

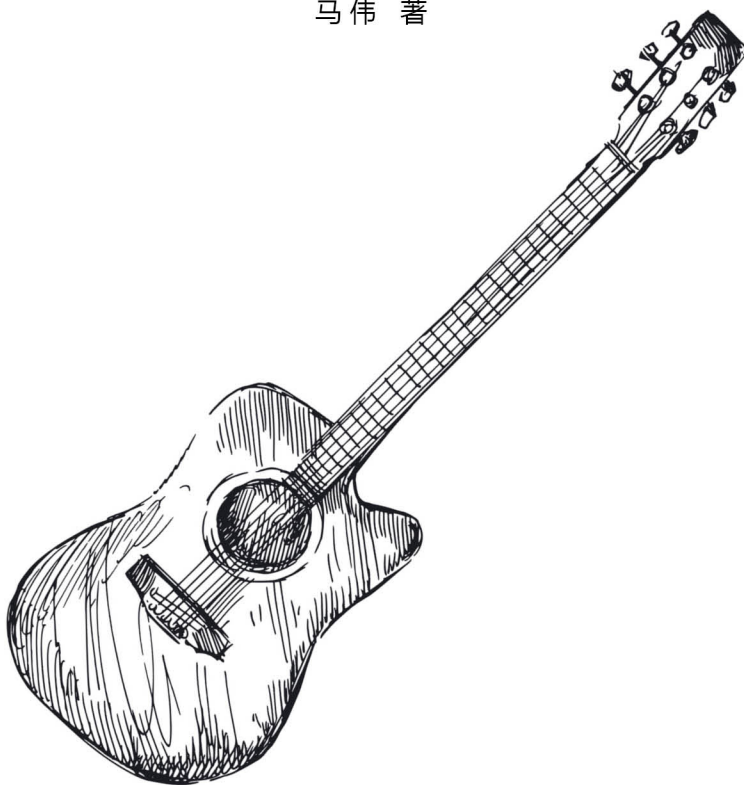
10余年开发经验的资深C语言专家全面从C语法和C11标准两大方面深入探讨编写高质量C代码的技巧、禁忌和最佳实践

Writing Solid Code
125 Suggestions to Improve Your C Program

编写高质量代码

改善C程序代码的 125个建议

马伟 著



机械工业出版社
China Machine Press

Effective 系列丛书

编写高质量代码：改善 C 程序代码的 125 个建议

马 伟 著



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

编写高质量代码：改善 C 程序代码的 125 个建议 / 马伟著. —北京：机械工业出版社，2016.1

(Effective 系列丛书)

ISBN 978-7-111-52434-2

I. 编… II. 马… III. C 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字 (2015) 第 313259 号



编写高质量代码：改善 C 程序代码的 125 个建议

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）

责任编辑：姜 影 高婧雅

责任校对：殷 虹

印 刷：

版 次：2016 年 1 月第 1 版第 1 次印刷

开 本：186mm×240mm 1/16

印 张：29.25

书 号：ISBN 978-7-111-52434-2

定 价：89.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：(010) 88379426 88361066

投稿热线：(010) 88379604

购书热线：(010) 68326294 88379649 68995259

读者信箱：hzit@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问：北京大成律师事务所 韩光 / 邹晓东

为什么写作本书

众所周知，C 语言是一门既具有高级语言特点，又有汇编语言特点的通用计算机编程语言，无论是操作系统（如 Microsoft Windows、Mac OS X、Linux 和 UNIX 等）、嵌入式系统与普通应用软件，还是目前流行的移动智能设备开发，随处都可以看见它依然矫健的身影。它能够轻松地应用于各类层次的开发中，从设备驱动程序和操作系统组件到大规模应用程序，它都能够很好地胜任。毋庸置疑，它是二十几年来使用最为广泛、生命力最强的编程语言，它的设计思想也影响了众多后来的编程语言，例如 C++、Objective-C、Java、C# 等。

尽管 C 语言有着悠久的历史 and 广泛的使用场景，但它依旧让大部分计算机编程人员望而生畏，相信绝大多数读者也还停留在“入门者”这个阶段。所谓“入门者”指的是已经可以简单使用 C 语言编写普通应用程序，但是却不明白如何编写高质量代码的人。面对这样的实际情况，在准备编写本书之前，一连串的问题深深地映入笔者的脑海：到底什么样的编程书籍才能够帮助“入门者”快速进阶？面对市面上众多的优秀 C 语言编程书籍，编写本书的价值何在？怎样的内容才能够与众不同？

带着这一连串的问题，笔者开始回顾自己这些年的开发生涯，发现如下几类问题经常困扰“入门者”：

- ❑ 基础数据类型问题：如数据取值范围、整数溢出与回绕、浮点数精度、数据类型转换的范围检查等。
- ❑ 数组与指针问题：指针与地址、野指针、空（null）指针、NULL 指针、void 指针、多级指针、指针函数与函数指针，以及数组越界与缓冲区溢出等。
- ❑ 内存管理问题：内存分配、内存释放、内存越界与内存泄漏等。
- ❑ 字符与字符串问题：串拷贝与内存拷贝，内存重叠与溢出，字符串查找等。

□ 高效设计问题：表达式设计、算法设计与函数设计，内联函数与宏的取舍等。

□ 其他杂项问题：信号处理、文件系统、断言与异常处理、内嵌汇编的使用等。

如果你同样也苦于处理这些问题，或者对这些问题模棱两可，那么本书正是为你所准备的。本书为普遍存在于初级与中级开发者脑海中的那些问题给出了经验性的解决方案。全书分为 15 章，通过 125 个建议深度剖析 C 语言程序设计中的常见性问题，并给出经验性的解决方案。除此之外，为了使读者能够尽量做到“知其所以然”，本书重点阐述了一些尖锐的问题，如 IEEE 754 浮点数、指针与数组、越界与溢出等问题。当然，这些经验和心得的积累并非我一人之力，“我只不过是站在巨人的肩膀上而已”。因此，在撰写本书的过程中也参考了大量的资料，如 www.securecoding.cert.org 的《SEI CERT C Coding Standard》、ISO/IEC 9899:1990、ISO/IEC 9899:1999 与 ISO/IEC 9899:201x 标准文档等。

如何阅读本书

本书适合那些有一定 C 语言基础并希望快速提升程序设计能力的初级与中级程序员。因此，本书并不会阐述 C 语言中的一些基础概念，而是将 C 语言编程过程中可能遇到的疑问或者障碍进行一一列举与剖析，并给出了经验性解决方案与建议。

如果你是一位有一定 C 语言编程基础的初中级读者，本书就是为你量身打造的。你可以逐章进行系统性学习，并结合我们提供的源码动手实践，巩固所学的知识。书中的大多数建议实战性很强，要完全理解其中的奥妙，请果断地放弃 `printf` 函数，多调试一下程序，编程高手都是调试出来的；如果你是一位编程经验非常丰富的高级读者，那么可以将书中的大部分经验与自己的一些经验进行融合，从而获得更多提高与升华。

资源及勘误

通常情况下，一个问题的解决方案往往不止一种，你可能会不同意本书中的一些观点，甚至强烈反对。同时，尽管笔者在本书的写作过程中非常认真与努力，但由于水平有限，书中难免存在错误和不足之处，恳请批评指正。如果你对本书有什么意见、问题或想法，欢迎使用下面的邮箱通知笔者，笔者将不胜感激。当然，也可以通过微信（sc-mawei）与笔者取得联系，共同进行技术交流。

Email: madengwei@hotmail.com

特别鸣谢

最后，要感谢那些所有帮助过笔者的人，没有他们的帮助与付出，这本书很难顺利完

成。尤其要感谢下面这些人：

首先，机械工业出版社的杨福川与姜影为本书的整体策划、审阅和出版做了大量的工作，与他们的合作是非常愉快的。同时，由于写作过程漫长，难免令笔者情绪波动，是他们给了我一如既往的支持与鼓励，当我想要放弃的时候，是他们的敦促让我对写作时刻保持着热情，坚持完成本书。也正因为他们对本书的不断要求，才使得本书的结构更加系统化，内容更加深刻，语言更加简单易懂。

其次，要感谢家人的支持。为了编写本书，笔者投入了大量的时间和精力，牺牲了许多可以陪家人的周末和节假日。

最后，要感谢那些曾经为本书的编写提过意见的朋友，感谢他们对本书的默默支持。

马 伟



目 录 *Contents*

前 言

第 1 章 数据，程序设计之根本 1

 建议 1：认识 ANSI C 1

 建议 2：防止整数类型产生回绕与溢出 6

 建议 2-1：char 类型变量的值应该限制在 signed char 与 unsigned char 的交集范围内 11

 建议 2-2：使用显式声明为 signed char 或 unsigned char 的类型来执行算术运算 11

 建议 2-3：使用 rsize_t 或 size_t 类型来表示一个对象所占用空间的整数值单位 13

 建议 2-4：禁止把 size_t 类型和它所代表的真实类型混用 16

 建议 2-5：小心使用无符号类型带来的陷阱 16

 建议 2-6：防止无符号整数回绕 19

 建议 2-7：防止有符号整数溢出 24

 建议 3：尽量少使用浮点类型 28

 建议 3-1：了解 IEEE 754 浮点数 29

 建议 3-2：避免使用浮点数进行精确计算 39

 建议 3-3：使用分数来精确表达浮点数 43

 建议 3-4：避免直接在浮点数中使用“==”操作符做相等判断 47

 建议 3-5：避免使用浮点数作为循环计数器 50

 建议 3-6：尽量将浮点运算中的整数转换为浮点数 51

 建议 4：数据类型转换必须做范围检查 52

 建议 4-1：整数转换为新类型时必须做范围检查 53

 建议 4-2：浮点数转换为新类型时必须做范围检查 56

建议 5: 使用有严格定义的数据类型	57
建议 6: 使用 typedef 来定义类型的新别名	61
建议 6-1: 掌握 typedef 的 4 种应用形式	61
建议 6-2: 小心使用 typedef 带来的陷阱	65
建议 6-3: typedef 不同于 #define	65
建议 7: 变量声明应该力求简洁	66
建议 7-1: 尽量不要在一个声明中声明超过一个的变量	67
建议 7-2: 避免在嵌套的代码块之间使用相同的变量名	68
建议 8: 正确地选择变量的存储类型	68
建议 8-1: 定义局部变量时应该省略 auto 关键字	69
建议 8-2: 慎用 extern 声明外部变量	70
建议 8-3: 不要混淆 static 变量的作用	72
建议 8-4: 尽量少使用 register 变量	75
建议 9: 尽量不要在可重入函数中使用静态 (或全局) 变量	76
建议 10: 尽量少使用全局变量	78
建议 11: 尽量使用 const 声明值不会改变的变量	78
第 2 章 保持严谨的程序设计, 一切从表达式开始做起	81
建议 12: 尽量减少使用除法运算与求模运算	81
建议 12-1: 用倒数相乘来实现除法运算	82
建议 12-2: 使用牛顿迭代法求除数的倒数	84
建议 12-3: 用减法运算来实现整数除法运算	86
建议 12-4: 用移位运算实现乘法运算	86
建议 12-5: 尽量将浮点除法转化为相应的整数除法运算	87
建议 13: 保证除法和求模运算不会导致除零错误	87
建议 14: 适当地使用位操作来提高计算效率	88
建议 14-1: 尽量避免对未知的有符号数执行位操作	89
建议 14-2: 在右移中合理地选择 0 或符号位来填充空出的位	90
建议 14-3: 移位的数量必须大于等于 0 且小于操作数的位数	90
建议 14-4: 尽量避免在同一个数据上执行位操作与算术运算	91
建议 15: 避免操作符混淆	92

建议 15-1: 避免 “=” 与 “==” 混淆	92
建议 15-2: 避免 “ ” 与 “ ” 混淆	94
建议 15-3: 避免 “&” 与 “&&” 混淆	95
建议 16: 表达式的设计应该兼顾效率与可读性	95
建议 16-1: 尽量使用复合赋值运算符	95
建议 16-2: 尽量避免编写多用途的、太复杂的复合表达式	97
建议 16-3: 尽量避免在表达式中使用默认的优先级	98

第 3 章 程序控制语句应该保持简洁高效

101

建议 17: if 语句应该尽量保持简洁, 减少嵌套的层数	101
建议 17-1: 先处理正常情况, 再处理异常情况	101
建议 17-2: 避免 “悬挂” 的 else	102
建议 17-3: 避免在 if/else 语句后面添加分号 “;”	105
建议 17-4: 对深层嵌套的 if 语句进行重构	106
建议 18: 谨慎 0 值比较	108
建议 18-1: 避免布尔型与 0 或 1 进行比较	108
建议 18-2: 整型变量应该直接与 0 进行比较	109
建议 18-3: 避免浮点变量用 “==” 或 “!=” 与 0 进行比较	109
建议 18-4: 指针变量应该用 “==” 或 “!=” 与 NULL 进行比较	111
建议 19: 避免使用嵌套的 “?:”	111
建议 20: 正确使用 for 循环	114
建议 20-1: 尽量使循环控制变量的取值采用半开半闭区间写法	114
建议 20-2: 尽量使循环体内工作量达到最小化	115
建议 20-3: 避免在循环体内修改循环变量	115
建议 20-4: 尽量使逻辑判断语句置于循环语句外层	116
建议 20-5: 尽量将多重循环中最长的循环放在最内层, 最短的循环放在最外层	117
建议 20-6: 尽量将循环嵌套控制在 3 层以内	117
建议 21: 适当地使用并行代码来优化 for 循环	117
建议 22: 谨慎使用 do/while 与 while 循环	118
建议 22-1: 无限循环优先选用 for(;;), 而不是 while(1)	118
建议 22-2: 优先使用 for 循环替代 do/while 与 while 循环	119

建议 23: 正确地使用 switch 语句	120
建议 23-1: 不要忘记在 case 语句的结尾添加 break 语句	120
建议 23-2: 不要忘记在 switch 语句的结尾添加 default 语句	122
建议 23-3: 不要为了使用 case 语句而刻意构造一个变量	122
建议 23-4: 尽量将长的 switch 语句转换为嵌套的 switch 语句	123
建议 24: 选择合理的 case 语句排序方法	124
建议 24-1: 尽量按照字母或数字顺序来排列各条 case 语句	124
建议 24-2: 尽量将情况正常的 case 语句排在最前面	125
建议 24-3: 尽量根据发生频率来排列各条 case 语句	125
建议 25: 尽量避免使用 goto 语句	125
建议 26: 区别 continue 与 break 语句	127
第 4 章 函数同样需要保持简洁高效	129
建议 27: 理解函数声明	129
建议 28: 理解函数原型	131
建议 29: 尽量使函数的功能单一	132
建议 30: 避免把没有关联的语句放在一个函数中	135
建议 31: 函数的抽象级别应该在同一层次	136
建议 32: 尽可能为简单功能编写函数	137
建议 33: 避免多段代码重复做同一件事情	138
建议 34: 尽量避免编写不可重入函数	140
建议 34-1: 避免在函数中使用 static 局部变量	140
建议 34-2: 避免函数返回指向静态数据的指针	140
建议 34-3: 避免调用任何不可重入函数	142
建议 34-4: 对于全局变量, 应通过互斥信号量 (即 P、V 操作) 或者中断机制等方法 来保证函数的线程安全	143
建议 34-5: 理解可重入函数与线程安全函数之间的关系	144
建议 35: 尽量避免设计多参数函数	145
建议 35-1: 没有参数的函数必须使用 void 填充	145
建议 35-2: 尽量避免在非调度函数中使用控制参数	147
建议 35-3: 避免将函数的参数作为工作变量	148

建议 35-4: 使用 const 防止指针类型的输入参数在函数体内被意外修改	149
建议 36: 没有返回值的函数应声明为 void 类型	149
建议 37: 确保函数体的“入口”与“出口”安全性	150
建议 37-1: 尽量在函数体入口处对参数做有效性检查	150
建议 37-2: 尽量在函数体出口处对 return 语句做安全性检查	151
建议 38: 在调用函数时, 必须对返回值进行判断, 同时对错误的返回值还要有 相应的错误处理	152
建议 39: 尽量减少函数本身或者函数间的递归调用	153
建议 40: 尽量使用 inline 内联函数来替代 #define 宏	154
第 5 章 不会使用指针的程序员是不合格的	157
建议 41: 理解指针变量的存储实质	157
建议 42: 指针变量必须初始化	162
建议 43: 区别 “int *p = NULL” 和 “*p = NULL”	163
建议 44: 理解空 (null) 指针与 NULL 指针	164
建议 44-1: 区别空 (null) 指针与 NULL 指针的概念	164
建议 44-2: 用 NULL 指针终止对递归数据结构的间接引用	166
建议 44-3: 用 NULL 指针作函数调用失败时的返回值	169
建议 44-4: 用 NULL 指针作警戒值	170
建议 44-5: 避免对 NULL 指针进行解引用	170
建议 45: 谨慎使用 void 指针	171
建议 45-1: 避免对 void 指针进行算术操作	172
建议 45-2: 如果函数的参数可以是任意类型指针, 应该将其参数声明为 void *	173
建议 46: 避免使用指针的长度确定它所指向类型的长度	175
建议 47: 避免把指针转换为对齐要求更严格的指针类型	176
建议 48: 避免将一种类型的操作符应用于另一种不兼容的类型	177
建议 49: 谨慎指针与整数之间的转换	180
建议 50: 区别 “const int *p” 与 “int *const p”	180
建议 51: 深入理解函数参数的传递方式	183
建议 51-1: 理解函数参数的传递过程	183
建议 51-2: 掌握函数的参数传递方式	188

建议 51-3: 如果函数的参数是指针, 避免用该指针去申请动态内存	191
建议 51-4: 尽量避免使用可变参数	195
第 6 章 数组并非指针	199
建议 52: 理解数组的存储实质	199
建议 52-1: 理解数组的存储布局	199
建议 52-2: 理解 &a[0] 和 &a 的区别	203
建议 52-3: 理解数组名 a 作为右值和左值的区别	203
建议 53: 避免数组越界	204
建议 53-1: 尽量显式地指定数组的边界	207
建议 53-2: 对数组做越界检查, 确保索引值位于合法的范围之内	209
建议 53-3: 获取数组的长度时不要对指针应用 sizeof 操作符	210
建议 54: 数组并非指针	213
建议 55: 理解数组与指针的可交换性	217
建议 56: 禁止将一个指向非数组对象的指针加上或减去一个整数	219
建议 57: 禁止对两个并不指向同一个数组的指针进行相减或比较	220
建议 58: 若结果值并不引用合法的数组元素, 不要将指针加上或减去一个整数	220
建议 59: 细说缓冲区溢出	220
建议 60: 区别指针数组和数组指针	226
建议 61: 深入理解数组参数	227
第 7 章 结构、位域和枚举	231
建议 62: 结构体的设计要遵循简单、单一原则	231
建议 62-1: 尽量使结构体的功能单一	232
建议 62-2: 尽量减小结构体间关系的复杂度	234
建议 62-3: 尽量使结构体中元素的个数适中	235
建议 62-4: 合理划分与改进结构体以提高空间效率	236
建议 63: 合理利用结构体内存对齐原理来提高程序效率	237
建议 64: 结构体的长度不一定等于各个成员的长度之和	249
建议 65: 避免在结构体之间执行逐字节比较	250
建议 66: 谨慎使用位域	251

建议 67: 谨慎使用枚举	252
建议 68: 禁止在位域成员上调用 <code>offsetof</code> 宏	254
建议 69: 深入理解结构体数组和结构体指针	255
第 8 章 字符与字符串	260
建议 70: 不要忽视字符串的 <code>null ('\\0')</code> 结尾符	260
建议 70-1: 正确认识字符数组和字符串	261
建议 70-2: 字符数组必须能够同时容纳字符数据和 <code>null</code> 结尾符	262
建议 70-3: 谨慎字符数组的初始化	263
建议 71: 尽量使用 <code>const</code> 指针来引用字符串常量	264
建议 72: 区别 <code>strlen</code> 函数与 <code>sizeof</code> 运算符	264
建议 73: 在使用不受限制的字符串函数时, 必须保证结果字符串不会溢出内存	265
建议 73-1: 避免字符串拷贝发生溢出	266
建议 73-2: 区别串拷贝 <code>strcpy</code> 与内存拷贝 <code>memcpy</code>	270
建议 73-3: 避免 <code>strcpy</code> 与 <code>memcpy</code> 函数内存重叠	273
建议 73-4: 区别字符串比较与内存比较	278
建议 73-5: 避免 <code>strcat</code> 函数发生内存重叠与溢出	283
建议 74: 谨慎 <code>strtok</code> 函数的不可重入性	287
建议 75: 掌握字符串查找技术	292
建议 75-1: 使用 <code>strchr</code> 与 <code>strrchr</code> 函数查找单个字符	292
建议 75-2: 使用 <code>strpbrk</code> 函数查找多个字符	293
建议 75-3: 使用 <code>strstr</code> 函数查找一个子串	294
建议 75-4: 区别 <code>strspn</code> 与 <code>strcspn</code> 函数	295
第 9 章 文件系统	298
建议 76: 谨慎使用 <code>printf</code> 和 <code>scanf</code> 函数	299
建议 77: 谨慎文件打开操作	308
建议 77-1: 正确指定 <code>fopen</code> 的 <code>mode</code> 参数	309
建议 77-2: 必须检查 <code>fopen</code> 函数的返回值	310
建议 77-3: 尽量避免重复打开已经被打开的文件	311

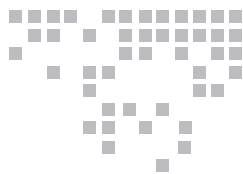
建议 77-4: 区别 <code>fopen</code> 与 <code>fopen_s</code> 函数	312
建议 77-5: 区别 <code>fopen</code> 与 <code>freopen</code> 函数	313
建议 78: 文件操作完成后必须关闭	313
建议 79: 正确理解 <code>EOF</code> 宏	314
建议 80: 尽量使用 <code>feof</code> 和 <code>ferror</code> 检测文件结束和错误	316
建议 81: 尽量使用 <code>fgets</code> 替换 <code>gets</code> 函数	319
建议 82: 尽量使用 <code>fputs</code> 替换 <code>puts</code> 函数	321
建议 83: 合理选择单个字符读写函数	323
建议 84: 区别格式化读写函数	324
建议 84-1: 区别 <code>printf/scanf</code> 、 <code>fprintf/fscanf</code> 和 <code>sprintf/sscanf</code>	325
建议 84-2: 尽量使用 <code>snprintf</code> 替代 <code>sprintf</code> 函数	327
建议 84-3: 区别 <code>vprintf/vscanf</code> 、 <code>vfprintf/vfscanf</code> 、 <code>vsprintf/vsscanf</code> 和 <code>vsnprintf</code>	328
建议 85: 尽量使用 <code>fread</code> 与 <code>fwrite</code> 函数来读写二进制文件	330
建议 86: 尽量使用 <code>fseek</code> 替换 <code>rewind</code> 函数	332
建议 87: 尽量使用 <code>setvbuf</code> 替换 <code>setbuf</code> 函数	334
建议 88: 谨慎 <code>remove</code> 函数删除已打开的文件	336
建议 89: 谨慎 <code>rename</code> 函数重命名已经存在的文件	337
 第 10 章 预处理器	 338
建议 90: 谨慎宏定义	338
建议 90-1: 在使用宏定义表达式时必须使用完备的括号	339
建议 90-2: 尽量消除宏的副作用	340
建议 90-3: 避免使用宏创建一种“新语言”	342
建议 91: 合理地选择函数与宏	343
建议 92: 尽量使用内联函数代替宏	345
建议 93: 掌握预定义宏	350
建议 94: 谨慎使用 “ <code>#include</code> ”	353
建议 94-1: 区别 “ <code>#include <filename.h></code> ” 与 “ <code>#include "filename.h"</code> ”	354
建议 94-2: 必须保证头文件名称的唯一性	355
建议 95: 掌握条件编译指令	355

建议 95-1: 使用 “#ifndef/#define/#endif” 防止头文件被重复引用	355
建议 95-2: 使用条件编译指令实现源代码的部分编译	357
建议 95-3: 妙用 “defined”	358
建议 96: 尽量避免在一个函数块中单独使用 “#define” 或 “#undef”	359
第 11 章 断言与异常处理	361
建议 97: 谨慎使用断言	361
建议 97-1: 尽量利用断言来提高代码的可测试性	362
建议 97-2: 尽量在函数中使用断言来检查参数的合法性	367
建议 97-3: 避免在断言表达式中使用改变环境的语句	368
建议 97-4: 避免使用断言去检查程序错误	369
建议 97-5: 尽量在防错性程序设计中 使用断言来进行错误报警	370
建议 97-6: 用断言保证没有定义的特性或功能不被使用	372
建议 97-7: 谨慎使用断言对程序开发环境中的假设进行检查	373
建议 98: 谨慎使用 <code>errno</code>	374
建议 98-1: 调用 <code>errno</code> 之前必须先将其清零	375
建议 98-2: 避免重定义 <code>errno</code>	377
建议 98-3: 避免使用 <code>errno</code> 检查文件流错误	379
建议 99: 谨慎使用函数的返回值来标志函数是否执行成功	380
建议 100: 尽量避免使用 <code>goto</code> 进行出错跳转	380
建议 101: 尽量避免使用 <code>setjmp</code> 与 <code>longjmp</code> 组合	381
第 12 章 内存管理	384
建议 102: 浅谈程序的内存结构	384
建议 103: 浅谈堆和栈	389
建议 104: 避免错误分配内存	396
建议 104-1: 对内存分配函数的返回值必须进行检查	397
建议 104-2: 内存资源的分配与释放应该限定在同一模块或者同一抽象层内进行	398
建议 104-3: 必须对内存分配函数的返回指针进行强制类型转换	400
建议 104-4: 确保指针指向一块合法的内存	401
建议 104-5: 确保为对象分配足够的内存空间	402

