序中，shape 作为一个基类可以派生出圆形、矩形、正方形、梯形等，如果我要求面积总和的话，那么会可以使用一个 shape * 的数组，只要依次调用派生类的 area()函数了。如果不用接口就没法定义成数组，因为既可以是 circle ,也可以是 square ,而且以后还可能加上 rectangle，等等.

三、相似概念

1、多态性

指相同对象收到不同消息或不同对象收到相同消息时产生不同的实现动作。C++支持两种多态性：编译时多态性，运行时多态性。

a.编译时多态性：通过重载函数实现

b.运行时多态性：通过虚函数实现。

2、虚函数

虚函数是在基类中被声明为 virtual，并在派生类中重新定义的成员函数，可实现成员函数的动态重载。

3、抽象类

包含纯虚函数的类称为抽象类。由于抽象类包含了没有定义的纯虚函数，所以不能定义抽象类的对象。

32. 虚基类、虚函数和抽象基类

①虚函数是在基类中被声明为 virtual，并在派生类中重新定义的成员函数，可实现成员函数的动态重载

②包含纯虚函数的类称为抽象基类

③虚继承和虚函数是完全无相关的两个概念。虚继承是解决 C++多重继承问题的一种手段　存在的问题：从不同途径继承来的同一基类，会在子类中存在多份拷贝。这将存在两个问题：其一，浪费存储空间；第二，存在二义性问题；解决办法：虚继承底层实现原理与编译器相关，一般通过虚基类指针和虚基类表实现，每个虚继承的子类都有一个虚基类指针（占用一个指针的存储空间，4 字节）和虚基类表（不占用类对象的存储空间）（需要强调的是，虚基类依旧会在子类里面存在拷贝，只是仅仅最多存在一份

而已，并不是不在子类里面了)；当虚继承的子类被当做父类继承时，虚基类指针也会被继承。

④当在多条继承路径上有一个公共的基类,在这些路径中的某几条汇合处，这个公共的基类就会产生多个实例(或多个副本)，若只想保存这个基类的一个实例，可以将这个公共基类说明为虚基类

⑤与虚函数的联系和区别：都利用了虚指针（均占用类的存储空间）和虚表（均不占用类的存储空间）； 虚基类依旧存在继承类中，只占用存储空间；虚函数不占用存储空间； 虚基类表存储的是虚基类相对直接继承类的偏移；而虚函数表存储的是虚函数地址。

33. 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。

34. 为什么友元函数必须在类内部声明?

1) 为什么要引入友元函数：为了使其他类的成员函数直接访问该类的私有变量
2) 引入友元函数的优点：能够减小系统开销，提高效率，表达简单、清晰
3) 引入友元函数的缺点：友元函数破坏了封装机制，尽量不使用友元函数，除非不得已的情况下才使用友元函数
4) 使用友元函数的场景：①运算符重载的某些场合；②两个类要共享数据的时候
5) 因为编译器必须能够读取这个结构的声明以理解这个数据类型的大小、行为等方面的所有规则。有一条规则在任何关系中都很重要，那就是谁可以访问我的私有部分。

35. 介绍一下 C++里面的多态?

（1）静态多态（函数重载，模板（泛型编程））

是在编译的时候，就确定调用函数的类型。

（2）动态多态（覆盖，虚函数实现）

在运行的时候，才确定调用的是哪个函数，动态绑定。运行基类指针指向派生类的对象，并调用派生类的函数。

虚函数实现原理：虚函数表和虚函数指针。

纯虚函数： virtual int fun() = 0;

函数的运行版本由实参决定，在运行时选择函数的版本，所以动态绑定又称为运行时绑定。当编译器遇到一个模板定义时，它并不生成代码。只有当实例化出模板的一个特定版本时，编译器才会生成代码。

36. 用 C 语言实现 C++的继承

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

//C++中的继承与多态
struct A
{
    virtual void fun()      //C++中的多态:通过虚函数实现
    {
        cout<<"A:fun()"<<endl;
    }

    int a;
};
struct B:public A           //C++中的继承:B 类公有继承 A 类
{
    virtual void fun()      //C++中的多态:通过虚函数实现（子类的关键字 virtual 可加可不加）
    {
        cout<<"B:fun()"<<endl;
    }

    int b;
};

//C 语言模拟 C++的继承与多态

typedef void (*FUN)();          //定义一个函数指针来实现对成员函数的继承

struct _A           //父类
```

```
{
    FUN _fun;    //由于 C 语言中结构体不能包含函数，故只能用函数指针在外面实现
    int _a;
};

struct _B            //子类
{
    _A _a_;        //在子类中定义一个基类的对象即可实现对父类的继承
    int _b;
};

void _fA()          //父类的同名函数
{
    printf("_A:_fun()\n");
}
void _fB()          //子类的同名函数
{
    printf("_B:_fun()\n");
}

void Test()
{
    //测试 C++中的继承与多态
    A a;      //定义一个父类对象 a
    B b;      //定义一个子类对象 b

    A* p1 = &a;     //定义一个父类指针指向父类的对象
    p1->fun();      //调用父类的同名函数
    p1 = &b;         //让父类指针指向子类的对象
    p1->fun();      //调用子类的同名函数

    //C 语言模拟继承与多态的测试
    _A _a;      //定义一个父类对象_a
```

```
    _B _b;      //定义一个子类对象_b
    _a._fun = _fA;          //父类的对象调用父类的同名函数
    _b._a_._fun = _fB;      //子类的对象调用子类的同名函数

    _A* p2 = &_a;    //定义一个父类指针指向父类的对象
    p2->_fun();      //调用父类的同名函数
    p2 = (_A*)&_b;   //让父类指针指向子类的对象,由于类型不匹配所以要进行强转
    p2->_fun();      //调用子类的同名函数
}
```

37. 继承机制中对象之间如何转换？指针和引用之间如何转换？

1) 向上类型转换

   将派生类指针或引用转换为基类的指针或引用被称为向上类型转换，向上类型转换会自动进行，而且向上类型转换是安全的。

2) 向下类型转换

   将基类指针或引用转换为派生类指针或引用被称为向下类型转换，向下类型转换不会自动进行，因为一个基类对应几个派生类，所以向下类型转换时不知道对应哪个派生类，所以在向下类型转换时必须加动态类型识别技术。RTTI 技术，用 dynamic_cast 进行向下类型转换。

38. 组合与继承优缺点？

一：继承

继承是 Is a 的关系，比如说 Student 继承 Person,则说明 Student is a Person。继承的优点是子类可以重写父类的方法来方便地实现对父类的扩展。

继承的缺点有以下几点：

①：父类的内部细节对子类是可见的。

②：子类从父类继承的方法在编译时就确定下来了，所以无法在运行期间改变从父类继承的方法的行为。

③：如果对父类的方法做了修改的话（比如增加了一个参数），则子类的方法必须做出相应的修改。所以说子类与父类是一种高耦合，违背了面向对象思想。

二：组合

组合也就是设计类的时候把要组合的类的对象加入到该类中作为自己的成员变量。

组合的优点：

①：当前对象只能通过所包含的那个对象去调用其方法，所以所包含的对象的内部细节对当前对象时不可见的。

②：当前对象与包含的对象是一个低耦合关系，如果修改包含对象的类中代码不需要修改当前对象类的代码。

③：当前对象可以在运行时动态的绑定所包含的对象。可以通过 set 方法给所包含对象赋值。

组合的缺点：①：容易产生过多的对象。②：为了能组合多个对象，必须仔细对接口进行定义。

39. 左值右值

1) 在 C++11 中所有的值必属于左值、右值两者之一，右值又可以细分为纯右值、将亡值。在 C++11 中可以取地址的、有名字的就是左值，反之，不能取地址的、没有名字的就是右值（将亡值或纯右值）。举个例子，int a = b+c, a 就是左值，其有变量名为 a，通过&a 可以获取该变量的地址；表达式 b+c、函数 int func()的返回值是右值，在其被赋值给某一变量前，我们不能通过变量名找到它，＆(b+c)这样的操作则不会通过编译。

2) C++11 对 C++98 中的右值进行了扩充。在 C++11 中右值又分为纯右值（prvalue，Pure Rvalue）和将亡值（xvalue，eXpiring Value）。其中纯右值的概念等同于我们在 C++98 标准中右值的概念，指的是临时变量和不跟对象关联的字面量值；将亡值则是 C++11 新增的跟右值引用相关的表达式，这样表达式通常是将要被移动的对象（移为他用），比如返回右值引用 T&&的函数返回值、std::move 的返回值，或者转换为 T&&的类型转换函数的返回值。将亡值可以理解为通过“盗取”其他变量内存空间的方式获取到的值。在确保其他变量不再被使用、或即将被销毁时，通过“盗取”的方式可以避免内存空间的释放和分配，能够延长变量值的生命期。

3) 左值引用就是对一个左值进行引用的类型。右值引用就是对一个右值进行引用的类型，事实上，由于右值通常不具有名字，我们也只能通过引用的方式找到它的存在。右值引用和左值引用都是属于引用类型。无论是声明一个左值引用还是右值引用，都必须立即

进行初始化。而其原因可以理解为是引用类型本身自己并不拥有所绑定对象的内存，只是该对象的一个别名。左值引用是具名变量值的别名，而右值引用则是不具名（匿名）变量的别名。左值引用通常也不能绑定到右值，但常量左值引用是个“万能”的引用类型。它可以接受非常量左值、常量左值、右值对其进行初始化。不过常量左值所引用的右值在它的“余生”中只能是只读的。相对地，非常量左值只能接受非常量左值对其进行初始化。

4) 右值值引用通常不能绑定到任何的左值，要想绑定一个左值到右值引用，通常需要std::move()将左值强制转换为右值。

40. 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。

41. C 语言的编译链接过程？

源代码 - - >预处理 - - >编译 - - >优化 - - >汇编 - - >链接-->可执行文件

1) 预处理

读取 c 源程序，对其中的伪指令（以#开头的指令）和特殊符号进行处理。包括宏定义替换、条件编译指令、头文件包含指令、特殊符号。预编译程序所完成的基本上是对源程序的“替代”工作。经过此种替代，生成一个没有宏定义、没有条件编译指令、没有特殊符号的输出文件。.i 预处理后的 c 文件，.ii 预处理后的 C++文件。

2) 编译阶段

编译程序所要作得工作就是通过词法分析和语法分析，在确认所有的指令都符合语法规则之后，将其翻译成等价的中间代码表示或汇编代码。.s 文件

3) 汇编过程

汇编过程实际上指把汇编语言代码翻译成目标机器指令的过程。对于被翻译系统处理的每一个 C 语言源程序，都将最终经过这一处理而得到相应的目标文件。目标文件中所存放的也就是与源程序等效的目标的机器语言代码。.o 目标文件

4) 链接阶段

链接程序的主要工作就是将有关的目标文件彼此相连接，也即将在一个文件中引用的符号同该符号在另外一个文件中的定义连接起来，使得所有的这些目标文件成为一个能够诶操作系统装入执行的统一整体。

42. vector 与 list 的区别与应用？怎么找某 vector 或者 list 的倒数第二个元素

1) vector 数据结构

vector 和数组类似，拥有一段连续的内存空间，并且起始地址不变。因此能高效的进行随机读取，时间复杂度为 o(1);但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)。另外，当数组中内存空间不够时，会重新申请一块内存空间并进行内存拷贝。连续存储结构：vector 是可以实现动态增长的对象数组，支持对数组高效率的访问和在数组尾端的删除和插入操作，在中间和头部删除和插入相对不易，需要挪动大量的数据。它与数组最大的区别就是 vector 不需程序员自己去考虑容量问题，库里面本身已经实现了容量的动态增长，而数组需要程序员手动写入扩容函数进形扩容。

2) list 数据结构

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 的随机读取非常没有效率，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。非连续存储结构：list 是一个双链表结构，支持对链表的双向遍历。每个节点包括三个信息：元素本身，指向前一个元素的节点（prev）和指向下一个元素的节点（next）。因此 list 可以高效率的对数据元素任意位置进行访问和插入删除等操作。由于涉及对额外指针的维护，所以开销比较大。

区别：

vector 的随机访问效率高，但在插入和删除时（不包括尾部）需要挪动数据，不易操作。list 的访问要遍历整个链表，它的随机访问效率低。但对数据的插入和删除操作等都比较方便，改变指针的指向即可。list 是单向的，vector 是双向的。vector 中的迭代器在使用后就失效了，而 list 的迭代器在使用之后还可以继续使用。

3) int mySize = vec.size();vec.at(mySize -2);

list 不提供随机访问，所以不能用下标直接访问到某个位置的元素，要访问 list 里的元素只能遍历，不过你要是只需要访问 list 的最后 N 个元素的话，可以用反向迭代器来遍历：

43. STL vector 的实现，删除其中的元素，迭代器如何变化？为什么是两倍扩容？释放空间？

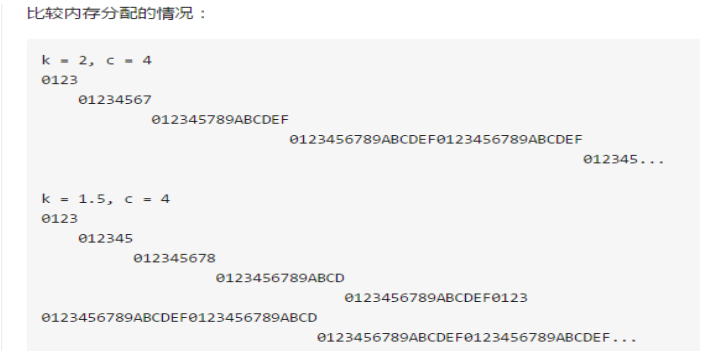
size()函数返回的是已用空间大小，capacity()返回的是总空间大小，capacity()-size()则是剩余的可用空间大小。当 size()和 capacity()相等，说明 vector 目前的空间已被用完，如果再添加新元素，则会引起 vector 空间的动态增长。

由于动态增长会引起重新分配内存空间、拷贝原空间、释放原空间，这些过程会降低程序效率。因此，可以使用 reserve(n)预先分配一块较大的指定大小的内存空间，这样当指定大小的内存空间未使用完时，是不会重新分配内存空间的，这样便提升了效率。只有当 n>capacity()时，调用 reserve(n)才会改变 vector 容量。

resize()成员函数只改变元素的数目，不改变 vector 的容量。

1. 空的 vector 对象，size()和 capacity()都为 0
2. 当空间大小不足时，新分配的空间大小为原空间大小的 2 倍。
3. 使用 reserve()预先分配一块内存后，在空间未满的情况下，不会引起重新分配，从而提升了效率。
4. 当 reserve()分配的空间比原空间小时，是不会引起重新分配的。
5. resize()函数只改变容器的元素数目，未改变容器大小。
6. 用 reserve(size_type)只是扩大 capacity 值，这些内存空间可能还是“野”的，如果此时使用“[ ]”来访问，则可能会越界。而 resize(size_type new_size)会真正使容器具有 new_size 个对象。

1. 不同的编译器，vector 有不同的扩容大小。在 vs 下是 1.5 倍，在 GCC 下是 2 倍；
2. 空间和时间的权衡。简单来说， 空间分配的多，平摊时间复杂度低，但浪费空间也多。
3. 使用 k=2 增长因子的问题在于，每次扩展的新尺寸必然刚好大于之前分配的总和，也就是说，之前分配的内存空间不可能被使用。这样对内存不友好。最好把增长因子设为(1,2)



4. 对比可以发现采用采用成倍方式扩容，可以保证常数的时间复杂度，而增加指定大小的容量只能达到 O(n)的时间复杂度，因此，使用成倍的方式扩容。

如何释放空间：

由于 vector 的内存占用空间只增不减，比如你首先分配了 10,000 个字节，然后 erase 掉后面 9,999 个，留下一个有效元素，但是内存占用仍为 10,000 个。所有内存空间是在 vector 析构时候才能被系统回收。empty()用来检测容器是否为空的，clear()可以清空所有元素。但是即使 clear()，vector 所占用的内存空间依然如故，无法保证内存的回收。

如果需要空间动态缩小，可以考虑使用 deque。如果 vector，可以用 swap()来帮助你释放内存。

vector(Vec).swap(Vec);

将 Vec 的内存空洞清除；

vector().swap(Vec);

清空 Vec 的内存；

44. 容器内部删除一个元素

1) 顺序容器

erase 迭代器不仅使所指向被删除的迭代器失效，而且使被删元素之后的所有迭代器失效(list 除外)，所以不能使用 erase(it++)的方式，但是 erase 的返回值是下一个有效迭代器；

It = c.erase(it);

2) 关联容器

erase 迭代器只是被删除元素的迭代器失效，但是返回值是 void，所以要采用 erase(it++)的方式删除迭代器；

c.erase(it++)

45. STL 迭代器如何实现

1. 迭代器是一种抽象的设计理念，通过迭代器可以在不了解容器内部原理的情况下遍历容器，除此之外，STL 中迭代器一个最重要的作用就是作为容器与 STL 算法的粘合剂。
2. 迭代器的作用就是提供一个遍历容器内部所有元素的接口，因此迭代器内部必须保存一个与容器相关联的指针，然后重载各种运算操作来遍历，其中最重要的是*运算符与->运算符，以及++、--等可能需要重载的运算符重载。这和 C++中的智能指针很像，智能指针也是将一个指针封装，然后通过引用计数或是其他方法完成自动释放内存的功能。
3. 最常用的迭代器的相应型别有五种：value type、difference type、pointer、reference、iterator catagoly;

46. set 与 hash_set 的区别

1. set 底层是以 RB-Tree 实现，hash_set 底层是以 hash_table 实现的；
2. RB-Tree 有自动排序功能，而 hash_table 不具有自动排序功能；
3. set 和 hash_set 元素的键值就是实值；
4. hash_table 有一些无法处理的型别；
5. set 是 RB_Tree， unordered_set 是 hash_table

47. map 与 hashmap 的区别

1. STL 的 map 底层是用红黑树实现的，查找时间复杂度是 log(n)；STL 的 hash_map 底层是用 hash 表存储的，查询时间复杂度是 O(1)；map 具有自动排序的功能；
2. hash_map 不具有自动排序的功能；
3. hashtable 有一些无法处理的型别；

使用时 map 的 key 需要定义 operator<。而 unordered_map 需要定义 hash_value 函数并且重载 operator==。但是很多系统内置的数据类型都自带这些

48. map、set 是怎么实现的，红黑树是怎么能够同时实现这两种容器？ 为什么使用红黑树？

1) 他们的底层都是以红黑树的结构实现，因此插入删除等操作都在 O(logn)时间内完成，因此可以完成高效的插入删除；
2) 在这里我们定义了一个模版参数，如果它是 key 那么它就是 set，如果它是 map，那么它就是 map；底层是红黑树，实现 map 的红黑树的节点数据类型是 key+value，

而实现 set 的节点数据类型是 value

3) 因为 map 和 set 要求是自动排序的，红黑树能够实现这一功能，而且时间复杂度比较低。

49. 如何在共享内存上使用 stl 标准库?

1) 想像一下把 STL 容器，例如 map, vector, list 等等，放入共享内存中，IPC 一旦有了这些强大的通用数据结构做辅助，无疑进程间通信的能力一下子强大了很多。我们没必要再为共享内存设计其他额外的数据结构，另外，STL 的高度可扩展性将为 IPC 所驱使。STL 容器被良好的封装，默认情况下有它们自己的内存管理方案。当一个元素被插入到一个 STL 列表(list)中时，列表容器自动为其分配内存，保存数据。考虑到要将 STL 容器放到共享内存中，而容器却自己在堆上分配内存。一个最笨拙的办法是在堆上构造 STL 容器，然后把容器复制到共享内存，并且确保所有容器的内部分配的内存指向共享内存中的相应区域，这基本是个不可能完成的任务。

2) 假设进程 A 在共享内存中放入了数个容器，进程 B 如何找到这些容器呢？一个方法就是进程 A 把容器放在共享内存中的确定地址上（fixed offsets），则进程 B 可以从该已知地址上获取容器。另外一个改进点的办法是，进程 A 先在共享内存某块确定地址上放置一个 map 容器，然后进程 A 再创建其他容器，然后给其取个名字和地址一并保存到这个 map 容器里。进程 B 知道如何获取该保存了地址映射的 map 容器，然后同样再根据名字取得其他容器的地址。

50. map 插入方式有几种?

1) 用 insert 函数插入 pair 数据，

