大家都知道，在堆上分配的内存，如果不再使用了，就应该及时释放，以便后面其他地方可以重用。而在 C 语言中，内存管理器不会自动回收不再使用的内存。如果忘了释放不再使用的内存，这些内存就不能被重用了，这就造成了内存泄漏。

内存泄漏几乎是很难避免的，不管是老手还是新手，都存在这个问题，甚至 Windows 与 Linux 这类系统软件也或多或少存在着内存泄漏。

也许对一般的应用软件来说，这个问题似乎不是那么突出与严重。一两处内存泄漏通常并不致于让程序崩溃，也不会带来逻辑上的错误，而且在进程退出时，系统会自动释放所有与该进程相关的内存（共享内存除外），所以内存泄漏的后果相对来说还是比较温和的。但是，量变会导致质变，一旦内存泄漏过多以致耗尽内存，后续内存分配将会失败，程序就可能因此而崩溃。

在常见情况下，内存泄漏的主要可见症状就是罪魁进程的速度减慢。原因是体积大的进程更有可能被系统换出，让别的进程运行，而且大的进程在换进换出时花费的时间也更多。即使泄漏的内存本身并不被引用，但它仍然可能存在于页面中（内容自然是垃圾），这样就增加了进程的工作页数量，降低了性能。

下面展示了一些导致内存泄漏的常见场景。

清单 1. 简单的潜在堆内存丢失和缓冲区覆盖

```
void f1(char *explanation)
 {
     char *p1;

     p1 = malloc(100);
     sprintf(p1,"The f1 error occurred because of '%s'.", explanation);
     local_log(p1);
 }
```

您看到问题了吗？除非 `local_log()` 对 `free()` 释放内存具有不寻常的响应能力，否则每次对 `f1` 的调用都会泄漏 100 字节。在记忆棒增量分发数兆字节内存时，一次泄漏是微不足道的，但是连续操作数小时后，即使如此小的泄漏也会削弱应用程序。

在实际的 C 和 C++ 编程中，这不足以影响您对 `malloc()` 或 `new` 的使用，本部分开头的句子提到了"资源"不是仅指"内存"，因为还有类似以下内容的示例（请参见清单 2）。`FILE` 句柄可能与内存块不同，但是必须对它们给予同等关注：

清单 2. 来自资源错误管理的潜在堆内存丢失

```
int getkey(char *filename)
 {
     FILE *fp;
     int key;

     fp = fopen(filename, "r");
     fscanf(fp, "%d", &key);
     return key;
 }
```

`fopen` 的语义需要补充性的 `fclose`。在没有 `fclose()` 的情况下，C 标准不能指定发生的情况时，很可能是内存泄漏。其他资源（如信号量、网络句柄、数据库连接等）同样值得考虑。

.棘手的内存泄漏

```
static char *important_pointer = NULL;
 void f9()
 {
     if (!important_pointer)
     important_pointer = malloc(IMPORTANT_SIZE);
            ...
     if (condition)
     /* Ooops!  We just lost the reference important_pointer already held. */
     important_pointer = malloc(DIFFERENT_SIZE);
            ...
 }
```

do not返回局部指针变量或者局部变量的指针，除非是一个static局部变量

```
char *f0()
{
       char temp[]="123456789";  //加上static 才是正确的


       return temp;
}
```