C 语言常见问题集

原著：Steve Summit
翻译：朱群英, 孙 云

修订版 0.9.4, 2005 年 6 月 23 日

目录

目录 **i**

前言 **xvii**

1 声明和初始化 **1**

1.1 我如何决定使用那种整数类型？ 1
1.2 64 位机上的 64 位类型是什么样的？ 1
1.3 怎样定义和声明全局变量和函数最好？ 2
1.4 extern 在函数声明中是什么意思？ 2
1.5 关键字 auto 到底有什么用途？ 2
1.6 我似乎不能成功定义一个链表。我试过 typedef struct { char *item; NODEPTR next; } *NODEPTR; 但是编译器报了错误信息。难道在C语言中一个结构不能包含指向自己的指针吗？ 3
1.7 怎样建立和理解非常复杂的声明？例如定义一个包含 N 个指向返回指向字符的指针的函数的指针的数组？ 3
1.8 函数只定义了一次, 调用了一次, 但编译器提示非法重定义了。 . . 4
1.9 main() 的正确定义是什么？ void main() 正确吗？ 4
1.10 对于没有初始化的变量的初始值可以作怎样的假定？如果一个全局变量初始值为 “零”, 它可否作为空指针或浮点零？ 4
1.11 代码 int f() { char a[] = "Hello, world!";} 不能编译。 5
1.12 这样的初始化有什么问题？char *p = malloc(10); 编译器提示 “非法初始式” 云云。 5
1.13 以下的初始化有什么区别？char a[] = "string literal"; char *p = "string literal"; 当我向 p[i] 赋值的时候, 我的程序崩溃了。 5
1.14 我总算弄清除函数指针的声明方法了, 但怎样才能初始化呢？ . . 5

2 结构、联合和枚举 **7**

2.1 声明 struct x1 { . . . }; 和 typedef struct { . . . } x2; 有什么不同？ . 7
2.2 为什么 struct x { . . . }; x thestruct; 不对？ 7
2.3 一个结构可以包含指向自己的指针吗？ 7
2.4 在 C 语言中实现抽象数据类型什么方法最好? 7
2.5 在 C 中是否有模拟继承等面向对象程序设计特性的好方法？ . . . 7

2.6 我遇到这样声明结构的代码: struct name { int namelen; char namestr[1];}; 然后又使用一些内存分配技巧使 namestr 数组用起来好像有多个元素。这样合法和可移植吗？ 8
2.7 是否有自动比较结构的方法？ 8
2.8 如何向接受结构参数的函数传入常数值？ 8
2.9 怎样从/向数据文件读/写结构？ 9
2.10 我的编译器在结构中留下了空洞, 这导致空间浪费而且无法与外部数据文件进行 "二进制" 读写。能否关掉填充, 或者控制结构域的对齐方式? 9
2.11 为什么 sizeof 返回的值大于结构的期望值, 是不是尾部有填充? . . 9
2.12 如何确定域在结构中的字节偏移？ 9
2.13 怎样在运行时用名字访问结构中的域？ 10
2.14 程序运行正确, 但退出时却 "core dump"了，怎么回事？ 10
2.15 可以初始化一个联合吗？ 10
2.16 枚举和一组预处理的 #define 有什么不同？ 10
2.17 有什么容易的显示枚举值符号的方法？ 11

3 表达式 13
3.1 为什么这样的代码: a[i] = i++; 不能工作？ 13
3.2 使用我的编译器,下面的代码 int i=7; printf("%d\n", i++ * i++); 返回 49？不管按什么顺序计算, 难道不该打印出56吗？ 13
3.3 对于代码 int i = 3; i = i++; 不同编译器给出不同的结果, 有的为 3, 有的为 4, 哪个是正确的？ 14
3.4 这是个巧妙的表达式: a ^= b ^= a ^= b 它不需要临时变量就可以交换 a 和 b 的值。 14
3.5 我可否用括号来强制执行我所需要的计算顺序？ 14
3.6 可是 && 和 || 运算符呢？我看到过类似 while((c = getchar()) != EOF && c != '\n') 的代码 …… 14
3.7 我怎样才能理解复杂表达式？"序列点" 是什么？ 15
3.8 那么, 对于 a[i] = i++; 我们不知道 a[] 的哪一个分量会被改写,但 i 的确会增加 1, 对吗？ 15
3.9 ++i 和 i++ 有什么区别？ 15
3.10 如果我不使用表达式的值, 我应该用 ++i 或 i++ 来自增一个变量吗？ 15
3.11 为什么如下的代码 int a = 100, b = 100; long int c = a * b; 不能工作？ 15
3.12 我需要根据条件把一个复杂的表达式赋值给两个变量中的一个。可以用下边这样的代码吗？ ((condition) ? a : b) = complicated_expression; 16

4 指针 **17**

4.1 我想声明一个指针并为它分配一些空间, 但却不行。这些代码有什么问题? char *p; *p = malloc(10); 17
4.2 *p++ 自增 p 还是 p 所指向的变量? 17
4.3 我有一个 char * 型指针正巧指向一些 int 型变量, 我想跳过它们。为什么如下的代码((int *)p)++; 不行? 17
4.4 我有个函数,它应该接受并初始化一个指针 void f(int *ip) { static int dummy = 5; ip = &dummy;} 但是当我如下调用时: int *ip; f(ip); 调用者的指针却没有任何变化。 18
4.5 我能否用 void** 指针作为参数, 使函数按引用接受一般指针? . . 18
4.6 我有一个函数 extern int f(int *); 它接受指向 int 型的指针。我怎样用引用方式传入一个常数? 下面这样的调用 f(&5); 似乎不行。 . 18
4.7 C 有 “按引用传递” 吗? 18
4.8 我看到了用指针调用函数的不同语法形式。到底怎么回事? . . . 19
4.9 我怎样把一个 int 变量转换为 char * 型? 我试了类型转换, 但是不行。 19

5 空 (null) 指针 **21**

5.1 臭名昭著的空指针到底是什么? 21
5.2 怎样在程序里获得一个空指针? 21
5.3 用缩写的指针比较 “if(p)” 检查空指针是否可靠? 如果空指针的内部表达不是 0 会怎么样? 22
5.4 NULL 是什么, 它是怎么定义的? 23
5.5 在使用非全零作为空指针内部表达的机器上, NULL 是如何定义的? 23
5.6 如果 NULL 定义成 #define NULL ((char *)0) 难道不就可以向函数传入不加转换的 NULL 了吗? 23
5.7 如果 NULL 和 0 作为空指针常数是等价的, 那我到底该用哪一个呢? 24
5.8 但是如果 NULL 的值改变了, 比如在使用非零内部空指针的机器上, 难道用 NULL (而不是 0) 不是更好吗? 24
5.9 用预定义宏 #define Nullptr(type) (type *)0 帮助创建正确类型的空指针。 24
5.10 这有点奇怪。NULL 可以确保是 0, 但空 (null) 指针却不一定? . . 24
5.11 为什么有那么多关于空指针的疑惑? 为什么这些问题如此经常地出现? 25
5.12 我很困惑。我就是不能理解这些空指针一类的东西。 25
5.13 考虑到有关空指针的所有这些困惑, 难道把要求它们内部表达都必须为 0 不是更简单吗? 26
5.14 说真的, 真有机器用非零空指针吗, 或者不同类型用不同的表达? 26

5.15 运行时的 “空指针赋值” 错误是什么意思？ 26

6 数组和指针 27

6.1 我在一个源文件中定义了 char a[6], 在另一个中声明了 extern char *a 。为什么不行？ 27
6.2 可是我听说 char a[ ] 和 char *a 是一样的。 27
6.3 那么, 在 C 语言中 “指针和数组等价” 到底是什么意思？ 28
6.4 那么为什么作为函数形参的数组和指针申明可以互换呢？ 28
6.5 如果你不能给它赋值, 那么数组如何能成为左值呢？ 29
6.6 现实地讲, 数组和指针地区别是什么？ 29
6.7 有人跟我讲, 数组不过是常指针。 29
6.8 我遇到一些 “搞笑” 的代码, 包含 5["abcdef"] 这样的 “表达式”。这为什么是合法的 C 表达式呢？ 29
6.9 既然数组引用会蜕化为指针, 如果 arr 是数组, 那么 arr 和 &arr 又有什么区别呢？ 30
6.10 我如何声明一个数组指针？ 30
6.11 我如何在运行期设定数组的大小？我怎样才能避免固定大小的数组？ 30
6.12 我如何声明大小和传入的数组一样的局部数组？ 30
6.13 我该如何动态分配多维数组？ 31
6.14 有个灵巧的窍门: 如果我这样写 int realarray[10]; int *array = &realarray[-1]; 我就可以把 “array” 当作下标从 1 开始的数组。 . . 32
6.15 当我向一个接受指针的指针的函数传入二维数组的时候, 编译器报错了。 32
6.16 我怎样编写接受编译时宽度未知的二维数组的函数？ 32
6.17 我怎样在函数参数传递时混用静态和动态多维数组？ 33
6.18 当数组是函数的参数时, 为什么 sizeof 不能正确报告数组的大小？ 34

7 内存分配 35

7.1 为什么这段代码不行？ char *answer; printf("Type something:\n"); gets(answer); printf("You typed \"%s\"\n", answer); 35
7.2 我的 strcat() 不行.我试了 char *s1 = "Hello, "; char *s2 = "world!"; char *s3 = strcat(s1, s2); 但是我得到了奇怪的结果。 35
7.3 但是 strcat 的手册页说它接受两个 char * 型参数。我怎么知道 (空间) 分配的事情呢？ 36
7.4 我刚才试了这样的代码 char *p; strcpy(p, "abc"); 而它运行正常？怎么回事？为什么它没有崩溃？ 36
7.5 一个指针变量分配多少内存？ 36
7.6 我有个函数, 本该返回一个字符串, 但当它返回调用者的时候, 返回串却是垃圾信息。 36

7.7 那么返回字符串或其它集合的争取方法是什么呢？ 37
7.8 为什么在调用 malloc() 时，我得到 “警告：整数赋向指针需要类型转换”？ 37
7.9 为什么有些代码小心地把 malloc 返回的值转换为分配的指针类型。 37
7.10 在调用 malloc() 的时候，错误 “不能把 void * 转换为 int *” 是什么意思？ 37
7.11 我见到了这样的代码 char *p = malloc(strlen(s) + 1); strcpy(p, s); 难道不应该是 malloc((strlen(s) + 1) * sizeof(char))？ 37
7.12 我如何动态分配数组？ 38
7.13 我听说有的操作系统程序使用的时候才真正分配 malloc 申请的内存。这合法吗？ 38
7.14 我用一行这样的代码分配一个巨大的数组，用于数字运算：double *array = malloc(300 * 300 * sizeof(double)); malloc() 并没有返回 null，但是程序运行得有些奇怪，好像改写了某些内存，或者 malloc() 并没有分配我申请的那么多内存，云云。 38
7.15 我的 PC 有 8 兆内存。为什么我只能分配 640K 左右的内存？ . . 38
7.16 我的程序总是崩溃，显然在 malloc 内部的某个地方。但是我看不出哪里有问题。是 malloc() 有 bug 吗？ 38
7.17 动态分配的内存一旦释放之后你就不能再使用，是吧？ 38
7.18 为什么在调用 free() 之后指针没有变空？使用 (赋值，比较) 释放之后的指针有多么不安全？ 39
7.19 当我 malloc() 为一个函数的局部指针分配内存时，我还需要用 free() 明确的释放吗？ 39
7.20 我在分配一些结构，它们包含指向其它动态分配的对象的指针。我在释放结构的时候，还需要释放每一个下级指针吗？ 39
7.21 我必须在程序退出之前释放分配的所有内存吗？ 40
7.22 我有个程序分配了大量的内存，然后又释放了。但是从操作系统看，内存的占用率却并没有回去。 40
7.23 free() 怎么知道有多少字节需要释放？ 40
7.24 那么我能否查询 malloc 包，可分配的最大块是多大？ 40
7.25 向 realloc() 的第一个参数传入空指针合法吗？你为什么要这样做？ 40
7.26 calloc() 和 malloc() 有什么区别？利用 calloc 的零填充功能安全吗？free() 可以释放 calloc() 分配的内存吗，还是需要一个 cfree()？ 40
7.27 alloca() 是什么？为什么不提倡使用它？ 41

8 字符和字符串 **43**
8.1 为什么 strcat(string, '!'); 不行？ 43
8.2 我在检查一个字符串是否跟某个值匹配。为什么这样不行？char *string; ...if(string == "value") { /* string matches "value" */ ...} 43

8.3 如果我可以写 char a[] = "Hello, world!"; 为什么我不能写 char a[14]; a = "Hello, world!"; 43
8.4 我怎么得到对应字符的数字 (字符集) 值, 或者相反? 44
8.5 我认为我的编译器有问题: 我注意到 sizeof('a') 是 2 而不是 1 (即, 不是 sizeof(char))。 44

9 布尔表达式和变量 45
9.1 C 语言中布尔值的候选类型是什么? 为什么它不是一个标准类型? 我应该用 #define 或 enum 定义 true 和 false 值吗? 45
9.2 因为在 C 语言中所有的非零值都被看作 "真", 是不是把 TRUE 定义为 1 很危险? 如果某个内置的函数或关系操作符 "返回" 不是 1 的其它值怎么办? 45
9.3 当 p 是指针时, if(p) 是合法的表达式吗? 46

10 C 预处理器 47
10.1 这些机巧的预处理宏: #define begin { #define end } 你觉得怎么样? 47
10.2 怎么写一个一般用途的宏交换两个值? 47
10.3 书写多语句宏的最好方法是什么? 47
10.4 我第一次把一个程序分成多个源文件, 我不知道该把什么放到 .c 文件, 把什么放到 .h 文件。(".h" 到底是什么意思?) 48
10.5 一个头文件可以包含另一头文件吗? 48
10.6 #include <> 和 #include "" 有什么区别? 48
10.7 完整的头文件搜索规则是怎样的? 49
10.8 我在文件的第一个声明就遇到奇怪的语法错误, 但是看上去没什么问题。 49
10.9 我包含了我使用的库函数的正确头文件, 可是连接器还是说它没有定义。 49
10.10 我在编译一个程序, 看起来我好像缺少需要的一个或多个头文件。谁能发给我一份? 49
10.11 我怎样构造比较字符串的 #if 预处理表达式? 49
10.12 sizeof 操作符可以用于 #if 预编译指令中吗? 50
10.13 我可以在 #include 行里使用 #ifdef 来定义两个不同的东西吗? . 50
10.14 对 typdef 的类型定义有没有类似 #ifdef的东西? 50
10.15 我如何用 #if 表达式来判断机器是高字节在前还是低字节在前? . 50
10.16 我得到了一些代码, 里边有太多的 #ifdef。我不想使用预处理器把所有的 #include 和 #ifdef 都扩展开, 有什么办法只保留一种条件的代码呢? 50
10.17 如何列出所有的预定义标识符? 50

10.18 我有些旧代码, 试图用这样的宏来构造标识符 #define Paste(a, b) a/**/b 但是现在不行了。 51
10.19 为什么宏 #define TRACE(n) printf("TRACE: %d\n", n) 报出警告 “用字符串常量代替宏”? 它似乎应该把 TRACE(count); 扩展为 printf("TRACE: %d\count", count); 51
10.20 使用 # 操作符时, 我在字符串常量内使用宏参数有问题。 51
10.21 我想用预处理做某件事情, 但却不知道如何下手。 51
10.22 怎样写参数个数可变的宏? 51

11 ANSI/ISO 标准 C **53**
11.1 什么是 “ANSI C 标准”? 53
11.2 我如何得到一份标准的副本? 53
11.3 我在哪里可以找到标准的更新? 54
11.4 很多 ANSI 编译器在遇到以下代码时都会警告类型不匹配。 extern int func(float); int func(x) float x; { 54
11.5 能否混用旧式的和新型的函数语法? 55
11.6 为什么声明 extern int f(struct x *p); 报出了一个奇怪的警告信息 “结构 x 在参数列表中声明”? 55
11.7 我不明白为什么我不能象这样在初始化和数组维度中使用常量: const int n = 5; int a[n]; 55
11.8 既然不能修改字符串常量, 为什么不把它们定义为字符常量的数组? 55
11.9 “const char *p” 和 “char * const p” 有何区别? 56
11.10 为什么我不能向接受 const char ** 的函数传入 char **? 56
11.11 怎样正确声明 main()? 56
11.12 我能否把 main() 定义为 void, 以避免扰人的 “main无返回值” 警告? 56
11.13 可 main() 的第三个参数 envp 是怎么回事? 57
11.14 我觉得把 main() 声明为 void 不会失败, 因为我调用了 exit() 而不是 return , 况且我的操作系统也忽略了程序的退出/返回状态。 . . 57
11.15 那么到底会出什么问题? 真的有什么系统不支持 void main() 吗? 57
11.16 我一直用的那本书《熟练傻瓜C语言》总是使用 void main()。 . . 57
11.17 从 main() 中, exit(status) 和返回同样的 status 真的等价吗? . . . 57
11.18 我试图用 ANSI “字符串化” 预处理操作符 # 向信息中插入符号常量的值, 但它字符串化的总是宏的名字而不是它的值。 58
11.19 警告信息 “warning: macro replacement within a string literal” 是什么意思? 58
11.20 在我用 #ifdef 去掉的代码里出现了奇怪的语法错误。 58
11.21 #pragma 是什么, 有什么用? 59
11.22 “#pragma once” 是什么意思? 我在一些头文件中看到了它。 . . 59

11.23 a[3] = "abc"; 合法吗？它是什么意思？ 59
11.24 为什么我不能对 void* 指针进行运算？ 59
11.25 memcpy() 和 memmove() 有什么区别？ 59
11.26 malloc(0) 有什么用？返回一个控指针还是指向 0 字节的指针？ . 59
11.27 为什么 ANSI 标准规定了外部标示符的长度和大小写限制？ . . . 60
11.28 我的编译对最简单的测试程序报出了一大堆的语法错误。 60
11.29 为什么有些 ASNI/ISO 标准库函数未定义？我明明使用的就是 ANSI 编译器。 60
11.30 谁有把旧的 C 程序转化为 ANSI C 或相反的工具, 或者自动生成原型的工具？ 60
11.31 为什么声称兼容 ANSI 的 Frobozz Magic C 编译器不能编译这些代码？我知道这些代码是 ANSI 的, 因为 gcc 可以编译。 60
11.32 人们好像有些在意实现定义 (implementation-defin-ed)、未明确 (unspecified) 和无定义 (undefined) 行为的区别。它们的区别到底在哪里？ 61
11.33 一个程序的 “合法”, “有效” 或 “符合” 到底是什么意思？ 61
11.34 我很吃惊, ANSI 标准竟然有那么多没有定义的东西。标准的唯一任务不就是让这些东西标准化吗？ 61
11.35 有人说 i = i++ 的行为是未定义的, 但是我刚在一个兼容 ANSI 的编译器上测试, 得到了我希望的结果。 62

12 标准输入输出库 **63**
12.1 这样的代码有什么问题？char c; while((c = getchar()) != EOF) ... 63
12.2 我有个读取直到 EOF 的简单程序, 但是我如何才能在键盘上输入那个 “EOF” 呢？ 63
12.3 为什么这些代码 while(!feof(infp)) { fgets(buf, MAXLINE, infp); fputs(buf, outfp); } 把最后一行复制了两遍？ 63
12.4 我的程序的屏幕提示和中间输出有时显示在屏幕上, 尤其是当我用管道向另一个程序输出的时候。 63
12.5 我怎样不等待回车键一次输入一个字符？ 64
12.6 我如何在 printf 的格式串中输出一个 ’%’？我试过 \%, 但是不行。 64
12.7 有人告诉我在 printf 中使用 %lf 不正确。那么, 如果 scanf() 需要 %lf, 怎么可以用在 printf() 中用 %f 输出双精度数呢？ 64
12.8 对于 size_t 那样的类型定义, 当我不知道它到底是 long 还是其它类型的时候, 我应该使用什么样的 printf 格式呢？ 64
12.9 我如何用 printf 实现可变的域宽度？就是说, 我想在运行时确定宽度而不是使用 %8d？ 64
12.10 如何输出在千位上用逗号隔开的数字？金额数字呢？ 65
12.11 为什么 scanf("%d", i) 调用不行？ 65
12.12 为什么 char s[30]; scanf("%s", s); 不用 & 也可以？ 65

12.13 为什么这些代码 double d; scanf("%f", &d); 不行？ 65
12.14 怎样在 scanf() 格式串中指定可变的宽度？ 65
12.15 当我用 “%d\n” 调用 scanf 从键盘读取数字的时候, 好像要多输入一行函数才返回。 65
12.16 我用 scanf %d 读取一个数字, 然后再用 gets() 读取字符串, 但是编译器好像跳过了 gets() 调用！ 66
12.17 我发现如果坚持检查返回值以确保用户输入的是我期待的数值, 则 scanf() 的使用会安全很多, 但有的时候好像会陷入无限循环。 . 66
12.18 为什么大家都说不要使用 scanf()？那我该用什么来代替呢？ . . . 66
12.19 我怎样才知道对于任意的 sprintf 调用需要多大的目标缓冲区？怎样才能避免 sprintf() 目标缓冲区溢出？ 66
12.20 为什么大家都说不要使用 gets()？ 67
12.21 为什么调用 printf() 之后 errno 内有 ENOTTY？ 67
12.22 fgetops/fsetops 和 ftell/fseek 之间有什么区别？ fgetops() 和 fsetops() 到底有什么用处？ 68
12.23 如何清除多余的输入, 以防止在下一个提示符下读入？fflush(stdin) 可以吗？ 68
12.24 既然 fflush() 不能, 那么怎样才能清除输入呢？ 68
12.25 对某些路径文件名调用 fopen() 总是失败。 68
12.26 我想用 “r+” 打开一个文件, 读出一个字符串, 修改之后再写入, 从而就地更新一个文件。可是这样不行。 69
12.27 怎样在程序里把 stdin 或 stdout 重定向到文件？ 69
12.28 一旦使用 freopen() 之后, 怎样才能恢复原来的 stdout (或 stdin)？ 69
12.29 怎样同时向两个地方输出, 如同时输出到屏幕和文件？ 69
12.30 怎样正确的读取二进制文件？我有时看到 0x0a 和 0x0d 混淆了, 而且如果数据中包含 0x1a 的话, 我好像会提前遇到 EOF。 70

13 库函数 **71**
13.1 怎样把数字转为字符串 (与 atoi 相反)？有 itoa() 函数吗？ 71
13.2 为什么 strncpy() 不能总在目标串放上终止符 ’\0’？ 71
13.3 为什么有些版本的 toupper() 对大写字符会有奇怪的反应？为什么有的代码在调用 toupper() 前先调用 tolower()？ 71
13.4 怎样把字符串分隔成用空白作间隔符的段？怎样实现类似传递给 main() 的 argc 和 argv？ 72
13.5 我需要一些处理正则表达式或通配符匹配的代码。 72
13.6 我想用 strcmp() 作为比较函数, 调用 qsort() 对一个字符串数组排序, 但是不行。 72
13.7 我想用 qsort() 对一个结构数组排序。我的比较函数接受结构指针, 但是编译器认为这个函数对于 qsort() 是错误类型。我要怎样转换这个函数指针才能避免这样的警告？ 73

13.8 怎样对一个链表排序？ 73
13.9 怎样对多于内存的数据排序？ 73
13.10 怎样在 C 程序中取得当前日期或时间？ 73
13.11 我知道库函数 localtime() 可以把 time_t 转换成结构 struct tm, 而 ctime() 可以把 time_t 转换成为可打印的字符串。怎样才能进行反向操作, 把 struct tm 或一个字符串转换成 time_t？ 74
13.12 怎样在日期上加 N 天？怎样取得两个日期的时间间隔？ 74
13.13 我需要一个随机数生成器。 75
13.14 怎样获得在一定范围内的随机数？ 75
13.15 每次执行程序, rand() 都返回相同顺序的数字。 75
13.16 我需要随机的真/假值, 所以我用直接用 rand() % 2, 可是我得到交替的 0, 1, 0, 1, 0 …… 76
13.17 怎样产生标准分布或高斯分布的随机数？ 76
13.18 我不断得到库函数未定义错误, 但是我已经 #inlude 了所有用到的头文件了。 77
13.19 虽然我在连接时明确地指定了正确的函数库, 我还是得到库函数未定义错误。 77
13.20 连接器说 _end 未定义代表什么意思？ 77
13.21 我的编译器提示 printf 未定义！这怎么可能？ 77

14 浮点运算 **79**
14.1 一个 float 变量赋值为 3.1 时, 为什么 printf 输出的值为 3.0999999？ 79
14.2 执行一些开方根运算, 可是得到一些疯狂的数字。 79
14.3 做一些简单的三角函数运算, 也引用了 #include <math.h>, 可是一直得到编译错误 “undefined: sin” (函数 sin 未定义)。 79
14.4 浮点计算程序表现奇怪, 在不同的机器上给出不同的结果。 79
14.5 有什么好的方法来验对浮点数在 “足够接近” 情况下的等值？ . . . 80
14.6 怎样取整数？ 80
14.7 为什么 C 不提供乘幂的运算符？ 80
14.8 为什么我机器上的 <math.h> 没有预定义常数 M_PI？ 80
14.9 怎样测试 IEEE NaN 以及其它特殊值？ 81
14.10 在 C 中如何很好的实现复数？ 81
14.11 我要寻找一些实现以下功能的程序源代码：快速傅立叶变换 (FFT)、矩阵算术 (乘法、倒置等函数)、复数算术。 81
14.12 Turbo C 的程序崩溃, 显示错误为 “floating point formats not linked” (浮点格式未连接)。 81

15 可变参数 **83**
15.1 为什么调用 printf() 前, 必须要用 #include <stdio.h>？ 83

15.2 为什么 %f 可以在 printf() 参数中, 同时表示 float 和 double? 他们难道不是不同类型吗? 83
15.3 为什么当 n 为 long int, printf("%d", n); 编译时没有匹配警告? 我以为 ANSI 函数原型可以防止这样的类型不匹配。 83
15.4 怎样写一个有可变参数的函数? 83
15.5 怎样写类似 printf() 的函数, 再把参数转传给 printf() 去完成大部分工作? 85
15.6 怎样写类似 scanf() 的函数, 再把参数转传给 scanf() 去完成大部分工作? 85
15.7 怎样知道实际上有多少个参数传入函数? 85
15.8 为什么编译器不让我定义一个没有固定参数项的可变参数函数? . 86
15.9 我有个接受 float 的可变参函数, 为什么 va_arg(argp, float) 不工作? 86
15.10 va_arg() 不能得到类型为函数指针的参数。 86
15.11 怎样实现一个可变参数函数, 它把参数再传给另一个可变参数函数? 86
15.12 怎样调用一个参数在执行是才建立的函数? 87

16 奇怪的问题 **89**
16.1 遇到不可理解的不合理语法错误, 似乎大段的程序没有编译。 . . 89
16.2 为什么过程调用不工作? 编译器似乎直接跳过去了。 89
16.3 程序在执行用之前就崩溃了, 用调试器单步跟进, 在 main() 之前就死了。 89
16.4 程序执行正确, 但退出时崩溃在 main() 最后一个语句之后。为什么会这样? 89
16.5 程序在一台机器上执行完美, 但在另一台上却得到怪异的结果。更奇怪的是, 增加或去除调试的打印语句, 就改变了症状…… . . . 90
16.6 为什么代码: char *p = "hello, worl!"; p[0] = 'H'; 会崩溃? . . . 90
16.7 "Segmentation violation", "Bus error" 和 "General protection fault" 意味着什么? 91

17 风格 **93**
17.1 什么是 C 最好的代码布局风格? 93
17.2 用 if(!strcmp(s1, s2)) 比较两个字符串等值,是否是个好风格? . . . 93
17.3 为什么有的人用 if (0 == x) 而不是 if (x == 0)? 93
17.4 原型说明 extern int func __((int, int)); 中, 那些多出来的括号和下划线代表了什么? 94
17.5 为什么有些代码在每次调用 printf() 前, 加了类型转换 (void)? . . 94
17.6 什么是 "匈牙利标志法" (Hungarian Notation)? 是否值得用? . . 94
17.7 哪里可以找到 "印第安山风格指南" (Indian Hill Style Guide) 及其它编码标准? 94

17.8 有些人说 goto 是邪恶的, 我应该永不用它。那是否太极端了? . . 95

18 工具和资源 97

18.1 常用工具列表。 97
18.2 怎样抓捕棘手的 malloc 问题? 98
18.3 有什么免费或便宜的编译器可以使用? 98
18.4 刚刚输入完一个程序, 但它表现的很奇怪。你可以发现有什么错误的地方吗? 98
18.5 哪里可以找到兼容 ANSI 的 lint? 99
18.6 难道 ANSI 函数原型说明没有使 lint 过时吗? 99
18.7 网上有哪些 C 的教程或其它资源? 99
18.8 哪里可以找到好的源代码实例, 以供研究和学习? 100
18.9 有什么好的学习 C 的书? 有哪些高级的书和参考? 100
18.10 哪里可以找到标准 C 函数库的源代码? 101
18.11 是否有一个在线的 C 参考指南? 101
18.12 哪里可以得到 ANSI/ISO C 标准? 101
18.13 我需要分析和评估表达式的代码。 101
18.14 哪里可以找到 C 的 BNF 或 YACC 语法? 101
18.15 谁有 C 编译器的测试套件? 102
18.16 哪里有一些有用的源代码片段和例子的收集? 102
18.17 我需要执行多精度算术的代码。 102
18.18 在哪里和怎样取得这些可自由发布的程序? 102

19 系统依赖 105

19.1 怎样从键盘直接读入字符而不用等 RETURN 键? 怎样防止字符输入时的回显? 105
19.2 怎样知道有未读的字符, 如果有, 有多少? 如果没有字符, 怎样使读入不阻断? 106
19.3 怎样显示一个百分比或 “转动的短棒” 的进展表示器? 106
19.4 怎样清屏? 怎样输出彩色文本? 怎样移动光标到指定位置? . . . 106
19.5 怎样读入方向键, 功能键? 107
19.6 怎样读入鼠标输入? 107
19.7 怎样做串口 (“comm”) 的输入输出? 107
19.8 怎样直接输出到打印机? 107
19.9 怎样发送控制终端或其它设备的逃逸指令序列? 108
19.10 怎样直接访问输入输出板? 108
19.11 怎样做图形? 108
19.12 怎样显示 GIF 和 JPEG 图象? 108
19.13 怎样检验一个文件是否存在? 108
19.14 怎样在读入文件前, 知道文件大小? 109

19.15 怎样得到文件的修改日期和时间？ 109
19.16 怎样缩短一个文件而不用清除或重写？ 109
19.17 怎样在文件中插入或删除一行 (或记录)？ 109
19.18 怎样从一个打开的流或文件描述符得到文件名？ 110
19.19 怎样删除一个文件？ 110
19.20 怎样复制一个文件？ 110
19.21 为什么用了详尽的路径还不能打开文件？ fopen("c:\ newdir \file.dat", "r") 返回错误。 110
19.22 fopen() 不让我打开文件: "$HOME/.profile" 和 "~/ .myrcfile"。 . 111
19.23 怎样制止 MS-DOS 下令人担忧的 “Abort, Retry, Ignore?” 信息？ 111
19.24 遇到 “Too many open files (打开文件太多）” 的错误, 怎样增加同时打开文件的允许数目？ 111
19.25 怎样在 C 中读入目录？ 111
19.26 怎样找出系统还有多少内存可用？ 111
19.27 怎样分配大于 64K 的数组或结构？ 111
19.28 错误信息 “DGROUP data allocation exceeds 64K (DGROUP 数据分配内存超过 64K)” 说明什么？我应该怎么做？我以为使用了大内存模型, 那我就可以使用多于 64K 的数据！ 112
19.29 怎样访问位于某的特定地址的内存 (内存映射的设备或图显内存)？ 112
19.30 怎样在一个 C 程序中调用另一个程序 (独立可执行的程序, 或系统命令)？ 112
19.31 怎样调用另一个程序或命令, 同时收集它的输出？ 113
19.32 怎样才能发现程序自己的执行文件的全路径？ 113
19.33 怎样找出和执行文件在同一目录的配置文件？ 113
19.34 一个进程如何改变它的调用者的环境变量？ 113
19.35 怎样读入一个对象文件并跳跃到其中的地址？ 114
19.36 怎样实现精度小于秒的延时或记录用户回应的时间？ 114
19.37 怎样抓获或忽略像 control-C 这样的键盘中断？ 114
19.38 怎样很好地处理浮点异常？ 115
19.39 怎样使用 socket？网络化？写客户/服务器程序？ 115
19.40 怎样调用 BIOS 函数？写 ISR？创建 TSR？ 115
19.41 编译程序, 编译器出示 “union REGS” 未定义错误信息, 连接器出示 “int86()” 的未定义错误信息。 115
19.42 什么是 “near” 和 “far” 指针？ 116
19.43 我不能使用这些非标准、依赖系统的函数, 程序需要兼容 ANSI! . 116

20 杂项 **117**
20.1 怎样从一个函数返回多个值？ 117
20.2 怎样访问命令行参数？ 117

20.3 怎样写数据文件, 使之可以在不同字大小、字节顺序或浮点格式的机器上读入？ 117
20.4 怎样调用一个由 char * 指针指向函数名的函数？ 117
20.5 怎样实现比特数组或集合？ 118
20.6 怎样判断机器的字节顺序是高字节在前还是低字节在前？ 118
20.7 怎样掉换字节？ 118
20.8 怎样转换整数到二进制或十六进制？ 119
20.9 我可以使用二进制常数吗？有 printf() 的二进制的格式符吗？ . . 119
20.10 什么是计算整数中比特为 1 的个数的最有效的方法？ 119
20.11 什么是提高程序效率的最好方法？ 119
20.12 指针真得比数组快吗？函数调用会拖慢程序多少？ ++i 比 i = i +1 快吗？ 120
20.13 人们说编译器优化的很好, 我们不在需要为速度而写汇编了, 但我的编译器连用移位代替 i/=2 都做不到。 120
20.14 怎样不用临时变量而交换两个值？ 120
20.15 是否有根据字符串做切换的方法？ 121
20.16 是否有使用非常量 case 标志的方法 (例如范围或任意的表达式)？ 121
20.17 return 语句外层的括号是否真的可选择？ 121
20.18 为什么 C 注释不能嵌套？怎样注释掉含有注释的代码？引用字符串内的注释是否合法？ 121
20.19 C 是个伟大的语言还是别的？哪个其它语言可以写象 a+++++b 这样的代码？ 122
20.20 为什么 C 没有嵌套函数？ 122
20.21 assert() 是什么？怎样用它？ 122
20.22 怎样从 C 中调用 FORTRAN (C++, BASIC, Pascal, Ada, LISP) 的函数？反之亦然？ 122
20.23 有什么程序可以做从 Pascal 或 Fortran (或 LISP, Ada, awk, “老” C) 到 C 的转换？ 123
20.24 C++ 是 C 的超集吗？可以用 C++ 编译器来编译 C 代码吗？ . . 123
20.25 需要用到 “近似” 的 strcmp, 比较两个字符串的近似度, 并不需要完全一样。 123
20.26 什么是散列法？ 124
20.27 由一个日期, 怎样知道是星期几？ 124
20.28 (year%4 == 0) 是否足够判断润年？2000 年是闰年吗？ 124
20.29 一个难题: 怎样写一个输出自己源代码的程序？ 124
20.30 什么是 “达夫设备” (Duff’s Device)？ 125
20.31 下届国际 C 混乱代码竞赛 (IOCCC) 什么时候进行？哪里可以找到当前和以前的获胜代码？ 125
20.32 [K&R1] 提到的关健字 entry 是什么？ 126

20.33 C 的名字从何而来？ 126
20.34 “char” 如何发音？ 126
20.35 “lvalue” 和 “rvalue” 代表什么意思？ 126
20.36 哪里可以取得本 FAQ (英文版) 的额外副本？ 126

21 感谢 **129**

文献 **131**

前言

本文从英文 C–FAQ (2004 年 7 月 3 日修订版) 翻译而来。本文的中文版权为朱群英和孙云所有。本文的内容可以自由用于个人目的，但是不可以未经许可出版发行。英文版权为 Steve Summit 所有，详情见下面的英文版权说明。

The English version of this FAQ list is Copyright 1990-2004 by Steve Summit. Content from the book 《C Programming FAQs: Frequently Asked Questions》 is made available here by permission of the author and the publisher as a service to the community. It is intended to complement the use of the published text and is protected by international copyright laws. The on-line content may be accessed freely for personal use but may not be republished without permission.

