## 指针和引用的区别

1. 指针保存的是指向对象的地址，引用相当于变量的别名
2. 引用在定义的时候必须初始化，指针没有这个要求
3. 指针可以改变保存的地址，引用必须从一而终
4. 不存在空应引用，但是存在空指针NULL，相对而言引用更加安全
5. 引用的创建不会调用类的拷贝构造函数

## new/delete与malloc/free的区别

1. new是运算符，malloc是C语言库函数
2. new可以重载，malloc不能重载
3. new的变量是数据类型，malloc的是字节大小
4. new可以调用构造函数，delete可以调用析构函数，malloc/free不能
5. new返回的是指定对象的指针，而malloc返回的是void\*，因此malloc的返回值一般都需要进行类型转化
6. malloc分配的内存不够的时候可以使用realloc扩容，new没有这样的操作
7. new内存分配失败抛出bad\_malloc，malloc内存分配失败返回NULL值

## 定义和声明的区别

1. 声明是告诉编译器变量的类型和名字，不会为变量分配空间
2. 定义需要分配空间，同一个变量可以被声明多次，但是只能被定义一次

## 构造函数为什么一般不定义为虚函数？而析构函数一般写成虚函数的原因 ？

1. 虚函数的执行依赖于虚函数表。而虚函数表需要在构造函数中进行初始化工作，即初始化vptr，让他指向正确的虚函数表。而在构造对象期间，虚函数表还没有被初始化，将无法进行。
2. 在类的继承中，如果有基类指针指向派生类，那么用基类指针delete时，如果不定义成虚函数，派生类中派生的那部分无法析构。
3. 构造函数不要调用虚函数。在基类构造的时候，虚函数是非虚，不会走到派生类中，既是采用的静态绑定。显然的是：当我们构造一个子类的对象时，先调用基类的构造函数，构造子类中基类部分，子类还没有构造，还没有初始化，如果在基类的构造中调用虚函数，如果可以的话就是调用一个还没有被初始化的对象，那是很危险的，所以C++中是不可以在构造父类对象部分的时候调用子类的虚函数实现。但是不是说你不可以那么写程序，你这么写，编译器也不会报错。只是你如果这么写的话编译器不会给你调用子类的实现，而是还是调用基类的实现。

## C++的内存管理

在C++中，内存被分成五个区：栈、堆、自由存储区、静态存储区、常量区

1. 栈：存放函数的参数和局部变量，编译器自动分配和释放
2. 堆：new关键字动态分配的内存，由程序员手动进行释放，否则程序结束后，由操作系统自动进行回收（）stack区起始地址是在高地址，即是从高地址向低地址延伸。而heap区起始地址是在低地址，即是从低地址向高地址延伸）
3. 自由存储区：由malloc分配的内存，和堆十分相似，由对应的free进行释放
4. 全局/静态存储区：存放全局变量和静态变量
5. 常量区：存放常量，不允许被修改
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。

## C++ const关键字

详情见笔记

## [C++中static关键字](https://www.cnblogs.com/songdanzju/p/7422380.html)

1. 函数体内： static 修饰的局部变量作用范围为该函数体，不同于auto变量，其内存只被分配一次，因此其值在下次调用的时候维持了上次的值
2. 模块内：static修饰全局变量或全局函数，可以被模块内的所有函数访问，但是不能被模块外的其他函数访问，使用范围限制在声明它的模块内
3. 类中：修饰成员变量，表示该变量属于整个类所有，对类的所有对象只有一份拷贝
4. 类中：修饰成员函数，表示该函数属于整个类所有，不接受this指针，只能访问类中的static成员变量
5. static的作用是默认初始化为0（static变量）

（不能同时使用const static关键字修饰成员函数，C++编译器在实现const的成员函数的时候为了确保该函数不能修改类的实例的状态，会在函数中添加一个隐式的参数const this\*。但当一个成员为static的时候，该函数是没有this指针的。也就是说此时const的用法和static是冲突的。）

## [C++中的volatile关键字](https://www.cnblogs.com/reality-soul/p/6140192.html)

1. 在本次线程内, 当读取一个变量时，为提高存取速度，编译器优化时有时会先把变量读取到一个寄存器中；以后，再取变量值时，就直接从寄存器中取值；当变量值在本线程里改变时，会同时把变量的新值copy到该寄存器中，以便保持一致。
2. 当变量在因别的线程等而改变了值，该寄存器的值不会相应改变，从而造成应用程序读取的值和实际的变量值不一致。
3. 当该寄存器在因别的线程等而改变了值，原变量的值不会改变，从而造成应用程序读取的值和实际的变量值不一致
4. 下面是volatile变量的几个例子：   
   1) 并行设备的硬件寄存器（如：状态寄存器）   
   2) 一个中断服务子程序中会访问到的非自动变量(Non-automatic variables)   
   3) 多线程应用中被几个任务共享的变量
5. 一个参数既可以是const还可以是volatile，volatile因为它可能被意想不到地改变。它是const因为程序不应该试图去修改它

## strcpy，sprintf与memecpy的区别

这些函数的区别在于 实现功能 以及 操作对象 不同。  
strcpy 函数操作的对象是 字符串 ，完成 从 源字符串 到 目的字符串 的 拷贝 功能。  
snprintf 函数操作的对象 不限于字符串 ：虽然目的对象是字符串，但是源对象可以是字符串、也可以是任意基本类型的数据。这个函数主要用来实现 （字符串或基本数据类型）向 字符串 的转换 功能。如果源对象是字符串，并且指定 %s 格式符，也可实现字符串拷贝功能。  
memcpy 函数顾名思义就是 内存拷贝 ，实现 将一个 内存块 的内容复制到另一个 内存块 这一功能。内存块由其首地址以及长度确定。程序中出现的实体对象，不论是什么类型，其最终表现就是在内存中占据一席之地（一个内存区间或块）。因此，memcpy 的操作对象不局限于某一类数据类型，或者说可 适用于任意数据类型 ，只要能给出对象的起始地址和内存长度信息、并且对象具有可操作性即可。鉴于 memcpy 函数等长拷贝的特点以及数据类型代表的物理意义，memcpy 函数通常限于同种类型数据或对象之间的拷贝，其中当然也包括字符串拷贝以及基本数据类型的拷贝。  
对于字符串拷贝来说，用上述三个函数都可以实现，但是其实现的效率和使用的方便程度不同：