```
mapStudent.insert(pair<int, string>(1, "student_one"));
```

2) 用 insert 函数插入 value_type 数据

```
mapStudent.insert(map<int, string>::value_type(1,"student_one"));
```

3) 在 insert 函数中使用 make_pair()函数

```
mapStudent.insert(make_pair(1,"student_one"));
```

4) 用数组方式插入数据

```
mapStudent[1] = "student_one";
```

51. STL 中 unordered_map(hash_map)和 map 的区别，hash_map 如何解决冲突以及扩容

1) unordered_map 和 map 类似，都是存储的 key-value 的值，可以通过 key 快速索引到 value。不同的是 unordered_map 不会根据 key 的大小进行排序，

2) 存储时是根据 key 的 hash 值判断元素是否相同，即 unordered_map 内部元素是无序的，而 map 中的元素是按照二叉搜索树存储，进行中序遍历会得到有序遍历。

3) 所以使用时 map 的 key 需要定义 operator<。而 unordered_map 需要定义 hash_value 函数并且重载 operator==。但是很多系统内置的数据类型都自带这些，

4) 那么如果是自定义类型，那么就需要自己重载 operator<或者 hash_value()了。

5) 如果需要内部元素自动排序，使用 map，不需要排序使用 unordered_map

6) unordered_map 的底层实现是 hash_table;

7) hash_map 底层使用的是 hash_table，而 hash_table 使用的开链法进行冲突避免，所有 hash_map 采用开链法进行冲突解决。

8) **什么时候扩容：**当向容器添加元素的时候，会判断当前容器的元素个数，如果大于等于阈值---即当前数组的长度乘以加载因子的值的时候，就要自动扩容啦。

9) **扩容(resize)**就是重新计算容量，向 HashMap 对象里不停的添加元素，而 HashMap 对象内部的数组无法装载更多的元素时，对象就需要扩大数组的长度，以便能装入更多的元素。

52. vector 越界访问下标，map 越界访问下标？vector 删除元素时会不会释放空间?

1) 通过下标访问 vector 中的元素时不会做边界检查，即便下标越界。也就是说，下标与 first 迭代器相加的结果超过了 finish 迭代器的位置，程序也不会报错，而是返回这个地址中存储的值。如果想在访问 vector 中的元素时首先进行边界检查，可以使用 vector 中的 at 函数。通过使用 at 函数不但可以通过下标访问 vector 中的元素，而且在 at 函数内部会对下标进行边界检查。

2) map 的下标运算符[]的作用是：将 key 作为下标去执行查找，并返回相应的值；如果不存在这个 key，就将一个具有该 key 和 value 的某人值插入这个 map。

3) erase()函数，只能删除内容，不能改变容量大小; erase 成员函数，它删除了 itVect 迭代器指向的元素，并且返回要被删除的 itVect 之后的迭代器，迭代器相当于一个智能指针;clear()函数，只能清空内容，不能改变容量大小;如果要想在删除内容的同时释放内存，那么你可以选择 deque 容器。

53. map[]与 find 的区别?

1) map 的下标运算符[]的作用是：将关键码作为下标去执行查找，并返回对应的值；如果不存在这个关键码，就将一个具有该关键码和值类型的默认值的项插入这个 map。
2) map 的 find 函数：用关键码执行查找，找到了返回该位置的迭代器；如果不存在这个关键码，就返回尾迭代器。

54. STL 中 list 与 queue 之间的区别

1) list 不再能够像 vector 一样以普通指针作为迭代器，因为其节点不保证在存储空间中连续存在；
2) list 插入操作和结合才做都不会造成原有的 list 迭代器失效;
3) list 不仅是一个双向链表，而且还是一个环状双向链表，所以它只需要一个指针；
4) list 不像 vector 那样有可能在空间不足时做重新配置、数据移动的操作，所以插入前的所有迭代器在插入操作之后都仍然有效；
5) deque 是一种双向开口的连续线性空间，所谓双向开口，意思是可以在头尾两端分别做元素的插入和删除操作；可以在头尾两端分别做元素的插入和删除操作；
6) deque 和 vector 最大的差异，一在于 deque 允许常数时间内对起头端进行元素的插入或移除操作，二在于 deque 没有所谓容量概念，因为它是动态地以分段连续空间组合而成，随时可以增加一段新的空间并链接起来，deque 没有所谓的空间保留功能。

55. STL 中的 allocator,deallocator

1) 第一级配置器直接使用 malloc()、free()和 relloc()，第二级配置器视情况采用不同的策略：当配置区块超过 128bytes 时，视之为足够大，便调用第一级配置器；当配置器区块小于 128bytes 时，为了降低额外负担，使用复杂的内存池整理方式，而不再用一级配置器；
2) 第二级配置器主动将任何小额区块的内存需求量上调至 8 的倍数，并维护 16 个 free-list，各自管理大小为 8~128bytes 的小额区块；
3) 空间配置函数 allocate()，首先判断区块大小，大于 128 就直接调用第一级配置器，小于 128 时就检查对应的 free-list。如果 free-list 之内有可用区块，就直接拿来用，

如果没有可用区块，就将区块大小调整至 8 的倍数，然后调用 refill()，为 free-list 重新分配空间；

4) 空间释放函数 deallocate()，该函数首先判断区块大小，大于 128bytes 时，直接调用一级配置器，小于 128bytes 就找到对应的 free-list 然后释放内存。

56. STL 中 hash_map 扩容发生什么?

1) hash table 表格内的元素称为桶（bucket),而由桶所链接的元素称为节点（node),其中存入桶元素的容器为 stl 本身很重要的一种序列式容器——vector 容器。之所以选择 vector 为存放桶元素的基础容器，主要是因为 vector 容器本身具有动态扩容能力，无需人工干预。

2) 向前操作：首先尝试从目前所指的节点出发，前进一个位置（节点），由于节点被安置于 list 内，所以利用节点的 next 指针即可轻易完成前进操作，如果目前正巧是 list 的尾端，就跳至下一个 bucket 身上，那正是指向下一个 list 的头部节点。

57. map 如何创建?

1.vector 底层数据结构为数组 ，支持快速随机访问

2.list 底层数据结构为双向链表，支持快速增删

3.deque 底层数据结构为一个中央控制器和多个缓冲区，详细见 STL 源码剖析 P146，支持首尾（中间不能）快速增删，也支持随机访问

deque 是一个双端队列(double-ended queue)，也是在堆中保存内容的.它的保存形式如下:

[堆 1] --> [堆 2] -->[堆 3] --> ...

每个堆保存好几个元素,然后堆和堆之间有指针指向,看起来像是 list 和 vector 的结合品.

4.stack 底层一般用 list 或 deque 实现，封闭头部即可，不用 vector 的原因应该是容量大小有限制，扩容耗时

5.queue 底层一般用 list 或 deque 实现，封闭头部即可，不用 vector 的原因应该是容量大小有限制，扩容耗时（stack 和 queue 其实是适配器,而不叫容器，因为是对容器的再封装)

6.priority_queue 的底层数据结构一般为 vector 为底层容器，堆 heap 为处理规则来管理底层容器实现

7.set 底层数据结构为红黑树，有序，不重复

8.multiset 底层数据结构为红黑树，有序，可重复

9.map 底层数据结构为红黑树，有序，不重复

10.multimap 底层数据结构为红黑树，有序，可重复

11.hash_set 底层数据结构为 hash 表，无序，不重复

12.hash_multiset 底层数据结构为 hash 表，无序，可重复

13.hash_map 底层数据结构为 hash 表，无序，不重复

14.hash_multimap 底层数据结构为 hash 表，无序，可重复

58. vector 的增加删除都是怎么做的？为什么是 1.5 倍？

1) 新增元素：vector 通过一个连续的数组存放元素，如果集合已满，在新增数据的时候，就要分配一块更大的内存，将原来的数据复制过来，释放之前的内存，在插入新增的元素；

2) 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 ；

3) 初始时刻 vector 的 capacity 为 0，塞入第一个元素后 capacity 增加为 1；

4) 不同的编译器实现的扩容方式不一样，VS2015 中以 1.5 倍扩容，GCC 以 2 倍扩容。

对比可以发现采用采用成倍方式扩容，可以保证常数的时间复杂度，而增加指定大小的容量只能达到 O(n)的时间复杂度，因此，使用成倍的方式扩容。

1) 考虑可能产生的堆空间浪费，成倍增长倍数不能太大，使用较为广泛的扩容方式有两种，以 2 二倍的方式扩容，或者以 1.5 倍的方式扩容。

2) 以 2 倍的方式扩容，导存的总和，导致之前分配的内存不能再被使用，所以最好倍增长致下一次申请的内存必然大于之前分配内因子设置为(1,2)之间：

3) 向量容器 vector 的成员函数 pop_back()可以删除最后一个元素.

4) 而函数 erase()可以删除由一个 iterator 指出的元素，也可以删除一个指定范围的元素。

5) 还可以采用通用算法 remove()来删除 vector 容器中的元素.

6) 不同的是：采用 remove 一般情况下不会改变容器的大小，而 pop_back()与 erase()等成员函数会改变容器的大小。

59. 函数指针?

1) 什么是函数指针?

函数指针指向的是特殊的数据类型,函数的类型是由其返回的数据类型和其参数列表共同决定的，而函数的名称则不是其类型的一部分。

一个具体函数的名字，如果后面不跟调用符号(即括号)，则该名字就是该函数的指针(注意：大部分情况下，可以这么认为，但这种说法并不很严格)。

2) 函数指针的声明方法

int (*pf)(const int&, const int&); (1)

上面的 pf 就是一个函数指针，指向所有返回类型为 int，并带有两个 const int&参数的函数。注意*pf 两边的括号是必须的，否则上面的定义就变成了：

int *pf(const int&, const int&); (2)

而这声明了一个函数 pf，其返回类型为 int *， 带有两个 const int&参数。

3) 为什么有函数指针

函数与数据项相似，函数也有地址。我们希望在同一个函数中通过使用相同的形参在不同的时间使用产生不同的效果。

4) 一个函数名就是一个指针，它指向函数的代码。一个函数地址是该函数的进入点，也就是调用函数的地址。函数的调用可以通过函数名，也可以通过指向函数的指针来调用。函数指针还允许将函数作为变元传递给其他函数；

5) 两种方法赋值：

指针名 = 函数名； 指针名 = &函数名

60. 说说你对 c 和 c++的看法，c 和 c++的区别?

1) 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?

①从问题出发，自顶向下、逐步求精的开发思想称为“面向过程的设计思想”。优点：程序结构简单；分而治之，各个击破；自顶向下。缺点：数据和操作是分离的，不具

有封装性。②对象都由两部分组成 - 描述对象状态或属性的数据（变量）以及描述对象行为或者功能的方法（函数）。并且与面向过程不同，面向对象是将数据和操作数据的函数紧密结合，共同构成对象来更加精确地描述现实世界，这是面向过程和面向对象两者最本质的区别。主要特点是封装、继承和多态。封装即将数据和操作封装在一起，并避免了局部变量的暴露，而只提供接口；继承可以在原来的基础上很快的产生新的对象；多态是同一个方法调用，针对不同的对象有不同的反应，这方便了程序的设计。

2) 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）；

3) 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;

4) 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；

5) 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）；

6) 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；

7) 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）；

8) 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； “`

61. c/c++的内存分配，详细说一下栈、堆、静态存储区？

1、栈区（stack）— 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。

2、堆区（heap） — 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（操作系统）回收。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表。

3、全局区（静态区）（static）—，全局变量和静态变量的存储是放在一块的，初始化的全局变量和静态变量在一块区域，未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。程序结束后由系统释放。

4、文字常量区 —常量字符串就是放在这里的。程序结束后由系统释放。

5、程序代码区 —存放函数体的二进制代码。

62. 堆与栈的区别?

1) *管理方式：*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：*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）。当然，我们可以修改： 打开工程，依次操作菜单如下：Project->Setting->Link，在 Category 中选中 Output，然后在 Reserve 中设定堆栈的最大值和 commit。 注意：reserve 最小值为 4Byte；commit 是保留在虚拟内存的页文件里面，它设置的较大会使栈开辟较大的值，可能增加内存的开销和启动时间。
3) *碎片问题：*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：*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：*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，它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：*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。

63. 野指针是什么？如何检测内存泄漏？

1) 野指针：指向内存被释放的内存或者没有访问权限的内存的指针。

2) “野指针”的成因主要有 3 种：

① 指针变量没有被初始化。任何指针变量刚被创建时不会自动成为 NULL 指针，它的缺省值是随机的，它会乱指一气。所以，指针变量在创建的同时应当被初始化，要么将指针设置为 NULL，要么让它指向合法的内存。例如

```
char *p = NULL;
char *str = new char(100);
```

② 指针 p 被 free 或者 delete 之后，没有置为 NULL；

③ 指针操作超越了变量的作用范围。

3) 如何避免野指针：

① 对指针进行初始化

①将指针初始化为 NULL。

```
char *    p   = NULL;
```

②用 malloc 分配内存

```
char * p = (char * )malloc(sizeof(char));
```

③用已有合法的可访问的内存地址对指针初始化

```
char num[ 30] = {0};
char *p = num;
```

② 指针用完后释放内存，将指针赋 NULL。

```
delete(p);
p = NULL;
```

64. 悬空指针和野指针有什么区别?

1) 野指针：就是没有被初始化过的指针
2) 悬空指针：一个指针的指向对象已被删除，那么就成了悬空指针。

65. 内存泄漏

3) 内存泄漏

   内存泄漏是指由于疏忽或错误造成了程序未能释放掉不再使用的内存的情况。内存泄漏并非指内存在物理上消失，而是应用程序分配某段内存后，由于设计错误，失去了对该段内存的控制；
4) 后果

   只发生一次小的内存泄漏可能不被注意，但泄漏大量内存的程序将会出现各种证照：性能下降到内存逐渐用完，导致另一个程序失败；
5) 如何排除

   使用工具软件 BoundsChecker，BoundsChecker 是一个运行时错误检测工具，它主要定位程序运行时期发生的各种错误；

   调试运行 DEBUG 版程序，运用以下技术：CRT(C run-time libraries)、运行时函数调用堆栈、内存泄漏时提示的内存分配序号(集成开发环境 OUTPUT 窗口)，综合分析内存泄漏的原因，排除内存泄漏。
6) 解决方法

   智能指针。
7) 检查、定位内存泄漏

   检查方法：在 main 函数最后面一行，加上一句_CrtDumpMemoryLeaks()。调试程序，自然关闭程序让其退出，查看输出：

   输出这样的格式{453}normal block at 0x02432CA8,868 bytes long

   被{}包围的 453 就是我们需要的内存泄漏定位值，868 bytes long 就是说这个地方有 868 比特内存没有释放。

   定位代码位置

   在 main 函数第一行加上_CrtSetBreakAlloc(453);意思就是在申请 453 这块内存的位置中断。然后调试程序，程序中断了，查看调用堆栈。加上头文件#include <crtdbg.h>

66. 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。

67. delete p;与 delete[]p，allocator

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。

68. 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(对象指针)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。

4、 针对复杂类型，new[]出来的内存只能由 delete[]释放

69. 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。

70. 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。

71. malloc、realloc、calloc、alloc、alloca 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。

<1> alloca 向 **栈 ** 申请内存，无需释放

<2> malloc 向 **堆** 申请内存，没有初始化 `void* malloc(unsigned size) `

<3> calloc 向 **堆** 申请内存，初始化为 0 `void* calloc(size_t numElements, size_t sizeOfElement);`参数 sizeOfElement 为申请地址的单位元素长度,numElements 为元素个数，即在内存中申请 numElements*sizeOfElement 字节大小的连续地址空间.

<4> realloc 对 malloc 申请的内存进行大小调整 void* realloc(void* ptr, unsigned newsize); realloc 并不保证调整后的内存空间和原来的内存空间保持同一内存地址.相反，realloc 返回的指针很可能指向一个新的地址

72. brk、mmap 的区别

从操作系统角度来看，进程分配内存有两种方式，分别由两个系统调用完成：brk 和 mmap（不考虑共享内存）。

1、brk 是将数据段(.data)的最高地址指针_edata 往高地址推;

2、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。

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。

73. __stdcall 和__cdecl 的区别?

1) __stdcall

__stdcall 是函数恢复堆栈，只有在函数代码的结尾出现一次恢复堆栈的代码；在编译时就规定了参数个数，无法实现不定个数的参数调用；

2) __cdecl

__cdecl 是调用者恢复堆栈，假设有 100 个函数调用函数 a，那么内存中就有 100 端恢复堆栈的代码；可以不定参数个数；每一个调用它的函数都包含清空堆栈的代码，所以产生的可执行文件大小会比调用__stacall 函数大。

74. 手写实现智能指针类

1) 智能指针是一个数据类型，一般用模板实现，模拟指针行为的同时还提供自动垃圾回收机制。它会自动记录 SmartPointer<T*>对象的引用计数，一旦 T 类型对象的引用计数为 0，就释放该对象。除了指针对象外，我们还需要一个引用计数的指针设定对象的值，并将引用计数计为 1，需要一个构造函数。新增对象还需要一个构造函数，析构函数负责引用计数减少和释放内存。通过覆写赋值运算符，才能将一个旧的智能指针赋值给另一个指针，同时旧的引用计数减 1，新的引用计数加 1

2) 一个构造函数、拷贝构造函数、复制构造函数、析构函数、移走函数；

75. 使用智能指针管理内存资源，RAII

1) RAII 全称是“Resource Acquisition is Initialization”，直译过来是“资源获取即初始化”，也就是说在构造函数中申请分配资源，在析构函数中释放资源。因为 C++的语言机制保证了，当一个对象创建的时候，自动调用构造函数，当对象超出作用域的时候会自动调用析构函数。所以，在 RAII 的指导下，我们应该使用类来管理资源，将资源和对象的生命周期绑定。

2) 智能指针（std::shared_ptr 和 std::unique_ptr）即 RAII 最具代表的实现，使用智能指针，可以实现自动的内存管理，再也不需要担心忘记 delete 造成的内存泄漏。毫不夸张的来讲，有了智能指针，代码中几乎不需要再出现 delete 了。

76. 内存对齐？位域?

1、 分配内存的顺序是按照声明的顺序。

2、 每个变量相对于起始位置的偏移量必须是该变量类型大小的整数倍，不是整数倍空出内存，直到偏移量是整数倍为止。

3、 最后整个结构体的大小必须是里面变量类型最大值的整数倍。

添加了#pragma pack(n)后规则就变成了下面这样：

1、 偏移量要是 n 和当前变量大小中较小值的整数倍

2、 整体大小要是 n 和最大变量大小中较小值的整数倍

3、 n 值必须为 1,2,4,8…，为其他值时就按照默认的分配规则

77. 结构体变量比较是否相等

1) 重载了“==”操作符

```
struct foo {
    int a;
    int b;
    bool operator==(const foo& rhs) // 操作运算符重载
    {
        return( a == rhs.a) && (b == rhs.b);
    }
};
```

2) 元素的话，一个个比；

3) 指针直接比较，如果保存的是同一个实例地址，则(p1==p2)为真；

78. 位运算

若一个数 m 满足 m = 2^n;那么 k%m=k&(m-1)

79. 为什么内存对齐

1、平台原因(移植原因)

1) 不是所有的硬件平台都能访问任意地址上的任意数据的；

2) 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

2、性能原因：

1) 数据结构(尤其是栈)应该尽可能地在自然边界上对齐。

2) 原因在于，为了访问未对齐的内存，处理器需要作两次内存访问；而对齐的内存访问仅需要一次访问。

80. 函数调用过程栈的变化，返回值和参数变量哪个先入栈?

1、调用者函数把被调函数所需要的参数按照与被调函数的形参顺序相反的顺序压入栈中,即:从右向左依次把被调函数所需要的参数压入栈;

2、调用者函数使用 call 指令调用被调函数,并把 call 指令的下一条指令的地址当成返回地址压入栈中(这个压栈操作隐含在 call 指令中);

3、在被调函数中,被调函数会先保存调用者函数的栈底地址(push ebp),然后再保存调用者函数的栈顶地址,即:当前被调函数的栈底地址(mov ebp,esp);

4、在被调函数中,从 ebp 的位置处开始存放被调函数中的局部变量和临时变量,并且这些变量的地址按照定义时的顺序依次减小,即:这些变量的地址是按照栈的延伸方向排列的,先定义的变量先入栈,后定义的变量后入栈;

81. 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。

82. 宏定义一个取两个数中较大值的功能

