**栈**，就是那些由编译器在需要的时候分配，在不需要的时候自动清除的变量的存储区。里面的变量通常是局部变量、函数参数等。在一个进程中，位于用户虚拟地址空间顶部的是用户栈，编译器用它来实现函数的调用。和堆一样，用户栈在程序执行期间可以动态地扩展和收缩。

**堆**，就是那些由 new 分配的内存块，他们的释放编译器不去管，由我们的应用程序去控制，一般一个 new 就要对应一个 delete。如果程序员没有释放掉，那么在程序结束后，操作系统会自动回收。堆可以动态地扩展和收缩。

**自由存储区**，就是那些由 malloc 等分配的内存块，他和堆是十分相似的，不过它是用 free 来结束自己的生命的。

**全局/静态存储区**，全局变量和静态变量被分配到同一块内存中，在以前的 C 语言中，全局变量又分为初始化的和未初始化的（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与静态变量在相邻的另一块区域，同时未被初始化的对象存储区可以通过 void\* 来访问和操纵，程序结束后由系统自行释放），在 C++ 里面没有这个区分了，他们共同占用同一块内存区。

**常量存储区**，这是一块比较特殊的存储区，他们里面存放的是常量，不允许修改（当然，你要通过非正当手段也可以修改，而且方法很多）

**明确区分堆与栈**

　　在 BBS 上，堆与栈的区分问题，似乎是一个永恒的话题，由此可见，初学者对此往往是混淆不清的，所以我决定拿他第一个开刀。

**首先，我们举一个例子：**

void f() { int\* p=new int[5]; }

　　这条短短的一句话就包含了堆与栈，看到 new，我们首先就应该想到，我们分配了一块堆内存，那么指针 p 呢？他分配的是一块栈内存，所以这句话的意思就是：在栈内存中存放了一个指向一块堆内存的指针 p。在程序会先确定在堆中分配内存的大小，然后调用 operator new 分配内存，然后返回这块内存的首地址，放入栈中，他在 VC6 下的汇编代码如下：

[复制代码](http://www.cnblogs.com/daocaoren/archive/2011/06/29/2092957.html)

　　00401028 push 14h  
  
　　0040102A call operator new (00401060)  
  
　　0040102F add esp,4  
  
　　00401032 mov dword ptr [ebp-8],eax  
  
　　00401035 mov eax,dword ptr [ebp-8]  
  
　　00401038 mov dword ptr [ebp-4],eax

[复制代码](http://www.cnblogs.com/daocaoren/archive/2011/06/29/2092957.html)

     这里，我们为了简单并没有释放内存，那么该怎么去释放呢？是 delete p 么？噢，错了，应该是 delete []p，这是为了告诉编译器：我删除的是一个数组，VC6 就会根据相应的 Cookie 信息去进行释放内存的工作。

**好了，我们回到我们的主题：堆和栈究竟有什么区别？**

**主要的区别由以下几点：**

　　1、管理方式不同；

　　2、空间大小不同；

　　3、能否产生碎片不同；

　　4、生长方向不同；

　　5、分配方式不同；

　　6、分配效率不同；

**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。

**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。

**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。

**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。

**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 malloc 函数进行分配，但是栈的动态分配和堆是不同的，他的动态分配是由编译器进行释放，无需我们手工实现。

**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。

　　从这里我们可以看到，堆和栈相比，由于大量 new/delete 的使用，容易造成大量的内存碎片；由于没有专门的系统支持，效率很低；由于可能引发用户态和核心态的切换，内存的申请，代价变得更加昂贵。所以栈在程序中是应用最广泛的，就算是函数的调用也利用栈去完成，函数调用过程中的参数，返回地址，EBP 和局部变量都采用栈的方式存放。所以，我们推荐大家尽量用栈，而不是用堆。

　　虽然栈有如此众多的好处，但是由于和堆相比不是那么灵活，有时候分配大量的内存空间，还是用堆好一些。

　　无论是堆还是栈，都要防止越界现象的发生（除非你是故意使其越界），因为越界的结果要么是程序崩溃，要么是摧毁程序的堆、栈结构，产生以想不到的结果,就算是在你的程序运行过程中，没有发生上面的问题，你还是要小心，说不定什么时候就崩掉，那时候 debug 可是相当困难的 ：）