最新的 HTML 中译版本可以在 http://c-faq-chn.sourceforge.net/ 取得。另外在同一地址还提供 PDF 版本的下载。在 http://sourceforge.net/projects/c-faq-chn 可以得到本文的 LaTeX 源文件。

有关英文原文的问题，请咨询 Steve Summit (scs@eskimo.com)。有关中文译稿的问题, 请联系孙云 (sunyun.s@gmail.com, 1–12章) 和朱群英 (zhu.qunying@gmail.com, 13–20章、LaTeX 文件编辑)。

第 1 章

声明和初始化

1.1 我如何决定使用那种整数类型？

如果需要大数值 (大于 32,767 或小于 −32,767), 使用 long 型。否则, 如果空间很重要 (如有大数组或很多结构), 使用 short 型。除此之外, 就使用 int 型。如果严格定义的溢出特征很重要而负值无关紧要, 或者你希望在操作二进制位和字节时避免符号扩展的问题, 请使用对应的无符号类型。但是, 要注意在表达式中混用有符号和无符号值的情况。

尽管字符类型 (尤其是无符号字符型) 可以当成 “小” 整型使用, 但由于不可预知的符号扩展和代码增大有时这样做可能得不偿失。使用无符号字符型有所帮助; 类似的问题参见问题 12.1。

在选择浮点型和双精度浮点型时也有类似的权衡。但如果一个变量的指针必须为特定的类型时, 以上规则不再适用。

如果因为某种原因你需要声明一个有**严格**大小的变量, 确保象 C99 的 <int-types.h> 那样用某种适当的 typedef 封装这种选择。通常, 这样做唯一的好原因是试图符合某种外部强加的存储方案, 请参见问题 20.3。

如果你需要操作超过 C 的内置类型支持的超大变量, 请参见问题 18.17。

参考资料: [K&R1, Sec. 2.2 p. 34]; [K&R2, Sec. 2.2 p. 36, Sec. A4.2 pp. 195-6, Sec. B11 p. 257]; [ISO, Sec. 5.2.4.2.1, Sec. 6.1.2.5]; [H&S, Secs. 5.1,5.2 pp. 110-114]。

1.2 64 位机上的 64 位类型是什么样的？

C99 标准定义了 long long 类型, 其长度可以保证至少 64 位, 这种类型在某些编译器上实现已经颇有时日了。其它的编译器则实现了类似 __longlong 的扩展。另一方面, 也可以实现 16 位的短整型、32 位的整型和 64 位的长整型, 有些编译器正是这样做的。

参见问题 18.17。

参考资料: [C9X, Sec. 5.2.4.2.1, Sec. 6.1.2.5]

1.3 怎样定义和声明全局变量和函数最好？

首先, 尽管一个全局变量或函数可以 (在多个编译单元中) 有多处 “声明”, 但是 “定义” 却只能允许出现一次。定义是分配空间并赋初值 (如果有) 的声明。最好的安排是在某个相关的 .c 文件中定义, 然后在头文件 (.h) 中进行外部声明, 在需要使用的时候, 只要包含对应的头文件即可。定义变量的 .c 文件也应该包含该头文件, 以便编译器检查定义和声明的一致性。

这条规则提供了高度的可移植性: 它和 ANSI C 标准一致, 同时也兼容大多数 ANSI 前的编译器和连接器。Unix 编译器和连接器通常使用 “通用模式” 允许多重定义, 只要保证最多对一处进行初始化就可以了; ANSI C 标准称这种行为为 “公共扩展”, 没有语带双关的意思。

可以使用预处理技巧来使类似

```
DEFINE(int, i);
```

的语句在一个头文件中只出现一次, 然后根据某个宏的设定在需要的时候转化成定义或声明。但不清楚这样的麻烦是否值得。

如果希望让编译器检查声明的一致性, 一定要把全局声明放到头文件中。特别是, 永远不要把外部函数的原型放到 .c 文件中: 通常它与定义的一致性不能得到检查, 而矛盾的原型比不用还糟糕。

参见问题 10.4 和 18.6。

参考资料: [K&R1, Sec. 4.5 pp. 76-7]; [K&R2, Sec. 4.4 pp. 80-1]; [ISO, Sec. 6.1.2.2, Sec. 6.7, Sec. 6.7.2, Sec. G.5.11]; [Rationale, Sec. 3.1.2.2]; [H&S, Sec. 4.8 pp. 101-104, Sec. 9.2.3 p. 267]; [CT&P, Sec. 4.2 pp. 54-56].

1.4 extern 在函数声明中是什么意思？

它可以用作一种格式上的提示表明函数的定义可能在另一个源文件中, 但在

```
extern int f();
```

和

```
int f();
```

之间并没有实质的区别。

参考资料: [ISO, Sec. 6.1.2.2, Sec. 6.5.1]; [Rationale, Sec. 3.1.2.2]; [H&S, Secs. 4.3,4.3.1 pp. 75-6].

1.5 关键字 auto 到底有什么用途？

毫无用途；它已经过时。参见问题 20.32。

参考资料: [K&R1, Sec. A8.1 p. 193]; [ISO, Sec. 6.1.2.4, Sec. 6.5.1;]; [H&S, Sec. 4.3 p. 75, Sec. 4.3.1 p. 76].

1.6 我似乎不能成功定义一个链表。我试过 typedef struct { char *item; NODEPTR next; } *NODEPTR; 但是编译器报了错误信息。难道在C语言中一个结构不能包含指向自己的指针吗？

C 语言中的结构当然可以包含指向自己的指针; [K&R2, 第 6.5 节] 的讨论和例子表明了这点。 NODEPTR 例子的问题是在声明 next 域的时候 typedef 还没有定义。为了解决这个问题, 首先赋予这个结构一个标签 ("struct node")。然后, 声明 "next" 域为 "struct node *", 或者分开 typedef 定义和结构定义, 或者两者都采纳。以下是一个修改后的版本:

```
struct node {
    char *item;
    struct node *next;
};

typedef struct node *NODEPTR;
```

至少还有三种同样正确的方法解决这个问题。

在用 typedef 定义互相引用的两个结构时也会产生类似的问题, 可以用同样的方法解决。

参见问题 2.1。

参考资料: [K&R1, Sec. 6.5 p. 101]; [K&R2, Sec. 6.5 p. 139]; [ISO, Sec. 6.5.2, Sec. 6.5.2.3]; [H&S, Sec. 5.6.1 pp. 132-3]。

1.7 怎样建立和理解非常复杂的声明？例如定义一个包含 N 个指向返回指向字符的指针的函数的指针的数组？

这个问题至少有以下 3 种答案:

1. char *(*(*a[N])())();
2. 用 typedef 逐步完成声明:

```
typedef char *pc;       /* 字符指针             */
typedef pc fpc();       /* 返回字符指针的函数   */
typedef fpc *pfpc;      /* 上面函数的指针       */
typedef pfpc fpfpc();   /* 返回函数指针的函数   */
typedef fpfpc *pfpfpc;  /* 上面函数的指针       */
pfpfpc a[N];            /* 上面指针的数组       */
```

3. 使用 cdecl 程序, 它可以把英文翻译成 C 或者把 C 翻译成英文:

```
cdecl> declare a as array of pointer to function returning
   pointer to function returning pointer to char
char *(*(*a[])())()
```

通过类型转换, cdecl 也可以用于解释复杂的声明, 指出参数应该进入哪一对括号 (如同在上述的复杂函数定义中)。参见问题 18.1。

一本好的 C 语言书都会解释如何 “从内到外” 解释和理解这样复杂的 C 语言声明 (“模拟声明使用”)。

上文的例子中的函数指针声明还没有包括参数类型信息。如果参数有复杂类型, 声明就会变得真正的混乱了。现代的 cdecl 版本可以提供帮助。

参考资料: [K&R2, Sec. 5.12 p. 122]; [ISO, Sec. 6.5ff (esp. Sec. 6.5.4)]; [H&S, Sec. 4.5 pp. 85-92, Sec. 5.10.1 pp. 149-50]。

1.8 函数只定义了一次, 调用了一次, 但编译器提示非法重定义了。

在范围内没有声明就调用 (可能是第一次调用在函数的定义之前) 的函数被认为返回整型 (int) (且没有任何参数类型信息), 如果函数在后边声明或定义成其它类型就会导致矛盾。所有函数 (非整型函数一定要) 必须在调用之前声明。

另一个可能的原因是该函数与某个头文件中声明的另一个函数同名。

参见问题 11.4 和 15.1

参考资料: [K&R1, Sec. 4.2 p. 70]; [K&R2, Sec. 4.2 p. 72]; [ISO, Sec. 6.3.2.2]; [H&S, Sec. 4.7 p. 101].

1.9 main() 的正确定义是什么？ void main() 正确吗？

参见问题 11.11 到 11.16。(这样的定义不正确)。

1.10 对于没有初始化的变量的初始值可以作怎样的假定？如果一个全局变量初始值为 “零”, 它可否作为空指针或浮点零？

具有 “静态” 生存期的未初始化变量 (即, 在函数外声明的变量和有静态存储类型的变量) 可以确保初始值为零, 就像程序员键入了 “=0” 一样。因此, 这些变量如果是指针会被初始化为正确的空指针, 如果是浮点数会被初始化为 0.0 (或正确的类型, 参见第 5 章)。

具有 “自动” 生存期的变量 (即, 没有静态存储类型的局部变量) 如果没有显示地初始化, 则包含的是垃圾内容。对垃圾内容不能作任何有用的假设。

这些规则也适用于数组和结构 (称为 “聚合体”); 对于初始化来说, 数组和结构都被认为是 “变量”。

用 malloc() 和 realloc() 动态分配的内存也可能包含垃圾数据, 因此必须由调用者正确地初始化。用 calloc() 获得的内存为全零, 但这对指针和浮点值不一定有用 (参见问题 7.26 和第 5 章)。

参考资料: [K&R1, Sec. 4.9 pp. 82-4]; [K&R2, Sec. 4.9 pp. 85-86]; [ISO, Sec. 6.5.7, Sec. 7.10.3.1, Sec. 7.10.5.3]; [H&S, Sec. 4.2.8 pp. 72-3, Sec. 4.6 pp. 92-3, Sec. 4.6.2 pp. 94-5, Sec. 4.6.3 p. 96, Sec. 16.1 p. 386.]。

1.11 代码 int f() { char a[] = "Hello, world!";} 不能编译。

可能你使用的是 ANSI 之前的编译器, 还不支持 “自动聚集”(automatic aggregates, 即非静态局部数组、结构和联合) 的初始化。参见问题 11.28。

1.12 这样的初始化有什么问题？char *p = malloc(10); 编译器提示 “非法初始式” 云云。

这个声明是静态或非局部变量吗？函数调用只能出现在自动变量 (即局部非静态变量) 的初始式中。

1.13 以下的初始化有什么区别？char a[] = "string literal"; char *p = "string literal"; 当我向 p[i] 赋值的时候, 我的程序崩溃了。

字符串常量有两种稍有区别的用法。用作数组初始值 (如同在 char a[] 的声明中), 它指明该数组中字符的初始值。其它情况下, 它会转化为一个无名的静态字符数组, 可能会存储在只读内存中, 这就是造成它不一定能被修改。在表达式环境中, 数组通常被立即转化为一个指针 (参见第 6 章), 因此第二个声明把 p 初始化成指向无名数组的第一个元素。

为了编译旧代码, 有的编译器有一个控制字符串是否可写的开关。

参见问题 1.11、 6.1、 6.2 和 6.6。

参考资料: [K&R2, Sec. 5.5 p. 104]; [ISO, Sec. 6.1.4, Sec. 6.5.7]; [Rationale, Sec. 3.1.4]; [H&S, Sec. 2.7.4 pp. 31-2]。

1.14 我总算弄清除函数指针的声明方法了, 但怎样才能初始化呢？

用下面这样的代码

```
extern int func();
int (*fp)() = func;
```

当一个函数名出现在这样的表达式中时, 它就会 “蜕变” 成一个指针 (即, 隐式地取出了它的地址), 这有点类似数组名的行为。

通常函数的显示声明需要事先知道 (也许在一个头文件中)。因为此处并没有隐式的外部函数声明 (初始式中函数名并非一个函数调用的一部分)。

参见问题 1.8 和 4.8。

第 2 章

结构、联合和枚举

2.1 声明 struct x1 { ... }; 和 typedef struct { ... } x2; 有什么不同？

第一种形式声明了一个 “结构标签”; 第二种声明了一个 “类型定义”。主要的区别是在后文中你需要用 “struct x1” 引用第一种, 而用 “x2” 引用第二种。也就是说, 第二种声明更像一种抽象类新 —— 用户不必知道它是一个结构, 而在声明它的实例时也不需要使用 struct 关键字。

2.2 为什么 struct x { ... }; x thestruct; 不对？

C 不是 C++。结构标签不能自动生成类型。参见问题 2.1。

2.3 一个结构可以包含指向自己的指针吗？

当然可以。参见问题 1.6。

2.4 在 C 语言中实现抽象数据类型什么方法最好?

让客户使用指向没有公开定义 (也许还隐藏在类型定义后边) 的结构类型的指针是一个好办法。只要不访问结构成员, 声明和使用 “匿名” 结构指针 (不完全结构类型指针)是合法的。这也是使用抽象数据类型的原因。

2.5 在 C 中是否有模拟继承等面向对象程序设计特性的好方法？

把函数指针直接加入到结构中就可以实现简单的 “方法”。你可以使用各种不雅而暴力的方法来实现继承, 例如通过预处理器或含有 “基类” 的结构作为开始的子集, 但这些方法都不完美。很明显, 也没有运算符的重载和覆盖 (例如, “导出类” 中的 “方法”), 那些必须人工去做。

显然的, 如果你需要 “真” 的面向对象的程序设计, 你需要使用一个支持这些特性的语言, 例如 C++。

2.6 我遇到这样声明结构的代码: struct name { int namelen; char namestr[1];}; 然后又使用一些内存分配技巧使 namestr 数组用起来好像有多个元素。这样合法和可移植吗？

这种技术十分普遍, 尽管 Dennis Ritchie 称之为 “和C 实现的无保证的亲密接触”。官方的解释认定它没有严格遵守 C 标准, 尽管它看来在所有的实现中都可以工作。仔细检查数组边界的编译器可能会发出警告。

另一种可能是把变长的元素声明为很大, 而不是很小; 在上例中:

```
...
char namestr[MAXSIZE];
```

MAXSIZE 比任何可能存储的 name 值都大。但是, 这种技术似乎也不完全符合标准的严格解释。这些 “亲密” 结构都必须小心使用, 因为只有程序员知道它的大小, 而编译器却一无所知。

C99 引入了 “灵活数组域” 概念, 允许结构的最后一个域省略数组大小。这为类似问题提供了一个圆满的解决方案。

参考资料: [Rationale, Sec. 3.5.4.2]; [C9X, Sec. 6.5.2.1]。

2.7 是否有自动比较结构的方法？

没有。编译器没有简单的好办法实现结构比较 (即, 支持结构的 == 操作符), 这也符合 C 的低层特性。简单的按字节比较会由于结构中没有用到的 “空洞” 中的随机数据 (参见问题 2.10) 而失败; 而按域比较在处理大结构时需要难以接受的大量重复代码。

如果你需要比较两个结构, 你必须自己写函数按域比较。

参考资料: [K&R2, Sec. 6.2 p. 129]; [Rationale, Sec. 3.3.9]; [H&S, Sec. 5.6.2 p. 133]。

2.8 如何向接受结构参数的函数传入常数值？

传统的 C 没有办法生成匿名结构值; 你必须使用临时结构变量或一个小的结构生成函数。

C99 标准引入了 “复合常量” (compound literals); 复合常量的一种形式就可以允许结构常量。例如, 向假想 plotpoint() 函数传入一个坐标对常数, 可以调用

```
plotpoint((struct point){1, 2});
```

与 “指定初始值” (designated initializers) (C99 的另一个功能) 结合, 也可以用成员名称确定成员值:

```
plotpoint((struct point){.x=1, .y=2});
```

参见问题 4.6。

参考资料: [C9X, Sec. 6.3.2.5, Sec. 6.5.8]。

2.9 怎样从/向数据文件读/写结构？

用 fwrite() 写一个结构相对简单:

```
fwrite(&somestruct, sizeof somestruct, 1, fp);
```

对应的 fread() 调用可以再把它读回来。但是这样写出的文件却不能移植 (参见问题 2.10 和 20.3)。同时注意如果结构包含任何指针, 则只有指针值会被写入文件, 当它们再次读回来的时候, 很可能已经失效。最后, 为了广泛的移植, 你必须用 “b” 标志打开文件; 参见问题 12.30。

移植性更好的方案是写一对函数, 用可移植 (可能甚至是人可读) 的方式按域读写结构, 尽管开始可能工作量稍大。

参考资料: [H&S, Sec. 15.13 p. 381]。

2.10 我的编译器在结构中留下了空洞, 这导致空间浪费而且无法与外部数据文件进行 ”二进制” 读写。能否关掉填充, 或者控制结构域的对齐方式?

这些 “空洞” 充当了 “填充”, 为了保持结构中后面的域的对齐, 这也许是必须的。为了高效的访问, 许多处理器喜欢 (或要求) 多字节对象 (例如, 结构中任何大于 char 的类型) 不能处于随意的内存地址, 而必须是 2 或 4 或对象大小的倍数。

编译器可能提供一种扩展用于这种控制 (可能是 #pragma; 参见问题 11.21), 但是没有标准的方法。

参见问题 20.3。

参考资料: [K&R2, Sec. 6.4 p. 138]; [H&S, Sec. 5.6.4 p. 135]。

2.11 为什么 sizeof 返回的值大于结构的期望值, 是不是尾部有填充?

为了确保分配连续的结构数组时正确对齐, 结构可能有这种尾部填充。即使结构不是数组的成员, 填充也会保持, 以便 sizeof 能够总是返回一致的大小。参见问题 2.10。

参考资料: [H&S, Sec. 5.6.7 pp. 139-40]。

2.12 如何确定域在结构中的字节偏移？

ANSI C 在 <stddef.h> 中定义了 offsetof() 宏, 用 offsetof(struct s, f) 可以计算出域 f 在结构 s 中的偏移量。如果出于某种原因, 你需要自己实现这个功能, 可以使用下边这样的代码:

```
#define offsetof(type, f) ((size_t) \
    ((char *)&((type *)0)->f - (char *)(type *)0))
```

这种实现不是 100% 的可移植; 某些编译器可能会合法地拒绝接受。

参考资料: [ISO, Sec. 7.1.6]; [Rationale, Sec. 3.5.4.2]; [H&S, Sec. 11.1 pp. 292-3]。

2.13 怎样在运行时用名字访问结构中的域？

保持用 offsetof() (参见问题 2.12) 计算的域偏移量。如果 structp 是个结构实体的指针, 而域 f 是个整数, 它的偏移量是 offsetf, f 的值可以间接地设置:

```
*(int *)((char *)structp + offsetf) = value;
```

2.14 程序运行正确, 但退出时却 “core dump” 了 , 怎么回事？

问题程序:

```
struct list {
    char *item;
    struct list *next;
}

    /* 这里是 main 程序 */

main(argc, argv)
{ ... }
```

缺少的一个分号使 main() 被定义为返回一个结构。由于中间的注释行, 这个联系不容易看出来。因为一般上, 返回结构的函数在实现时, 会加入一个隐含的返回指针, 这个产生的 main() 函数代码试图接受三个参数, 而实际上只有两个传入 (这里, 由 C 的启动代码传入)。参见问题 10.8 和 16.4。

参考资料: [CT&P, Sec. 2.3 pp. 21-2]。

2.15 可以初始化一个联合吗？

在原来的 ANSI C 中, 只有联合中的第一个命名成员可以被初始化。C99 引入了 “指定初始值”, 可以用来初始化任意成员。

参考资料: [K&R2, Sec. 6.8 pp. 148-9]; [ISO, Sec. 6.5.7]; [C9X, Sec. 6.5.8]; [H&S, Sec. 4.6.7 p. 100]。

2.16 枚举和一组预处理的 #define 有什么不同？

只有很小的区别。 C 标准中允许枚举和其它整形类别自由混用而不会出错。（但是, 假如编译器不允许在未经明确类型转换的情况下混用这些类型，则聪明地使用枚举可以捕捉到某些程序错误。）

枚举的一些优点: 自动赋值; 调试器在检验枚举变量时, 可以显示符号值; 它们服从数据块作用域规则。（编译器也可以对在枚举变量被任意地和其它类型混用时, 产生非重要的警告信息, 因为这被认为是坏风格。）一个缺点是程序员不能控制这些对非重要的警告; 有些程序员则反感于无法控制枚举变量的大小。

参考资料: [K&R2, Sec. 2.3 p. 39, Sec. A4.2 p. 196]; [ISO, Sec. 6.1.2.5, Sec. 6.5.2, Sec. 6.5.2.2, Annex F]; [H&S, Sec. 5.5 pp. 127-9, Sec. 5.11.2 p. 153]。

2.17 有什么容易的显示枚举值符号的方法？

没有。你可以写一个小函数, 把一个枚举常量值映射到字符串。（为了调试的目的, 一个好的调试器, 应该可以自动显示枚举常量值符号。）

第 3 章

表达式

3.1 为什么这样的代码: a[i] = i++; 不能工作？

子表达式 i++ 有一个副作用 —— 它会改变 i 的值 —— 由于 i 在同一表达式的其它地方被引用, 这会导致无定义的结果, 无从判断该引用(左边的 a[i] 中)是旧值还是新值。(注意, 尽管在 K&R 中建议这类表达式的行为不确定, 但 C 标准却强烈声明它是无定义的, 参见问题 11.32。

参考资料: [K&R1, Sec. 2.12]; [K&R2, Sec. 2.12]; [ISO, Sec. 6.3]; [H&S, Sec. 7.12 pp. 227-9]。

3.2 使用我的编译器,下面的代码 int i=7; printf("%d\n", i++ * i++); 返回 49？不管按什么顺序计算, 难道不该打印出56吗？

尽管后缀自加和后缀自减操作符 ++ 和 -- 在输出其旧值之后才会执行运算, 但这里的"之后"常常被误解。**没有**任何保证确保自增或自减会在输出变量原值之后和对表达式的其它部分进行计算之前立即进行。也不能保证变量的更新会在表达式 "完成" (按照 ANSI C 的术语, 在下一个 "序列点" 之前, 参见问题 3.7) 之前的某个时刻进行。本例中, 编译器选择使用变量的旧值相乘以后再对二者进行自增运算。

包含多个不确定的副作用的代码的行为总是被认为未定义。(简单而言, "多个不确定副作用" 是指在同一个表达式中使用导致同一对象修改两次或修改以后又被引用的自增, 自减和赋值操作符的任何组合。这是一个粗略的定义; 严格的定义参见问题 3.7, "未定义" 的含义参见问题 11.32。) 甚至都不要试图探究这些东西在你的编译器中是如何实现的 (这与许多 C 教科书上的弱智练习正好相反); 正如 K&R 明智地指出, "如果你不知道它们在不同的机器上**如何**实现, 这样的无知可能恰恰会有助于保护你。"

参考资料: [K&R1, Sec. 2.12 p. 50]; [K&R2, Sec. 2.12 p. 54]; [ISO, Sec. 6.3]; [H&S, Sec. 7.12 pp. 227-9]; [CT&P, Sec. 3.7 p. 47]; [PCS, Sec. 9.5 pp. 120-1]。

3.3 对于代码 int i = 3; i = i++; 不同编译器给出不同的结果, 有的为 3, 有的为 4, 哪个是正确的？

没有正确答案;这个表达式无定义。参见问题 3.1, 3.7 和 11.32。同时注意, i++ 和 ++i 都不同于 i+1。如果你要使 i 自增 1, 使用 i=i+1, i+=1, i++ 或 ++i, 而不是任何组合, 参见问题 3.10。

3.4 这是个巧妙的表达式: a ^= b ^= a ^= b 它不需要临时变量就可以交换 a 和 b 的值。

这不具有可移植性。它试图在序列点之间两次修改变量 a, 而这是无定义的。

例如,有人报告如下代码:

```
int a = 123, b = 7654;
a ^= b ^= a ^= b;
```

在 SCO 优化 C 编译器 (icc) 下会把 b 置为 123, 把 a 置为 0。

参见问题 3.1、3.7 和 20.14。

3.5 我可否用括号来强制执行我所需要的计算顺序？

一般来讲, 不行。运算符优先级和括弧只能赋予表达是计算部分的顺序. 在如下的代码中

```
f() + g() * h()
```

尽管我们知道乘法运算在加法之前, 但这并不能说明这三个函数哪个会被首先调用。

如果你需要确保子表达式的计算顺序, 你可能需要使用明确的临时变量和独立的语句。

参考资料: [K&R1, Sec. 2.12 p. 49, Sec. A.7 p]; [K&R2, Sec. 2.12 pp. 52-3, Sec. A.7 p. 200.]。

3.6 可是 && 和 || 运算符呢？我看到过类似 while((c = getchar()) != EOF && c != '\n') 的代码 ……

这些运算符在此处有一个特殊的 "短路" 例外: 如果左边的子表达式决定最终结果 (即,真对于 || 和假对于 &&) 则右边的子表达式不会计算。因此, 从左至右的计算可以确保, 对逗号表达式也是如此。而且, 所有这些运算符 (包括 ? :) 都会引入一个额外的内部序列点 (参见问题 3.7)。

参考资料: [K&R1, Sec. 2.6 p. 38, Secs. A7.11-12 pp. 190-1]; [K&R2, Sec. 2.6 p. 41, Secs. A7.14-15 pp. 207-8]; [ISO, Sec. 6.3.13, Sec. 6.3.14, Sec. 6.3.15]; [H&S, Sec. 7.7 pp. 217-8, Sec. 7.8 pp. 218-20, Sec. 7.12.1 p. 229]; [CT&P, Sec. 3.7 pp. 46-7]。

3.7 我怎样才能理解复杂表达式？“序列点” 是什么？

序列点是一个时间点(在整个表达式全部计算完毕之后或在 ||、 &&、 ? : 或逗号运算符处, 或在函数调用之前), 此刻尘埃落定, 所有的副作用都已确保结束。ANSI/ISO C 标准这样描述:

> 在上一个和下一个序列点之间, 一个对象所保存的值至多只能被表达式的计算修改一次。而且前一个值只能用于决定将要保存的值。

第二句话比较费解。它说在一个表达式中如果某个对象需要写入, 则在同一表达式中对该对象的访问应该只局限于直接用于计算将要写入的值。这条规则有效地限制了只有能确保在修改之前才访问变量的表达式为合法。例如 i = i+1 合法, 而 a[i] = i++ 则非法 (参见问题 3.1)。

参见下边的问题 3.8。

参考资料: [ISO, Sec. 5.1.2.3, Sec. 6.3, Sec. 6.6, Annex C]; [Rationale, Sec. 2.1.2.3]; [H&S, Sec. 7.12.1 pp. 228-9]。

3.8 那么, 对于 a[i] = i++; 我们不知道 a[] 的哪一个分量会被改写,但 i 的确会增加 1, 对吗？

不一定！如果一个表达式和程序变得未定义, 则它的所有方面都会变成未定义。参见问题 3.2, 3.3, 11.32 和 11.35。

3.9 ++i 和 i++ 有什么区别？

如果你的 C 语言书没有说明它们的区别, 那么买一本好的。简单而言: ++i 在 i 存储的值上增加一并向使用它的表达式 “返回” 新的, 增加后的值; 而 i++ 对 i 增加一, 但返回原来的是未增加的值。

3.10 如果我不使用表达式的值, 我应该用 ++i 或 i++ 来自增一个变量吗？

由于这两种格式区别仅在于生成的值, 所以在仅使用它们的副作用时, 二者完全一样。但是, 在 C++ 中, 前缀方式却是首选。参见问题 3.3。

3.11 为什么如下的代码 int a = 100, b = 100; long int c = a * b; 不能工作？

根据 C 的内部类型转换规则, 乘法是用 int 进行的, 而其结果可能在转换为 long 型并赋给左边的 c 之前溢出或被截短。可以使用明确的类型转换, 强迫乘法以 long 型进行:

```
long int c = (long int)a * b;
```

注意，(long int)(a * b) 不能达到需要的效果。

当两个整数做除法而结果赋与一个浮点变量时，也有可能有同样类型的问题，解决方法也是类似的。

参考资料：[K&R1, Sec. 2.7 p. 41]; [K&R2, Sec. 2.7 p. 44]; [ISO, Sec. 6.2.1.5]; [H&S, 使用我的编辑器,下面的代码]; [CT&P, Sec. 3.9 pp. 49-50]。

3.12 我需要根据条件把一个复杂的表达式赋值给两个变量中的一个。可以用下边这样的代码吗？((condition) ? a : b) = complicated_expression;

不能。? : 操作符，跟多数操作符一样，生成一个值，而不能被赋值。换言之，? : 不能生成一个 “左值”。如果你真的需要，你可以试试下面这样的代码：

```
*((condition) ? &a : &b) = complicated_expression;
```

尽管这毫无优雅可言。

参考资料：[ISO, Sec. 6.3.15]; [H&S, Sec. 7.1 pp. 179-180]。

第 4 章

指针

4.1 我想声明一个指针并为它分配一些空间，但却不行。这些代码有什么问题？char *p; *p = malloc(10);

你所声明的指针是 p，而不是 *p，当你操作指针本身时 (例如当你对其赋值，使之指向别处时)，你只需要使用指针的名字即可：

```
p = malloc(10);
```

当你操作指针指向的内存时，你才需要使用 * 作为间接操作符：

```
*p = 'H';
```

参见问题 1.7, 7.1, 7.5 和 8.3。

参考资料：[CT&P, Sec. 3.1 p. 28]。

4.2 *p++ 自增 p 还是 p 所指向的变量？

后缀 ++ 和 -- 操作符本质上比前缀一目操作的优先级高，因此 *p++ 和 *(p++) 等价，它自增 p 并返回 p 自增之前所指向的值。要自增 p 指向的值，使用 (*p)++，如果副作用的顺序无关紧要也可以使用 ++*p。

参考资料：[K&R1, Sec. 5.1 p. 91]; [K&R2, Sec. 5.1 p. 95]; [ISO, Sec. 6.3.2, Sec. 6.3.3]; [H&S, Sec. 7.4.4 pp. 192-3, Sec. 7.5 p. 193, Secs. 7.5.7,7.5.8 pp. 199-200]。

4.3 我有一个 char * 型指针正巧指向一些 int 型变量，我想跳过它们。为什么如下的代码((int *)p)++; 不行？

在 C 语言中，类型转换意味着 “把这些二进制位看作另一种类型，并作相应的对待”；这是一个转换操作符，根据定义它只能生成一个右值 (rvalue)。而右值既不能赋值，也不能用 ++ 自增。(如果编译器支持这样的扩展，那要么是一个错误，要么是有意作出的非标准扩展。) 要达到你的目的可以用：

```
p = (char *)((int *)p + 1);
```

或者,因为 p 是 char * 型，直接用

```
p += sizeof(int);
```

但是，在可能的情况下，你还是应该首先选择适当的指针类型，而不是一味地试图李代桃僵。

参考资料：[K&R2, Sec. A7.5 p. 205]; [ISO, Sec. 6.3.4]; [Rationale, Sec. 3.3.2.4]; [H&S, Sec. 7.1 pp. 179-80]。

4.4 我有个函数,它应该接受并初始化一个指针 void f(int *ip) { static int dummy = 5; ip = &dummy;} 但是当我如下调用时: int *ip; f(ip); 调用者的指针却没有任何变化。

你确定函数初始化的是你希望它初始化的东西吗？请记住在 C 中, 参数是通过值传递的。被调函数仅仅修改了传入的指针副本。你需要传入指针的地址 (函数变成接受指针的指针), 或者让函数返回指针。

参见问题 4.5 和 4.7。

4.5 我能否用 void** 指针作为参数, 使函数按引用接受一般指针？

不可移植。C 中没有一般的指针的指针类型。void* 可以用作一般指针只是因为当它和其它类型相互赋值的时候, 如果需要, 它可以自动转换成其它类型; 但是, 如果试图这样转换所指类型为 void* 之外的类型的 void** 指针时, 这个转换不能完成。

4.6 我有一个函数 extern int f(int *); 它接受指向 int 型的指针。我怎样用引用方式传入一个常数？下面这样的调用 f(&5); 似乎不行。

在 C99 中, 你可以使用 “复合常量”:

```
f((int[]){5});
```

在 C99 之前, 你不能直接这样做; 你必须先定义一个临时变量, 然后把它的地址传给函数:

```
int five = 5;
f(&five);
```

参见问题 2.8, 4.4 和 20.1。

4.7 C 有 “按引用传递” 吗？

真的没有。

严格地讲, C 总是按值传递。你可以自己模拟按引用传递, 定义接受指针的函数, 然后在调用时使用 & 操作符。事实上, 当你向函数传入数组 (传入指针的情况参见问题 6.4 及其它) 时, 编译器本质上就是在模拟按引用传递。但是 C 没有任何真正等同于正式的按引用传递或 C++ 的引用参数的东西。另一方面, 类似函数的预处理宏可以提供一种“按名称传递”的形式。

参见问题 4.4 和 20.1。

参考资料: [K&R1, Sec. 1.8 pp. 24-5, Sec. 5.2 pp. 91-3]; [K&R2, Sec. 1.8 pp. 27-8, Sec. 5.2 pp. 95-7]; [ISO, Sec. 6.3.2.2]; [H&S, Sec. 9.5 pp. 273-4]。

4.8 我看到了用指针调用函数的不同语法形式。到底怎么回事？

最初, 一个函数指针必须用 * 操作符 (和一对额外的括弧) “转换为” 一个 “真正的” 函数才能调用:

```
int r, func(), (*fp)() = func;
r = (*fp)();
```

而函数总是通过指针进行调用的, 所有 “真正的” 函数名总是隐式的退化为指针 (在表达式中, 正如在初始化时一样。参见问题 1.14)。这个推论表明无论 fp 是函数名和函数的指针

```
r = fp();
```

ANSI C 标准实际上接受后边的解释, 这意味着 * 操作符不再需要, 尽管依然允许。

参见问题 1.14。

参考资料: [K&R1, Sec. 5.12 p. 116]; [K&R2, Sec. 5.11 p. 120]; [ISO, Sec. 6.3.2.2]; [Rationale, Sec. 3.3.2.2]; [H&S, Sec. 5.8 p. 147, Sec. 7.4.3 p. 190]。

4.9 我怎样把一个 int 变量转换为 char * 型？我试了类型转换, 但是不行。

这取决于你希望做什么。如果你的类型转换不成功, 你可能是企图把整数转为字符串, 这种情况参见问题 13.1。如果你试图把整数转换为字符, 参见问题 8.4。如果你试图让一个指针指向特定的内存地址, 参见问题 19.29。

第 5 章

空 (null) 指针

5.1 臭名昭著的空指针到底是什么？

语言定义中说明, 每一种指针类型都有一个特殊值 —— “空指针” —— 它与同类型的其它所有指针值都不相同, 它 “与任何对象或函数的指针值都不相等”。也就是说, 取地址操作符 & 永远也不能得到空指针, 同样对 malloc() 的成功调用也不会返回空指针, 如果失败, malloc() 的确返回空指针, 这是空指针的典型用法: 表示 “未分配” 或者 “尚未指向任何地方” 的指针。

空指针在概念上不同于未初始化的指针。空指针可以确保不指向任何对象或函数; 而未初始化指针则可能指向任何地方。参见问题 1.10、 7.1 和 7.26。

如上文所述, 每种指针类型都有一个空指针, 而不同类型的空指针的内部表示可能不尽相同。尽管程序员不必知道内部值, 但编译器必须时刻明确需要那种空指针, 以便在需要的时候加以区分 (参见问题 5.2、 5.5 和 5.6)。

参考资料： [K&R1, Sec. 5.4 pp. 97-8]; [K&R2, Sec. 5.4 p. 102]; [ISO, Sec. 6.2.2.3]; [Rationale, Sec. 3.2.2.3]; [H&S, Sec. 5.3.2 pp. 121-3]。

5.2 怎样在程序里获得一个空指针？

根据语言定义, 在指针上下文中的常数 0 会在编译时转换为空指针。也就是说, 在初始化、赋值或比较的时候, 如果一边是指针类型的值或表达式, 编译器可以确定另一边的常数 0 为空指针并生成正确的空指针值。因此下边的代码段完全合法：

```
char *p = 0;
if(p != 0)
```

参见问题 5.3。

然而, 传入函数的参数不一定被当作指针环境, 因而编译器可能不能识别未加修饰的 0 “表示” 指针。在函数调用的上下文中生成空指针需要明确的类型转换, 强制把 0 看作指针。例如, Unix 系统调用 execl 接受变长的以空指针结束的字符指针参数。它应该如下正确调用：

```
execl("/bin/sh", "sh", "-c", "date", (char *)0);
```

如果省略最后一个参数的 (char *) 转换, 则编译器无从知道这是一个空指针, 从而当作一个 0 传入。(注意很多 Unix 手册在这个例子上都弄错了。)

如果范围内有函数原型, 则参数传递变为 “赋值上下文”, 从而可以安全省略多数类型转换, 因为原型告知编译器需要指针, 使之把未加修饰的 0 正确转换为适当的指针。函数原型不能为变长参数列表中的可变参数提供类型。(参见问题 15.3) 在函数调用时对所有的空指针进行类型转换可能是预防可变参数和无原型函数出问题的最安全的办法。

摘要:

可以使用未加修饰的 0:	需要显示的类型转换:
初始化	函数调用, 作用域内无原型
赋值	变参函数调用中的可变参数
比较	
固定参数的函数调用且在作用域内有原型	

参考资料： [K&R1, Sec. A7.7 p. 190, Sec. A7.14 p. 192]; [K&R2, Sec. A7.10 p. 207, Sec. A7.17 p. 209]; [ISO, Sec. 6.2.2.3]; [H&S, Sec. 4.6.3 p. 95, Sec. 6.2.7 p. 171]。