```
#define MAX（x,y）((x>y?)x:y)
```

83. define、const、typedef、inline 使用方法?

一、 const **与**#define **的区别:**

1) const 定义的常量是变量带类型，而#define 定义的只是个常数不带类型;
2) define 只在预处理阶段起作用，简单的文本替换，而 const 在编译、链接过程中起作用;
3) define 只是简单的字符串替换没有类型检查。而 const 是有数据类型的，是要进行判断的，可以避免一些低级错误;
4) define 预处理后，占用代码段空间，const 占用数据段空间;
5) const 不能重定义，而 define 可以通过#undef 取消某个符号的定义，进行重定义;
6) define 独特功能，比如可以用来防止文件重复引用。

二、 #define **和别名** typedef **的区别**

1) 执行时间不同，typedef 在编译阶段有效，typedef 有类型检查的功能；#define 是宏定义，发生在预处理阶段，不进行类型检查;
2) 功能差异，typedef 用来定义类型的别名，定义与平台无关的数据类型，与 struct 的

结合使用等。#define 不只是可以为类型取别名，还可以定义常量、变量、编译开关等。

3) 作用域不同，#define 没有作用域的限制，只要是之前预定义过的宏，在以后的程序中都可以使用。而 typedef 有自己的作用域。

三、 define 与 inline 的区别

1) #define 是关键字，inline 是函数；
2) 宏定义在预处理阶段进行文本替换，inline 函数在编译阶段进行替换；
3) inline 函数有类型检查，相比宏定义比较安全；

84. printf 实现原理？

在 C/C++中，对函数参数的扫描是从后向前的。C/C++的函数参数是通过压入堆栈的方式来给函数传参数的（堆栈是一种先进后出的数据结构），最先压入的参数最后出来，在计算机的内存中，数据有 2 块，一块是堆，一块是栈（函数参数及局部变量在这里），而栈是从内存的高地址向低地址生长的，控制生长的就是堆栈指针了，最先压入的参数是在最上面，就是说在所有参数的最后面，最后压入的参数在最下面，结构上看起来是第一个，所以最后压入的参数总是能够被函数找到，因为它就在堆栈指针的上方。printf 的第一个被找到的参数就是那个字符指针，就是被双引号括起来的那一部分，函数通过判断字符串里控制参数的个数来判断参数个数及数据类型，通过这些就可算出数据需要的堆栈指针的偏移量了，下面给出 printf("%d,%d",a,b);（其中 a、b 都是 int 型的）的汇编代码.

85. #include 的顺序以及尖叫括号和双引号的区别

表示编译器只在系统默认目录或尖括号内的工作目录下搜索头文件，并不去用户的工作目录下寻找，所以一般尖括号用于包含标准库文件；

表示编译器先在用户的工作目录下搜索头文件，如果搜索不到则到系统默认目录下去寻找，所以双引号一般用于包含用户自己编写的头文件。

86. lambda 函数

1) 利用 lambda 表达式可以编写内嵌的匿名函数，用以替换独立函数或者函数对象；
2) 每当你定义一个 lambda 表达式后，编译器会自动生成一个匿名类（这个类当然重载了()运算符），我们称为闭包类型（closure type）。那么在运行时，这个 lambda 表达式就会返回一个匿名的闭包实例，其实一个右值。所以，我们上面的 lambda 表达式的结果就是一个个闭包。闭包的一个强大之处是其可以通过传值或者引用的方式捕

捉其封装作用域内的变量，前面的方括号就是用来定义捕捉模式以及变量，我们又将其称为 lambda 捕捉块。

3) lambda 表达式的语法定义如下：