建议 104-6: 禁止执行零长度的内存分配	405
建议 104-7: 避免大型的堆栈分配	405
建议 104-8: 避免内存分配成功, 但并未初始化	407
建议 105: 确保安全释放内存	407
建议 105-1: malloc 等内存分配函数与 free 必须配对使用	407
建议 105-2: 在 free 之后必须为指针赋一个新值	409
建议 106: 避免内存越界	411
建议 106-1: 避免数组越界	412
建议 106-2: 避免 sprintf、vsprintf、strcpy、strcat 与 gets 越界	413
建议 106-3: 避免 memcpy 与 memset 函数长度越界	413
建议 106-4: 避免忽略字符串最后的 '\0' 字符而导致的越界	413
建议 107: 避免内存泄漏	415
建议 108: 避免 calloc 参数相乘的值超过 size_t 表示的范围	417
第 13 章 信号处理	418
建议 109: 理解信号	418
建议 110: 尽量使用 sigaction 替代 signal	423
建议 111: 避免在信号处理函数内部访问或修改共享对象	428
建议 112: 避免以递归方式调用 raise 函数	429
第 14 章 了解 C11 标准	432
建议 113: 谨慎使用 _Generic	433
建议 114: 尽量使用 gets_s 替换 gets 函数	436
建议 115: 尽量使用带边界检查的字符串操作函数	436
建议 116: 了解 C11 多线程编程	438
建议 117: 使用静态断言 _Static_assert 执行编译时检查	442
建议 118: 使用 _Noreturn 标识不返回值的函数	442
第 15 章 保持良好的设计	443
建议 119: 避免错误地变量初始化	443
建议 120: 谨慎使用内联函数	444

建议 121: 避免在函数内定义占用内存很大的局部变量	445
建议 122: 谨慎设计函数参数的顺序和个数	446
建议 123: 谨慎使用标准函数库	447
建议 124: 避免不必要的函数调用	447
建议 125: 谨慎程序中嵌入汇编代码	448





数据，程序设计之根本

数据是程序设计最基础的概念，程序对数据进行操作。换句话说，任何一个完整的程序都可以看成是一组数据和作用于这组数据上的操作的说明。同时，程序中的每个数据项也都有一个与之相关的类型，称为“数据类型”。

这样，在程序中就可以使用数据类型来区分不同的数据，进而根据实际需要为这些数据分配不同的存储空间。这就像成年人必须睡成人床，而给婴儿配备婴儿床就足够了，如果你给婴儿分配一张成人床就会造成资源浪费，相反给成年人分配一张婴儿床则有可能发生“溢出”。数据类型也一样，由于不同的数据所需要的存储容量各不相同，因此需要分配的内存空间大小也会不一样，这样才能够保证内存资源的合理配置，使程序性能达到最优化。因此，如何合理、安全地使用这些数据类型是每个程序员必须掌握的。本章将围绕这一话题进行讨论。

建议 1：认识 ANSI C

谈到 C 语言的发展历程，就不得不从最早的二进制语言说起。大家都知道，二进制语言可以说是世界上最早的计算机语言，它只允许程序设计人员使用计算机能够直接识别和执行的二进制代码（即 0 和 1，其中，0 代表低电压，1 代表高电压）来编写程序。可想而知，这样的编码方式对程序设计人员来说是多么困难与枯燥。因此，为了提高程序设计效率并减轻程序设计人员的负担，所以后来很快便出现了汇编语言（Assembly Language）。

与二进制语言一样，汇编语言也是面向机器的程序设计语言，不同类型的计算机上需要提供不同的汇编语言。但与二进制语言不同的是，汇编语言使用助记符（Memoni）来代替操

作码，并用地址符号（Symbol）或标号（Label）来代替地址码。由于汇编语言采用符号代替二进制代码，因此，它也被称为符号语言。使用汇编语言编写的程序机器并不能直接识别，而需要通过一种程序将汇编语言翻译成机器能够识别的二进制语言，这种起翻译作用的程序就是汇编程序。

相对于二进制语言，汇编语言不仅使开发效率得到了很大提升，而且它还具有许多优点，比如它能够直接同计算机的底层软件或硬件进行交互，直接访问与硬件相关的存储器或 I/O 端口；能够不受编译器的限制，对生成的二进制代码进行完全控制；能够对关键的代码进行更准确地控制，避免因线程共同访问或硬件设备共享而引起的死锁；能够根据特定的应用对代码进行最佳优化，提高运行速度等。

尽管如此，汇编语言依旧是一种层次非常低的语言，它仅仅高于直接手工编写二进制的机器指令码。在实际应用中，它仍然暴露了一些不可避免的缺陷：如编写的代码非常难以阅读，不好维护；很容易产生 bug，难于调试；一般只能针对特定的体系结构和处理器进行优化；开发效率很低，时间长且单调等。因此，我们更加需要一种设计描述简单，能脱离对机型的要求，并且能在任何计算机上运行的计算机语言，我们称这种语言为高级语言。这样，程序设计人员就可以将问题及解决问题的算法过程描述出来，利用这种高级语言直接写出各种表达式来描述简单的计算过程，而无须针对不同的机型编写不同的代码。



注释 高级语言编写的程序称为源程序，源程序不能在计算机上直接运行，必须将其翻译成二进制代码后才能执行。一般有两种翻译方式：一种是“解释程序”方式，即将源程序作为输入，翻译一句后就提交计算机执行一句，这种方式并不形成目标程序；另一种是“编译程序”方式，即将源程序作为输入，全部翻译成二进制代码后再执行，编译后的二进制程序称为目标程序。

世界上出现的第一种高级语言是 Algol 语言，它也可以算作 C 语言的前身。它和普通语言表达式非常接近，适用于数值计算，所以 Algol 多用于科学计算机。1960 年 Algol 60 版本推出后，很受程序设计人员欢迎。Algol 60 推出了许多新的概念，如局部性概念、动态、递归、巴科斯 - 诺尔范式（Backus-Naur Form, BNF）等。从某种意义上讲，Algol 60 应该是程序设计语言发展史上的一个里程碑，它标志着程序设计语言已成为一门独立的科学学科，并为后来的软件自动化及软件可靠性的发展奠定了基础。

虽然使用 Algol 60 来描述算法很方便，但是它离计算机硬件系统却很远，不宜用来编写系统程序。1963 年英国剑桥大学在 Algol 语言的基础上增添了处理硬件的能力，并命名为“CPL”（Combined Programming Language），即复合程序设计语言。但由于 CPL 的规模很大，学习和掌握都比较困难，因此没有流行。1967 年剑桥大学的 Martin Richards 对 CPL 语言进行了简化，推出 BCPL（Basic Combined Programming Language），即基本复合程序设计语言。它是典型的面向过程的高级语言，它的语法更加靠近机器本身，适合于开发精巧、高要求的

应用程序，而且它对编译器的要求也不高。同时，BCPL 也是最早使用库函数封装基本输入输出的语言之一，这使得它的跨平台可移植性很好。

1969 年，美国通用电气公司、麻省理工学院与贝尔实验室联合创建了一个庞大的项目，命名为 Muktics 工程。该项目的目的是创建一个操作系统，不过由于该项目过于复杂和庞大，最终失败了。这也致使项目的参与者之一通用电气公司退出软件领域，同时，贝尔实验室的专家们也撤出了 Muktics 工程，转而研究新的领域。之后，贝尔实验室的一位名为 Ken Thompson 研究员和他的同事 Dennis Ritchie 组成一个非正式的小组，开始进行一些其他方面的研究。为了自娱自乐，Ken Thompson 把他的“太空旅行”软件移植到不太常用的 PDP-7 系统上。与此同时，Ken Thompson 还为 PDP-7 系统编写了一个简单的操作系统。该操作系统比起 Muktics 工程简单了许多，采用汇编语言编写，1970 年 Brian Kernighan 为其取名为 UNIX。



从这里可以看出，著名的操作系统 UNIX 是早于 C 语言出现的，后来才用 C 语言重写此系统，这一点一定要注意。

不过使用汇编语言编写程序不仅吃力而且效率低下，所以 Ken Thompson 就考虑利用高级语言的特性来解决这一问题。1970 年，Ken Thompson 进一步简化了 BCPL，突出硬件的处理能力，并取“BCPL”的第一个字母“B”作为新语言的名称，即 B 语言。同时，他还使用 B 语言编写了 UNIX 操作系统程序。不过 B 语言还是存在许多问题，最大问题就在于无法表达不同的数据类型，而且效率不高，这也迫使 Ken Thompson 后来不得不在 PDP-11 的基础上重新使用汇编语言来实现 UNIX。面对 B 语言存在的问题，1971 年 Dennis Ritchie 通过增加类型扩展了 B 语言，这次采用的是编译模式而不是解释模式，并且引入了类型系统，每个变量在使用前必须声明。这种扩展的 B 语言称为 NB，即 New B 的缩写。

1972 年，Ken Thompson 和 Dennis Ritchie 继续对 B 语言进行完善和扩充，他们在保留 B 语言强大硬件处理能力的基础上，扩充了数据类型，恢复了通用性，并取“BCPL”的第二个字母“C”作为新语言的名称，即 C 语言。其实，C 语言除了增加类型系统外，它还增加了许多方便编译器设计者设计的新特性，主要表现在以下几个方面：

数组下标从 0 开始，而不是从 1 开始。例如，我们定义一个数组 `arr[50]`，因为 C 语言的数组下标是从 0 开始的，所以它的合法范围是 `arr[0] ~ arr[49]`。因此，你不能够向 `arr[50]` 里存储数据。

- ❑ 可以把数组看作指针，它简化了参数的传递方法，使大家不必忍受传递一个数组到函数时需要复制所有数组内容的低效率。不过，值得注意的是，数组与指针并非在任何情况下都是等效的，这一点会在后面进行详细阐述。
- ❑ `float` 类型被自动扩展为 `double` 类型。虽然在 ANSI C 中情况不再如此，但最初浮点常量的精度都是 `double` 类型的，所有表达式中 `float` 类型的变量总会被自动转化为

double 类型。

- ❑ 增加 register 关键字，用此关键字告诉编译器设计者哪些变量被放到了寄存器中，从而简化编译器，但却也因此给程序员带来了无穷无尽的麻烦，这一点会在后面的章节中详细阐述。

此后，两人又合作重写了 UNIX 操作系统，C 语言也伴随着 UNIX 操作系统成为一种广受欢迎的计算机语言。图 1-1 按时间顺序阐述 C 语言的由来。

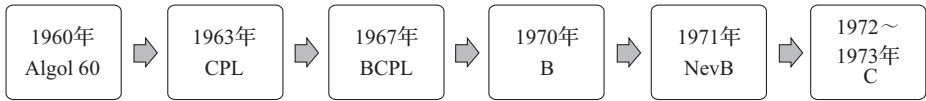


图 1-1 C 语言的由来

1978 年，为了让 C 语言脱离 UNIX 操作系统，成为任何计算机上都能运行的通用计算机语言，Brian Kernighan 和 Dennis Ritchie 共同撰写了《The C Programming Language》的第 1 版，该著作简称为“K&R”。书中对 C 语言的语法进行了规范化描述，书末的参考指南则给出了当时 C 语言的完整定义，这也成为当时 C 语言事实上的标准，此标准称为“K&R C”。从此以后，C 语言被移植到各种机型上，并受到广泛支持。

随着 C 语言在多个领域的推广和应用，一些新的特性不断被各种编译器实现并添加进来。于是建立一个新的“无歧义、与具体平台无关的 C 语言定义”就成为越来越重要的事情。1983 年，美国国家标准委员会 ANSI（American National Standards Institute）属下专门负责信息技术标准化的机构 ASC X3（现已更名为国际信息技术标准委员会（International Committee for Information Technology Standards, INCITS））成立了一个专门的技术委员会 J11（J11 是委员会编号，全称是 X3J11），用于起草关于 C 语言的标准草案。1989 年，ANSI 正式通过 C 语言标准草案，至此该标准成为美国国家标准，此标准也称为 C89 标准。

随后，《The C Programming Language》第 2 版出版发行，书中内容根据 ANSI C（C89）进行了更新。1990 年，在 ISO/IEC JTC1/SC22/WG14（即 ISO/IEC 联合技术第 I 委员会第 22 分委员会第 14 工作组）的努力下，ISO 批准 ANSI C 成为国际标准，于是 ISO C（又称为 C90）诞生。与 C89 相比，C90 除了标准文档的印刷编排细节有些不同外（主要表现在删除了“Rationale”一节，并把文档的格式与段落编码作了改动），它们在技术上是完全一样的。到目前为止，C89 是 C 语言运用得最广泛的标准，基本上所有的 C 语言编译器都完全支持该标准。相对于“K&R C”，C89 主要做了以下几方面的改进：

- ❑ 增加了新特性——原型。原型是函数声明的扩展，它使得编译器很容易根据函数的定义检查函数的用法。
- ❑ 增加了一些新的关键字，如 enum、const、volatile、signed 与 void。C89 的关键字见表 1-1。

表 1-1 C89 关键字表

auto	double	int	struct	break	else	long	switch
case	enum	register	typedef	char	extern	return	union
const	float	short	unsigned	continue	for	signed	void
default	goto	sizeof	volatile	do	if	static	while

❑ 除此之外，C89 还做了许多其他的改进，如增强了预处理指令，定义了相关的宏，允许将结构本身作为参数传递给函数，从“无符号保留”转到“值保留”等。

自 ISO C (C90) 推出之后，ISO 又于 1994 年与 1996 年分别出版了 C90 的技术勘误文档，更正了一些印刷错误，同时，在 1995 年还通过了一份 C90 的技术补充，这份补充对 C90 进行了微小扩充，扩充后的 ISO C 被称为 C95。

1999 年，ANSI 和 ISO 又通过了最新版本的 C 语言标准和技术勘误文档，该标准被称为 C99。这里需要说明的是，与 C89 不同，并非市面上所有的编译器都支持 C99，并且有的编译器只支持 C99 的部分新特性。相对于 C89，C99 主要做了以下几方面的改进：

- ❑ 增加了 restrict 与 inline 关键字。
- ❑ 新增 _Bool、_Complex 与 _Imaginary 3 种数据类型，如 C99 中定义的复数类型为：float_Complex、float_Imaginary、double_Complex、double_Imaginary、long double_Complex 与 long double_Imaginary。
- ❑ 增强数组的功能，支持可变长数组等。
- ❑ 支持复合赋值。
- ❑ 增强预处理程序，如引入 _Pragma 运算符，并增加了一些内部宏等。
- ❑ 支持柔性数组结构成员，即允许结构中的最后一个元素是未知大小的数组。

由于技术的发展日新月异，因此虽然 C99 还没有得到完全支持，但在 2007 年，标准委员会就又开始起草新的 C 语言标准来取代现有的 C99 标准，该标准命名为 C1X，C1X 是一个非正式名字。2011 年 12 月，ANSI 正式采纳了 ISO/IEC 9899:2011 标准，即 C11 标准。相对于 C99，C11 主要做了如下几方面的改进：

- ❑ 采用新的对齐规范，包括 _Alignas 说明符、_Alignof 运算符、aligned_alloc 函数与 <stdalign.h> 头文件。
- ❑ 增加 _Noreturn 函数标记。
- ❑ 增加 _Generic 关键词。
- ❑ 增加静态断言 _Static_assert()。
- ❑ 删除 gets() 函数，C99 中已经将此函数标记为过时，推荐新的替代函数 gets_s()。
- ❑ 采用新的 fopen() 模式。
- ❑ 增加匿名结构体 / 联合体。
- ❑ 支持多线程技术，包括 _Thread_local 与头文件 <threads.h>。
- ❑ 增加 _Atomic 类型修饰符和头文件 <stdatomic.h>。

- ❑ 带边界检查 (bounds-checking) 的函数接口，定义了新的安全的函数，例如 `fopen_s()`、`strcat_s()` 等。
- ❑ 改进 Unicode 支持与头文件 `<uchar.h>`。
- ❑ 增加 `quick_exit()` 函数作为第三种终止程序的方式。
- ❑ 可以创建复数的宏。
- ❑ 增加更多处理浮点数的宏。
- ❑ `struct timespec` 成为 `time.h` 的一部分，以及宏 `TIME_UTC` 和函数 `timespec_get()`。

综上所述，可以用图 1-2 来直观地阐述 C 语言标准的发展历程。

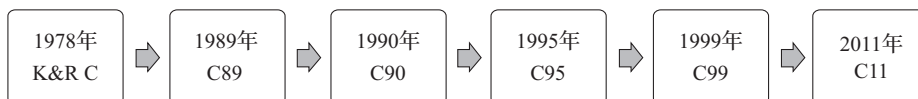


图 1-2 C 语言标准的发展过程

在 GCC 编译器中，针对不同版本的 C 语言标准，可以通过在命令行中使用“-std”选项来选择所需要使用的 C 语言标准版本。

1) C89 或者 C90

```
-ansi
-std=c90
-std=iso9899:1990
```

2) C95

```
-std=iso9899:199409
```

3) C99

```
-std=c99
-std=iso9899:1999
```

4) C11

```
-std=c11
-std=iso9899:2011
```

5) 除此之外，如果需要在 GCC 中使用 C 扩展，还可以通过如下参数形式实现：

```
C89 或者 C90: -std=gnu90
C99: -std=gnu99
C11: -std=gnu11
```

建议 2：防止整数类型产生回绕与溢出

到 C99 为止，C 语言为我们提供了 12 个相关的数据类型关键字来表达各种数据类型。

如表 1-2 所示，K&R C 提供了 7 个，C89/C90 新增了 2 个，C99 新增了 3 个。

表 1-2 C 的数据类型关键字

K&R C 的关键字	C89/C90 关键字 (新增)	C99 关键字 (新增)
int	signed	_Bool
long	void	_Complex
short		_Imaginary
unsigned		
char		
float		
double		

整型是 C 语言最基本的数据类型，它以二进制编码的方式进行存储，具体可以包括字符、短整型、整型和长整型等。例如，整数 2 的二进制表示为 10，它在 8 位与 32 位的操作系统中存储方式如图 1-3 所示。

8位存储方式:	0	0	0	0	0	0	1	0
32位存储方式:	00000000	00000000	00000000	00000010				

图 1-3 整数 2 的二进制编码存储方式

虽然在计算机中整数是以二进制编码方式进行存储的，但为了便于表达，有时候又会用十六进制编码方式表示（例如，在 32 位操作系统下，整数 2 的十六进制编码方式为 0x00000002），二进制和十六进制之间能够很方便地进行转换。

与此同时，整数类型又可分为有符号（signed）和无符号（unsigned）两种类型，limits.h 文件定义了整型数据类型的表达值范围。在 GCC 4.8.3 中，limits.h 文件定义如下：

```

/*
 * ISO C99 Standard: 7.10/5.2.4.2.1 Sizes of integer types <limits.h>
 */

#ifndef _LIBC_LIMITS_H_
#define _LIBC_LIMITS_H_ 1

#include <features.h>

/* Maximum length of any multibyte character in any locale.
   We define this value here since the gcc header does not define
   the correct value. */
#define MB_LEN_MAX 16

/* If we are not using GNU CC we have to define all the symbols ourself.

```


8 编写高质量代码：改善 C 程序代码的 125 个建议

```
Otherwise use gcc's definitions (see below). */
#if !defined __GNUC__ || __GNUC__ < 2

/* We only protect from multiple inclusion here, because all the other
   #include's protect themselves, and in GCC 2 we may #include_next through
   multiple copies of this file before we get to GCC's. */
# ifndef _LIMITS_H
#  define _LIMITS_H 1

#include <bits/wordsize.h>

/* We don't have #include_next.
   Define ANSI <limits.h> for standard 32-bit words. */

/* These assume 8-bit `char's, 16-bit `short int's,
   and 32-bit `int's and `long int's. */

/* Number of bits in a `char'. */
#  define CHAR_BIT 8

/* Minimum and maximum values a `signed char' can hold. */
#  define SCHAR_MIN      (-128)
#  define SCHAR_MAX      127

/* Maximum value an `unsigned char' can hold. (Minimum is 0.) */
#  define UCHAR_MAX      255

/* Minimum and maximum values a `char' can hold. */
#  ifdef __CHAR_UNSIGNED__
#    define CHAR_MIN 0
#    define CHAR_MAX UCHAR_MAX
#  else
#    define CHAR_MIN SCHAR_MIN
#    define CHAR_MAX SCHAR_MAX
#  endif

/* Minimum and maximum values a `signed short int' can hold. */
#  define SHRT_MIN      (-32768)
#  define SHRT_MAX      32767

/* Maximum value an `unsigned short int' can hold. (Minimum is 0.) */
#  define USHRT_MAX      65535

/* Minimum and maximum values a `signed int' can hold. */
#  define INT_MIN (-INT_MAX - 1)
#  define INT_MAX 2147483647

/* Maximum value an `unsigned int' can hold. (Minimum is 0.) */
#  define UINT_MAX      4294967295U
```

```

/* Minimum and maximum values a 'signed long int' can hold. */
# if __WORDSIZE == 64
#   define LONG_MAX      9223372036854775807L
#   else
#   define LONG_MAX      2147483647L
#   endif
#   define LONG_MIN      (-LONG_MAX - 1L)

/* Maximum value an 'unsigned long int' can hold. (Minimum is 0.) */
# if __WORDSIZE == 64
#   define ULONG_MAX     18446744073709551615UL
#   else
#   define ULONG_MAX     4294967295UL
#   endif

#   ifdef __USE_ISOC99

/* Minimum and maximum values a 'signed long long int' can hold. */
#   define LLONG_MAX     9223372036854775807LL
#   define LLONG_MIN     (-LLONG_MAX - 1LL)

/* Maximum value an 'unsigned long long int' can hold. (Minimum is 0.) */
#   define ULLONG_MAX    18446744073709551615ULL

#   endif /* ISO C99 */

#   endif /* limits.h */
#   endif /* GCC 2. */

#   endif /* !_LIBC_LIMITS_H_ */

/* The <limits.h> files in some gcc versions don't define LLONG_MIN, LLONG_MAX,
   and ULLONG_MAX. Instead only the values gcc defined for ages are available. */
#   if defined __USE_ISOC99 && defined __GNUC__
#   ifndef LLONG_MIN
#   define LLONG_MIN      (-LLONG_MAX-1)
#   endif
#   ifndef LLONG_MAX
#   define LLONG_MAX      __LONG_LONG_MAX__
#   endif
#   ifndef ULLONG_MAX
#   define ULLONG_MAX     (LLONG_MAX * 2ULL + 1)
#   endif
#   endif

```

表 1-3 描述了以 ANSI 标准定义的整数类型。

表 1-3 ANSI 标准定义的整数类型

类型	位数	最小取值范围
char	8	-127 ~ 127
unsigned char	8	0 ~ 255
signed char	8	-127 ~ 127
int	16/32	-32767 ~ 32767
unsigned int	16/32	0 ~ 65535
signed int	16/32	-32767 ~ 32767
short int	16	-32767 ~ 32767
unsigned short int	16	0 ~ 65535
signed short int	16	-32767 ~ 32767
long int	32	-2147483647 ~ 2147483647
unsigned long int	32	0 ~ 4294967295
signed long int	32	-2147483647 ~ 2147483647
long long int	64	$-(2^{63}-1) \sim 2^{63}-1$
unsigned long long int	64	$2^{64}-1$

简单地讲，有符号和无符号整数间的区别在于怎样解释整数的最高位。如果定义一个有符号整数，则 C 编译程序生成的代码认为该数最高位是符号标志：符号标志为 0，则该数为正；符号标志为 1，则该数为负。

负数采用 2 的补码的形式来表示，即对原码各位求反（符号位除外），再将求反的结果加 1，最后将符号位设置为 1。例如，在 32 位操作系统中，有符号整数 -2 的存储方法如下。

第一步：取绝对值 2 的二进制编码。

00000000 00000000 00000000 00000010

第二步：求反（符号位除外）。

01111111 11111111 11111111 11111101

第三步：将求反的结果加 1。

01111111 11111111 11111111 11111110

第四步：将符号位设置为 1。

11111111 11111111 11111111 11111110

因此，有符号整数 -2 的二进制编码为 11111111 11111111 11111111 11111110，十六进制编码为 0xFFFFFFFFE。

最后还需要说明的是，当类型修饰符被自身使用时（即它不在基本类型之前时），假定其为 int 型。也就是说，表 1-4 的两种类型是等效的。

表 1-4 等效的整数类型

修饰符	等效于	修饰符	等效于
signed	signed int	long	long int
unsigned	unsigned int	short	short int

建议 2-1：char 类型变量的值应该限制在 signed char 与 unsigned char 的交集范围内

大家应该都知道，C 语言设计 char 类型的目的是存储字母和标点符号之类的字符。实际上，char 类型存储的是整数而不是字符。为了处理字符，计算机使用一种数字编码的方式来操作，如常见的 ASCII 就是用特定整数来表示特定字符的。例如，要在 ASCII 码中存储字母 B，实际上只需要存储整数 66。因此，可以使用下面的方法为 char 类型的变量赋值。

```
char c=66;
```

在 ASCII 码中，整型数据 66 在 char 类型的大小范围之内，所以这样的赋值方式是完全允许的，但不推荐使用这样的赋值方式。

这里需要注意的是，采用这样的赋值方式有个前提条件，即必须是在 ASCII 码中。有时候不同的计算机系统也会使用完全不同的编码，如一些 IBM 主机就使用一种称为 EBCDIC (Extended Binary-Coded Decimal Interchange Code，扩充的二进制编码的十进制交换码) 的编码方式。如果采用的是其他编码方式，这样的赋值方式所得到的结果就不一样了。因此，我们推荐使用字符常量的方式进行赋值，如下面的代码所示：

```
char c='B';
```