* strcpy 无疑是最合适的选择：效率高且调用方便。
* snprintf 要额外指定格式符并且进行格式转化，麻烦且效率不高。
* memcpy 虽然高效，但是需要额外提供拷贝的内存长度这一参数，易错且使用不便；并且如果长度指定过大的话（最优长度是源字符串长度 + 1），还会带来性能的下降。其实 strcpy 函数一般是在内部调用 memcpy 函数或者用汇编直接实现的，以达到高效的目的。因此，使用 memcpy 和 strcpy 拷贝字符串在性能上应该没有什么大的差别。

## C++的空类有哪些成员函数

1. 缺省的构造函数
2. 缺省的拷贝构造函数
3. 缺省的析构函数
4. 缺省的赋值运算符
5. 缺省取址运算符
6. 缺省取址运算符const
7. 有些书上只是简单的介绍了前四个函数，没有提及后面两个函数，但后两个函数也是空类的默认函数。另外西药注意的是，只有当实际使用这些函数的时候，编译器才会定义他们。

## 简述函数重载、重写和重定义

详情见笔记

## typedef和define有什么区别

1. 用法不同：typedef用来定义一种数据类型的别名，增强程序的可读性，define主要用来定义常量，以及书写复杂使用频繁的宏
2. 执行时间不同，typedef是编译过程的一部分，有类型检查得功能，define是宏定义，是预编译的步部分，起发生在编译之前，只是简单的进行字符串的替换，不进行类型的检查
3. 作用域不同：typedef有作用域限定，define不受作用域约束，只要define声明后的引用都是正确的。
4. 对指针的操作不同，typedef和define定义的指针时有很大的区别
5. 注意：typedef定义是语句，因为句尾要加分号，而define不是语句，千万不能在句尾加分号。

## #define和const的区别

1. #define定义的常量没有类型，所给出的是一个立即数；const定义的常量有类型名字，存放在静态区域
2. 处理阶段不同，#define定义的宏变量在预处理时进行替换，可能有多个拷贝，const所定义的变量在编译时确定其值，只有一个拷贝。
3. #define定义的常量是不可以用指针去指向，const定义的常量可以用指针去指向该常量的地址
4. #define可以定义简单的函数，const不可以定义函数

## C++多态

多态是指通过基类的指针或者引用，在运行时会根据调用函数的对象的类型来执行不同的函数。与之相对应的编译时绑定函数称为静态绑定。

多态的作用：

1. 应用程序不必为每一个派生类编写功能调用，只需要对抽象基类进行处理即可。大大提高程序的可复用性。//继承
2. 派生类的功能可以被基类的方法或引用变量所调用，这叫向后兼容，可以提高可扩充性和可维护性。 //多态的真正作用，以前需要用switch实现

多态分为静态多态和动态多态。静态多态是通过重载和模板技术实现，在编译的时候确定。动态多态通过虚函数和继承关系来实现，执行动态绑定，在运行的时候确定。

静态联编：又称静态绑定，指在调用同名函数（即重载函数）时编译器将根据调用时所使用的实参在编译时就确定下来应该调用的函数实现。它是在程序编译连接阶段进行联编的，这种联编又称为早期联编，这是因为这种联编工作是在程序运行之前完成的。它的优点是速度快，效率高，但灵活性不够。编译时所进行的联编又称为静态束定。束定是指确定所调用的函数与执行该函数代码之间的关系。

动态联编：也称动态绑定，是指在程序运行时，根据当时的情况来确定调用的同名函数的实现，实际上就是在运行时选择虚函数的实现。这种联编又称为晚期联编或动态(束定。实现条件：①要有继承性且要求创建子类型关系；)②要有虚函数；③通过基类的对象指针或引用访问虚函数。继承是动态联编的基础，虚函数是动态联编的关键，虚函数经过派生之后，在类族中就可以实现运行过程中的多态。动态联编要求在运行时解决程序中的函数调用与执行该函数代码间的关系，调用虚函数的对象是在运行时确定的。对于同一个对象的引用，采用不同的联编方式将会被联编到不同类的对象上。即不同联编可以选择不同的实现，这便是多态性。它的优点是灵活性强，但效率较低。

多态的实现：简而言之编译器根据虚函数表找到恰当的虚函数。对于一个父类的对象指针类型变量，如果给他赋父类对象的指针，那么他就调用父类中的函数，如果给他赋子类对象的指针，他就调用子类中的函数。函数执行之前通过查虚函数表来查找调用的函数。

## 什么情况下会调用拷贝构造函数

1. 用类的一个对象去初始化另一个对象的时候
2. 当函数的参数是类的对象时，就是值传递的时候，如果是引用传递则不会调用
3. 当函数的返回值是类的对象或者引用的时候

## 友元函数和友元类

1. 友元提供了不同类的成员函数之间、类的成员函数和一般函数之间进行数据共享的机制。
2. 通过友元，一个不同函数或者另一个类中的成员函数可以访问类中的私有成员和保护成员。
3. 友元的正确使用能提高程序的运行效率，但同时也破坏了类的封装性和数据的隐藏性，导致程序可维护性变差。
4. 有元函数是可以访问类的私有成员的非成员函数。它是定义在类外的普通函数，不属于任何类，但是需要在类的定义中加以声明。
5. 一个函数可以是多个类的友元函数，只需要在各个类中分别声明。
6. 友元类的所有成员函数都是另一个类的友元函数，都可以访问另一个类中的隐藏信息（包括私有成员和保护成员）。
7. 使用友元类时注意：