5.3 用缩写的指针比较 “if(p)” 检查空指针是否可靠？如果空指针的内部表达不是 0 会怎么样？

当 C 在表达式中要求布尔值时, 如果表达式等于 0 则认为该值为假, 否则为真。换言之, 只要写出

`if(expr)`

无论 “expr” 是任何表达式, 编译器本质上都会把它当

`if((expr) != 0)`

处理。

如果用指针 p 代替 “expr” 则

`if(p)` 等价于 `if(p != 0)`。

而这是一个比较上下文, 因此编译器可以看出 0 实际上是一个空指针常数, 并使用正确的空指针值。这里没有任何欺骗; 编译器就是这样工作的, 并为、二者生成完全一样的代码。空指针的内部表达**无关紧要**。

布尔否操作符 ! 可如下描述:

!expr 本质上等价于 (expr)? 0:1
或等价于 ((expr) == 0)

从而得出结论

if(!p) 等价于 if(p == 0)

类似 if(p) 这样的 “缩写”, 尽管完全合法, 但被一些人认为是不好的风格 (另外一些人认为恰恰是好的风格; 参见问题 17.8)。

参见问题 9.2。

参考资料： [K&R2, Sec. A7.4.7 p. 204]; [ISO, Sec. 6.3.3.3, Sec. 6.3.9, Sec. 6.3.13, Sec. 6.3.14, Sec. 6.3.15, Sec. 6.6.4.1, Sec. 6.6.5]; [H&S, Sec. 5.3.2 p. 122]。

5.4 NULL 是什么, 它是怎么定义的？

作为一种风格, 很多人不愿意在程序中到处出现未加修饰的 0。因此定义了预处理宏 NULL (在 <stdio.h> 和其它几个头文件中) 为空指针常数, 通常是 0 或者 ((void *)0) (参见问题 5.6)。希望区别整数 0 和空指针 0 的人可以在需要空指针的地方使用 NULL。

使用 NULL 只是一种风格习惯; 预处理器把所有的 NULL 都还原回 0, 而编译还是依照上文的描述处理指针上下文的 0。特别是, 在函数调用的参数里, NULL 之前 (正如在 0 之前) 的类型转换还是需要。问题 5.2 下的表格对 0 和 NULL 都有效 (带修饰的 NULL 和带修饰的 0 完全等价)。

NULL **只能**用作指针常数; 参见问题 5.7。

参考资料： [K&R1, Sec. 5.4 pp. 97-8]; [K&R2, Sec. 5.4 p. 102]; [ISO, Sec. 7.1.6, Sec. 6.2.2.3]; [Rationale, Sec. 4.1.5]; [H&S, Sec. 5.3.2 p. 122, Sec. 11.1 p. 292]。

5.5 在使用非全零作为空指针内部表达的机器上, NULL 是如何定义的？

跟其它机器一样: 定义为 0 (或某种形式的 0; 参见问题 5.4)。

当程序员请求一个空指针时, 无论写 “0” 还是 “NULL”, 都是有编译器来生成适合机器的空指针的二进制表达形式。因此, 在空指针的内部表达不为 0 的机器上定义 NULL 为 0 跟在其它机器上一样合法：编译器在指针上下文看到的未加修饰的 0 都会被生成正确的空指针。参见问题 5.2、 5.8 和 5.14。

参考资料： [ISO, Sec. 7.1.6]; [Rationale, Sec. 4.1.5]。

5.6 如果 NULL 定义成 #define NULL ((char *)0) 难道不就可以向函数传入不加转换的 NULL 了吗？

一般情况下, 不行。复杂之处在于, 有的机器不同类型数据的指针有不同的内部表达。这样的 NULL 定义对于接受字符指针的的函数没有问题, 但对于其它类型的指针参数仍然有问题 (在缺少原型的情况下), 而合法的构造如

```
FILE *fp = NULL;
```

则会失败。

不过, ANSI C 允许 NULL 的可选定义

```
#define NULL ((void *)0)
```

除了潜在地帮助错误程序运行 (仅限于使用同样类型指针的机器, 因此帮助有限) 以外, 这样的定义还可以发现错误使用 NULL 的程序 (例如, 在实际需要使用 ASCII NUL 字符的地方; 参见问题 5.7)。

无论如何, ANSI 函数原型确保大多数 (尽管不是全部; 参见问题 5.2)指针参数在传入函数时正确转换。因此, 这个问题有些多余。

参考资料： [Rationale, Sec. 4.1.5]。

5.7 如果 NULL 和 0 作为空指针常数是等价的，那我到底该用哪一个呢？

许多程序员认为在所有的指针上下文中都应该使用 NULL, 以表明该值应该被看作指针。另一些人则认为用一个宏来定义 0, 只不过把事情搞得更复杂, 反而令人困惑。因而倾向于使用未加修饰的 0。没有正确的答案。(参见问题 9.2 和 17.8) C 程序员应该明白, 在指针上下文中 NULL 和 0 是完全等价的, 而未加修饰的 0 也完全可以接受。任何使用 NULL (跟 0 相对) 的地方都应该看作一种温和的提示, 是在使用指针; 程序员 (和编译器都) 不能依靠它来区别指针 0 和整数 0。

在需要其它类型的 0 的时候, 即便它可能工作也不能使用 NULL, 因为这样做发出了错误的格式信息。(而且, ANSI 允许把 NULL 定义为 ((void *)0), 这在非指针的上下文中完全无效。特别是, 不能在需要 ASCII 空字符 (NUL) 的地方用 NULL。如果有必要, 提供你自己的定义

```
#define NUL '\0'
```

参考资料： [K&R1, Sec. 5.4 pp. 97-8]; [K&R2, Sec. 5.4 p. 102]。

5.8 但是如果 NULL 的值改变了，比如在使用非零内部空指针的机器上，难道用 NULL (而不是 0) 不是更好吗？

不。(用 NULL 可能更好, 但不是这个原因。) 尽管符号常量经常代替数字使用以备数字的改变, 但这不是用 NULL 代替 0 的原因。语言本身确保了源码中的 0 (用于指针上下文) 会生成空指针。NULL 只是用作一种格式习惯。参见问题 5.5 和 9.2。

5.9 用预定义宏 #define Nullptr(type) (type *)0 帮助创建正确类型的空指针。

这种技巧, 尽管很流行而且表面上看起来很有吸引力, 但却没有多少意义。在赋值和比较时它并不需要; 参见问题 5.2。它甚至都不能节省键盘输入。参见问题 9.1 和10.1。

5.10 这有点奇怪。NULL 可以确保是 0, 但空 (null) 指针却不一定？

随便使用术语 “null” 或 “NULL” 时, 可能意味着以下一种或几种含义:

1. 概念上的空指针, 问题 5.1 定义的抽象语言概念。它使用以下的东西实现的 ……

2. 空指针的内部 (或运行期) 表达形式, 这可能并不是全零, 而且对不用的指针类型可能不一样。真正的值只有编译器开发者才关心。C 程序的作者永远看不到它们, 因为他们使用 ……

3. 空指针常数, 这是一个常整数 0 (参见问题 5.2)。它通常隐藏在 ……

4. NULL 宏, 它被定义为 0 (参见问题 5.4)。最后转移我们注意力到 ……

5. ASCII 空字符 (NUL), 它的确是全零, 但它和空指针除了在名称上以外, 没有任何必然关系; 而……

6. “空串” (null string), 它是内容为空的字符串 (""). 在 C 中使用空串这个术语可能令人困惑, 因为空串包括空字符 ('\0'),但不包括空指针, 这让我们绕了一个完整的圈子 ……

本文用短语 “空指针” (“null pointer”, 小写) 表示第一种含义, 标识 “0” 或短语 “空指针常数” 表示含义 3, 用大写 NULL 表示含义 4。

5.11 为什么有那么多关于空指针的疑惑？为什么这些问题如此经常地出现？

C 程序员传统上喜欢知道很多 (可能比他们需要知道的还要多) 关于机器实现的细节。空指针在源码和大多数机器实现中都用零来表示的事实导致了很多无根据的猜测。而预处理宏 (NULL) 的使用又似乎在暗示这个值可能在某个时刻或者在某种怪异的机器上会改变。“if(p == 0)” 这种结构又很容易被误认为在比较之前把 p 转成了整数类型, 而不是把 0 转成了指针类型。最后, 术语 “空” 的几种用法 (如上文问题 5.10 所列出的) 之间的区别又可能被忽视。

冲出这些迷惘的一个好办法是想象 C 使用一个关键字 (或许象 Pascal 那样, 用 “nil”) 作为空指针常数。编译器要么在源代码没有歧义的时候把 “nil” 转成适当类型的空指针, 或者有歧义的时候发出提示。现在事实上, C 语言的空指针常数关键字不是 “nil” 而是 “0”, 这在多数情况下都能正常工作, 除了一个未加修饰的 “0” 用在非指针上下文的时候, 编译器把它生成整数 0 而不是发出错误信息, 如果那个未加修饰的 0 是应该是空指针常数, 那么生成的程序不行。

5.12 我很困惑。我就是不能理解这些空指针一类的东西。

有两条简单规则你必须遵循:

1. 当你在源码中需要空指针常数时, 用 “0” 或 “NULL”。

2. 如果在函数调用中 “0” 或 “NULL” 用作参数, 把它转换成被调函数需要的指针类型

讨论的其它内容是关于别人的误解, 关于空指针的内部表达 (这你无需了解), 和关于函数原型的复杂性的。(考虑到这些复杂性, 我们发现规则 2 有些保守; 但它没什么害处。)理解问题 5.1、 5.2 和 5.4, 考虑问题 5.3、 5.7、 5.10 和 5.11 你就就会变得清晰。

5.13 考虑到有关空指针的所有这些困惑, 难道把要求它们内部表达都必须为 0 不是更简单吗？

如果没有其它的原因, 这样做会是没脑筋的。因为它会不必要地限制某些实现, 阻止它们用特殊的非全零值表达空指针, 尤其是当那些值可以为非法访问引发自动的硬件陷阱的时候。

况且, 这样的要求真正完成了什么呢？对空指针的正确理解不需要内部表达的知识, 无论是零还是非零。假设空指针内部表达为零不会使任何代码的编写更容易 (除了一些不动脑筋的 calloc() 调用; 参见问题 7.26)。用零作空指针的内部表达也不能消除在函数调用时的类型转换, 因为指针的**大小**可能和 int 型的大小依然不同。(如果象上文问题 5.11 所述, 用 “nil” 来请求空指针, 则用 0 作空指针的内部表达的想法都不会出现。)

5.14 说真的, 真有机器用非零空指针吗, 或者不同类型用不同的表达？

至少 PL/I, Prime 50 系列用段 07777, 偏移 0 作为空指针。后来的型号使用段 0, 偏移 0 作为 C 的空指针, 迫使类似 TCNP (测试 C 空指针) 的指令明显地成了现成的作出错误猜想的蹩脚 C 代码。旧些的按字寻址的 Prime 机器同样因为要求字节指针 (char *) 比字指针 (int *) 长而臭名昭著。

Data General 的 Eclipse MV 系列支持三种结构的指针格式 (字、字节和比特指针), C 编译器使用了其中之二：char * 和 void * 使用字节指针, 而其它的使用字指针。

某些 Honeywell-Bull 大型机使用比特模式 06000 作为 (内部的) 空指针。

CDC Cyber 180 系列使用包含环 (ring), 段和位移的 48 位指针。多数用户 (在环 11 上) 使用的空指针为 0xB00000000000。在旧的 1 次补码的 CDC 机器上用全 1 表示各种数据, 包括非法指针, 是十分常见的事情。

旧的 HP 3000 系列对字节地址和字地址使用不同的寻址模式; 正如上面的机器一样, 它因此也使用不同的形式表达 char * 和 void * 型指针及其它指针。

Symbolics Lisp 机器是一种标签结构, 它甚至没有传统的数字指针; 它使用 <NIL, 0> 对 (通常是不存在的 <对象, 偏移> 句柄) 作为 C 空指针。

根据使用的 “内存模式”, 8086 系列处理器 (PC 兼容机) 可能使用 16 位的数据指针和 32 位的函数指针, 或者相反。

一些 64 位的 Cray 机器在一个字的低 48 位表示 int *; char * 使用高 16 位的某些位表示一个字节在一个字中的偏移。

参考资料： [K&R1, Sec. A14.4 p. 211]。

5.15 运行时的 “空指针赋值” 错误是什么意思？

这个信息, 通常由 MS-DOS 编译器发出, 表明你通过空指针向非法地址 (可能是缺省数据段的偏移 0 位置) 写入了数据。参见问题 16.7。

第 6 章

数组和指针

6.1 我在一个源文件中定义了 char a[6], 在另一个中声明了 extern char *a 。为什么不行？

你在一个源文件中定义了一个字符串, 而在另一个文件中定义了指向字符的指针。extern char * 的申明不能和真正的定义匹配。类型 T 的指针和类型 T 的数组并非同种类型。请使用 extern char a[ ]。

参考资料： [ISO, Sec. 6.5.4.2]; [CT&P, Sec. 3.3 pp. 33-4, Sec. 4.5 pp. 64-5]。

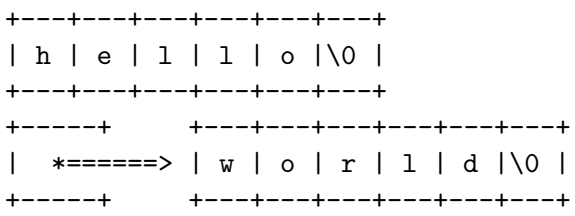
6.2 可是我听说 char a[ ] 和 char *a 是一样的。

并非如此。(你所听说的应该跟函数的形式参数有关；参见问题 6.4) 数组不是指针。数组定义 char a[6] 请求预留 6 个字符的位置, 并用名称 “a” 表示。也就是说, 有一个称为 “a” 的位置, 可以放入 6 个字符。而指针申明 char *p, 请求一个位置放置一个指针, 用名称 “p” 表示。这个指针几乎可以指向任何位置: 任何字符和任何连续的字符, 或者哪里也不指(参见问题 5.1 和 1.10)。

一个图形胜过千言万语。声明

```
char a[] = "hello";
char *p = "world";
```

将会初始化下图所示的数据结果:



根据 x 是数组还是指针, 类似 x[3] 这样的引用会生成不同的代码。认识到这一点大有裨益。以上面的声明为例, 当编译器看到表达式 a[3] 的时候, 它生成代码从 a 的位置开始跳过 3 个, 然后取出那个字符. 如果它看到 p[3], 它生成代码找到 “p” 的位置, 取出其中的指针值, 在指针上加 3 然后取出指向的字符。换言之, a[3] 是**名为** a 的对象 (的起始位置) 之后 3 个位置的值, 而 p[3] 是 p **指向**的对象的 3 个位置之后的值. 在上例中, a[3] 和 p[3] 碰巧都是 ’l’ , 但是编译器到达那里的途径不尽相同。本质的区别在于类似 a 的数组和类似 p 的指针一旦在表达式中出现就会

按照不同的方法计算, 不论它们是否有下标。下一问题继续深入解释。参见问题 1.13。

参考资料： [K&R2, Sec. 5.5 p. 104]; [CT&P, Sec. 4.5 pp. 64-5]。

6.3 那么, 在 C 语言中 “指针和数组等价” 到底是什么意思？

在 C 语言中对数组和指针的困惑多数都来自这句话。说数组和指针 “等价” 不表示它们相同, 甚至也不能互换。它的意思是说数组和指针的算法定义可以用指针方便的访问数组或者模拟数组。

特别地, 等价的基础来自这个关键定义:

> 一个 T 的数组类型的左值如果出现在表达式中会蜕变为一个指向数组第一个成员的指针(除了三种例外情况); 结果指针的类型是 T 的指针。

这就是说, 一旦数组出现在表达式中, 编译器会隐式地生成一个指向数组第一个成员地指针, 就像程序员写出了 &a[0] 一样。例外的情况是, 数组为 sizeof 或 & 操作符的操作数, 或者为字符数组的字符串初始值。

作为这个这个定义的后果, 编译器并那么不严格区分数组下标操作符和指针。在形如 a[i] 的表达式中, 根据上边的规则, 数组蜕化为指针然后按照指针变量的方式如 p[i] 那样寻址, 如问题 6.2 所述, 尽管最终的内存访问并不一样。如果你把数组地址赋给指针:

```
p = a;
```

那么 p[3] 和 a[3] 将会访问同样的成员。

参见问题 6.6 和 6.11。

参考资料： [K&R1, Sec. 5.3 pp. 93-6]; [K&R2, Sec. 5.3 p. 99]; [ISO, Sec. 6.2.2.1, Sec. 6.3.2.1, Sec. 6.3.6]; [H&S, Sec. 5.4.1 p. 124]。

6.4 那么为什么作为函数形参的数组和指针申明可以互换呢？

这是一种便利。

由于数组会马上蜕变为指针, 数组事实上从来没有传入过函数。允许指针参数声明为数组只不过是为让它看起来好像传入了数组, 因为该参数可能在函数内当作数组使用。特别地, 任何声明 “看起来象” 数组的参数, 例如

```
void f(char a[])
{ ... }
```

在编译器里都被当作指针来处理, 因为在传入数组的时候,那正是函数接收到的.

```
void f(char *a)
{ ... }
```

这种转换仅限于函数形参的声明, 别的地方并不适用。如果这种转换令你困惑, 请避免它; 很多程序员得出结论, 让形参声明 “看上去象” 调用或函数内的用法所带来的困惑远远大于它所提供的方便。

参见问题 6.18。

参考资料： [K&R1, Sec. 5.3 p. 95, Sec. A10.1 p. 205]; [K&R2, Sec. 5.3 p. 100, Sec. A8.6.3 p. 218, Sec. A10.1 p. 226]; [ISO, Sec. 6.5.4.3, Sec. 6.7.1, Sec. 6.9.6]; [H&S, Sec. 9.3 p. 271]; [CT&P, Sec. 3.3 pp. 33-4]。

6.5 如果你不能给它赋值, 那么数组如何能成为左值呢？

ANSI C 标准定义了“可变左值”, 而数组不是。

参考资料: [ISO, Sec. 6.2.2.1]; [Rationale, Sec. 3.2.2.1]; [H&S, Sec. 7.1 p. 179]。

6.6 现实地讲, 数组和指针地区别是什么？

数组自动分配空间, 但是不能重分配或改变大小。指针必须明确赋值以指向分配的空间 (可能使用 malloc), 但是可以随意重新赋值 (即, 指向不同的对象), 同时除了表示一个内存块的基址之外, 还有许多其它的用途。

由于数组和指针所谓的等价性(参见问题 6.3), 数组和指针经常看起来可以互换, 而事实上指向 malloc 分配的内存块的指针通常被看作一个真正的数组(也可以用 [ ] 引用)。参见问题 6.11 和 6.13。但是, 要小心 sizeof。

参见问题 1.13 和 20.12。

6.7 有人跟我讲, 数组不过是常指针。

这有些过度单纯化了。数组名之所以为 “常数” 是因为它不能被赋值, 但是数组**不是**指针, 问题 6.2 的讨论和图画可以说明这个。参见问题 6.3 和 6.6。

6.8 我遇到一些 “搞笑” 的代码, 包含 5["abcdef"] 这样的 “表达式”。这为什么是合法的 C 表达式呢？

是的, 弗吉尼亚[1], 数组和下标在 C 语言中可以互换。这个奇怪的事实来自数组下标的指针定义, 即对于**任何**两个表达式 a 和 e, 只要其中一个是指针表达式而另一个为整数, 则 a[e] 和 *((a)+(e)) 完全一样。这种交换性在许多 C 语言的书中被看作值得骄傲的东西, 但是它除了在混乱 C 语言竞赛之外, 其实鲜有用武之地。

参考资料: [Rationale, Sec. 3.3.2.1]; [H&S, Sec. 5.4.1 p. 124, Sec. 7.4.1 pp. 186-7]。

[1]这里有个美国典故, 在 1897 年, 有个叫弗吉尼亚 (Virginia) 的八岁小女孩, 她对圣诞老人是否存在感到困惑, 因而写了封寻问信给《纽约太阳报》, 于是就有了 “Yes, Virginia, there is a Santa Claus” 这篇评论, 有兴趣的朋友可以在 http://www.hymnsandcarolsofchristmas.com/santa/virginia's_question.htm 找到相关资料。

6.9 既然数组引用会蜕化为指针, 如果 arr 是数组, 那么 arr 和 &arr 又有什么区别呢？

区别在于类型。

在标准 C 中, &arr 生成一个 “T 型数组” 的指针, 指向整个数组。在 ANSI 之前的 C 中, &arr 中的 & 通常会引起一个警告, 它通常被忽略。在所有的 C 编译器中, 对数组的简单引用(不包括 & 操作符)生成一个 T 的指针类型的指针, 指向数组的第一成员。参见问题 6.3, 6.10 和6.15。

参考资料: [ISO, Sec. 6.2.2.1, Sec. 6.3.3.2]; [Rationale, Sec. 3.3.3.2]; [H&S, Sec. 7.5.6 p. 198]。

6.10 我如何声明一个数组指针？

通常, 你不需要。当人们随便提到数组指针的时候, 他们通常想的是指向它的第一个元素的指针。

考虑使用指向数组某个元素的指针, 而不是数组的指针。类型 T 的数组蜕变成类型 T 的指针(参见问题 6.3), 这很方便; 在结果的指针上使用下标或增量就可以访问数组中单独的成员。而真正的数组指针, 在使用下标或增量时, 会跳过整个数组, 通常只在操作数组的数组时有用 —— 如果还有一点用的话。参见问题 6.15。

如果你真的需要声明指向整个数组的指针, 使用类似 “int (*ap)[N];” 这样的声明。其中 N 是数组的大小 (参见问题 1.7)。如果数组的大小未知, 原则上可以省略 N, 但是这样生成的类型, “指向大小未知的数组的指针”, 毫无用处。

参见问题 6.9。

参考资料: [ISO, Sec. 6.2.2.1]。

6.11 我如何在运行期设定数组的大小？我怎样才能避免固定大小的数组？

由于数组和指针的等价性 (参见问题 6.3), 可以用指向 malloc 分配的内存的指针来模拟数组。执行

```
#include <stdlib.h>
int *dynarray;
dynarray = malloc(10 * sizeof(int));
```

以后 (如果 malloc 调用成功), 你可以象传统的静态分配的数组那样引用 dynarry[i] (i 从 0 到 9)。唯一的区别是 sizeof 不能给出 “数组” 的大小。参见问题 1.12、 6.13 和7.9。

6.12 我如何声明大小和传入的数组一样的局部数组？

直到最近以前, 你都不能; C 语言的数组维度一直都是编译时常数。但是, C99 引入了变长数组(VLA), 解决了这个问题; 局部数组的大小可以用变量或其它表达

式设置, 可能也包括函数参数。(gcc 提供参数化数组作为扩展已经有些时候了。) 如果你不能使用 C99 或 gcc, 你必须使用 malloc(), 并在函数返回之前调用 free()。参见问题 6.11, 6.13, 6.16, 7.19 和7.27。

参考资料: [ISO, Sec. 6.4, Sec. 6.5.4.2]; [C9X, Sec. 6.5.5.2]。

6.13 我该如何动态分配多维数组？

传统的解决方案是分配一个指针数组, 然后把每个指针初始化为动态分配的“列”。以下为一个二维的例子:

```
#include <stdlib.h>

    int **array1 = malloc(nrows * sizeof(int *));
    for(i = 0; i < nrows; i++)
        array1[i] = malloc(ncolumns * sizeof(int));
```

当然, 在真实代码中, 所有的 malloc 返回值都必须检查。你也可以使用 sizeof(*array1) 和 sizeof(**array1) 代替 sizeof(int *) 和 sizeof(int)。

你可以让数组的内容连续, 但在后来重新分配列的时候会比较困难, 得使用一点指针算术:

```
    int **array2 = malloc(nrows * sizeof(int *));
    array2[0] = malloc(nrows * ncolumns * sizeof(int));
    for(i = 1; i < nrows; i++)
        array2[i] = array2[0] + i * ncolumns;
```

在两种情况下, 动态数组的成员都可以用正常的数组下标 arrayx[i][j] 来访问 (for 0 <= i <nrows 和 0 <= j <ncolumns)。

如果上述方案的两次间接因为某种原因不能接受, 你还可以同一个单独的动态分配的一维数组来模拟二维数组:

```
int *array3 = malloc(nrows * ncolumns * sizeof(int));
```

但是, 你现在必须手工计算下标, 用 array3[i * ncolumns + j] 访问第 i, j 个成员。使用宏可以隐藏显示的计算, 但是调用它的时候要使用括号和逗号, 这看起来不太象多维数组语法, 而且宏需要至少访问一维。参见问题 6.16。

另一种选择是使用数组指针:

```
int (*array4)[NCOLUMNS] = malloc(nrows * sizeof(*array4));
```

但是这个语法变得可怕而且运行时最多只能确定一维。

当然, 使用这些技术, 你都必须记住在不用的时候释放数组 (这可能需要多个步骤; 参见问题 7.20)。而且你可能不能混用动态数组和传统的静态分配数组。参见问题 6.17 和 6.15。

最后, 在 C99 中你可以使用变长数组。

所有这些技术都可以延伸到三维或更多的维数。

参考资料: [C9X, Sec. 6.5.5.2]。

6.14 有个灵巧的窍门: 如果我这样写 int realarray[10]; int *array = &realarray[-1]; 我就可以把 “array” 当作下标从 1 开始的数组。

尽管这种技术颇有吸引力 (而且在《Numerical Recipes in C》一书的旧版中使用过), 它却不完全符合 C 的标准。指针算术只有在指针所指的内存块之内, 或者指向虚构的 “终结” 元素后的一个时才有定义; 否则, **即使指针并未解参考**, 其行为仍然是未定义的。如果在用偏移作下标运算的时候生成了非法地址 (可能因为地址在经过某个内存段之后 “回绕”), 则这段代码会失败。

参考资料: [K&R2, Sec. 5.3 p. 100, Sec. 5.4 pp. 102-3, Sec. A7.7 pp. 205-6]; [ISO, Sec. 6.3.6]; [Rationale, Sec. 3.2.2.3]。

6.15 当我向一个接受指针的指针的函数传入二维数组的时候, 编译器报错了。

数组蜕化为指针的规则 (参见问题 6.3) **不能**递归应用。数组的数组 (即 C 语言中的二维数组) 蜕化为数组的指针, 而不是指针的指针。数组指针常常令人困惑, 需要小心对待; 参见问题 6.10。

如果你向函数传递二位数组:

```
int array[NROWS][NCOLUMNS];
f(array);
```

那么函数的声明必须匹配:

```
void f(int a[][NCOLUMNS])
{ ... }
```

或者

```
void f(int (*ap)[NCOLUMNS]) /* ap 是个数组指针 */
{ ... }
```

在第一个声明中, 编译器进行了通常的从 “数组的数组” 到 “数组的指针” 的隐式转换 (参见问题 6.3 和6.4); 第二种形式中的指针定义显而易见。因为被调函数并不为数组分配地址, 所以它并不需要知道总的大小, 所以行数 NROWS 可以省略。但数组的宽度依然重要, 所以列维度 NCOLUMNS (对于三维或多维数组, 相关的维度) 必须保留。

如果一个函数已经定义为接受指针的指针, 那么几乎可以肯定直接向它传入二维数组毫无意义。

参见问题 6.9 和 6.12。

参考资料: [K&R1, Sec. 5.10 p. 110]; [K&R2, Sec. 5.9 p. 113]; [H&S, Sec. 5.4.3 p. 126]。

6.16 我怎样编写接受编译时宽度未知的二维数组的函数?

这并非易事。一种办法是传入指向 [0][0] 成员的的指针和两个维数, 然后 “手工” 模拟数组下标。

```
void f2(int *aryp, int nrows, int ncolumns)
{ ... array[i][j] is accessed as aryp[i * ncolumns + j] ... }
```

这个函数可以用问题 6.15 的数组如下调用:

```
f2(&array[0][0], NROWS, NCOLUMNS);
```

但是, 必须注明的一点是, 用这种方法通过 “手工” 方式模拟下标的程序未能严格遵循 ANSI C 标准; 根据官方的解释, 当 x >= NCOLUMNS 时, 访问 &array[0][0][x] 的结果未定义。

C99 允许变长数组, 一旦接受 C99 扩展的编译器广泛流传以后, VLA 可能是首选的解决方案。gcc 支持可变数组已经有些时日了。

当你需要使用各种大小的多维数组的函数时, 一种解决方案是象问题 6.13 那样动态模拟所有的数组。

参见问题 6.15, 6.17, 6.12。

参考资料: [ISO, Sec. 6.3.6]; [C9X, Sec. 6.5.5.2]。

6.17 我怎样在函数参数传递时混用静态和动态多维数组?

没有完美的方法。假设有如下声明

```
int array[NROWS][NCOLUMNS];
int **array1;                 /* 不齐的    */
int **array2;                 /* 连续的    */
int *array3;                  /* "变平的" */
int (*array4)[NCOLUMNS];
```

指针的初始值如问题 6.13 的程序片段, 函数声明如下

```
void f1a(int a[][NCOLUMNS], int nrows, int ncolumns);
void f1b(int (*a)[NCOLUMNS], int nrows, int ncolumns);
void f2(int *aryp, int nrows, int ncolumns);
void f3(int **pp, int nrows, int ncolumns);
```

其中 f1a() 和 f1b() 接受传统的二维数组, f2() 接受 “扁平的” 二维数组, f3() 接受指针的指针模拟的数组 (参见问题 6.15 和 6.16), 下面的调用应该可以如愿运行:

```
f1a(array, NROWS, NCOLUMNS);
f1b(array, NROWS, NCOLUMNS);
f1a(array4, nrows, NCOLUMNS);
f1b(array4, nrows, NCOLUMNS);
f2(&array[0][0], NROWS, NCOLUMNS);
f2(*array, NROWS, NCOLUMNS);
f2(*array2, nrows, ncolumns);
f2(array3, nrows, ncolumns);
f2(*array4, nrows, NCOLUMNS);
f3(array1, nrows, ncolumns);
f3(array2, nrows, ncolumns);
```

下面的调用在大多数系统上可能可行, 但是有可疑的类型转换, 而且只有动态 ncolumns 和静态 NCOLUMNS 匹配才行:

```
f1a((int (*)[NCOLUMNS])(*array2), nrows, ncolumns);
f1a((int (*)[NCOLUMNS])(*array2), nrows, ncolumns);
f1b((int (*)[NCOLUMNS])array3, nrows, ncolumns);
f1b((int (*)[NCOLUMNS])array3, nrows, ncolumns);
```

同时必须注意向 f2() 传递 &array[0][0] (或者等价的 *array) 并不完全符合标准; 参见问题 6.16。

如果你能理解为何上述调用可行且必须这样书写, 而未列出的组合不行, 那么你对 C 语言中的数组和指针就有了**很**好的理解了。

为免受这些东西的困惑, 一种使用各种大小的多维数组的办法是令它们 “全部” 动态分配, 如问题 6.13 所述。如果没有静态多维数组 —— 如果所有的数组都按问题 6.13 的 array1 和 array2 分配 —— 那么所有的函数都可以写成 f3() 的形式。

6.18 当数组是函数的参数时, 为什么 sizeof 不能正确报告数组的大小?

编译器把数组参数当作指针对待 (参见问题 6.4), 因而报告的时指针的大小。

参考资料: [H&S, Sec. 7.5.2 p. 195]。

第 7 章

内存分配

7.1 为什么这段代码不行？char *answer; printf("Type something:\n"); gets(answer); printf("You typed \"%s\"\n", answer);

指针变量 answer, 传入 gets(), 意在指向保存得到的应答的位置, 但却没有指向任何合法的位置。换言之, 我们不知道指针 answer 指向何处。因为局部变量没有初始化, 通常包含垃圾信息, 所以甚至都不能保证 answer 是一个合法的指针。参见问题 1.10 和 5.1。

改正提问程序的最简单方案是使用局部数组, 而不是指针, 让编译器考虑分配的问题:

```
#include <stdio.h>
#include <string.h>

    char answer[100], *p;
    printf("Type something:\n");
    fgets(answer, sizeof answer, stdin);
    if((p = strchr(answer, '\n')) != NULL)
        *p = '\0';
    printf("You typed \"%s\"\n", answer);
```

本例中同时用 fgets() 代替 gets(), 以便 array 的结束符不被改写。参见问题 12.20。不幸的是, 本例中的 fgets() 不会象 gets() 那样自动地去掉结尾的 \n。也可以用 malloc() 分配 answer 缓冲区。

7.2 我的 strcat() 不行.我试了 char *s1 = "Hello, "; char *s2 = "world!"; char *s3 = strcat(s1, s2); 但是我得到了奇怪的结果。

跟前面的问题 7.1 一样, 这里主要的问题是没有正确地为连接结果分配空间。C 没有提供自动管理的字符串类型。C 编译器只为源码中明确提到的对象分配空间 (对于字符串, 这包括字符数组和串常量)。程序员必须为字符串连接这样的运行期操作的结果分配足够的空间, 通常可以通过声明数组或调用 malloc() 完成。

strcat() 不进行任何分配; 第二个串原样不动地附加在第一个之后。因此, 一种解决办法是把第一个串声明为数组:

```
char s1[20] = "Hello, ";
```

由于 strcat() 返回第一个参数的值, 本例中为 s1, s3 实际上是多余的; 在 strcat() 调用之后, s1 包含结果。

提问中的 strcat() 调用实际上有两个问题: s1 指向的字符串常数, 除了空间不足以放入连接的字符串之外, 甚至都不一定可写。参见问题 1.13。

参考资料: [CT&P, Sec. 3.2 p. 32]。

7.3 但是 strcat 的手册页说它接受两个 char * 型参数。我怎么知道 (空间) 分配的事情呢?

一般地说, 使用指针的时候, 你必须**总是**考虑内存分配, 除非明确知道编译器替你做了此事。如果一个库函数的文档没有明确提到内存分配, 那么通常需要调用者来考虑。

Unix 型的手册页顶部的大纲段落或 ANSI C 标准有些误导作用。那里展示的程序片段更像是实现者使用的函数定义而不是调用者使用的形式。特别地, 很多接受指针 (如结构指针或串指针) 的函数通常在调用时都用到某个由调用者分配的对象 (结构, 或数组 —— 参见问题 6.3 和 6.4) 的指针。其它的常见例子还有 time() (参见问题 13.10) 和 stat()。

7.4 我刚才试了这样的代码 char *p; strcpy(p, "abc"); 而它运行正常?怎么回事?为什么它没有崩溃?

你的运气来了, 我猜。未初始化的指针 p 所指向的随机地址恰好对你来说是可写的, 而且很显然也没有用于什么关键的数据。参见问题 11.35。

7.5 一个指针变量分配多少内存?

这是个挺有误导性的问题。当你象这样声明一个指针变量的时候,

```
char *p;
```

你 (或者, 更准确地讲, 编译器) 只分配了足够容纳指针本身的内存; 也就是说, 这种情况下, 你分配了 sizeof(char*) 个字节的内存。但你还没有分配**任何**让指针指向的内存。参见问题 7.1 和 7.2。

7.6 我有个函数, 本该返回一个字符串, 但当它返回调用者的时候, 返回串却是垃圾信息。

确保指向的内存已经正确分配了。例如, 确保你**没有**做下面这样的事情:

```
char *itoa(int n)
{
    char retbuf[20];                /* 错!  */
    sprintf(retbuf, "%d", n);
    return retbuf;                  /*  错! */
}
```

一种解决方案是把返回缓冲区声明为

```
static char retbuf[20];
```

本方案并非完美, 尤其是有问题的函数可能会递归调用, 或者会同时使用到它的多个返回值时。

参见问题 7.7, 12.19 和 20.1。

参考资料: [ISO, Sec. 6.1.2.4]。

7.7 那么返回字符串或其它集合的争取方法是什么呢？

返回指针必须是静态分配的缓冲区 (如问题 7.6 的答案所述), 或者调用者传入的缓冲区, 或者用 malloc() 获得的内存, 但**不能是**局部 (自动) 数组。

参见问题 20.1。

7.8 为什么在调用 malloc() 时, 我得到 “警告: 整数赋向指针需要类型转换” ?

你包含了 <stdlib.h> 或者正确声明了 malloc() 吗? 参见问题 1.8。

参考资料: [H&S, Sec. 4.7 p. 101]。

7.9 为什么有些代码小心地把 malloc 返回的值转换为分配的指针类型。

在 ANSI/ISO 标准 C 引入 void * 一般指针类型之前, 这种类型转换通常用于在不兼容指针类型赋值时消除警告 (或许也可能导致转换)。

在 ANSI/ISO 标准 C 下, 这些转换不再需要, 而起事实上现代的实践也不鼓励这样做, 因为它们可能掩盖 malloc() 声明错误时产生的重要警告; 参见上面的问题 7.8。(但是, 因为这样那样的原因, 为求与 C++ 兼容, C 程序中常常能见到这样的转换。在 C++ 中从 void * 的明确转换是必须的。)

参考资料: [H&S, Sec. 16.1 pp. 386-7]。

7.10 在调用 malloc() 的时候, 错误 “不能把 void * 转换为 int *” 是什么意思？

说明你用的是 C++ 编译器而不是 C 编译器。参见问题 7.9。

7.11 我见到了这样的代码 char *p = malloc(strlen(s) + 1); strcpy(p, s); 难道不应该是 malloc((strlen(s) + 1) * sizeof(char)) ?