除此之外，在表 1-3 中还可以看出，默认的 char 类型可以是 signed char 类型（取值范围为 -127 ~ 127），也可以是 unsigned char 类型（取值范围为 0 ~ 255），具体取决于编译器。也就是说，不同的机器上 char 可能拥有不同范围的值。因此，为了使程序保持良好的可移植性，我们所声明的 char 类型变量的值应该限制在 signed char 与 unsigned char 的交集范围内。例如，ASCII 字符集中的字符都在这个范围内。

当然，在一个把字符当做整数值的处理程序中，可以显式地把这类变量声明为 signed char 或 unsigned char，从而确保不同的机器中在字符是否为有符号值方面保持一致，以此来提高程序的可移植性。另一方面，许多处理字符的库函数把它们的参数都声明为 char，如果我们把这些参数显式地声明为 signed char 或 unsigned char，可能会带来兼容性问题；并且有些机器处理 signed char 的效率更高些，如果硬要把它改成 unsigned char，效率很可能会因此而受损。所以把所有的 char 变量统一声明为 signed char 或 unsigned char 未必就是好的解决方案。因此，最佳的解决方案就是把 char 类型变量的值限制在 signed char 与 unsigned char 的交集范围内，这样既可以获得最大程度的可移植性，同时又不会牺牲效率。

建议 2-2：使用显式声明为 signed char 或 unsigned char 的类型来执行算术运算

在讨论本建议话题之前，我们先看看下面的这段代码的输出结果，如代码清单 1-1 所示。

代码清单 1-1 char 使用示例

```
#include <stdio.h>
int main(void)
{
    char c=150;
    int i=900;
    printf("i/c=%d\n", i/c);
    return 0;
}
```

在代码清单 1-1 中，或许大多数人都认为它输出的结果应该是“i/c= 6”，但实际的输出结果却大相径庭。前面已经讲过，char 类型的变量 c 可以有两种类型：有符号的（signed char）和无符号的（unsigned char）。这里假设 char 是 8 位的补码字符类型，那么代码清单 1-1 就可能输出“i/c=-8”（signed char）或者“i/c= 6”（unsigned char）两种结果。其中，在 Microsoft Visual Studio 2010 与 GCC 中的输出结果都是“i/c=-8”，如图 1-4 与图 1-5 所示。

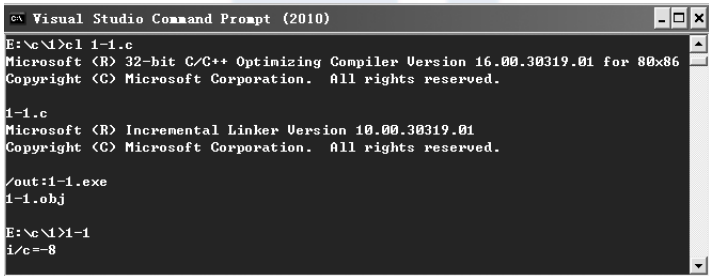


图 1-4 代码清单 1-1 在 Microsoft Visual Studio 2010 中的输出结果

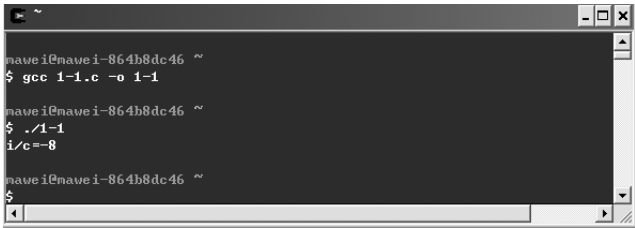


图 1-5 代码清单 1-1 在 GCC 中的输出结果

其实，导致这种结果最根本的原因就在于我们不能够准确地确定 char 类型的变量 c 究竟是 signed char 类型还是 unsigned char 类型。因此，我们把决策权交给编译器，而不同的编译器默认的 char 类型是不同的，所以最后得到的结果也就不相同。

解决这种问题的办法很简单，就是显式地将 char 类型的变量 c 声明为 signed char 或 unsigned char 类型，这样可保证结果的唯一性，如代码清单 1-2 所示。

代码清单 1-2 unsigned char 使用示例

```
#include <stdio.h>
int main(void)
{
    unsigned char c=150;
    int i=900;
    printf("i/c=%d\n", i/c);
    return 0;
}
```

这样就显式地将 char 类型的变量 c 声明为 unsigned char 类型，现在，后面的除法运算 (i/c) 与 char 的符号无关，所以代码清单 1-2 输出的结果为 “i/c= 6”。

建议 2-3: 使用 rsize_t 或 size_t 类型来表示一个对象所占用空间的整数值单位

C 语言标准规定 size_t 是一种无符号整数类型，编译器可以根据操作系统的不同而用 typedef 来定义不同的 size_t 类型，即在不同的操作系统上所定义的 size_t 可能不一样。例如在 32 位操作系统上可以将 size_t 定义为 unsigned int 类型，而在 64 位操作系统上则可以定义为 unsigned long int 类型，甚至还可以将 size_t 定义为 unsigned long long int 类型等，如下面的示例所示。

在 GCC 的 stddef.h 文件中将 size_t 定义为：

```
#ifndef __SIZE_TYPE__
#define __SIZE_TYPE__ long unsigned int
#endif
#if !(defined (__GNU__) && defined (size_t))
typedef __SIZE_TYPE__ size_t;
#endif
#ifdef __BEOS__
typedef long ssize_t;
#endif /* __BEOS__ */
```

而在 VC++ 2010 的 crtdefs.h 文件中将 size_t 定义为：

```
#ifndef _SIZE_T_DEFINED
#ifdef _WIN64
typedef unsigned __int64 size_t;
#else
typedef _W64 unsigned int size_t;
#endif
#define _SIZE_T_DEFINED
#endif
```

从上面的定义可以看出，size_t 类型的引入增强了程序在不同平台上的可移植性，而它也正是为了方便系统之间的移植而定义的。size_t 类型的变量大小足以保证存储内存中对象的大小，任何表示对象长度的变量，包括作为大小、索引、循环计数和长度的整数值，都可以声明为 size_t 类型。比如我们常用的 sizeof 操作符的结果返回的就是 size_t 类型，该类型

保证能容纳实现所建立的最大对象的字节大小。`size_t` 类型的限制是由 `SIZE_MAX` 宏指定的。

接下来看看 `size_t` 类型的使用示例，如代码清单 1-3 所示。

代码清单 1-3 `size_t` 类型的使用示例

```
char *copy(size_t n, const char *str)
{
    int i;
    char *p;
    if (n == 0)
    {
        /* 处理 n==0 的情况 */
    }
    p = (char *)malloc(n);
    if (p == NULL)
    {
        /* 处理 p==NULL 的情况 */
    }
    for (i = 0; i < n; ++i)
    {
        p[i] = *str++;
    }
    return p;
}
```

不难发现，代码清单 1-3 中存在着一个严重的问题：当 `p` 所引用的动态分配的缓冲区在 `n > INT_MAX` 时将会发生溢出。我们知道，`int` 类型的限制是由 `INT_MAX` 宏指定的，而 `size_t` 类型代表的是一个无符号整数类型，它可能包含一个大于 `INT_MAX` 的值。因此，当 `n` 的值为 `0 < n <= INT_MAX` 时，执行循环 `n` 次，代码如预期一样正常运行；但当 `n` 的值为 `INT_MAX < n <= SIZE_MAX`，且整型变量 `i` 的增值超过 `INT_MAX` 时，`i` 的值将是从小于 `INT_MIN` 开始的负值。这时，`p[i]` 所引用的内存位置是在 `p` 所引用的内存之前，这就会导致写入发生在数组边界之外。

因此，为了避免发生这种潜在性的错误，应该将变量 `i` 也声明成 `size_t` 类型，如代码清单 1-4 所示。

代码清单 1-4 代码清单 1-3 的解决方法

```
char *copy(size_t n, const char *str)
{
    size_t i;
    char *p;
    if (n == 0 || n > SIZE_MAX)
    {
        /* 处理 n==0 的情况 */
    }
    p = (char *)malloc(n);
    if (p == NULL)
```

```

{
    /* 处理 p==NULL 的情况 */
}
for ( i = 0; i < n; ++i )
{
    p[i] = *str++;
}
return p;
}

```

除了 `size_t` 类型之外，ISO/IEC TR 24731-1:2007 中引入了一种新类型 `rsize_t`，虽然它被定义为 `size_t` 类型，但它明确地表示是用于保存单个对象的长度的。

在 VC++ 2010 的 `crtdefs.h` 文件中将 `rsize_t` 定义为：

```

#if __STDC_WANT_SECURE_LIB__
#ifndef _RSIZE_T_DEFINED
typedef size_t rsize_t;
#define _RSIZE_T_DEFINED
#endif
#endif

```

在支持 `rsize_t` 类型的代码中，你可以检查对象的长度，验证它不大于 `RSIZE_MAX`（一个正常单个对象的最大长度），库函数也可以使用 `rsize_t` 进行输入校验。

在 VC++ 2010 的 `limits.h` 文件中将 `RSIZE_MAX` 定义为：

```

#if __STDC_WANT_SECURE_LIB__
#ifndef RSIZE_MAX
#define RSIZE_MAX SIZE_MAX
#endif
#endif

```

这样就消除了示例整数溢出的可能性，现在我们可以将代码清单 1-3 中的变量 `i` 声明成 `rsize_t` 类型，同时也可将参数 `n` 修改成 `rsize_t` 类型，并与 `RSIZE_MAX` 进行比较以验证数据的合法范围，如代码清单 1-5 所示。

代码清单 1-5 代码清单 1-3 的 `rsize_t` 解决方法

```

char *copy(rsize_t n, const char *str)
{
    rsize_t i;
    char *p;
    if (n == 0 || n > RSIZE_MAX)
    {
        /* 处理 n==0 || n > RSIZE_MAX 的情况 */
    }
    p = (char *)malloc(n);
    if (p == NULL)
    {

```



```

        /* 处理 p==NULL 的情况 */
    }
    for (i = 0; i < n; ++i)
    {
        p[i] = *str++;
    }
    return p;
}

```

建议 2-4：禁止把 size_t 类型和它所代表的真实类型混用

我们知道，size_t 类型代表的是一种无符号整数类型，现在有这样一个问题：既然 size_t 类型是一种无符号整数类型，那么它是否可以直接与它所代表的真实实际类型混合使用呢？带着这个问题，我们来看下面这段代码：

```

unsigned int x;
size_t y;
x = y;

```

在上面的代码中，变量 x 被声明为 unsigned int 类型（即无符号整数类型），变量 y 虽然被声明为 size_t 类型，但它同样是一种无符号整数类型。因此，从表面上看，语句“x = y”完全是可行的，但实际情况并非如此。

上面已经阐述过，size_t 类型在不同的平台上很可能代表的是 unsigned int、unsigned long int 或者 unsigned long long int 类型。当代表 unsigned int 类型时，执行语句“x = y”不会出现什么问题；但如果代表的是 unsigned long int 或 unsigned long long int 类型，那么执行语句“x = y”时，就可能把 y 的高位给截掉，从而导致结果出错。因此，我们千万不能在程序中混用 size_t 类型和它所代表的真实类型，这一点一定要注意。

建议 2-5：小心使用无符号类型带来的陷阱

有过面试经历的同学可能曾碰到如代码清单 1-6 所示的问题。

代码清单 1-6 一道典型的面试示例

```

#include <stdio.h>
int main(void)
{
    int array[] = { 1,2,3,4,5,6 };
    int i = -1;
    if ( i <= sizeof(array))
    {
        printf(" i <= sizeof(array)\n");
    }
    else
    {

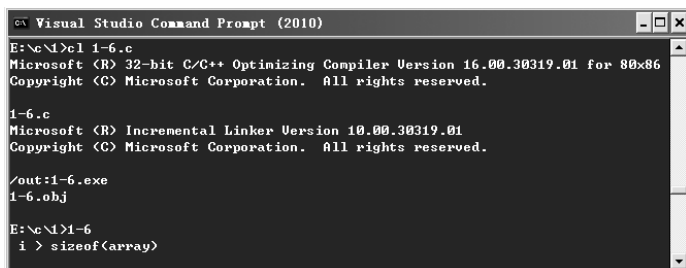
```

```

        printf(" i > sizeof(array)\n");
    }
    return 0;
}

```

对代码清单 1-6 进行初步分析可以得出，sizeof(array) 的返回结果为 24，而 i 的值为 -1，因此执行语句“if (i <= sizeof(array))”所返回的结果应该为 true，即输出结果为“i <= sizeof(array)”。但实际情况并非如此，其输出结果如图 1-6 所示。



```

Visual Studio Command Prompt (2010)
E:\c\1>cl 1-6.c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.30319.01 for 80x86
Copyright (C) Microsoft Corporation. All rights reserved.

1-6.c
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation. All rights reserved.

/out:1-6.exe
1-6.obj
E:\c\1>1-6
i > sizeof(array)

```

图 1-6 代码清单 1-6 的输出结果

那么，究竟是什么原因导致出现这样的输出结果呢？

其实，要回答这个问题并不难。我们知道 sizeof() 的返回结果是 size_t 类型，而 size_t 类型是一种无符号整数类型。当有符号整数类型和无符号整数类型进行运算时，有符号整数类型会先自动转化成无符号整数类型（请特别注意这一点）。

因此，在代码清单 1-6 中，当 i 与 sizeof(array) 进行比较时，即执行语句“if (i <= sizeof(array))”，i 会自动升级为无符号整数类型。又因为 i 的值为 -1，在它转换为无符号整数类型后就变成一个非常大的正整数（如 -1 在 32 位机器上存储为 0xffffffff，而它被解释为无符号整数时就是 $2^{32}-1$ ，即 4294967295），远远大于 sizeof(array) 的返回结果 24。

为了加深读者的理解，我们再来看代码清单 1-7 所示的这个例子。

代码清单 1-7 无符号类型运算示例

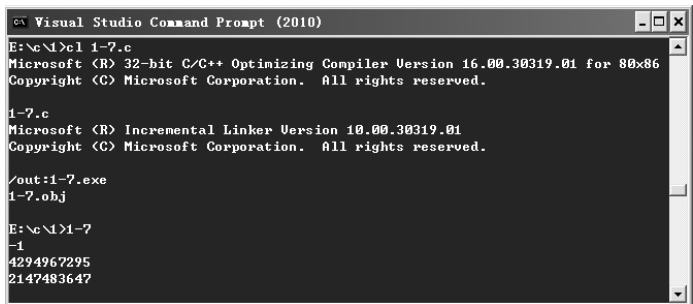
```

#include<stdio.h>
int main(void)
{
    int a = 3;
    unsigned int b = 4;
    printf("%d\n", a - b);
    printf("%u\n", a - b);
    printf("%d\n", (a - b) >> 1);
    return 0;
}

```

在代码清单 1-7 中，当执行语句“a-b”时，变量 a 会自动由 int 类型转换为 unsigned int 类型，再与变量 b 执行减法运算（即“a-b”），“a-b”的运算结果为 0xffffffff。当程序使

用 “%d”（有符号十进制整数）格式输出时，0xffffffff 被转换为 -1；当程序使用 “%u”（无符号十进制整数）格式输出时，0xffffffff 被转换为 4294967295；最后，程序执行 “0xffffffff >> 1” 运算时，其运算结果为 0x7fffffff。当程序使用 “%d”（有符号十进制整数）格式输出时，0x7fffffff 被转换为 2147483647。代码清单 1-7 的输出结果如图 1-7 所示。



```
Visual Studio Command Prompt (2010)
E:\c>cl 1-7.c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.30319.01 for 80x86
Copyright (C) Microsoft Corporation. All rights reserved.

1-7.c
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation. All rights reserved.

/out:1-7.exe
1-7.obj
E:\c>1-7
1
4294967295
2147483647
```

图 1-7 代码清单 1-7 的输出结果

由上面两个例子可以看出，将有符号类型与无符号类型混合使用是很危险的。因此，我们一定要小心这个数据转换陷阱，尽量少在代码中使用无符号类型，以免增加不必要的复杂性。尤其是不要仅仅因为无符号数不存在负值而用它来表示某些数量（如年龄、人口等无负数的值）。建议尽量使用像 int 这样的类型，这样在设计升级混合类型的复杂细节时，就不必担心边界情况了（比如不用担心 -1 被翻译为非常大的正整数）。如果必须使用无符号类型，则应该在表达式中使用强制类型转换，使操作数均为有符号类型或者无符号类型，这样就不必由编译器来选择结果的类型，从而避免存在潜在错误的可能性。比如，我们可以通过强制转换将代码清单 1-6 改写为代码清单 1-8。

代码清单 1-8 代码清单 1-6 的解决方法

```
#include <stdio.h>
int main(void)
{
    int array[] = { 1,2,3,4,5,6 };
    int i = -1;
    if ( i <= (int)sizeof(array))
    {
        printf(" i <= sizeof(array)\n");
    }
    else
    {
        printf(" i > sizeof(array)\n");
    }
    return 0;
}
```

在代码清单 1-8 中，我们将语句 “if(i <= sizeof(array))” 改成 “if(i <= (int)sizeof(array))”，

也就是通过强制类型将其转换成 int 类型，因而程序的输出结果为 `i <= sizeof(array)`。

建议 2-6：防止无符号整数回绕

C99 第 6.2.5 节的第 9 条规定是：涉及无符号操作数的计算永远不会产生溢出，因为无法由最终的无符号整型表示的结果将会根据这种最终类型可以表示的最大值加 1 执行求模操作。也就是说，如果数值超过无符号整型数据的限定长度时就会发生回绕，即如果无符号整型变量的值超过了无符号整型的上限，就会返回 0，然后又从 0 开始增大；如果无符号整型变量的值低于无符号整型的下限，那么就会到达无符号整型的上限，然后从上限开始减小。这就像一个人绕着跑道跑步一样，绕了一圈，又返回到出发点，因此称为回绕。

为了加深大家对无符号整数运算产生回绕的理解，我们继续来看代码清单 1-9 所示的一个简单例子。

代码清单 1-9 无符号整数运算示例

```
#include <stdio.h>
int main(void)
{
    unsigned int a = 4294967295;
    unsigned int b = 2;
    unsigned int c=4;
    printf("%u\n", a + b);
    printf("%u\n", b -c);
    return 0;
}
```

在代码清单 1-9 中，我们定义了 3 个无符号整型变量 a、b 与 c。其中将变量 a 的值初始化为 4294967295（即在 32 位机器上存储为 0xffffffff）。当程序执行语句“a+b”时，其结果超出了无符号整型的限定值（UINT_MAX：0xffffffff），于是便产生向下回绕，因此输出的结果为 1（即 0xffffffff+0x00000002=0x00000001）；当程序执行语句“b-c”时，其结果为负数，于是便产生向上回绕，因此返回的结果为 4294967294（即 0x00000002-0x00000004=0xffffffe）。具体运行结果如图 1-8 所示。

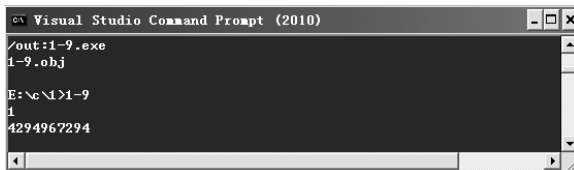


图 1-8 代码清单 1-9 的输出结果

从代码清单 1-9 中可以看出，无符号整数运算产生的回绕会给程序带来严重的后果，尤其是作为数组索引、指针运算、对象的长度或大小、循环计数器与内存分配函数的实参等的

时候是绝对不允许产生回绕的。因此，针对无符号整数的运算，应该采用适当的方法来防止产生回绕。例如，代码清单 1-10 演示了如何简单地处理代码清单 1-9 中所产生的回绕。

代码清单 1-10 代码清单 1-9 的解决方法

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    unsigned int a = 4294967295;
    unsigned int b = 2;
    unsigned int c=4;
    if(UINT_MAX-a<b)
    {
        /* 处理错误条件 */
    }
    else
    {
        printf("%u\n", a + b);
    }
    if(b<c)
    {
        /* 处理错误条件 */
    }
    else
    {
        printf("%u\n", b -c);
    }
    return 0;
}
```

在上面的代码中，通过一些条件对无符号操作数进行测试，从而避免了无符号操作数运算产生回绕。在实际的编程环境中，无符号整数的回绕很可能会导致缓冲区溢出，甚至导致攻击者可执行任意代码。例如，程序绕过代码中的大小判断部分的边界检测，可以导致缓冲区溢出，只要使用一般的技术就能够利用这个溢出程序。演示示例如代码清单 1-11 所示。

代码清单 1-11 回绕导致的溢出示例

```
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[])
{
    unsigned short s;
    int i;
    char buf[100];
    if(argc < 3)
    {
        return -1;
    }
}
```

```

i = atoi(argv[1]);
s = i;
if(s >= 100)
{
    printf("拷贝字节数太大, 请退出!\n");
    return -1;
}
printf("s = %d\n", s);
memcpy(buf, argv[2], i);
buf[i] = '\0';
printf("成功拷贝 %d 个字节\n", i);
printf("buf=%s\n", buf);
return 0;
}

```

在代码清单 1-11 中，程序需要将 argv[2] 的内容复制到 buf 中，并由 argv[1] 指定复制的字节数。这里需要特别注意的语句是“if(s >= 100)”，利用该语句进行了相对严格的大小检查：如果 argv[1] 的值大于等于 buf 数组的大小（100），则不进行复制。

运行代码清单 1-11，当我们执行命令“1-11 4 mawei”时，程序运行正常，并成功地复制了字符串“mawe”到 buf 中，运行结果如图 1-9 所示。

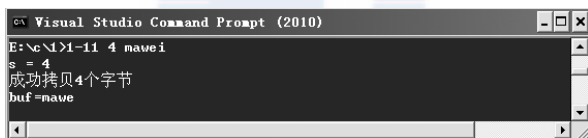


图 1-9 代码清单 1-11（执行“1-11 4 mawei”）的运行结果

当我们执行命令“1-11 200 mawei”时，程序同样运行正常，运行结果如图 1-10 所示。



图 1-10 代码清单 1-11（执行“1-11 200 mawei”）的运行结果

可当我们执行命令“1-11 65536 mawei”时，程序却意外地绕过了大小检查语句“if(s >= 100)”来执行相关的操作。原因很简单，程序从命令行参数中得到一个整数值并存放在整型变量 i 中，然后这个值被赋予了 unsigned short 类型的整数 s，由于 s 在内存中是用 16 位进行存储的，而 16 位能够存储的最大十进制数是 65535（即 unsigned short 存储的范围是 0 ~ 65535），如果这个值不在 unsigned short 类型的存储范围内（0 ~ 65535），就会产生回绕。因此，当我们输入 65536 时，系统将会转换为 0，从而绕过大小检查语句“if(s >= 100)”来执行余下的操作。可是这里我们将 buf 数组的大小初始化为 100，所以在执行语句

“memcpy(buf, argv[2], i)”时，程序就会产生异常而导致崩溃。其运行结果如图 1-11 与图 1-12 所示。



图 1-11 代码清单 1-11（执行“1-11 65536 mawei”）的运行结果

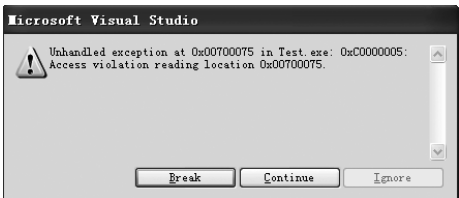


图 1-12 代码清单 1-11（执行“1-11 65536 mawei”）导致程序崩溃

其实，这类 Bug 很常见，而且很容易被攻击，这都是由于无符号整数发生回绕导致的。由于存在回绕，当一个有符号整数被解释成一个无符号整数时，它可能变得很大。比如，-1 被当成无符号数时将会是十进制的 4294967295，它是 32 位整数的最大值。如果我们加入的这个值被用作 memcpy 的参数，memcpy 就会试图复制 4GB 数据，很明显这可能导致错误或破坏堆栈。

除此之外，无符号整数的回绕最可能被利用的情况之一就是利用计算结果来决定将要分配的缓冲区的大小。通常情况下，在程序需要为一组对象分配内存空间时，会将对象的个数乘以单个对象大小，然后用所乘结果来作为参数，从而调用 malloc() 或 calloc() 函数来分配内存。这时候，只要我们能够控制对象的个数或单个对象的大小，就有可能让程序分配错误大小的缓冲区。演示示例如代码清单 1-12 所示。

代码清单 1-12 回绕导致的错误分配缓冲区示例

```
#include <stdio.h>
#include <stdlib.h>
int* copyarray(int *arr, int len);
int main(int argc, char *argv[])
{
    int arr[] = {1,2,3,4,5};
    copyarray(arr,atoi(argv[1]));
    return 0;
}
int* copyarray(int *arr, int len)
{
    int i=0;
```