(1) 友元关系不能被继承。

(2) 友元关系是单向的，不具有交换性。若类B是类A的友元，类A不一定是类B的友元，要看在类中是否有相应的声明。

(3) 友元关系不具有传递性。若类B是类A的友元，类C是B的友元，类C不一定是类A的友元，同样要看类中是否有相应的申明

## C++字节对齐

详情见笔记

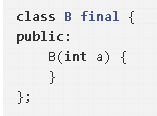
## 必须在构造函数初始化式里进行初始化的数据成员有哪些

1. 常量成员，因为常量只能初始化不能赋值，所以必须放在初始化列表里面
2. 引用类型，引用必须在定义的时候初始化，并且不能重新赋值，所以也要写在初始化列表里面
3. 没有默认构造函数的类类型，因为使用初始化列表可以不必调用默认构造函数来初始化，而是直接调用拷贝构造函数初始化

## C++四种类型转换

1. static\_cast：static\_cast基本上拥有与C旧式转型相同的威力和意义，以及相同的限制。但是，该类型转换操作符不能移除常量性，因为有一个专门的操作符用来移除常量性。
2. const\_cast：用来改变表达式中的常量性（constness）或者易变形（volatileness），只能用于此功能。
3. dynamic\_cast：将指向基类basic class object的pointer或者reference转型为指向派生类derived（或这sibling base）class object的pointer或者reference中，并且可以获知是否转型成功：如果转型失败，当转型对象是指针的时候会返回一个null指针；当转型对象是reference会抛出一个异常exception。dynamic\_cast无法应用在缺乏虚函数的类型上，也不能改变类型的常量性。  
   此外，dynamic\_cast还有一个用途就是找出被对象占用的内存的起始点。
4. reinterpret\_cast：这个操作符的转换结果几乎总是和编译器平台相关，所以不具有移植性。reinterpret\_cast的最常用用途

## C++如何创建一个类，使得他只能在堆或者栈上创建？不能被继承的类

1. 只能在堆上生成对象：将析构函数设置为私有。  
   原因：C++是静态绑定语言，编译器管理栈上对象的生命周期，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。若析构函数不可访问，则不能在栈上创建对象。
2. 只能在栈上生成对象：将new 和 delete 重载为私有。  
   原因：在堆上生成对象，使用new关键词操作，其过程分为两阶段：第一阶段，使用new在堆上寻找可用内存，分配给对象；第二阶段，调用构造函数生成对象。将new操作设置为私有，那么第一阶段就无法完成，就不能够再堆上生成对象。
3. 利用C++11的新关键字final：

## 栈溢出的原因以及解决方法

1. 函数调用层次过深,每调用一次,函数的参数、局部变量等信息就压一次栈
2. 局部变量体积太大。

1、增加栈内存的数目；增加栈内存方法如下,在vc6种依次选择Project->Setting->Link,在Category中选择output,在Reserve中输入16进制的栈内存大小如:0x10000000

2、使用堆内存；具体实现由很多种方法可以直接把数组定义改成指针,然后动态申请内存;也可以把局部变量变成全局变量,一个偷懒的办法是直接在定义前边加个static,呵呵,直接变成静态变量(实质就是全局变量)

## extern“C”作用

extern "C"告知编译器以C的形式编译，因为c中没有函数重载，函数在底层的签名是**函数名**，而c＋＋的函数签名是**函数名**返回值**参数类型1**参数类型2\_\_

Extern “C”是由Ｃ＋＋提供的一个连接交换指定符号，用于告诉Ｃ＋＋这段代码是Ｃ函数。这是因为C++编译后库中函数名会变得很长，与C生成的不一致，造成Ｃ＋＋不能直接调用C函数，加上extren “c”后，C++就能直接调用C函数了。  
  
Extern “C”主要使用正规DLL函数的引用和导出 和 在C++包含C函数或C头文件时使用。使用时在前面加上extern “c” 关键字即可。可以用一句话概括extern “C”这个声明的真实目的：实现C++与C及其它语言的混合编程。

*extern作用声明外部变量*，现在现代编译器一般采用按文件编译的方式，因此在编译时，各个文件中定义的全局变量是互相透明的，也就是说，*在编译时，全局变量的可见域限制在文件内部。*

## 什么是野指针

野指针不是NULL指针，是未初始化或者未清零的指针，它指向的内存地址不是程序员所期望的，可能指向了受限的内存

成因：

1. 指针变量没有被初始化
2. 指针指向的内存被释放了，但是指针没有置NULL
3. 指针超过了变量了的作用范围，比如b[10]，指针b+11

## C++中内存泄漏的几种情况

内存泄漏是指己动态分配的堆内存由于某种原因程序未释放或无法释放，造成系统内存的浪费，导致程序运行速度减慢甚至系统崩溃等严重后果。

1. 类的构造函数和析构函数中new和delete没有配套
2. 在释放对象数组时没有使用delete[]，使用了delete
3. 没有将基类的析构函数定义为虚函数，当基类指针指向子类对象时，如果基类的析构函数不是virtual，那么子类的析构函数将不会被调用，子类的资源没有正确释放，因此造成内存泄露
4. 没有正确的清楚嵌套的对象指针

## 定位内存泄漏

Windows平台下面Visual Studio 调试器和 C 运行时 (CRT) 库为我们提供了检测和识别内存泄漏的有效方法，原理大致如下：内存分配要通过CRT在运行时实现，只要在分配内存和释放内存时分别做好记录，程序结束时对比分配内存和释放内存的记录就可以确定是不是有内存泄漏。