```
[capture] （parameters） mutable ->return-type {statement};
```

4) lambda 必须使用尾置返回来指定返回类型，可以忽略参数列表和返回值，但必须永远包含捕获列表和函数体；

87. hello world 程序开始到打印到屏幕上的全过程?

1.用户告诉操作系统执行 HelloWorld 程序（通过键盘输入等）

2．操作系统：找到 helloworld 程序的相关信息，检查其类型是否是可执行文件；并通过程序首部信息，确定代码和数据在可执行文件中的位置并计算出对应的磁盘块地址。

3．操作系统：创建一个新进程，将 HelloWorld 可执行文件映射到该进程结构，表示由该进程执行 helloworld 程序。

4．操作系统：为 helloworld 程序设置 cpu 上下文环境，并跳到程序开始处。

5．执行 helloworld 程序的第一条指令，发生缺页异常

6．操作系统：分配一页物理内存，并将代码从磁盘读入内存，然后继续执行 helloworld 程序

7．helloword 程序执行 puts 函数（系统调用），在显示器上写一字符串

8．操作系统：找到要将字符串送往的显示设备，通常设备是由一个进程控制的，所以，操作系统将要写的字符串送给该进程

9．操作系统：控制设备的进程告诉设备的窗口系统，它要显示该字符串，窗口系统确定这是一个合法的操作，然后将字符串转换成像素，将像素写入设备的存储映像区

10．视频硬件将像素转换成显示器可接收和一组控制数据信号

11．显示器解释信号，激发液晶屏

12．OK，我们在屏幕上看到了 HelloWorld

88. 模板类和模板函数的区别是什么?

函数模板的实例化是由编译程序在处理函数调用时自动完成的,而类模板的实例化必须由程序员在程序中显式地指定。即函数模板允许隐式调用和显式调用而类模板只能显示调用。在使用时类模板必须加<T>，而函数模板不必

89. 为什么模板类一般都是放在一个 h 文件中

1) 模板定义很特殊。由 template<…>处理的任何东西都意味着编译器在当时不为它分配存储空间，它一直处于等待状态直到被一个模板实例告知。在编译器和连接器的某一处，有一机制能去掉指定模板的多重定义。所以为了容易使用，几乎总是在头文件中放置全部的模板声明和定义。
2) 在分离式编译的环境下,编译器编译某一个.cpp 文件时并不知道另一个.cpp 文件的存在，也不会去查找（当遇到未决符号时它会寄希望于连接器）。这种模式在没有模板的情况下运行良好,但遇到模板时就傻眼了,因为模板仅在需要的时候才会实例化出来，所以，当编译器只看到模板的声明时，它不能实例化该模板，只能创建一个具有外部连接的符号并期待连接器能够将符号的地址决议出来。然而当实现该模板的.cpp 文件中没有用到模板的实例时，编译器懒得去实例化，所以，整个工程的.obj 中就找不到一行模板实例的二进制代码，于是连接器也黔驴技穷了。

90. C++中类成员的访问权限和继承权限问题。

1) 三种访问权限

① public:用该关键字修饰的成员表示公有成员,该成员不仅可以在类内可以被 访问，在类外也是可以被访问的，是类对外提供的可访问接口;

② private:用该关键字修饰的成员表示私有成员，该成员仅在类内可以被访问，在类体外是隐藏状态;

③ protected:用该关键字修饰的成员表示保护成员，保护成员在类体外同样是隐藏状态，但是对于该类的派生类来说，相当于公有成员，在派生类中可以被访问。

2) 三种继承方式

① 若继承方式是 public，基类成员在派生类中的访问权限保持不变，也就是说，基类中的成员访问权限，在派生类中仍然保持原来的访问权限;

② 若继承方式是 private，基类所有成员在派生类中的访问权限都会变为私有(private)权限；

③ 若继承方式是 protected，基类的共有成员和保护成员在派生类中的访问权限都会变为保护(protected)权限，私有成员在派生类中的访问权限仍然是私有(private)权限。

91. cout 和 printf 有什么区别?

cout<<是一个函数，cout<<后可以跟不同的类型是因为 cout<<已存在针对各种类型数据的重载，所以会自动识别数据的类型。输出过程会首先将输出字符放入缓冲区，然后输出到屏幕。

cout 是有缓冲输出:

cout < < "abc " < <endl;

或 cout < < "abc\n ";cout < <flush; 这两个才是一样的.

endl 相当于输出回车后，再强迫缓冲输出。

flush 立即强迫缓冲输出。

printf 是无缓冲输出。有输出时立即输出

92. 重载运算符?

1、我们只能重载已有的运算符，而无权发明新的运算符；对于一个重载的运算符，其优先级和结合律与内置类型一致才可以；不能改变运算符操作数个数；
2、. :: ? : sizeof typeid **不能重载；
3、两种重载方式，成员运算符和非成员运算符，成员运算符比非成员运算符少一个参数；下标运算符、箭头运算符必须是成员运算符；
4、引入运算符重载，是为了实现类的多态性；
5、当重载的运算符是成员函数时，this 绑定到左侧运算符对象。成员运算符函数的参数数量比运算符对象的数量少一个；至少含有一个类类型的参数；
6、从参数的个数推断到底定义的是哪种运算符，当运算符既是一元运算符又是二元运算符（+，-，*，&）；
7、下标运算符必须是成员函数，下标运算符通常以所访问元素的引用作为返回值，同时最好定义下标运算符的常量版本和非常量版本；
8、箭头运算符必须是类的成员，解引用通常也是类的成员；重载的箭头运算符必须返回类的指针；

93. 函数重载函数匹配原则

1) 名字查找
2) 确定候选函数
3) 寻找最佳匹配

94. 定义和声明的区别

1. 如果是指变量的声明和定义

   从编译原理上来说，声明是仅仅告诉编译器，有个某类型的变量会被使用，但是编译器并不会为它分配任何内存。而定义就是分配了内存。
2. 如果是指函数的声明和定义

   声明：一般在头文件里，对编译器说：这里我有一个函数叫 function() 让编译器知道这个函数的存在。

   定义：一般在源文件里，具体就是函数的实现过程 写明函数体。

95. C++类型转换有四种

1) static_cast 能进行基础类型之间的转换，也是最常看到的类型转换。它主要有如下几种用法：

   1．用于类层次结构中父类和子类之间指针或引用的转换。进行上行转换（把子类的指针或引用转换成父类表示）是安全的；

   2．进行下行转换（把父类指针或引用转换成子类指针或引用）时，由于没有动态类型检查，所以是不安全的；

   3．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。

   4．把 void 指针转换成目标类型的指针（不安全!!）

   5．把任何类型的表达式转换成 void 类型。
2) const_cast 运算符用来修改类型的 const 或 volatile 属性。除了去掉 const 或 volatile 修饰之外， type_id 和 expression 得到的类型是一样的。但需要特别注意的是 const_cast 不是用于去除变量的常量性，而是去除指向常数对象的指针或引用的常量性，其去除常量性的对象必须为指针或引用。
3) reinterpret_cast 它可以把一个指针转换成一个整数，也可以把一个整数转换成一个指针（先把一个指针转换成一个整数，在把该整数转换成原类型的指针，还可以得到原先的指针值）。
4) dynamic_cast 主要用在继承体系中的安全向下转型。它能安全地将指向基类的指针

转型为指向子类的指针或引用，并获知转型动作成功是否。转型失败会返回 null（转型对象为指针时）或抛出异常 bad_cast（转型对象为引用时）。 dynamic_cast 会动用运行时信息（RTTI）来进行类型安全检查，因此 dynamic_cast 存在一定的效率损失。当使用 dynamic_cast 时，该类型必须含有虚函数，这是因为 dynamic_cast 使用了存储在 VTABLE 中的信息来判断实际的类型，RTTI 运行时类型识别用于判断类型。typeid 表达式的形式是 typeid(e)，typeid 操作的结果是一个常量对象的引用，该对象的类型是 type_info 或 type_info 的派生。