永远也不必乘上 sizeof(char), 因为根据定义, sizeof(char) 严格为1。另一方面, 乘上 sizeof(char) 也没有害处, 有时候还可以帮忙为表达式引入 size_t 类型。参见问题 8.5。

参考资料: [ISO, Sec. 6.3.3.4]; [H&S, Sec. 7.5.2 p. 195]。

7.12 我如何动态分配数组？

参见问题 6.11 和 6.13。

7.13 我听说有的操作系统程序使用的时候才真正分配 malloc 申请的内存。这合法吗？

很难说。标准没有说系统可以这样做, 但它也没有明确说不能。
参考资料: [ISO, Sec. 7.10.3]。

7.14 我用一行这样的代码分配一个巨大的数组, 用于数字运算: double *array = malloc(300 * 300 * sizeof(double)); malloc() 并没有返回 null, 但是程序运行得有些奇怪, 好像改写了某些内存, 或者 malloc() 并没有分配我申请的那么多内存, 云云。

注意 300 * 300 是 90,000, 这在你乘上 sizeof(double) 以前就已经不能放入 16 位的 int 中了。如果你需要分配这样大的内存空间, 你可得小心为妙。如果在你的机器上 size_t (malloc() 接受的类型) 是 32位, 而 int 为 16 位, 你可以写 300 * (300 * sizeof(double)) 来避免这个问题。(参见问题 3.11)。否则, 你必须把你的数据结构分解为更小的块, 或者使用 32 位的机器或编译器, 或者使用某种非标准的内存分配函数。参见问题 19.27。

7.15 我的 PC 有 8 兆内存。为什么我只能分配 640K 左右的内存？

在 PC 兼容的分段结构下, 很难透明地分配超过 640K 的内存, 尤其是在 MS-DOS 下。参见问题 19.27。

7.16 我的程序总是崩溃, 显然在 malloc 内部的某个地方。但是我看不出哪里有问题。是 malloc() 有 bug 吗？

很不幸, malloc 的内部数据结构很容易被破坏, 而由此引发的问题会十分棘手。最常见的问题来源是向 malloc 分配的区域写入比所分配的还多的数据; 一个常见的 bug 是用 malloc(strlen(s)) 而不是 strlen(s) + 1。其它的问题还包括使用指向已经释放了的内存的指针, 释放未从 malloc 获得的内存, 或者两次释放同一个指针, 或者试图重分配空指针, 参见问题 7.25。

参见问题 7.23, 16.7 和 18.2。

7.17 动态分配的内存一旦释放之后你就不能再使用, 是吧？

是的。有些早期的 malloc() 文档提到释放的内存中的内容会 “保留”, 但这个欠考虑的保证并不普遍而且也不是 C 标准要求的。

几乎没有那个程序员会有意使用释放的内存, 但是意外的使用却是常有的事。考虑下面释放单链表的正确代码:

```
struct list *listp, *nextp;
for(listp = base; listp != NULL; listp = nextp) {
    nextp = listp->next;
    free(listp);
}
```

请注意如果在循环表达式中没有使用临时变量 nextp, 而使用 listp = listp->next会产生什么恶劣后果。

参考资料: [K&R2, Sec. 7.8.5 p. 167]; [ISO, Sec. 7.10.3]; [Rationale, Sec. 4.10.3.2]; [H&S, Sec. 16.2 p. 387]; [CT&P, Sec. 7.10 p. 95]。

7.18 为什么在调用 free() 之后指针没有变空？使用 (赋值, 比较) 释放之后的指针有多么不安全？

当你调用 free() 的时候, 传入指针指向的内存被释放, 但调用函数的指针值可能保持不变, 因为 C 的按值传参语义意味着被调函数永远不会永久改变参数的值。参见问题 4.4。

严格的讲, 被释放的指针值是无效的, 对它的**任何**使用, 即使没有解参照, 也可能带来问题, 尽管作为一种实现质量的表现, 多数实现都不会对无伤大雅的无效指针使用产生例外。

参考资料: [ISO, Sec. 7.10.3]; [Rationale, Sec. 3.2.2.3]。

7.19 当我 malloc() 为一个函数的局部指针分配内存时, 我还需要用 free() 明确的释放吗？

是的。记住指针和它所指向的东西是完全不同的。局部变量在函数返回时就会释放, 但是在指针变量这个问题上, 这表示指针被释放, 而**不是**它所指向的对象。用 malloc() 分配的内存直到你明确释放它之前都会保留在那里。一般地, 对于每一个 malloc() 都必须有个对应的 free() 调用。

7.20 我在分配一些结构, 它们包含指向其它动态分配的对象的指针。我在释放结构的时候, 还需要释放每一个下级指针吗？

是的。一般地, 你必须分别向 free() 传入 malloc() 返回的每一个指针, 仅仅一次 (如果它的确要被释放的话)。一个好的经验法则是对于程序中的每一个 malloc() 调用, 你都可以找到一个对应的 free() 调用以释放 malloc() 分配的内存。

参见问题 7.21。

7.21 我必须在程序退出之前释放分配的所有内存吗？

你不必这样做。一个真正的操作系统毫无疑问会在程序退出的时候回收所有的内存和其它资源。然而, 有些个人电脑据称不能可靠地释放内存, 从 ANSI/ISO C 的角度来看这不过是一个 “实现的质量问题”。

参考资料: [ISO, Sec. 7.10.3.2]。

7.22 我有个程序分配了大量的内存, 然后又释放了。但是从操作系统看, 内存的占用率却并没有回去。

多数 malloc/free 的实现并不把释放的内存返回操作系统, 而是留着供同一程序的后续 malloc() 使用。

7.23 free() 怎么知道有多少字节需要释放？

malloc/free 的实现会在分配的时候记下每一块的大小, 所以在释放的时候就不必再考虑了。

7.24 那么我能否查询 malloc 包, 可分配的最大块是多大？

不幸的是, 没有标准的或可移植的办法。某些编译器提供了非标准的扩展。

7.25 向 realloc() 的第一个参数传入空指针合法吗？你为什么要这样做？

ANSI C 批准了这种用法, 以及相关的 realloc(..., 0), 用于释放, 尽管一些早期的实现不支持, 因此可能不完全可移植。向 realloc() 传入置空的指针可以更容易地写出自开始的递增分配算法。

参考资料: [ISO, Sec. 7.10.3.4]; [H&S, Sec. 16.3 p. 388]。

7.26 calloc() 和 malloc() 有什么区别？利用 calloc 的零填充功能安全吗？free() 可以释放 calloc() 分配的内存吗, 还是需要一个 cfree()？

calloc(m, n) 本质上等价于

```
p = malloc(m * n);
memset(p, 0, m * n);
```

填充的零是全零, 因此**不能**确保生成有用的空指针值或浮点零值 (参见第 5 章)。free() 可以安全地用来释放 calloc() 分配的内存。

参考资料: [ISO, Sec. 7.10.3 to 7.10.3.2]; [H&S, Sec. 16.1 p. 386, Sec. 16.2 p. 386]; [PCS, Sec. 11 pp. 141,142]。

7.27 alloca() 是什么？为什么不提倡使用它？

在调用 alloca() 的函数返回的时候, 它分配的内存会自动释放。也就是说, 用 alloca 分配的内存在某种程度上局部于函数的 “堆栈帧” 或上下文中。

alloca() 不具可移植性, 而且在没有传统堆栈的机器上很难实现。当它的返回值直接传入另一个函数时会带来问题, 如 fgets(alloca(100), 100, stdin)。

由于这些原因, alloca() 不合标准, 不宜使用在必须广泛移植的程序中, 不管它可能多么有用。既然 C99 支持变长数组(VLA), 它可以用来更好的完成 alloca() 以前的任务。

参见问题 7.19。

参考资料: [Rationale, Sec. 4.10.3]。

第 8 章

字符和字符串

8.1 为什么 strcat(string, '!'); 不行？

字符和字符串的区别显而易见, 而 strcat() 用于连接**字符串**。

C 中的字符用它们的字符集值对应的小整数表示, 参见下边的问题 8.4。字符串用字符数组表示; 通常你操作的是字符数组的第一个字符的指针。二者永远不能混用。要为一个字符串增加 !, 需要使用

```
strcat(string, "!");
```

参见问题 1.13, 7.2 和 16.6。

参考资料: [CT&P, Sec. 1.5 pp. 9-10]。

8.2 我在检查一个字符串是否跟某个值匹配。为什么这样不行？char *string; ... if(string == "value") { /* string matches "value" */ ... }

C 中的字符串用字符的数组表示, 而 C 语言从来不会把数组作为一个整体操作 (赋值, 比较等)。上面代码段中的 == 操作符比较的是两个指针 —— 指针变量 string 的值和字符串常数 "value" 的指针值 —— 看它们是否相等, 也就是说, 看它们是否指向同一个位置。它们可能并不相等, 所以比较决不会成功。

要比较两个字符串, 一般使用库函数 strcmp():

```
if(strcmp(string, "value") == 0) {
        /* string matches "value" */
}
```

8.3 如果我可以写 char a[] = "Hello, world!"; 为什么我不能写 char a[14]; a = "Hello, world!";

字符串是数组, 而你不能直接用数组赋值。可以使用 strcpy() 代替:

```
strcpy(a, "Hello, world!");
```

参见问题 1.13, 4.1 和 7.2。

8.4 我怎么得到对应字符的数字 (字符集) 值, 或者相反？

在 C 语言中字符用它们的字符集值对应的小整数表示。因此, 你不需要任何转换函数: 如有你有字符, 你就有它的值。

数字字符和它们对应的 0-9 的数字之间相互转换时, 加上或减去常数 '0', 也就是说, '0' 的字符值。

参见问题 13.1 和 20.8。

8.5 我认为我的编译器有问题: 我注意到 sizeof('a') 是 2 而不是 1 (即, 不是 sizeof(char))。

可能有些令人吃惊, C 语言中的字符常数是 int 型, 因此 sizeof('a') 是 sizeof(int), 这是另一个与 C++ 不同的地方。参见问题 7.11。

参考资料: [ISO, Sec. 6.1.3.4]; [H&S, Sec. 2.7.3 p. 29]。

第 9 章

布尔表达式和变量

9.1 C 语言中布尔值的候选类型是什么？为什么它不是一个标准类型？我应该用 #define 或 enum 定义 true 和 false 值吗？

C 语言没有提供标准的布尔类型, 部分因为选一个这样的类型涉及最好由程序员决定的空间/时间折衷。(使用 int 可能更快, 而使用 char 可能更节省数据空间。然而, 如果需要和 int 反复转换, 那么小类型也可能生成更大或更慢的代码。)

使用 #define 还是枚举常数定义 true/false 可以随便, 无关大雅 (参见问题 2.16 和 17.8)。使用以下任何一种形式

```
#define TRUE  1                 #define YES 1
#define FALSE 0                 #define NO  0

enum bool {false, true};    enum bool {no, yes};
```

或直接使用 1 和 0 , 只要在同一程序或项目中一致即可。如果你的调试器在查看变量的时候能够显示枚举常量的名字, 可能使用枚举更好。

有些人更喜欢这样的定义

```
#define TRUE (1==1)
#define FALSE (!TRUE)
```

或者定义这样的 “辅助” 宏

```
#define Istrue(e) ((e) != 0)
```

但这样于事无益, 参见下边的问题 9.2, 5.9 和 10.1。

9.2 因为在 C 语言中所有的非零值都被看作 “真”, 是不是把 TRUE 定义为 1 很危险？如果某个内置的函数或关系操作符 “返回” 不是 1 的其它值怎么办？

C 语言中的确任何非零值都都被看作真, 但这仅限于 “输入”, 也就是说, 仅限于需要布尔值的地方。内建操作符生成布尔值时, 可以保证为 1 或 0。因此, 这样的测试

```
if((a == b) == TRUE)
```

能如愿运行 (只要 TRUE 为 1), 但显然这很傻。事实上, 跟 TRUE 和 FALSE 的跟 TRUE 和 FALSE 的显示比较都不合适, 因为有些库函数 (如 isupper(), isalpha() 等) 在成功时返回非零值, 但不一定为1。(再说, 如果你认为 “if((a == b)

== TRUE)" 比 "if(a == b)" 好, 为什么就此打住呢? 为什么不使用 "if(((a == b) == TRUE) == TRUE)" 呢?) 一般规则是只在向布尔变量赋值或函数参数中才使用 TRUE 和 FALSE (或类似的东西), 或者用于函数的返回值, 但决不用于比较。

预处理宏 TRUE 和 FALSE (当然还有 NULL) 只是用于增加代码可读性, 而不是因为其值可能改变。(参见问题 5.3 和 5.8。)

尽管使用 TRUE 和 FALSE 这样的宏 (或者 YES 和 NO) 看上去更清楚, 布尔值和定义在 C 语言中的复杂性让很多程序员觉得 TRUE 和 FALSE 宏不过更令人迷惑, 因而更喜欢使用 1 和 0。参见问题 5.7。

参考资料: [K&R1, Sec. 2.6 p. 39, Sec. 2.7 p. 41]; [K&R2, Sec. 2.6 p. 42, Sec. 2.7 p. 44, Sec. A7.4.7 p. 204, Sec. A7.9 p. 206]; [ISO, Sec. 6.3.3.3, Sec. 6.3.8, Sec. 6.3.9, Sec. 6.3.13, Sec. 6.3.14, Sec. 6.3.15, Sec. 6.6.4.1, Sec. 6.6.5]; [H&S, Sec. 7.5.4 pp. 196-7, Sec. 7.6.4 pp. 207-8, Sec. 7.6.5 pp. 208-9, Sec. 7.7 pp. 217-8, Sec. 7.8 pp. 218-9, Sec. 8.5 pp. 238-9, Sec. 8.6 pp. 241-4]; "What the Tortoise Said to Achilles"。

9.3 当 p 是指针时, if(p) 是合法的表达式吗?

是的. 参见问题 5.3。

第 10 章

C 预处理器

10.1 这些机巧的预处理宏: #define begin { #define end } 你觉得怎么样？

参见第 17 章。

10.2 怎么写一个一般用途的宏交换两个值？

对于这个问题没有什么好的答案。如果这两个值是整数, 可以使用异或的技术, 但是这对浮点值或指针却不行, 对同一个值也无能为力。(参见问题 3.4 和 20.14。) 如果希望这个宏用于任何类型 (通常的目标), 那么它不能使用临时变量, 因为不知道需要什么类型的临时变量 (即使知道也难以找出一个名字), 而且标准 C 也没有提供 typeof 操作符。

最好的全面解决方案可能就是忘掉宏这回事, 除非你还准备把类型作为第三个参数传入。

10.3 书写多语句宏的最好方法是什么？

通常的目标是书写一个象包含一个单独的函数调用语句的宏。这意味着 “调用者” 需要提供最终的分号, 而宏体则不需要。因此宏体不能为简单的括弧包围的复合语句, 因为如果这样, 调用的时候就会发生语法错 (明显是一个单独语句, 但却多了一个额外的分号), 就像在 if/else 语句的 if 分支中多了一个 else 分句一样。

所以, 传统的结局方案就是这样使用:

```
    #define MACRO(arg1, arg2) do {  \
            /* declarations */      \
            stmt1;                   \
            stmt2;                   \
            /* ... */                \
            } while(0)          /* 没有结尾的 ; */
```

当调用者加上分号后, 宏在任何情况下都会扩展为一个单独的语句。优化的编译器会去掉条件为 0 的 “无效” 测试或分支, 尽管 lint 可能会警告。

如果宏体内的语句都是简单语句, 没有声明或循环, 那么还有一种技术, 就是写一个单独的, 用一个或多个逗号操作符分隔的表达式。例如, 问题 10.22 的第一个 DEBUG() 宏。这种技术还可以 “返回” 一个值。

参考资料: [H&S, Sec. 3.3.2 p. 45]; [CT&P, Sec. 6.3 pp. 82-3]。

10.4 我第一次把一个程序分成多个源文件, 我不知道该把什么放到 .c 文件, 把什么放到 .h 文件。(“.h” 到底是什么意思？)

作为一般规则, 你应该把这些东西放入头 (.h) 文件中:

- 宏定义 (预处理 #defines)
- 结构、联合和枚举声明
- typedef 声明
- 外部函数声明(参见问题 1.4)
- 全局变量声明

当声明或定义需要在多个文件中共享时, 尤其需要把它们放入头文件中。特别是, 永远不要把外部函数原型放到 .c 文件中。参见问题 1.3。

另一方面, 如果定义或声明为一个 .c 文件私有, 则最好留在 .c 文件中。

参见问题 1.3 和 10.5。

参考资料: [K&R2, Sec. 4.5 pp. 81-2]; [H&S, Sec. 9.2.3 p. 267]; [CT&P, Sec. 4.6 pp. 66-7]。

10.5 一个头文件可以包含另一头文件吗？

这是个风格问题, 因此有不少的争论。很多人认为 “嵌套包含文件” 应该避免: 盛名远播的 “印第安山风格指南”(Indian Hill Style Guide, 参见问题 17.7) 对此嗤之以鼻; 它让相关定义更难找到; 如果一个文件被包含了两次, 它会导致重复定义错误; 同时他会令 makefile 的人工维护十分困难。另一方面, 它使模块化使用头文件成为一种可能 (一个头文件可以包含它所需要的一切, 而不是让每个源文件都包含需要的头文件); 类似 grep 的工具 (或 tags 文件) 使搜索定义十分容易, 无论它在哪里; 一种流行的头文件定义技巧是:

```
#ifndef HFILENAME_USED
#define HFILENAME_USED
... 头文件内容 ...
#endif
```

每一个头文件都使用了一个独一无二的宏名。这令头文件可自我识别，以便可以安全的多次包含; 而自动 Makefile 维护工具 (无论如何, 在大型项目中都是必不可少的) 可以很容易的处理嵌套包含文件的依赖问题。参见问题 17.8。

参考资料: [Rationale, Sec. 4.1.2]。

10.6 #include <> 和 #include "" 有什么区别？

<> 语法通常用于标准或系统提供的头文件, 而 "" 通常用于程序自己的头文件。

10.7 完整的头文件搜索规则是怎样的？

准确的的行为是由实现定义的，这就是应该有文档说明; 参见问题 11.32。通常, 用 <> 括起来的头文件会先在一个或多个标准位置搜索。用 "" 括起来的头文件会首先在 “当前目录” 中搜索, 然后 (如果没有找到) 再在标准位置搜索。

参考资料: [K&R2, Sec. A12.4 p. 231]; [ISO, Sec. 6.8.2]; [H&S, Sec. 3.4 p. 55]。

10.8 我在文件的第一个声明就遇到奇怪的语法错误, 但是看上去没什么问题。

可能你包含的最后一个头文件的最后一行缺一个分号。参见问题 2.14, 11.28 和 16.1。

10.9 我包含了我使用的库函数的正确头文件, 可是连接器还是说它没有定义。

参见问题 13.18。

10.10 我在编译一个程序, 看起来我好像缺少需要的一个或多个头文件。谁能发给我一份？

根据 “缺少的” 头文件的种类, 有几种情况。

如果缺少的头文件是标准头文件, 那么你的编译器有问题。你得向你的供货商或者精通你的编译器的人求助。

对于非标准的头文件问题更复杂一些。有些完全是系统或编译器相关的。某些是完全没有必要的, 而且应该用它们的标准等价物代替。例如, 用 <stdlib.h> 代替 <malloc.h>。其它的头文件, 如跟流行的附加库相关的, 可能有相当的可移植性。

标准头文件存在的部分原因就是提供适合你的编译器, 操作系统和处理器的定义。你不能从别人那里随便拷贝一份就指望它能工作, 除非别人跟你使用的是同样的环境. 你可能事实上有移植性问题 (参见第 19 章) 或者编译器问题。否则, 参见问题 18.18。

10.11 我怎样构造比较字符串的 #if 预处理表达式？

你不能直接这样做; #if 预处理指令只处理整数。有一种替代的方法是定义多个整数值不一样的宏, 用它们来实现条件比较。

参见问题 20.15。

参考资料: [K&R2, Sec. 4.11.3 p. 91; ISO Sec. 6.8.1]; [H&S, Sec. 7.11.1 p. 225]。

10.12 sizeof 操作符可以用于 #if 预编译指令中吗？

不行。预编译在编译过程的早期进行, 此时尚未对类型名称进行分析。作为替代, 可以考虑使用 ANSI 的 <limits.h> 中定义的常量, 或者使用 “配置” (configure) 脚本。更好的办法是, 书写与类型大小无关的代码; 参见问题 1.1。

10.13 我可以在 #include 行里使用 #ifdef 来定义两个不同的东西吗？

不行。你不能 “让预处理器自己运行”。你能做的就是根据 #ifdef 设置使用两个完全不同的单独 #define 行之一。

参考资料: [ISO, Sec. 6.8.3, Sec. 6.8.3.4]; [H&S, Sec. 3.2 pp. 40-1]。

10.14 对 typdef 的类型定义有没有类似 #ifdef的东西？

不幸的是, 没有。你必须保存一套预处理宏 (如 MY_TYPE_DEFINED) 来记录某个类型是否用 typdef 声明了。参见问题 10.12。

参考资料: [ISO, Sec. 5.1.1.2, Sec. 6.8.1]; [H&S, Sec. 7.11.1 p. 225]。

10.15 我如何用 #if 表达式来判断机器是高字节在前还是低字节在前？

恐怕你不能。(预处理运算仅仅使用长整型, 而且没有寻址的概念。) 你是否真的需要明确知道机器的字节顺序呢？通常写出与字节顺序无关的代码更好。

参考资料: [ISO, Sec. 6.8.1]; [H&S, Sec. 7.11.1 p. 225]。

10.16 我得到了一些代码, 里边有太多的 #ifdef。我不想使用预处理器把所有的 #include 和 #ifdef 都扩展开, 有什么办法只保留一种条件的代码呢？

有几个程序 unifdef, rmifdef 和 scpp (selective C preprocessor) 正是完成这种工作的。参见问题 18.18。

10.17 如何列出所有的预定义标识符？

尽管这是种常见的需求, 但却没有什么标准的办法。gcc 提供了和 -E 一起使用的 -dM 选项, 其它编译器也有类似的选项。如果编译器文档没有帮助, 那么可以使用类似 Unix 字符串工具的程序取出编译和预处理生成的可执行文件中的可打印字符串。请注意, 很多传统的系统相关的预定义标识符 (如 “unix”) 并不标准 (因为和用户的名字空间冲突), 因而会被删除或改名。

10.18 我有些旧代码, 试图用这样的宏来构造标识符 #define Paste(a, b) a/**/b 但是现在不行了。

这是有些早期预处理器实现 (如 Reiser) 的未公开的功能, 注释完全消失, 因而可以用来粘结标识符。但 ANSI 确认 (如 K&R所言) 注释用空白代替。然而对粘结标识符的需求却十分自然和广泛, 因此 ANSI 引入了一个明确定义的标识符粘结操作符 —— ##, 它可以象这样使用

```
#define Paste(a, b) a##b
```

参见问题 11.18。

参考资料: [ISO, Sec. 6.8.3.3]; [Rationale, Sec. 3.8.3.3]; [H&S, Sec. 3.3.9 p. 52]。

10.19 为什么宏 #define TRACE(n) printf("TRACE: %d\n", n) 报出警告 “用字符串常量代替宏” ? 它似乎应该把 TRACE(count); 扩展为 printf("TRACE: %d\count", count);

参见问题 11.19。

10.20 使用 # 操作符时, 我在字符串常量内使用宏参数有问题。

参见问题 11.18 和 11.19。

10.21 我想用预处理做某件事情, 但却不知道如何下手。

C 的预处理器并不是一个全能的工具。注意, 甚至都不能保证有一个单独的程序。与其强迫它做一些不适当的事情, 还不如考虑自己写一个专用的预处理工具。你可以很容易就得到一个类似 make(1) 那样的工具帮助你自动运行。

如果你要处理的不是 C 程序, 可以考虑使用一个多用途的预处理器。在多数 Unix 系统上有 m4 工具。

10.22 怎样写参数个数可变的宏 ?

一种流行的技巧是用一个单独的用括弧括起来的的 “参数” 定义和调用宏, 参数在宏扩展的时候成为类似 printf() 那样的函数的整个参数列表。

```
#define DEBUG(args) (printf("DEBUG: "), printf args)

if(n != 0) DEBUG(("n is %d\n", n));
```

明显的缺陷是调用者必须记住使用一对额外的括弧。

gcc 有一个扩展可以让函数式的宏接受可变个数的参数。但这不是标准。另一种可能的解决方案是根据参数个数使用多个宏 (DEBUG1, DEBUG2, 等等), 或者用逗号玩个这样的花招:

```
#define DEBUG(args) (printf("DEBUG: "), printf(args))
#define _ ,

DEBUG("i = %d" _ i);
```

C99 引入了对参数个数可变的函数式宏的正式支持。在宏 “原型” 的末尾加上符号 ...(就像在参数可变的函数定义中), 宏定义中的伪宏 __VA_ARGS__ 就会在调用是替换成可变参数。

最后, 你总是可以使用真实的函数, 接受明确定义的可变参数。参见问题 15.4 和 15.5。如果你需要替换宏, 使用一个函数和一个非函数式宏, 如 #define printf myprintf。

参考资料: [C9X, Sec. 6.8.3, Sec. 6.8.3.1]。

第 11 章

ANSI/ISO 标准 C

11.1 什么是 “ANSI C 标准” ?

1983 年, 美国国家标准协会 (ANSI) 委任一个委员会 X3J11 对 C 语言进行标准化。经过长期艰苦的过程, 该委员会的工作于 1989 年 12 月 14 日正式被批准为 ANSX3.159-1989 并于 1990 年春天颁布。ANSI C 主要标准化了现存的实践, 同时增加了一些来自 C++ 的内容 (主要是函数原型) 并支持多国字符集 (包括备受争议的三字符序列)。ANSI C 标准同时规定了 C 运行期库例程的标准。

一年左右以后, 该标准被接受为国际标准, ISO/IEC 9899:1990, 这个标准甚至在美国国内 (在这里它被称作 ANSI/ISO 9899-1990 [1992]) 代替了早先的 X3.159。作为一个 ISO 标准, 它会以发行技术勘误和标准附录的形式不断更新。

1994年, 技术勘误 1 (TC1) 修正标准中 40 处地方, 多数都是小的修改或明确, 而标准附录 1 (NA1) 增加了大约 50 页的新材料, 多数是规定国际化支持的新库函数。1995 年, TC2 增加了一些更多的小修改。

最近, 该标准的一个重大修订, “C99”, 已经完成并被接受。

该标准的数个版本, 包括 C99 和原始的 ANSI 标准, 都包括了一个 “基本原理” (Rational), 解释它的许多决定并讨论了很多细节问题, 包括本文中提及的某些内容。

11.2 我如何得到一份标准的副本 ?

可以用 18 美元从 www.ansi.org 联机购买一份电子副本 (PDF)。在美国可以从一下地址获取印刷版本

```
American National Standards Institute
11 W. 42nd St., 13th floor
New York, NY  10036  USA
(+1) 212 642 4900
```

和

```
Global Engineering Documents
15 Inverness Way E
Englewood, CO  80112  USA
(+1) 303 397 2715
(800) 854 7179  (U.S. & Canada)
```

其它国家, 可以联系适当的国内标准组织, 或 Geneva 的 ISO, 地址是:

```
ISO Sales
Case Postale 56
CH-1211 Geneve 20
Switzerland
```

或者参见 URL http://www.iso.ch 或查阅 comp.std.internat FAQ 列表, Standards.Faq。

由 Herbert Schild 注释的名不副实的《ANSI C 标准注解》包含 ISO 9899 的多数内容; 这本书由 Osborne/McGraw-Hill 出版, ISBN 为 0-07-881952-0, 在美国售价大约 40 美圆。有人认为这本书的注解并不值它和官方标准的差价: 里边错漏百出, 有些标准本身的内容甚至都不全。网上又很多人甚至建议完全忽略里边的注解。在 http://www.lysator.liu.se/c/schildt.html 可以找到 Clive Feather 对该注解的评论 ("注解的注解")。

"ANSI 基本原理" 的原本文本可以从 ftp://ftp.uu.net/doc/standards/ansi/X3.159-1989 匿名 ftp 下载 (参见问题 18.18), 也可以在万维网从 http://www.lysator.liu.se/c/rat/title.html 得到。这本基本原理由 Silicon Press 出版, ISBN为0-929306-07-4。

C9X 的公众评论草稿可以从 ISO/IEC JTC1/SC22/WG14 的网站得到, 地址为 http://www.dkuug.dk/JTC1/SC22/WG14/。

参见问题 11.3。

11.3 我在哪里可以找到标准的更新?

你可以在以下网站找到相关信息 (包括C9X草案): http://www.lysator.liu.se/c/index.html, http://www.dkuug.dk/JTC1/SC22/WG14/ 和 http://www.dmk.com/。

11.4 很多 ANSI 编译器在遇到以下代码时都会警告类型不匹配。 extern int func(float); int func(x) float x; { ...

你混用了新型的原型声明 "extern int func(float);" 和老式的定义 "int func(x) float x;"。通常这两种风格可以混同, 但是这种情况下不行。

旧的 C 编译器 (包括未使用原型和变长参数列表的 ANSI C) 会 "放宽" 传入函数的某些参数。浮点数被升为双精度浮点数, 字符和段整型被升为整型。对于旧式的函数定义, 参数值会在被调函数的内部自动转换为对应的较窄的类型, 如果在函数中那样声明了。

这个问题可以通过在定义中使用新型的语法一致性:

```
int func(float x) { ... }
```

或者把新型原型声明改成跟旧式定义一致。

```
extern int func(double);
```

这种情况下, 如果可能, 最好把就是定义也改成使用双精度数。

毫无疑问, 在函数参数和返回值中避免使用 “窄的” (char, short int 和 float) 类型要安全得多。

参见问题 1.8。

参考资料: [K&R1, Sec. A7.1 p. 186]; [K&R2, Sec. A7.3.2 p. 202]; [ISO, Sec. 6.3.2.2, Sec. 6.5.4.3]; [Rationale, Sec. 3.3.2.2, Sec. 3.5.4.3]; [H&S, Sec. 9.2 pp. 265-7, Sec. 9.4 pp. 272-3]。

11.5 能否混用旧式的和新型的函数语法？

这样做是合法的, 但是需要小心为妙, 特别参见问题 11.4。但是, 现代的做法是在声明和定义的时候都是用原型形式。旧式的语法被认为已经废弃, 所以某一天对它的官方支持可能会取消。

参考资料: [ISO, Sec. 6.7.1, Sec. 6.9.5]; [H&S, Sec. 9.2.2 pp. 265-7, Sec. 9.2.5 pp. 269-70]。

11.6 为什么声明 extern int f(struct x *p); 报出了一个奇怪的警告信息 “结构 x 在参数列表中声明”？

与 C 语言通常的作用范围规则大相径庭的是, 在原型中第一次声明 (甚至提到) 的结构不能和同一源文件中的其它结构兼容, 它在原型的结束出就超出了作用范围。

要解决这个问题, 在同一源文件的原型之前放上这样的声明:

```
struct x;
```

它在文件范围内提供了一个不完整的结构 x 的声明, 这样, 后续的用到结构 x 的声明至少能够确定它们引用的是同一个结构 x。

参考资料: [ISO, Sec. 6.1.2.1, Sec. 6.1.2.6, Sec. 6.5.2.3]。

11.7 我不明白为什么我不能象这样在初始化和数组维度中使用常量: const int n = 5; int a[n];

const 限定词真正的含义是 “只读的”; 用它限定的对象是运行时 (同常) 不能被赋值的对象。因此用 const 限定的对象的值并**不完全是**一个真正的常量。在这点上 C 和 C++ 不一样。如果你需要真正的运行时常量, 使用预定义宏 #define (或enum)。

参考资料: [ISO, Sec. 6.4]; [H&S, Secs. 7.11.2,7.11.3 pp. 226-7]。

11.8 既然不能修改字符串常量, 为什么不把它们定义为字符常量的数组？

一个原因是太多的代码包含

```
char *p = "Hello, world!";
```

这样并不正确的语句。这样的语句要受诊断信息的困扰，但真正的问题却出现在改变 p 所指目的的任何企图。

参见问题 1.13。

11.9 “const char *p” 和 “char * const p” 有何区别？

“const char *p” (也可以写成 “char const *p”) 声明了一个指向字符常量的指针, 因此不能改变它所指向的字符; “char * const p” 声明一个指向 (可变) 字符的指针常量, 就是说, 你不能修改指针。

“从里到外” 看就可以理解它们; 参见问题 1.7。

参考资料: [ISO, Sec. 6.5.4.1]; [Rationale, Sec. 3.5.4.1]; [H&S, Sec. 4.4.4 p. 81]。

11.10 为什么我不能向接受 const char ** 的函数传入 char **？

你可以向接受 const-T 的指针的地方传入 T 的指针 (任何类型 T 都适用)。但是, 这个允许在带修饰的指针类型上轻微不匹配的规则 (明显的例外) 却不能递归应用, 而只能用于最上层。

如果你必须赋值或传递除了在最上层还有修饰符不匹配的指针, 你必须明确使用类型转换 (本例中, 使用 (const char **)), 不过, 通常需要使用这样的转换意味着还有转换所不能修复的深层次问题。

参考资料: [ISO, Sec. 6.1.2.6, Sec. 6.3.16.1, Sec. 6.5.3]; [H&S, Sec. 7.9.1 pp. 221-2]。

11.11 怎样正确声明 main()？

int main(), int main(void) 或者 int main(int argc, char *argv[]) (显然 argc 和 argv 的拼写可以随便)。参见问题 11.12 到 11.16。

参考资料: [ISO, Sec. 5.1.2.2.1, Sec. G.5.1]; [H&S, Sec. 20.1 p. 416]; [CT&P, Sec. 3.10 pp. 50-51]。

11.12 我能否把 main() 定义为 void, 以避免扰人的 “main无返回值” 警告？

不能。main() 必须声明为返回 int, 且没有参数或者接受适当类型的两个参数。如果你调用了 exit() 但还是有警告信息, 你可能需要插入一条冗余的 return 语句 (或者使用某种 “未到达” 指令, 如果有的话）。

把函数声明为 void 并不仅仅关掉了警告信息：它可能导致与调用者（对于 main(), 就是 C 运行期初始代码) 期待的不同的函数调用/返回顺序。

注意, 这里讨论的 main() 是相对于 “宿体” 的实现; 它们不适用于 “自立” 的实现, 因为它们可能连 main() 都没有。但是，自立的实现相对比较少，如果你在使用这样的系统, 你也许已经知道了。如果你从没听说过它们之间的不同, 你可能正在使用 “宿体” 的实现, 那我们的讨论就适用。

参考资料: [ISO, 5.1.2.2.1, Sec. G.5.1]; [H&S, Sec. 20.1 p. 416]; [CT&P, Sec. 3.10 pp. 50-51]。

11.13 可 main() 的第三个参数 envp 是怎么回事？

这是一个 (尽管很常见但却) 非标准的扩展。如果你真的需要用 getenv() 函数提供的标准方法之外的办法读写环境变量, 可能使用全局变量 environ 会更好 (尽管它也同样并不标准)。

参考资料: [ISO, Sec. G.5.1]; [H&S, Sec. 20.1 pp. 416-7]。

11.14 我觉得把 main() 声明为 void 不会失败, 因为我调用了 exit() 而不是 return , 况且我的操作系统也忽略了程序的退出/返回状态。

这跟 main() 函数返回与否, 或者是否使用返回状态都没有关系; 问题是如果 main() 声明得不对, 它的调用者 (运行期初始代码) 可能甚至都不能正确**调用**它 (因为可能产生调用习惯冲突; 参见问题 11.12)。

你的操作系统可能会忽略退出状态, 而 void main() 在你那里也可能可行, 但这不可移植而且不正确。

11.15 那么到底会出什么问题？真的有什么系统不支持 void main() 吗？

有人报告用 BC++4.5 编译的使用 void main() 的程序会崩溃。某些编译器 (包括 DEC C V4.1 和打开某些选项的 gcc) 会对 void main() 提出警告。

11.16 我一直用的那本书《熟练傻瓜C语言》总是使用 void main()。

可能这本书的作者把自己也归为目标读者的一员。很多书不负责任地在例子中使用 void main(), 并宣称这样是正确的。但他们错了。

11.17 从 main() 中, exit(status) 和返回同样的 status 真的等价吗？

是也不是。标准声称它们等价。但是如果在退出的时候需要使用 main() 的局部数据, 那么从 main() return 恐怕就不行了; 参见问题 16.4。少数非常古老不符合标准的系统可能对其中的某种形式有问题。最后, 在 main() 函数的递归调用时, 二者显然不能等价。

参考资料: [K&R2, Sec. 7.6 pp. 163-4]; [ISO, Sec. 5.1.2.2.3]。

11.18 我试图用 ANSI "字符串化" 预处理操作符 # 向信息中插入符号常量的值, 但它字符串化的总是宏的名字而不是它的值。

你可以用下面这样的两步方法迫使宏既字符串化又扩展:

```
#define Str(x) #x
#define Xstr(x) Str(x)
#define OP plus
char *opname = Xstr(OP);
```

这段代码把 opname 置为 "plus" 而不是 "OP"。

在使用符号粘接操作符 ## 连接两个宏的值 (而不是名字) 时也要采用同样的 "迂回战术"。

参考资料: [ISO, Sec. 6.8.3.2, Sec. 6.8.3.5]。

11.19 警告信息 "warning: macro replacement within a string literal" 是什么意思?