 \_CRTDBG\_MAP\_ALLOC 时，还会显示在其中分配泄漏的内存的文件。 文件名后括号中的数字（本示例中为 10）是该文件中的行号。

注意：如果程序总是在同一位置退出，调用 [\_CrtDumpMemoryLeaks](http://msdn.microsoft.com/zh-cn/library/d41t22sb.aspx) 将非常容易。 如果程序从多个位置退出，则无需在每个可能退出的位置放置对 \_CrtDumpMemoryLeaks 的调用，而可以在程序开始处包含以下调用：

|  |  |
| --- | --- |
| 1 | \_CrtSetDbgFlag ( \_CRTDBG\_ALLOC\_MEM\_DF | \_CRTDBG\_LEAK\_CHECK\_DF ); |

该语句在程序退出时自动调用 \_CrtDumpMemoryLeaks。 必须同时设置\_CRTDBG\_ALLOC\_MEM\_DF 和 \_CRTDBG\_LEAK\_CHECK\_DF 两个位域。

通过上面的方法，我们几乎可以定位到是哪个地方调用内存分配函数malloc和new等，如上例中的GetMemory函数中，即第10行！但是不能定位到，在哪个地方调用GetMemory()导致的内存泄漏，而且在大型项目中可能有很多处调用GetMemory。如何要定位到在哪个地方调用GetMemory导致的内存泄漏？

定位内存泄漏的另一种技术涉及在关键点对应用程序的内存状态拍快照。 CRT 库提供一种结构类型 \_CrtMemState，您可用它存储内存状态的快照：

|  |  |
| --- | --- |
| 1 | \_CrtMemState s1, s2, s3; |

若要在给定点对内存状态拍快照，请向 [\_CrtMemCheckpoint](http://msdn.microsoft.com/zh-cn/library/h3z85t43.aspx) 函数传递 \_CrtMemState 结构。 该函数用当前内存状态的快照填充此结构：

|  |  |
| --- | --- |
| 1 | \_CrtMemCheckpoint( &s1 ); |

通过向 [\_CrtMemDumpStatistics](http://msdn.microsoft.com/zh-cn/library/swh3417y.aspx) 函数传递 \_CrtMemState 结构，可以在任意点转储该结构的内容：

|  |  |
| --- | --- |
| 1 | \_CrtMemDumpStatistics( &s1 ); |

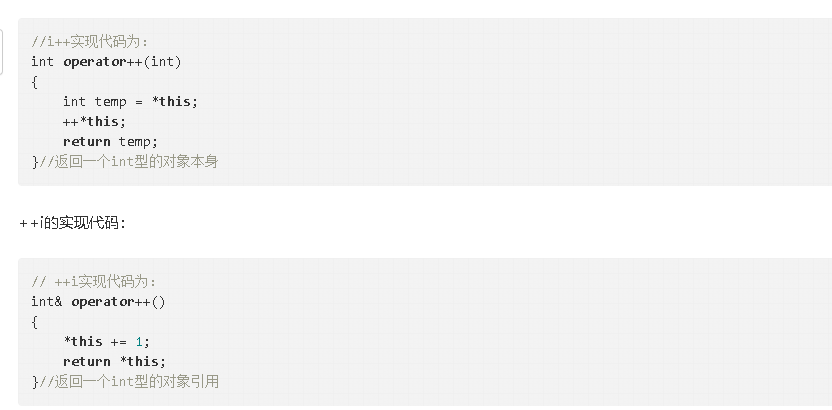
若要确定代码中某一部分是否发生了内存泄漏，可以在该部分之前和之后对内存状态拍快照，然后使用 [\_CrtMemDifference](http://msdn.microsoft.com/zh-cn/library/k4htzb06.aspx)比较这两个状态：

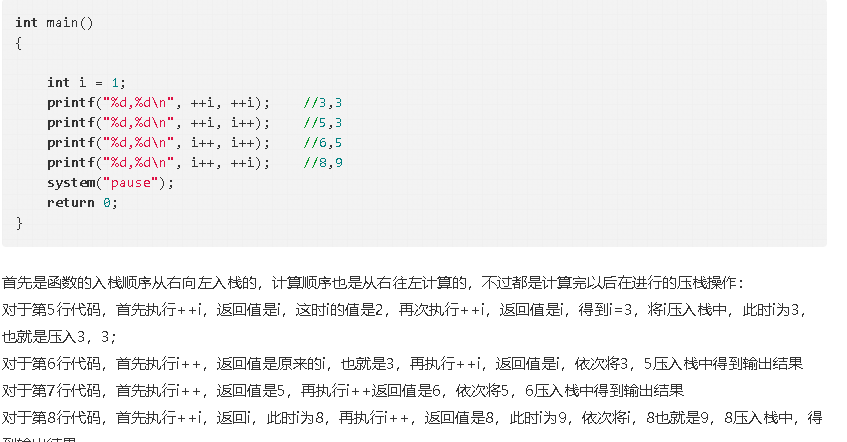
|  |  |
| --- | --- |
| 1  2  3  4  5  6 | \_CrtMemCheckpoint( &s1 );  // memory allocations take place here  \_CrtMemCheckpoint( &s2 );    if ( \_CrtMemDifference( &s3, &s1, &s2) )     \_CrtMemDumpStatistics( &s3 ); |

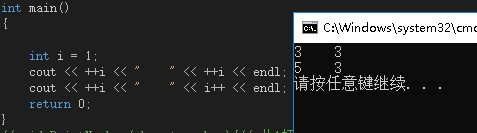
顾名思义，\_CrtMemDifference 比较两个内存状态（s1 和 s2），生成这两个状态之间差异的结果（s3）。 在程序的开始和结尾放置 \_CrtMemCheckpoint 调用，并使用\_CrtMemDifference 比较结果，是检查内存泄漏的另一种方法。 如果检测到泄漏，则可以使用 \_CrtMemCheckpoint 调用通过二进制搜索技术来划分程序和定位泄漏。