96. 全局变量和 static 变量的区别

1、全局变量（外部变量）的说明之前再冠以 static 就构成了静态的全局变量。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并无不同。这两者的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原文件组成时，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，在同一源程序的其它源文件中不能使用它。由于静态全局变量的作用域限于一个源文件内，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。static 全局变量与普通的全局变量的区别是 static 全局变量只初始化一次，防止在其他文件单元被引用。

2.static 函数与普通函数有什么区别?

static 函数与普通的函数作用域不同。尽在本文件中。只在当前源文件中使用的函数应该说明为内部函数（static），内部函数应该在当前源文件中说明和定义。对于可在当前源文件以外使用的函数应该在一个头文件中说明，要使用这些函数的源文件要包含这个头文件。

static 函数与普通函数最主要区别是 static 函数在内存中只有一份，普通静态函数在每个被调用中维持一份拷贝程序的局部变量存在于（堆栈）中，全局变量存在于（静态区）中，动态申请数据存在于（堆）

97. 静态成员与普通成员的区别

1) 生命周期

静态成员变量从类被加载开始到类被卸载，一直存在；

普通成员变量只有在类创建对象后才开始存在，对象结束，它的生命期结束；

2) 共享方式

静态成员变量是全类共享；普通成员变量是每个对象单独享用的；

3) 定义位置

普通成员变量存储在栈或堆中，而静态成员变量存储在静态全局区；

4) 初始化位置

普通成员变量在类中初始化；静态成员变量在类外初始化；

5) 默认实参

可以使用静态成员变量作为默认实参，

98. 说一下理解 ifdef endif

1) 一般情况下，源程序中所有的行都参加编译。但是有时希望对其中一部分内容只在满足一定条件才进行编译，也就是对一部分内容指定编译的条件，这就是“条件编译”。有时，希望当满足某条件时对一组语句进行编译，而当条件不满足时则编译另一组语句。

2) 条件编译命令最常见的形式为：

#ifdef 标识符
程序段 1
#else
程序段 2
#endif

它的作用是：当标识符已经被定义过(一般是用#define 命令定义)，则对程序段 1 进行编译，否则编译程序段 2。
其中#else 部分也可以没有，即：
#ifdef
程序段 1
#denif

3) 在一个大的软件工程里面，可能会有多个文件同时包含一个头文件，当这些文件编译链接成一个可执行文件上时，就会出现大量 “重定义” 错误。在头文件中使用#define、#ifndef、#ifdef、#endif 能避免头文件重定义。

99. 隐式转换，如何消除隐式转换？

1. C++的基本类型中并非完全的对立，部分数据类型之间是可以进行隐式转换的。所谓隐式转换，是指不需要用户干预，编译器私下进行的类型转换行为。很多时候用户可能都不知道进行了哪些转换

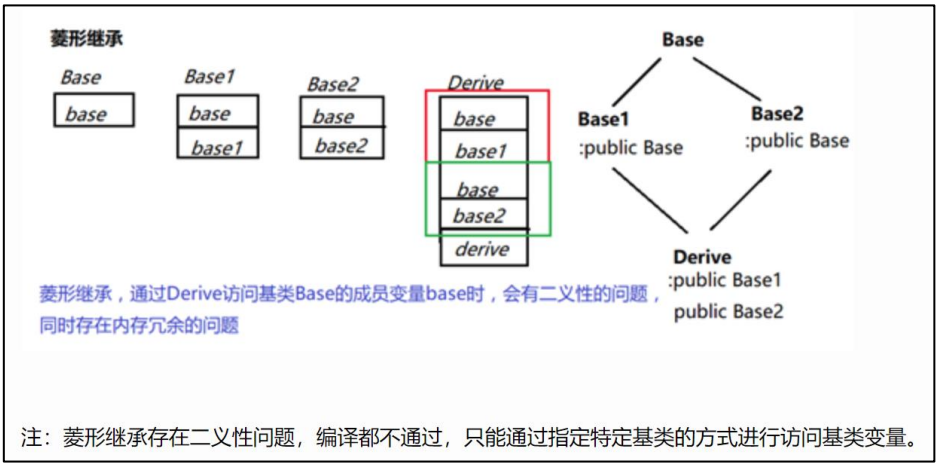
2. C++面向对象的多态特性，就是通过父类的类型实现对子类的封装。通过隐式转换，你可以直接将一个子类的对象使用父类的类型进行返回。在比如，数值和布尔类型的转换，整数和浮点数的转换等。某些方面来说，隐式转换给 C++程序开发者带来了不小的便捷。C++是一门强类型语言，类型的检查是非常严格的。
3. 基本数据类型，基本数据类型的转换以取值范围的作为转换基础（保证精度不丢失）。隐式转换发生在从小->大的转换中。比如从 char 转换为 int。从 int->long。自定义对象子类对象可以隐式的转换为父类对象。
4. C++中提供了 explicit 关键字，在构造函数声明的时候加上 explicit 关键字，能够禁止隐式转换。
5. 如果构造函数只接受一个参数，则它实际上定义了转换为此类类型的隐式转换机制。可以通过将构造函数声明为 explicit 加以制止隐式类型转换，关键字 explicit 只对一个实参的构造函数有效，需要多个实参的构造函数不能用于执行隐式转换，所以无需将这些构造函数指定为 explicit。

100. 虚函数的内存结构，那菱形继承的虚函数内存结构呢

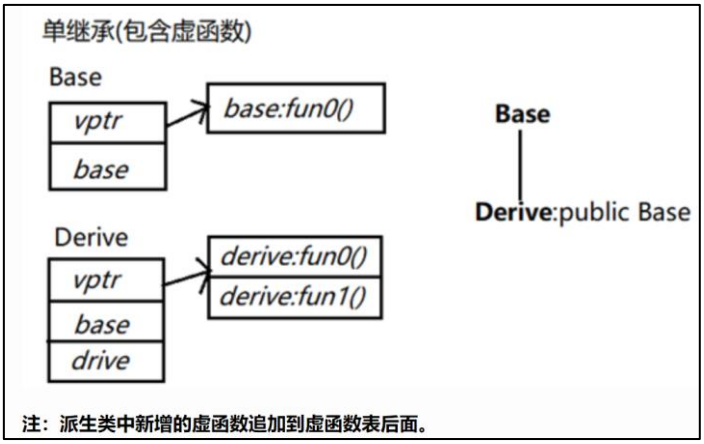
1. virtual table 存放着指针，这些指针指向该类每一个虚函数。虚表中的函数地址将按声明时的顺序排列。vtbl 在类声明后就形成了，vptr 是编译器生成的。

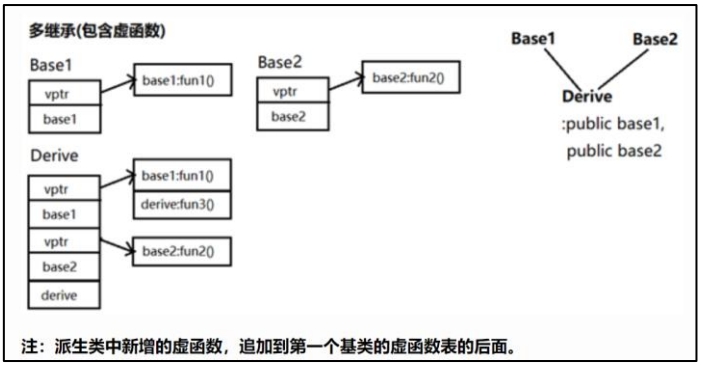
2. 虚函数表的位置在类对象的最前端

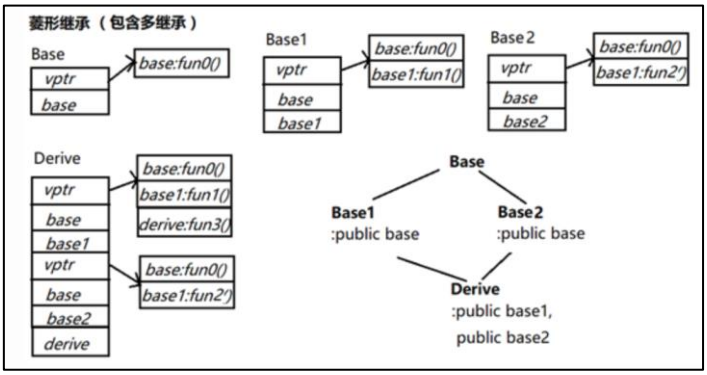
3. 普通继承（不包括虚函数）：菱形继承



4.普通继承（包括虚函数）

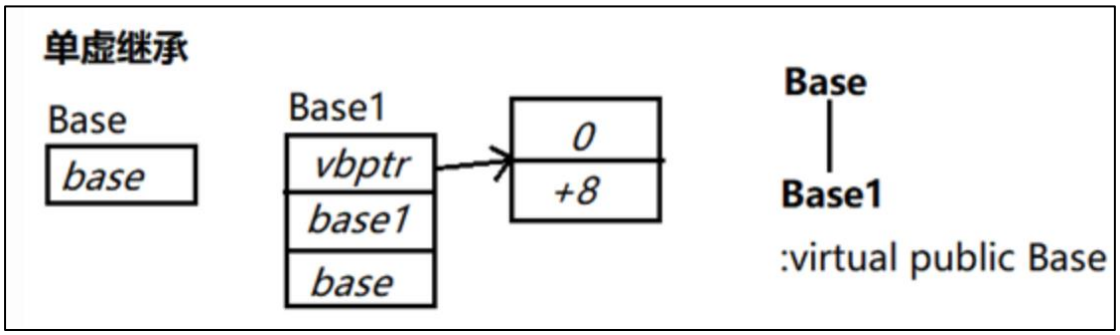




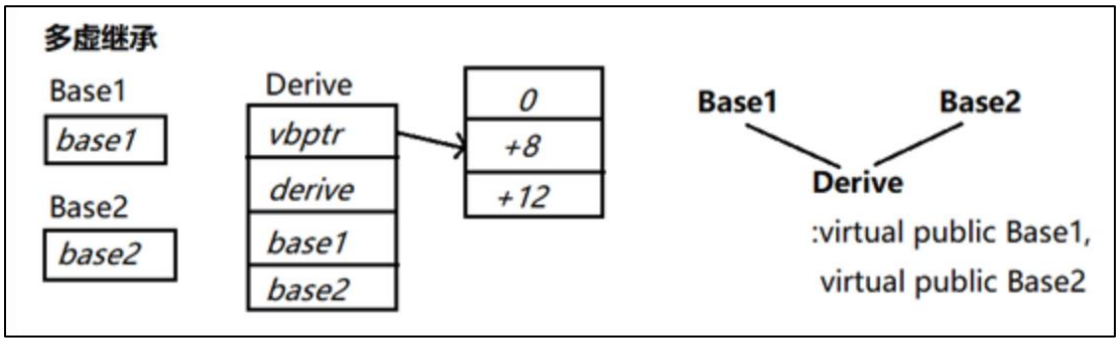


5.虚继承（不包含虚函数）

新增虚基类指针，指向虚基类表，虚基类表中首项存储虚基类指针的偏移量，接下来依次存储虚基类的偏移量（偏移量是相对于虚基类表指针的存储地址）



注：在上图中，base 中有一个 int，base1 中同样有一个 int。



101. 多继承的优缺点，作为一个开发者怎么看待多继承

1) C++允许为一个派生类指定多个基类，这样的继承结构被称做多重继承。
2) 多重继承的优点很明显，就是对象可以调用多个基类中的接口；
3) 如果派生类所继承的多个基类有相同的基类，而派生类对象需要调用这个祖先类的接口方法，就会容易出现二义性
4) 加上全局符确定调用哪一份拷贝。比如 pa.Author::eat()调用属于 Author 的拷贝。
5) 使用虚拟继承，使得多重继承类 Programmer_Author 只拥有 Person 类的一份拷贝。

102. 迭代器++it,it++哪个好，为什么

1) 前置返回一个引用，后置返回一个对象

```
// ++i 实现代码为:
int& operator++()
{
    *this += 1;
    return *this;
}
```

2) 前置不会产生临时对象，后置必须产生临时对象，临时对象会导致效率降低

```
//i++实现代码为:
int operator++(int)
{
int temp = *this;
    ++*this;
    return temp;
```

```
}
```

103. C++如何处理多个异常的?

1) **C++中的异常情况：**

   语法错误（编译错误）：比如变量未定义、括号不匹配、关键字拼写错误等等编译器在编译时能发现的错误，这类错误可以及时被编译器发现，而且可以及时知道出错的位置及原因，方便改正。

   运行时错误：比如数组下标越界、系统内存不足等等。这类错误不易被程序员发现，它能通过编译且能进入运行，但运行时会出错，导致程序崩溃。为了有效处理程序运行时错误，C++中引入异常处理机制来解决此问题。

2) **C++异常处理机制：**

   异常处理基本思想：执行一个函数的过程中发现异常，可以不用在本函数内立即进行处理， 而是抛出该异常，让函数的调用者直接或间接处理这个问题。

   C++异常处理机制由 3 个模块组成：try(检查)、throw(抛出)、catch(捕获)

   抛出异常的语句格式为：throw 表达式；如果 try 块中程序段发现了异常则抛出异常。