有些 ANSI 前的编译器/预处理器把下面这样的宏

```
#define TRACE(var, fmt) printf("TRACE: var = fmt\n", var)
```

解释为

```
TRACE(i, %d);
```

这样的调用会被扩展为

```
printf("TRACE: i = %d\n", i);
```

换言之, 字符串常量内部也作了宏参数扩展。

K&R 和标准 C 都**没有**定义这样的宏扩展。当你希望把宏参数转成字符串时, 你可以使用新的预处理操作符 # 和字符串常量连接 (ANSI 的另一个新功能):

```
#define TRACE(var, fmt) \
    printf("TRACE: " #var " = " #fmt "\n", var)
```

参见问题 11.18。

参考资料: [H&S, Sec. 3.3.8 p. 51]。

11.20 在我用 #ifdef 去掉的代码里出现了奇怪的语法错误。

在 ANSI C 中, 被 #if, #ifdef 或 #ifndef "关掉" 的代码仍然必须包含 "合法的预处理符号"。这意味着字符 " 和 ' 必须像在真正的 C 代码中那样严格配对, 且这样的配对不能跨行。特别要注意缩略语中的撇号看起来很像字符常量的开始。因此, 自然语言的注释和伪代码必须写在 "正式的" 注释分界符 /* 和 */ 中。但是请参见问题 20.18 和 10.21。

参考资料: [ISO, Sec. 5.1.1.2, Sec. 6.1]; [H&S, Sec. 3.2 p. 40]。

11.21 #pragma 是什么, 有什么用？

#pragam 指令提供了一种单一的明确定义的 “救生舱”, 可以用作各种 (不可移植的) 实现相关的控制和扩展: 源码表控制、结构压缩、警告去除 (就像 lint 的老 /* NOTREACHED */ 注释), 等等。

参考资料: [ISO, Sec. 6.8.6]; [H&S, Sec. 3.7 p. 61]。

11.22 “#pragma once” 是什么意思？我在一些头文件中看到了它。

这是某些预处理器实现的扩展用于使头文件自我识别; 它跟问题 10.5 中讲到的 #ifndef 技巧等价, 不过移植性差些。

11.23 a[3] = "abc"; 合法吗？它是什么意思？

尽管只在极其有限的环境下有用, 可它在 ANSI C (可能也包括一些 ANSI 之前的系统) 中是合法的。它声明了一个长度为 3 的数组, 把它的三个字符初始化为 'a', 'b' 和 'c', 但却**没有**通常的 '\0' 字符。因此该数组并不是一个真正的 C 字符串从而不能用在 strcpy, printf %s 等当中。

多数时候, 你应该让编译器计算数组初始化的初始值个数, 在初始值 “abc” 中, 计算得长度当然应该是 4。

参考资料: [ISO, Sec. 6.5.7]; [H&S, Sec. 4.6.4 p. 98]。

11.24 为什么我不能对 void* 指针进行运算？

编译器不知道所指对象的大小。在作运算之前, 把指针转化为 char * 或你准备操作的其它指针类型, 但是请参考问题 4.3。

参考资料: [ISO, Sec. 6.1.2.5, Sec. 6.3.6]; [H&S, Sec. 7.6.2 p. 204]。

11.25 memcpy() 和 memmove() 有什么区别？

如果源和目的参数有重叠, memmove() 提供有保证的行为。而 memcpy() 则不能提供这样的保证, 因此可以实现得更加有效率。如果有疑问, 最好使用 memmove()。

参考资料: [K&R2, Sec. B3 p. 250]; [ISO, Sec. 7.11.2.1, Sec. 7.11.2.2]; [Rationale, Sec. 4.11.2]; [H&S, Sec. 14.3 pp. 341-2]; [PCS, Sec. 11 pp. 165-6]。

11.26 malloc(0) 有什么用？返回一个控指针还是指向 0 字节的指针？

ANSI/ISO 标准声称它可能返回任意一种; 其行为由实现定义。

参考资料: [ISO, Sec. 7.10.3]; [PCS, Sec. 16.1 p. 386]。

11.27 为什么 ANSI 标准规定了外部标示符的长度和大小写限制？

问题在于连接器既不受 ANSI/ISO 标准的控制也不遵守 C 编译器开发者的规定。限制仅限于标识符开始的几个字符而不是整个标识符。在原来的 ANSI 标准中限制为 6 个字符, 但在 C99 中放宽到了 31 个字符。

参考资料: [ISO, Sec. 6.1.2, Sec. 6.9.1]; [Rationale, Sec. 3.1.2]; [C9X, Sec. 6.1.2]; [H&S, Sec. 2.5 pp. 22-3]。

11.28 我的编译对最简单的测试程序报出了一大堆的语法错误。

可能是个 ANSI 前的编译器, 不能接受函数原型或类似的东西。

参见问题 1.11, 10.8, 11.29 和 16.1。

11.29 为什么有些 ASNI/ISO 标准库函数未定义？我明明使用的就是 ANSI 编译器。

你很可能有一个接受 ANSI 语法的编译器, 但并没有安装兼容 ANSI 的头文件或运行库。事实上, 这种情形在使用非供货商提供的编译器, 如 gcc 时非常常见。参见问题 11.28, 12.22 和 13.19。

11.30 谁有把旧的 C 程序转化为 ANSI C 或相反的工具, 或者自动生成原型的工具？

有两个程序 protoize 和 unprotoize 可以在有原型和无原型的函数定义和声明之间相互转换。这些程序**不能**完全完成 “经典” C 和 ANSI C 之间的转换。这些程序是 FSF 的 GNU C 编译器发布的一部分; 参加问题 18.3。

GNU GhostScript 包提供了一个叫 ansi2knr 的程序。

从 ANSI C 向旧式代码转化之前, 请注意这样的转化不能总是正确和自动。ANSI C 引入了 K&R C 没有提供的诸多新功能和复杂性。你得特别小心有原型的函数调用; 也可能需要插入明确的类型转换。参加问题 11.4 和 11.28。

存在几个类型生成器, 其中多数都是对 lint 的修改。1992 年 3 月在 comp.sources.misc 上发布了一个叫做 CPROTO 的程序。还有一个叫做 “cextract” 的程序。很多供货商都会随他们的编译器提供类似的小工具。参见问题 18.18。但在为 “窄” 参数的旧函数生成原型可要小心; 参加问题 11.4。

11.31 为什么声称兼容 ANSI 的 Frobozz Magic C 编译器不能编译这些代码？我知道这些代码是 ANSI 的, 因为 gcc 可以编译。

许多编译器都支持一些非标准的扩展, gcc 尤甚。你能确认被拒绝的代码不依赖这样的扩展吗？通常用试验特定的编译器来确定一种语言的特性是个坏主意; 使用的标准可能允许变化, 而编译器也可能有错。参见问题 11.35。

11.32 人们好像有些在意实现定义 (implementation-defin-ed)、未明确 (unspecified) 和无定义 (undefined) 行为的区别。它们的区别到底在哪里？

简单地说: 实现定义意味着实现必须选择某种行为并提供文档。未明确意味着实现必须选择某种行为但不必提供文档。未定义意味着任何事情都可能发生。标准在任何情况下都不强加需求; 前两种情况下, 它有时建议一组可能的行为 (也可能要求从中选择一种)。

注意既然标准对无定义行为**没有**强制要求, 编译器就绝对可以做任何事情。特别地, 对程序其它部分的正常运行没有任何保证; 参见问题 3.2, 有一个相对简单的例子。

如果你对书写可移植代码有兴趣, 你可以忽略它们的区别, 因为通常你都希望避免依赖三种行为中的任何一种。

参见问题 3.8 和 11.34。

第四种不那么严格定义的行为是 “场景特定” (locale-specific)。

参考资料: [ISO, Sec. 3.10, Sec. 3.16, Sec. 3.17]; [Rationale, Sec. 1.6]。

11.33 一个程序的 “合法”, “有效” 或 “符合” 到底是什么意思？

简单地说, 标准谈到了三种符合: 符合程序、严格符合程序和符合实现。

“符合程序” 是可以由符合实现接受的程序。

“严格符合程序” 是完全按照标准规定使用语言, 不依赖任何实现定义、未确定或未定义功能的程序。

“符合实现” 是按标准声称的实现的程序。参考资料: [ISO, Sec.]; [Rationale, Sec. 1.7]。

11.34 我很吃惊, ANSI 标准竟然有那么多没有定义的东西。标准的唯一任务不就是让这些东西标准化吗？

某些构造随编译器和硬件的实现而变化, 这一直是 C 语言的一个特点。这种有意的不严格规定可以让编译器生成效率更高的代码, 而不必让所有程序为了不合理的情况承担额外的负担。因此, 标准只是把现存的实践整理成文。

编程语言标准可以看作是语言使用者和编译器实现者之间的协议。协议的一部分是编译器实现者同意提供, 用户可以使用的功能。而其它部分则包括用户同意遵守, 编译器实现者认为会被最受的规则。只要双方都恪守自己的保证, 程序就可以正确运行。如果任何一方违背它的诺言, 则结果肯定失败。

参见问题 11.35。

参考资料: [Rationale, Sec. 1.1]。

11.35 有人说 i = i++ 的行为是未定义的, 但是我刚在一个兼容 ANSI 的编译器上测试, 得到了我希望的结果。

面对未定义行为的时候, 包括范围内的实现定义行为和未确定行为, 编译器可以做任何实现, 其中也包括你所有期望的结果。但是依靠这个实现却不明智。参加问题 7.4, 11.31, 11.32 和 11.34。

第 12 章

标准输入输出库

12.1 这样的代码有什么问题？char c; while((c = getchar()) != EOF) ...

第一, 保存 getchar 的返回值的变量必须是 int 型。getchar() 可能返回任何字符值, 包括 EOF。如果把 getchar 的返回值截为 char 型, 则正常的字符可能会被错误的解释为 EOF, 或者 EOF 可能会被修改 (尤其是 char 型为无符号的时候), 从而永不出现。

参考资料: [K&R1, Sec. 1.5 p. 14]; [K&R2, Sec. 1.5.1 p. 16]; [ISO, Sec. 6.1.2.5, Sec. 7.9.1, Sec. 7.9.7.5]; [H&S, Sec. 5.1.3 p. 116, Sec. 15.1, Sec. 15.6]; [CT&P, Sec. 5.1 p. 70]; [PCS, Sec. 11 p. 157]。

12.2 我有个读取直到 EOF 的简单程序, 但是我如何才能在键盘上输入那个 "EOF" 呢？

其实, 你的 C 程序看到的 EOF 的值和你用键盘发出的文件结束按键组合之间没有任何直接联系。根据你的操作系统, 你可能使用不同的按键组合来表示文件结束, 通常是 Control-D 或 Control-Z。

12.3 为什么这些代码 while(!feof(infp)) { fgets(buf, MAXLINE, infp); fputs(buf, outfp); } 把最后一行复制了两遍？

在 C 语言中, 只有输入例程试图读并失败**以后**才能得到文件结束符。换言之, C 的 I/O 和 Pascal 的不一样。通常你只需要检查输入例程的返回值, 例如, fgets() 在遇到文件结束符的时候返回 NULL。实际上, 在任何情况下, 都完全没有必要使用 feof()。

参考资料: [K&R2, Sec. 7.6 p. 164]; [ISO, Sec. 7.9.3, Sec. 7.9.7.1, Sec. 7.9.10.2]; [H&S, Sec. 15.14 p. 382]。

12.4 我的程序的屏幕提示和中间输出有时显示在屏幕上, 尤其是当我用管道向另一个程序输出的时候。

在输出需要显示的时候最好使用明确的 fflush(stdout) 调用, 尤其是当显示的

文本没有 \n 结束符时。有几种机制会努力帮助你在 “适当的时机” 执行 fflush, 但这仅限于 stdout 为交互终端的时候。参见问题 12.21。

参考资料: [ISO, Sec. 7.9.5.2]。

12.5 我怎样不等待回车键一次输入一个字符？

参见问题 19.1。

12.6 我如何在 printf 的格式串中输出一个 ’%’？我试过 \%, 但是不行。

只需要重复百分号: %%。

\%不行, 因为反斜杠 \ 是**编译器**的转义字符, 而这里我们的问题最终是 printf 的转义字符。

参见问题 19.21。

参考资料: [K&R1, Sec. 7.3 p. 147]; [K&R2, Sec. 7.2 p. 154]; [ISO, Sec. 7.9.6.1]。

12.7 有人告诉我在 printf 中使用 %lf 不正确。那么, 如果 scanf() 需要 %lf, 怎么可以用在 printf() 中用 %f 输出双精度数呢？

printf 的 %f 标识符的确既可以输出浮点数又可以输出双精度数。根据 “缺省参数扩展” 规则, 不论范围内有没有原形都会在在类似 printf 的可变长度参数列表中采用, 浮点型的变量或扩展为双精度型, 因此 printf() 只会看到双精度数。printf() 的确接受 %Lf, 用于输出长双精度数。参见问题 12.11 和 15.2。

参考资料: [K&R1, Sec. 7.3 pp. 145-47, Sec. 7.4 pp. 147-50]; [K&R2, Sec. 7.2 pp. 153-44, Sec. 7.4 pp. 157-59]; [ISO, Sec. 7.9.6.1, Sec. 7.9.6.2]; [H&S, Sec. 15.8 pp. 357-64, Sec. 15.11 pp. 366-78]; [CT&P, Sec. A.1 pp. 121-33]。

12.8 对于 size_t 那样的类型定义, 当我不知道它到底是 long 还是其它类型的时候, 我应该使用什么样的 printf 格式呢？

把那个值转换为一个已知的长度够大的类型, 然后使用与之对应的 printf 格式。例如, 输出某种类型的长度, 你可以使用

```
printf("%lu", (unsigned long)sizeof(thetype));
```

12.9 我如何用 printf 实现可变的域宽度？就是说, 我想在运行时确定宽度而不是使用 %8d？

printf("%*d", width, x) 就能达到你的要求。

参见问题 12.14。

参考资料: [K&R1, Sec. 7.3]; [K&R2, Sec. 7.2]; [ISO, Sec. 7.9.6.1]; [H&S, Sec. 15.11.6]; [CT&P, Sec. A.1]。

12.10 如何输出在千位上用逗号隔开的数字？金额数字呢？

<locale.h> 提供了一些函数可以完成这些操作, 但是没有完成这些任务的标准方法。printf() 惟一一处对应 locale 的地方就是改变它的小数点字符。

参考资料: [ISO, Sec. 7.4]; [H&S, Sec. 11.6 pp. 301-4]。

12.11 为什么 scanf("%d", i) 调用不行？

传给 scanf() 的参数必须是指针。改为 scanf("%d", &i) 即可修正上面的问题。

12.12 为什么 char s[30]; scanf("%s", s); 不用 & 也可以？

需要**指针**; 并不表示一定需要 & 操作符。当你向 scanf() 传入一个指针的时候, 你不需要使用 &, 因为不论是否带 & 操作符, 数组总是以指针形式传入函数的。参见问题 6.3 和 6.4。

12.13 为什么这些代码 double d; scanf("%f", &d); 不行？

跟 printf() 不同, scanf() 用 %lf 代表双精度数, 用 %f 代表浮点数。参见问题 12.7。

12.14 怎样在 scanf() 格式串中指定可变的宽度？

不能: scanf() 格式串中的星号表示禁止赋值。你可以使用 ANSI 的字符串化和字符连接完成同样的事情, 或者你可以在运行时创建 scanf 格式串。

12.15 当我用 “%d\n” 调用 scanf 从键盘读取数字的时候, 好像要多输入一行函数才返回。

可能令人吃惊, \n 在 scanf 格式串中**不**表示等待换行符, 而是读取并放弃所有的空白字符。参见问题 12.18。

参考资料: [K&R2, Sec. B1.3 pp. 245-6]; [ISO, Sec. 7.9.6.2]; [H&S, Sec. 15.8 pp. 357-64]。

12.16 我用 scanf %d 读取一个数字, 然后再用 gets() 读取字符串, 但是编译器好像跳过了 gets() 调用！

scanf %d 不处理结尾的换行符。如果输入的数字后边紧接着一个换行符, 则换行符会被 gets() 处理。

作为一个一般规则, 你不能混用 scanf() 和 gets(), 或任何其它的输入例程的调用; scanf 对换行符的特殊处理几乎一定会带来问题。要么就用 scanf() 处理所有的输入, 要么干脆不用。

参见问题 12.18 和 12.20。

参考资料: [ISO, Sec. 7.9.6.2]; [H&S, Sec. 15.8 pp. 357-64]。

12.17 我发现如果坚持检查返回值以确保用户输入的是我期待的数值, 则 scanf() 的使用会安全很多, 但有的时候好像会陷入无限循环。

在 scanf() 转换数字的时候, 它遇到的任何非数字字符都会终止转换**并被保留在输入流中**。因此, 除非采用了其它的步骤, 那么未预料到的非数字输入会不断“阻塞” scanf(): scanf() 永远都不能越过错误的非数字字符而处理后边的合法数字字符。如果用户在数字格式的 scanf 如 %d 或 %f 中输入字符 ‘x’, 那么提示后并用同样的 scanf() 调用重试的代码会立即遇到同一个 ’x’。

参见问题 12.18。

参考资料: [ISO, Sec. 7.9.6.2]; [H&S, Sec. 15.8 pp. 357-64]。

12.18 为什么大家都说不要使用 scanf()？那我该用什么来代替呢？

scanf() 有很多问题 —— 参见问题 12.15, 12.16 和 12.17。而且, 它的 %s 格式有着和 gets() 一样的问题 (参见问题 12.20) —— 很难保证接收缓冲不溢出。

更一般地讲, scanf() 的设计使用于相对结构化的, 格式整齐的输入。设计上, 它的名称就是来自于 “scan formatted”。如果你注意到, 它会告诉你成功或失败, 但它只能提供失败的大略位置, 至于失败的原因, 就无从得知了。对 scanf() 多得体的错误恢复几乎是不可能的; 通常先用类似 fgets() 的函数读入整行, 然后再用 sscanf() 或其它技术解释。strtol(), strtok() 和 atoi() 等函数通常有用; 参见问题 13.4。如果你真的要用任何 scanf 的变体, 你要确保检查返回值, 以确定找到了期待的值。而使用 %s 格式的时候, 一定要小心缓冲区溢出。

参考资料: [K&R2, Sec. 7.4 p. 159]。

12.19 我怎样才知道对于任意的 sprintf 调用需要多大的目标缓冲区？怎样才能避免 sprintf() 目标缓冲区溢出？

当用于 sprintf() 的格式串已知且相对简单时, 你有时可以预测出缓冲区的大小。如果格式串中包含一个或两个 %s, 你可以数出固定字符的个数再加上对插入的字符串的 strlen() 调用的返回值。对于整形, %d 输出的字符数不会超过

```
((sizeof(int) * CHAR_BIT + 2) / 3 + 1)  /* +1 for '-' */
```

CHAR_BIT 在 <limits.h> 中定义, 但是这个计算可能有些过于保守了。它计算的是数字以八进制存储需要的字节数; 十进制的存储可以保证使用同样或更少的字节数。

当格式串更复杂或者在运行前未知的时候, 预测缓冲区大小会变得跟重新实现 sprintf 一样困难, 而且会很容易出错。有一种最后防线的技术, 就是 fprintf() 向一块内存区或临时文件输出同样的内容, 然后检查 fprintf 的返回值或临时文件的大小, 但请参见问题 19.14, 并提防写文件错误。

如果不能确保缓冲区足够大, 你就不能调用 sprintf(), 以防缓冲区溢出后改写其它的内存区。如果格式串已知, 你可以用 %.Ns 控制 %s 扩展的长度, 或者使用 %.*s, 参见问题 12.9。

要避免溢出问题, 你可以使用限制长度的 sprintf() 版本, 即 snprintf()。这样使用:

```
snprintf(buf, bufsize, "You typed \"%s\"", answer);
```

snprintf() 在几个 stdio 库中已经提供好几年了, 包括 GNU 和 4.4bsd。在 C99 中已经被标准化了。

作为一个额外的好处, C99 的 snprintf() 提供了预测任意 sprintf() 调用所需的缓冲区大小的方法。C99 的 snprintf() 返回它可能放到缓冲区的字符数, 而它又可以用 0 作为缓冲区大小进行调用。因此

```
nch = snprintf(NULL, 0, fmtstring, /* 其它参数 */ );
```

这样的调用就可以预测出格式串扩展后所需要的字符数。

另一个 (非标准的) 选择是 asprintf() 函数, 在 bsd 和 GNU 的 C 库中都有提供, 它调用 malloc 为格式串分配空间, 并返回分配内存区的指针。这样使用:

```
char *buf;
asprintf(&buf, "%d = %s", 42, "forty-two");
/* 现在, buf 指向含有格式串的 malloc 的内存 */
```

参考资料: [C9X, Sec. 7.13.6.6]。

12.20 为什么大家都说不要使用 gets() ?

跟 fgets() 不同, gets() 不能被告知输入缓冲区的大小, 因此不能避免缓冲区的溢出。标准库的 fgets() 函数对 gets() 作了很大的改进, 尽管它仍然不完善。如果真的可能输入很长的行, 还是需要仔细思考, 正确处理。参见问题 7.1 用 fgets() 代替 gets() 的代码片断。

参考资料: [Rationale, Sec. 4.9.7.2]; [H&S, Sec. 15.7 p. 356]。

12.21 为什么调用 printf() 之后 errno 内有 ENOTTY ?

如果 stdout 是终端, 很多 stdio 包都会对其行为进行细微的调整。为了做出判断, 这些实现会执行某些当 stdout 为终端时会失败的操作。尽管输出操作成功完

成, errno 还是会被置为 ENOTTY。注意, 只有当函数报告错误之后检查 errno 的内容才有意义。errno 在其它情况下也不保证为 0。

参考资料: [ISO, Sec. 7.1.4, Sec. 7.9.10.3]; [CT&P, Sec. 5.4 p. 73]; [PCS, Sec. 14 p. 254]。

12.22 fgetops/fsetops 和 ftell/fseek 之间有什么区别？ fgetops() 和 fsetops() 到底有什么用处？

ftell() 和 fseek() 用长整型表示文件内的偏移 (位置), 因此, 偏移量被限制在 20 亿 ($2^{31} - 1$) 以内。而新的 fgetpos() 和 fsetpos() 函数使用了一个特殊的类型定义 fpos_t 来表示偏移量。这个类型会适当选择, 因此, fgetpos() 和 fsetpos 可以表示任意大小的文件偏移。fgetpos() 和 gsetpos() 也可以用来记录多字节流式文件的状态。参见问题 1.2。

参考资料: [K&R2, Sec. B1.6 p. 248]; [ISO, Sec. 7.9.1, Secs. 7.9.9.1,7.9.9.3]; [H&S, Sec. 15.5 p. 252]。

12.23 如何清除多余的输入, 以防止在下一个提示符下读入？fflush(stdin) 可以吗？

fflush() 仅对输出流有效。因为它对 “flush” 的定义是用于完成缓冲字符的写入, 而对于输入流 fflush 并不是用于放弃剩余的输入。

参考资料: [ISO, Sec. 7.9.5.2]; [H&S, Sec. 15.2]。

12.24 既然 fflush() 不能, 那么怎样才能清除输入呢？

这取决于你要做什么。如果你希望丢掉调用 scanf() (参见问题 12.16 - 12.17) 之后所剩下的换行符和未预知的输入, 你可能需要重写你的 scanf() 或者换掉它, 参见问题 12.18。或者你可以用下边这样的代码吃掉一行中多余的字符

```
while((c = getchar()) != '\n' && c != EOF)
        /* 丢弃 */ ;
```

你也可以使用 curses 的 flushinp() 函数。

没有什么标准的办法可以丢弃标准输入流的未读取字符, 即使有, 那也不够, 因为未读取字符也可能来自其它的操作系统级的输入缓冲区。如果你希望严格丢弃多输入的字符 (可能是预测发出临界提示), 你可能需要使用系统相关的技术; 参加问题 19.1 和 19.2。

参考资料: [ISO, Sec. 7.9.5.2]; [H&S, Sec. 15.2]。

12.25 对某些路径文件名调用 fopen() 总是失败。

参见问题 19.21 和 19.22。

12.26 我想用 “r+” 打开一个文件, 读出一个字符串, 修改之后再写入, 从而就地更新一个文件。可是这样不行。

确保在写操作之前先调用 fseek, 回到你准备覆盖的字串的开始, 况且在读写 “+” 模式下的读和写操作之间总是需要 fseek 或 fflush。同时, 记住改写同样数量的字符, 而且在文本模式下改写可能会在改写处把文件长度截断, 因而你可能需要保存行长度。参见问题 19.17。

参考资料: [ISO, Sec. 7.9.5.3]。

12.27 怎样在程序里把 stdin 或 stdout 重定向到文件?

使用 freopen(), 但请参见下边的问题 12.28。

12.28 一旦使用 freopen() 之后, 怎样才能恢复原来的 stdout (或 stdin)?

没有什么好办法。如果你需要恢复回去, 那么最好一开始就不要使用 freopen()。可以使用你自己的可以随意赋值的输出 (输入) 流变量, 而不要去动原来的输出 (或输入) 流。

有一种不可移植的办法, 可以在调用 freopen() 之前保存流的信息, 以便其后恢复原来的流。一种办法是使用系统相关的调用如 dup(), dup2() 等。另一种办法是复制或查看 FILE 结构的内容, 但是这种方法完全没有可移植性而且很不可靠。

12.29 怎样同时向两个地方输出, 如同时输出到屏幕和文件?

直接做不到这点。但是你可以写出你自己的 printf 变体, 把所有的内容都输出两次。下边有个简单的例子:

```
#include <stdio.h>
#include <stdarg.h>

void f2printf(FILE *fp1, FILE *fp2, char *fmt, ...)
{
    va_list argp;
    va_start(argp, fmt); vfprintf(fp1, fmt, argp); va_end(argp);
    va_start(argp, fmt); vfprintf(fp2, fmt, argp); va_end(argp);
}
```

这里的 f2printf() 就跟 fprintf() 一样, 除了它接受两个文件指针并同时输出到两个文件。

参见问题 15.5。

12.30 怎样正确的读取二进制文件？我有时看到 0x0a 和 0x0d 混淆了，而且如果数据中包含 0x1a 的话，我好像会提前遇到 EOF。

读取二进制数据文件的时候你应该用 “rb” 调用 fopen(), 确保不会发生文本文件的解释。类似的, 写二进制文件时, 使用 “wb”。

注意文本/二进制区别只是发生在文件打开时: 一旦文件打开之后, 在其上调用何种 I/O 函数无关紧要。

参考资料: [ISO, Sec. 7.9.5.3]; [H&S, Sec. 15.2.1 p. 348]。

第 13 章

库函数

13.1 怎样把数字转为字符串 (与 atoi 相反)？有 itoa() 函数吗？

用 sprintf() 就可以了。不需担心用 sprintf() 会小题大作, 也不必担心会浪费运行时间或代码空间; 实践中它工作得挺好。参见问题 7.6 答案中的实例; 以及问题 8.4 和 12.19。

你也可以用 sprintf() 把长整形或浮点数转换成字符串 (使用 %ld 或 %f)。

参考资料: [K&R1, Sec. 3.6 p. 60]; [K&R2, Sec. 3.6 p. 64]。

13.2 为什么 strncpy() 不能总在目标串放上终止符 '\0'？

strncpy() 最初被设计为用来处理一种现在已经废弃的数据结构——定长, 不必 '\0' 结束的 "字符串"。strncpy 的另一个怪癖是它会用多个 '\0' 填充短串, 直到达到指定的长度。在其它环境中使用 strncpy() 有些麻烦, 因为你必须经常在目的串末尾手工加 '\0'。

你可以用 strncat 代替 strncpy 来绕开这个问题: 如果目的串开始时为空 (就是说, 如果你先用 *dest = '\0'), strncat() 就可以完成你希望 strncpy() 完成的事情。另外一个方法是用 sprintf(dest, "%.*s", n, source)。

如果需要复制任意字节 (而不是字符串), memcpy() 是个比 strncpy() 更好的选择。

13.3 为什么有些版本的 toupper() 对大写字符会有奇怪的反应？为什么有的代码在调用 toupper() 前先调用 tolower()？

老版的 toupper() 和 tolower() 不一定能够正常处理不需要转换的字符参数, 例如数字、标点或已经符合请求的字符。在 ANSI/ISO 标准 C 中, 这些函数保证对所有的字符参数正常处理。

参考资料: [ISO, Sec. 7.3.2]; [H&S, Sec. 12.9 pp. 320-1]; [PCS, p. 182]。

13.4 怎样把字符串分隔成用空白作间隔符的段？怎样实现类似传递给 main() 的 argc 和 argv？

标准中唯一用于这种分隔的函数是 strtok(), 虽然用起来需要些技巧, 而且不一定能做到你所要求的所有事。例如, 它不能处理引用。

参考资料: [K&R2, Sec. B3 p. 250]; [ISO, Sec. 7.11.5.8]; [H&S, Sec. 13.7 pp. 333-4]; [PCS, p. 178]。

13.5 我需要一些处理正则表达式或通配符匹配的代码。

确保你知道经典的正则表达式和文件名通配符的不同。前者的变体在 Unix 工具 ed 和 grep 等中使用, 后者的变体在多数操作系统中使用。

有许多匹配正则表达式的包可以利用。很多包都是用成对的函数, 一个 “编译” 正则表达式, 另一个 “执行” 它, 即用它比较字符串。查查头文件 <regex.h> 或 <regexp.h> 和函数 regcmp/regex, regcomp/regexec, 或 re_comp/re_exec。这些函数可能在一个单独的 regexp 库中。在 ftp://ftp.cs.toronto.edu/pub/regexp.shar.Z 或其它地方可以找到一个 Henry Spencer 开发的广受欢迎的 regexp 包, 这个包也可自由再发布。GNU 工程有一个叫做 rx 的包。参见问题 18.18。

文件名通配符匹配 (有时称之为 “globbing”) 在不同的系统上有不同的实现。在 Unix 上, shell 会在进程调用之前自动扩展通配符, 因此, 程序几乎从不需要专门考虑它们。在 MS-DOS 下的编译器中, 通常都可以在建立 argv 的时候连接一个用来扩展通配符的特殊目标文件。有些系统 (包括 MS-DOS 和 VMS) 会提供通配符指定文件的列表和打开的系统服务。参见问题 19.25 和 20.2。

13.6 我想用 strcmp() 作为比较函数, 调用 qsort() 对一个字符串数组排序, 但是不行。

你说的 “字符串数组” 实际上是 “字符指针数组”。qsort 比较函数的参数是被排序对象的指针, 在这里, 也就是字符指针的指针。然而 strcmp() 只接受字符指针。因此, 不能直接使用 strcmp()。写一个下边这样的间接比较函数:

```
/* 通过指针比较字符串 */
int pstrcmp(const void *p1, const void *p2)
{
    return strcmp(*(char * const *)p1, *(char * const *)p2);
}
```

比较函数的参数表示为 “一般指针” const void *。然后, 它们被转换回本来表示的类型 (指向字符指针的指针), 再复引用, 生成可以传入 strcmp() 的 char*。

不要被 [K&R2] 5.11 节 119-20页的讨论所误导, 那里讨论的不是标准库中的 qsort。

参考资料: [ISO, Sec. 7.10.5.2]; [H&S, Sec. 20.5 p. 419]。

13.7 我想用 qsort() 对一个结构数组排序。我的比较函数接受结构指针，但是编译器认为这个函数对于 qsort() 是错误类型。我要怎样转换这个函数指针才能避免这样的警告？

这个转换必须在比较函数中进行, 而函数必须定义为接受 “一般指针” (const void*) 的类型, 就象上文问题 13.6 中所讨论的。比较函数可能像这样:

```
int mystructcmp(const void *p1, const void *p2)
{
    const struct mystruct *sp1 = p1;
    const struct mystruct *sp2 = p2;
    /* 现在比较 sp1->whatever 和 sp2-> ... */
```

从一般指针到结构 mystruct 指针的转换过程发生在 sp1 = p1 和 sp2 = p2 的初始化中; 由于 p1 和 p2 都是 void 指针, 编译器隐式的进行了类型转换。

另一方面, 如果你对结构的指针进行排序, 则如问题 13.6 所示, 你需要间接使用：sp1 = *(struct mystruct * const *)p1。

一般而言, 为了让编译器 “闭嘴” 而进行类型转换是一个坏主意。编译器的警告信息通常希望告诉你某些事情, 忽略或轻易去掉会让你陷入危险, 除非你明确知道你在做什么。参见问题 4.5。

参考资料: [ISO, Sec. 7.10.5.2]; [H&S, Sec. 20.5 p. 419]。

13.8 怎样对一个链表排序？

有时侯, 有时侯, 在建立链表时就一直保持链表的顺序要简单些 (或者用树代替)。插入排序和归并排序算法用在链表最合适了。

如果你希望用标准库函数, 你可以分配一个暂时的指针数组, 填入链表中所有节点的地址, 再调用 qsort(), 最后依据排序后的数组重新建立链表。

参考资料: [Knuth, Sec. 5.2.1 pp. 80-102, Sec. 5.2.4 pp. 159-168]; [Sedgewick, Sec. 8 pp. 98-100, Sec. 12 pp. 163-175]。

13.9 怎样对多于内存的数据排序？

你可以用 “外部排序”法, [Knuth] 第三卷中有详情。基本的思想是对数据分段进行排序, 每次的大小尽可能多的填入内存中, 把排好序的数据段存入暂时文件中, 再归并它们。如果你的操作系统提供一个通用排序工具, 你可以从程序中调用: 参见问题 19.30 和 19.31。

参考资料: [Knuth, Sec. 5.4 pp. 247-378]; [Sedgewick, Sec. 13 pp. 177-187]。

13.10 怎样在 C 程序中取得当前日期或时间？

只要使用函数 time(), ctime(), localtime() 和/或 strftime() 就可以了。下面是个简单的例子:

```
#include <stdio.h>
#include <time.h>

int main()
{
    time_t now;
    time(&now);
    printf("It's %s", ctime(&now));
    return 0;
}
```

用函数 strftime() 可以控制输出的格式。如果你需要小于秒的解析度, 参见问题 19.36。

参考资料: [K&R2, Sec. B10 pp. 255-7]; [ISO, Sec. 7.12]; [H&S, Sec. 18]。

13.11 我知道库函数 localtime() 可以把 time_t 转换成结构 struct tm, 而 ctime() 可以把 time_t 转换成为可打印的字符串。怎样才能进行反向操作, 把 struct tm 或一个字符串转换成 time_t ?

ANSI C 提供了库函数 mktime(), 它把 struct tm 转换成 time_t。

把一个字符串转换成 time_t 比较难些, 这是由于可能遇到各种各样的日期和时间格式。某些系统提供函数 strptime(), 基本上是 strftime() 的反向函数。其它常用的函数有 partime() (与 RCS 包一起被广泛的发布) 和 getdate() (还有少数其它函数, 发布在 C 的新闻组)。参见问题 18.18。

参考资料: [K&R2, Sec. B10 p. 256]; [ISO, Sec. 7.12.2.3]; [H&S, Sec. 18.4 pp. 401-2]。

13.12 怎样在日期上加 N 天 ? 怎样取得两个日期的时间间隔 ?

