C 编程指南学习(七) PDF转换可能丢失图片或格式，建议阅读原文
https://www.100test.com/kao_ti2020/449/2021_2022_C___E7_BC_96_E7_A8_8B_E6_c97_449182.htm 第7章 内存管理 欢迎进入内存这片雷区。伟大的Bill Gates 曾经失言： 640K ought to be enough for everybody Bill Gates 1981 程序员们经常编写内存管理程序，往往提心吊胆。如果不想触雷，唯一的解决办法就是发现所有潜伏的地雷并且排除它们，躲是躲不了的。本章的内容比一般教科书的要深入得多，读者需细心阅读，做到真正地通晓内存管理。 7.1内存分配方式 内存分配方式有三种：（1） 从静态存储区域分配。内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在。例如全局变量，static变量。 （2） 在栈上创建。在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。 （3） 从堆上分配，亦称动态内存分配。程序在运行的时候用malloc或new申请任意多少的内存，程序员自己负责在何时用free或0delete释放内存。动态内存的生存期由我们决定，使用非常灵活，但问题也最多。 7.2常见的内存错误及其对策 发生内存错误是件非常麻烦的事情。编译器不能自动发现这些错误，通常是在程序运行时才能捕捉到。而这些错误大多没有明显的症状，时隐时现，增加了改错的难度。有时用户怒气冲冲地把你找来，程序却没有发生任何问题，你一走，错误又发作了。 常见的内存错误及其对策如下： u 内存分配未成功，却使用了它。 编程新手常犯这种错误，因为他们没有意识

到内存分配会不成功。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(p!=NULL)进行检查。如果是用malloc或new来申请内存，应该用if(p==NULL) 或if(p!=NULL)进行防错处理。 u 内存分配虽然成功，但是尚未初始化就引用它。 犯这种错误主要有两个起因：一是没有初始化的观念；二是误以为内存的缺省初值全为零，导致引用初值错误（例如数组）。 内存的缺省初值究竟是什么并没有统一的标准，尽管有些时候为零值，我们宁可信其无不可信其有。所以无论用何种方式创建数组，都别忘了赋初值，即便是赋零值也不可省略，不要嫌麻烦。 u 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。 u 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/0delete同理）。 u 释放了内存却继续使用它。 有三种情况： （1）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。 （2）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。 （3）使用free或0delete释放了内存后，没有将指针设置为NULL。导致产生“ 野指针 ”。 l 【规则7-2-1】

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。l【规则7-2-2】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。l【规则7-2-3】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。l【规则7-2-4】动态内存的申请与释放必须配对，防止内存泄漏。l【规则7-2-5】用free或0delete释放了内存之后，立即将指针设置为NULL，防止产生“野指针”。 7.3指针与数组的对比 C/C程序中，指针和数组在不少地方可以相互替换着用，让人产生一种错觉，以为两者是等价的。 数组要么在静态存储区被创建（如全局数组），要么在栈上被创建。数组名对应着（而不是指向）一块内存，其地址与容量在生命期内保持不变，只有数组的内容可以改变。 指针可以随时指向任意类型的内存块，它的特征是“可变”，所以我们常用指针来操作动态内存。指针远比数组灵活，但也更危险。 下面以字符串为例比较指针与数组的特性。 7.3.1 修改内容 示例7-3-1中，字符数组a的容量是6个字符，其内容为hello\0。a的内容可以改变，如a[0]=‘X’。指针p指向常量字符串“world”（位于静态存储区，内容为world\0），常量字符串的内容是不可以被修改的。从语法上看，编译器并不觉得语句p[0]=‘X’有什么不妥，但是该语句企图修改常量字符串的内容而导致运行错误。 char a[] =“hello”. a[0] =‘X’. cout〈〈a〈〈endl. char *p =“world”. // 注意p指向常量字符串p[0] =‘X’. // 编译器不能发现该错误 cout〈〈p〈〈endl. 示例7-3-1 修改数组和指针的内容 7.3.2 内容复制与比较 不能对数组名进行直接复制与比较。示例7-3-2中，若想把数组a的内容复制给