　　对了，还有一件事，如果有人把堆栈合起来说，那它的意思是栈，可不是堆，呵呵，清楚了？

**static 用来控制变量的存储方式和可见性**

　　函数内部定义的变量，在程序执行到它的定义处时，编译器为它在栈上分配空间，函数在栈上分配的空间在此函数执行结束时会释放掉，这样就产生了一个问题: 如果想将函数中此变量的值保存至下一次调用时，如何实现？ 最容易想到的方法是定义一个全局的变量，但定义为一个全局变量有许多缺点，最明显的缺点是破坏了此变量的访问范围（使得在此函数中定义的变量，不仅仅受此 函数控制）。需要一个数据对象为整个类而非某个对象服务，同时又力求不破坏类的封装性，即要求此成员隐藏在类的内部，对外不可见。

**static 的内部机制**：

　　静态数据成员要在程序一开始运行时就必须存在。因为函数在程序运行中被调用，所以静态数据成员不能在任何函数内分配空间和初始化。这样，它的空间分配有三个可能的地方，一是作为类的外部接口的头文件，那里有类声明；二是类定义的内部实现，那里有类的成员函数定义；三是应用程序的 main(）函数前的全局数据声明和定义处。

　　静态数据成员要实际地分配空间，故不能在类的声明中定义（只能声明数据成员）。类声明只声明一个类的“尺寸和规格”，并不进行实际的内存分配，所以在类声明中写成定义是错误的。它也不能在头文件中类声明的外部定义，因为那会造成在多个使用该类的源文件中，对其重复定义。

　　static 被引入以告知编译器，将变量存储在程序的静态存储区而非栈上空间，静态数据成员按定义出现的先后顺序依次初始化，注意静态成员嵌套时，要保证所嵌套的成员已经初始化了。消除时的顺序是初始化的反顺序。

**static 的优势**：

　　可以节省内存，因为它是所有对象所公有的，因此，对多个对象来说，静态数据成员只存储一处，供所有对象共用。静态数据成员的值对每个对象都是一样，但它的 值是可以更新的。只要对静态数据成员的值更新一次，保证所有对象存取更新后的相同的值，这样可以提高时间效率。引用静态数据成员时，采用如下格式：

　　<类名>::<静态成员名>

　　如果静态数据成员的访问权限允许的话(即 public 的成员)，可在程序中，按上述格式来引用静态数据成员。

**Ps**：

　　(1) 类的静态成员函数是属于整个类而非类的对象，所以它没有this指针，这就导致了它仅能访问类的静态数据和静态成员函数。

　　(2) 不能将静态成员函数定义为虚函数。

　　(3) 由于静态成员声明于类中，操作于其外，所以对其取地址操作，就多少有些特殊，变量地址是指向其数据类型的指针，函数地址类型是一个“nonmember 函数指针”。

　　(4) 由于静态成员函数没有 this 指针，所以就差不多等同于 nonmember 函数，结果就产生了一个意想不到的好处：成为一个 callback 函数，使得我们得以将 c++ 和 c-based x window 系统结合，同时也成功的应用于线程函数身上。

　　(5) static 并没有增加程序的时空开销，相反她还缩短了子类对父类静态成员的访问时间，节省了子类的内存空间。

　　(6) 静态数据成员在<定义或说明>时前面加关键字 static。

　　(7) 静态数据成员是静态存储的，所以必须对它进行初始化。

　　(8) 静态成员初始化与一般数据成员初始化不同：

　　初始化在类体外进行，而前面不加 static，以免与一般静态变量或对象相混淆；

　　初始化时不加该成员的访问权限控制符 private、public；

　　初始化时使用作用域运算符来标明它所属类；

　　所以我们得出静态数据成员初始化的格式：

　　<数据类型><类名>::<静态数据成员名>=<值>

 　　(9) 为了防止父类的影响，可以在子类定义一个与父类相同的静态变量，以屏蔽父类的影响。这里有一点需要注意：我们说静态成员为父类和子类共享，但我们有重复定义了静态成员，这会不会引起错误呢？不会，我们的编译器采用了一种绝妙的手法：name-mangling 用以生成唯一的标志。