```

int *newarray = (int *)malloc(len*sizeof(int));
if(newarray == NULL)
{
    /* 处理 newarray == NULL 的情况 */
}
printf(" 为 newarray 成功分配 %d 字节内存 \n", len*sizeof(int));
printf(" 循环运行次数: %d(0x%x)\n", len, len);
for(i = 0; i < len; i++)
{
    newarray[i] = arr[i];
}
return newarray;
}

```

在代码清单 1-12 中，函数 “int* copyarray(int *arr, int len)” 需要将 arr 的内容复制到 newarray 中，对象的个数由 len 参数来指定。其中，程序使用了对对象的个数乘以单个对象大小的乘积来作为 malloc() 函数的参数，从而对 newarray 进行内存分配，即内存分配语句为 “int *newarray = (int *)malloc(len*sizeof(int))”。

运行代码清单 1-12，当我们执行命令 “1-12 8” 时，程序运行正常，并成功地 newarray 分配了内存，并将 arr 的内容复制到 newarray 中，运行结果如图 1-13 所示。

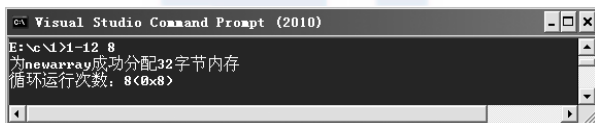


图 1-13 代码清单 1-12 (执行 “1-12 8”) 的运行结果

这样看来，程序貌似没有任何问题。但是当我们执行命令 “1-12 1073741824” 时，问题就出现了，抛出异常 “Unhandled exception at 0x004010d2 in 1-12.exe: 0xC0000005: Access violation writing location 0x00387000。”。运行结果如图 1-14 所示。

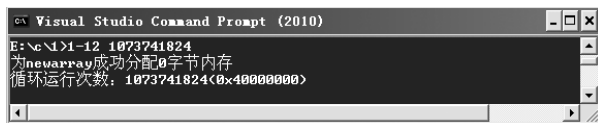


图 1-14 代码清单 1-12 (执行 “1-12 1073741824”) 的运行结果

是什么原因导致这样的结果呢？

其实很简单，是因为函数 “int* copyarray(int *arr, int len)” 没有检查参数 len 而导致运算回绕失败。在通过语句 “int *newarray =(int *) malloc(len*sizeof(int))” 给 newarray 分配内存时，这里将参数设置为 1073741824 (十六进制是 0x40000000)，而 “sizeof(int)” 的返回结果为 4 (十六进制是 0x4)。当运算表达式 “0x40000000*0x4” 时，就发生了无符号整数运算回绕，所得的结果为 0x0 (即 0x40000000*0x4=0x0)。因此，为 newarray 分配的内存为 0。

除此之外，在通过语句“`int *newarray =(int *) malloc(len*sizeof(int))`”给 `newarray` 分配内存时，由于参数 `len` 的原因而造成运算回绕，所以我们可以利用它来分配一个任意长度的缓冲区。如上面将 `len` 参数设置为 `1073741824`，就可能出现在没有为 `newarray` 分配内存的情况下，却向其中复制了数组元素，而且循环的次数还非常多，严重时会造成系统崩溃。当然，你还可以通过选择合适的值赋给 `len` 参数以使得循环反复执行导致缓冲区溢出。同时，还可以通过覆盖 `malloc` 的控制结构来执行任意恶意代码，从而实施对堆溢出的攻击。

在本节的最后，还需要说明的是，并不是每种运算符号都会令无符号操作数运算产生回绕，表 1-5 给出了可能会导致回绕的操作符。

表 1-5 可能导致回绕的操作符

操作符	是否回绕	操作符	是否回绕	操作符	是否回绕	操作符	是否回绕
+	是	--	是	<<	是	<	否
-	是	*=	是	>>	否	>	否
*	是	/=	否	&	否	>=	否
/	否	%=	否		否	<=	否
%	否	<<=	是	^	否	==	否
++	是	>>=	否	~	否	!=	否
--	是	&=	否	!	否	&&	否
=	否	=	否	单目 +	否		否
+=	是	^=	否	单目 -	是	?:	否

建议 2-7：防止有符号整数溢出

整数溢出是一种常见、难预测且严重的软件漏洞，由它引发的程序 Bug 可能比格式化字符串与缓冲区溢出等缺陷更难于发现。C99 标准中规定，当两个操作数都是有符号整数时，就有可能发生整数溢出，它将会导致“不能确定的行为”。也就是说整数溢出是一种未定义的行为，这也就意味着编译器在处理有符号整数的溢出时具有很多的选择，遵循标准的编译器可以做它们想做的任何事，比如完全忽略该溢出或终止进程。大多数编译器都会忽略这种溢出，这可能会导致不确定的值或错误的值保存在整数变量中。

整数溢出有时候是很难发现的，一般情况下在整数溢出发生之前，你都无法知道它是否会发生溢出，即使你的代码经过仔细审查，有时候溢出也是不可避免的。因此，程序很难区分先前计算出的结果是否正确，而且如果计算结果将作为一个缓冲区的大小、数组的下标、循环计数器与内存分配函数的实参等时将会非常危险。当然，因为无法直接改写内存单元，所以大多数整数溢出是没有办法利用的。但是，有时候整数溢出将会导致其他类型的缺陷发生，比如很容易发生的缓冲区溢出等。代码清单 1-13 是一个简单的整数溢出示例。

代码清单 1-13 整数溢出示例

```
#include <stdio.h>
int main(void)
```

```

{
    int s1 = 2147483647;
    int s2 = 1073741824;
    int s3 = -1879048193;
    int s4=1;
    int s5=4;
    printf("%d(0x%x)+%d(0x%x)=%d(0x%x)\n", s1, s1, s4, s4,
           s1+s4, s1+s4);
    printf("%d(0x%x)-%d(0x%x)=%d(0x%x)\n", s2, s2, s3, s3,
           s2-s3, s2-s3);
    printf("%d(0x%x)*%d(0x%x)=%d(0x%x)\n", s2, s2, s5, s5,
           s2*s5, s2*s5);
    return 0;
}

```

在 32 位操作系统中，类型 `int` 的取值范围为“-2147483647 ~ 2147483647”，限制是由 `INT_MIN` 与 `INT_MAX` 宏指定的，如下面的代码所示：

```

#define INT_MIN      (-2147483647 - 1)
#define INT_MAX      2147483647

```

而在代码清单 1-13 中，当程序执行语句“`s1+s4`、`s2-s3` 与 `s2*s5`”时，其结果都超过类型 `int` 的取值范围，因此发生溢出行为，运行结果如图 1-15 所示。

```

E:\c\1>1-13
2147483647(0x7fffffff)+1(0x1)=-2147483648(0x80000000)
1073741824(0x40000000)--1879048193(0x8fffffff)--1342177279(0xb0000001)
1073741824(0x40000000)*4(0x4)=0(0x0)

```

图 1-15 代码清单 1-13 的运行结果

当然，面对这些简单的有符号整数运算溢出，简单地通过对操作数进行预测的方法就能够避免发生有符号整数运算溢出。比如，代码清单 1-14 就采用了补码的表示形式来对操作数进行预测。

代码清单 1-14 采用补码的表示形式来对操作数进行预测

```

#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int s1 = 2147483647;
    int s2 = 1073741824;
    int s3 = -1879048193;
    int s4=1;
    if(((s1^s4)|(((s1^(~(s1^s4)
        &(1<<(sizeof(int)*CHAR_BIT-1))))+s4)^s4))>=0)
    {

```

```

        /* 处理溢出条件 */
    }
    else
    {
        printf("%d(0x%x)+%d(0x%x)=%d(0x%x)\n", s1, s1,s4,s4,
            s1+s4, s1+s4);
    }
    if(((s2^s3)&(((s2^((s2^s3)
        &(1<<(sizeof(int)*CHAR_BIT-1))))-s3)^s3))<0)
    {
        /* 处理溢出条件 */
    }
    else
    {
        printf("%d(0x%x)-%d(0x%x)=%d(0x%x)\n", s2, s2, s3, s3,
            s2-s3, s2-s3);
    }
    return 0;
}

```

如上面的代码所示，这种方式可以有效地避免发生简单的有符号整数运算溢出，有兴趣的朋友可以自己测试。其实，不只算术运算可能造成溢出，任何企图改变该有符号整型变量值的操作都可能造成溢出。示例如代码清单 1-15 所示。

代码清单 1-15 溢出示例

```

#include <stdio.h>
int main(void)
{
    int si1= 1073741824;
    int si2=0;
    int si3= -1073741824;
    int si4=4;
    int si5=-1;
    printf("si1 = %d (0x%x)\n", si1, si1);
    si2 = si1 + si3;
    printf("si1 + %d(0x%x) = %d (0x%x)\n",si3,si3, si2, si2);
    si2 = si1 * si4;
    printf("si1 * %d(0x%x) = %d (0x%x)\n",si4,si4, si2, si2);
    si2 = si1 - si5;
    printf("si1 - %d(0x%x) = %d (0x%x)\n",si5,si5, si2, si2);
    return 0;
}

```

代码清单 1-15 的运行结果如图 1-16 所示。

与无符号整数的回绕相似，并不是每种运算符号都会令有符号操作数运算产生溢出，表 1-6 给出了可能会导致溢出的操作符。

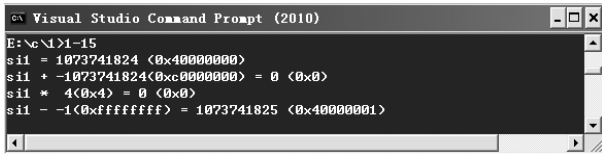


图 1-16 代码清单 1-15 的运行结果

表 1-6 可能导致溢出的操作符

操作符	是否溢出	操作符	是否溢出	操作符	是否溢出	操作符	是否溢出
+	是	--	是	<<	是	<	否
-	是	*=	是	>>	否	>	否
*	是	/=	是	&	否	>=	否
/	是	%=	是		否	<=	否
%	是	<<=	是	^	否	==	否
++	是	>>=	否	~	否	!=	否
--	是	&=	否	!	否	&&	否
=	否	=	否	单目 +	否		否
+=	是	^=	否	单目 -	是	?:	否

与前面所讲的无符号整数回绕一样，有符号整数的这种溢出也很容易导致缓冲区溢出，同时也很容易让攻击者可执行任意代码，演示示例如代码清单 1-16 所示。

代码清单 1-16 溢出导致的结果示例


```

#include <stdio.h>
#include <stdlib.h>
int copychar(char *c1,int len1, char *c2, int len2);
int main(int argc, char *argv[])
{
    copychar(argv[1],atoi(argv[2]),argv[3],atoi(argv[4]));
    return 0;
}
int copychar(char *c1,int len1, char *c2, int len2)
{
    char buf[100];
    if((len1 + len2) > 100)
    {
        printf(" 超出 buf 容纳范围 (100)!\n");
        return -1;
    }
    memcpy(buf, c1, len1);
    memcpy(buf+len1, c2, len2);
    printf(" 复制 %d+%d=%d 个字节到 buf!\n",len1,len2,len1+len2);
    printf("buf=%s\n", buf);
    return 0;
}

```

在代码清单 1-16 中，程序需要将 c1 与 c2 的内容复制到 buf 中，并分别由 len1 与 len2 来指定复制的字节数。这里需要特别注意的语句是“if(len1 + len2) > 100)”，我们利用该语句进行了相对严格的大小检查：如果 len1 + len2 的值大于 buf 数组的大小（100），则不进行复制。

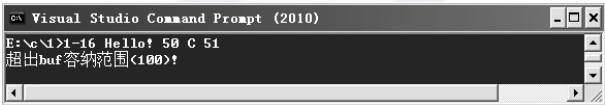
运行代码清单 1-16，当我们执行命令“1-16 Hello! 6 C 2”时，程序运行正常，并成功地将字符串复制到 buf 中，运行结果如图 1-17 所示。



```
Visual Studio Command Prompt (2010)
E:\c>1-16 Hello! 6 C 2
复制6+2=8个字节到buf!
buf=Hello!C
```

图 1-17 代码清单 1-16（执行“1-16 Hello! 6 C 2”）的运行结果

当我们执行命令“1-16 Hello! 50 C 51”时，程序同样运行正常，运行结果如图 1-18 所示。



```
Visual Studio Command Prompt (2010)
E:\c>1-16 Hello! 50 C 51
超出buf容纳范围(100)!
```

图 1-18 代码清单 1-16（执行“1-16 Hello! 50 C 51”）的运行结果

可当我们执行命令“1-16 Hello! 2147483647 C 2”时，程序却意外地绕过大小检查语句“if(len1 + len2) > 100)”来执行相关的操作。是什么原因导致这种情况发生的呢？

其实很简单，就是由于整数溢出而导致的。从执行的命令“1-16 Hello! 2147483647 C 2”可以得出，len1 的值为 2147483647（即十六进制为 0x7fffffff），len2 值为 2（即十六进制为 0x00000002），当执行语句“len1 + len2（即 0x7fffffff+0x00000002）”时会发生溢出，所得结果为 -2147483647（即十六进制为 0x80000001）。因为 -2147483647 远远小于 100，从而使程序绕过大小检查语句“if(len1 + len2) > 100)”来执行余下的操作。也正因为如此，在执行语句“memcpy(buf, c1, len1)”时便导致异常“Unhandled exception at 0x65726f66 in 1-16.exe: 0xC0000005: Access violation reading location 0x65726f66”的发生。

建议 3：尽量少使用浮点类型

C 语言标准规定的浮点数据类型有 float、double、long double 三种，如表 1-7 所示。

表 1-7 ANSI 标准定义的浮点数据类型

类型	位数	最小取值范围
float	32	6 位精度，1E-37 ~ 1E+37
double	64	10 位精度，1E-37 ~ 1E+37
long double	80	10 位精度，1E-37 ~ 1E+37

和整型一样, 浮点数据类型既没有规定每种类型占多少字节, 也没有规定采用哪种表示形式。因此, 浮点数据类型的实现在各种平台上差异很大, 有的处理器有浮点运算单元 (Floating Point Unit, FPU), 称为硬浮点 (Hard-float) 实现; 而有的处理器没有浮点运算单元, 只能做整数运算, 也就是需要用整数运算来模拟浮点运算, 这种实现方式称为软浮点 (Soft-float) 实现。迄今为止, 大部分平台的浮点数实现都遵循 IEEE 754 标准 (IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985)。

这里需要特别说明的是, ANSI C 并未规定 long double 类型的准确精度。正因为如此, 对于不同的平台, long double 类型可能有不同的实现, 有的是 8 字节, 有的是 10 字节, 还有的是 12 字节或更多。在 x86 平台上, 大多数编译器实现的 long double 类型是 80 位, 因为 x86 的浮点运算单元具有 80 位精度。如在 VC++ 2010 中运行 “sizeof(long double)” 所得的结果为 8, 而在 GCC 中运行 “sizeof(long double)” 所得的结果则为 12 (即 96 位)。但一般来说, long double 类型的精度要高于 double 类型, 至少它们也应该相等。

建议 3-1: 了解 IEEE 754 浮点数

1. 浮点数简介

在计算机系统的发展过程中, 业界曾经提出过许多种实数的表达方法, 比较典型的有相对于浮点数 (Floating Point Number) 的定点数 (Fixed Point Number)。在定点数表达法中, 其小数点固定地位于实数所有数字中间的某个位置。例如, 货币的表达就可以采用这种表达方式, 如 55.00 或者 00.55 可以用于表达具有 4 位精度, 小数点后有两位的货币值。由于小数点位置固定, 所以可以直接用 4 位数值来表达相应的数值。但我们不难发现, 定点数表达法的缺点就在于其形式过于僵硬, 固定的小数点位置决定了固定位数的整数部分和小数部分, 不利于同时表达特别大的数或者特别小的数。因此, 最终绝大多数现代的计算机系统都采纳了所谓的浮点数表达法。

浮点数表达法采用了科学计数法来表达实数, 即用一个有效数字[⊖]、一个基数 (Base)、一个指数 (Exponent) 以及一个表示正负的符号来表达实数。比如, 666.66 用十进制科学计数法可以表达为 6.6666×10^2 (其中, 6.6666 为有效数字, 10 为基数, 2 为指数)。浮点数利用指数达到了浮动小数点的效果, 从而可以灵活地表达更大范围的实数。

当然, 对实数的浮点表示仅作如上的规定是不够的, 因为同一实数的浮点表示还不是唯一的。例如, 上面例子中的 666.66 可以表达为 0.66666×10^3 、 6.6666×10^2 或者 66.666×10^1 三种方式。因为这种表达的多样性, 因此有必要对其加以规范化以达到统一表达的目标。规范的浮点数表达方式具有如下形式:

$$\pm d.dd\dots d \times \beta^e \quad (0 \leq d_i < \beta)$$

其中, $d.dd\dots d$ 为有效数字, β 为基数, e 为指数。

[⊖] Significand, 有效数字也被称为尾数 (Mantissa)。

有效数字中数字的个数称为精度，我们可以用 p 来表示，即可称为 p 位有效数字精度。每个数字 d 介于 0 和基数 β 之间，包括 0。更精确地说， $\pm d_0.d_1d_2\cdots d_{p-1} \times \beta^e$ 表示以下数：

$$\pm (d_0 + d_1\beta^{-1} + \cdots + d_{p-1}\beta^{-(p-1)})\beta^e \quad (0 \leq d_i < \beta)$$

其中，对十进制的浮点数，即基数 β 等于 10 的浮点数而言，上面的表达式非常容易理解。如 12.34，我们可以根据上面的表达式表达为： $1 \times 10^1 + 2 \times 10^0 + 3 \times 10^{-1} + 4 \times 10^{-2}$ ，其规范浮点数表达为 1.234×10^1 。

但对二进制来说，上面的表达式同样可以简单地表达。唯一不同之处在于：二进制的 β 等于 2，而每个数字 d 只能在 0 和 1 之间取值。如二进制数 1001.101，我们可以根据上面的表达式表达为： $1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}$ ，其规范浮点数表达为 1.001101×2^3 。

现在，我们就可以这样简单地把二进制转换为十进制，如二进制数 1001.101 转换成十进制为：

$$\begin{aligned} &1001.101 \\ &= 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} \\ &= 8 + 0 + 0 + 1 + \frac{1}{2} + 0 + \frac{1}{8} \\ &= 9\frac{5}{8} \\ &= 9.625 \end{aligned}$$

由上面的等式，我们可以得出：向左移动二进制小数点一位相当于这个数除以 2，而向右移动二进制小数点一位相当于这个数乘以 2。如 $101.11 = \frac{3}{4}$ ，而 $10.111 = \frac{7}{8}$ 。除此之外，我们还可以得到这样一个基本规律：一个十进制小数要能用浮点数精确地表示，最后一位必须是 5（当然这是必要条件，并非充分条件）。规律推演如下面的示例所示：

$$\begin{aligned} 0.1 &= 2^{-1} = 0.5 \\ 0.01 &= 2^{-2} = 0.25 \\ 0.001 &= 2^{-3} = 0.125 \\ 0.0001 &= 2^{-4} = 0.0625 \\ 0.00001 &= 2^{-5} = 0.03125 \\ 0.000001 &= 2^{-6} = 0.015625 \\ 0.0000001 &= 2^{-7} = 0.0078125 \\ 0.00000001 &= 2^{-8} = 0.00390625 \end{aligned}$$

我们也可以使用一段 C 程序来验证，如代码清单 1-17 所示。

代码清单 1-17 浮点数示例代码