数组b，不能用语句 b = a ，否则将产生编译错误。应该用标准库函数strcpy进行复制。同理，比较b和a的内容是否相同，不能用if(b==a) 来判断，应该用标准库函数strcmp进行比较。语句p = a 并不能把a的内容复制指针p，而是把a的地址赋给了p。要想复制a的内容，可以先用库函数malloc为p申请一块容量为strlen(a) 1个字符的内存，再用strcpy进行字符串复制。同理，语句if(p==a) 比较的不是内容而是地址，应该用库函数strcmp来比较。 // 数组… char a[] = "hello". char b[10]. strcpy(b, a). // 不能用 b = a. if(strcmp(b, a) == 0) // 不能用 if (b == a) … // 指针… int len = strlen(a). char *p = (char *)malloc(sizeof(char)*(len 1)). strcpy(p,a). // 不要用 p = a. if(strcmp(p, a) == 0) // 不要用 if (p == a) … 示例7-3-2 数组和指针的内容复制与比较 7.3.3 计算内存容量 用运算符sizeof可以计算出数组的容量（字节数）。示例7-3-3（a）中，sizeof(a)的值是12（注意别忘了 ’ \0 ’ ）。指针p指向a，但是sizeof(p)的值却是4。这是因为sizeof(p)得到的是一个指针变量的字节数，相当于sizeof(char*)，而不是p所指的内存容量。C /C语言没有办法知道指针所指的内存容量，除非在申请内存时记住它。 注意当数组作为函数的参数进行传递时，该数组自动退化为同类型的指针。示例7-3-3（b）中，不论数组a的容量是多少，sizeof(a)始终等于sizeof(char *)。 char a[] = "hello world". char *p = a. cout 〈〈 sizeof(a) 〈〈 endl. // 12字节 cout 〈〈 sizeof(p) 〈〈 endl. // 4字节 示例7-3-3（a） 计算数组和指针的内存容量 void Func(char a[100]) { cout 〈〈 sizeof(a) 〈〈 endl. // 4字节而不是100字节 } 示例7-3-3（b） 数组退化为指针 7.4 指针参数是如何传递内存的？ 如果函数的参数是一个指针，

不要指望用该指针去申请动态内存。示例7-4-1中，Test函数的语句GetMemory(str, 200)并没有使str获得期望的内存，str依旧是NULL，为什么？ void GetMemory(char *p, int num) { p = (char *)malloc(sizeof(char) * num). } void Test(void) { char *str = NULL. GetMemory(str, 100). // str 仍然为 NULL strcpy(str, "hello"). // 运行错误 } 示例7-4-1 试图用指针参数申请动态内存毛病出在函数GetMemory中。编译器总是要为函数的每个参数制作临时副本，指针参数p的副本是 _p，编译器使 _p = p。如果函数体内的程序修改了_p的内容，就导致参数p的内容作相应的修改。这就是指针可以用作输出参数的原因。在本例中，_p申请了新的内存，只是把_p所指的内存地址改变了，但是p丝毫未变。所以函数GetMemory并不能输出任何东西。事实上，每执行一次GetMemory就会泄露一块内存，因为没有用free释放内存。 如果非得要用指针参数去申请内存，那么应该改用“ 指向指针的指针 ”，见示例7-4-2。 void GetMemory2(char **p, int num) { *p = (char *)malloc(sizeof(char) * num). } void Test2(void) { char *str = NULL. GetMemory2(amp.str，而不是str strcpy(str, "hello"). cout 〈〈str 〈〈endl. free(str). } 示例7-4-2用指向指针的指针申请动态内存由于“ 指向指针的指针 ”这个概念不容易理解，我们可以用函数返回值来传递动态内存。这种方法更加简单，见示例7-4-3。 char *GetMemory3(int num) { char *p = (char *)malloc(sizeof(char) * num). return p. } void Test3(void) { char *str = NULL. str = GetMemory3(100). strcpy(str, "hello"). cout 〈〈str 〈〈endl. free(str). } 示例7-4-3 用函数返回值来传递动态内存 用函数返回值来传递动态内存这种方法虽然好用，但是常

常有人把return语句用错了。这里强调不要用return语句返回指向“ 栈内存 ”的指针，因为该内存在函数结束时自动消亡，见示例7-4-4。char *GetString(void) { char p[] = "hello world". return p. // 编译器将提出警告 } void Test4(void) { char *str = NULL. 100Test 下载频道开通，各类考试题目直接下载。详细请访问 www.100test.com