常见的 C 语言内存错误及对策

一、指针没有指向一块合法的内存

定义了指针变量，但是没有为指针分配内存，即指针没有指向一块合法的内存。浅显的例子就不举了，这里举几个比较隐蔽的例子。

1、结构体成员指针未初始化

```
struct student
{
  char *name;
  int score;
}stu,*pstu;
intmain()
{
  strcpy(stu.name,"Jimy");
  stu.score = 99;
  return 0;
}
```

很多初学者犯了这个错误还不知道是怎么回事。这里定义了结构体变量 stu，但是他没想到这个结构体内部 char *name 这成员在定义结构体变量 stu 时，只是给 name 这个指针变量本身分配了 4 个字节。name 指针并没有指向一个合法的地址，这时候其内部存的只是一些乱码。所以在调用 strcpy 函数时，会将字符串"Jimy"往乱码所指的内存上拷贝，而这块内存 name 指针根本就无权访问，导致出错。解决的办法是为 name 指针 malloc 一块空间。

同样，也有人犯如下错误：

```
intmain()
{
  pstu = (struct student*)malloc(sizeof(struct student));
  strcpy(pstu->name,"Jimy");
  pstu->score = 99;
  free(pstu);
  return 0;
}
```

为指针变量 pstu 分配了内存，但是同样没有给 name 指针分配内存。错误与上面第一种情况一样，解决的办法也一样。这里用了一个 malloc 给人一种错觉，以为也给 name 指针分配了内存。

2、没有为结构体指针分配足够的内存

```
intmain()
{
  pstu = (struct student*)malloc(sizeof(struct student*));
  strcpy(pstu->name,"Jimy");
```

```
    pstu->score = 99;
    free(pstu);
    return 0;
}
```

为 pstu 分配内存的时候，分配的内存大小不合适。这里把 sizeof(struct student)误写为 sizeof(struct student*)。当然 name 指针同样没有被分配内存。解决办法同上。

3、函数的入口校验

不管什么时候，我们使用指针之前一定要确保指针是有效的。

一般在函数入口处使用 assert(NULL != p)对参数进行校验。在非参数的地方使用 if（NULL != p）来校验。但这都有一个要求，即 p 在定义的同时被初始化为 NULL 了。比如上面的例子，即使用 if（NULL != p）校验也起不了作用，因为 name 指针并没有被初始化为 NULL，其内部是一个非 NULL 的乱码。

assert 是一个宏，而不是函数，包含在 assert.h 头文件中。如果其后面括号里的值为假，则程序终止运行，并提示出错；如果后面括号里的值为真，则继续运行后面的代码。这个宏只在 Debug 版本上起作用，而在 Release 版本被编译器完全优化掉，这样就不会影响代码的性能。

有人也许会问，既然在 Release 版本被编译器完全优化掉，那 Release 版本是不是就完全没有这个参数入口校验了呢？这样的话那不就跟不使用它效果一样吗？

是的，使用 assert 宏的地方在 Release 版本里面确实没有了这些校验。但是我们要知道，assert 宏只是帮助我们调试代码用的，它的一切作用就是让我们尽可能的在调试函数的时候把错误排除掉，而不是等到 Release 之后。它本身并没有除错功能。再有一点就是，参数出现错误并非本函数有问题，而是调用者传过来的实参有问题。assert 宏可以帮助我们定位错误，而不是排除错误。

二、为指针分配的内存太小

为指针分配了内存，但是内存大小不够，导致出现越界错误。

```
    char *p1 = “abcdefg”;
    char *p2 = (char *)malloc(sizeof(char)*strlen(p1));
    strcpy(p2,p1);
```

p1 是字符串常量，其长度为 7 个字符，但其所占内存大小为 8 个 byte。初学者往往忘了字符串常量的结束标志“\0”。这样的话将导致 p1 字符串中最后一个空字符“\0”没有被拷贝到 p2 中。解决的办法是加上这个字符串结束标志符：

```
    char *p2 = (char *)malloc(sizeof(char)*strlen(p1)+1*sizeof(char));
```

这里需要注意的是，只有字符串常量才有结束标志符。比如下面这种写法就没有结束标志符了：

```
    char a[7] = {‘a’,’b’,’c’,’d’,’e’,’f’,’g’};
```

另外，不要因为 char 类型大小为 1 个 byte 就省略 sizof（char）这种写法。这样只会使你的代码可移植性下降。

三、内存分配成功，但并未初始化

犯这个错误往往是由于没有初始化的概念或者是以为内存分配好之后其值自然为 0。未初始化指针变量也许看起来不那么严重，但是它确确实实是个非常严重的问题，而且往往出现这种错误很难找到原因。