```
#include <stdio.h>
int main(void)
{
```



```

float f1=34.6;
float f2=34.5;
float f3=34.0;
printf("34.6-34.0=%f\n",f1-f3);
printf("34.5-34.0=%f\n",f2-f3);
return 0;
}

```

代码清单 1-17 的运行结果如图 1-19 所示。



图 1-19 代码清单 1-17 的运行结果

之所以“ $34.6-34.0=0.599998$ ”，产生这个误差的原因是 34.6 无法精确地表达为相应的浮点数，而只能保存为经过舍入的近似值。而这个近似值与 34.0 之间的运算自然无法产生精确的结果。

上面阐述了二进制数转换十进制数，如果你要将十进制数转换成二进制数，则需要把整数部分和小数部分分别转换。其中，整数部分除以 2，取余数；小数部分乘以 2，取整数位。如将 13.125 转换成二进制数如下：

首先转换整数部分（13），除以 2，取余数，所得结果为 1101。

其次转换小数部分（0.125），乘以 2，取整数位。转换过程如下：

$0.125 \times 2 = 0.25$ 取整数位 0

$0.25 \times 2 = 0.5$ 取整数位 0

$0.5 \times 2 = 1$ 取整数位 1

小数部分所得结果为 001，即 $13.125=1101.001$ ，用规范浮点数表达为 1.101001×2^3 。

除此之外，与浮点表示法相关联的其他两个参数是“最大允许指数”和“最小允许指数”，即 e_{\max} 和 e_{\min} 。由于存在 β^p 个可能的有效数字，以及 $e_{\max}-e_{\min}+1$ 个可能的指数，因此浮点数可以按 $[\log_2(e_{\max}-e_{\min}+1)]+[\log_2(\beta^p)]+1$ 位编码，其中最后的 +1 用于符号位。

2. 浮点数表示法

直到 20 世纪 80 年代（即在没有制定 IEEE 754 标准之前），业界还没有一个统一的浮点数标准。相反，很多计算机制造商根据自己的需要来设计自己的浮点数表示规则，以及浮点数的执行运算细节。另外，他们常常并不太关注运算的精确性，而把实现的速度和简易性看得比数字的精确性更重要，而这就给代码的可移植性造成了重大的障碍。

直到 1976 年，Intel 公司打算为其 8086 微处理器引进一种浮点数协处理器时，意识到作为芯片设计者的电子工程师和固体物理学家也许并不能通过数值分析来选择最合理的浮点数二进制格式。于是，他们邀请加州大学伯克利分校的 William Kahan 教授（当时最优秀的数

值分析家) 来为 8087 浮点处理器 (FPU) 设计浮点数格式。而这时, William Kahan 教授又找来两个专家协助他, 于是就有了 KCS 组合 (Kahn、Coonan 和 Stone), 并共同完成了 Intel 公司的浮点数格式设计。

由于 Intel 公司的 KCS 浮点数格式完成得如此出色, 以致 IEEE (Institute of Electrical and Electronics Engineers, 电子电气工程师协会) 决定采用一个非常接近 KCS 的方案作为 IEEE 的标准浮点格式。于是, IEEE 于 1985 年制订了二进制浮点运算标准 IEEE 754 (IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985), 该标准限定指数的底为 2, 并于同年被美国引用为 ANSI 标准。目前, 几乎所有的计算机都支持 IEEE 754 标准, 它大大地改善了科学应用程序的可移植性。

考虑到 IBM System/370 的影响, IEEE 于 1987 年推出了与底数无关的二进制浮点运算标准 IEEE 854, 并于同年被美国引用为 ANSI 标准。1989 年, 国际标准组织 IEC 批准 IEEE 754/854 为国际标准 IEC 559:1989。后来经修订后, 标准号改为 IEC 60559。现在, 几乎所有的浮点处理器完全或基本支持 IEC 60559。同时, C99 的浮点运算也支持 IEC 60559。

IEEE 浮点数标准是从逻辑上用三元组 {S, E, M} 来表示一个数 V 的, 即 $V=(-1)^S \times M \times 2^E$, 如图 1-20 所示。

S (符号位)	E (指数位)	M (有效数字位)
---------	---------	-----------

图 1-20 IEEE 浮点数表示形式

其中:

- ❑ 符号位 s(Sign) 决定数是正数 ($s = 0$) 还是负数 ($s = 1$), 而对于数值 0 的符号位解释则作为特殊情况处理。
- ❑ 有效数字位 M (Significand) 是二进制小数, 它的取值范围为 $1 \sim 2-\varepsilon$, 或者为 $0 \sim 1-\varepsilon$ 。它也被称为尾数位 (Mantissa)、系数位 (Coefficient), 甚至还被称作“小数”。
- ❑ 指数位 E (Exponent) 是 2 的幂 (可能是负数), 它的作用是对浮点数加权。

浮点数格式是一种数据结构, 它规定了构成浮点数的各个字段、这些字段的布局及算术解释。IEEE 754 浮点数的数据位被划分为三个段, 从而对以上这些值进行编码。其中:

- ❑ 一个单独的符号位 s 直接编码符号 s。
- ❑ k 位的指数段 $\text{exp}=e_{k-1} \cdots e_1 e_0$, 编码指数 E。
- ❑ n 位的小数段 $\text{frac}=f_{n-1} \cdots f_1 f_0$, 编码有效数字 M, 但是被编码的值也依赖于指数域的值是否等于 0。

根据 exp 的值, 被编码的值可以分为如下几种不同的情况。

(1) 格式化值

当指数段 exp 的位模式既不全为 0 (即数值 0), 也不全为 1 (即单精度数值为 255, 以单精度数为例, 8 位的指数为可以表达 $0 \sim 255$ 的 255 个指数值; 双精度数值为 2047) 的时候,

就属于这类情况。如图 1-21 所示。

S	$\neq 0 \& \neq 255$	f
---	----------------------	---

图 1-21 格式化值（单精度）

我们知道，指数可以为正数，也可以为负数。为了处理负指数的情况，实际的指数值按
要求需要加上一个偏置（Bias）值作为保存在指数段中的值。因此，这种情况下的指数段被
解释为以偏置形式表示的有符号整数。即指数的值为：

$$E=e-Bias$$

其中， e 是无符号数，其位表示为 $e_{k-1} \cdots e_1 e_0$ ，而 $Bias$ 是一个等于 $2^{k-1}-1$ （单精度是 127，
双精度是 1023）的偏置值。由此产生指数的取值范围是：单精度为 $-126 \sim +127$ ，双精度
为 $-1022 \sim +1023$ 。

对小数段 $frac$ ，可解释为描述小数值 f ，其中 $0 \leq f < 1$ ，其二进制表示为 $0.f_{n-1} \cdots f_1 f_0$ ，
也就是二进制小数点在最高有效位的左边。有效数字定义为 $M = 1 + f$ 。有时候，这种方式
也叫作隐含的以 1 开头的表示法，因为我们可以把 M 看成一个二进制表达式为 $1.f_{n-1}f_{n-2} \cdots f_0$
的数字。既然我们总是能够调整指数 E ，使得有效数字 M 的范围为 $1 \leq M < 2$ （假设没有溢
出），那么这种表示方法是一种轻松获得一个额外精度位的技巧。同时，由于第一位总是等
于 1，因此我们就不需要显式地表示它。拿单精度数为例，按照上面所介绍的知识，实际
上可以用 23 位长的有效数字来表达 24 位的有效数字。比如，对单精度数而言，二进制的
1001.101（即十进制的 9.625）可以表达为 1.001101×2^3 ，所以实际保存在有效数字位中的
值为：

001101000000000000000000

即去掉小数点左侧的 1，并用 0 在右侧补齐。

根据上面所阐述的规则，下面以实数 -9.625 为例，来看看如何将其表达为单精度的浮点
数格式。具体转换步骤如下：

首先，需要将 -9.625 用二进制浮点数表达出来，然后变换为相应的浮点数格式。
即 -9.625 的二进制为 1001.101，用规范的浮点数表达应为 1.001101×2^3 。

其次，因为 -9.625 是负数，所以符号段为 1。而这里的指数为 3，所以指数段为 $3 + 127 =$
130，即二进制的 10000010。有效数字省略掉小数点左侧的 1 之后为 001101，然后在右侧用零
补齐。因此所得的最终结果为：

1	10000010	001101000000000000000000
---	----------	--------------------------

最后，我们还可以将浮点数形式表示为十六进制的数据，如下所示：

(续)

描述	指数	小数	单精度		双精度	
			值	十进制	值	十进制
最小非格式化数	00...00	0...01	$2^{-23} \times 2^{-126}$	1.4×10^{-45}	$2^{-52} \times 2^{-1022}$	4.9×10^{-324}
最大非格式化数	00...00	1...11	$(1-\epsilon) \times 2^{-126}$	1.2×10^{-38}	$(1-\epsilon) \times 2^{-1022}$	2.2×10^{-308}
最小格式化数	00...01	0...00	1×2^{-126}	1.2×10^{-38}	(1×2^{-1022})	2.2×10^{-308}
最大格式化数	11...10	1...11	$(2-\epsilon) \times 2^{127}$	3.4×10^{38}	$(2-\epsilon) \times 2^{1023}$	1.8×10^{308}

3. 标准浮点格式

IEEE 754 标准准确地定义了单精度和双精度浮点格式，并为这两种基本格式分别定义了扩展格式，如下所示：

- ❑ 单精度浮点格式（32 位）。
- ❑ 双精度浮点格式（64 位）。
- ❑ 扩展单精度浮点格式（≥ 43 位，不常用）。
- ❑ 扩展双精度浮点格式（≥ 79 位，一般情况下，Intel x86 结构的计算机采用的是 80 位，而 SPARC 结构的计算机采用的是 128 位）。

其中，只有 32 位单精度浮点数是本标准强烈要求支持的，其他都是可选部分。下面就来对单精度浮点与双精度浮点的存储格式做一些简要的阐述。

(1) 单精度浮点格式

单精度浮点格式共 32 位，其中，s、exp 和 frac 段分别为 1 位、k = 8 位和 n = 23 位，如图 1-25 所示。

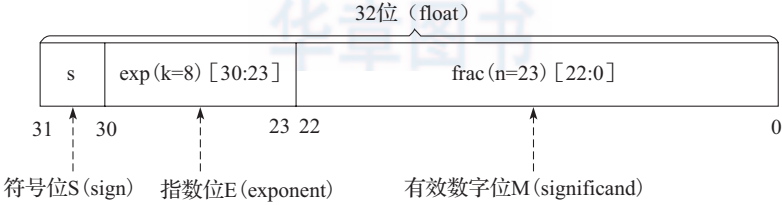


图 1-25 单精度浮点数的存储格式

其中，32 位中的第 0 位存放小数段 frac 的最低有效位 LSB（least significant bit），第 22 位存放小数段 frac 的最高有效位 MSB（most significant bit）；第 23 位存放指数段 exp 的最低有效位 LSB，第 30 位存放指数段 exp 的最高有效位 MSB；最高位，即第 31 位存放符号 s。例如，单精度数 8.25 的存储方式如图 1-26 所示。

(2) 双精度浮点格式

双精度浮点格式共 64 位，其中，s、exp 和 frac 段分别为 1 位、k = 11 位和 n = 52 位，如图 1-27 所示。

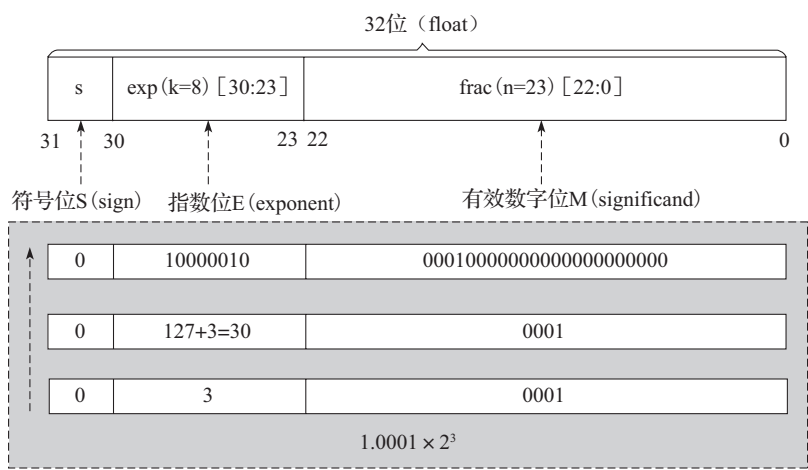


图 1-26 8.25 的存储方式（单精度）

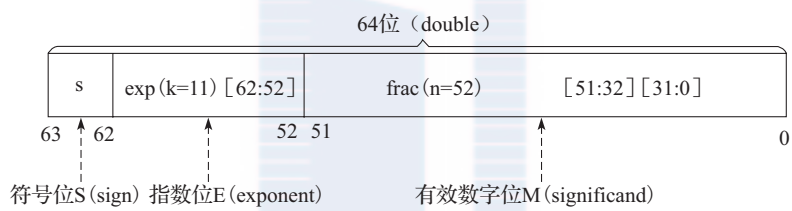


图 1-27 双精度浮点数的存储格式

其中，frac[31:0] 存放小数段的低 32 位（即第 0 位存放整个小数段的最低有效位 LSB，第 31 位存放小数段低 32 位的最高有效位 MSB）；frac [51:32] 存放小数段的高 20 位（即第 32 位存放高 20 位的最低有效位 LSB，第 51 位存放整个小数段的最高有效位 MSB）；第 52 位存放指数段 exp 的最低有效位 LSB，第 62 位存放指数段 exp 的最高有效位 MSB ；最高位，即第 63 位存放符号 s。

在 Intel x86 结构的计算机中，数据存放采用的是小端法（Little Endian），故较低地址的 32 位的字中存放小数段的 frac[31:0] 位。而在 SPARC 结构的计算机中，因其数据存放采用的是大端法（Big Endian），故较高地址的 32 位字中存放小数段的 frac[31:0] 位。

前面主要讨论了 IEEE 754 的单精度与双精度浮点格式，表 1-11 对浮点数的相关参数进行了总结，有兴趣的读者可以根据此表对其他浮点格式进行深入解读。

表 1-11 浮点格式参数总结

参数	浮点格式			
	单精度	双精度	扩展双精度（Intel x86）	扩展双精度（SPARC）
小数段 frac（n）	23	52	63	112
前导有效位	隐含	隐含	显示	隐含

(续)

参数	浮点格式			
	单精度	双精度	扩展双精度 (Intel x86)	扩展双精度 (SPARC)
有效数字 M	24	53	64	113
指数段 exp (k)	8	11	15	15
偏置值 Bias	+127	+1023	+16383	+16383
符号位 s	1	1	1	1
存储格式宽度	32	64	80	128

4. 舍入误差

舍入误差是指运算得到的近似值和精确值之间的差异。大家知道，由于计算机的字长有限，因此在进行数值计算的过程中，对计算得到的中间结果数据要使用相关的舍入规则来取近似值，而这导致计算结果产生误差。

在浮点数的舍入问题上，IEEE 浮点格式定义了 4 种不同的舍入方式，如表 1-12 所示。其中，默认的舍入方法是向偶数舍入，而其他三种可用于计算上界和下界。

表 1-12 四种舍入方式

名 称	描 述
向偶数舍入	也称为向最接近的值舍入，会将结果舍入为最接近且可以表示的值
向 0 舍入	会将结果朝 0 的方向舍入
向上舍入	向 $+\infty$ 方向舍入，会将结果朝正无穷大的方向舍入
向下舍入	向 $-\infty$ 方向舍入，会将结果朝负无穷大的方向舍入

表 1-13 是 4 种舍入方式的应用举例。这里需要特别说明的是，向偶数舍入（向最接近的值舍入）方式会试图找到一个最接近的匹配值。因此，它将 1.4 舍入成 1，将 1.6 舍入成 2，而将 1.5 和 2.5 都舍入成 2。

表 1-13 4 种舍入方式的取值示例演示

值 名称	1.4	1.5	1.6	2.5	-1.5
向偶数舍入	1	2	2	2	-2
向 0 舍入	1	1	1	2	-1
向上舍入	2	2	2	3	-1
向下舍入	1	1	1	2	-2

或许看了上面的内容你会问：为什么要采用向偶数舍入这样的舍入策略，而不直接使用我们已经习惯的“四舍五入”呢？

其原因我们可以这样来理解：在进行舍入的时候，最后一位数字从 1 到 9，舍去的有 1、2、3、4；它正好可以和进位的 9、8、7、6 相对应，而 5 却被单独留下。如果我们采用四舍

五入每次都 5 进位的话，在进行一些大量数据的统计时，就会累积比较大的偏差。而如果采用向偶数舍入的策略，在大多数情况下，5 舍去还是进位概率是差不多的，统计时产生的偏差也就相应要小一些。

同样，针对浮点数据，向偶数舍入方式只需要简单地考虑最低有效数字是奇数还是偶数即可。例如，假设我们想将十进制数舍入到最接近的百分位。不管用哪种舍入方式，我们都将把 1.2349999 舍入到 1.23，而将 1.2350001 舍入到 1.24，因为它们不是在 1.23 和 1.24 的正中间。另一方面我们将把两个数 1.2350000 和 1.2450000 都舍入到 1.24，因为 4 是偶数。

由 IEEE 浮点格式定义的舍入方式可知，不论使用哪种舍入方式，都会产生舍入误差。如果在一系列运算中的一步或几步产生了舍入误差，在某些情况下，这个误差将会随着运算次数的增加而积累得很大，最终会得出没有意义的运算结果。因此，建议不要将浮点数用于精确计算。

当然，理论上增加数字位数可以减少可能会产生的舍入误差。但是，位数是有限的，在表示无限浮点数时仍然会产生误差。在用常规方法表示浮点数的情况下，这种误差是不可避免的，但是可以通过设置警戒位来减小。

除此之外，IEEE 754 还提出 5 种类型的浮点异常，即上溢、下溢、除以零、无效运算和不精确。其中，每类异常都有单独的状态标志。鉴于篇幅有限，本节就不再详细介绍，有兴趣的读者可以参考 IEEE 754 标准文档《IEEE Standard 754 for Binary Floating-Point Arithmetic》进行学习。

建议 3-2：避免使用浮点数进行精确计算

前面已经阐述过，由于计算机的字长有限，浮点数能够精确表示的数是有限的。因此在进行数值计算时，有可能要对计算得到的中间结果数据使用相关的舍入规则来取近似值，而这就会导致计算过程产生误差。示例如代码清单 1-18 所示。

代码清单 1-18 浮点数计算示例

```
#include <stdio.h>
#include <limits.h>
float average(float *arr, size_t size);
enum { array_size = 10 };
int main(void)
{
    float arr[array_size];
    size_t i=0;
    for (i = 0; i < array_size; i++)
    {
        arr[i] = 5.1;
    }
    printf("total / size = %f\n", average(arr, array_size));
    return 0;
}
```

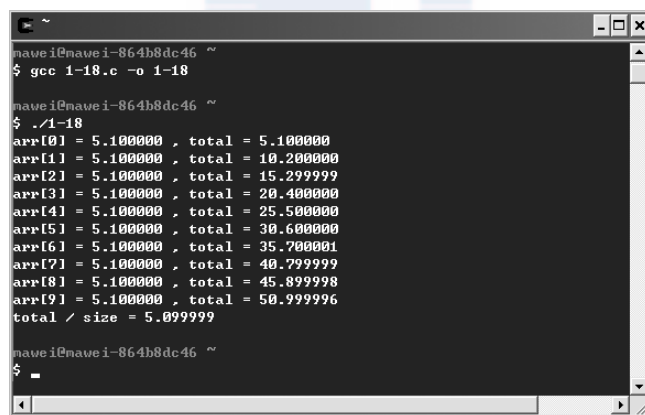


```

float average(float *arr, size_t size)
{
    float total = 0.0;
    size_t i=0;
    if (size > 0&&size<=SIZE_MAX)
    {
        for (i = 0; i < size; i++)
        {
            total += arr[i];
            printf("arr[%d] = %f , total = %f\n",
                i, arr[i], total);
        }
        return total / size;
    }
    else
    {
        return 0.0;
    }
}

```

在代码清单 1-18 中，程序取了 10 个相同的浮点数（5.1）来计算其平均值。理论上，由于这 10 个浮点数是相同的，都是 5.1，因此所计算得出的平均值也应该是 5.1。但实际计算结果并非如此，如图 1-28 所示。



```

naive i@naive i-864b8dc46 ~
$ gcc 1-18.c -o 1-18

naive i@naive i-864b8dc46 ~
$ ./1-18
arr[0] = 5.100000 , total = 5.100000
arr[1] = 5.100000 , total = 10.200000
arr[2] = 5.100000 , total = 15.299999
arr[3] = 5.100000 , total = 20.400000
arr[4] = 5.100000 , total = 25.500000
arr[5] = 5.100000 , total = 30.600000
arr[6] = 5.100000 , total = 35.700001
arr[7] = 5.100000 , total = 40.799999
arr[8] = 5.100000 , total = 45.899998
arr[9] = 5.100000 , total = 50.999996
total / size = 5.099999

naive i@naive i-864b8dc46 ~
$

```

图 1-28 代码清单 1-18 在 GCC 中的运行结果

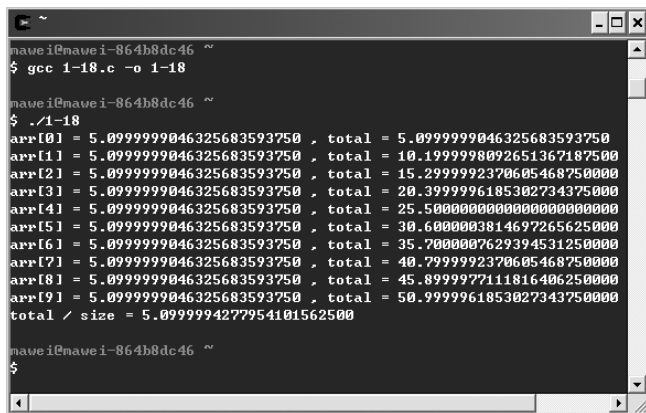
是什么原因导致产生图 1-28 所示的结果呢？

如果你详细看完建议 3-1 中的内容，相信找出答案并不难。其实，之所以得到这样的运行结果，归根结底就是因为 5.1 无法精确地表达为相应的浮点数，而只能保存为经过舍入计算的近似值。这个近似值再进行累加运算之后，自然无法产生精确的结果，最后的平均值结果也就自然而然地产生了误差。

为了让大家能够更加清楚地看见其结果的舍入情况，笔者把代码清单 1-18 里的浮点数保留到小数点 22 位，如下所示：

```
printf("total / size = %.22f\n", average(arr,array_size));
...
printf("arr[%d] = %.22f , total = %.22f\n", i, arr[i], total);
```

运行调整后的代码清单 1-18, 你就可以清楚地看见其舍入结果, 如图 1-29 所示。



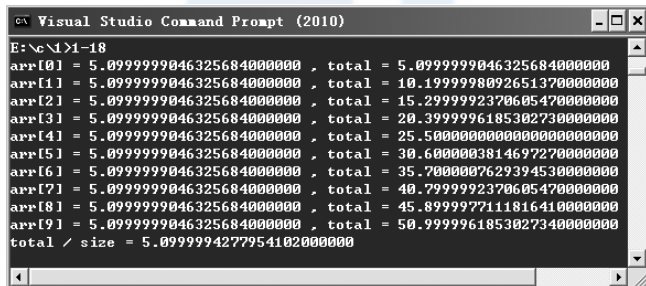
```
mauve i@mauve i-864b8dc46 ~
$ gcc 1-18.c -o 1-18

mauve i@mauve i-864b8dc46 ~
$ ./1-18
arr[0] = 5.0999999046325683593750 , total = 5.0999999046325683593750
arr[1] = 5.0999999046325683593750 , total = 10.1999998092651367187500
arr[2] = 5.0999999046325683593750 , total = 15.2999992370605468750000
arr[3] = 5.0999999046325683593750 , total = 20.3999996185302734375000
arr[4] = 5.0999999046325683593750 , total = 25.5000000000000000000000
arr[5] = 5.0999999046325683593750 , total = 30.6000003814697265625000
arr[6] = 5.0999999046325683593750 , total = 35.7000007629394531250000
arr[7] = 5.0999999046325683593750 , total = 40.7999992370605468750000
arr[8] = 5.0999999046325683593750 , total = 45.8999997711816406250000
arr[9] = 5.0999999046325683593750 , total = 50.9999961853027343750000
total / size = 5.0999994277954101562500

mauve i@mauve i-864b8dc46 ~
$
```

图 1-29 调整后的代码清单 1-18 在 GCC 中的运行结果 (浮点数保留 22 位小数)

当然, 这种结果并不是唯一的, 如果编译器不同, 舍入的值也有可能会不同。例如, 在 VC++ 2010 中运行调整后的代码清单 1-18 的结果如图 1-30 所示。



```
Visual Studio Command Prompt (2010)
E:\c\1>1-18
arr[0] = 5.0999999046325684000000 , total = 5.0999999046325684000000
arr[1] = 5.0999999046325684000000 , total = 10.1999998092651370000000
arr[2] = 5.0999999046325684000000 , total = 15.2999992370605470000000
arr[3] = 5.0999999046325684000000 , total = 20.3999996185302730000000
arr[4] = 5.0999999046325684000000 , total = 25.5000000000000000000000
arr[5] = 5.0999999046325684000000 , total = 30.6000003814697270000000
arr[6] = 5.0999999046325684000000 , total = 35.7000007629394530000000
arr[7] = 5.0999999046325684000000 , total = 40.7999992370605470000000
arr[8] = 5.0999999046325684000000 , total = 45.8999997711816410000000
arr[9] = 5.0999999046325684000000 , total = 50.9999961853027340000000
total / size = 5.0999994277954102000000
```

图 1-30 调整后的代码清单 1-18 在 VC++ 2010 中的运行结果 (浮点数保留 22 位小数)

由此可以看出, 浮点数据会因为其舍入误差而导致运算结果产生误差, 从而失去精确性。因此, 我们应该尽量避免使用浮点数进行精确计算。

当然, 对于代码清单 1-18, 我们还可以通过整数代替浮点数的方法来执行内部加法, 以保证其结果的精确性。而浮点数只用在打印结果及执行除法计算算术平均值的时候, 如代码清单 1-19 所示。

代码清单 1-19 整数代替浮点数示例

```
#include <stdio.h>
#include <limits.h>
float average(int *arr,size_t size);
```

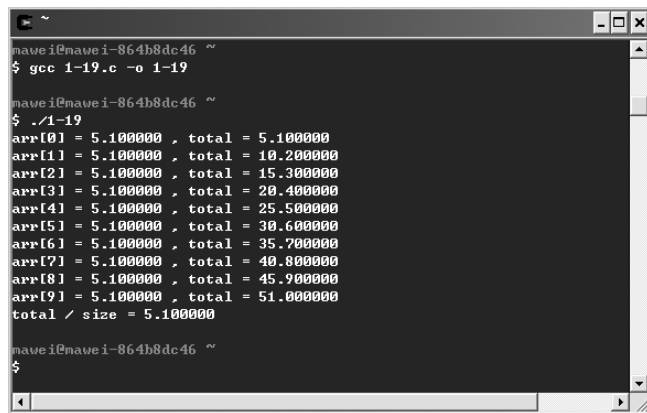
```

enum { array_size = 10 };
int main(void)
{
    int arr[array_size];
    size_t i=0;
    for (i = 0; i < array_size; i++)
    {
        arr[i] = 510;
    }
    printf("total / size = %f\n", average(arr,array_size));
    return 0;
}

float average(int *arr,size_t size)
{
    int total = 0;
    size_t i=0;
    if (size > 0&&size<=SIZE_MAX)
    {
        for (i = 0; i < size; i++)
        {
            total += arr[i];
            printf("arr[%d] = %f , total = %f\n", i,
                (float)arr[i]/100, (float)total/100);
        }
        return (float)(total/ size)/100;
    }
    else
    {
        return 0.0;
    }
}

```

上面代码的运行结果如图 1-31 所示。虽然采用这种方法就可以避免浮点数的舍入误差带来的不精确性，但在一般情况下不建议这样做。



```

navei@navei-864b8dc46 ~
$ gcc 1-19.c -o 1-19

navei@navei-864b8dc46 ~
$ ./1-19
arr[0] = 5.100000 , total = 5.100000
arr[1] = 5.100000 , total = 10.200000
arr[2] = 5.100000 , total = 15.300000
arr[3] = 5.100000 , total = 20.400000
arr[4] = 5.100000 , total = 25.500000
arr[5] = 5.100000 , total = 30.600000
arr[6] = 5.100000 , total = 35.700000
arr[7] = 5.100000 , total = 40.800000
arr[8] = 5.100000 , total = 45.900000
arr[9] = 5.100000 , total = 51.000000
total / size = 5.100000

navei@navei-864b8dc46 ~
$

```

图 1-31 代码清单 1-19 在 GCC 中的运行结果

建议 3-3: 使用分数来精确表达浮点数

在上面的建议 3-1 与建议 3-2 中, 已经明确阐述, 由于计算机的字长有限, 浮点数能够精确表示的数是有限的。因此, 在 C 语言中使用 float 或者 double 类型来存储小数是不能得到精确值的。虽然在建议 3-2 的最后提出使用整数的方法来解决浮点数的精确计算问题, 但这种方案不具备代表性和通用性。因此, 本建议将向你介绍第二种精确表示浮点数的解决方法, 即用分数来表示浮点数。

对于一个浮点数, 我们可以简单地把它分解成整数和纯小数两个部分来分开表示。比如浮点数据 10.234, 它的整数部分是 10, 纯小数部分是 0.234。而对于纯小数部分, 受计算机字长的限制, 存储时会因为舍入规则而导致失去精确性。因此, 这个时候就可以将纯小数部分使用分数来表示。比如, 对于有限小数, 我们可以这样来表示:

$$0.5 = \frac{1}{2}$$

$$0.3 = \frac{3}{10}$$

由此, 我们可以很简单地推导出它的表示形式。对于有限小数 $X=0.a_1a_2\cdots a_n$ (其中, a_1, a_2, \cdots, a_n 为 0 ~ 9 之间的数字), 它的分数表示形式如下:

$$X = \frac{a_1a_2\cdots a_n}{10^n}$$

上面阐述了有限小数的表示形式, 但对于无限循环小数如何表示呢?

其实仍然可以使用上面的这种形式来表示。但无限循环小数比有限小数多了一个循环部分, 因此我们可以用如下方式来表示:

$$0.3(3) = \frac{1}{3}$$

其中, (3) 表示循环体。即, 对于无限循环小数 $X=0.a_1a_2\cdots a_n(b_1b_2\cdots b_m)$ (其中, a_1, a_2, \cdots, a_n 与 b_1, b_2, \cdots, b_m 都是 0 ~ 9 的数字, $a_1a_2\cdots a_n$ 表示非循环部分, 括号部分 $(b_1b_2\cdots b_m)$ 表示循环部分), 它的表现形式如下:

$$X = \frac{(a_1a_2\cdots a_n + 0.(b_1b_2\cdots b_m))}{10^n}$$

至于循环部分 $(b_1b_2\cdots b_m)$, 我们可以这样处理, 即令 $Y=0.b_1b_2\cdots b_m$, 那么:

$$\begin{aligned} 10^m \times Y &= b_1b_2\cdots b_m(b_1b_2\cdots b_m) \\ \Rightarrow 10^m \times Y &= b_1b_2\cdots b_m + 0.(b_1b_2\cdots b_m) \\ \Rightarrow Y &= \frac{b_1b_2\cdots b_m}{(10^m - 1)} \end{aligned}$$

将 Y 代入 X, 即:

$$X = \frac{(a_1a_2\cdots a_n + Y)}{10^n}$$

$$\begin{aligned}
&= \frac{\left(a_1 a_2 \cdots a_n + \frac{b_1 b_2 \cdots b_m}{(10^m - 1)}\right)}{10^n} \\
&= \frac{\left(a_1 a_2 \cdots a_n + \frac{b_1 b_2 \cdots b_m}{(10^m - 1)}\right)}{10^n} \\
&= \frac{(a_1 a_2 \cdots a_n) \times (10^m - 1) + (b_1 b_2 \cdots b_m)}{((10^m - 1) \times 10^n)}
\end{aligned}$$

也就是：

$$X = \frac{(a_1 a_2 \cdots a_n) \times (10^m - 1) + (b_1 b_2 \cdots b_m)}{((10^m - 1) \times 10^n)}$$

例如，对于小数 0.3（3），根据上述方法转化为分数应为：

$$\begin{aligned}
X &= \frac{(3 \times 9 + 3)}{9 \times 10} \\
&= \frac{30}{90} \\
&= \frac{1}{3}
\end{aligned}$$

下面，我们通过一个示例程序演示一下如何在 C 语言程序中使用这种表达方式将浮点数表示为分数形式。值得注意的是，这里的精度只能达到 long int 类型，再大就会发生溢出。如代码清单 1-20 所示。

代码清单 1-20 分数表达浮点数示例