ANSI/ISO 标准 C 函数 mktime() 和 difftime() 对这两个问题提供了一些有限的支持。mktime() 接受没有规格化的日期, 所以可以用一个日期的 struct tm 结构, 直接在 tm_mday 域进行加或减, 然后调用 mktime() 对年、月、日域进行规格化, 同时也转换成了 time_t 值。可以用 mktime() 来计算两个日期的 time_t 值, 然后用 difftime() 计算两个 time_t 值的秒数差分。

但是, 这些方法只有日期在 time_t 表达范围内才保证工作正常。对于保守的 time_t, 通常范围是从 1970 年到大约 2037 年; 注意有些 time_t 的表达不是按照 Unix 和 Posix 标准的。tm_mday 域是个 int, 所以日偏移量超出 32,736 就会上溢。还要注意, 在夏令时转换的时候, 一天并不是 24 小时, 所以不要假设可以用 86400 整除。

另一个解决的方法是用 "Julian 日期", 这可以支持更宽的时间范围。处理 Julian 日期的代码可以在以下地方找到: Snippets 收集 (参见问题 18.16); Simtel/Oakland 站点 (文件 JULCAL10.ZIP, 参见问题 18.18) 和 文献中提到的文章 "Date conversions" [Burki]。

参见问题 13.11, 20.27 和 20.28。

参考资料: [K&R2, Sec. B10 p. 256]; [ISO, Secs. 7.12.2.2,7.12.2.3]; [H&S, Secs. 18.4,18.5 pp. 401-2]; [Burki]。

13.13 我需要一个随机数生成器。

标准 C 库函数就有一个: rand()。你系统上的实现可能并不完美, 但写一个更好的并不是一件容易的事。

如果你需要实现自己的随机数生成器, 有许多这方面的文章可供参考; 象下面的文献或 sci.math.num-analysis 的 FAQ。网上也有许多这方面的包: 老的可靠的包有 r250, RANLIB 和 FSULTRA (参见问题 18.18), 还有由 Marsaglia, Matumoto 和 Nishimura 新近的成果 “Mersenne Twister”, 另外就是 Don Knuth 个人网页上收集的代码。

参考资料: [K&R2, Sec. 2.7 p. 46, Sec. 7.8.7 p. 168]; [ISO, Sec. 7.10.2.1]; [H&S, Sec. 17.7 p. 393]; [PCS, Sec. 11 p. 172]; [Knuth, Vol. 2 Chap. 3 pp. 1-177]; [P&M]。

13.14 怎样获得在一定范围内的随机数？

直接的方法是

```
rand() % N      /* 不好 */
```

试图返回从 0 到 N − 1 的数字。但这个方法不好, 因为许多随机数发生器的低位比特并**不**随机, 参见问题 13.16。一个较好的方法是:

```
(int)((double)rand() / ((double)RAND_MAX + 1) * N)
```

如果你不希望使用浮点, 另一个方法是:

```
rand() / (RAND_MAX / N + 1)
```

两种方法都需要知道 RAND_MAX, 而且假设 N 要远远小于 RAND_MAX。RAND_MAX 在 ANSI 里 #define 在 <stdlib.h>。

顺便提一下, RAND_MAX 是个**常数**, 它告诉你 C 库函数 rand() 的固定范围。你不可以设 RAND_MAX 为其它的值, 也没有办法要求 rand() 返回其它范围的值。

如果你用的随机数发生器返回的是 0 到 1 的浮点值, 要取得范围在 0 到 N − 1 内的整数, 只要将随机数乘以 N 就可以了。

参考资料: [K&R2, Sec. 7.8.7 p. 168]; [PCS, PCS Sec. 11 p. 172]。

13.15 每次执行程序，rand() 都返回相同顺序的数字。

你可以调用 srand() 来初始化模拟随机数发生器的种子, 用的值可以是真正随机数或至少是个变量, 例如当前时间。这儿有个例子:

```
#include <stdlib.h>
#include <time.h>

srand((unsigned int)time((time_t *)NULL));
```

不幸的是, 这个代码并不完美——其中, time() 返回的 time_t 可能是浮点值, 转换到无符号整数时有可能上溢, 这造成不可移植。参见问题: 19.36。

还要注意到, 在一个程序执行中多次调用 srand() 并不见得有帮助; 特别是不要为了试图取得 “真随机数” 而在每次调用 rand() 前都调用 srand()。

参考资料: [K&R2, Sec. 7.8.7 p. 168]; [ISO, Sec. 7.10.2.2]; [H&S, Sec. 17.7 p. 393]。

13.16 我需要随机的真/假值, 所以我用直接用 rand() % 2, 可是我得到交替的 0, 1, 0, 1, 0 ……

这是个低劣的伪随机数生成器, 在低位比特中不随机。很不幸, 某些系统就提供这样的伪随机数生成器。尝试用高位比特; 参见问题 13.14。

参考资料: [Knuth, Sec. 3.2.1.1 pp. 12-14]。

13.17 怎样产生标准分布或高斯分布的随机数？

这里有一个由 Marsaglia 首创 Knuth 推荐的方法:

```
#include <stdlib.h>
#include <math.h>

    double gaussrand()
    {
        static double V1, V2, S;
        static int phase = 0;
        double X;

        if(phase == 0) {
            do {
                double U1 = (double)rand() / RAND_MAX;
                double U2 = (double)rand() / RAND_MAX;

                V1 = 2 * U1 - 1;
                V2 = 2 * U2 - 1;
                S = V1 * V1 + V2 * V2;
            } while(S >= 1 || S == 0);

            X = V1 * sqrt(-2 * log(S) / S);
        } else
            X = V2 * sqrt(-2 * log(S) / S);
```

```
        phase = 1 - phase;

        return X;
    }
```

其它的方法参见本文的扩展版本, 参见问题 20.36。

参考资料: [Knuth, Sec. 3.4.1 p. 117]; [Marsaglia&Bray]; [Press et al., Sec. 7.2 pp. 288-290]。

13.18 我不断得到库函数未定义错误, 但是我已经 #inlude 了所有用到的头文件了。

通常, 头文件只包含外部说明。某些情况下, 特别是如果是非标准函数, 当你连接程序时, 需要指定正确的函数库以得到函数的**定义**。#include 头文件并不能给出定义。参见问题 10.10, 11.29, 13.19, 14.3 和 19.39。

13.19 虽然我在连接时明确地指定了正确的函数库, 我还是得到库函数未定义错误。

许多连接器只对对象文件和函数库进行一次扫描, 同时从函数库中提取适合当前未定义函数的模块。所以函数库和对象文件 (以及对象文件之间) 的连接顺序很重要; 通常, 你希望最后搜索函数库。例如, 在 Unix 系统中, 把 -l 参数放在命令行的后部。参见问题 13.20。

13.20 连接器说 _end 未定义代表什么意思?

这是个老 Unix 系统中的连接器所用的俏皮话。当有其它符号未定义时, 你才会得到 _end 未定义的信息, 解决了其它的问题, 有关 _end 的错误信息就会消失。参见问题 13.18 和 13.19。

13.21 我的编译器提示 printf 未定义! 这怎么可能?

据传闻, 某些用于微软视窗系统的 C 编译器不支持 printf()。你也许可以让这样的编译器认为你写的是 “控制台程序”, 这样编译器会打开 “控制台窗口” 从而支持 printf()。

第 14 章

浮点运算

14.1 一个 float 变量赋值为 3.1 时, 为什么 printf 输出的值为 3.0999999？

大多数电脑都是用二进制来表示浮点和整数的。在十进制里, 0.1 是个简单、精确的小数, 但是用二进制表示起来却是个循环小数 0.0001100110011 . . . 。所以 3.1 在十进制内可以准确地表达, 而在二进制下不能。

在对一些二进制中无法精确表示的小数进行赋值或读入再输出时, 也就是从十进制转成二进制再转回十进制, 你会观察到数值的不一致. 这是由于编译器二进制/十进制转换例程的精确度引起的, 这些例程也用在 printf 中。参见问题 14.6。

14.2 执行一些开方根运算, 可是得到一些疯狂的数字。

确定你用了 #include <math.h>, 以及正确说明了其它相关函数返回值为 double。另外一个需要注意的库函数是 atof(), 其原型说明在 <stdlib.h> 中。参见问题 14.3。

参考资料: [CT&P, Sec. 4.5 pp. 65-6]。

14.3 做一些简单的三角函数运算, 也引用了 #include <math.h>, 可是一直得到编译错误 “undefined: sin” (函数 sin 未定义)。

确定你真的连接了数学函数库 (math library)。例如, 在 Unix 或Linux 系统中, 有一个存在了很久的bug, 你需要把参数 -lm 加在编译或连接命令行的**最后**。参见问题 13.18, 13.19 和 14.2。

14.4 浮点计算程序表现奇怪, 在不同的机器上给出不同的结果。

首先阅读问题 14.2.

如果问题并不是那么简单, 那么回想一下, 电脑一般都是用一种浮点的格式来近似的模拟实数的运算, 注意是近似, 不是完全。下溢、误差的累积和其它非常规性是常遇到的麻烦。

不要假设浮点运算结果是精确的, 特别是别假设两个浮点值可以进行等价比较。也不要随意的引入 “模糊因素”; 参见问题 14.5。

这并不是 C 特有的问题, 其它电脑语言有一样的问题。浮点的某些方面被通常定义为 “中央处理器 (CPU) 是这样做的” (参见问题 11.34), 否则在一个没有 “正确” 浮点模型的处理器上, 编译器要被迫做代价非凡的仿真。

本文不打算列举在处理浮点运算上的潜在难点和合适的做法。一本好的有关数字编程的书能涵盖基本的知识。参见下面的参考资料。

参考资料: [K&P, Sec. 6 pp. 115-8]; [Knuth, Volume 2 chapter 4]; [Goldberg]。

14.5 有什么好的方法来验对浮点数在 “足够接近” 情况下的等值？

浮点数的定义决定它的绝对精确度会随着其代表的值变化, 所以比较两个浮点数的最好方法就要利用一个精确的阈值。这个阈值和作比较的浮点数值大小有关。不要用下面的代码：

```
double a, b;
...
if (a == b)     /* 错！ */
```

要用类似下列的方法：

```
#include <math.h>

if (fabs(a - b)  <= epsilon * fabs(a))
```

epsilon 被赋为一个选定的值来控制 “接近度”。你也要确定 a 不会为 0。

参考资料: [Knuth, Sec. 4.2.2 pp. 217-8]。

14.6 怎样取整数？

最简单、直接的方法：

```
(int)(x + 0.5)
```

这个方法对于负数并不正常工作。可以使用一个类似的方法：

```
(int)(x < 0 ? x - 0.5 : x + 0.5)
```

14.7 为什么 C 不提供乘幂的运算符？

因为提供乘幂指令的处理器非常少。 C 有一个 pow() 标准函数, 原型说明在 <math.h>。而对于小的正整数指数, 直接用乘法一般会更有效。

14.8 为什么我机器上的 <math.h> 没有预定义常数 M_PI？

这个常数不包含在标准内, 它应该是定义准确到机器精度的 π 值。如果你需要用到 π, 你需要自己定义, 或者用 4*atan(1.0) 或 acos(-1.0) 来计算出来。

参考资料: [PCS, Sec. 13 p. 237]。

14.9 怎样测试 IEEE NaN 以及其它特殊值？

许多实现高质量 IEEE 浮点的系统会提供简洁的工具去处理这些特殊值。例如, 在 <math.h> 以非标准扩展功能, 或可能以 <ieee.h> 或 <nan.h> 提供预定义常数, 及象 isnan() 这类的函数。这些工具的标准化进程正在进行中。一个粗陋但通常有效的测试 NaN 的方法：

```
#define isnan(x)    ((x) != (x))
```

虽然一些不支持 IEEE 的编译器可能会把这个判断优化掉。

C99 提供 isnan(), fpclassify() 及其它一些类别的例程。

必要时, 还可以用 sprintf() 格式化需测试的值, 在许多系统上, 它会产生 “NaN” 或 “Inf” 的字符串。你就可以比较了。

参见问题 19.38。

参考资料: [C9X, Sec. 7.7.3]。

14.10 在 C 中如何很好的实现复数？

这其实非常直接, 定义一个简单结构和相关的算术函数就可以了。C99 在标准中支持复数类别。参见问题 2.8, 14.11。

参考资料: [C9X, Sec. 6.1.2.5, Sec. 7.8]。

14.11 我要寻找一些实现以下功能的程序源代码：快速傅立叶变换 (FFT)、矩阵算术 (乘法、倒置等函数)、复数算术。

Ajay Shah 整理了一个免费算术软件列表。这个列表在互联网有广泛的归档。其中一个 URL 是 ftp://ftp.math.psu.edu/pub/FAQ/numcomp-free-c。参见问题 18.8, 18.10. 18.16, 18.18。

14.12 Turbo C 的程序崩溃, 显示错误为 “floating point formats not linked” (浮点格式未连接)。

一些在小型机器上使用的编译器, 包括 Turbo C (和 Richie 最初用在 PDP-11 上的编译器), 编译时会忽略掉某些它认为不需要的浮点支持。特别是用非浮点版的 printf() 和 scanf() 以节省一些空间, 也就是忽略处理 %e、%f 和 %g 的编码。

然而, Borland 用来确定程序是否使用了浮点运算的探测法并不充分, 程序员有时必需调用一个空浮点库函数 (例如 sqrt(), 或任何一个函数都可以) 以强制装载浮点支持。参见 comp.os.msdos.programmer FAQ 以获取更多信息。

第 15 章

可变参数

15.1 为什么调用 printf() 前, 必须要用 #include <stdio.h>？

为了把 printf() 的正确原型说明引入作用域。

对于用可变参数的函数, 编译器可能用不同的调用次序。例如, 如果可变参数的调用比固定参数的调用效率低。所以在调用可变参数的函数前, 它的原型说明必须在作用域内, 编译器由此知道要用不定长调用机制。在原型说明中用省略号 "..." 来表示可变参数。

参考资料: [ISO, Sec. 6.3.2.2, Sec. 7.1.7]; [Rationale, Sec. 3.3.2.2, Sec. 4.1.6]; [H&S, Sec. 9.2.4 pp. 268-9, Sec. 9.6 pp. 275-6]。

15.2 为什么 %f 可以在 printf() 参数中, 同时表示 float 和 double？他们难道不是不同类型吗？

"参数默认晋级" 规则适用于在可变参数中的可变动部分: char 和 short int 晋级到 int, float 晋级到 double。同样的晋级也适用于在作用域中没有原型说明的函数调用, 即所谓的 "旧风格" 函数调用, 参见问题 11.4。所以 printf 的 %f 格式总是得到 double。类似的, %c 总是得到 int, %hd 也是。参见问题 12.7, 12.13。

参考资料: [ISO, Sec. 6.3.2.2]; [H&S, Sec. 6.3.5 p. 177, Sec. 9.4 pp. 272-3]。

15.3 为什么当 n 为 long int, printf("%d", n); 编译时没有匹配警告？我以为 ANSI 函数原型可以防止这样的类型不匹配。

当一个函数用可变参数时, 它的原型说明没有也不能提供可变参数的数目和类型。所以通常的参数匹配保护不适用于可变参数中的可变部分。编译器不能执行内含的转换或警告不匹配问题[1]。

参见问题 5.2, 11.4, 12.7 和 15.2。

15.4 怎样写一个有可变参数的函数？

用 <stdarg.h> 提供的辅助设施。

[1]译者注: 现代的编译器 (例如 gcc), 如果打开编译警告参数, 编译器对标准中的可变参数函数 (printf, scanf ... 等) 会进行匹配测试。象问题中的源代码, 用 "gcc -Wall" 进行编译, 会给出这样的警告: "warning: int format, long int arg (arg 2)"

下面是一个把任意个字符串连接起来的函数, 结果存在 malloc 的内存中:

```
#include <stdlib.h>          /* 说明 malloc, NULL, size_t */
#include <stdarg.h>          /* 说明 va_ 相关类型和函数 */
#include <string.h>          /* 说明 strcat 等 */

char *vstrcat(const char *first, ...)
{
    size_t len;
    char *retbuf;
    va_list argp;
    char *p;

    if(first == NULL)
        return NULL;

    len = strlen(first);

    va_start(argp, first);

    while((p = va_arg(argp, char *)) != NULL)
        len += strlen(p);

    va_end(argp);

    retbuf = malloc(len + 1);   /* +1 包含终止符 \0 */

    if(retbuf == NULL)
        return NULL;            /* 出错 */

    (void)strcpy(retbuf, first);

    va_start(argp, first);            /* 重新开始扫描 */

    while((p = va_arg(argp, char *)) != NULL)
        (void)strcat(retbuf, p);

    va_end(argp);

    return retbuf;
}
```

调用如下:

```
    char *str = vstrcat("Hello, ", "world!", (char *)NULL);
```

注意最后一个参数的类型重置; 参见问题 5.2, 15.3。注意调用者要释放返回的存储空间, 那是用 malloc 分配的。

参考资料: [K&R2, Sec. 7.3 p. 155, Sec. B7 p. 254]; [ISO, Sec. 7.8]; [Rationale, Sec. 4.8]; [H&S, Sec. 11.4 pp. 296-9]; [CT&P, Sec. A.3 pp. 139-141]; [PCS, Sec. 11 pp. 184-5, Sec. 13 p. 242]。

15.5 怎样写类似 printf() 的函数, 再把参数转传给 printf() 去完成大部分工作？

用 vprintf(), vfprintf() 或 vsprintf()。

下面是一个 error() 函数, 它列印一个出错信息, 在信息前加入字符串 “error: ” 和在信息后加入换行符：

```
#include <stdio.h>
#include <stdarg.h>

void error(const char *fmt, ...)
{
    va_list argp;
    fprintf(stderr, "error: ");
    va_start(argp, fmt);
    vfprintf(stderr, fmt, argp);
    va_end(argp);
    fprintf(stderr, "\n");
}
```

参考资料: [K&R2, Sec. 8.3 p. 174, Sec. B1.2 p. 245]; [ISO, Secs. 7.9.6.7,7.9.6.8,7.9.6.9]; [H&S, Sec. 15.12 pp. 379-80]; [PCS, Sec. 11 pp. 186-7]。

15.6 怎样写类似 scanf() 的函数, 再把参数转传给 scanf() 去完成大部分工作？

C99 支持 vscanf(), vfscanf() 和 vsscanf(), C99 以前的标准不支持。

参考资料: [C9X, Secs. 7.3.6.12-14]。

15.7 怎样知道实际上有多少个参数传入函数？

这一段信息不可移植。一些旧系统提供一个非标准函数 nargs()。然而它的可信度值得怀疑, 因为它的典型返回值是参数的字节长度, 而不是参数的个数。结构、整数和浮点类型的值一般需要几个字节的长度。

任何接收可变参数的函数都应该可以从传入的**参数本身**来得到参数的数目。类 printf 函数从格式字符串中的格式说明符来确定参数个数, 就象 %d 这样的格式说明符。所以如果格式字符串和参数数目不符时, 此类函数会出错的很厉害。

还有一个常用的技巧, 如果所有的参数是同一个类型, 在参数列最后加一个标记值。通常用 0、-1 或适当类型转换的空指针, 参见问题 5.2 和 15.4 例子中 exec1() 和 vstrcat() 的用法。

最后, 如果类型是可预见的, 你可以加一个参数数目的参数。当然调用者通常是很不喜欢这种做法的。

参考资料: [PCS, Sec. 11 pp. 167-8]。

15.8 为什么编译器不让我定义一个没有固定参数项的可变参数函数？

标准 C 要求用可变参数的函数至少有一个固定参数项, 这样你才可以使用 va_start()。所以编译器**不会接受**下面定义的函数：

```
int f(...)
{
    ...
}
```

参见问题 15.9。

参考资料: [ISO, Sec. 6.5.4, Sec. 6.5.4.3, Sec. 7.8.1.1]; [H&S, Sec. 9.2 p. 263]。

15.9 我有个接受 float 的可变参函数, 为什么 va_arg(argp, float) 不工作？

“参数默认晋级” 规则适用于在可变参数中的可变动部分: 参数类型为 float 的总是晋级 (扩展) 到 double, char 和 short int 晋级到 int。所以 va_arg(arpg, float) 是错误的用法。应该总是用 va_arg(arpg, double)。同理, 要用 va_arg(argp, int) 来取得原来类型是 char, short 或 int 的参数。基于相同理由, 传给 va_start() 的最后一个 “固定” 参数项的类型不会被晋级。参见问题 11.4 和 15.2。

参考资料: [ISO, Sec. 6.3.2.2]; [Rationale, Sec. 4.8.1.2]; [H&S, Sec. 11.4 p. 297]。

15.10 va_arg() 不能得到类型为函数指针的参数。

宏 va_arg() 所用的类型重写不能很好地操作于象函数指针这类过度复杂的类型。但是如果你用 typedef 定义一个函数指针类型, 那就一切正常了。参见问题 1.7。

参考资料: [ISO, Sec. 7.8.1.2]; [Rationale, Sec. 4.8.1.2]。

15.11 怎样实现一个可变参数函数, 它把参数再传给另一个可变参数函数？

通常来说, 你做不到。理想情况下, 你应该提供另一个版本的函数, 这个函数接受 va_list 指针类型的参数。类似于 vfprintf(), 参见问题 15.5。如果所有的参数必须完整的传给另一个函数, 或者你不能重写另一个函数为一个接受 va_list 指针类型参数的函数, 这并没有一个可移植的解决方法。也许可以通过求助于机器的汇编语言来实现。参见问题 15.12。

15.12 怎样调用一个参数在执行是才建立的函数？

这没有一个保证工作或可移植的方法。如果你好奇, 可以问本文的编辑 (Steve Summit), 他有一些古怪的点子, 也许你可以试试……

也许你可以试着传一个无值型指针 (void *) 数组, 而不是一个参数序列。被调用函数遍历这个数组, 就象 main() 遍历 argv 一样。当然这一切都建立在你能控制所有的调用函数上。参见问题 19.35。

第 16 章

奇怪的问题

16.1 遇到不可理解的不合理语法错误, 似乎大段的程序没有编译。

检查是否有没有结束的注释, 不匹配的 #if/#ifdef/#ifndef/#else/#endif 指令, 又或者没有完成的引号。记得还要检查头文件。参见问题 2.14, 10.8 和 11.28。

16.2 为什么过程调用不工作？编译器似乎直接跳过去了。

代码是否看起来象这样：

```
myprocedure;    /* 我的过程 */
```

C 只有函数, 而函数调用总要用圆括号将参数括起来, 即使是无参数的函数。用下列代码:

```
myprocedure();
```

16.3 程序在执行用之前就崩溃了, 用调试器单步跟进, 在 main() 之前就死了。

也许你定义了一个或多个非常大的局部数组 (超过上千字节)。许多系统只有固定大小的堆栈, 即使那些自动动态堆栈分配的系统也会因为一次性要分配大段堆栈而失败。

一般对大规模数组, 定义为静态的数组会更好。如果由于递归的原因, 每次都需要一组新的数组, 可以用 malloc() 动态申请内存, 参见问题 1.11。

参见问题 11.12, 16.4, 16.5 和 18.4。

16.4 程序执行正确, 但退出时崩溃在 main() 最后一个语句之后。为什么会这样？

注意是否错误说明了 main(), 参见问题 2.14, 10.8, 11.12 和 11.14。是否把局部缓冲传给了 setbuf() 或 setvbuf()。又或者问题出在注册于 atexit() 的清理函数。参见问题 7.6 和 11.17。

参考资料: [CT&P, Sec. 5.3 pp. 72-3]。

16.5 程序在一台机器上执行完美, 但在另一台上却得到怪异的结果。更奇怪的是, 增加或去除调试的打印语句, 就改变了症状……

许多地方有可能出错。下面是一些通常的检查要点:

- 未初始化的局部变量, 参见问题 7.1。
- 整数上溢, 特别是在一些 16 比特的机器上, 一些中间计算结果可能上溢, 象 a * b / c, 参见问题 3.11。
- 未定义的求值顺序, 参见问题 3.1 到 3.5。
- 忽略了外部函数的说明, 特别是返回值不是 int 的函数, 或是参数 “缩小” 或可变的函数。参见问题 1.8, 11.4 和 15.1。
- 复引用空指针, 参见第 5 章。
- malloc/free 的不适当使用: 假设 malloc 的内存都被清零、已释放的内存还可用、再次释放已释放内存、malloc 的内部被破坏, 参见问题 7.16 和 7.17。
- 指针类常规问题, 参见问题 16.7。
- printf() 格式与参数不符, 特别是用 %d 输出 long int, 参见问题 12.7。
- 试图分配的内存大小超出一个 unsigned int 类型的范围, 特别是在内存有限的机器上, 参见问题 7.14和 19.27。
- 数组边界问题, 特别是暂时的小缓冲, 也许用于 sprinf() 来构造一个字符串, 参见问题 7.1 和 12.19。
- 错误的假设了 typedef 的映射类型, 特别是 size_t。
- 浮点问题, 参见问题 14.1 和 14.4。
- 任何你自己认为聪明的在特定机器上的机器代码生成小技巧。

正确使用函数原型说明能够捕捉到一些以上的问题。lint 会捕捉到更多。参见问题 16.3, 16.4 和 18.4。

16.6 为什么代码: char *p = "hello, worl!"; p[0] = 'H'; 会崩溃?

字符串实字并不总是可以修改的, 除非是用在字符数组的初试化。试用:

```
char a[] = "hello, world!";
```

参见问题 1.13。

参考资料: [ISO, Sec. 6.1.4]; [H&S, Sec. 2.7.4 pp. 31-2]。

16.7 “Segmentation violation”, “Bus error” 和 “General protection fault” 意味着什么？

通常, 这意味着你的程序试图访问不该访问的内存地址, 一般是由于堆栈出错或是不正确的使用指针。可能的原因有: 局部数组溢出 (用堆栈分配的自动变量); 不小心, 用了空指针 (参见问题 5.2 和 5.15)、未初始化指针、地址未对齐的指针或其它没有适当分配的指针 (参见问题 7.1 和 7.2); malloc 内部被破坏 (参见问题 7.16); 函数调用参数不匹配, 特别是如果用了指针, 两个可能出错的函数是 scanf() (参见问题 12.11) 和 fprintf() (确定他的第一个参数是 FILE *)。

参见问题 16.3 和 16.4。

第 17 章

风格

17.1 什么是 C 最好的代码布局风格？

K&R 提供了最常被抄袭的实例, 同时他并不要求大家沿用他的风格:

> 大括号的位置并不重要, 尽管人们对此有着执着的热情。我们在几种流行的风格中选了一种。选一个适合你的风格, 然后坚持使用这一风格。

保持布局风格对自己, 对邻近及通用源代码的一致比使之完美跟重要。如果你的编码环境 (本地习惯或公司政策) 没有建议一个风格, 你也不想发明自己的风格, 可以沿用 K&R 中的风格。各种缩进和大括号的放置之间的好与坏可以详尽而细致地探讨, 但本文不再次重复了。可以参见《印第安山风格指南》(Indian Hill Style Guide), 问题 17.7。

“好风格” 的品质并不简单, 它包含的内容远远不止代码的布局细节。不要把时间都花在格式上而忽略了更实质性的代码本身的质量。

参见问题 10.4。

参考资料: [K&R1, Sec. 1.2 p. 10]; [K&R2, Sec. 1.2 p. 10]。

17.2 用 if(!strcmp(s1, s2)) 比较两个字符串等值,是否是个好风格？

这并不是个很好的风格, 虽然这是个流行的习惯用法。如果两个字符串相等, 这个测试返回为真, 但 ! (“非”) 的使用, 容易引起误会, 以为测试不等值情况。

另一个选择是用一个宏:

```
#define Streq(s1, s2) (strcmp((s1), (s2)) == 0)
```

参见问题 17.8。

17.3 为什么有的人用 if (0 == x) 而不是 if (x == 0)？

这是用来防护一个通常错误的小技巧:

```
if (x = 0)
```

如果你养成了把常量放在 == 前面的习惯, 当你意外的把代码写成了:

```
if (0 = x)
```

那编译器就会报怨。明显的, 一些人会觉得记住反换测试比记住输入双 = 号容易。当然这个技巧只对和常量比较的情况有用。

参考资料: [H&S, Sec. 7.6.5 pp. 209-10]。

17.4 原型说明 extern int func __((int, int)); 中, 那些多出来的括号和下划线代表了什么？

这是为了可以在使用 ANSI 以前的编译器时, 关掉说明中的原型部分。这是技巧的一部分。

在别的地方, 宏 __ 被定义为类似下面的代码：

```
#ifdef __STDC__
#define __(proto) proto
#else
#define __(proto) ()
#endif
```

原型说明中额外的括号是为了让原型列表被当作宏的单一参数。

17.5 为什么有些代码在每次调用 printf() 前, 加了类型转换 (void)？

printf() 确实返回一个值, 虽然极少数程序员去检验每次调用的返回值。由于有些编译器和 lint 对于被丢弃的返回值会报警告, 清楚的用 (void) 作类型转换相当于说: “我决定忽略这次调用的返回值, 请继续对于其他忽略返回值的情况 (也许是不应该的) 提出警告。” 通常, 无值类型转换也用于 strcpy() 和 strcat() 的调用, 他们的返回值从不会令人惊讶。

参考资料: [K&R2, Sec. A6.7 p. 199]; [Rationale, Sec. 3.3.4]; [H&S, Sec. 6.2.9 p. 172, Sec. 7.13 pp. 229-30]。

17.6 什么是 “匈牙利标志法” (Hungarian Notation)？是否值得用？

匈牙利标志法是一种命名约定, 由 Charles Simonyi 发明。他把变量的类型 (或者它的预期使用) 等信息编码在变量名中。在某些圈子里, 它被高度热爱, 而在另一些地方, 它被严厉地批评。它的主要优势在于变量名就说明了它的类型或者使用。它的主要缺点在于类型信息并不值得放在变量名中。

参考资料: [Simonyi&Heller]。

17.7 哪里可以找到 “印第安山风格指南” (Indian Hill Style Guide) 及其它编码标准？

各种文档在匿名 ftp 都可以得到：

地址	文档及目录
ftp.cs.washington.edu	pub/cstyle.tar.Z (更新的印第安山风格指南, (Indian Hill Guide))
ftp.cs.toronto.edu	doc/programming (包括 Henry Spencer 的《C 程序员的十诫》(“10 Commandments for C Programmers”))
ftp.cs.umd.edu	pub/style-guide

也许你会对这些书感兴趣: 《The Elements of Programming Style》[K&P], 《Plum Hall Programming Guidelines》[Plum], 《C Style: Standards and Guidelines》[Straker]。参见文献 [21]。

参见问题 18.7。

17.8 有些人说 goto 是邪恶的, 我应该永不用它。那是否太极端了?

程序设计风格, 就象写作风格一样, 是某种程度的艺术, 不可以被僵化的教条所束缚。虽然风格的探讨经常都是围绕着这些条例。

对于 goto 语句, 很早以前, 就被注意到, 随意的使用 goto 会很快的导致象面糊一样难以维护的代码。然而, 不经思考就简单的禁止 goto 的使用, 并不能立即导至好程序。一个无规划的程序员可以不用任何 goto 语句而构造出复杂难解的代码, 也许使用奇怪的嵌套循环和布尔变量来取代 goto。

通常, 把这些程序设计风格的评论或者 “条例” 当作指导准则比当作条例要更好。当程序员理解这些指导准则所要实现的目标, 就会工作的更加之好。盲目的回避某种构造或者死套条例而不融会贯通, 最终还会导致这些条例试图避免的问题。

此外, 许多程序设计风格的意见只是意见。通常卷入 “风格战争” 是毫无意义的。某些问题 (象问题 5.3, 5.7, 9.2 和 10.5), 争辩的双方是不可能同意, 认同对方的不同或者是停止争论的。

第 18 章

工具和资源

注意: 本章中的信息比较旧, 有些可能已经过时了, 特别是各个公共包的URL。

18.1 常用工具列表。

工具	程序名 (参见问题 18.18)
C 交叉引用生成器	cflow, cxref, calls, cscope, xscope, ixfw
C 源代码美化器/美化打印	cb, indent, GNU indent, vgrind
版本控制和管理工具	CVS, RCS, SCCS
C 源代码扰乱器 (遮蔽器)	obfus, shroud, opqcp
“make” 从属关系生成器	makedepend, cc -M 或 cpp -M
计算源代码度规工具	ccount, Metre, lcount, csize; McCable and Associates 也有一个商业包出售
C 源代码行数计数器	可以用 UNIX 的标准工具 wc 作个大概的计算, 比用 grep -c ";" 要好
C 说明帮助 (cdecl)	见 comp.sources.unix 第14卷 (参见问题 18.18) 和 [K&R2]
原型发生器	参见问题 11.30
malloc 问题抓捕工具	参见问题 18.2
“选择性” 的 C 预处理器	参见问题 10.16
语言翻译工具	参见问题 11.30 和 20.23
C 校对机 (lint)	参见问题 18.5
C 编译器	参见问题 18.3

这个工具列表并不完全, 如果你知道有没列出的工具, 欢迎联系本表的维护者。

其它工具列表和关于它们的讨论可以在 Usenet 的新闻组 comp.compilers 和 comp.software-eng 找到。

参见问题 18.3 和 18.18。

18.2 怎样抓捕棘手的 malloc 问题？

有好几个调试工具包可以用来抓捕 malloc 问题。其中一个流行的工具是 Conor P. Cahill 的 dbmalloc, 公布在 comp.sources.misc 1992 年第 32 卷。还有 leak 公布在 comp.sources.unix 档案第 27 卷; “Snippets” 合集中的 JMalloc.c, JMalloc.h; MEMDEBUG (ftp://ftp.crpht.lu/pub/sources/memdebug); Electric Fence。参见问题 18.18。

还有一些商业调试工具, 对调试 malloc 等棘手问题相当有用:

- CodeCenter (Saber-C)，出品 Centerline Software (http://www.centerline.com/)
- Insight (now Insure?), 出品 ParaSoft Corporation (http://www.parasoft.com/)
- Purify, 出品 Rational Software (http://www-306.ibm.com/software/rational/, 原来是 Pure Software, 现在是 IBM 的一部分)
- ZeroFault, 出品 The ZeroFault Group (http://www.zerofault.com/)

18.3 有什么免费或便宜的编译器可以使用？

自由软件基金的 GNU C (gcc, http://gcc.gnu.org/) 是个流行而高质量的免费 C 编译器. djgpp (http://www.delorie.com/djgpp/) 是移植到 MS-DOS 的 GCC 版本。据我所知, 有移植到 Macs 和 Windwos 上的 GCC 版本。[1]

lcc 是另外一个流行的编译器 http://www.cs.virginia.edu/~lcc-win32/, http://www.cs.princeton.edu/software/lcc/）。

Power C 是 Mix Sotfware 公司提供的一个非常便宜的 MS-DOS 下的编译器。公司地址：1132 Commerce Drive, Richardson, TX 75801, USA, 214-783-6001。

ftp://ftp.hitech.com.au/hitech/pacific 是个 MS-DOS 下的试用 C 编译器。非商业用途的不一定要注册。

新闻组 comp.compilers 的档案中有许多有关各种语言的编译器、解释器、语法规则的信息。新闻组在 http://compilers.iecc.com/ 的档案包含了一个 FAQ 列表和免费编译器的目录。

参见问题 18.18。

18.4 刚刚输入完一个程序, 但它表现的很奇怪。你可以发现有什么错误的地方吗？

看看你是否能先用 lint 跑一遍 (用 -a, -c, -h, -p 或别的参数)。许多 C 编译器

[1]译者注: Windows 下有两个移植版本 cygwin (http://www.cygwin.com/) 和 MinGW (http://http://www.mingw.org/)。

实际上只是半个编译器, 它们选择不去诊断许多源程序中不会妨碍代码生成的难点。

参见问题 16.5、16.7 和 18.5。

参考资料: [LINT]。

18.5 哪里可以找到兼容 ANSI 的 lint？

PC-Lint 和 FlexeLint 是 Gimpel Software 公司的产品 (http://www.gimpel.com/)。

Unix System V 版本 4 的 lint 兼容 ANSI。可以从 UNIX Support Labs 或 System V 的销售商单独得到 (和其它 C 工具捆绑在一起)。

另外一个兼容 ANSI 的 lint 是 Splint (以前叫 lclint, http://www.splint.org/)。它可以作一些高级别的正式检验。

如果没有 lint, 许多现代的编译器可以作出几乎和 lint 一样多的诊断。许多网友推荐 gcc -Wall -pedantic。

18.6 难道 ANSI 函数原型说明没有使 lint 过时吗？

实际上不是。首先, 原型说明只有在它们存在和正确的情况下才工作。一个无心的错误原型说明比没有更糟。其次, lint 会检查多个源程序文档的一致性, 以及数据和函数的说明。最后, 像 lint 这样独立的程序在加强兼容的、可移植的代码惯例上会比任何特定的、特殊实现的、充满特性和扩展功能的编译器更加谨慎。

如果你确实要用函数原型说明而不是 lint 来作多文件一致性检查, 务必保证原型说明在头文件中的正确性。参见问题 1.3 和 10.4。

18.7 网上有哪些 C 的教程或其它资源？

有许多个:

在 http://cprog.tomsweb.net 有个 Tom Torfs 的不错的教程。

Christopher Sawtell 写的《给 C 程序员的便笺》(Notes for C programmers)。在下面的地址可以得到: ftp://svr-ftp.eng.cam.ac.uk/misc/sawtell_C.shar, ftp://garbo.uwasa.fi/pc/c-lang/c-lesson.zip, http://www.fi.uib.no/Fysisk/Teori/KURS/OTHER/newzealand.html。

Time Love 的《程序员的 C》(C for Programmers)。 http://www-h.eng.cam.ac.uk/help/tpl/languages/C/teaching_C/

The Coronado Enterprises C 教程在 Simtel 镜像点目录 `pub/msdos/c`, 或在 http://www.coronadoenterprises.com/tutorials/c/index.html。

Steve Holmes 的在线教程 http://www.strath.ac.uk/IT/Docs/Ccourse/。

Martin Brown 的网页有一些 C 教程的资料 http://www-isis.ecs.soton.ac.uk/computing/c/Welcome.html。

在一些 UNIX 的机器上, 在 shell 命令行, 可以试试"*learn c*"。注意教程可能比较旧了。

最后, 本 FAQ 的作者以前教授一些 C 的课程, 这些笔记都放在了网上 http://www.eskimo.com/~scs/cclass/cclass.html。

【不承诺申明: 我没有检阅这些我收集的教程, 它们可能含有错误。除了那个有我名字的教程, 我不能为它们提供保证。而这些信息会很快的变得过时, 也许, 当你读到这想用上面的地址时, 它们已经不能用了】

这其中的几个教程, 再加上许多其它 C 的信息, 可以从 http://www.lysator.liu.se/c/index.html 得到。

Vinit Carpenter 维护着一个学习 C 和 C++ 的资源列表, 公布在新闻组 comp.lang.c 和 comp.lang.c++, 也归档在本 FAQ 所在 (参见问题 20.36), 或者 http://www.cyberdiem.com/vin/learn.html。

参见问题 18.8、18.9 和 18.16。

18.8 哪里可以找到好的源代码实例, 以供研究和学习?

