<https://blog.csdn.net/c_base_jin/article/details/26149653>

# 常见的错误

关于内存的一些知识已在[内存分配](http://blog.csdn.net/c_base_jin/article/details/25558929)中提及，现记录与分享常见的内存错误与对策。

类型 1：内存未分配成功，却使用了它。

方   法：在使用之前检查指针是否为NULL。

             1）当指针p是函数的参数时，在函数入口处用语句assert(p!=NULL)进行断言检查。

             2）当使用malloc或new来申请内存时，应该用if（p != NULL）进行防错检查。

类型 2：引用了尚未初始化的指针

原   因：内存的缺省初始值究竟是什么并没有统一的标准，在使用之前都进行初始化。

              1）没有初始化的观念。

              2）内存的缺省值是未定义，即垃圾值。

类型 3：越界操作内存

原   因：内存分配成功且初始了，但越界操作是不允许的。

例   如：在使用数组时经常发生下标“多1”或“少1”，特别是在for循环语句时。

类型 4：忘记释放内存，造成内存泄漏。

原   因：含有这种类型错误的函数，每被调用一次，就丢失一块内存。当内存充足时看不到这种错误带来的影响，当内存耗尽时系统提示：“内存耗尽”。因此，动态内存的申请与释放必须配对，程序中malloc与free的使用次数要相同。

类型 5：释放了内存却继续使用它

原   因：对应的情况有2种

              1）返回了“栈内存的指针或引用”，因为堆栈中的变量在函数结束后自动销毁。

              2）某块内存被free后，没有将指向该内存的指针设置为NULL，导致产生“野指针”。

## 使用规则

      为了保证代码的健壮和安全，可以参考如下的规则：

规则1：使用malloc申请的内存时，必须要立即检查相对应的指针是否为NULL。

规则2：初始化数组和动态内存。

规则3：避免数组或指针下标越界。

规则4：动态内存的申请和释放必须相配对，防止内存泄漏。

规则5：free释放某块内存之后，要立即将指针设置为NULL，防止产生野指针。

# 几个重要的概念

## 1.野指针

       概念：“野指针”不是NULL指针，是指指向“垃圾”内存的指针。即指针指向的内容是不确定的。

       产生的原因：1）指针变量没有初始化。因此，创建指针变量时，该变量要被置为NULL或者指向合法的内存单元。

                             2）指针p被free之后，没有置为NULL，让人误以为p是个合法的指针。

                             3）指针跨越合法范围操作。不要返回指向栈内存的指针或引用

例子1-1：引用尚未初始化的指针

**[cpp]** [view plain](https://blog.csdn.net/c_base_jin/article/details/26149653) [copy](https://blog.csdn.net/c_base_jin/article/details/26149653)

1. **char** \*p;
2. \*p = 'A';//error，p指向未定义

例子1-2：return语句返回指向“栈内存”的指针

**[cpp]** [view plain](https://blog.csdn.net/c_base_jin/article/details/26149653) [copy](https://blog.csdn.net/c_base_jin/article/details/26149653)

1. **char** \*GetString1(**void**)
2. {
3. **char** p[] = "hello world!";
4. //p在栈区，常量字符串在常量字符区
6. **return** p;//error，返回栈内存的地址
7. }

例子1-3：使用了被释放的内存

**[cpp]** [view plain](https://blog.csdn.net/c_base_jin/article/details/26149653) [copy](https://blog.csdn.net/c_base_jin/article/details/26149653)

1. **char** \*pstr = (**char** \*)malloc(**sizeof**(**char**)\*100);
2. free(pstr); //pstr所指的内存被释放
3. **if** (NULL !=pstr)//没起到作用
4. {
5. strcpy(pstr,"string!");//error，有时候程序不会提示有误，但还是不允许
6. }

注意：free()释放的是指针指向的内存！不是指针变量！这点非常非常重要！指针是一个变量，只有程序结束时才被销毁。释放了内存空间后，原来指向这块空间的指针还是存在！只不过现在指针指向的内容的垃圾，是未定义的，所以说是垃圾。因此，前面我已经说过了，释放内存后把指针指向NULL，防止指针在后面不小心又被解引用。

对比下面的例子，加深理解

例子1-4：函数返回值传递动态内存

**[cpp]** [view plain](https://blog.csdn.net/c_base_jin/article/details/26149653) [copy](https://blog.csdn.net/c_base_jin/article/details/26149653)

1. **char**\* GetMemory(**int** num)
2. {
3. **char** \*p = (**char** \*)malloc(**sizeof**(**char**) \* num);
4. **return** p ;//ok,返回堆区的地址值
5. }

例子1-5：

**[cpp]** [view plain](https://blog.csdn.net/c_base_jin/article/details/26149653) [copy](https://blog.csdn.net/c_base_jin/article/details/26149653)

1. **char** \*GetString(**void**)
2. {
3. **char** \*p = "hello world!";
4. //指针变量p在栈区，指向文字常量区的字符
6. **return** p;//ok，返回字符串的地址
7. }

## 2.内存泄漏

    概念：用动态内存分配函数动态开辟的空间，在使用完毕后未释放，结果导致一直占据该内存单元，直到程序结束。

注意：内存泄漏是指堆内存的泄漏。它的一般表现方式是程序运行时间越长，占用内存越多，最终用尽全部内存，整个系统崩溃。

例子2-1：内存泄漏，共9\*100字节发生泄漏

**[cpp]** [view plain](https://blog.csdn.net/c_base_jin/article/details/26149653) [copy](https://blog.csdn.net/c_base_jin/article/details/26149653)

1. **void** Test(**void**)
2. {
3. **char** \*p = NULL;
4. **for** ( **int** i = 0; i<10; i++)
5. {
6. p = (**char**\*)malloc(100);//没循环一次内存泄漏一块，最后一次得到正确使用
7. }
9. strncpy(p,"string!");
10. free(p);
11. }

## 3.内存溢出

   概念：系统分配的内存不足以放下数据,称为内存溢出。

例子3-1：运行时提示出错

**[cpp]** [view plain](https://blog.csdn.net/c_base_jin/article/details/26149653) [copy](https://blog.csdn.net/c_base_jin/article/details/26149653)

1. **char** str[10]={0};
3. strcpy(str,"hello world!");//error！

版权声明：本文为博主原创文章，未经博主允许不得转载。 https://blog.csdn.net/xiao3404/article/details/26149653