**分配方式简介**

　在C++中，内存分成5个区.

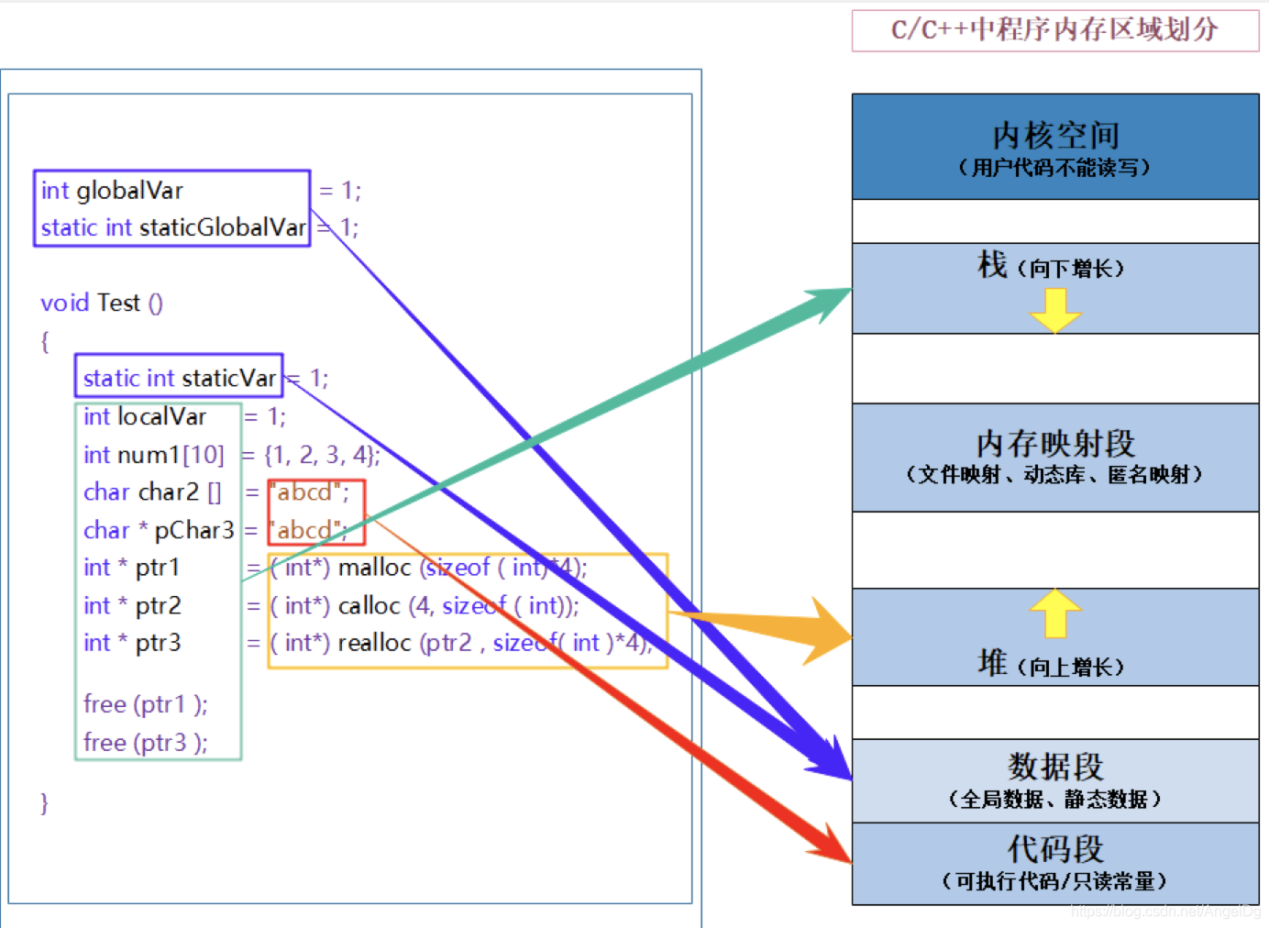
**栈（stack）**：指那些由编译器在需要的时候分配，不需要时⾃动清除的变量所在的存储区，效率高，分配的内存空间有限，形参和局部变量分配在栈区，栈是向低地址生长的数据结构，是一块连续的内存