这里有几个连接可以参考：ftp://garbo.uwasa.fi/pc/c-lang/00index.txt, http://www.eskimo.com/~scs/src/。

小心, 网上也有数之不尽的非常糟糕的代码。不要从坏代码中"学习", 这是每个人都可以做到的, 你可以做的更好。参见问题 18.7, 18.10, 18.16 和 18.18。

18.9 有什么好的学习 C 的书? 有哪些高级的书和参考?

有无数有关 C 的书, 我们无法一一列出, 也无法评估所有的书。许多人相信最好的书, 也是第一本：由 Kernighan 和 Richie 编写的《The C programming Language》("K&R", 现在是第二版了 [K&R2])。对这本书是否适合初学者有不同的意见; 我们当中许多人是从这本书学的 C, 而且学的不错; 然而, 有些人觉得这本书太客观了点, 不大适合那些没有太多程序设计经验的人作为第一个教程。网上有一些评注和勘误表, 例如: http://www.csd.uwo.ca/~jamie/.Refs/.Footnotes/C-annotes.html, http://www.eskimo.com/~scs/cclass/cclass.html, http://cm.bell-labs.com/cm/cs/cbook/2ediffs.html。

许多活跃在新闻组 comp.lang.c 的人推荐 K.N. King 写的《C: A Modern Approach》。

一本极好的参考书是由 Samuel P. Harbison 和 Guy L. Steele 和写的《C: A Reference Manual》[H&S]。

虽然并不适合从头开始学 C, 本 FAQ (英文版)已经出版了, 见文献 [CFAQ]。

C 和 C++ 用户协会 (Association of C and C++, ACCU) 维护着一份很全面的有关 C/C++ 的书目评论 (http://www.accu.org/bookreviews/public/)。

参见问题 18.7。

18.10 哪里可以找到标准 C 函数库的源代码？

GNU 工程有一个完全实现的 C 函数库 (http://www.gnu.org/software/libc/)。另一个来源是由 P.J. Plauger 写的书《The Standard C Library》[Plauger], 然而它不是公共版权的。参见问题 18.8, 18.16 和 18.18。

18.11 是否有一个在线的 C 参考指南？

提供两个选择: http://www.cs.man.ac.uk/standard_c/_index.html, http://www.dinkumware.com/htm_cl/index.html

18.12 哪里可以得到 ANSI/ISO C 标准？

参见问题 11.2。

18.13 我需要分析和评估表达式的代码。

有两个软件包可用: “defunc”, 在 1993 年 12 月 公布于新闻组 comp.sources.misc (V41 i32,33), 1994 年 1 月公布于新闻组 alt.sources。可以在这个 URL 得到: ftp://sunsite.unc.edu/pub/packages/development/libraries/defunc-1.3.tar.Z; “parse”, 可以从 lamont.ldgo.columbia.edu 得到。其它选择包括 S-Lang 注释器 (http://www.s-lang.org/), 共享软件 Cmm (“C 减减” 或 “去掉困难部分的C”)。参见问题 18.18 和 20.4。

《Software Solutions in C》[Schumacher, ed.]中也有一些分析和评估的代码 (第 12 章, 235 到 255 页)。

18.14 哪里可以找到 C 的 BNF 或 YACC 语法？

ANSI 标准中的语法是最权威的。由 Jim Roskind 写的一个语法在 ftp://ftp.eskimo.com/u/s/scs/roskind_grammar.Z。一个 Jeff Lee 做的, 新鲜出炉的 ANSI C90 语法工作实例可以在 ftp://ftp.uu.net/usenet/net.sources/ansi.c.grammar.Z 得到, 还包含了一个相配的 lexer。FSF 的 GNU C 编译器也含有一个语法, 当然还有 K&R 的附录也有一个。

新闻组 comp.compilers 的档案中含有更多的有关语法的信息, 参见问题 18.3。

参考资料: [K&R1, Sec. A18 pp. 214-219]; [K&R2, Sec. A13 pp. 234-239]; [ISO, Sec. B.2]; [H&S, pp. 423-435 Appendix B]。

18.15 谁有 C 编译器的测试套件？

Plum Hall (以前在 Cardiff, NJ, 现在在 Hawaii) 有一个套件出售; Ronald Guilmette 的 RoadTest™ 编译器测试套件 (更多信息在 ftp://netcom.com/pub/rfg/roadtest/announce.txt); Nullstone 的自动编译器性能分析工具 (http://www.nullstone.com)。 FSF 的 GNU C (gcc) 发布中含有一个许多编译器通常问题的 C 严酷测试。 Kahan 的偏执狂的测试 (ftp://netlib.att.com/netlib/paranoia), 尽其所能的测试 C 实现的浮点能力。

18.16 哪里有一些有用的源代码片段和例子的收集？

Bob Stout 的 "SNIPPETS" 是个很流行的收集 (ftp://ftp.brokersys.com/pub/snippets 或 http://www.brokersys.com/snippets/)。

Lars Wirzenius 的 "publib" 函数库 (ftp://ftp.funet.fi/pub/languages/C/Publib/)。

参考问题 14.11, 18.7, 18.8, 18.10 和 18.18。

18.17 我需要执行多精度算术的代码。

一些流行的软件包是: "quad", 函数在 BSD Unix 系统的 libc 中 (ftp.uu.net, /systems/unix/bsd-sources/.../src/lib/libc/quad/*); GNU MP 函数库 "libmp"; MIRACL 软件包 (http://indigo.ie/~mscott/); David Bell 和 Landon Curt Noll 写的"'calc" 程序; 以及老 Unix 的 libmp.a。参见问题 14.11 和 18.18。

参考资料: [Schumacher, ed., Sec.17 pp. 343-454]。

18.18 在哪里和怎样取得这些可自由发布的程序？

随着可利用程序数目, 公共可访问的存档网站数目以及访问的人数的增加, 这个问题回答起来变得即容易又困难。

有几个比较大的公共存档网站, 例如: ftp.uu.net, archive.umich.edu, oak.oakland.edu, sumex-aim.stanford.edu 和 wuarchive.wustl.edu。它们免费提供极多的软件和信息。 FSF GNU 工程的中央发布地址是 ftp.gnu.org。这些知名的网站往往非常繁忙而难以访问, 但也有不少 "镜像" 网站来分担负载。

在互联网上, 传统取得档案文件的方法是通过匿名 ftp。对于不能使用 ftp 的人, 有几个 ftp-by-mail 的服务器可供使用。越来越多的, 万维网 (WWW) 被使用在文件的宣告、索引和传输上。也许还会有新的访问方法。

这些是问题中比较容易回答的部分。困难的部分在于详情——本文不能追踪或列表所有的文档网站或各种访问的方法。如果你已经可以访问互联网了, 你可以取得比本文更加及时的活跃网站信息和访问方法。

问题的另一个即难也易的方面是找到哪个网站有你所要的。在这方面有极多的研究, 几乎每天都有可能有新的索引服务出台。其中最早的服务之一

是 “archie”, 当然还有许多高曝光的商业网络索引和搜索服务, 例如 Alta Vista, Excite 和 Yahoo。

如果你可以访问 Usenet, 请查看定期发布在新闻组 comp.sources.unix 和 comp.sources.misc 的邮件, 其中有说明新闻组归档的政策和怎样访问档案。其中两个是: ftp://gatekeeper.dec.com/pub/usenet/comp.sources.unix/, ftp://ftp.uu.net/usenet/comp.sources.unix/。新闻组 comp.archives 包括了多数的有关各种东西的匿名 ftp 网站公告。最后, 通常新闻组 comp.sources.wanted 是个适合询问源代码的地方, 不过在发贴前, 请先查看它的 FAQ “怎样查找资源 (How to find sources)”。

参见问题 14.11, 18.8, 18.10 和 18.16。

第 19 章

系统依赖

19.1 怎样从键盘直接读入字符而不用等 RETURN 键？怎样防止字符输入时的回显？

唉, 在 C 里没有一个标准且可移植的方法。在标准中跟本就没有提及屏幕和键盘的概念, 只有基于字符 “流” 的简单输入输出。

在某个级别, 与键盘的交互输入一般上都是由系统取得一行的输入才提供给需要的程序。这给操作系统提供了一个加入行编辑的机会 (退格、删除、消除等), 使得系统地操作具一致性, 而不用每一个程序自己建立。当用户对输入满意, 并键入 RETURN (或等价的键)后, 输入行才被提供给需要的程序。即使程序中用了读入单个字符的函数 (例如 getchar() 等), 第一次调用就会等到完成了一整行的输入才会返回。这时, 可能有许多字符提供给了程序, 以后的许多调用 (象 getchar() 的函数) 都会马上返回。

当程序想在一个字符输入时马上读入, 所用的方式途径就采决于行处理在输入流中的位置, 以及如何使之失效。在一些系统下 (例如 MS-DOS, VMS 的某些模态), 程序可以使用一套不同或修改过的操作系统函数来扰过行输入模态。在另外一些系统下 (例如 Unix, VMS 的另一些模态), 操作系统中负责串行输入的部分 (通常称为 “终端驱动”) 必须设置为行输入关闭的模态, 这样, 所有以后调用的常用输入函数 (例如 read(), getchar() 等) 就会立即返回输入的字符。最后, 少数的系统 (特别是那些老旧的批处理大型主机) 使用外围处理器进行输入, 只有行处理模式。

因此, 当你需要用到单字符输入时 (关闭键盘回显也是类似的问题), 你需要用一个针对所用系统的特定方法, 假如系统提供的话。新闻组 comp.lang.c 讨论的问题基本上都是 C 语言中有明确支持的, 一般上你会从针对个别系统的新闻组以及相对应的常用问题集中得到更好的解答, 例如 comp.unix.questions 或 comp.os.msdos.programmer。另外要注意, 有些解答即使是对相似系统的变种也不尽相同, 例如 Unix 的不同变种。同时也要记住, 当回答一些针对特定系统的问题时, 你的答案在你的系统上可以工作并不代表可以在所有人的系统上都工作。

然而, 这类问题被经常的问起, 这里提供一个对于通常情况的简略回答。

某些版本的 curses 函数库包含了 cbreak(), noecho() 和 getch() 函数, 这些函数可以做到你所需的。如果你只是想要读入一个简短的口令而不想回显的话, 可以试试 getpass()。在 Unix 系统下, 可以用 ioctl() 来控制终端驱动的模式, “传

统”系统下有 CBREAK 和 RAW 模式, System V 或 POSIX 系统下有 ICANON, c_cc[VMIN] 和 c_cc[VTIME] 模式, 而 ECHO 模式在所有系统中都有。必要时, 用函数 system() 和 stty 命令。更多的信息可以查看所用的系统, 传统系统下, 查看 <sgtty.h> 和 tty(4), System V 下, 查看 <termio.h> 和 termio(4), POSIX 下, 查看 <termios.h> 和 termios(4)。在 MS-DOS 系统下, 用函数 getch() 或 getche(), 或者相对应的 BIOS 中断。在 VMS 下, 使用屏幕管理例程 (SMG$), 或 curses 函数库, 或者低层 $QIO 的 IO$_READVBLK 函数, 以及 IO$M_NOECHO 等其它函数。也可以通过设置 VMS 的终端驱动, 在单字符输入或 “通过” 模式间切换。如果是其它操作系统, 你就要靠自己了。

另外需要说明一点, 简单的使用 setbuf() 或 setvbuf() 来设置 sdtin 为无缓冲, 通常并不能切换到单字符输入模式。

如果你在试图写一个可移植的程序, 一个比较好的方法是自己定义三套函数： 1) 设置终端驱动或输入系统进入单字符输入模式, (如果有必要的话), 2) 取得字符, 3) 程序使用结束后的终端驱动复原。理想上, 也许有一天, 这样的一组函数可以成为标准的一部分。本常用问题集的扩充版 (参见问题 20.36) 含有一套适用于几个流行系统的函数。

参见问题 19.2

参考资料: [PCS, Sec. 10 pp. 128-9, Sec. 10.1 pp. 130-1]; [POSIX, Sec. 7]。

19.2 怎样知道有未读的字符, 如果有, 有多少？如果没有字符, 怎样使读入不阻断？

这个问题也是完全和操作系统有关。某些版本的 curses 函数库有 nodelay() 的函数。根据所用系统的不同, 也许你可以使用 “不阻断输入输出 (nonblocking I/O)”, 或者系统函数 select 或 poll, 或者用 ioctl 的 FIONREAD, c_cc[VTIME], kbhit(), rdchk(), open() 或 fcntl() 的参数 O_NDELAY。参见问题 19.1。

19.3 怎样显示一个百分比或 “转动的短棒” 的进展表示器？

这个简单的事情, 你可以做到相当的可移植。输出字符 `'\r'` 通常可以得到一个回车而没有换行, 这样你就可以复写当前行。字符 `'\b'` 代表退格, 通常会使光标左移一格。记住要调用 fflush()。

参考资料: [ISO, Sec. 5.2.2]。

19.4 怎样清屏？怎样输出彩色文本？怎样移动光标到指定位置？

这些功能跟你所用的终端类型 (或显示器) 有关。你需要使用 termcap, terminfo 或 curses 类的函数库, 或者系统提供的特殊函数。在 MS-DOS 系统下, 有两个函数可以使用 clrscr() 和 gotoxy()。

有一个不彻底的可移植的清屏方法: 输出卷纸字符 ('\f'), 可以清除一部分的显示。还有个更加可移植的办法 (尽管很简陋), 输出足够多的换行使当前屏幕清空。最后一个方法: 使用 system() 函数 (参见问题 19.30) 来调用操作系统的清屏指令。

参考资料[PCS, Sec. 5.1.4 pp. 54-60, Sec. 5.1.5 pp. 60-62]。

19.5 怎样读入方向键, 功能键?

terminfo, 某些版本的 termcap, 以及某些版本的 curses 函数库有对这些非 ASCII 键的支持。典型的, 一个特殊键会发送一个多字符序列 (通常以 ESC ['\033'] 字符开头)。分析这个多字符序列比较麻烦。如果你首先调用了 keypad(), curses 会帮你做分析。

在 MS-DOS 下, 如果你在读入键盘输入时, 收到一个值为 0 的字符 (不是字符 '0'), 这就标志着下一个读入的值代表一个特殊键。有关键盘的编码可参见任何 DOS 的编程指南。简单的说明: 上、下、左、右键的编码是 72, 80, 75, 77, 功能键从 59 到 68。

参考资料: [PCS, Sec. 5.1.4 pp. 56-7]。

19.6 怎样读入鼠标输入?

请查阅你的系统文档, 或者在特定系统的新闻组寻问, 请先查看其组的 FAQ。鼠标的处理在 X Windown 系统, MD-DOS, Macintosh 下是完全不同的, 也许每个系统都不一样。

参考资料[PCS, Sec. 5.5 pp. 78-80]。

19.7 怎样做串口 ("comm") 的输入输出?

这也是跟所用系统有关的。在 Unix 下, 你通常打开、读入、写出在 /dev 下的一个设备文件, 使用终端驱动提供的工具来调整设备的特性。(参见问题 19.1 和 19.2)。在 MS-DOS 下, 你可一使用预定义的 stdaux 流, 或特殊文件 COM1, 或基础的 BIOS 中断, 又或者你需要更好的性能, 使用任何一个中断驱动的串口输入输出包。许多网友推荐 Joe Campbell 的书《C 程序员串口通讯指南》("C Programmer's Guide to Serial Communications")。

19.8 怎样直接输出到打印机?

Unix 系统下, 使用 popen() (参见问题 19.31) 来把输出写到 lp 或 lpr 中, 或者打开特殊文件 /dev/lp。 MS-DOS 下, 写向预定义的 stdprn 流 (非标准), 或者打开特殊文件 PRN 或 LPT1。在某些情况下, 另一个方法 (也许是唯一的方法) 是用窗口管理者的屏幕截图功能, 然后打印得到的图形。

参考资料: [PCS, Sec. 5.3 pp. 72-74]。

19.9 怎样发送控制终端或其它设备的逃逸指令序列？

如果你能够找到发送字符到设备的方法 (参见问题 19.8), 发送逃逸指令序列是件很容易的事。在 ASCII 代码中, 逃逸字符的代码是 033 (十进制 27), 以下代码就可以发送逃逸序列 ESC[J:

```
fprintf(ofd, "\033[J");
```

19.10 怎样直接访问输入输出板？

通常, 有两种办法: 使用特定的系统函数, 例如 inport 和 outport (如果设备由输入输出接口访问), 或者使用人为的指针变量来访问内存映射的输入输出 (memory-mapped I/O) 设备。参见问题 19.29。

19.11 怎样做图形？

从前, Unix 下有一套相当不错且小巧的设备独立的绘制函数 (plot(3) 和 plot(5))。由 Robert Maier 写的 GNU libplot 函数库保持了同样的精神, 支持许多现代的绘制设备 (http://www.gnu.org/software/plotutils/plotutils.html)。

OpenGL 是一个现代的平台独立的制图函数库, 它也支持三维制图和动画。其它有关的制图标准有 GKS 和 PHIGS。

如果你在 MS-DOS 下编程, 你大概需要用到符合 VESA 或 BGI 标准的函数库。

如果你要和某一个特定的制图仪打交道, 通常发送适当的逃逸序列就可以绘图了, 参见问题 19.9. 销售商可能会提供一个可以从 C 调用的函数库, 或者你也许可以在网上找到。

如果你需要在某个特定的视窗环境下编程 (Macintosh, X Window, Microsoft Windows), 你使用其提供的工具; 参阅相关的文档、新闻组或 FAQ。

参考资料: [PCS, Sec. 5.4 pp. 75-77]。

19.12 怎样显示 GIF 和 JPEG 图象？

这跟你用的显示环境有关, 有可能环境已经提供了这些函数。 http://www.ijg.org/files/ 有个可供参考的 JPEG 软件。

19.13 怎样检验一个文件是否存在？

要做到可靠而可移植的检验出乎意料的困难。如果从你检验到你打开文件前, 这个文件被 (别的进程) 生成或删除了, 所做的任何检验都会失效。

三个可能用作检验的函数是 stat(), access() 和 fopen()。当使用 fopen() 作近似检验时, 用只读打开, 然后马上关闭, 但是失败并不代表不存在。这里, 只有

fopen() 据有广泛的可移植性, 如果系统提供 access, 而程序用了 Unix 的 UID 设置特性, 要特别小心使用。

不要去预测像打开文件这类操作是否成功, 通常直接尝试再查验返回值会更好, 如果失败了再申诉。当然, 如果你要避免复写已存在的文件, 这个方法并不适用, 除非打开文件有象 O_EXCL 的参数, 那就可以做到你所要的效果。

参考资料： [PCS, Sec. 12 pp. 189,213]; [POSIX, ec. 5.3.1, Sec. 5.6.2, Sec. 5.6.3.]。

19.14 怎样在读入文件前, 知道文件大小？

如果文件大小指的是你从 C 程序中可以读进的字符数量, 要得到这个精确的数字可能困难或不可能。

Unix 系统函数 stat() 会给出准确的答案。有些系统提供了类似 Unix 的 stat() 函数, 但只返回一个近似值。你可以调用 fseek() 搜索到文件尾, 再调用 ftell(), 或者调用 fstat(), 然而这些方法都有同样的问题: fstat() 不可移植, 通常返回和 stat() 一样的值; ftell() 并不保证可以返回字符计数, 除非是用于二进制文件, 但是, 严格来讲, 二进制文件并不一定支持 fseek 搜索到 SEEK_END。某些系统提供 filesize() 或 filelength() 的函数, 但是它们明显的不可移植。

你是否确定你必须预先知道文件大小？作为一个 C 程序最准确的方法就是打开文件并读入, 也许可以重新整理代码, 使其边读入边计算文件大小。

参考资料: [ISO, Sec. 7.9.9.4]; [H&S, Sec. 15.5.1]; [PCS, Sec. 12 p. 213]; [POSIX, Sec. 5.6.2]。

19.15 怎样得到文件的修改日期和时间？

Unix 和 POSIX 函数是 stat(), 某些其它系统也提供。参见问题 19.14。

19.16 怎样缩短一个文件而不用清除或重写？

BSD 系统提供函数 ftruncate(), 某些其它系统提供 chsize(), 还有少数系统提供用于 fcntl 的参数 F_FREESP。 MS-DOS 下, 某些时候你可以用 write(fd, "", 0)。然而, 没有一个可移植的方法, 也没有办法删除在文件开头的数据块。参见问题 19.17。

19.17 怎样在文件中插入或删除一行 (或记录)？

如果你不能重写文件, 你大概做不了。通常的做法就是重写文件。也许你可以用简单的标志记录删除来取代真的删除, 这样可以避免重写。另外一个可能性, 是使用数据库来取代平坦文件。参见问题 12.26 和 19.16。

19.18 怎样从一个打开的流或文件描述符得到文件名？

通常情况下, 这是没办法解决的。在 Unix 系统下, 理论上需要搜索整个磁盘, 也许还牵扯到特殊许可的问题, 如果文件描述符是连接到管道 (pipe) 或者指向一个已被删除的文件 (如果文件有多个连接, 也许会返回误导的信息), 那搜索也会失败。最好的方法就是你在打开文件时, 自己记住 (或许用一个 fopen() 的包裹函数)。

19.19 怎样删除一个文件？

标准 C 库函数是 remove()。这是本章里少有的几个与系统无关的问题。在一些 ANSI 之前的老 Unix 系统, remove() 可能不存在, 那么你可以试试 unlink()。

参考资料[K&R2, Sec. B1.1 p. 242; ISO Sec. 7.9.4.1]; [H&S, Sec. 15.15 p. 382]; [PCS, Sec. 12 pp. 208,220-221]; [POSIX, Sec. 5.5.1, Sec. 8.2.4]。

19.20 怎样复制一个文件？

可以用函数 system() 调用你所用操作系统的文件复制工具, 参见问题 19.30。或者打开源文件和目标文件 (用 fopen() 或一些低层的打开文件的系统函数), 读入字符或数据段, 再写出到目标文件中。

参考资料: [K&R2, Sec. 1, Sec. 7]。

19.21 为什么用了详尽的路径还不能打开文件？ fopen("c:\ newdir \file.dat", "r") 返回错误。

你实际请求的文件名内含有字符 \n 和 \f, 可能并不存在, 也不是你希望的。

在字符常量和字符串中, 反斜杠 \ 是逃逸字符, 它赋予后面紧跟的字符特殊意义。为了正确的把反斜杠传递给 fopen() (或其它函数), 必须成双的使用, 这样第一个反斜杠引述了第二个:

```
fopen("c:\\newdir\\file.dat", "r")
```

另一个选择, 在 MS-DOS 下, 正斜杠也被接受为路径分隔符, 所以也可以这样用:

```
fopen("c:/newdir/file.dat", "r")
```

注意, 顺便提一下, 用于预处理 #include 指令的头文件名不是字符串文字, 所以不必担心反斜杠的问题。

19.22 fopen() 不让我打开文件: "$HOME/.profile" 和 "~/ .myrcfile"。

至少在 Unix 系统下, 像 $HOME 这样的环境变量和家目录的表示符 ~ 是由 shell 来展开的。不存在一个调用 fopen() 时的自动扩展机制。

19.23 怎样制止 MS-DOS 下令人担忧的 “Abort, Retry, Ignore?” 信息？

除了其它的事, 你需要截取 DOS 的重要错误中断, 中断 24H。详情请参阅 comp.os.msdos.programmer 的 FAQ。

19.24 遇到 “Too many open files (打开文件太多)” 的错误, 怎样增加同时打开文件的允许数目？

通常有至少两个资源限制了同时打开文件的数目: 操作系统可用的低层 “文件说明符” 或 “文件句柄” 的数目; 和标准 stdio 函数库可用的 FILE 结构数目。两个条件必须符合。在 MS-DOS 下, 可以通过设置 CONFIG.SYS, 可以控制系统文件 handle 的数目。一些编译器附有增加 stdio 的 FILE 结构数目的指令 (也许是一两个源文件)。

19.25 怎样在 C 中读入目录？

试试能否使用 opendir() 和 readdir() 函数, 它们是 POSIX 标准的一部分, 大多数 Unix 变体都支持。MS-DOS, VMS 和其它系统下也有这些函数的实现。MS-DOS 还有 FINDFIRST 和 FINDNEXT 函数, 它们做的事基本一样, MS Windows 有 FindFirstFile 和 FindNextFile。readdir() 只返回文件名, 如果你需要该文件更多的信息, 试用 stat()。如果想匹配文件名和通配符式样, 参见问题 13.5。

参考资料: cite[Sec. 8.6 pp. 179-184]kr2; [PCS, Sec. 13 pp. 230-1]; [POSIX, Sec. 5.1]; [Schumacher, ed., Sec. 8]。

19.26 怎样找出系统还有多少内存可用？

你所用的系统可能会提供一个例程返回你所需的信息, 但是这跟系统相当有关。

19.27 怎样分配大于 64K 的数组或结构？

一台合理的电脑应该可以让你透明地访问所有的有效内存。如果, 你很不幸, 你可能需要重新考虑程序使用内存的方式, 或者用各种针对系统的技巧。

64K 仍然是一块相当大的内存。不管你的电脑有多少内存, 分配这么一大段连续的内存是个不小的要求。标准 C 不保证一个单独的对象可以大于 32K, 或者 C99 的 64K。通常, 设计数据结构时的一个好的思想, 是使它不要求所有的内存都连续。对于动态分配的多维数组, 你可以使用指针的指针, 在问题 6.13 中有举例说明。你可以用链接或结构指针数组来代替一个大的结构数组。

如果你使用的是 PC 兼容机 (基于 8086) 系统, 遇到了 64K 或 640K的限制, 可以考虑使用 “huge” (巨大) 内存模型, 或者扩展或延伸内存, 或 malloc 的变体函数 halloc() 和 farmalloc(), 或者用一个 32 比特的 “平直” 编译器 (例如 djgpp, 参见问题 18.3), 或某种 DOS 扩充器, 或换一个操作系统。

参考资料: [ISO, Sec. 5.2.4.1]; [C9X, Sec. 5.2.4.1]。

19.28 错误信息 “DGROUP data allocation exceeds 64K (DGROUP 数据分配内存超过 64K)” 说明什么？我应该怎么做？我以为使用了大内存模型, 那我就可以使用多于 64K 的数据！

即使使用了大内存模型, MS-DOS 的编译器明显地把某些数据 (字符串, 已初始化的全局或静态变量) 都放在了一个缺省的数据段, 而这个数据段溢出了。要么减少全局数据, 或者已经限制在一个合理的范围 (而引起问题的是由于字符串的数目), 你可以告诉编译器对这么大的数据不要使用缺省数据段。某些编译器只把 “小” 数据放在缺省数据段, 也提供了设置 “小” 的临界值的方法, 例如, Microsoft 的编译器可以用参数 /Gt。

19.29 怎样访问位于某的特定地址的内存 (内存映射的设备或图显内存)？

设置一个适当类型的指针去正确的值, 使用明示的类型重制, 以保证编译器知道这个不可移植转换是你的意图:

```
unsigned int *magicloc = (unsigned int *)0x12345678;
```

那么, *magiloc 就指向你所要的地址。如果地址是个内存映射设备的寄存器, 你大概需要使用限定词 volatile。MS-DOS 下, 在和段、偏移量打交道时, 你会发现像 MK_FP 这类宏非常好用。

参考资料: [K&R1, Sec. A14.4 p. 210]; [K&R2, Sec. A6.6 p. 199]; [ISO, Sec. 6.3.4;]; [Rationale, Sec. 3.3.4]; [H&S, Sec. 6.2.7 pp. 171-2]。

19.30 怎样在一个 C 程序中调用另一个程序 (独立可执行的程序, 或系统命令)？

使用库函数 system(), 它的功能正是你所要的。注意, 系统返回的值最多是命令的退出状态值 (但这并不是一定的), 通常和命令的输出无关。还要注意, system() 只接受一个单独的字符串参数来表述调用程序。如果你要建立复杂的命令行, 可以使用 sprintf()。

跟据你使用的系统, 也许你还可以使用系统函数, 例如 exec 或 spawn (或 execl, execv, spawnl, spawnv 等)。

参见问题 19.31。

参考资料: [K&R1, Sec. 7.9 p. 157]; [K&R2, Sec. 7.8.4 p. 167, Sec. B6 p. 253]; [ISO, Sec. 7.10.4.5]; [H&S, Sec. 19.2 p. 407]; [PCS, Sec. 11 p. 179]。

19.31 怎样调用另一个程序或命令, 同时收集它的输出?

Unix 和其它一些系统提供了 popen() 函数, 它在联通运行命令的进程管道设置了 stdio 流, 所以输出可以被读取 (或提供输入)。记住, 结束使用后, 要调用函数 pclose()。

如果你不能使用 popen(), 你应该可以调用 system(), 并输出到一个你可以打开读取的文件。

如果你使用 Unix, 觉得 popen() 不够用, 你可以学习用 pipe(), dup(), fork() 和 exec()。

顺便提一下, freopen() 可能并不工作。

参考资料: [PCS, Sec. 11 p. 169]。

19.32 怎样才能发现程序自己的执行文件的全路径?

arg[0] 也许含有全部或部分路径, 或者什么也没有。如果 arg[0] 中的路径不全, 你也许可以重复命令语言注释器的路径搜索逻辑。但是, 没有保证的解决方法。

参考资料: [K&R1, Sec. 5.11 p. 111]; [K&R2, Sec. 5.10 p. 115]; [ISO, Sec. 5.1.2.2.1]; [H&S, Sec. 20.1 p. 416.]。

19.33 怎样找出和执行文件在同一目录的配置文件?

这非常困难。参见问题 19.32。即使你可以找出一个可行的方法, 你可能会考虑通过环境变量或别的方法使程序的辅助 (函数库) 目录可配置。如果程序会被数个人使用, 例如在多用户系统中, 那允许配置文件的变量存放变得特别重要。

19.34 一个进程如何改变它的调用者的环境变量?

这有可能完全做不到。不同的系统使用不同的方法来实现像 Unix 系统的全局名字/值功能。环境是否可以被运行的进程有效的改变, 以及如果可以, 又怎样去做, 这些都依赖于系统。

在 Unix 下, 一个进程可以改变自己的环境 (某些系统为此提供了 setenv() 或 putenv() 函数), 被改变的环境通常会被传给子进程, 但是这些改变不会传递到父进程。在 MS-DOS 下, 总环境是可以操作的, 但是这需要晦涩难解的技巧。参见 MS-DOS 的 FAQ。

19.35 怎样读入一个对象文件并跳跃到其中的地址？

你需要一个动态的连接程序或载入程序。也许可以 malloc 一段内存, 再读入对象文件, 但是你需要知道极多的有关对象文件格式、地址变换等知识。在 BSD Unix 下, 你可以使用 system() 和 ld -A 来实现连接。许多 SunOS 和 System V 的版本有 -ldl 函数库, 允许动态载入对象文件。在 VMS 下, 使用 LIB$FIND_IMAGE_SYMBOL。 GNU 有个 dld 的包可以用。参见问题 15.12。

19.36 怎样实现精度小于秒的延时或记录用户回应的时间？

很不幸, 这没有可移植解决方法。下面是一些你可以在你的系统中寻找的函数: clock(), delay(), ftime(), getimeofday(), msleep(), nap(), napms(), nanaosleep(), setitimer(), sleep(), Sleep(), times() 和 usleep。至少在 Unix 系统下, 函数 wait() 不是你想要的。函数 select() 和 poll() (如果存在) 可以用来实现简单的延时。在 MS-DOS 下, 可以重新对系统计时器和计时器中断编程。

这些函数中, 只有 clock() 在 ANSI 标准中。两次调用 clock() 之间的差分就是执行所用的时间, 如果 CLOCKS_PER_SEC 的值大于 1, 你可以得到精确度小于秒的计时。但是, clock() 返回的是执行程序使用的处理器的时间, 在多任务系统下, 有可能和真实的时间相差很多。

如果你需要实现一个延时, 而你只有报告时间的函数可用, 你可以实现一个繁忙等待。但是这只是在单用户, 单任务系统下可选, 因为这个方法对于其它进程极不友好。在多任务系统下, 确保你调用函数, 让你的进程在这段时间进入休眠状态。可用函数 sleep(), select() 或 poll() 和 alarm() 或 setitimer()实现。

对于非常短暂的延时, 使用一个空循环颇据有诱惑力:

```
long int i;
for (i = 0; i < 1000000; ++i)
    ;
```

但是请尽量抵制这个诱惑！因为, 经过你仔细计算的延时循环可能在下个月因为更快的处理器出现而不能正常工作。更糟糕的是, 一个聪明的编译器可能注意到这个循环什么也没做, 而把它完全优化掉。

参考资料: [H&S, Sec. 18.1 pp. 398-9]; [PCS, Sec. 12 pp. 197-8,215-6]; [POSIX, Sec. 4.5.2]。

19.37 怎样抓获或忽略像 control-C 这样的键盘中断？

基本步骤是调用 signal()：

```
#include <signal.h>
singal(SIGINT, SIG_IGN);
```

就可以忽略中断信号, 或者:

```
extern void func(int);
signal(SIGINT, func);
```

使程序在收到中断信号时, 调用函数 func()。

在多任务系统下 (例如 Unix), 最好使用更加深入的技巧:

```
extern void func(int);
if(signal(SIGINT, SIG_IGN) != SIG_IGN)
    signal(SIGINT, func);
```

这个测试和额外的调用可以保证前台的键盘中断不会因疏忽而中断了在后台运行的进程, 在所有的系统中都用这种调用 signal 的方法并不会带来负作用。

在某些系统中, 键盘中断处理也是终端输入系统模式的功能, 参见问题 19.1。在某些系统中, 程序只有在读入输入时, 才查看键盘中断, 因此键盘中断处理就依赖于调用的输入例程 (以及输入例程是否有效)。在 MS-DOS 下, 可以使用 setcbrk() 或 ctrlbrk()。

参考资料: [ISO, Secs. 7.7,7.7.1]; [H&S, Sec. 19.6 pp. 411-3]; [PCS, Sec. 12 pp. 210-2]; [POSIX, Secs. 3.3.1,3.3.4]。

19.38 怎样很好地处理浮点异常?

在许多系统中, 你可以定义一个 matherr() 的函数, 当出现某些浮点错误时 (例如 <math> 中的数学例程), 它就会被调用。你也可以使用 signal() 函数 (参见问题 19.37) 截取 SIGFPE 信号。参见问题 14.9。

参考资料: [Rationale, Sec. 4.5.1]。

19.39 怎样使用 socket? 网络化? 写客户/服务器程序?

所有这些问题都超出了本文的范围, 它们跟你所有的网络设备比 C 更有关系。有一些这方面主题的好书: Douglas Comer 撰写的三卷 《Internetworking with TCP/IP》 和 W. R. Stevens撰写的 《UNIX Network Programming》。网上也有相当多这方面的信息, 有 "Unix Socket FAQ" (http://www.developerweb.net/sock-faq/), "Beej's Guide to Network Programming" (http://www.ecst.csuchico.edu/~beej/guide/net/)。

一个提示: 针对你所用的操作系统, 你也许需要明示的请求 -lsocket 或 -lnsl 函数库; 参见问题 13.18。

19.40 怎样调用 BIOS 函数? 写 ISR? 创建 TSR?

这些都是针对某一特定系统的问题 (最可能的是运行 MS-DOS 的 PC 兼容机)。在针对系统的新闻组 comp.os.msdos.programmer 或它的 FAQ 里你会取得更好的信息; 另一个很好的资源是 Ralf Brown 的中断列表。

19.41 编译程序, 编译器出示 "union REGS" 未定义错误信息, 连接器出示 "int86()" 的未定义错误信息。

这些都跟 MS-DOS 的中断编程有关。它们在其它系统中不存在。

19.42 什么是 “near” 和 “far” 指针？

如今, 它们差不多被废弃了; 它们肯定于特定系统有关。如果你真的要知道, 参阅针对 DOS 或 Windows 的编程参考资料。

19.43 我不能使用这些非标准、依赖系统的函数, 程序需要兼容 ANSI!