```

#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include<string.h>
long dtol( double d );
long gcd(long a, long b);
void convertdata(char *str);
struct
{
    long  zhengshu;
    long  xiaoshu;
    long  xunhuan;
    long  fenmu;
    long  fenzi;
}fenshu;
union udtol
{
    double d;
    long l;
};

```

```

long dtol( double d )
{
    union udtol to;
    to.d = d + 6755399441055744.0;
    return to.l;
}
long gcd(long a, long b)
{
    if(!b)
    {
        return a;
    }
    else
    {
        return gcd(b, a % b);
    }
}
void convertdata(char *str)
{
    int n = 0;
    int m = 0;
    int len=0;
    int len_1 = 0;
    int len_2 = 0;
    int len_3 = 0;
    char *p1;
    char *p2;
    char *p3;
    int i=0;
    int j=0;
    int z=0;
    long gcb=0;
    fenshu.zhengshu =0;
    fenshu.xiaoshu =0;
    fenshu.xunhuan = 0;
    len = strlen(str);
    len_1 = len;
    p1 = strchr(str, '.');
    p2 = strchr(str, '(');
    p3 = strchr(str, ')');
    if(p1)
    {
        len_1 = p1 - str;
    }
    if(!p2)
    {
        len_2 = len - len_1 - 1;
    }
    if(p3)
    {

```



```

        len_2 = p2 - p1 - 1;
        len_3 = p3 - p2 - 1;
    }
    n = len_2;
    m = len_3;
    for(i = 0; i < len_1; i++)
    {
        fenshu.zhengshu *= 10;
        fenshu.zhengshu += str[i] - '0';
    }
    for(j = 0; j < len_2; j++)
    {
        fenshu.xiaoshu *= 10;
        fenshu.xiaoshu += str[len_1+1+j] - '0';
    }
    for(z = 0; z < len_3; z++)
    {
        fenshu.xunhuan *= 10;
        fenshu.xunhuan += str[len_1+len_2+2+z] - '0';
    }
    fenshu.fenmu = dtol((pow(10.0, (double)(m)) - 1.0)
        * (pow(10.0, (double)(n))));
    fenshu.fenzi = dtol(fenshu.xiaoshu
        * (pow(10.0, (double)(m)) - 1.0)
        + fenshu.xunhuan);
    gcb = gcd(fenshu.fenzi, fenshu.fenmu);
    fenshu.fenmu /= gcb;
    fenshu.fenzi /= gcb;
}
int main(void)
{
    for(;;)
    {
        char str[200] = "";
        printf(" 请输入要转换的数 ( 如 25.444(234)) :");
        scanf("%s",&str);
        convertdata(str);
        printf(" 整数部分 :%ld-- 小数部分 :%ld-- 循环部分 :%ld\n",
            fenshu.zhengshu, fenshu.xiaoshu, fenshu.xunhuan);
        printf(" 所得分数为 :%ld/%ld\n",
            fenshu.fenzi, fenshu.fenmu);
        printf("-----\n");
    }
    return 0;
}

```

代码清单 1-20 的运行结果如图 1-32 所示。

在图 1-32 中，通过分数的形式表示浮点数，从而避免浮点数因为舍入误差而导致的不精确性问题。有兴趣的朋友可以在这个示例代码的基础继续深入研究，从而使该方案能够用于实际的开发环境中。

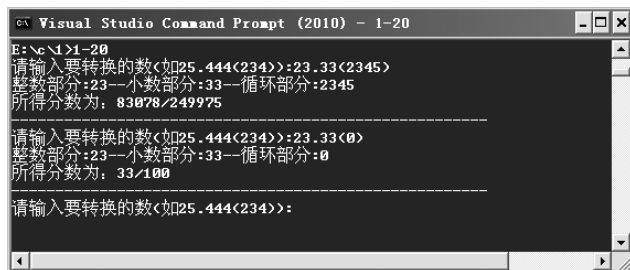


图 1-32 代码清单 1-20 的运行结果

建议 3-4：避免直接在浮点数中使用“==”操作符做相等判断

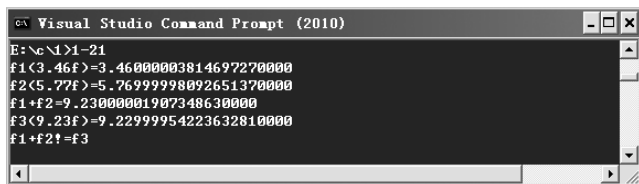
在整型数据中，我们一般都使用“==”操作符来判断两个数是否相等。在浮点数据的运算中，也存在着“==”操作符，那么是否也可以使用这个“==”操作符来判断两个浮点数是否相等呢？带着这个问题，示例程序如代码清单 1-21 所示。

代码清单 1-21 浮点数相等判断示例

```
#include <stdio.h>
int main(void)
{
    float f1=3.46f;
    float f2=5.77f;
    float f3=9.23f;
    printf("f1(3.46f)=%0.20f\nf2(5.77f)=%0.20f\nf1+f2=%0.20f\n\n"
           "f3(9.23f)=%0.20f\n", f1, f2, f1+f2, f3);
    if(f1+f2==f3)
    {
        printf("f1+f2==f3\n");
    }
    else
    {
        printf("f1+f2!=f3\n");
    }
    return 0;
}
```

在代码清单 1-21 中，分别定义了 3 个 float 变量 f1、f2 与 f3。从表面上看，f1+f2 的值应该是 9.23，因此执行条件判断语句“if(f1+f2==f3)”时，应该返回 true。但实际的运行结果并非如此，如图 1-33 所示。

要想使语句“if(f1+f2==f3)”返回 true，那么 f1+f2 和 f3 在浮点格式的精度限制内必须严格相等。这也就意味着，一般情形下（0 除外），浮点格式中的每一个位都必须相等。



```
Visual Studio Command Prompt (2010)
E:\vc\1>1-21
f1<3.46f>=3.46000003814697270000
f2<5.77f>=5.76999998092651370000
f1+f2=9.23000001907348630000
f3<9.23f>=9.22999954223632810000
f1+f2!=f3
```

图 1-33 代码清单 1-21 的运行结果

由于浮点数存在误差，即使是同一意义上的值，如果来源不同，那么判断也就可能不会为 true。换句话说，在浮点计算中，“==”的作用是比较两个浮点数是否具有完全相同的格式数据，而不是一般数学或工程意义上的相等。在浮点计算中两个数据相等的含义通常是指在误差范围内，两个数据的意义一致（即二者描述的物理量的取值一致，或者说相容），因此不能使用“==”操作符进行判断。

既然不能使用“==”操作符进行判断，那么我们又应该怎样正确判断两个浮点数是否相等呢？

一般情况下，浮点数的相等判断通常使用如下形式，即：

$$|V-V_0|<\Delta$$

示例代码如下所示：

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

其中，epsilon 是一个绝对的数据。采用这种形式来判断相等，很显然，如何确定 Δ 就成了问题的关键所在。 Δ 值的确定需要考虑数值背后的含义，而且它总是与误差的概念相随。

（1）依据数据误差进行判断

如果两个数据相差 Δ ，假设一个数据的误差是 Δ_1 ，另一个数据的误差是 Δ_2 ，那么一个简单的判据是：

$$|V-V_0|<\Delta_1+\Delta_2$$

实际上，如果数据不是直接来自某个测量设备，而是某个仿真系统的输出或者是测量数据经过一系列处理的结果，那么 Δ_1 和 Δ_2 大多没有确定的值。此外，这种方法在理论上也不够严谨，只是便于使用而已。

（2）依据允许误差进行判断

在许多情况下，计算精度和数据精度均远远超过了实际需求，使用数据误差进行相等判断除了加大计算量之外，没有实际意义。此时，则可根据实际精度需求确定允许误差，然后用允许误差替代数据误差进行相等判断。这种方法更简单，而且允许误差一般远大于数据误差，可以减小计算量。不过，所谓的允许误差往往没有确定的值，主要依据经验来判断，因此有较大的不确定性。

虽然相对于“==”操作符，使用 `if(fabs(a-b) < epsilon)` 形式进行判断是一个比较好的解决方案，但它却存在着一定的局限性。比如，epsilon 的取值为 0.0001，而 a 和 b 的数值大小

也在 0.0001 附近，那么它显然是不合适的。另外，对于 a 和 b 大小是 10000 这样的数据，它也不合适，因为 10000 和 10001 也可以认为是相等的。

既然这种绝对误差形式 “`if(fabs(a-b) < epsilon)`” 存在着局限性，那么我们可以尝试使用相对误差的形式 “`fabs ((a-b)/a) < epsilon`” 进行判断，示例代码如下所示：

```
bool IsEqual(float a, float b, float epsilon )
{
    return ( fabs ( (a-b)/a ) < epsilon ) ? true : false;
}
```

这样的判断形式看起来是可行的，但它同样存在着局限性。因为它是拿固定的第一个参数做比较的，如果我们分别调用 `IsEqual(a, b, epsilon)` 和 `IsEqual(b, a, epsilon)`，那么可能会得到不同的结果。与此同时，如果第一个参数是 0，很可能会产生除 0 溢出。因此，我们可以把上面的判断形式改造为：除数选取为 a 和 b 当中绝对数值较大的即可，示例代码如下所示：

```
bool IsEqual(float a, float b, float epsilon )
{
    if (fabs(a)>fabs(b))
    {
        return ( fabs((a-b)/a) < epsilon ) ? true : false;
    }
    else
    {
        return (fabs( (a-b)/b) < epsilon ) ? true : false;
    }
}
```

这样看起来就更加完善了。当然，在某些特殊的情况下，相对误差也不能代表全部。因此，我们还需要将相对误差和绝对误差结合使用。完整的比较示例代码如下所示：

```
bool IsEqual(float a, float b, float epsilon )
{
    if (a==b)
    {
        return true;
    }
    if (fabs(a-b)<epsilon )
    {
        return true;
    }
    if (fabs(a)>fabs(b))
    {
        return ( fabs((a-b)/a) < epsilon ) ? true : false;
    }
    else
    {
        return (fabs( (a-b)/b) < epsilon ) ? true : false;
    }
}
```

建议 3-5：避免使用浮点数作为循环计数器

由于不同的系统具有不同的浮点数精度限制，为了使代码保持良好的可移植性，我们应该坚决避免使用浮点数来作为循环计数器。

为了让读者更加深刻地了解使用浮点数作为循环计数器会产生的严重后果，接下来看一段示例代码：

```
float x;  
for(x=100000001.0f;x<=100000010.0f;x+=1.0f)  
{  
    printf("%f",x);  
}
```

在上面的代码中，我们使用了一个非常大的浮点数（100000001.0f）来作为循环计数器，并且每循环一次就会使变量 x 增加 1.0f。从表面上来看，上面的代码应该会循环执行 10 次累加 1.0f。

但实际情况并非如此，相对于浮点数（100000001.0f）而言，如果一个浮点循环计数器的增量太小，那么它就无法在它的精度下更改值。因此，在许多编译器中，上面的这段代码将会产生一个无限循环。例如在 VC++ 2010 中的运行结果如图 1-34 所示。

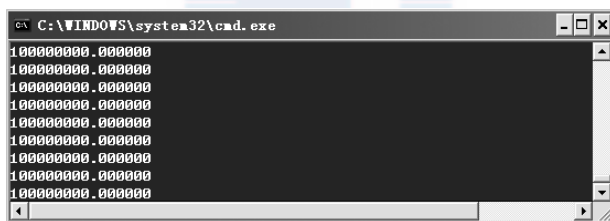


图 1-34 在 VC++ 2010 中的运行结果

上面的示例代码演示了以一个大的浮点数（100000001.0f）来作为循环计数器的情况。或许这个时候有人会想，如果换用一个小的浮点数作为循环计数器结果会怎样呢？

带着这个问题，我们继续看下面的示例代码：

```
float x;  
for(x=0.1f;x<=1.0f;x+=0.1f)  
{  
    printf("%f\n",x);  
}
```

很显然，上面的代码使用了一个比较小的浮点数（1.0f）作为循环计数器，并且每循环一次变量 x 就会增加 0.1f。但是在这里问题出现了，十进制的 0.1 用二进制表示是一个重复的无限循环小数，也就是说无法准确地使用二进制表示，它的精度会受损。如果在 VC++ 2010 中运行上面这段代码，那么循环只会执行 9 次，如图 1-35 所示。



图 1-35 在 VC++ 2010 中的运行结果

由上面的两个示例代码可以看出，不论是使用大的还是小的浮点数来作为循环计数器，都会导致出乎意料的结果。因此，我们应该避免使用浮点数作为循环计数器。如果必须这么做，我们也应该想办法将其转换成整数的形式进行。如下面的示例代码所示：

```
float x;
size_t i;
for(i=1;i<=10;i+=1)
{
    x=i/10.0f;
    printf("%f\n",x);
}
```

建议 3-6：尽量将浮点运算中的整数转换为浮点数

在讨论这个话题之前，我们先看一个示例程序，如代码清单 1-22 所示。

代码清单 1-22 浮点数运算示例

```
#include <stdio.h>
int main(void)
{
    int i1=321;
    float f1=(float)(i1/9);
    float f2=i1/9.0f;
    float f3=(float)i1/9;
    float f4=(float)i1/9.0f;
    printf("(float)(i1/9)=%f\ni1/9.0f=%f\n(float)i1/9=%f\n"
           "(float)i1/9.0f=%f\n",f1,f2,f3,f4);
}
```

在代码清单 1-22 中，我们定义了一个 int 类型的变量 i1，并将 i1 除以 9 的值分别赋给了 float 类型的变量 f1、f2、f3 与 f4。从表面上来看，最后 f1、f2、f3 与 f4 的结果应该完全相同。

但实际情况并非如此，当程序执行语句“float f1=(float)(i1/9)”时，它会先执行表达式“i1/9”，然后将表达式“i1/9”的执行结果 35 转换成浮点类型再赋给 f1，这样就严重地导致数据丢失。执行“float f1=(float)(i1/9)”语句在 VC++ 2010 中产生的汇编代码如下所示：

```
004113A5  mov          eax,dword ptr [i1]
004113A8  cdq
004113A9  mov          ecx,9
004113AE  idiv         eax,ecx
004113B0  mov          dword ptr [ebp-100h],eax
004113B6  fild         dword ptr [ebp-100h]
004113BC  fstp         dword ptr [f1]
```

但当我们计算表达式中的一个整数转换为浮点类型时，如执行语句“float f2=i1/9.0f”时，它将先执行数据转换操作，即先将变量 i1 转换为浮点类型，然后再执行除法计算。执行“float f2=i1/9.0f”语句在 VC++ 2010 中产生的汇编代码如下所示：

```
004113BF  fild         dword ptr [i1]
004113C2  fdiv         qword ptr [__real@4022000000000000 (415788h)]
004113C8  fstp         dword ptr [f2]
```

代码清单 1-22 的运行结果如图 1-36 所示。

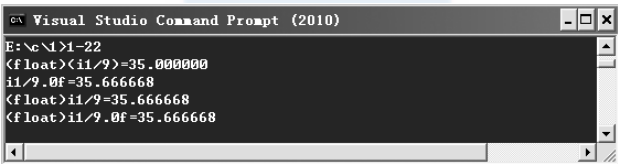


图 1-36 代码清单 1-22 的运行结果

因此，为了避免出现这种信息丢失的情况，我们在使用整数运算计算一个值并把它赋给浮点变量时，必须将表达式中的一个或全部整数转换为浮点数。

建议 4：数据类型转换必须做范围检查

在 C 语言中，数据类型转换一般可分为隐式转换和显式转换，也称为自动转换和强制转换。其中，常见的隐式转换有 4 种，如下所示。

1) 一般算术转换：通过某些运算符将操作数的值从一种类型自动转换成另一种类型，这一规则为“由低级向高级转换”，具体如图 1-37 所示。

根据图 1-37 所示的规则可知，若参与运算的变量类型不同，则先将变量的类型转换成同一类型，然后再进行运算。例如，int 类型的变量和 long 类型的变量参与运算时，则会先把 int 类型的变量转成 long 类型，然后再进行运算。

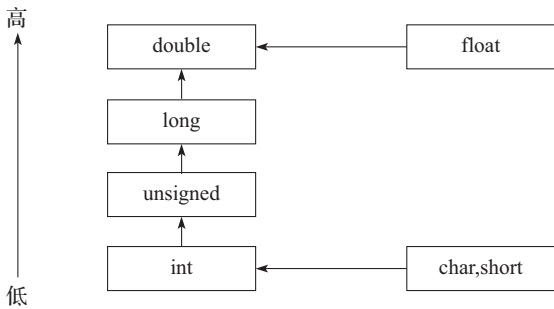


图 1-37 数据类型转换规则

这里需要特别注意的是，所有的浮点运算都是以双精度进行的，即使表达式中仅含 float 单精度变量，也要先将其转换成 double 类型后再进行运算。同时，如果 char 类型的变量和 short 类型的变量参与运算，则必须先转换成 int 类型。

2) 输出转换：输出的操作数类型与输出的格式不一致时所进行的数据类型的转换。如下面的示例代码所示：

```
printf("%u", -1);
```

在 VC++ 2010 中，上面的代码将输出的值为 4294967295。

3) 赋值转换：在赋值运算过程中将赋值运算符右侧的操作数类型转换成左侧操作数据的类型。

4) 函数调用转换：当实参类型和形参类型不一致时数据所进行的转换。

同样，显式转换也提供了两种转换方法，如下所示。

1) 强制性数据类型转换：它是将一种类型的数据强制转换成为另一种数据类型。其格式为：

(数据类型标识符) 表达式；

其作用是将表达式的数据类型强制转换成数据类型标识符所表示的类型。示例代码如下所示：

```
int i1=321;
float f4=(float)i1/9.0f;
```

2) 利用 C 语言提供的标准函数转换，示例代码如下所示：

```
int i1;
char *c;
c="123";
i1=atoi(c);
printf("%d", i1);
```

建议 4-1：整数转换为新类型时必须做范围检查

关于整数类型数据的转换原则，在 C99 的 6.3.1.3 节中做了非常重要的阐述，其表达的主要意思如下：

当我们将一个整数类型的数据转换成除 _Bool 类型之外的另一个整数类型时，如果这个值可以被新的整数类型所表示，那么它就不会被修改，可以正确转换；如果所转换的新类型是无符号的，那么这个值就会反复加上或减去这个新类型可以表示的最大值加 1，直到这个值位于这种新类型的范围之内；如果所转换的新类型是有符号的，并且这个值无法用新类型表示，那么它的结果是由编译器定义的。

因此，为了保证整型数据转换时不会发生丢失或错误解释数据的情况，我们必须做一定

的范围检查，以保证要转换的数据的值在新类型的取值范围之内。而在头文件 `limits.h` 中就定义了相关整型数据的取值范围，例如，在 VC++ 2010 中定义的 `limits.h` 部分代码如下所示：

```
#define CHAR_BIT      8                /* number of bits in a char */
#define SCHAR_MIN     (-128)           /* minimum signed char value */
#define SCHAR_MAX     127             /* maximum signed char value */
#define UCHAR_MAX     0xff            /* maximum unsigned char value */
#ifndef _CHAR_UNSIGNED
#define CHAR_MIN       SCHAR_MIN       /* minimum char value */
#define CHAR_MAX       SCHAR_MAX       /* maximum char value */
#else
#define CHAR_MIN       0
#define CHAR_MAX       UCHAR_MAX
#endif /* _CHAR_UNSIGNED */
#define MB_LEN_MAX     5               /* max. # bytes in multibyte char */
#define SHRT_MIN       (-32768)        /* minimum (signed) short value */
#define SHRT_MAX       32767           /* maximum (signed) short value */
#define USHRT_MAX      0xffff          /* maximum unsigned short value */
#define INT_MIN        (-2147483647 - 1) /* minimum (signed) int value */
#define INT_MAX        2147483647      /* maximum (signed) int value */
#define UINT_MAX       0xffffffff      /* maximum unsigned int value */
#define LONG_MIN       (-2147483647L - 1) /* minimum (signed) long value */
#define LONG_MAX       2147483647L     /* maximum (signed) long value */
#define ULONG_MAX      0xffffffffUL    /* maximum unsigned long value */
#define LLONG_MAX      9223372036854775807i64 /* maximum signed
                                              long long int value */
#define LLONG_MIN      (-9223372036854775807i64 - 1) /* minimum
                                              signed long long int value */
#define ULLONG_MAX     0xffffffffffffffffui64 /* maximum unsigned
                                              long long int value */

#define _I8_MIN        (-127i8 - 1)    /* minimum signed 8 bit value */
#define _I8_MAX        127i8           /* maximum signed 8 bit value */
#define _UI8_MAX       0xffui8         /* maximum unsigned 8 bit value */
#define _I16_MIN       (-32767i16 - 1) /* minimum signed 16 bit value */
#define _I16_MAX       32767i16        /* maximum signed 16 bit value */
#define _UI16_MAX      0xffffui16      /* maximum unsigned 16 bit value */
#define _I32_MIN       (-2147483647i32 - 1) /* minimum signed 32
                                              bit value */
#define _I32_MAX       2147483647i32    /* maximum signed 32 bit value */
#define _UI32_MAX      0xffffffffui32   /* maximum unsigned 32 bit value */
/* minimum signed 64 bit value */
#define _I64_MIN       (-9223372036854775807i64 - 1)
/* maximum signed 64 bit value */
#define _I64_MAX       9223372036854775807i64
/* maximum unsigned 64 bit value */
#define _UI64_MAX      0xffffffffffffffffui64
#if _INTEGRAL_MAX_BITS >= 128
```

```

/* minimum signed 128 bit value */
#define _I128_MIN
(-170141183460469231731687303715884105727i128 - 1)
/* maximum signed 128 bit value */
#define _I128_MAX
170141183460469231731687303715884105727i128
/* maximum unsigned 128 bit value */
#define _UI128_MAX 0xfffffffffffffffffffffffffffffffffui128
#endif

```

举个例子，从一种无符号类型转换为一种有符号类型时，就可能发生数据的高位被截断而导致数据丢失，或者符号位丢失，所以在转换之前要对取值范围进行验证。下面的示例代码演示了如何从 `unsigned int` 类型转换为 `signed char` 类型：

```

unsigned int uil=12345;
signed char scl;
if(uil>SCHAR_MAX)
{
}
else
{
    scl=(signed)uil;
}

```

同样，如果将有符号类型转换为无符号类型，也必须进行取值范围的验证，示例代码如下所示：

```

signed int sil=-12345;
unsigned int uil= 0;
if(sil<0||sil>UINT_MAX)
{
}
else
{
    uil=(unsigned int)sil;
}

```

在数据类型由“高级向低级”转换的时候，同样必须进行取值范围验证，示例代码如下所示：

```

long long int lli1=LLONG_MAX;
int il= 0;
if(lli1<INT_MIN||lli1>INT_MAX)
{
}
else
{
    il=(int)lli1;
}

```


建议 4-2：浮点数转换为新类型时必须做范围检查

关于浮点类型数据的转换原则，在 C99 的 6.3.1.4 节与 6.3.1.5 节中做了非常重要的阐述，其表达的主要意思如下：

当我们将一个浮点类型的数据转换成除 `_Bool` 类型之外的一个整型数据时，该浮点数的小数部分须被丢弃，只保留它的整数部分。如果浮点数整数部分的值无法使用这种整型表示方法时，其行为是未定义的。

与此同时，如果我们将一个整型数据转换成一个浮点类型时，如果该整型数据的值在该浮点数的取值范围内，并且能够被浮点类型精确表示，那么将会被正确转换；如果该整型数据的值在该浮点数的取值范围内，但不能够被浮点类型精确表示，那么转换的结果是最邻近的稍大或者稍小的可表示值；但如果该整型数据的值在该浮点数的取值范围外，其行为是未定义的。

当我们将一个 `double` 类型降级转换为 `float` 类型、将 `long double` 类型降级转换到 `double` 或者 `float` 类型时，如果转换的值在新类型的取值范围内，并且能够被新类型精确表示，那么将会被正确转换；如果转换的值在新类型的取值范围内，但不能够被新类型精确表示，那么转换的结果是最邻近的稍大或者稍小的可表示值；但如果转换的值在新类型的取值范围外，其行为是未定义的。

由此可见，为了避免浮点数据转换时导致的未定义行为，我们应该在转换时对数据进行相关的范围检查。例如，下面的代码清单 1-23 演示了如何将 `double` 类型转换为 `int` 类型。

代码清单 1-23 double 转换为 int 类型示例

```
#include <stdio.h>
#include<limits.h>
int main(void)
{
    double d1=2147483648.01;
    int i1=0;
    if(d1>(double)INT_MAX||d1<(double)INT_MIN)
    {
    }
    else
    {
        i1=(int)d1;
    }
    printf("i1=%d\n",i1);
    return 0;
}
```

在上面的程序中，我们通过语句“`if(d1>(double)INT_MAX||d1<(double)INT_MIN)`”来对程序做类型转换时的取值范围检查，这样就可以避免在执行语句“`i1=(int)d1`”时发生未定义行为。

但需要特别强调的是，上面的程序是建立在 `double` 类型的取值范围大于 `int` 类型的取值

范围的基础之上的。因此，在使用这种方法做取值范围检查时，你必须完全明白不同编译器所对应的相关类型的取值范围。假设在某个编译器中，double 类型的取值范围小于 int 类型的取值范围，那么上面这种方法将是不可行的，实际上这种情况基本没有。

相对于浮点数与整数之间的转换，浮点数与浮点数之间的转换就简单多了。演示示例如代码清单 1-24 所示。

代码清单 1-24 double 与 float 类型转换示例