曾经有一个学生在写一个 windows 程序时，想调用字库的某个字体。而调用这个字库需要填充一个结构体。他很自然的定义了一个结构体变量，然后把他想要的字库代码赋值给了相关的变量。但是，问题就来了，不管怎么调试，他所需要的这种字体效果总是不出来。我在检查了他的代码之后，没有发现什么问题，于是单步调试。在观察这个结构体变量的内存时，发现有几个成员的值为乱码。就是其中某一个乱码惹得祸！因为系统会按照这个结构体中的某些特定成员的值去字库中寻找匹配的字体，当这些值与字库中某种字体的某些项匹配时，就调用这种字体。但是很不幸，正是因为这几个乱码，导致没有找到相匹配的字体！因为系统并无法区分什么数据是乱码，什么数据是有效的数据。只要有数据，系统就理所当然的认为它是有效的。

也许这种严重的问题并不多见，但是也绝不能掉以轻心。所以在定义一个变量时，第一件事就是初始化。你可以把它初始化为一个有效的值，比如：

```
int i = 10;
char *p = (char *)malloc(sizeof(char));
```

但是往往这个时候我们还不确定这个变量的初值，这样的话可以初始化为 0 或 NULL。

```
int i = 0;
char *p = NULL;
```

如果定义的是数组的话，可以这样初始化：

```
int a[10] = {0};
```

或者用 memset 函数来初始化为 0：

```
memset（a,0,sizeof(a)）;
```

memset 函数有三个参数，第一个是要被设置的内存起始地址；第二个参数是要被设置的值；第三个参数是要被设置的内存大小，单位为 byte。这里并不想过多的讨论 memset 函数的用法，如果想了解更多，请参考相关资料。

至于指针变量如果未被初始化，会导致 if 语句或 assert 宏校验失败。这一点，上面已有分析。

四、内存越界

内存分配成功，且已经初始化，但是操作越过了内存的边界。这种错误经常是由于操作数组或指针时出现"多 1"或"少 1"。比如：

```
int a[10] = {0};
for (i=0; i<=10; i++)
{
  a[i] = i;
}
```

所以，for 循环的循环变量一定要使用半开半闭的区间，而且如果不是特殊情况，循环变量尽量从 0 开始。

五、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。甚至包括 windows，Linux 这类软件，都或多或少有内存泄漏。也许对于一般的应用软件来说，这个问题似乎不是那么突出，重启一下也不会造成太大损失。但是如果你开发的是嵌入式系统软件呢？比如汽车制动系统，心脏起搏器等对安全要求非常高的系统。你总不能让心脏起搏器重启吧，人家阎王老爷是非常好客的。

会产生泄漏的内存就是堆上的内存（这里不讨论资源或句柄等泄漏情况），也就是说由 malloc 系列函数或 new 操作符分配的内存。如果用完之后没有及时 free 或 delete，这块内存就无法释放，直到整个程序终止。

1、告老还乡求良田

怎么去理解这个内存分配和释放过程呢？先看下面这段对话：
万岁爷：爱卿，你为朕立下了汗马功劳，想要何赏赐啊？
某功臣：万岁，黄金白银，臣视之如粪土。臣年岁已老，欲告老还乡。臣乞良田千亩以荫后世，别无他求。
万岁爷：爱卿，你劳苦功高，却仅要如此小赏，朕今天就如你所愿。户部刘侍郎，查看湖广一带是否还有千亩上等良田未曾封赏。
刘侍郎：长沙尚有五万余亩上等良田未曾封赏。
万岁爷：在长沙拨良田千亩封赏爱卿。爱卿，良田千亩，你欲何用啊？
某功臣：谢万岁。长沙一带，适合种水稻，臣想用来种水稻。种水稻需要把田分为一亩一块，方便耕种。
。。。。

2、如何使用 malloc 函数

不要莫名其妙，其实上面这段小小的对话，就是 malloc 的使用过程。malloc 是一个函数，专门用来从堆上分配内存。使用 malloc 函数需要几个要求：
内存分配给谁？这里是把良田分配给某功臣。
分配多大内存？这里是分配一千亩。
是否还有足够内存分配？这里是还有足够良田分配。
内存的将用来存储什么格式的数据，即内存用来做什么?
这里是用来种水稻，需要把田分成一亩一块。分配好的内存在哪里？这里是在长沙。

如果这五点都确定，那内存就能分配。下面先看 malloc 函数的原型：

```
(void *)malloc(int size)
```

malloc 函数的返回值是一个 void 类型的指针，参数为 int 类型数据，即申请分配的内存大小，单位是 byte。内存分配成功之后，malloc 函数返回这块内存的首地址。你需要一个指针来接收这个地址。但是由于函数的返回值是 void *类型的，所以必须强制转换成你所接收的类型。也就是说，这块内存将要用来存储什么类型的数据。比如：

```
char *p = (char *)malloc(100);
```