## 当i是一个整数的时候++i和i++那个更快一点？i++和++i的区别是什么？

理论上++i更快，实际与编译器优化有关，通常几乎无差别。  
i++实现的代码为：







## C语言的函数调用过程

函数的调用过程：

1）从栈空间分配存储空间

2）从实参的存储空间复制值到形参栈空间

3）进行运算

形参在函数未调用之前都是没有分配存储空间的，在函数调用结束之后，形参弹出栈空间，清除形参空间。

数组作为参数的函数调用方式是地址传递，形参和实参都指向相同的内存空间，调用完成后，形参指针被销毁，但是所指向的内存空间依然存在，不能也不会被销毁。

当函数有多个返回值的时候，不能用普通的 return 的方式实现，需要通过传回地址的形式进行，即地址/指针传递。

传值：传值，实际是把实参的值赋值给行参，相当于copy。那么对行参的修改，不会影响实参的值 。

传址： 实际是传值的一种特殊方式，只是他传递的是地址，不是普通的赋值，那么传地址以后，实参和行参都指向同一个对象，因此对形参的修改会影响到实参。

## STL常见问题

<https://blog.csdn.net/zzb2019/article/details/81195294>

// list 不能使用 STL 算法 sort ()

// 因为 STL 算法 sort() 只接受 RandomAccessIterator

// 本函数采用 mergesort

## STL中排序算法的实现是什么

STL中的sort()，在数据量大时，采用quicksort，分段递归排序；一旦分段后的数量小于某个门限值，改用Insertion sort，避免quicksort深度递归带来的过大的额外负担，如果递归层次过深，还会改用heapsort。

## auto和decltype类型指示符

double tempA = 3.0;

const double ctempA = 5.0;

const double ctempB = 6.0；

const double \*const cptrTempA = &ctempA;

/\*1.dclTempA推断为const double（保留顶层const，此处与auto不同）\*/

decltype(ctempA) dclTempA = 4.1;

/\*2.dclTempA为const double，不能对其赋值，编译不过\*/

dclTempA = 5;

/\*3.dclTempB推断为const double \* const\*/

decltype(cptrTempA) dclTempB = &ctempA;

/\*4.输出为4（32位计算机）和5\*/

cout<<sizeof(dclTempB)<<" "<<\*dclTempB<<endl;

/\*5.保留顶层const，不能修改指针指向的对象，编译不过\*/

dclTempB = &ctempB;

/\*6.保留底层const，不能修改指针指向的对象的值，编译不过\*/

\*dclTempB = 7.0;

int tempA = 0, &refTempA = tempA;

/\*1.dclTempA为引用，绑定到tempA\*/

decltype(refTempA) dclTempA = tempA;

/\*2.dclTempB为引用，必须绑定到变量，编译不过\*/

decltype(refTempA) dclTempB = 0;

/\*3.dclTempC为引用，必须初始化，编译不过\*/

decltype(refTempA) dclTempC;

/\*4.双层括号表示引用，dclTempD为引用，绑定到tempA\*/

decltype((tempA)) dclTempD = tempA;

const int ctempA = 1, &crefTempA = ctempA;

/\*5.dclTempE为常量引用，可以绑定到普通变量tempA\*/

decltype(crefTempA) dclTempE = tempA;

/\*6.dclTempF为常量引用，可以绑定到常量ctempA\*/

decltype(crefTempA) dclTempF = ctempA;

/\*7.dclTempG为常量引用，绑定到一个临时变量\*/

decltype(crefTempA) dclTempG = 0;

/\*8.dclTempH为常量引用，必须初始化，编译不过\*/

decltype(crefTempA) dclTempH;

/\*9.双层括号表示引用,dclTempI为常量引用，可以绑定到普通变量tempA\*/

decltype((ctempA)) dclTempI = ctempA;

int tempA = 2;

int \*ptrTempA = &tempA;

/\*1.常规使用dclTempA为一个int \*的指针\*/

decltype(ptrTempA) dclTempA;

/\*2.需要特别注意，表达式内容为解引用操作，dclTempB为一个引用，引用必须初始化，故编译不过\*/

decltype(\*ptrTempA) dclTempB;

int i = 4;

int arr[5] = { 0 };

int \*ptr = arr;

struct S{ double d; }s;

void Overloaded(int);

void Overloaded(char);//重载的函数

int RvalRef();

const bool Func(int);

//规则一：推导为其类型

decltype (arr) var1; //int [5] 标记符表达式

decltype (ptr) var2;//int \* 标记符表达式

decltype(s.d) var3;//doubel 成员访问表达式

//decltype(Overloaded) var4;//重载函数。编译错误。

//规则二：将亡值。推导为类型的右值引用。

decltype (RvalRef()) var5 = 1;

//规则三：左值，推导为类型的引用。

decltype ((i))var6 = i; //int&

decltype (true ? i : i) var7 = i; //int& 条件表达式返回左值。

decltype (++i) var8 = i; //int& ++i返回i的左值。

decltype(arr[5]) var9 = i;//int&. []操作返回左值

decltype(\*ptr)var10 = i;//int& \*操作返回左值

decltype("hello")var11 = "hello"; //const char(&)[9] 字符串字面常量为左值，且为const左值。

decltype总结

　　decltype和auto都可以用来推断类型，但是二者有几处明显的差异：1.auto忽略顶层const，decltype保留顶层const；2.对引用操作，auto推断出原有类型，decltype推断出引用；3.对解引用操作，auto推断出原有类型，decltype推断出引用；4.auto推断时会实际执行，decltype不会执行，只做分析。