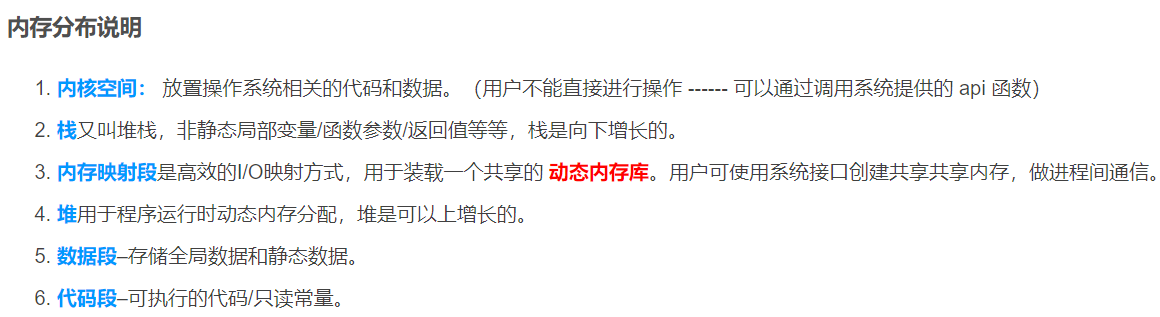
**堆（heap）**：由程序员控制内存的分配和释放的存储区，是向高地址生长的数据结构，是不连续的存储空间，堆的分配(malloc)和释放(free)有程序员控制，容易造成二次删除和内存泄漏

**静态存储区（static）**：存放全局变量和静态变量的存储区，初始化的变量放在初始化区，未初始化的变量放在未初始化区。在程序结束后释放这块空间

**常量存储区（const）**：存放常量字符串的存储区，只能读不能写，const修饰的局部变量存储在常量区（取决于编译器），const修饰的局部变量在栈区

**程序代码区**：存放源程序二进制代码





**明确区分堆与栈**

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

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。

　　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。

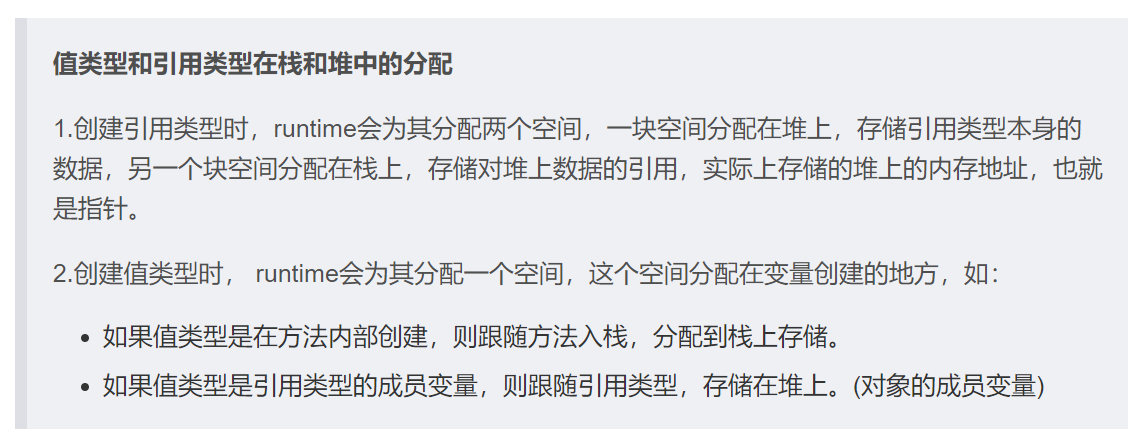
　　生长方向：对于堆来讲，生长方向是向上的，也就是向着内存地址增加的方向；对于栈来讲，它的生长方向是向下的，是向着内存地址减小的方向增长。