你很不走运。要么你误解了要求, 要么这不可能做到。 ANSI/ISO C 标准没有定义做这些事的方法; 它是个语言的标准, 不是操作系统的标准。国际标准 POSIX (IEEE 1003.1, ISO/IEC 9945-1) 倒是定义了许多这方面的方法, 而许多系统 (不只是 Unix) 都有兼容 POSIX 的编程接口。

可以做到, 也是可取的做法是使程序的大部分兼容 ANSI, 将依赖系统的功能集中到少数的例程和文件中。这些例程或文件可以大量使用 #ifdef 或针对每一个移植的系统重写。

第 20 章

杂项

20.1 怎样从一个函数返回多个值？

可以传入多个指针指向不同的地址, 函数可以填入, 或者使函数返回一个包含了需要值的结构, 又或者理论上你可以使用全局变量。参见问题 4.4 和 7.6。

20.2 怎样访问命令行参数？

当 main() 被调用时, 它们储存在数组 argv 中。参见问题 8.2, 13.5 和 19.25。

参考资料: [K&R1, Sec. 5.11 pp. 110-114]; [K&R2, Sec. 5.10 pp. 114-118]; [ISO, Sec. 5.1.2.2.1]; [H&S, Sec. 20.1 p. 416]; [PCS, Sec. 5.6 pp. 81-2, Sec. 11 p. 159, pp. 339-40 Appendix F]; [Schumacher, ed., Sec. 4 pp. 75-85]。

20.3 怎样写数据文件, 使之可以在不同字大小、字节顺序或浮点格式的机器上读入？

最可移植的方法是是用文本文件 (通常是 ASCII), 用 fprintf() 写入, 用 fscanf() 读入, 或类似的函数。同理, 这也适用于网络协议。不必太相信那些说文本文件太大或读写太慢的论点。大多数现实情况下, 操作的效率是可接受的, 而可以在不同机器间交换和用标准工具就可以对其进行操作是个巨大的优势。

如果你必须使用二进制文件, 你可以通过使用某些标准格式来提高可移植性, 还可以利用已经写好的 I/O 函数库。这些格式包括： Sun 的 XDR (RFC 1014)、OSI 的 ASN.1 (在 CCITT X.409 和 ISO 8825 “Basic Encoding Rules” 中都有引用)、 CDF、 netCDF 或 HDF。参见问题 2.10 和 12.30。

参考资料: [PCS, Sec. 6 pp. 86, 88]。

20.4 怎样调用一个由 char * 指针指向函数名的函数？

最直接的方法就是维护一个名字和函数指针的列表：

```
int one_func(), two_func();
int red_func(), blue_func();

struct { char *name; int (*funcptr)(); } symtab[] = {
    "one_func", one_func,
```

```
    "two_func", two_func,
    "red_func", red_func,
    "blue_func",blue_func,
};
```

然后搜索函数名, 就可以调用关联的函数指针。参见问题 2.13, 18.13 和 19.35。

参考资料: [PCS, Sec. 11 p. 168]。

20.5 怎样实现比特数组或集合？

使用 int 或 char 数组, 再加上访问所需比特的几个宏。这里有一些简单的宏定义, 用于 char 数组:

```
#include <limits.h>     /* for CHAR_BIT */

#define BITMASK(b) (1 << ((b) % CHAR_BIT))
#define BITSLOT(b) ((b) / CHAR_BIT)
#define BITSET(a, b) ((a)[BITSLOT(b)] |= BITMASK(b))
#define BITTEST(a, b) ((a)[BITSLOT(b)] & BITMASK(b))
```

如果你没有 <limits.h>, 可以定义 CHAR_BIT 为 8。

参考资料: [H&S, Sec. 7.6.7 pp. 211-216]。

20.6 怎样判断机器的字节顺序是高字节在前还是低字节在前？

有个使用指针的方法:

```
int x = 1;
if(*(char *)&x == 1)
    printf("little-endian\n");
else
    printf("big-endian\n");
```

另外一个可能是用联合。

参见问题 10.15 和 20.7。

参考资料: [H&S, Sec. 6.1.2 pp. 163-4]。

20.7 怎样掉换字节？

V7 Unix 有一个 swap() 的函数, 但似乎被遗忘了。

使用明示的字节调换代码有个问题, 你必须决定是否要调用; 参见问题 20.6。更好的方法是使用函数 (例如 BSD 系统中网络函数 ntohs() 等), 函数会进行已知字符顺序和机器顺序 (未知) 之间的转换, 对于已经和机器匹配的字符顺序, 函数不作任何转换。

如果你必须自己写字符转换的代码, 两个明显的方法就是使用指针或联合, 就象问题 20.6 一样。

参考资料: [PCS, Sec. 11 p. 179]。

20.8 怎样转换整数到二进制或十六进制？

确定你真的知道你在问什么。整数是以二进制存储的, 虽然对于大多数情况下, 把它们当成是八进制、十进制或十六进制并没有错, 只要方便就好。数字表达的进制只有在读入或写出到外部世界时才起作用。

在源程序中, 非十进制的数字由在前的 0 或 0x 表示 (分别位八进制和十六进制)。在进行 I/O 操作时, 数字格式的进制在 printf 和 scanf 这类函数里, 由格式符决定 (%d, %o 和 %x 等); 在 strtol() 和 strtoul() 中, 则由他们的第三个参数决定。如果你想要输出任意进制的数字字符串, 你需要自己提供相关的函数 (基本上是 strtol 的反函数)。在进行二进制 I/O 时, 进制就不相干了。

更多的有关二进制的 I/O, 参见问题 2.9。还有问题 8.4 和 13.1。

参考资料: [ISO, Secs. 7.10.1.5,7.10.1.6]。

20.9 我可以使用二进制常数吗？有 printf() 的二进制的格式符吗？

两个都不行。你可以用 strtol() 把二进制的字符串转换成整数。参见问题 20.8。

20.10 什么是计算整数中比特为 1 的个数的最有效的方法？

许多象这样的比特问题可以使用查找表格来提供效率和速度 (参见问题 20.11)。

20.11 什么是提高程序效率的最好方法？

选择好的算法, 小心地实现, 同时确定程序不做额外的事。例如, 即使世界上最优化的字符复制循环也比不上不用复制。

当担心效率时, 要保持几样事情在视野中, 这很重要。首先, 虽然效率是个非常流行的话题, 它并不总是象人们想的那样重要。大多数程序的大多数代码并不是时间紧要的。当代码不是时间紧要时, 通常把代码写得清楚和可移植比达到最大效率更重要。记住, 电脑运行得非常非常快, 那些看起来 “低效率” 的代码, 也许可以编译得比较有效率, 而运行起来也没有明显的延时。

试图预知程序的 “热点” 是个非常困难的事。当要关心效率时, 使用 profiling 软件来确定程序中需要得到关注的地方。通常, 实际计算时间都被外围任务占用了 (例如 I/O 或内存的分配), 可以通过使用缓冲和超高速缓存来提高速度。

即使对于时间紧要的代码, 最无效的优化技巧是忙乱于代码细节。许多常被建议的 “有效的代码技巧”, 即使是很简单的编译器也会自动完成 (例如, 用移位运算符代替二的幂次方乘)。非常多的手动优化有可能是代码变得笨重而使效率反而低下了, 同时几乎不可移植。例如, 也许可以在某台机器上提了速, 但在另一台机器上去变慢了。任何情况下, 修整代码通常最多得到线性信能提高; 更好的算法可以得到更好的回报。

有关效率的更多讨论, 以及当效率很重要时, 如何提高效率的建议, 可以从以下书中得到： Kernighan 和 Plauger 的 《The Elements of Programming Style》[K&P] 中的第七章, 和 Jon Bentley 的 《Writing Efficient Programs》[Bentley]。

20.12 指针真得比数组快吗？函数调用会拖慢程序多少？ ++i 比 i = i +1 快吗？

这些问题的精确回答, 跟你所用的处理器和编译器有关。如果你必须知道, 你就得小心的给程序计时。通常, 差别是很小的, 小到要经过千万次迭代才能看到不同。如果可能, 查看编译器的汇编输出, 看看这两种方法是否被编译的一样。

一般的机器, 通常遍历大的数组时, 用指针比用数组要快, 但是某些处理器就相反。

函数调用, 虽然明显比内联代码要慢, 但是它对程序的模块化和代码清晰度的贡献, 很少有好的理由来避免它。

在修整象 i = i + 1 这样的代码前, 记住你是在跟编译器打交道, 而不是键击编程的计算器。对于 ++i, i += 1 和 i = i + 1, 任何好的编译器都会生成完全一样的代码。使用任何一种形式只跟风格有关, 于效率无关。参见问题 3.10。

20.13 人们说编译器优化的很好, 我们不在需要为速度而写汇编了, 但我的编译器连用移位代替 i/=2 都做不到。

i 是有符号还是无符号？如果是有符号, 移位并不等价 (提示: 想想如果 i 是个负的奇数), 所以编译器没有使用是对的。

20.14 怎样不用临时变量而交换两个值？

一个标准而古老的汇编程序员的技巧是：

```
a ^= b;
b ^= a;
a ^= b;
```

但是这样的代码在现代高级程序设计语言中没什么用处。临时变量基本上是自由使用的, 一般上的三个赋值是:

```
int t = a;
a = b;
b = t;
```

这不只对读者更清晰, 更有可能被编译器辨别出来而变成最有效的代码 (例如有可能使用 EXCH 指令)。后面的代码明显的可以用于指针和浮点值, 而不象 XOR 技巧只能用于整型。参见问题 3.4 和 10.2。

20.15 是否有根据字符串做切换的方法？

没有直接的方法。有些时候, 更适合使用一个单独的函数以映射字符串到整数代码, 然后根据整数代码做切换。否则, 你当然也可以使用 strcmp() 和传统的 if/else 链。参见问题 10.11, 20.16 和 20.26。

参考资料: [K&R1, Sec. 3.4 p. 55]; [K&R2, Sec. 3.4 p. 58]; [ISO, Sec. 6.6.4.2]; [H&S, Sec. 8.7 p. 248]。

20.16 是否有使用非常量 case 标志的方法 (例如范围或任意的表达式)？

没有。最初设计 switch 语句就是为编译器能简单的做转换, 所以 case 标志被限制在单个、整形、常量的表达式。如果你不介意详述的列出所有的情况, 你可以把几个 case 标志连到同个语句, 这样你可以覆盖一个小的范围。

如果你想要根据任意范围或非常量表达式进行选择, 你只能用 if/else 链。

参见问题 20.15。

参考资料: [K&R1, Sec. 3.4 p. 55]; [K&R2, Sec. 3.4 p. 58]; [ISO, Sec. 6.6.4.2]; [Rationale, Sec. 3.6.4.2]; [H&S, Sec. 8.7 p. 248]。

20.17 return 语句外层的括号是否真的可选择？

是的。

很久以前, 在 C 刚起步的时候, 它们是必须的, 刚好那时有足够的人学习了 C, 他们写的代码如今还在使用, 所以还是需要括号的想法被广泛的流传。

碰巧的是, 在某些起况下, sizeof 运算符的括号也是可选择的。

参考资料: [K&R1, Sec. A18.3 p. 218]; [ISO, Sec. 6.3.3, Sec. 6.6.6]; [H&S, Sec. 8.9 p. 254.]。

20.18 为什么 C 注释不能嵌套？怎样注释掉含有注释的代码？引用字符串内的注释是否合法？

C 注释不能嵌套最可能的原因是 PL/I 的注释也不可以, C 是借鉴了它而成的。所以, 通常使用 #ifdef 或 #if 0 来 “注释” 掉大段代码, 其中可能含有注释 (参见问题 11.20)。

字符序列 /* 和 */ 在双引号内的字符串没有特殊含义, 所以不要在其中加入注释, 程序可能想输出它们 (特别是要产生 C 代码作为输出的程序)。

注意 // 在 C99 中才成为合法的注释符。

参考资料: [K&R1, Sec. A2.1 p. 179]; [K&R2, Sec. A2.2 p. 192]; [ISO, Sec. 6.1.9, Annex F; Rationale Sec. 3.1.9]; [H&S, Sec. 2.2 pp. 18-9]; [PCS, Sec. 10 p. 130]。

20.19 C 是个伟大的语言还是别的？哪个其它语言可以写象 a+++++b 这样的代码？

在 C 中, 写成这样也是没有意义的。词汇分析的规则规定, 在一个简单的从左到右扫描中的每个点, 最长的记号被划分, 不管这样出来的记号序列是否有意义。问题中的片段被解释为:

```
a ++ ++ + b
```

语法上是个不合法的表达式。

参考资料: [K&R1, Sec. A2 p. 179]; [K&R2, Sec. A2.1 p. 192]; [ISO, Sec. 6.1]; [H&S, Sec. 2.3 pp. 19-20]。

20.20 为什么 C 没有嵌套函数？

实现嵌套函数不是件简单的事, 它们需要可以正当的访问包含函数的本地变量, 为了简单化, 这个功能是被故意舍弃的。gcc 的扩展功能允许函数嵌套。许多可能使用嵌套函数的地方 (例如 qsort 的比较函数), 一个充分但少许麻烦的解决方法是使用一个定义为静态 (static) 的邻近函数, 如果需要, 可以通过少量静态变量进行通讯。一个干净些的方法是传递一个包含所需内容的结构指针, 虽然 qsort 不支持这种方法。

20.21 assert() 是什么？怎样用它？

这是个定义在 <assert.h> 中的宏, 用来测试断言。一个断言本质上是写下程序员的假设, 如果假设被违反, 那表明有个严重的程序错误。例如, 一个假设只接受非空指针的函数, 可以写:

```
assert(p != NULL);
```

一个失败的断言会中断程序。断言不应该用来捕捉意料中的错误, 例如 malloc() 或 fopen() 的失败。

参考资料: [K&R2, Sec. B6 pp. 253-4]; [ISO, Sec. 7.2]; [H&S, Sec. 19.1 p. 406]。

20.22 怎样从 C 中调用 FORTRAN (C++, BASIC, Pascal, Ada, LISP) 的函数？反之亦然？

这完全依赖于机器以及使用的各个编译器的特别调用顺序, 有可能完全做不到。仔细阅读编译器的文档, 有些时候有个 “混合语言编程指南”, 尽管传递参数以及保证正确的运行启动的技巧通常很晦涩难懂。

对于 FORTRAN, 更多的信息可以从 Glenn Geers 的 FORT.gz 找到, 这个文档可以从匿名 ftp 网站 suphys.physics.su.oz.au 的 src 目录取得。 Burkhard Burow 写的头文件 cfortran.h 简化了许多流行机器上的 C/FORTRAN 接口。可以从匿名 ftp 网站 zebra.desy.de 或 http://www-zeus.desy.de/~burow 取得。

C++ 中, 外部函数说明的 “C” 修改量表明函数应该按 C 的调用约定使用。
参考资料: [H&S, Sec. 4.9.8 pp. 106-7]。

20.23 有什么程序可以做从 Pascal 或 Fortran (或 LISP, Ada, awk, “老” C) 到 C 的转换?

有几个自由发布的程序可以使用:

p2c 由 Dave Gillespie 写的 Pascal 到 C 的转换器, 发布于新闻组 comp.sources.unix 1990 年三月 (第 21 卷); 也可以从 ftp://csvax.cs.caltech.edu/pub/p2c-1.20.tar.Z 取得。

ptoc 另外一个 Pascal 到 C 的转换器, 它是用 Pascal 写的。 (comp.sources.unix, 第 10 卷, 补丁在 第 13 卷)。

f2c Fortran 到 C 的转换器, 由 Bell Labs, Bellcore 和 Carnegie Mellon 的人员共同开发的。可以用以下方法得到更多的 f2c 信息: 发 email 信息 “send index from f2c” 到 netlib@research.att.com 或者 research!netlib。(在匿名 ftp 网站 netlib.att.com 的 netlib/f2c 目录也可取得。)

本 FAQ 的维护者也有一个别的转换器的列表。
参见问题 11.30 和 18.18。

20.24 C++ 是 C 的超集吗? 可以用 C++ 编译器来编译 C 代码吗?

C++ 源自 C, 而且大部分都建立在 C 的基础上, 但是有一些合法的 C 代码在 C++ 中不合法。相反的, ANSI C 继承了 C++ 的几个特性, 包括原型和常量, 所以这两个语言并不是另一个的超集或子集; 而且它们在一些通用构造的定义上也不同。尽管有这些不同, 许多 C 程序在 C++ 环境中编译正确, 许多最新的编译器同时提供 C 和 C++ 的编译模式。但是, 把 C 代码当成 C++ 来编译通常是个坏的注意; 两个语言的差异普遍上足够让你得到不好的结果。参见问题 8.5 和 20.18。

参考资料: [H&S, p. xviii, Sec. 1.1.5 p. 6, Sec. 2.8 pp. 36-7, Sec. 4.9 pp. 104-107]。

20.25 需要用到 “近似” 的 strcmp, 比较两个字符串的近似度, 并不需要完全一样。

Sun Wu 和 Udi Manber 写的文章 “AGREP – A Fast Approximate Pattern-Matching Tool”[AGREP] 中有一些有用的信息, 近似字符串匹配的算法以及有用的参考文献。

另外一个方法牵涉到 “soundex” 算法, 它把发音相近的词映射到同一个代码。它是为发现近似发音的名字而设计的 (作为电话号码目录的帮助), 但是它可以调整用于任意词处理的服务。

参考资料: [Knuth, Sec. 6 pp. 391-2 Volume 3]; [AGREP]。

20.26 什么是散列法？

散列法是把字符串映射到整数的处理, 通常是到一个相对小的范围。一个 “散列函数” 映射一个字符串 (或其它的数据结构) 到一个有界的数字 (散列存贮桶), 这个数字可以更容易的用于数组的索引或者进行反复的比较。明显的, 一个从潜在的有很多组的字符串到小范围整数的映射不是唯一的。任何使用散列的算法都要处理 “冲突” 的可能。有许多散列函数和相关的算法被开发了出来; 一个全面的说明已经超出了本文的范围。

参考资料: [K&R2, Sec. 6.6]; [Knuth, Sec. 6.4 pp. 506-549 Volume 3]; [Sedgewick, Sec. 16 pp. 231-244]。

20.27 由一个日期, 怎样知道是星期几？

用 mktime() 或 localtime() (参见问题 13.11 和 13.12, 如果 tm_hour 的值位 0, 要注意 DST (夏时制) 的调整); 或者 Zeller 的 congruence (参阅 sci.math FAQ); 或者这个由 Tomohiko Sakamoto 提供的优雅的代码:

```
int dayofweek(int y, int m, int d)  /* 0 = Sunday */
{
static int t[] = {0, 3, 2, 5, 0, 3, 5, 1, 4, 6, 2, 4};
y -= m < 3;
return (y + y/4 - y/100 + y/400 + t[m-1] + d) % 7;
}
```

参见问题 13.12 和 20.28。

参考资料: [ISO, Sec. 7.12.2.3]。

20.28 (year%4 == 0) 是否足够判断润年？2000 年是闰年吗？

这个测试并不足够 (2000 年是闰年)。对于当前用的格里高力历法, 完整的表达式为:

```
year % 4 == 0 && (year % 100 != 0 || year % 400 == 0)
```

详情请参阅一本好的天文历法的书或其它参考资料。预防无休止的辩论; 那些主张还有一个 4000 年规则的参考资料是错的。参见问题 13.12。

20.29 一个难题: 怎样写一个输出自己源代码的程序？

要写一个可移植的自我再生的程序是件很困难的事, 部分原因是因为引用和字符集的难度。

这里是个经典的例子 (应该以一行表示的, 虽然第一次执行后它后自我修复):

```
char*s="char*s=%c%s%c;main(){printf(s,34,s,34);}";
main(){printf(s,34,s,34);}
```

这段程序有一些依赖, 忽略了 #include <stdio.h>, 还假设了双引号 " 的值为 34, 和 ASCII 中的值一样。

这里还有一个有 James Hu 发布的改进版:

```
#define q(k)main(){return!puts(#k"\nq("#k")");}
q(#define q(k)main(){return!puts(#k"\nq("#k")");})
```

20.30 什么是 “达夫设备” (Duff's Device)?

这是个很棒的迂回循环展开法, 由 Tom Duff 在 Lucasfilm 时所设计。它的 “传统” 形态, 是用来复制多个字节:

```
register n = (count + 7) / 8;   /* count > 0 assumed */
switch (count % 8)
{
case 0:    do { *to = *from++;
case 7:      *to = *from++;
case 6:      *to = *from++;
case 5:      *to = *from++;
case 4:      *to = *from++;
case 3:      *to = *from++;
case 2:      *to = *from++;
case 1:      *to = *from++;
       } while (--n > 0);
}
```

这里 count 个字节从 from 指向的数组复制到 to 指向的内存地址 (这是个内存映射的输出寄存器, 这也是为什么它没有被增加)。它把 swtich 语句和复制 8 个字节的循环交织在一起, 从而解决了剩余字节的处理问题 (当 count 不是 8 的倍数时)。相信不相信, 象这样的把 case 标志放在嵌套在 swtich 语句内的模块中是合法的。当他公布这个技巧给 C 的开发者和世界时, Duff 注意到 C 的 swtich 语法, 特别是 “跌落” 行为, 一直是被争议的, 而 “这段代码在争论中形成了某种论据, 但我不清楚是赞成还是反对”。

20.31 下届国际 C 混乱代码竞赛 (IOCCC) 什么时候进行? 哪里可以找到当前和以前的获胜代码?

竞赛的时间表随着时间而变化; 当前详情请参考 http://www.ioccc.org/index.html。

竞赛的优胜者通常在 Usenix 会议上公布, 结果会在晚些时候公布在网上。前几年 (追述到 1984 年) 的获胜代码在 ftp.uu.net 有档案 (参见问题 18.18), 在目录 pub/ioccc/ 下; 参阅 http://www.ioccc.org/index.html。

20.32 [K&R1] 提到的关健字 entry 是什么？

它是保留起来允许可能有某些函数有多个不同名字的进入点, 就象 FORTRAN 那样。据所有人所知, 它从没被实现过 (也没人记得为它设想的语法是怎样的)。它被放弃了, 也不是 ANSI C 的关健字。参见问题 1.5。

参考资料: [K&R2, p. 259 Appendix C]

20.33 C 的名字从何而来？

C 源自 Ken Thompson 的实验性语言 B, 而 B 由 Martin Richards 的 BCPL (Basic Combined Programming Language) 得到灵感, 而 BCPL 是 CPL (Combined Programming Language 或也许是 Cambridge Programming Language) 的简化版。有一段时间, 人们猜测 C 的后继者会命名为 P (BCPL 的第三个字母) 而不是 D, 当然, 如今最显见的后裔语言是 C++。

20.34 “char” 如何发音？

C 关健字 “char” 至少有三种发音: 象英文词“char”, “care” 或 “car” (又或者“character”); 你可以任选一个。

20.35 “lvalue” 和 “rvalue” 代表什么意思？

简单的说, “lvalue” 是个可以出现在赋值语句左方的表达式; 你也可以把它想象成有地址的对象。有关数组的, 参见问题 6.5。“rvalue” 就是有值的表达式, 所以可以用在赋值语句的右方。

20.36 哪里可以取得本 FAQ (英文版) 的额外副本？

最新的副本可以从 ftp.eskimo.com 的目录 u/s/scs/C-faq/ 取得. 你也可以从网上下载。通常, 本 FAQ 在每个月的第一天会发布在新闻组 comp.lang.c。其中含有期限行, 应该可以保留一个月。同时, 还有一个简略的版本 (也发布在新闻组), 以及对于每一个有显著更新的版本, 都会有个改变列表。

本 FAQ 的各个版本也会发布在新闻组 comp.answers 和 news.answers。有几个网站归档了 news.answers 的帖子和其它 FAQ 单子, 包括本文; 其中的两个网站: rtfm.mit.edu (目录 pub/usenet/news.answers/C-faq/ 和 pub/usenet/comp.lang.c/); ftp.uu.net (目录 usenet/news.answers/C-faq/)。如果你没有 ftp 的访问, rtfm.mit.edu 有个邮件服务器可以电邮给你 FAQ 的名单: 发个只有一个词 “help” 的电邮到 mail-server@rtfm.mit.edu。更多的信息, 请参阅 news.answers 中的 meta-FAQ。

本 FAQ 还有一个超文本 (HTML) 的版本在万维网上; URL http://www.eskimo.com/~scs/C-faq/top.html。一个非常全面的含有所有 Usenet FAQ 的网站是 http://www.faqs.org/faqs/。

本 FAQ 的扩展版本已经由 Addison-Wesley 出版了: 《C Programming FAQs: Frequently Asked Questions》(ISBN 0-201-84519-9) [CFAQ]。勘误表在 http://www.eskimo.com/~scs/C-faq/book/Errata.html 以及 ftp://ftp.eskimo.com/u/s/scs/ftp/C-faq/book/Errata。

【译者注】最新的 HTML 中译版本可以在 http://c-faq-chn.sourceforge.net/ 取得。另外在同一地址还提供 PDF 版本的下载。

第 21 章

感谢

感谢：Jamshid Afshar, Lauri Alanko, Michael B. Allen, David Anderson, Jens Andreasen, Tanner Andrews, Sudheer Apte, Joseph Arceneaux, Randall Atkinson, Kaleb Axon, Daniel Barker, Rick Beem, Peter Bennett, Mathias Bergqvist, Wayne Berke, Dan Bernstein, Tanmoy Bhattacharya, John Bickers, Kevin Black, Gary Blaine, Yuan Bo, Mark J. Bobak, Anthony Borla, Dave Boutcher, Alan Bowler, breadbox@muppetlabs.com, Michael Bresnahan, Walter Briscoe, Vincent Broman, Robert T. Brown, Stan Brown, John R. Buchan, Joe Buehler, Kimberley Burchett, Gordon Burditt, Scott Burkett, Eberhard Burr, Burkhard Burow, Conor P. Cahill, D'Arcy J.M. Cain, Christopher Calabrese, Ian Cargill, Vinit Carpenter, Paul Carter, Mike Chambers, Billy Chambless, C. Ron Charlton, Franklin Chen, Jonathan Chen, Raymond Chen, Richard Cheung, Avinash Chopde, Steve Clamage, Ken Corbin, Dann Corbit, Ian Cottam, Russ Cox, Jonathan Coxhead, Lee Crawford, Nick Cropper, Steve Dahmer, Jim Dalsimer, Andrew Daviel, James Davies, John E. Davis, Ken Delong, Norm Diamond, Jamie Dickson, Bob Dinse, dlynes@plenary-software, Colin Dooley, Jeff Dunlop, Ray Dunn, Stephen M. Dunn, Andrew Dunstan, Michael J. Eager, Scott Ehrlich, Arno Eigenwillig, Yoav Eilat, Dave Eisen, Joe English, Bjorn Engsig, David Evans, Andreas Fassl, Clive D.W. Feather, Dominic Feeley, Simao Ferraz, Pete Filandr, Bill Finke Jr., Chris Flatters, Rod Flores, Alexander Forst, Steve Fosdick, Jeff Francis, Ken Fuchs, Tom Gambill, Dave Gillespie, Samuel Goldstein, Willis Gooch, Tim Goodwin, Alasdair Grant, W. Wesley Groleau, Ron Guilmette, Craig Gullixson, Doug Gwyn, Michael Hafner, Zhonglin Han, Darrel Hankerson, Tony Hansen, Douglas Wilhelm Harder, Elliotte Rusty Harold, Joe Harrington, Guy Harris, John Hascall, Adrian Havill, Richard Heathfield, Des Herriott, Ger Hobbelt, Sam Hobbs, Joel Ray Holveck, Jos Horsmeier, Syed Zaeem Hosain, Blair Houghton, Phil Howard, Peter Hryczanek, James C. Hu, Chin Huang, Jason Hughes, David Hurt, Einar Indridason, Vladimir Ivanovic, Jon Jagger, Ke Jin, Kirk Johnson, David Jones, Larry Jones, Morris M. Keesan, Arjan Kenter, Bhaktha Keshavachar, James Kew, Bill Kilgore, Darrell Kindred, Lawrence Kirby, Kin-ichi Kitano, Peter Klausler, John Kleinjans, Andrew Koenig, Thomas Koenig, Adam Kolawa, Jukka Korpela, Przemyslaw Kowalczyk, Ajoy Krishnan T, Anders Kristensen, Jon Krom, Markus Kuhn, Deepak Kulkarni, Yohan Kun, B. Kurtz, Kaz Kylheku, Oliver Laumann, John Lauro, Felix Lee, Mike Lee, Timothy J. Lee, Tony Lee, Marty Leisner, Eric Lemings, Dave Lewis, Don Libes, Brian Liedtke, Philip Lijnzaad, James D. Lin, Keith Lindsay, Yen-Wei Liu, Paul Long, Patrick J. LoPresti, Christopher Lott, Tim Love, Paul Lutus, Mike McCarty, Tim McDaniel, Michael MacFaden, Allen Mcintosh, J. Scott McKellar, Kevin McMahon, Stuart MacMartin, John

R. MacMillan, Robert S. Maier, Andrew Main, Bob Makowski, Evan Manning, Barry Margolin, George Marsaglia, George Matas, Brad Mears, Wayne Mery, De Mickey, Rich Miller, Roger Miller, Bill Mitchell, Mark Moraes, Darren Morby, Bernhard Muenzer, David Murphy, Walter Murray, Ralf Muschall, Ken Nakata, Todd Nathan, Taed Nelson, Pedro Zorzenon Neto, Daniel Nielsen, Landon Curt Noll, Tim Norman, Paul Nulsen, David O'Brien, Richard A. O'Keefe, Adam Kolawa, Keith Edward O'hara, James Ojaste, Max Okumoto, Hans Olsson, Thomas Otahal, Lloyd Parkes, Bob Peck, Harry Pehkonen, Andrew Phillips, Christopher Phillips, Francois Pinard, Nick Pitfield, Wayne Pollock, Polver@aol.com, Dan Pop, Don Porges, Claudio Potenza, Lutz Prechelt, Lynn Pye, Ed Price, Kevin D. Quitt, Pat Rankin, Arjun Ray, Eric S. Raymond, Christoph Regli, Peter W. Richards, James Robinson, Greg Roelofs, Eric Roode, Manfred Rosenboom, J.M. Rosenstock, Rick Rowe, Michael Rubenstein, Erkki Ruohtula, John C. Rush, John Rushford, Kadda Sahnine, Tomohiko Sakamoto, Matthew Saltzman, Rich Salz, Chip Salzenberg, Matthew Sams, Paul Sand, DaviD W. Sanderson, Frank Sandy, Christopher Sawtell, Jonas Schlein, Paul Schlyter, Doug Schmidt, Rene Schmit, Russell Schulz, Dean Schulze, Jens Schweikhardt, Chris Sears, Peter Seebach, Gisbert W. Selke, Patricia Shanahan, Girija Shanker, Clinton Sheppard, Aaron Sherman, Raymond Shwake, Nathan Sidwell, Thomas Siegel, Peter da Silva, Andrew Simmons, Joshua Simons, Ross Smith, Thad Smith, Henri Socha, Leslie J. Somos, Eric Sosman, Henry Spencer, David Spuler, Frederic Stark, James Stern, Zalman Stern, Michael Sternberg, Geoff Stevens, Alan Stokes, Bob Stout, Dan Stubbs, Tristan Styles, Richard Sullivan, Steve Sullivan, Melanie Summit, Erik Talvola, Christopher Taylor, Dave Taylor, Clarke Thatcher, Wayne Throop, Chris Torek, Steve Traugott, Brian Trial, Nikos Triantafillis, Ilya Tsindlekht, Andrew Tucker, Goran Uddeborg, Rodrigo Vanegas, Jim Van Zandt, Momchil Velikov, Wietse Venema, Tom Verhoeff, Ed Vielmetti, Larry Virden, Chris Volpe, Mark Warren, Alan Watson, Kurt Watzka, Larry Weiss, Martin Weitzel, Howard West, Tom White, Freek Wiedijk, Stephan Wilms, Tim Wilson, Dik T. Winter, Lars Wirzenius, Dave Wolverton, Mitch Wright, Conway Yee, James Youngman, Ozan S. Yigit, and Zhuo Zang。他们为本文提供了直接或间接的贡献。

感谢对出版版本进行审核的：Mark Brader, Vinit Carpenter, Stephen Clamage, Jutta Degener, Doug Gwyn, Karl Heuer, and Joseph Kent. 感谢 Addison-Wesley 的 Debbie Lafferty 和 Tom Stone 的鼓励，以及允许从书中引入新的内容到本文。特别感谢 Karl Heuer, Jutta Degener, 特别是 Mark Brader, 他 (借用 Steve Johnson 的一句话) 为了不断的追求一个更好的 FAQ 而鞭策我超越自己的爱好，偶而超出了我的忍耐性。

Steve Summit
scs@eskimo.com

文献

[ANSI] American National Standards Institute, 《American National Standard for Information Systems – Programming Language – C》, ANSI X3.159-1989, (参见问题 11.2)

[Rationale] American National Standards Institute, 《Rationale for American National Standard for Information Systems – Programming Language – C》 (参见问题 11.2)

[Bentley] Jon Bentley, 《Writing Efficient Programs》, Prentice-Hall, 1982, ISBN 0-13-970244-X.

[Burki] David Burki, "Date Conversions," 《The C Users Journal》, February 1993, pp. 29-34.

[LINT] Ian F. Darwin, 《Checking C Programs with lint》, O'Reilly, 1988, ISBN 0-937175-30-7.

[Goldberg] David Goldberg, "What Every Computer Scientist Should Know about Floating-Point Arithmetic," 《ACM Computing Surveys》, Vol. 23 #1, March, 1991, pp. 5-48.

[H&S] Samuel P. Harbison and Guy L. Steele, Jr., 《C: A Reference Manual》, Fourth Edition, Prentice-Hall, 1995, ISBN 0-13-326224-3. [There is also a fifth edition: 2002, ISBN 0-13-089592-X.]

[PCS] Mark R. Horton, 《Portable C Software》, Prentice Hall, 1990, ISBN 0-13-868050-7.

[POSIX] Institute of Electrical and Electronics Engineers, 《Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API) [C Language]》, IEEE Std. 1003.1, ISO/IEC 9945-1.

[ISO] International Organization for Standardization, ISO 9899:1990 (参见问题 11.2)

[C9X] International Organization for Standardization, WG14/N794 Working Draft (参见问题 11.1 和 11.3)

[K&P] Brian W. Kernighan and P.J. Plauger, 《The Elements of Programming Style》, Second Edition, McGraw-Hill, 1978, ISBN 0-07-034207-5.

[K&R1] Brian W. Kernighan and Dennis M. Ritchie, 《The C Programming Language》, Prentice-Hall, 1978, ISBN 0-13-110163-3.

[K&R2] Brian W. Kernighan and Dennis M. Ritchie, 《The C Programming Language》, Second Edition, Prentice Hall, 1988, ISBN 0-13-110362-8, 0-13-110370-9. （参见问题 18.9）

[Knuth] Donald E. Knuth, 《The Art of Computer Programming》. Volume 1: 《Fundamental Algorithms》, Third Edition, Addison-Wesley, 1997, ISBN 0-201-89683-4. Volume 2: 《Seminumerical Algorithms》, Third Edition, 1997, ISBN 0-201-89684-2. Volume 3: 《Sorting and Searching》, Second Edition, 1998, ISBN 0-201-89685-0.

[CT&P] Andrew Koenig, 《C Traps and Pitfalls》, Addison-Wesley, 1989, ISBN 0-201-17928-8.

[Marsaglia&Bray] G. Marsaglia and T.A. Bray,"A Convenient Method for Generating Normal Variables," 《SIAM Review》, Vol. 6 #3, July, 1964.

[P&M] Stephen K. Park and Keith W. Miller, "Random Number Generators: Good Ones are Hard to Find," 《Communications of the ACM》, Vol. 31 #10, October, 1988, pp. 1192-1201 (also technical correspondence August, 1989, pp. 1020-1024, and July, 1993, pp. 108-110).

[Plauger] P.J. Plauger, 《The Standard C Library》, Prentice Hall, 1992, ISBN 0-13-131509-9.

[Plum] Thomas Plum, 《C Programming Guidelines》, Second Edition, Plum Hall, 1989, ISBN 0-911537-07-4.

[Press et al.] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, 《Numerical Recipes in C》, Second Edition, Cambridge University Press, 1992, ISBN 0-521-43108-5.

[Sedgewick] Robert Sedgewick, 《Algorithms in C》, Addison-Wesley, 1990, ISBN 0-201-51425-7. (A new edition is being prepared; the first two volumes are ISBN 0-201-31452-5 and 0-201-31663-3.)

[Simonyi&Heller] Charles Simonyi and Martin Heller,"The Hungarian Revolution," 《Byte》, August, 1991, pp. 131-13.

[Straker] David Straker, 《C Style: Standards and Guidelines》, Prentice Hall, ISBN 0-13-116898-3.

[CFAQ] Steve Summit, 《C Programming FAQs: Frequently Asked Questions》, Addison- Wesley, 1995, ISBN 0-201-84519-9. [The book version of this FAQ list; see also http://www.eskimo.com/~scs/C-faq/book/Errata.html .]

[Expert] Peter van der Linden, 《Expert C Programming: Deep C Secrets》, Prentice Hall, 1994, ISBN 0-13-177429-8.

[AGREP] Sun Wu and Udi Manber, "AGREP – A Fast Approximate Pattern-Matching Tool," USENIX Conference Proceedings, Winter, 1992, pp. 153-162.

[Schumacher, ed.] Schumacher, ed., Software Solutions in C。

[IHSG] 印第安山风格指南（参见问题 17.7）的修订版有另外一个参考书目。参见问题 18.9。