## constexpr

将变量声明为constexpr类型以便由编译器来验证变量是否是一个常量表达式（不会改变，在编译过程中就能得到计算结果的表达式）。是一种比const更强的约束，这样可以得到更好的效率和安全性

constexpr指针不能用局部变量赋值，const指针可以；constexpr指针里是顶层const，即指针是常量，而不是所指向的类型是常量，如果要指向的类型也为常量，要用constexpr const来修饰。

int tempA = 4;

const int conTempA = 4;

constexpr int conexprTempA = 4;

/\*3.第一个通过，后面两个不过,因为constexpr int \*所限定的是指针是常量，故不能将常量的地址赋给顶层const\*/

constexpr int \*conexprPtrD = &g\_tempA;

constexpr int \*conexprPtrE = &g\_conTempA

constexpr int \*conexprPtrF = &g\_conexprTempA;

修饰引用, 确保其实常量引用需要constexpr const来修饰

int tempA = 4;

const int conTempA = 4;

constexpr int conexprTempA = 4;

/\*1.正常运行，编译通过\*/

const int &conptrA = tempA;

const int &conptrB = conTempA;

const int &conptrC = conexprTempA;

/\*2.有两个问题：一是引用到局部变量，不能再编译器确定；二是conexprPtrB和conexprPtrC应该为constexpr const类型，编译不过\*/

constexpr int &conexprPtrA = tempA;

constexpr int &conexprPtrB = conTempA

constexpr int &conexprPtrC = conexprTempA;

## insert和emplace

emplace 最大的作用是避免产生不必要的临时变量，因为它可以完成 in place 的构造，举个例子：

struct Foo {

Foo(int n, double x);

};

std::vector<Foo> v;

v.emplace(someIterator, 42, 3.1416); // 没有临时变量产生

v.insert(someIterator, Foo(42, 3.1416)); // 需要产生一个临时变量

v.insert(someIterator, {42, 3.1416}); // 需要产生一个临时变量

* 这是 emplace 和 insert 最大的区别点。emplace 的语法看起来不可思议，在上 面的例子中后面两个参数自动用来构造 vector 内部的 Foo 对象。做到这一点主要 使用了 C++11 的两个新特性 变参模板 和 完美转发。”变参模板”使得 emplace 可以接受任意参数，这样就可以适用于任意对象的构建。  
  ”完美转发”使得接收下来的参数 能够原样的传递给对象的构造函数，这带来另一个方便性就是即使是构造函数声明为explicit 它还是可以正常工作，因为它不存在临时变量和隐式转换。

## capacity和size 以及 empty()与size()

1. size()指容器当前拥有的元素个数；
2. capacity（）指容器在必须分配存储空间之前可以存储的元素总数

1、二者的作用是一样的，结果也是等价的。就是判断集合是否为空。

2、二者是等价的，为什么强调使用empty，因为empty效率更高。

3、在STL中，对于一般的集合，empty和size都是常数时间。但是对于list，empty是常数时间，size是线性时间；

　　考虑增删操作，对于一般的集合，增删是线性时间，因为涉及到元素的移动，增删的同时也就更新了元素个数。但是对list增删，是常数时间，有时候不会同事更新节点个数。

　　因此，对于一般的集合，size是实时更新的，empty与size可认为是等价的。但是对于list：对于empty，只需要检查head是否为end就可以了，为常数时间。对于size，必须遍历，为线性时间。比如在List中把一段list1.cbegin()到list.cend()移动到list2的结尾，插入操作为常数，但是不知道list1.cbegin()到list.cend()中间有多少个元素，需要n时间遍历

## 智能指针

<https://blog.csdn.net/weixin_43166107/article/details/89307437>

如何回答C++面试中关于智能指针的问题？

1、  什么是智能指针？

2、为什么要用智能指针以及智能指针的原理是什么？

3、  分析下常见的智能指针有哪些？

4、实现一个智能指针呗？（没具体说写哪个，建议默认写：unique\_ptr（scoped\_ptr））

1、答：智能指针(smart pointer)是存储指向动态分配（堆）对象指针的类，用于生存期控制，能够确保自动正确的销毁动态分配的对象，防止内存泄露（利用自动调用类的析构函数来释放内存）。它的一种通用实现技术是使用引用计数（除此之外还有资源独占，如(auto\_ptr）,只引用，不计数（weak\_ptr））。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）。

2、答：原因:

1、（手动malloc/new出来的资源，容易忘记free/delete；

2、 影响执行流的地方需要注意释放释放资源，容易导致资源泄漏（如free/delete在return之后）。

 3、中途抛出异常，无法释放资源。如：int\*p1=new int；int\*p2=new int [10000000]；delete p1;delele [ ] p2;因为p2new的内存比较大，万一new失败，则导致p1永远无法释放。

             原理：为了解决以（1）、（2）、（3）上问题，创建一份资源出来的时候，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。除此之外，通过运算符重载（重载\*和重在-> 等），可以向指针一样使用。

3、常见的智能指针以及解析：

**auto\_ptr**

（1）它是C++标准库提供的类模板，auto\_ptr对象通过初始化指向由new创建的动态内存，它是这块内存的拥有者，一块内存不能同时被分给两个拥有者（资源独占）。当auto\_ptr对象生命周期结束时，其析构函数会将auto\_ptr对象拥有的动态内存自动释放。

（2）auto\_ptr不能指向数组，因为auto\_ptr在析构的时候只是调用delete,而数组应该要调用delete[]。（但uniquearray管理的是一段连续的空间，它也是防拷贝的，功能类似于vector。）