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

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。

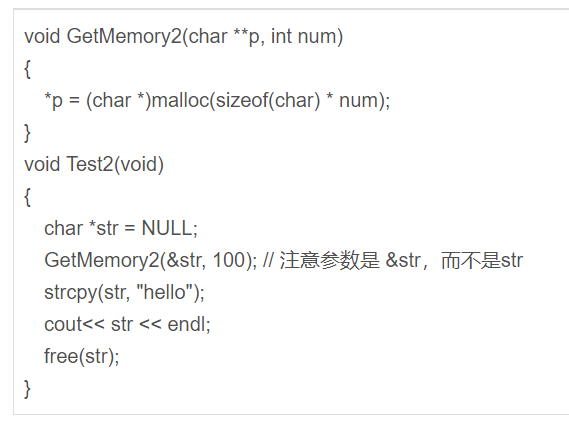


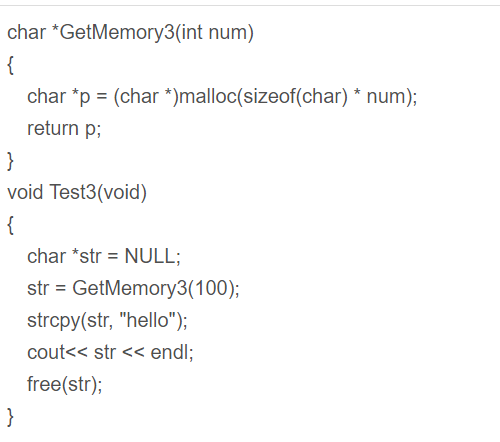
**总结栈是系统提供的功能，特点是快速高效，缺点是有限制，数据不灵活；而堆是函数库提供的功能，特点是灵活方便，数据适应面广泛，但是效率有一定降低。**



**内存泄漏问题**



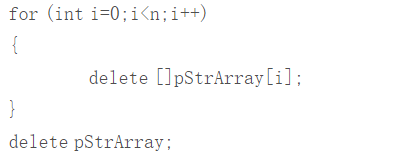


s

\*p=new

delete []p;

释放内存：



**常见的内存错误及其对策**

（1）内存分配未成功，却使用了它

常用解决办法是，在使用内存之前检查指针是否为NULL。如果指针p是函数的参数，那么在函数的入口处用assert(NULL != p)进行检查。如果是用malloc或new来申请内存，应该用if(NULL == p) 或if(NULL!=p)进行防错处理。

（2）内存分配虽然成功，但是尚未初始化就引用它

（3）内存分配成功并且已经初始化，但操作越过了内存的边界

例如在使用数组时经常发生下标“多1”或者“少1”的操作。特别是在for循环语句中，循环次数很容易搞错，导致数组操作越界。

（4）忘记了释放内存，造成内存泄露

含有这种错误的函数每被调用一次就丢失一块内存。刚开始时系统的内存充足，你看不到错误。终有一次程序突然死掉，系统出现提示：内存耗尽。

动态内存的申请与释放必须配对，程序中malloc与free的使用次数一定要相同，否则肯定有错误（new/delete同理）。

（5）释放了内存却继续使用它，有3种情况：

程序中的对象调用关系过于复杂，实在难以搞清楚某个对象究竟是否已经释放了内存，此时应该重新设计数据结构，从根本上解决对象管理的混乱局面。

函数的return语句写错了，注意不要返回指向“栈内存”的“指针”或者“引用”，因为该内存在函数体结束时被自动销毁。

使用free或delete释放了内存后，没有将指针设置为NULL。导致产生“野指针”。

对策：

【规则1】用malloc或new申请内存之后，应该立即检查指针值是否为NULL。防止使用指针值为NULL的内存。

【规则2】不要忘记为数组和动态内存赋初值。防止将未被初始化的内存作为右值使用。

【规则3】避免数组或指针的下标越界，特别要当心发生“多1”或者“少1”操作。

【规则4】动态内存的申请与释放必须配对，防止内存泄漏。

【规则5】用free或delete释放了内存之后，立即将指针设置为NULL，防止产生“野指针”。