引言

对于任何使用 C 语言的人，如果问他们 C 语言的最大烦恼是什么，其中许多人可能会回答说是*指针和内存泄漏*。这些的确是消耗了开发人员大多数调试时间的事项。指针和内存泄漏对某些开发人员来说似乎令人畏惧，但是一旦您了解了指针及其关联内存操作的基础，它们就是您在 C 语言中拥有的最强大工具。

本文将与您分享开发人员在开始使用指针来编程前应该知道的秘密。本文内容包括：

- 导致内存破坏的指针操作类型
- 在使用动态内存分配时必须考虑的检查点
- 导致内存泄漏的场景

如果您预先知道什么地方可能出错，那么您就能够小心避免陷阱，并消除大多数与指针和内存相关的问题。

什么地方可能出错？

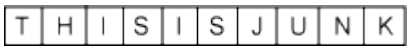
有几种问题场景可能会出现，从而可能在完成生成后导致问题。在处理指针时，您可以使用本文中的信息来避免许多问题。

未初始化的内存

在本例中，p 已被分配了 10 个字节。这 10 个字节可能包含垃圾数据，如图 1 所示。

```
1    char *p = malloc ( 10 );
```

图 1. 垃圾数据



如果在对这个 p 赋值前，某个代码段尝试访问它，则可能会获得垃圾值，您的程序可能具有不可预测的行为。p 可能具有您的程序从未曾预料到的值。

良好的实践是始终结合使用 memset 和 malloc，或者使用 calloc。

```
1    char *p = malloc (10);
2    memset(p,'',10);
```

现在，即使同一个代码段尝试在对 p 赋值前访问它，该代码段也能正确处理 Null 值（在理想情况下应具有的值），然后将具有正确的行为。

内存覆盖

由于 p 已被分配了 10 个字节，如果某个代码片段尝试向 p 写入一个 11 字节的值，则该操作将在不告诉您的情况下自动从其他某个位置“吃掉”一个字节。让我们假设指针 q 表示该内存。

图 2. 原始 q 内容

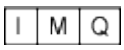
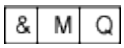


图 3. 覆盖后的 q 内容



结果，指针 q 将具有从未预料到的内容。即使您的模块编码得足够好，也可能由于某个共存模块执行某些内存操作而具有不正确的行为。下面的示例代码片段也可以说明这种场景。

```
1    char *name = (char *) malloc(11);
2    // Assign some value to name
3    memcpy ( p,name,11); // Problem begins here
```

在本例中，memcpy 操作尝试将 11 个字节写到 p，而后者仅被分配了 10 个字节。

作为良好的实践，每当向指针写入值时，都要确保对可用字节数和所写入的字节数进行交叉核对。一般情况下，memcpy 函数将是用于此目的的检查点。

内存读取越界

内存读取越界 (overread) 是指所读取的字节数多于它们应有的字节数。这个问题并不太严重，在此就不再详述了。下面的代码提供了一个示例。

```
1    char *ptr = (char *)malloc(10);
2    char name[20] ;
3    memcpy ( name,ptr,20); // Problem begins here
```

在本例中，memcpy 操作尝试从 ptr 读取 20 个字节，但是后者仅被分配了 10 个字节。这还会导致不希望的输出。

内存泄漏

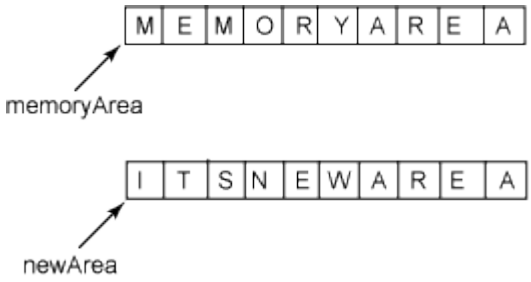
内存泄漏可能真正令人讨厌。下面的列表描述了一些导致内存泄漏的场景。

- 重新赋值我将使用一个示例来说明重新赋值问题。

```
1    char *memoryArea = malloc(10);
```

```
2     char *newArea = malloc(10);
```