（3）auto\_ptr不能作为容器对象，因为它不支持拷贝构造与赋值（出错了也不容易发现），STL容器中的元素经常要支持拷贝，赋值等操作，在这过程中auto\_ptr会传递所有权，那么就会出错。

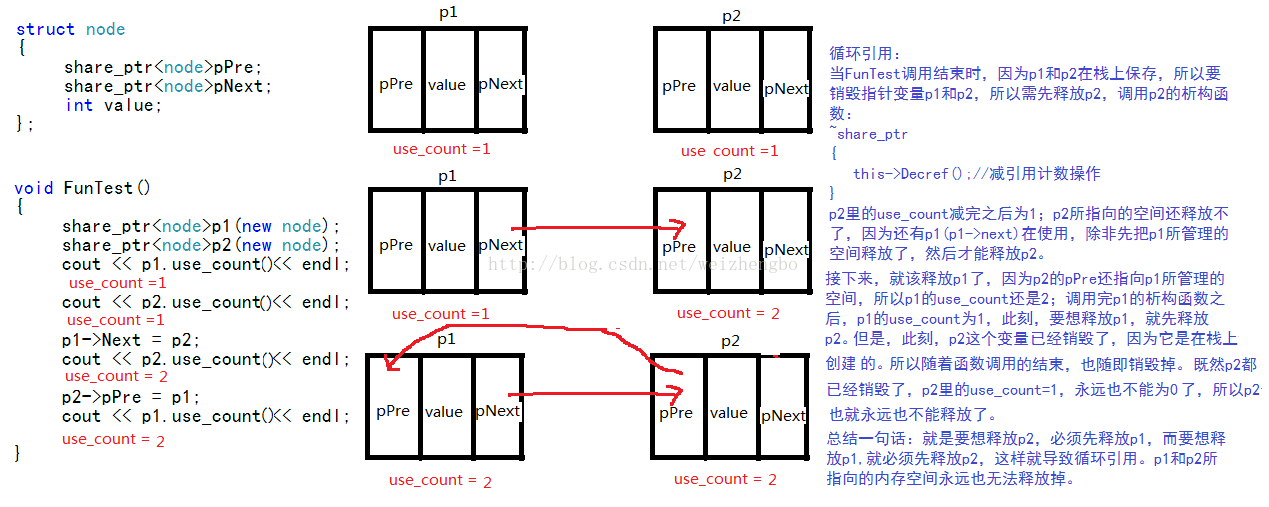
**unique\_ptr**

它是（ C++11引入的，前身是scoped\_ptr，scoped\_ptr是boost库里的），也不支持拷贝构造和赋值，但比auto\_ptr好，直接赋值会编译出错（与auto\_ptr最大的不同就是类内私有的声明了拷贝构造函数和赋值运算符重载，是针对auto\_ptr的缺点而出现的）。

**shared\_ptr**

C++11或boost的shared\_ptr，基于引用计数的智能指针。可随意赋值，直到内存的引用计数为0的时候这个内存会被释放。环状的链式结构可能会形成内存泄露（循环引用）

循环引用问题（常问到哦）



为了解决类似这样的问题，C++11引入了weak\_ptr，来打破这种循环引用。

**weak\_ptr**

C++11或boost的weak\_ptr，弱引用。 引用计数有一个问题就是互相引用形成环，这样两个指针指向的内存都无法释放。需要手动打破循环引用或使用weak\_ptr。顾名思义，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，在使用之前需要检查weak\_ptr是否为空指针。

weak\_ptr并没有重载operator->和operator \*操作符，因此不可直接通过weak\_ptr使用对象，典型的用法是调用其lock函数来获得shared\_ptr示例，进而访问原始对象。

Q：如何判断weak\_ptr的对象是否失效？   
A：1、expired()：检查被引用的对象是否已删除。   
2、lock()会返回shared指针，判断该指针是否为空。   
3、use\_count()也可以得到shared引用的个数，但速度较慢。

STL中的sort并非只是普通的快速排序，除了对普通的快速排序进行优化，它还结合了插入排序和堆排序。根据不同的数量级别以及不同情况，能自动选用合适的排序方法。当数据量较大时采用快速排序，分段递归。一旦分段后的数据量小于某个阀值，为避免递归调用带来过大的额外负荷，便会改用插入排序。而如果递归层次过深，有出现最坏情况的倾向，还会改用堆排序。

Register 将变量定义为存储在寄存器中，因为CPU访问寄存器的速度比访问内存的速度快，所以使用频繁的变量可以定义为Register 来加快程序运行速度。但是因为保存在寄存器中，所以不能取地址。

C语言中使用auto关键字声明一个变量为自动变量（即局部变量），是C语言中应用最广泛的一种类型，在函数内定义变量时，如果没有被声明为其他类型的变量都是自动变量，也就是说，省去类型说明符auto的都是自动变量。

比如局部变量 int a;应该被定义为 auto int a;