```
try
{
可能抛出异常的语句；（检查）
}
catch（类型名[形参名]）//捕获特定类型的异常
{
//处理 1；
}
catch（类型名[形参名]）//捕获特定类型的异常
{
//处理 2；
}
catch（…）//捕获所有类型的异常
{
}
```

104. 模板和实现可不可以不写在一个文件里面？为什么？

因为在编译时模板并不能生成真正的二进制代码，而是在编译调用模板类或函数的 CPP 文件时才会去找对应的模板声明和实现，在这种情况下编译器是不知道实现模板类或函数的 CPP 文件的存在，所以它只能找到模板类或函数的声明而找不到实现，而只好创建一个符号寄希望于链接程序找地址。但模板类或函数的实现并不能被编译成二进制代码，结果链接程序找不到地址只好报错了。

《C++编程思想》第 15 章(第 300 页)说明了原因：模板定义很特殊。由 template<…> 处理的任何东西都意味着编译器在当时不为它分配存储空间，它一直处于等待状态直到被一个模板实例告知。在编译器和连接器的某一处，有一机制能去掉指定模板的多重定义。所以为了容易使用，几乎总是在头文件中放置全部的模板声明和定义。

105. 在成员函数中调用 delete this 会出现什么问题？对象还可以使用吗？

1. 在类对象的内存空间中，只有数据成员和虚函数表指针，并不包含代码内容，类的成员函数单独放在代码段中。在调用成员函数时，隐含传递一个 this 指针，让成员函数知道当前是哪个对象在调用它。当调用 delete this 时，类对象的内存空间被释放。在 delete this 之后进行的其他任何函数调用，只要不涉及到 this 指针的内容，都能够正常运行。一旦涉及到 this 指针，如操作数据成员，调用虚函数等，就会出现不可预期的问题。

2. 为什么是不可预期的问题？

   delete this 之后不是释放了类对象的内存空间了么，那么这段内存应该已经还给系统，不再属于这个进程。照这个逻辑来看，应该发生指针错误，无访问权限之类的令系统崩溃的问题才对啊？这个问题牵涉到操作系统的内存管理策略。delete this 释放了类对象的内存空间，但是内存空间却并不是马上被回收到系统中，可能是缓冲或者其他什么原因，导致这段内存空间暂时并没有被系统收回。此时这段内存是可以访问的，你可以加上 100，加上 200，但是其中的值却是不确定的。当你获取数据成员，可能得到的是一串很长的未初始化的随机数；访问虚函数表，指针无效的可能性非常高，造成系统崩溃。

3. 如果在类的析构函数中调用 delete this，会发生什么？

   会导致堆栈溢出。原因很简单，delete 的本质是“先调用析构函数，再调用 operator delete”“为将被释放的内存调用一个或多个析构函数，然后，释放内存”。显然，

delete this 会去调用本对象的析构函数，而析构函数中又调用 delete this，形成无限递归，造成堆栈溢出，系统崩溃。

106. 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<memory>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_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
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.

107. 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<Type>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。

108. class、union、struct 的区别

1) C 语言中，struct 只是一个聚合数据类型，没有权限设置，无法添加成员函数，无法实现面向对象编程，且如果没有 typedef 结构名，声明结构变量必须添加关键字 struct。
2) C++中，struct 功能大大扩展，可以有权限设置（默认权限为 public），可以像 class 一样有成员函数，继承（默认 public 继承），可以实现面对对象编程，允许在声明结构变量时省略关键字 struct。
3) C 与 C++中的 union:一种数据格式，能够存储不同的数据类型，但只能同时存储其中的一种类型。C++ union 结构式一种特殊的类。它能够包含访问权限、成员变量、成员函数（可以包含构造函数和析构函数）。它不能包含虚函数和静态数据变量。它也不能被用作其他类的基类，它本身也不能有从某个基类派生而来。Union 中得默认访问权限是 public。union 类型是共享内存的，以 size 最大的结构作为自己的大小。每个数据成员在内存中的起始地址是相同的。

4) 在 C/C++程序的编写中，当多个基本数据类型或复合数据结构要占用同一片内存时，我们要使用联合体；当多种类型，多个对象，多个事物只取其一时（我们姑且通俗地称其为“n 选 1”），我们也可以使用联合体来发挥其长处。在某一时刻，一个 union 中只能有一个值是有效的。union 的一个用法就是可以用来测试 CPU 是大端模式还是小端模式：

109. 动态联编与静态联编

1) 在 C++中，联编是指一个计算机程序的不同部分彼此关联的过程。按照联编所进行的阶段不同，可以分为静态联编和动态联编；
2) 静态联编是指联编工作在编译阶段完成的，这种联编过程是在程序运行之前完成的，又称为早期联编。要实现静态联编，在编译阶段就必须确定程序中的操作调用（如函数调用）与执行该操作代码间的关系，确定这种关系称为束定，在编译时的束定称为静态束定。静态联编对函数的选择是基于指向对象的指针或者引用的类型。其优点是效率高，但灵活性差。
3) 动态联编是指联编在程序运行时动态地进行，根据当时的情况来确定调用哪个同名函数，实际上是在运行时虚函数的实现。这种联编又称为晚期联编，或动态束定。动态联编对成员函数的选择是基于对象的类型，针对不同的对象类型将做出不同的编译结果。C++中一般情况下的联编是静态联编，但是当涉及到多态性和虚函数时应该使用动态联编。动态联编的优点是灵活性强，但效率低。动态联编规定，只能通过指向基类的指针或基类对象的引用来调用虚函数，其格式为：指向基类的指针变量名->虚函数名（实参表）或基类对象的引用名.虚函数名（实参表）
4) 实现动态联编三个条件：
   必须把动态联编的行为定义为类的虚函数；
   类之间应满足子类型关系，通常表现为一个类从另一个类公有派生而来；
   必须先使用基类指针指向子类型的对象，然后直接或间接使用基类指针调用虚函数；

110. 动态编译与静态编译

1) 静态编译，编译器在编译可执行文件时，把需要用到的对应动态链接库中的部分提取出来，连接到可执行文件中去，使可执行文件在运行时不需要依赖于动态链接库；
2) 动态编译的可执行文件需要附带一个动态链接库，在执行时，需要调用其对应动态链接库的命令。所以其优点一方面是缩小了执行文件本身的体积，另一方面是加快了编译速度，节省了系统资源。缺点是哪怕是很简单的程序，只用到了链接库的一两条命令，也需要附带一个相对庞大的链接库；二是如果其他计算机上没有安装对应的运行库，则

用动态编译的可执行文件就不能运行。

111. 动态链接和静态链接区别

1) https://blog.csdn.net/kang___xi/article/details/80210717
2) 静态连接库就是把(lib)文件中用到的函数代码直接链接进目标程序，程序运行的时候不再需要其它的库文件；动态链接就是把调用的函数所在文件模块（DLL）和调用函数在文件中的位置等信息链接进目标程序，程序运行的时候再从 DLL 中寻找相应函数代码，因此需要相应 DLL 文件的支持。
3) 静态链接库与动态链接库都是共享代码的方式，如果采用静态链接库，则无论你愿不愿意，lib 中的指令都全部被直接包含在最终生成的 EXE 文件中了。但是若使用 DLL，该 DLL 不必被包含在最终 EXE 文件中，EXE 文件执行时可以“动态”地引用和卸载这个与 EXE 独立的 DLL 文件。静态链接库和动态链接库的另外一个区别在于静态链接库中不能再包含其他的动态链接库或者静态库，而在动态链接库中还可以再包含其他的动态或静态链接库。
4) 动态库就是在需要调用其中的函数时，根据函数映射表找到该函数然后调入堆栈执行。如果在当前工程中有多处对 dll 文件中同一个函数的调用，那么执行时，这个函数只会留下一份拷贝。但是如果有多处对 lib 文件中同一个函数的调用，那么执行时，该函数将在当前程序的执行空间里留下多份拷贝，而且是一处调用就产生一份拷贝。

112. 在不使用额外空间的情况下，交换两个数?

1) 算术

```
    x = x + y;
y = x - y;
    x = x - y;
```

2) 异或

```
    x = x^y;// 只能对 int,char..
y = x^y;
x = x^y;
x ^= y ^= x;
```

113. strcpy 和 memcpy 的区别

1、复制的内容不同。strcpy 只能复制字符串，而 memcpy 可以复制任意内容，例如字

符数组、整型、结构体、类等。

2、复制的方法不同。strcpy 不需要指定长度，它遇到被复制字符的串结束符"\0"才结束，所以容易溢出。memcpy 则是根据其第 3 个参数决定复制的长度。

3、用途不同。通常在复制字符串时用 strcpy，而需要复制其他类型数据时则一般用 memcpy

114. 执行 int main(int argc, char *argv[])时的内存结构

参数的含义是程序在命令行下运行的时候，需要输入 argc 个参数，每个参数是以 char 类型输入的，依次存在数组里面，数组是 argv[]，所有的参数在指针 char * 指向的内存中，数组的中元素的个数为 argc 个，第一个参数 argv[0]为程序的名称。

115. volatile 关键字的作用?

volatile 关键字是一种类型修饰符，用它声明的类型变量表示可以被某些编译器未知的因素更改，比如：操作系统、硬件或者其它线程等。遇到这个关键字声明的变量，编译器对访问该变量的代码就不再进行优化，从而可以提供对特殊地址的稳定访问。声明时语法：int volatile vInt; 当要求使用 volatile 声明的变量的值的时候，系统总是重新从它所在的内存读取数据，即使它前面的指令刚刚从该处读取过数据。而且读取的数据立刻被保存。

volatile 用在如下的几个地方：

1) 中断服务程序中修改的供其它程序检测的变量需要加 volatile；

2) 多任务环境下各任务间共享的标志应该加 volatile；

3) 存储器映射的硬件寄存器通常也要加 volatile 说明，因为每次对它的读写都可能由不同意义；

116. 讲讲大端小端，如何检测（三种方法）

大端模式：是指数据的高字节保存在内存的低地址中，而数据的低字节保存在内存的高地址端。

小端模式，是指数据的高字节保存在内存的高地址中，低位字节保存在在内存的低地址端。

1) 直接读取存放在内存中的十六进制数值，取低位进行值判断

```
int a = 0x12345678;
int *c = &a;
```

c[0] == 0x12　大端模式

c[0] == 0x78　小段模式

2) 用共同体来进行判断

union 共同体所有数据成员是共享一段内存的，后写入的成员数据将覆盖之前的成员数据，成员数据都有相同的首地址。Union 的大小为最大数据成员的大小。

union 的成员数据共用内存，并且首地址都是低地址首字节。Int i= 1 时：大端存储 1 放在最高位，小端存储 1 放在最低位。当读取 char ch 时，是最低地址首字节，大小端会显示不同的值。

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

struct Test
{
    int a;
    char b[sizeof(a)];
};

int main(){
    Test test;
    test.a = 0x1234;
    if (test.b[0] == 0x12 && test.b[1] = 0x34)
        cout << "Big" << endl;
    else if (test.b[0] == 0x34 && test.b[1] = 0x12)
        cout << "Small" << endl;
    else
        cout << "unknown" << endl;
    return 0;
}
```

117. 查看内存的方法

1. 首先打开 vs 编译器，创建好项目，并且将代码写进去，这里就不贴代码了，你可以随便的写个做个测试;
2. 调试的时候做好相应的断点，然后点击**开始调试**;
3. 程序调试之后会在你设置断点的地方暂停，然后选择**调试->窗口->内存**，就打开了内存数据查看的窗口了。

118. 空类会默认添加哪些东西？怎么写？

1) Empty(); // 缺省构造函数//
2) Empty(const Empty&); // 拷贝构造函数//
3) ~Empty(); // 析构函数//
4) Empty& operator=(const Empty&); // 赋值运算符//

119. 标准库是什么？

1) C++ 标准库可以分为两部分：

标准函数库： 这个库是由通用的、独立的、不属于任何类的函数组成的。函数库继承自 C 语言。

面向对象类库： 这个库是类及其相关函数的集合。

2) 输入/输出 I/O、字符串和字符处理、数学、时间、日期和本地化、动态分配、其他、宽字符函数

3) 标准的 C++ I/O 类、String 类、数值类、STL 容器类、STL 算法、STL 函数对象、STL 迭代器、STL 分配器、本地化库、异常处理类、杂项支持库

120. const char* 与 string 之间的关系，传递参数问题？

1) string 是 c++标准库里面其中一个，封装了对字符串的操作，实际操作过程我们可以用 const char*给 string 类初始化
2) 三者的转化关系如下所示：
   a) string 转 const char*

```
string s = “abc”;
const char* c_s = s.c_str();
```

   b) const char* 转 string，直接赋值即可

```
const char* c_s = “abc”;
string s(c_s);
```

   c) string 转 char*

```
string s = “abc”;
char* c;
const int len = s.length();
c = new char[len+1];
strcpy(c,s.c_str());
```

d) char* 转 string

char* c = “abc”;

string s(c);

e) const char* 转 char*

const char* cpc = “abc”;

char* pc = new char[strlen(cpc)+1];

strcpy(pc,cpc);

f) char* 转 const char*，直接赋值即可

char* pc = “abc”;

const char* cpc = pc;

121. new、delete、operator new、operator delete、placement new、placement delete

1) new operator

new operator 完成了两件事情：用于申请内存和初始化对象。

例如：string* ps = new string("abc");

2) operator new

operator new 类似于 C 语言中的 malloc，只是负责申请内存。

例如：void* buffer = operator new(sizeof(string)); 注意这里 new 前要有个 operator。

3) placement new

用于在给定的内存中初始化对象。

例如：void* buffer = operator new(sizeof(string));buffer = new(buffer) string("abc"); 调用了 placement new，在 buffer 所指向的内存中创建了一个 string 类型的对象并且初始值为“abc”。

4) 因此可以看出：new operator 可以分解 operator new 和 placement new 两个动作，是 operator new 和 placement new 的结合。与 new 对应的 delete 没有 placement delete 语法，它只有两种，分别是 delete operator 和 operator delete。delete operator 和 new operator 对应，完成析构对象和释放内存的操作。而 operator delete 只是用于内存的释放，与 C 语言中的 free 相似。

122. 为什么拷贝构造函数必须传引用不能传值?

1) 拷贝构造函数的作用就是用来复制对象的，在使用这个对象的实例来初始化这个对象的一个新的实例。

2) 参数传递过程到底发生了什么?

将地址传递和值传递统一起来，归根结底还是传递的是"值"(地址也是值，只不过通过它可以找到另一个值)!

i)值传递:

对于内置数据类型的传递时，直接赋值拷贝给形参(注意形参是函数内局部变量);

对于类类型的传递时，需要首先调用该类的拷贝构造函数来初始化形参(局部对象);

如 void foo(class_type obj_local){}, 如果调用 foo(obj); 首先 class_type obj_local(obj) ,这样就定义了局部变量 obj_local 供函数内部使用

ii)引用传递:

无论对内置类型还是类类型，传递引用或指针最终都是传递的地址值！而地址总是指针类型(属于简单类型)，显然参数传递时，按简单类型的赋值拷贝，而不会有拷贝构造函数的调用(对于类类型).