在堆上分配了 100 个字节内存，返回这块内存的首地址，把地址强制转换成 char *类型后赋给 char *类型的指针变量 p。同时告诉我们这块内存将用来存储 char 类型的数据。也就是说你只能通过指针变量 p 来操作这块内存。这块内存本身并没有名字，对它的访问是匿名访问。

上面就是使用 malloc 函数成功分配一块内存的过程。但是，每次你都能分配成功吗？

不一定。上面的对话，皇帝让户部侍郎查询是否还有足够的良田未被分配出去。使用 malloc 函数同样要注意这点：如果所申请的内存块大于目前堆上剩余内存块（整块），则内存分配会失败，函数返回 NULL。注意这里说的"堆上剩余内存块"不是所有剩余内存块之和，因为 malloc 函数申请的是连续的一块内存。

既然 malloc 函数申请内存有不成功的可能，那我们在使用指向这块内存的指针时，必须用 if（NULL ！= p）语句来验证内存确实分配成功了。

3、用 malloc 函数申请 0 字节内存

另外还有一个问题：用 malloc 函数申请 0 字节内存会返回 NULL 指针吗？

可以测试一下，也可以去查找关于 malloc 函数的说明文档。申请 0 字节内存，函数并不返回 NULL，而是返回一个正常的内存地址。但是你却无法使用这块大小为 0 的内存。这好尺子上的某个刻度，刻度本身并没有长度，只有某两个刻度一起才能量出长度。对于这一点一定要小心，因为这时候 if（NULL ！= p）语句校验将不起作用。

4、内存释放

既然有分配，那就必须有释放。不然的话，有限的内存总会用光，而没有释放的内存却在空闲。与 malloc 对应的就是 free 函数了。free 函数只有一个参数，就是所要释放的内存块的首地址。比如上例：

```
free(p);
```

free 函数看上去挺狠的，但它到底作了什么呢？其实它就做了一件事：斩断指针变量与这块内存的关系。比如上面的例子，我们可以说 malloc 函数分配的内存块是属于 p 的，因为我们对这块内存的访问都需要通过 p 来进行。free 函数就是把这块内存和 p 之间的所有关系斩断。从此 p 和那块内存之间再无瓜葛。至于指针变量 p 本身保存的地址并没有改变，但是它对这个地址处的那块内存却已经没有所有权了。那块被释放的内存里面保存的值也没有改变，只是再也没有办法使用了。

这就是 free 函数的功能。按照上面的分析，如果对 p 连续两次以上使用 free 函数，肯定会发生错误。因为第一使用 free 函数时，p 所属的内存已经被释放，第二次使用时已经无内存可释放了。关于这点，我上课时让学生记住的是：一定要一夫一妻制，不然肯定出错。

malloc 两次只 free 一次会内存泄漏；malloc 一次 free 两次肯定会出错。也就是说，在程序中 malloc 的使用次数一定要和 free 相等，否则必有错误。这种错误主要发生在循环使用 malloc 函数时，往往把 malloc 和 free 次数弄错了。这里留个练习：
写两个函数，一个生成链表，一个释放链表。两个函数的参数都只使用一个表头指针。

5、内存释放之后

既然使用 free 函数之后指针变量 p 本身保存的地址并没有改变，那我们就需要重新把 p 的值变为 NULL：

```
p = NULL;
```

这个 NULL 就是我们前面所说的“栓野狗的链子”。如果你不栓起来迟早会出问题的。比如：

在 free（p）之后，你用 if（NULL ！= p）这样的校验语句还能起作用吗？例如：

```
char *p = (char *)malloc(100);
strcpy(p, “hello”);
free(p); /* p 所指的内存被释放，但是 p 所指的地址仍然不变*/
…
if (NULL != p)
{
    /* 没有起到防错作用*/
    strcpy(p, “world”); /* 出错*/
}
```

释放完块内存之后，没有把指针置 NULL，这个指针就成为了“野指针”，也有书叫“悬垂指针”。这是很危险的，而且也是经常出错的地方。所以一定要记住一条：free 完之后，一定要给指针置 NULL。

同时留一个问题：对 NULL 指针连续 free 多次会出错吗?为什么？如果让你来设计 free 函数，你会怎么处理这个问题？

六、内存已经被释放了，但是继续通过指针来使用

这里一般有三种情况：

第一种：就是上面所说的，free（p）之后，继续通过 p 指针来访问内存。解决的办法就是给 p 置 NULL。

第二种：函数返回栈内存。这是初学者最容易犯的错误。比如在函数内部定义了一个数组，却用 return 语句返回指向该数组的指针。解决的办法就是弄明白栈上变量的生命周期。

第三种：内存使用太复杂，弄不清到底哪块内存被释放，哪块没有被释放。解决的办法是重新设计程序，改善对象之间的调用关系。

上面详细讨论了常见的六种错误及解决对策，希望读者仔细研读，尽量使自己对每种错误发生的原因及预防手段烂熟于胸。一定要多练，多调试代码，同时多总结经验。