```
#include <stdio.h>
#include<limits.h>
#include<float.h>
int main(void)
{
    long double ld1=1.7976931348623158e+308;
    double d1=1.0;
    double d2=1.0;
    float f1=1.0f;
    float f2=1.0f;
    /*double->float*/
    if(d1>FLT_MAX||d1<FLT_MIN)
    {
    }
    else
    {
        f1=(float)d1;
    }
    /*long double->double*/
    if(ld1>DBL_MAX||ld1<DBL_MIN)
    {
    }
    else
    {
        d2=(double)ld1;
    }
    /*long double->float*/
    if(ld1>FLT_MAX||ld1<FLT_MIN)
    {
    }
    else
    {
        f2=(float)ld1;
    }
    return 0;
}
```

建议 5：使用有严格定义的数据类型

大家都知道，C 语言是一种既具有高级语言的特点，又具有汇编语言特点的程序设计语

言。它既可以作为系统设计语言来编写系统应用程序，也可以作为应用程序设计语言来编写不依赖计算机硬件的应用程序。因此，它是一种可移植性很高的语言，用它所写的程序可以很方便地部署到不同的平台之上。

尽管如此，C 语言在可移植性方面实际上还是存在着许多重要的问题。除了不同的系统使用的 C 语言标准库不同之外，预处理程序和语言本身在许多重要方面也会不尽相同。我们知道，ANSI 委员会对 C 语言的大多数问题进行了标准化，从而使程序员可以很方便地写出可移植的代码。但是，ANSI 标准却并没有准确定义像 `char`、`int` 和 `long` 这样的内部数据类型，而是将这些重要的实现细节留给编译程序的研制者来决定。

例如，某一个 ANSI 标准的编译程序可能具有 32 位的 `int` 和 `char` 类型，它们在默认状态下是有符号的；而另一个 ANSI 标准的编译程序可能有 16 位的 `int` 和 `char` 类型，默认状态下是无符号的。尽管如此不同，但这两个编译程序却都是严格符合 ANSI 标准的。为了让读者更加深入地了解这种情况，我们来看下面一段示例代码：

```
char ch;
ch= (char)0xff;
if(ch == 0xff)
{
}
```

在上面的代码中，我们先将整数 `0xff` 赋给 `char` 类型的变量 `ch`，然后再将 `ch` 变量与整数 `0xff` 进行比较。从表面上看，语句“`if(ch == 0xff)`”应该返回真。但实际情况并非如此，语句“`if(ch == 0xff)`”的具体返回值因系统而异，也就是说它有可能返回真，也有可能返回假。或许有人会疑惑，这么明显的一个语句怎么会发生这种情况呢？

其实，原因很简单。上面我们说过，ANSI 标准确并没有准确定义像 `char`、`int` 和 `long` 这样的内部数据类型，而是将这些重要的实现细节留给了编译程序。因此，它的结果完全依赖于编译程序。如果默认字符是无符号的，则语句“`if(ch == 0xff)`”的返回值肯定为真；但对字符为有符号的编译程序而言，语句“`if(ch == 0xff)`”的返回值却会为假。

在上面的代码中，字符 `ch` 要与整型数 `0xff` 进行比较。根据 C 语言的转换规则，编译程序必须首先将 `ch` 转换为整型 `int`，待两者类型一致后再进行比较。这样，如果 `int` 是 32 位的，则在转换中会将其值从 `0xff` 扩充为 `0xffffffff`。因此，语句“`if(ch == 0xff)`”的返回值就为假了。

其实，对于上面的这些问题，ANSI 委员会成员并非视而不见。实际上，他们考查了大量的 C 语言实现并得出了这样的结论：由于各编译程序之间的类型定义是如此不同，以致定义严格的标准将会使大量现存代码无效。而这就恰恰违背了他们的一个重要指导原则：“现存代码是非常重要的。”

除此之外，对类型进行严格约束也将违背委员会的另外一个指导原则：“保持 C 语言的活力，即使不能保证它具有可移植性，也要使其运行速度快。”因此，如果实现者感到有符号字符对给定的机器来说更有效，那么就使用有符号字符吧！同样，硬件实现者可以将 `int`

选择为 16 位、32 位或别的位数。也就是说，在默认状态下，用户并不知道位域是有符号的还是无符号的。

当然，这种内部类型在其规格说明中存在着一个不足之处，在今后升级或改变编译程序时，或者移到新的目标环境时，或者与其他单位共享代码时，甚至在改变工作且所用编译程序的规则全部改变时，这个不足就会体现出来。但是，这并不意味着用户就不能安全地使用这些内部类型。其实，只要用户不对 ANSI 标准没有明确说明的类型再作假设，用户就可以安全使用内部类型。例如，对于 `char` 数据类型，只要它能提供 0 ~ 127 的值（即有符号字符和无符号字符域的交集），一般就是可移植的。例如下面这段代码：

```
int strcmp(const char *strLeft,const char *strRight)
{
    assert(strLeft!=NULL&&strRight!=NULL);
    int ret=0;
    while(!(ret= *strLeft-*strRight) && *strRight)
    {
        strLeft++,strRight++;
    }
    if(ret<0)
    {
        ret=-1;
    }
    else if(ret>0)
    {
        ret=1;
    }
    else
    {
        ret=0;
    }
    return ret;
}
```

在上面代码中，`strcmp` 函数用于比较两个字符串。如果 `strLeft < strRight`，则返回 -1；如果 `strLeft==strRight`，则返回 0；如果 `strLeft>strRight`，则返回 1。从表面上看，`strcmp` 函数并没有什么大的问题，但如果仔细观察，你会发现 `strcmp` 函数在可移植方面存在问题。因此，我们需要将 `strLeft` 和 `strRight` 参数声明为无符号字符指针，如下面的代码所示：

```
int strcmp(const unsigned char *strLeft,
           const unsigned char *strRight)
{
    assert(strLeft!=NULL&&strRight!=NULL);
    int ret=0;
    while(!(ret= *strLeft-*strRight) && *strRight)
    {
        strLeft++,strRight++;
    }
}
```

```

    if(ret<0)
    {
        ret=-1;
    }
    else if(ret>0)
    {
        ret=1;
    }
    else
    {
        ret=0;
    }
    return ret;
}

```

当然，我们也可以直接在函数里对其进行修改，如下面的代码所示：

```

int strcmp(const char *strLeft,const char *strRight)
{
    assert(strLeft!=NULL&&strRight!=NULL);
    int ret=0;
    while(!(ret= *(unsigned char*)strLeft
        -*(unsigned char*)strRight) && *strRight)
    {
        strLeft++,strRight++;
    }
    if(ret<0)
    {
        ret=-1;
    }
    else if(ret>0)
    {
        ret=1;
    }
    else
    {
        ret=0;
    }
    return ret;
}

```

其实，面对上面的问题时，只需要记住一个简单的原则，就是不要在表达式中使用“简单的”字符。当然，位域也有同样的问题，因此也有一个类似的原则：任何时候都不要使用“简单的”位域。例如，下面的代码在任何编译程序上都可以工作，因为它没有对域作假定。

```

char* strcpy(char *strDst,const char *strSrc)
{
    char *ret = strDst;
    assert(strDst!=NULL&&strSrc!=NULL);
    while ((*strDst++=*strSrc++)&&strlen(strDst)!=0)

```

```

        NULL;
    return ret;
}

```

最后，对于编写可移植程序还有这样一个问题：有些程序员可能会认为使用可移植的类型比使用“自然的”类型效率更低。例如，假定 `int` 类型的物理字长对目标硬件是最有效的。这就意味着这种“自然的”位数可能大于 16 位，所保持的值也可能大于 32767。现在假定用户的编译程序使用的是 32 位的 `int`，且要求使用 0 至 40000 的值域。那么，是为了使机器可以在 `int` 内有效地处理 40000 个值而使用 `int` 呢，还是坚持使用可移植类型，而用 `long` 代替 `int` 呢？

其实这要具体情况具体分析。

对于对效率要求比较高的程序。大家一致认为如果能够使用 `char` 定义的变量，就不要使用 `int` 定义的变量；能够使用 `int` 定义的变量，就不要用 `long` 变量来定义；能不使用浮点型变量就不要使用浮点型变量。当然，在定义变量后不要让变量超过其作用范围，如果超过变量的范围赋值，C 语言编译器并不会报错，但程序的运行结果却错了，而且这样的错误很难发现。

如果基于可移植性来考虑，要是机器使用的是 32 位 `int`，那么也可以使用 32 位 `long`，因为这两者产生的代码即使不相同也很相似，所以我们可以使用 `long`。用户即便担心在将来必须支持的机器上使用 `long` 可能会效率低一些，那也应该坚持使用可移植类型。

总之，不论怎样，我们都应该坚持这样一个原则：那就是尽量使用严格形式定义的、可移植的数据类型，尽量不要使用与具体硬件或软件环境关系密切的变量。

建议 6：使用 typedef 来定义类型的新别名

C 语言允许用户使用 `typedef` 关键字来定义自己习惯的数据类型名称，来替代系统默认的基本类型名称、数组类型名称、指针类型名称与用户自定义的结构型名称、共用型名称、枚举型名称等。一旦用户在程序中定义了自己的数据类型名称，就可以在该程序中用自己的数据类型名称来定义变量的类型、数组的类型、指针变量的类型与函数的类型等。

例如，C 语言在 C99 之前并未提供布尔类型，但我们可以使用 `typedef` 关键字来定义一个简单的布尔类型，如下面的代码所示：

```

typedef int BOOL;
#define TRUE 1
#define FALSE 0

```

定义好之后，就可以像使用基本类型数据一样使用它了，如下面的代码所示：

```

BOOL bflag=TRUE;

```

建议 6-1：掌握 typedef 的 4 种应用形式

在实际使用中，`typedef` 的应用主要有如下 4 种形式。

1. 为基本数据类型定义新的类型名

也就是说，系统默认的所有基本类型都可以利用 `typedef` 关键字来重新定义类型名，示例代码如下所示：

```
typedef unsigned int COUNT;
```

而且，我们还可以使用这种方法来定义与平台无关的类型。比如，要定义一个叫 `REAL` 的浮点类型，在目标平台一上，让它表示最高精度的类型，即：

```
typedef long double REAL;
```

在不支持 `long double` 的平台二上，改为：

```
typedef double REAL;
```

甚至还可以在连 `double` 都不支持的平台三上，改为：

```
typedef float REAL;
```

这样，当跨平台移植程序时，我们只需要修改一下 `typedef` 的定义即可，而不用对其他源代码做任何修改。其实，标准库中广泛地使用了这个技巧，比如 `size_t` 在 VC++ 2010 的 `crtdefs.h` 文件中的定义如下所示：

```
#ifndef _SIZE_T_DEFINED
#ifdef _WIN64
typedef unsigned __int64 size_t;
#else
typedef _W64 unsigned int size_t;
#endif
#define _SIZE_T_DEFINED
#endif
```

2. 为自定义数据类型（结构体、共用体和枚举类型）定义简洁的类型名称

以结构体为例，下面我们定义一个名为 `Point` 的结构体：

```
struct Point
{
    double x;
    double y;
    double z;
};
```

在调用这个结构体时，我们必须像下面的代码这样来调用这个结构体：

```
struct Point oPoint1={100,100,0};
struct Point oPoint2;
```

在这里，结构体 `struct Point` 为新的数据类型，在定义变量的时候均要向上面的调用方法一样有保留字 `struct`，而不能像 `int` 和 `double` 那样直接使用 `Point` 来定义变量。现在，我们利

用 `typedef` 定义这个结构体，如下面的代码所示：

```
typedef struct tagPoint
{
    double x;
    double y;
    double z;
} Point;
```

在上面的代码中，实际上完成了两个操作：

❑ 定义了一个新的结构类型，代码如下所示：

```
struct tagPoint
{
    double x;
    double y;
    double z;
};
```

其中，`struct` 关键字和 `tagPoint` 一起构成了这个结构类型，无论是否存在 `typedef` 关键字，这个结构都存在。

❑ 使用 `typedef` 为这个新的结构起了一个别名，叫 `Point`，即：

```
typedef struct tagPoint Point
```

因此，现在你就可以像 `int` 和 `double` 那样直接使用 `Point` 定义变量，如下面的代码所示：

```
Point oPoint1={100,100,0};
Point oPoint2;
```

为了加深对 `typedef` 的理解，我们再来看一个结构体例子，如下面的代码所示：

```
typedef struct tagNode
{
    char *pItem;
    pNode pNext;
} *pNode;
```

从表面上看，上面的示例代码与前面的定义方法相同，所以应该没有什么问题。但是编译器却报了一个错误，为什么呢？莫非 C 语言不允许在结构中包含指向它自己的指针？

其实问题并非在于 `struct` 定义的本身，大家应该都知道，C 语言是允许在结构中包含指向它自己的指针的，我们可以在建立链表等数据结构的实现上看到很多这类例子。那问题在哪里呢？其实，根本问题还是在于 `typedef` 的应用。

在上面的代码中，新结构建立的过程中遇到了 `pNext` 声明，其类型是 `pNode`。这里要特别注意的是，`pNode` 表示的是该结构体的新别名。于是问题出现了，在结构体类型本身还没有建立完成的时候，编译器根本就不认识 `pNode`，因为这个结构体类型的新别名还不存在，所以自然就会报错。因此，我们要做一些适当的调整，比如将结构体中的 `pNext` 声明修改成

如下方式：

```
typedef struct tagNode
{
    char *pItem;
    struct tagNode *pNext;
} *pNode;
```

或者将 struct 与 typedef 分开定义，如下面的代码所示：

```
typedef struct tagNode *pNode;
struct tagNode
{
    char *pItem;
    pNode pNext;
};
```

在上面的代码中，我们同样使用 typedef 给一个还未完全声明的类型 tagNode 起了一个新别名。不过，虽然 C 语言编译器完全支持这种做法，但不推荐这样做。建议还是使用如下规范定义方法：

```
struct tagNode
{
    char *pItem;
    struct tagNode *pNext;
};
typedef struct tagNode *pNode;
```

3. 为数组定义简洁的类型名称

它的定义方法很简单，与为基本数据类型定义新的别名方法一样，示例代码如下所示：

```
typedef int INT_ARRAY_100[100];
INT_ARRAY_100 arr;
```

4. 为指针定义简洁的名称

对于指针，我们同样可以使用下面的方式来定义一个新的别名：

```
typedef char* PCHAR;
PCHAR pa;
```

对于上面这种简单的变量声明，使用 typedef 来定义一个新的别名或许会感觉意义不大，但在比较复杂的变量声明中，typedef 的优势马上就体现出来了，如下面的示例代码所示：

```
int *(*a[5])(int, char*);
```

对于上面变量的声明，如果我们使用 typedef 来给它定义一个别名，这会非常有意义，如下面的代码所示：

```
// PFun 是我们创建的一个类型别名
typedef int *(*PFun)(int, char*);
// 使用定义的新类型来声明对象，等价于 int* (*a[5])(int, char*);
PFun a[5];
```

建议 6-2：小心使用 typedef 带来的陷阱

接下来看一个简单的 typedef 使用示例，如下面的代码所示：

```
typedef char* PCHAR;
int strcmp(const PCHAR, const PCHAR);
```

在上面的代码中，“const PCHAR”是否相当于“const char*”呢？

答案是否定的，原因很简单，typedef 是用来定义一种类型的新别名的，它不同于宏，不是简单的字符串替换。因此，“const PCHAR”中的 const 给予了整个指针本身常量性，也就是形成了常量指针“char* const（一个指向 char 的常量指针）”。即它实际上相当于“char* const”，而不是“const char*（指向常量 char 的指针）”。当然，要想让 const PCHAR 相当于 const char* 也很容易，如下面的代码所示：

```
typedef const char* PCHAR;
int strcmp(PCHAR, PCHAR);
```

其实，无论什么时候，只要为指针声明 typedef，那么就应该在最终的 typedef 名称中加一个 const，以使得该指针本身是常量。

还需要特别注意的是，虽然 typedef 并不真正影响对象的存储特性，但在语法上它还是一个存储类的关键字，就像 auto、extern、static 和 register 等关键字一样。因此，像下面这种声明方式是不可行的：

```
typedef static int INT_STATIC;
```

不可行的原因是不能声明多个存储类关键字，由于 typedef 已经占据了存储类关键字的位置，因此，在 typedef 声明中就不能够再使用 static 或任何其他存储类关键字了。当然，编译器也会报错，如在 VC++ 2010 中的报错信息为“无法指定多个存储类”。

建议 6-3：typedef 不同于 #define

前面已经特别强调过，typedef 是用来定义一种类型的新别名的，它不同于宏（#define），不是简单的字符串替换。它的新名字具有一定的封装性，所以新命名的标识符具有更易定义变量的功能，它是语言编译过程的一部分，但它并不实际分配内存空间。

而 #define 只是简单的字符串替换（原地扩展），它本身并不在编译过程中进行，而是在这之前（预处理过程）就已经完成了。因此，它不会做正确性检查，不管含义是否正确它照样会带入，只有在编译已被展开的源程序时才会发现可能的错误并报错。

接下来看下面的示例代码：

```
typedef char * PCHAR1;  
#define PCHAR2 char *  
.....  
/* c1、c2 都为 char *,typedef 为 char * 引入了一个新的别名 */  
PCHAR1 c1, c2;  
/* 相当于 char * c3, c4;c3 是 char *, 而 c4 是 char */  
PCHAR2 c3, c4;
```

在定义上述的变量时，c1、c2 与 c3 按照预期都被定义成 char * 类型。值得注意的是，c4 却被定义成 char 类型，而不是我们所预期的 char *。其根本原因就在于 #define 只是简单的字符串替换，而 typedef 则是为一个类型引入一个新的别名。

建议 7：变量声明应该力求简洁

对于“变量”这个词语，相信大家再熟悉不过了，任何一种编程语言都离不开变量。变量是在内存或寄存器中用一个标识符命名的存储单元，可以用来存储一个特定类型的数据，并且数据的值在程序运行过程中可以修改。例如：

```
int i;
```

上面这个语句定义了一个 int 类型的变量 i，即它要求系统在内存中分配一个类型为 int 型的存储空间。因此，执行语句“int i”后，内存中的映像可能会如图 1-38 所示。

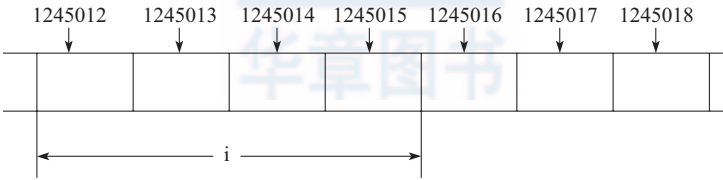


图 1-38 变量 (int i) 的存储

在 32 位计算机系统中，int 型变量占用 4 个字节（即图 1-38 中编号为 1245012 ~ 1245015 的 4 个存储单元）。当然，你也可以使用语句“sizeof(i)”得到存储字节。同时，还可以从图 1-38 中看出，变量名实质是内存单元地址的一个符号，比如，变量 i 就代表着内存地址 1245012，即变量所占内存单元的首地址。

由此可见，变量首先是一个标识符或名称，就像一个客房的编号一样，有了这个编号我们在交流中就可以方便表达，否则，我们只可意会，那多不方便。为了方便，我们在给变量命名时，首先，最好符合大多数人的习惯，基本可以望名知义，这就会便于交流和维护；其次，变量是唯一确定的对应内存若干存储单元或者某个寄存器的。当用户使用变量时，其本质是访问该变量所对应的内存单元。

一旦定义了变量，那么变量就至少需要为我们提供两个信息：一是变量的地址，即操作系统为变量在内存中分配的若干内存的首地址；二是变量的值，即变量在内存中所分配的那些内存单元中所存放的数据。

因此，我们至少还需要给上面的变量 *i* 赋上一个初值，如下面的代码所示：

```
i=100;
```

上面的语句“*i*=100”表示将整型常量 100 保存到 *i* 中，实质上是将 100 保存到内存中以 1245012 为起始地址的 4 个存储单元（即 1245012 ~ 1245015）。因此，执行语句“*i*=100”后，可想象内存映像如图 1-39 所示。

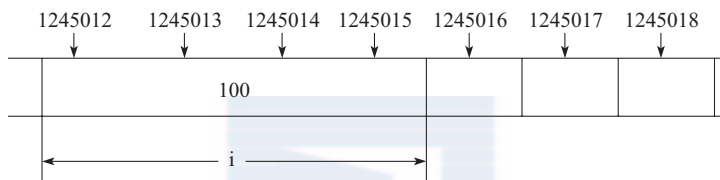


图 1-39 在内存中存入数据 (int *i*=100)

建议 7-1：尽量不要在一个声明中声明超过一个的变量

变量声明应该力求简洁明了，每一行应该只声明一个变量，不要把多个变量的声明或初始化放在同一行中。尽管这样的声明方式是 C 语言所允许的，但我们还是建议你不要这样做。来看下面的代码：

```
int i1,i2;
int i3=0,i4=1;
```

很显然，上面的这种变量声明方式虽然节省了行数，但却也失去了简洁性。所以，建议使用下面的这种声明方式：

```
int i1;
int i2;
int i3=0;
int i4=1;
```

上面的变量声明示例或许会让部分读者不以为然，但如果遇到下面这种变量声明方式，估计会令人混淆不清。

```
char* p1,p2;
char *p3,p4;
```

因此，我们应该避免这种声明方法。

除此之外，建议尽可能在声明变量的同时初始化该变量。如果变量的引用处和其定义处相隔比较远，变量的初始化就很容易被忘记，而要是引用了未被初始化的变量，很可能会导

致程序错误的。初始化示例代码如下所示：

```
int width = 10;
int height = 10;
int depth = 10;
```

建议 7-2：避免在嵌套的代码块之间使用相同的变量名

当一个作用域嵌套在另一个作用域内部时，我们应该避免在这两个作用域中使用相同的变量名称。例如，如果某些局部变量位于一个全局变量的子作用域中，那么这些局部变量都不应该与这个全局变量同名。

例如下面的示例代码：

```
int height = 100;
double mtocm( double value )
{
    double height=0;
    height=value*height;
    return height;
}
```

在上面的代码中，首先声明了一个 `int` 类型的全局变量 `height`，同时又在 `mtocm` 函数里声明了一个与全局变量名称相同的 `double` 类型的局部变量 `height`。最后，我们希望将 `value*height`（全局变量）的值赋给 `double` 类型的局部变量 `height`，并返回。

当执行“`mtocm(1.78)`”语句时，问题发生了。因为虽然 C 语言允许在同一源文件中全局变量和局部变量同名，但在局部变量的作用域内，全局变量将不起任何作用。所以，执行语句“`mtocm(1.78)`”返回的结果将会是 0。因此，我们必须将全局变量名称与局部变量名称区分开，如下面的代码所示：

```
int g_height = 100;
double mtocm( double value )
{
    double height=0;
    height=value*g_height;
    return height;
}
```

现在，再执行语句“`mtocm(1.78)`”时，它将返回正确的结果。

除此之外，如果一个代码块位于其他代码块的内部，同样不应该在此代码块中声明与外层代码块中的任何变量具有相同名称的变量。

建议 8：正确地选择变量的存储类型

在计算机中，保存变量当前值的存储单元有两类：一类是内存，另一类是 CPU 的寄存

器。变量的存储类型关系到变量的存储位置，在C语言中，为变量提供了4种存储类型：`auto`（自动）型、`static`（静态）型、`register`（寄存器）型和`extern`（外部）型。它们关系到变量在内存中的存放位置，由此决定了变量的保留时间和变量的作用范围。

变量的保留时间又称为生存期，从时间的角度来看，可将变量分为静态存储和动态存储两种情况。静态存储是指变量存储在内存的静态存储区中，在编译时就为它分配了存储空间，在整个程序的运行期间，该变量占有固定的存储单元，程序执行结束后，这部分空间才会释放，变量的值在整个程序中始终存在；动态存储是指变量存储在内存的动态存储区中，在程序的运行过程中，只有当变量所在的函数被调用时，编译系统才临时为该变量分配一段内存单元，函数调用结束时，该变量空间就会释放，变量的值只在函数调用期存在。

变量的作用范围又称为作用域，从空间角度来看，可以将变量分为局部变量和全局变量。局部变量是在一个函数或复合语句内定义的变量，它仅在函数或复合语句内有效，编译时，编译系统不为局部变量分配内存单元，而是在程序运行过程中，当局部变量所在的函数被调用时，编译系统才会根据需要临时分配内存，调用结束后，释放空间；全局变量是在函数之外定义的变量，其作用范围为从定义处开始到本文件结束，编译时，编译系统会为其分配固定的内存单元，在程序运行的自始至终它都占用着固定的单元。

建议 8-1：定义局部变量时应该省略 `auto` 关键字

在默认情况下，所有的局部变量都是 `auto` 型的变量（也称为自动变量），而且会为这些变量动态分配存储空间，数据则存储在动态存储区中。因此，它的生存期比较短暂：当调用函数时，系统为该函数的自动变量分配内存，等程序从该函数返回，即调用过程结束时，系统就会释放所有该函数的自动变量。这个过程是通过一个堆栈机制实现的，为自动变量分配内存就压栈，当函数返回时则退栈。

需要说明的是，既然自动变量就是指在函数内部定义使用的变量（局部变量），那么也就只允许在定义它的函数内部使用它，在函数外的其他任何地方都不能使用该变量。当然，这也充分说明自动变量没有链接性，因为它不允许其他的文件进行访问。因此，这也就允许我们在这个函数以外的其他任何地方或其他函数内部定义同名的变量，并且它们之间不会发生任何冲突。虽然这种变量的命名方式不是我们所推荐的，但却是C语言所允许的。

来看一个自动变量的定义示例：