上述 1) 2)回答了为什么拷贝构造函数使用值传递会产生无限递归调用，内存溢出。

拷贝构造函数用来初始化一个非引用类类型对象，如果用传值的方式进行传参数，那么构造实参需要调用拷贝构造函数，而拷贝构造函数需要传递实参，所以会一直递归。

123. 空类的大小是多少？为什么?

1) C++空类的大小不为 0，不同编译器设置不一样，vs 设置为 1;
2) C++标准指出，不允许一个对象（当然包括类对象）的大小为 0，不同的对象不能具有相同的地址;
3) 带有虚函数的 C++类大小不为 1，因为每一个对象会有一个 vptr 指向虚函数表，具体大小根据指针大小确定;
4) C++中要求对于类的每个实例都必须有独一无二的地址,那么编译器自动为空类分配一个字节大小，这样便保证了每个实例均有独一无二的内存地址。

124. 你什么情况用指针当参数，什么时候用引用，为什么?

1) 使用引用参数的主要原因有两个:

程序员能修改调用函数中的数据对象

通过传递引用而不是整个数据–对象，可以提高程序的运行速度

2) 一般的原则：

对于使用引用的值而不做修改的函数：

如果数据对象很小，如内置数据类型或者小型结构，则按照值传递；

如果数据对象是数组，则使用指针（唯一的选择），并且指针声明为指向 const 的指针；

如果数据对象是较大的结构，则使用 const 指针或者引用，已提高程序的效率。这样可以节省结构所需的时间和空间；

如果数据对象是类对象，则使用 const 引用（传递类对象参数的标准方式是按照引用传递）；

3) 对于修改函数中数据的函数：

如果数据是内置数据类型，则使用指针

如果数据对象是数组，则只能使用指针

如果数据对象是结构，则使用引用或者指针

如果数据是类对象，则使用引用

125. 大内存申请时候选用哪种？C++变量存在哪？变量的大小存在哪？符号表存在哪？

1. 大内存申请时，采用堆申请空间，用 new 申请；
2. 不同的变量存储在不同的地方，局部变量、全局变量、静态变量；
3. C++对变量名不作存储，在汇编以后不会出现变量名，变量名作用只是用于方便编译成汇编代码，是给编译器看的，是方便人阅读的

126. 为什么会有大端小端，htol 这一类函数的作用

1) 这是因为在计算机系统中，我们是以字节为单位的，每个地址单元都对应着一个字节，一个字节为 8bit。但是在 C 语言中除了 8bit 的 char 之外，还有 16bit 的 short 型，32bit 的 long 型（要看具体的编译器），另外，对于位数大于 8 位的处理器，例如 16 位或者 32 位的处理器，由于寄存器宽度大于一个字节，那么必然存在着一个如何将多个字节安排的问题。因此就导致了大端存储模式和小端存储模式。例如一个 16bit 的 short 型 x，在内存中的地址为 0x0010，x 的值为 0x1122，那么 0x11 为高字节，0x22 为低字节。对于 大端模式，就将 0x11 放在低地址中，即

0x0010 中，0x22 放在高地址中，即 0x0011 中。小端模式，刚好相反。我们常用的 X86 结构是小端模式，而 KEIL C51 则为大端模式。很多的 ARM，DSP 都为小端模式。有些 ARM 处理器还可以由硬件来选择是大端模式还是小端模式。

127. 静态函数能定义为虚函数吗？常函数？

1、static 成员不属于任何类对象或类实例，所以即使给此函数加上 virutal 也是没有任何意义的。2. 静态与非静态成员函数之间有一个主要的区别。那就是静态成员函数没有 this 指针。虚函数依靠 vptr 和 vtable 来处理。vptr 是一个指针，在类的构造函数中创建生成，并且只能用 this 指针来访问它，因为它是类的一个成员，并且 vptr 指向保存虚函数地址的 vtable.对于静态成员函数，它没有 this 指针，所以无法访问 vptr. 这就是为何 static 函数不能为 virtual.虚函数的调用关系：this -> vptr -> vtable ->virtual function

128. this 指针调用成员变量时，堆栈会发生什么变化？

当在类的非静态成员函数访问类的非静态成员时，编译器会自动将对象的地址传给作为隐含参数传递给函数，这个隐含参数就是 this 指针。即使你并没有写 this 指针，编译器在链接时也会加上 this 的，对各成员的访问都是通过 this 的。例如你建立了类的多个对象时，在调用类的成员函数时，你并不知道具体是哪个对象在调用，此时你可以通过查看 this 指针来查看具体是哪个对象在调用。This 指针首先入栈，然后成员函数的参数从右向左进行入栈，最后函数返回地址入栈。

129. 静态绑定和动态绑定的介绍

1) 对象的静态类型：对象在声明时采用的类型。是在编译期确定的。
2) 对象的动态类型：目前所指对象的类型。是在运行期决定的。对象的动态类型可以更改，但是静态类型无法更改。
3) 静态绑定：绑定的是对象的静态类型，某特性（比如函数）依赖于对象的静态类型，发生在编译期。
4) 动态绑定：绑定的是对象的动态类型，某特性（比如函数）依赖于对象的动态类型，发生在运行期。

130. 设计一个类计算子类的个数

1. 为类设计一个 static 静态变量 count 作为计数器；

2. 类定义结束后初始化 count;
3. 在构造函数中对 count 进行+1;
4. 设计拷贝构造函数，在进行拷贝构造函数中进行 count +1，操作;
5. 设计复制构造函数，在进行复制函数中对 count+1 操作;
6. 在析构函数中对 count 进行-1;

131. 怎么快速定位错误出现的地方

1. 如果是简单的错误，可以直接双击错误列表里的错误项或者生成输出的错误信息中带行号的地方就可以让编辑窗口定位到错误的位置上。
2. 对于复杂的模板错误，最好使用生成输出窗口。多数情况下出发错误的位置是最靠后的引用位置。如果这样确定不了错误，就需要先把自己写的代码里的引用位置找出来，然后逐个分析了。

132. 虚函数的代价?

1) 带有虚函数的类,每一个类会产生一个虚函数表,用来存储指向虚成员函数的指针，增大类;
2) 带有虚函数的类的每一个对象，都会有有一个指向虚表的指针，会增加对象的空间大小;
3) 不能再是内敛的函数，因为内敛函数在编译阶段进行替代，而虚函数表示等待，在运行阶段才能确定到低是采用哪种函数，虚函数不能是内敛函数。

133. 类对象的大小

1) 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;
2) 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;
3) 虚函数的话，会在类对象插入 vptr 指针，加上指针大小;
4) 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。

134. 移动构造函数

1) 有时候我们会遇到这样一种情况，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；
3) C++引入了移动构造函数，专门处理这种，用 a 初始化 b 后，就将 a 析构的情况；
4) 与拷贝类似，移动也使用一个对象的值设置另一个对象的值。但是，又与拷贝不同的是，移动实现的是对象值真实的转移（源对象到目的对象）：源对象将丢失其内容，其内容将被目的对象占有。移动操作的发生的时候，是当移动值的对象是未命名的对象的时候。这里未命名的对象就是那些临时变量，甚至都不会有名称。典型的未命名对象就是函数的返回值或者类型转换的对象。使用临时对象的值初始化另一个对象值，不会要求对对象的复制：因为临时对象不会有其它使用，因而，它的值可以被移动到目的对象。做到这些，就要使用移动构造函数和移动赋值：当使用一个临时变量对象进行构造初始化的时候，调用移动构造函数。类似的，使用未命名的变量的值赋给一个对象时，调用移动赋值操作；
5)

```
Example6 (Example6&& x) : ptr(x.ptr)
    {
        x.ptr = nullptr;
    }
    // move assignment
    Example6& operator= (Example6&& x)
    {
        delete ptr;
        ptr = x.ptr;
        x.ptr=nullptr;
        return *this;
}
```

135. 何时需要合成构造函数

1) 如果一个类没有任何构造函数，但他含有一个成员对象，该成员对象含有默认构造函数，那么编译器就为该类合成一个默认构造函数，因为不合成一个默认构造函数那么该成员对象的构造函数不能调用；
2) 没有任何构造函数的类派生自一个带有默认构造函数的基类，那么需要为该派生类合成一个构造函数，只有这样基类的构造函数才能被调用；

3) 带有虚函数的类，虚函数的引入需要进入虚表，指向虚表的指针，该指针是在构造函数中初始化的，所以没有构造函数的话该指针无法被初始化；
4) 带有一个虚基类的类

1) 并不是任何没有构造函数的类都会合成一个构造函数
2) 编译器合成出来的构造函数并不会显式设定类内的每一个成员变量

136. 何时需要合成复制构造函数

有三种情况会以一个对象的内容作为另一个对象的初值：
1) 对一个对象做显示的初始化操作，X xx = x;
2) 当对象被当做参数交给某个函数时；
3) 当函数传回一个类对象时；

1) 如果一个类没有拷贝构造函数，但是含有一个类类型的成员变量，该类型含有拷贝构造函数，此时编译器会为该类合成一个拷贝构造函数；
2) 如果一个类没有拷贝构造函数，但是该类继承自含有拷贝构造函数的基类，此时编译器会为该类合成一个拷贝构造函数；
3) 如果一个类没有拷贝构造函数，但是该类声明或继承了虚函数，此时编译器会为该类合成一个拷贝构造函数；
4) 如果一个类没有拷贝构造函数，但是该类含有虚基类，此时编译器会为该类合成一个拷贝构造函数；

137. 何时需要成员初始化列表？过程是什么？

1) 当初始化一个引用成员变量时；
2) 初始化一个 const 成员变量时；
3) 当调用一个基类的构造函数，而构造函数拥有一组参数时；
4) 当调用一个成员类的构造函数，而他拥有一组参数；
5) 编译器会一一操作初始化列表，以适当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前。list 中的项目顺序是由类中的成员声明顺序决定的，不是初始化列表中的排列顺序决定的。

138. 程序员定义的析构函数被扩展的过程？

1) 析构函数函数体被执行；

2) 如果 class 拥有成员类对象，而后者拥有析构函数，那么它们会以其声明顺序的相反顺序被调用；
3) 如果对象有一个 vptr，现在被重新定义
4) 如果有任何直接的上一层非虚基类拥有析构函数，则它们会以声明顺序被调用；
5) 如果任何虚基类拥有析构函数

139. 构造函数的执行算法?

1) 在派生类构造函数中，所有的虚基类及上一层基类的构造函数调用；
2) 对象的 vptr 被初始化；
3) 如果有成员初始化列表，将在构造函数体内扩展开来，这必须在 vptr 被设定之后才做；
4) 执行程序员所提供的代码；

140. 构造函数的扩展过程?

1) 记录在成员初始化列表中的数据成员初始化操作会被放在构造函数的函数体内，并与成员的声明顺序为顺序；
2) 如果一个成员并没有出现在成员初始化列表中，但它有一个默认构造函数，那么默认构造函数必须被调用；
3) 如果 class 有虚表，那么它必须被设定初值；
4) 所有上一层的基类构造函数必须被调用；
5) 所有虚基类的构造函数必须被调用。

141. 哪些函数不能是虚函数

1) 构造函数，构造函数初始化对象，派生类必须知道基类函数干了什么，才能进行构造；当有虚函数时，每一个类有一个虚表，每一个对象有一个虚表指针，虚表指针在构造函数中初始化；
2) 内联函数，内联函数表示在编译阶段进行函数体的替换操作，而虚函数意味着在运行期间进行类型确定，所以内联函数不能是虚函数；
3) 静态函数，静态函数不属于对象属于类，静态成员函数没有 this 指针，因此静态函数设置为虚函数没有任何意义。
4) 友元函数，友元函数不属于类的成员函数，不能被继承。对于没有继承特性的函数没有虚函数的说法。
5) 普通函数，普通函数不属于类的成员函数，不具有继承特性，因此普通函数没有虚

函数。

142. sizeof 和 strlen 的区别

1) strlen 计算字符串的具体长度（只能是字符串），不包括字符串结束符。返回的是字符个数。
2) sizeof 计算声明后所占的内存数（字节大小），不是实际长度。
3) sizeof 是一个取字节运算符，而 strlen 是个函数。
4) sizeof 的返回值=字符个数*字符所占的字节数，字符实际长度小于定义的长度，此时字符个数就等于定义的长度。若未给出定义的大小，分类讨论，对于字符串数组，字符大 小等于实际的字符个数+1；对于整型数组，字符个数为实际的字符个数。字符串每个字符占 1 个字节，整型数据每个字符占的字节数需根据系统的位数类确定，32 位占 4 个字节。
5) sizeof 可以用类型做参数，strlen 只能用 char*做参数，且必须以‘\0’结尾，sizeof 还可以用函数做参数；
6) 数组做 sizeof 的参数不退化，传递给 strlen 就退化为指针；

143. 简述 strcpy、sprintf 与 memcpy 的区别

1) 操作对象不同

① strcpy 的两个操作对象均为字符串

② sprintf 的操作源对象可以是多种数据类型，目的操作对象是字符串

③ memcpy 的两个对象就是两个任意可操作的内存地址，并不限于何种数据类型。

2) 执行效率不同

memcpy 最高，strcpy 次之，sprintf 的效率最低。

3) 实现功能不同

① strcpy 主要实现字符串变量间的拷贝

② sprintf 主要实现其他数据类型格式到字符串的转化

③ memcpy 主要是内存块间的拷贝。

144. 编码实现某一变量某位清 0 或置 1

```
#define BIT3 (0x1 << 3 ) Satic int a;
//设置 a 的 bit 3:
void set_bit3( void )
{
        a |= BIT3; //将 a 第 3 位置 1
}
//清 a 的 bit 3
void set_bit3( void )
{
      a &= ~BIT3; //将 a 第 3 位清零
}
```

145. 将“引用”作为函数参数有哪些特点?