C++ 中 auto 用来作为类型推导，因此声明时必须赋值。

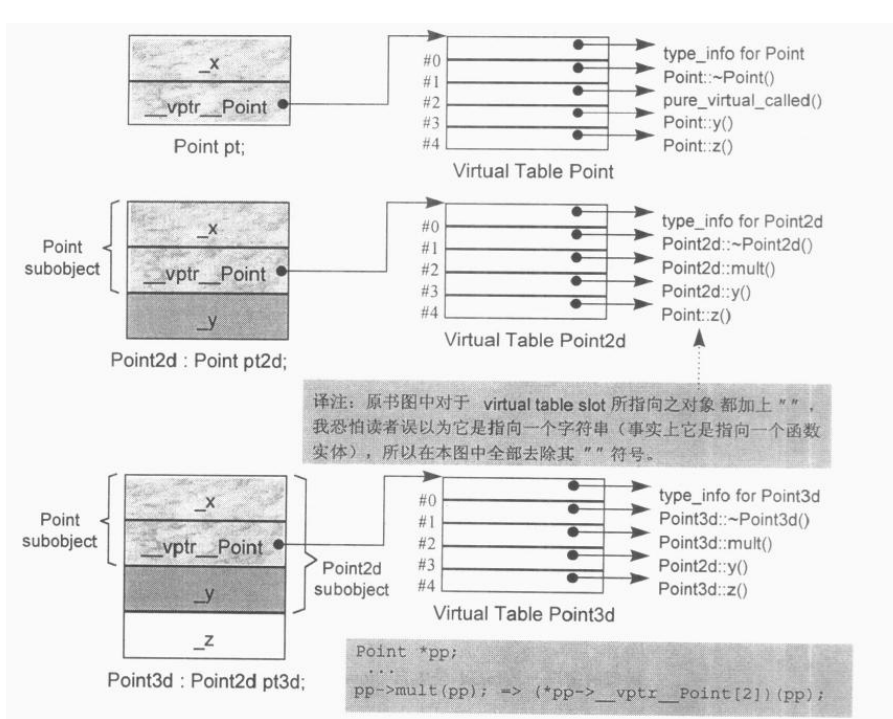
C++11新特性

<https://blog.csdn.net/jiange_zh/article/details/79356417>

C++对象模型

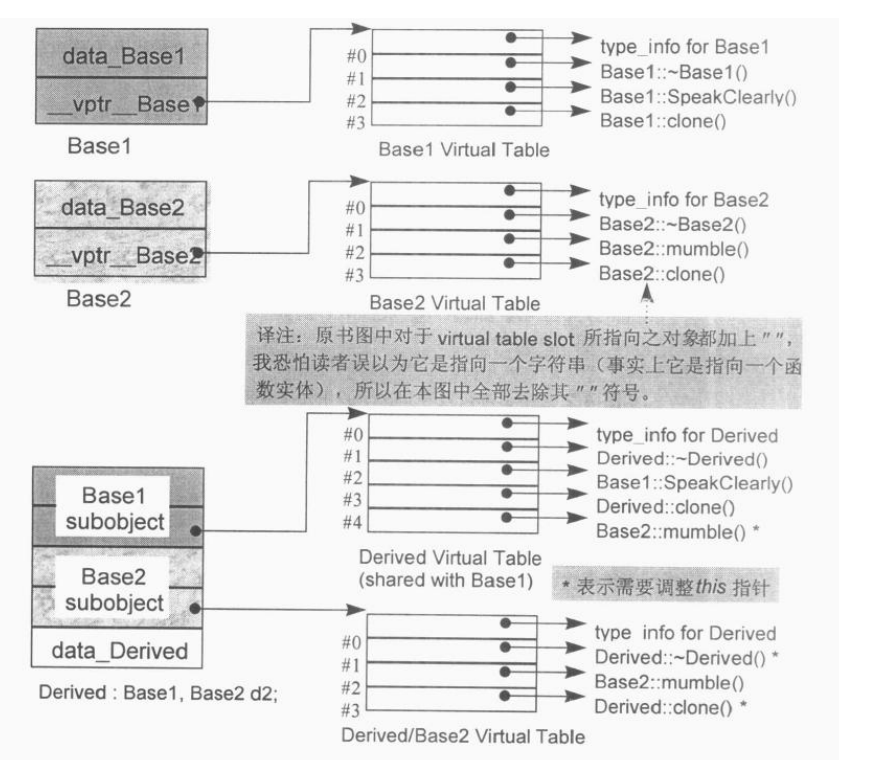
有虚函数，单继承时，不管有几层，只有一个虚表指针：

类的大小： 爷爷类数据大小+父类数据大小+指向子类的虚表指针

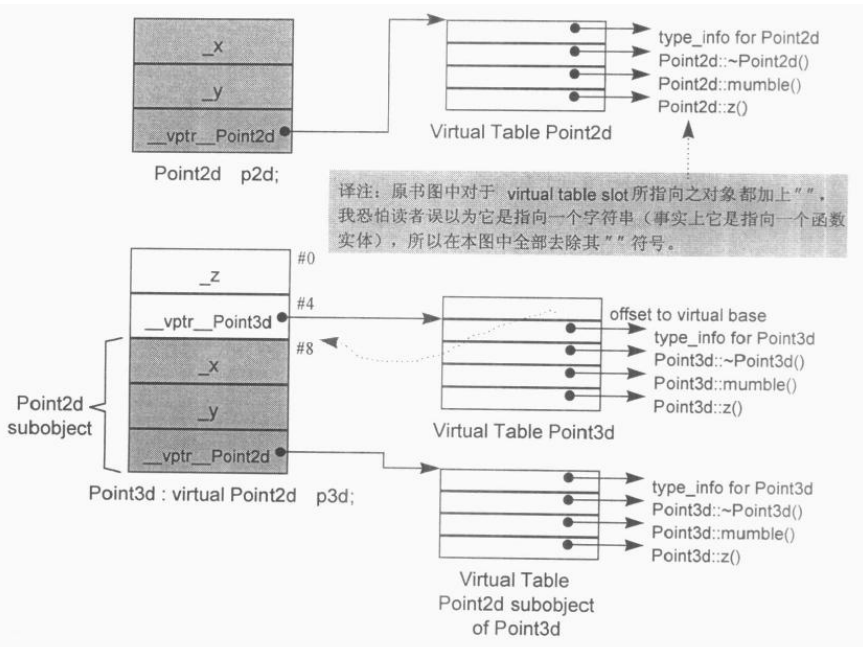


有虚函数的多重继承时

类的大小：



没有虚函数，虚拟继承时：



多重继承中，有普通继承和虚继承时：

<https://www.cnblogs.com/yiruhua/p/5616953.html>