- 这向如下面的图 4 所示的内存位置赋值。
- 图 4. 内存位置



memoryArea 和 newArea 分别被分配了 10 个字节，它们各自的内容如图 4 所示。

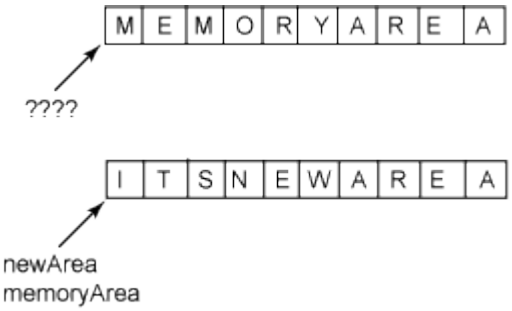
如果某人执行如下所示的语句（指针重新赋值）……

```
1     memoryArea = newArea;
```

则它肯定会在该模块开发的后续阶段给您带来麻烦。

在上面的代码语句中，开发人员将 memoryArea 指针赋值给 newArea 指针。结果，memoryArea 以前所指向的内存位置变成了孤立的，如下面的图 5 所示。它无法释放，因为没有指向该位置的引用。这会导致 10 个字节的内存泄漏。

图 5. 内存泄漏



在对指针赋值前，请确保内存位置不会变为孤立的。

- 首先释放父块

假设有一个指针 memoryArea，它指向一个 10 字节的内存位置。该内存位置的第三个字节又指向某个动态分配的 10 字节的内存位置，如图 6 所示。

图 6. 动态分配的内存


```
1     free(memoryArea)
```

如果通过调用 free 来释放了 memoryArea，则 newArea 指针也会因此而变得无效。newArea 以前所指向的内存位置无法释放，因为已经没有指向该位置的指针。换句话说，newArea 所指向的内存位置变为了孤立的，从而导致了内存泄漏。

每当释放结构化的元素，而该元素又包含指向动态分配的内存位置的指针时，应首先遍历子内存位置（在此例中为 newArea），并从那里开始释放，然后再遍历回父节点。

这里的正确实现应该为：

```
1    free( memoryArea->newArea);
2    free(memoryArea);
```

- 返回值的不正确处理

有时，某些函数会返回对动态分配的内存的引用。跟踪该内存位置并正确地处理它就成为了 calling 函数的职责。

```
1    char *func ( )
2    {
3          return malloc(20); // make sure to memset this location to ''…
4    }
5
6    void callingFunc ( )
7    {
8          func ( ); // Problem lies here
9    }
```

在上面的示例中，callingFunc() 函数中对 func() 函数的调用未处理该内存位置的返回地址。结果，func() 函数所分配的 20 个字节的块就丢失了，并导致了内存泄漏。

归还您所获得的

在开发组件时，可能存在大量的动态内存分配。您可能会忘了跟踪所有指针（指向这些内存位置），并且某些内存段没有释放，还保持分配给该程序。

始终要跟踪所有内存分配，并在任何适当的时候释放它们。事实上，可以开发某种机制来跟踪这些分配，比如在链表节点本身中保留一个计数器（但您还必须考

虑该机制的额外开销）。

访问空指针

访问空指针是非常危险的，因为它可能使您的程序崩溃。始终要确保您*不是*在访问空指针。

总结

本文讨论了几种在使用动态内存分配时可以避免的陷阱。要避免内存相关的问题，良好的实践是：

- 始终结合使用 memset 和 malloc，或始终使用 calloc。
- 每当向指针写入值时，都要确保对可用字节数和所写入的字节数进行交叉核对。
- 在对指针赋值前，要确保没有内存位置会变为孤立的。
- 每当释放结构化的元素（而该元素又包含指向动态分配的内存位置的指针）时，都应首先遍历子内存位置并从那里开始释放，然后再遍历回父节点。
- 始终正确处理返回动态分配的内存引用的函数返回值。
- 每个 malloc 都要有一个对应的 free。
- 确保您不是在访问空指针。