1) 传递引用给函数与传递指针的效果是一样的。这时，被调函数的形参就成为原来主调函数中的实参变量或对象的一个别名来使用，所以在被调函数中对形参变量的操作就是对其相应的目标对象（在主调函数中）的操作。
2) 使用引用传递函数的参数，在内存中并没有产生实参的副本，它是直接对实参操作；而使用一般变量传递函数的参数，当发生函数调用时，需要给形参分配存储单元，形参变量是实参变量的副本；如果传递的是对象，还将调用拷贝构造函数。因此，当参数传递的数据较大时，用引用比用一般变量传递参数的效率和所占空间都好。
3) 使用指针作为函数的参数虽然也能达到与使用引用的效果，但是，在被调函数中同样要给形参分配存储单元，且需要重复使用"*指针变量名"的形式进行运算，这很容易产生错误且程序的阅读性较差；另一方面，在主调函数的调用点处，必须用变量的地址作为实参。而引用更容易使用，更清晰。

146. 分别写出 BOOL,int,float,指针类型的变量 a 与“零”的比较语句。

```
bool : if ( !a ) or if(a)
int : if ( a == 0)
float : const EXPRESSION EXP = 0.000001
if ( a < EXP && a >-EXP)
```

pointer : if (a != NULL) or if(a == NULL)

无论是float还是double类型的变量，都有精度限制。所以一定要避免将浮点变量用“==”或“！ =”与数字比较，应该设法转化成“>=”或“<=”形式。

147. 局部变量全局变量的问题?

1) 局部会屏蔽全局。要用全局变量，需要使用"::"局部变量可以与全局变量同名，在函数内引用这个变量时，会用到同名的局部变量，而不会用到全局变量。对于有些编译器而言，在同一个函数内可以定义多个同名的局部变量，比如在两个循环体内都定义一个同名的局部变量，而那个局部变量的作用域就在那个循环体内。
2) 如何引用一个已经定义过的全局变量，可以用引用头文件的方式，也可以用 extern 关键字，如果用引用头文件方式来引用某个在头文件中声明的全局变理，假定你将那个变写错了，那么在编译期间会报错，如果你用 extern 方式引用时，假定你犯了同样的错误，那么在编译期间不会报错，而在连接期间报错。
3) 全局变量可不可以定义在可被多个.C 文件包含的头文件中，在不同的 C 文件中以 static 形式来声明同名全局变量。可以在不同的 C 文件中声明同名的全局变量，前提是其中只能有一个 C 文件中对此变量赋初值，此时连接不会出错

148. 数组和指针的区别?

1) 数组在内存中是连续存放的，开辟一块连续的内存空间；数组所占存储空间：sizeof（数组名）；数组大小：sizeof(数组名)/sizeof(数组元素数据类型)；
2) 用运算符 sizeof 可以计算出数组的容量（字节数）。sizeof(p),p 为指针得到的是一个指针变量的字节数，而不是 p 所指的内存容量。
3) 编译器为了简化对数组的支持，实际上是利用指针实现了对数组的支持。具体来说，就是将表达式中的数组元素引用转换为指针加偏移量的引用。
4) 在向函数传递参数的时候，如果实参是一个数组，那用于接受的形参为对应的指针。也就是传递过去是数组的首地址而不是整个数组，能够提高效率；
5) 在使用下标的时候，两者的用法相同，都是原地址加上下标值，不过数组的原地址就是数组首元素的地址是固定的，指针的原地址就不是固定的。

149. C++如何阻止一个类被实例化？一般在什么时候将构造函数声明为 private?

1) 将类定义为抽象基类或者将构造函数声明为 private；

2) 不允许类外部创建类对象，只能在类内部创建对象

150. 如何禁止自动生成拷贝构造函数?

1) 为了阻止编译器默认生成拷贝构造函数和拷贝赋值函数，我们需要手动去重写这两个函数，某些情况下，为了避免调用拷贝构造函数和拷贝赋值函数，我们需要将他们设置成 private，防止被调用。
2) 类的成员函数和 friend 函数还是可以调用 private 函数，如果这个 private 函数只声明不定义，则会产生一个连接错误；
3) 针对上述两种情况，我们可以定一个 base 类，在 base 类中将拷贝构造函数和拷贝赋值函数设置成 private,那么派生类中编译器将不会自动生成这两个函数，且由于 base 类中该函数是私有的，因此，派生类将阻止编译器执行相关的操作。

151. assert 与 NDEBUGE

1) assert 宏的原型定义在<assert.h>中，其作用是如果它的条件返回错误，则终止程序执行，原型定义：

```
#include <assert.h>
void assert( int expression );
```

assert 的作用是现计算表达式 expression ，如果其值为假（即为 0），那么它先向 stderr 打印一条出错信息，然后通过调用 abort 来终止程序运行。如果表达式为真，assert 什么也不做。
2) NDEBUG 宏是 Standard C 中定义的宏，专门用来控制 assert()的行为。如果定义了这个宏，则 assert 不会起作用。定义 NDEBUG 能避免检查各种条件所需的运行时开销，当然此时根本就不会执行运行时检查。
3) C Standard 中规定了 assert 以宏来实现。<assert.h>被设计来可以被多次包含，其中一上来就 undef assert，然后由 NDEBUG 宏来决定其行为。

152. Debug 和 release 的区别

1) 调试版本，包含调试信息，所以容量比 Release 大很多，并且不进行任何优化（优化会使调试复杂化，因为源代码和生成的指令间关系会更复杂），便于程序员调试。Debug 模式下生成两个文件，除了.exe 或.dll 文件外，还有一个.pdb 文件，该文件记录了代码中断点等调试信息；
2) 发布版本，不对源代码进行调试，编译时对应用程序的速度进行优化，使得程序在代码大小和运行速度上都是最优的。（调试信息可在单独的 PDB 文件中生成）。

Release 模式下生成一个文件.exe 或.dll 文件。

3) 实际上，Debug 和 Release 并没有本质的界限，他们只是一组编译选项的集合，编译器只是按照预定的选项行动。事实上，我们甚至可以修改这些选项，从而得到优化过的调试版本或是带跟踪语句的发布版本。

153. main 函数有没有返回值

1) 程序运行过程入口点 main 函数，main（）函数返回值类型必须是 int，这样返回值才能传递给程序激活者（如操作系统）表示程序正常退出。main（int args, char **argv） 参数的传递。参数的处理，一般会调用 getopt（）函数处理，但实践中，这仅仅是一部分，不会经常用到的技能点。

154. 写一个比较大小的模板函数

```
#include<iostream>
using namespace std;
template<typename type1,typename type2>//函数模板
type1 Max(type1 a,type2 b)
{
    return a > b ? a : b;
}
void main()
{
    cout<<"Max = "<<Max(5.5,'a')<<endl;
}
```

155. c++怎么实现一个函数先于 main 函数运行

1) 如果在 main 函数之前声明一个类的全局的对象。那么其执行顺序，根据全局对象的生存期和作用域，肯定先于 main 函数。

```
class simpleClass
{
public:
        simpleClass( )
        {
                cout << "simpleClass constructor.." << endl;            }
```

```
};
simpleClass g_objectSimple;              //step1 全局对象
int _tmain(int argc, _TCHAR* argv[])    //step3
{
        return 0;
}
```

2) 定义在 main( )函数之前的全局对象、静态对象的构造函数在 main( )函数之前执行。
3) Main 函数执行之前，主要就是初始化系统相关资源；
   ① 设置栈指针
   ② 初始化 static 静态和 global 全局变量，即 data 段的内容
   ③ 将未初始化部分的全局变量赋初值：数值型 short，int，long 等为 0，bool 为 FALSE，指针为 NULL，等等，即.bss 段的内容
   ④ 全局对象初始化，在 main 之前调用构造函数
   ⑤ 将 main 函数的参数，argc，argv 等传递给 main 函数，然后才真正运行 main 函数
4) Main 函数执行之后
   ① 全局对象的析构函数会在 main 函数之后执行；
   ② 可以用_onexit 注册一个函数，它会在 main 之后执行;

156. 虚函数与纯虚函数的区别在于

1) 纯虚函数只有定义没有实现，虚函数既有定义又有实现；
2) 含有纯虚函数的类不能定义对象，含有虚函数的类能定义对象；

157. 智能指针怎么用？智能指针出现循环引用怎么解决？

1) shared_ptr

   调用一个名为 make_shared 的标准库函数，shared_ptr<int> p = make_shared<int>(42);通常用 auto 更方便，auto p = …;shared_ptr<int> p2(new int(2));

   每个 shared_ptr 都有一个关联的计数器，通常称为引用计数，一旦一个 shared_ptr 的计数器变为 0，它就会自动释放自己所管理的对象；shared_ptr 的析构函数就会递减它所指的对象的引用计数。如果引用计数变为 0，shared_ptr 的析构函数就会销毁对象，并释放它占用的内存。

2) unique_ptr

一个 unique_ptr 拥有它所指向的对象。某个时刻只能有一个 unique_ptr 指向一个给定对象。当 unique_ptr 被销毁时，它所指向的对象也被销毁。

3) weak_ptr

weak_ptr 是一种不控制所指向对象生存期的智能指针，它指向由一个 shared_ptr 管理的对象，将一个 weak_ptr 绑定到一个 shared_ptr 不会改变引用计数，一旦最后一个指向对象的 shared_ptr 被销毁，对象就会被释放，即使有 weak_ptr 指向对象，对象还是会被释放。

4) 弱指针用于专门解决 shared_ptr 循环引用的问题，weak_ptr 不会修改引用计数，即其存在与否并不影响对象的引用计数器。循环引用就是：两个对象互相使用一个 shared_ptr 成员变量指向对方。弱引用并不对对象的内存进行管理，在功能上类似于普通指针，然而一个比较大的区别是，弱引用能检测到所管理的对象是否已经被释放，从而避免访问非法内存。

158. strcpy 函数和 strncpy 函数的区别？哪个函数更安全？

1) 函数原型

char* strcpy(char* strDest, const char* strSrc)

char* strncpy(char* strDest, const char* strSrc, int pos)

2) strcpy 函数：如果参数 dest 所指的内存空间不够大，可能会造成缓冲溢出(buffer Overflow)的错误情况，在编写程序时请特别留意，或者用 strncpy()来取代。

strncpy 函数：用来复制源字符串的前 n 个字符，src 和 dest 所指的内存区域不能重叠，且 dest 必须有足够的空间放置 n 个字符。

3) 如果目标长>指定长>源长，则将源长全部拷贝到目标长，自动加上'\0'

如果指定长<源长，则将源长中按指定长度拷贝到目标字符串，不包括'\0'

如果指定长>目标长，运行时错误；

159. 为什么要用 static_cast 转换而不用 c 语言中的转换？

1) 更加安全；

2) 更直接明显，能够一眼看出是什么类型转换为什么类型，容易找出程序中的错误；可清楚地辨别代码中每个显式的强制转；可读性更好，能体现程序员的意图

160. 成员函数里 memset(this,0,sizeof(*this))会发生什么

1) 有时候类里面定义了很多 int,char,struct 等 c 语言里的那些类型的变量，我习惯在构

造函数中将它们初始化为 0，但是一句句的写太麻烦，所以直接就 memset(this, 0, sizeof *this);将整个对象的内存全部置为 0。对于这种情形可以很好的工作，但是下面几种情形是不可以这么使用的；

2) 类含有虚函数表：这么做会破坏虚函数表，后续对虚函数的调用都将出现异常；
3) 类中含有 C++类型的对象：例如，类中定义了一个 list 的对象，由于在构造函数体的代码执行之前就对 list 对象完成了初始化，假设 list 在它的构造函数里分配了内存，那么我们这么一做就破坏了 list 对象的内存。

161. 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

162. MFC 消息处理如何封装的?

163. 回调函数的作用

1) 当发生某种事件时，系统或其他函数将会自动调用你定义的一段函数;
2) 回调函数就相当于一个中断处理函数，由系统在符合你设定的条件时自动调用。为此，你需要做三件事：1，声明；2，定义；3，设置触发条件，就是在你的函数中把你的回调函数名称转化为地址作为一个参数，以便于系统调用;
3) 回调函数就是一个通过函数指针调用的函数。如果你把函数的指针（地址）作为参数传递给另一个函数，当这个指针被用为调用它所指向的函数时，我们就说这是回调函数;
4) 因为可以把调用者与被调用者分开。调用者不关心谁是被调用者，所有它需知道的，只是存在一个具有某种特定原型、某些限制条件（如返回值为 int）的被调用函数。

164. 随机数的生成

1) #include<time.h> srand((unsigned)time(NULL)); cout<<(rand()%(b-a))+a;
2) 由于 rand()的内部实现是用线性同余法做的，所以生成的并不是真正的随机数，而是在一定范围内可看为随机的伪随机数。
3) 种子写为 srand(time(0))代表着获取系统时间，电脑右下角的时间，每一秒后系统时间的改变，数字序列的改变得到的数字不同，这才得带不同的数字，形成了真随机数，即使是真随机数，也是有规律可循。