```
int main(void)
{
    /* 定义整型变量 x 为自动变量 */
    auto int x=0;
    /* 定义整型变量 y，缺省存储类型时为自动变量 */
    int y=0;
    .....
}
```

在上面的代码中，默认情况下，所有的局部变量都是自动变量，所以说变量 *x* 与变量 *y* 一样，都是自动变量。因此，我们在声明局部变量时，应该省略 *auto* 关键字。

建议 8-2：慎用 *extern* 声明外部变量

我们都知道，程序的编译单位是源程序文件，一个源文件可以包含一个或若干个函数。在函数内定义的变量是局部变量，而在函数之外定义的变量则称为外部变量，外部变量也就是我们所讲的全局变量。它的存储方式为静态存储，其生存周期为整个程序的生存周期。全局变量可以为本文件中的其他函数所共用，它的有效范围为从定义变量的位置开始到本源文件结束。

然而，如果全局变量不在文件的开头定义，有效的作用范围将只限于其定义处到文件结束。如果在定义点之前的函数想引用该全局变量，则应该在引用之前用关键字 *extern* 对该变量作“外部变量声明”，表示该变量是一个已经定义的外部变量。有了此声明，就可以从“声明”处起，合法地使用该外部变量。

来看一个简单的例子，如代码清单 1-25 所示。

代码清单 1-25 *extern* 使用示例

```
#include <stdio.h>
int max(int x,int y);
int main(void)
{
    int result;
    /* 外部变量声明 */
    extern int g_X;
    extern int g_Y;
    result = max(g_X,g_Y);
    printf("the max value is %d\n",result);
    return 0;
}
/* 定义两个全局变量 */
int g_X = 10;
int g_Y = 20;
int max(int x, int y)
{
    return (x>y ? x : y);
}
```

在代码清单 1-25 中，全局变量 *g_X* 与 *g_Y* 是在 *main* 函数之后声明的，因此它的作用范围不在 *main* 函数中。如果我们需要在 *main* 函数中调用它们，就必须使用 *extern* 来对变量 *g_X* 与 *g_Y* 作“外部变量声明”，以扩展全局变量的作用域。也就是说，如果在变量定义之前要使用该变量，则应在使用之前加 *extern* 声明变量，使作用域扩展到从声明开始到本文件结束。

如果整个工程由多个源文件组成，在一个源文件中想引用另外一个源文件中已经定义的外部变量，同样只需在引用变量的文件中用 `extern` 关键字加以声明即可。下面就来看一个多文件的示例，如代码清单 1-26-max 与代码清单 1-26-main 所示。

代码清单 1-26-max 1-26-max.c

```
#include <stdio.h>
/* 外部变量声明 */
extern int g_X ;
extern int g_Y ;
int max()
{
    return (g_X > g_Y ? g_X : g_Y);
}
```

代码清单 1-26-main 1-26-main.c

```
#include <stdio.h>
/* 定义两个全局变量 */
int g_X=10;
int g_Y=20;
int max();
int main(void)
{
    int result;
    result = max();
    printf("the max value is %d\n",result);
    return 0;
}
```

代码清单 1-26-max 与代码清单 1-26-main 的运行结果如图 1-40 所示。

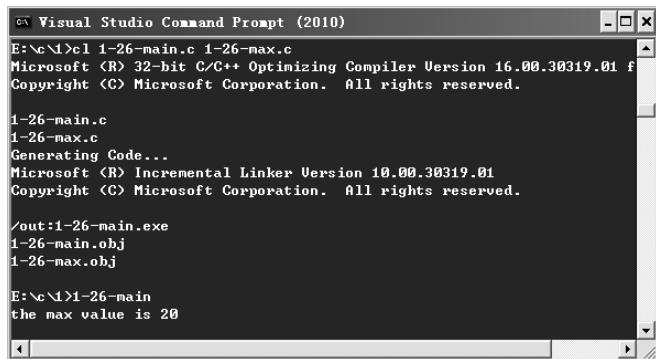


图 1-40 代码清单 1-26-max 与代码清单 1-26-main 的运行结果

对于多个文件的工程，都可以采用上面这种方法来操作。对于模块化的程序文件，可在其文件中预先留好外部变量的接口，也就是只采用 `extern` 声明变量，而不定义变量，代码

清单 1-26-max 里的 g_X 与 g_Y 就是如此操作的。通常，这些外部变量的接口都是在模块程序的头文件中声明的，当需要使用该模块时，只需要在使用时具体定义一下这些外部变量即可。代码清单 1-26-main 里的 g_X 与 g_Y 则是相关示例。

不过，需要特别注意的是，由于用 extern 引用外部变量，可以在引用的模块内修改其变量的值，因此，如果有多个文件同时要对应用的变量进行操作，而且可能会修改该变量，那就就会影响其他模块的使用。因此，我们要慎重使用。

建议 8-3：不要混淆 static 变量的作用

在 C 语言中，static 关键字不仅可以用来修饰变量，还可以用来修饰函数。在使用 static 关键字修饰变量时，我们称此变量为静态变量。静态变量的存储方式与全局变量一样，都是静态存储方式。但这里需要特别说明的是，静态变量属于静态存储方式，属于静态存储方式的变量却不一定就是静态变量。例如，全局变量虽然属于静态存储方式，但并不是静态变量，它必须由 static 加以定义后才能成为静态全局变量。

考虑到可能会有不少读者对静态变量作用不太清楚，本节就来详细讨论一下它的主要作用。

1. 隐藏与隔离的作用

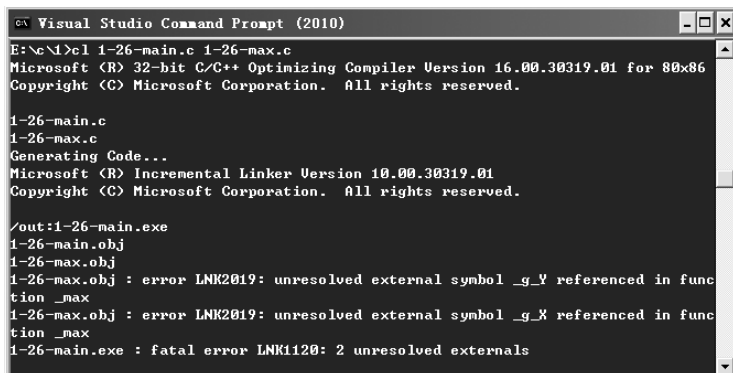
上面已经阐述过，全局变量虽然属于静态存储方式，但并不是静态变量。全局变量的作用域是整个源程序，当一个源程序由多个源文件组成时，全局变量在各个源文件中都是有效的，比如上面代码清单 1-26-main 中的全局变量 g_X 与 g_Y 就是如此。

如果我们希望全局变量仅限于在本源文件中使用，在其他源文件中不能引用，也就是说限制其作用域只在定义该变量的源文件内有效，而在同一源程序的其他源文件中不能使用。这时，就可以通过在全局变量之前加上关键字 static 来实现，即使全局变量被定义成为一个静态全局变量。下面将代码清单 1-26-main 中的全局变量 g_X 与 g_Y 全部修改为静态全局变量，代码如下所示：

```
#include <stdio.h>
/* 定义两个静态全局变量 */
static int g_X=10;
static int g_Y=20;
int max();
int main(void)
{
    int result;
    result = max();
    printf("the max value is %d\n",result);
    return 0;
}
```

这时候，我们再来编译该程序，代码清单 1-26-max 将无法调用代码清单 1-26-main 中的静态全局变量 g_X 与 g_Y。也就是说静态全局变量 g_X 与 g_Y 只能在代码清单 1-26-main 中

使用，如图 1-41 所示。



```

Visual Studio Command Prompt (2010)
E:\c\1>cl 1-26-main.c 1-26-max.c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.30319.01 for x86
Copyright (C) Microsoft Corporation. All rights reserved.

1-26-main.c
1-26-max.c
Generating Code...
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation. All rights reserved.

/out:1-26-main.exe
1-26-main.obj
1-26-max.obj
1-26-max.obj : error LNK2019: unresolved external symbol _g_Y referenced in func
tion _max
1-26-max.obj : error LNK2019: unresolved external symbol _g_X referenced in func
tion _max
1-26-main.exe : fatal error LNK1120: 2 unresolved externals

```

图 1-41 代码清单 1-26-main 将 `g_X` 与 `g_Y` 定义为静态全局变量的运行结果

在图 1-41 中，虽然 `g_X` 与 `g_Y` 变量仍然是全局的，但由于静态全局变量的作用域局限於一个源文件内，所以只能为该源文件内的函数所共用，这样就可以避免在其他源文件中引起的错误。也就起到了对其他源文件进行隐藏与隔离错误的作用，有利于模块化程序设计。

2. 保持变量内容的持久性

有时候，我们希望函数中局部变量的值在函数调用结束之后不会消失，而仍然保留其原值。即它所占用的存储单元不释放，在下次调用该函数时，其局部变量的值仍然存在，也就是上一次函数调用结束时的值。这时候，我们就应该将该局部变量用关键字 `static` 声明为“静态局部变量”。

当将局部变量声明为静态局部变量的时候，也就改变了局部变量的存储位置，即从原来的栈中存放改为静态存储区存放。这让它看起来很像全局变量，其实静态局部变量与全局变量的主要区别就在于可见性，静态局部变量只在其被声明的代码块中是可见的。

对某些必须在调用之间保持局部变量的值的子程序而言，静态局部变量是特别重要的。如果没有静态局部变量，则必须在这类函数中使用全局变量，由此也就打开了引入副作用的大门。使用静态局部变量最好的示例就是实现统计次数的功能，如代码清单 1-27 所示。

代码清单 1-27 静态局部变量使用示例

```

#include <stdio.h>
void count();
int main(void)
{
    int i=0;
    for (i = 0; i <= 5; i++)
    {
        count();
    }
}

```

```
    return 0;
}
void count()
{
    /* 声明一个静态局部变量 */
    static num = 0;
    num++;
    printf("%d\n", num);
}
```

在代码清单 1-27 中，我们通过 `count()` 函数里声明一个静态局部变量 `num` 来作为计数器。因为静态局部变量是在编译时赋初值的，且只赋初值一次，在程序运行时它已有初值。以后在每次调用函数时就不再重新赋初值，而是保留上次函数调用结束时的值。这样，`count()` 函数每次被调用的时候，静态局部变量 `num` 就会保持上一次调用的值，然后再执行自增运算，这样就实现了计数功能。同时，它又避免了使用全局变量。

通过上面的示例，我们可以得出静态局部变量一般的使用场景，如下所示：

- ❑ 需要保留函数上一次调用结束时的值。
- ❑ 如果初始化后，变量只会被引用而不会改变其值，则这时用静态局部变量比较方便，以免每次调用时重新赋值。

3. 默认初始化为 0

在静态数据区，内存中所有的字节默认值都是 `0x00`。静态变量与全局变量也一样，它们都存储在静态数据区中，因此其变量的值默认也为 0。演示示例如代码清单 1-28 所示。

代码清单 1-28 默认为 0 的演示示例

```
#include <stdio.h>
static int g_x;
int g_y;
int main(void)
{
    static int x;
    printf("g_x:%d\ng_y:%d\nx:%d", g_x, g_y, x);
    return 0;
}
```

代码清单 1-28 的运行结果如图 1-42 所示。



图 1-42 代码清单 1-28 的运行结果

建议 8-4：尽量少使用 register 变量

前面内容就已经阐述过，计算机中保存变量当前值的存储单元有两类：一类是内存，另一类是 CPU 的寄存器，而 register 变量则将其值存储到 CPU 的寄存器中。通常，寄存器变量比存储于内存的变量访问效率更高。但是，编译器并不一定会理睬 register 关键字，如果有太多的变量被声明为 register，它只会选取前几个实际存储于寄存器中，其余的就按普通变量进行处理。如果一个编译器自己具有一套寄存器优化方法，它也可能会忽略 register 关键字，其依据是由编译器决定哪些变量存储于寄存器中要比人脑决定更为合理一些。

在典型情况下，通常会希望把使用频率更高的那些变量声明为寄存器变量。在有些计算机中，如果把指针变量声明为寄存器变量，程序的效率将能得到提高，尤其是那些频繁执行间接访问操作的指针。你也可以把函数的形式参数声明为寄存器变量，这样编译器会在函数的起始位置生成指令，然后把这些值从内存复制到寄存器中。但是，完全有可能这个优化措施所节省的时间和空间的开销还抵不上复制这几个值所花的开销。

register 变量的使用示例如代码清单 1-29 所示。

代码清单 1-29 register 变量的使用示例

```
#include <stdio.h>
size_t fac(size_t n);
int main(void)
{
    size_t i;
    for(i=1;i<=5;i++)
    {
        printf("%d! = %d\n",i,fac(i));
    }
    return 0;
}
size_t fac(size_t n)
{
    register size_t result=1;
    register size_t i=1;
    for(i=1;i<=n;i++)
    {
        result=result*i;
    }
    return result;
}
```

在代码清单 1-29 中，我们在 fac 函数中实现了阶乘功能。因为 fac 函数中的变量 result 与 i 需要在 for 循环中反复调用，所以这里把它们定义成寄存器变量。其运行结果如图 1-43 所示。

寄存器变量的创建和销毁时间与自动变量相同，但它需要做一些额外的工作。在一个使用寄存器变量的函数返回之前，这些寄存器必须恢复先前所存储的值，确保调用者的寄存器变量未被破坏。许多机器使用运行时堆栈来完成这个任务。当函数开始执行时，它把需要使用的所有寄存器的内容都保存到内存中，当函数返回时，这些值会再复制到寄存器中。

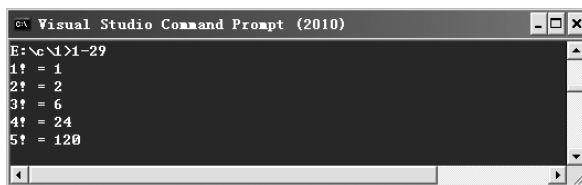


图 1-43 代码清单 1-29 的运行结果

值得注意的是，在许多机器的硬件实现中，并不会为寄存器指定地址。同样，某个特定的寄存器在不同的时刻所保存的值也不一定相同。基于这些原因，机器并不会向你提供寄存器变量的地址。因此，在 C 语言中，你不能够通过“&”操作符来访问 register 变量的地址。

最后，在使用 register 变量时，还需要注意如下几个方面：

- ❑ 只有局部自动变量和形式参数可以作为寄存器变量，否则将会导致无法编译。
- ❑ 一个计算机系统上的寄存器数目有限，不能定义任意多个寄存器变量。
- ❑ 局部静态变量不能定义为寄存器变量。
- ❑ register 变量一般只对整型和字符型数据有用。
- ❑ 不能使用取地址运算符“&”求寄存器变量的地址。

建议 9：尽量不要在可重入函数中使用静态（或全局）变量

前面介绍过，静态变量的存储方式与全局变量一样，都是静态存储方式。因此，在使用全局变量、静态变量的函数时，需要考虑重入问题。所谓的可重入函数是指函数可以由多于一个的任务并发使用，而不必担心数据错误。反之，不可重入函数不能由超过一个的任务所共享，除非能够确保函数的互斥性（或者使用信号量，或者在代码的关键部分禁用中断）。

来看一个不可重入函数的示例，代码如下所示：

```
size_t sum_index( size_t index )
{
    size_t i;
    static size_t sum=0;
    for (i = 1; i <= index; i++)
    {
        sum += i;
    }
    return sum;
}
```

上面的 sum_index 函数之所以是不可重入的，就是因为函数中使用了 static 变量。前面已经阐述过，静态局部变量是在编译时赋初值的，且只赋初值一次，在程序运行时它已有初值。以后在每次调用函数时不再重新赋初值，而是保留上次函数调用结束时的值。所以，这样的函数又被称为带“内部存储器”功能的函数。

函数 `sum_index` 的调用示例如下：

```
int main(void)
{
    printf("%d\n", sum_index(1));
    printf("%d\n", sum_index(2));
    printf("%d\n", sum_index(3));
}
```

运行结果如图 1-44 所示。

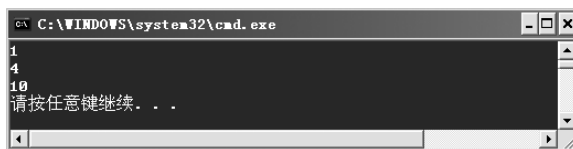


图 1-44 调用 `sum_index` 函数的运行结果

由于全局变量与静态变量一样，都是静态存储方式，因此，它同样可以导致不可重入函数，如下面的代码所示：

```
size_t g_sum = 0;
size_t sum_index( size_t index )
{
    size_t i;
    for (i = 1; i <= index; i++)
    {
        g_sum += i;
    }
    return g_sum;
}
```

因此，如果需要一个可重入的函数，那么一定要尽量避免使用 `static` 变量与全局变量，能不用则尽量不用。当然，有些时候在函数中是必须使用 `static` 变量的，比如当某函数的返回值为指针类型时，则必须以 `static` 局部变量的地址为返回值，若为 `auto` 类型，则返回为错指针。

如果我们需要将上面的函数修改为可重入的函数，其实也很简单，只要将声明 `sum` 变量中的 `static` 关键字去掉，将变量 `sum` 变为一个 `auto` 类型的变量，函数即可变为一个可重入的函数。如下面的代码所示：

```
size_t sum_index( size_t index )
{
    size_t i;
    size_t sum=0;
    for (i = 1; i <= index; i++)
    {
        sum += i;
    }
}
```



```
    return sum;
}
```

建议 10：尽量少使用全局变量

全局变量的作用前面已经介绍了许多，其主要作用就是增加函数间数据联系的渠道。由于同一源文件或多个源文件中的所有函数都能够引用全局变量的值，因此，如果在一个函数中改变全局变量的值，就会影响到其他的函数，相当于各个函数间有直接的传递通道。由于函数的调用只能带回一个返回值，因此有时可以利用全局变量来增加函数联系的渠道，从而使函数可以得到一个以上的返回值。

尽管如此，过多使用全局变量也会给我们带来许多的麻烦，主要表现为：

- ❑ 由于全局变量是静态存储方式，因此它在程序的全部执行过程中都会占用存储单元，而不是仅在需要时才开辟存储单元。
- ❑ 它使函数的通用性降低，因为函数在执行时依赖其所在的外部变量。在程序设计时，我们要求模块的功能单一，各模块之间的相互影响尽量少，而用全局变量很显然是不符合这个原则的。通常，我们都会要求把 C 程序中的函数做成一个封闭体，从而通过“实参 - 形参”的渠道来实现与外界的联系，这样的程序移植性好，可读性强。
- ❑ 使用的全局变量过多，会降低程序的清晰性，我们往往难以清楚地判断出每个瞬时各个外部变量的值。由于在各个函数执行时都可能改变外部变量的值，这就很容易导致程序出错。
- ❑ 如果在同一个源文件中，外部变量与局部变量同名，则在局部变量的作用范围内，外部变量会被“屏蔽”，即它将不起任何作用。

基于上面的原因，建议尽量减少全局变量的使用，同时建议你：

- ❑ 如果全局变量仅需要在单个源文件中访问，则可以将这个变量修改为静态全局变量，以降低模块间的耦合度。
- ❑ 若全局变量仅由单个函数访问，则可以将这个变量改为该函数的静态局部变量，以降低模块间的耦合度。
- ❑ 使用全局变量、静态全局变量与静态局部变量的函数时，需要考虑重入问题。

建议 11：尽量使用 `const` 声明值不会改变的变量

从字面上理解，`const` 是 `constant` 的缩写，是恒定不变的意思，也翻译为常量、常数等。正是因为这一点，看到 `const` 关键字，很多人就认为被 `const` 修饰的值是常量。其实，在 C 语言中，关键字 `const` 的功能非常强大，它不仅可以用来修饰普通变量、数组变量与指针变量等，还可以用来修饰函数的参数、返回值与函数本身。这些将会在后面的章节逐一详细讲

解，本节只讨论使用 `const` 来修饰变量的情况。

对变量来说，`const` 关键字可以限定一个变量的值不允许被改变，从而保护被修饰的东西，防止意外修改发生，这在一定程度上可以提高程序的安全性和可靠性。但是要正确地使用 `const` 变量，我们必须弄清以下几点。

1. `const` 变量是只读变量，不是常量

提到 `const` 变量，总有人喜欢把它和常量混为一谈。其实 `const` 变量不是常量，准确地说它应该是只读的变量。为了让大家更好地区分这两个概念，我们来看下面的例子：

```
const int array_num=10;
int arr[array_num]={1,2,3,4,5,6,7,8,9};
```

在标准的 ANSI C 中，上面的这种写法是错误的，因为数组的大小应该是个常量，而 `const int array_num` 只是一个只读变量。虽然常量也是只读的、不可修改的（例如 10、‘C’等），但只读变量却不等于常量，只读变量被编译器限定为不能够被修改。

实际上，根据编译过程及内存分配来看，这种用法本来应该是合理的，只是 ANSI C 对数组的规定限制了它。而在 C++ 中，上面的这种写法确实是正确的。

2. 确保变量的值不被修改

可以说，`const` 关键字的最大作用就是确保变量赋初值之后，其值不会被任何程序修改。如下面的示例代码所示：

```
const int array_num=10;
array_num++;           // 错误，不可以被修改
```

很显然，语句“`array_num++`”是错误的。同样，对于外部变量，`const` 一样可以确保其变量的值不被修改，如下面的代码所示：

```
extern const int i;      // 正确的引用
extern const int i="10"; // 错误，不可以被再次赋值
```

尽管如此，网上仍然流传着一种修改 `const` 变量的说法，如代码清单 1-30 所示。

代码清单 1-30 `const` 变量使用示例

```
#include <stdio.h>
int main(void)
{
    const int i1=10;
    int *p1=(int *)&i1;
    int *p2=p1;
    *p1=100;
    printf("%d %d %d\n",i1,*p1,*p2);
    printf("%p %p %p\n\n",&i1,p1,p2);
    return 0;
}
```

有人说，这样 `const` 变量就能被修改了，但运行上面的程序，`const` 变量的值根本没有改变，代码清单 1-30 的运行结果如图 1-45 所示。

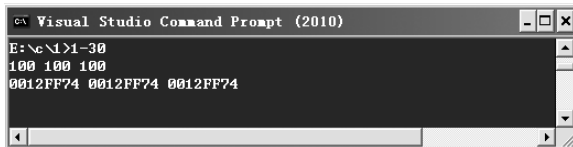


图 1-45 代码清单 1-30 的运行结果

在图 1-45 中，虽然 “`i1,*p1,*p2`” 所得的值不同，但 “`&i1,p1,p2`” 却是同一个地址，既然是同一地址，为什么输出不一样的值？

为了进一步查看其原因，我们在代码清单 1-30 中添加如下一条语句：

```
printf("%d\n",*((int *)0x0012FF74));
```

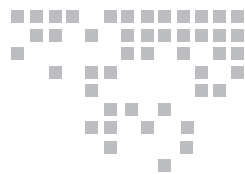
这样就可以直接输出地址所存的值了。得到的结果在我们的预料之中，为 “100”。或许大家认为这样就改变了 `const` 变量的值。

其实，“`int *p1=(int *)&i1;`” 表面上给了它们相同的指针，但是 `const` 变量的值是保存在数据段（只读）的，通过地址 `0x0012FF74` 查内存文件可得知其属于堆栈段。也就是说，虽然地址相同，但 `const` 变量读取的是数据段，而通过指针读取到的是堆栈段。

由此可见，`const` 提供了一种保护机制，能在编译阶段阻止其变量的值被修改。但它并不能完全防止在程序的内部（甚至是外部）来修改这个值。也就是说，`const` 变量是只读变量，既然是变量，那么就可以获取其地址，然后修改其值。因此，当汇编成二进制之后，这种保护机制就不复存在了。

3. 节省空间，避免不必要的内存分配

使用 `const` 变量除了可以确保变量值不被修改之外，同时它还可以节省存储空间，避免不必要的内存分配。通常，编译器并不为普通 `const` 只读变量分配存储空间，而是将它们保存在符号表中，这使得它成为一个编译期间的值，没有了存储与读内存的操作，从而提高效率。



保持严谨的程序设计， 一切从表达式开始做起

C 语言的表达式遵循一般代数规则，由常量、变量、函数和运算符构成。相对于其他计算机语言来说，C 语言表达式的功能更强大，语法更灵活，种类也比较繁多。我们可以根据运算符把它们简单地划分为算术表达式、关系表达式、逻辑表达式、赋值表达式、条件表达式和逗号表达式。

建议 12：尽量减少使用除法运算与求模运算

对计算机来说，除法与求模是整数算术运算中最复杂的运算。相对其他运算（如加法与减法）来说，这两种算法的执行速度非常慢。例如，ARM 硬件上不支持除法指令，编译器调用 C 库函数来实现除法运算。直接利用 C 库函数中的标准整数除法程序要花费 20 ~ 100 个周期，消耗较多资源。

在非嵌入式领域，因为 CPU 运算速度快、存储器容量大，所以执行除法运算和求模运算消耗的这些资源对计算机来说不算什么。但是在嵌入式领域，消耗大量资源带来的影响不言而喻。因此，从理论上讲，我们应该在程序表达式中尽量减少对除法运算与求模运算的使用，尽量使用其他方法来代替除法与求模运算。例如，对于下面的示例代码：

```
if (x/y>z)
{
    // ...
}
```

我们可以将其修改成如下形式：

```
if (( (y>0) && (x>y*z) ) || ( (y<0) && (x<y*z) ))
{
    // ...
}
```

这样就简单地避免了一些除法运算。同时，也可以在表达式中通过合并除法的方式来减少除法运算，下面通过示例来讲解。对于如下代码：

```
double x=a/b/c;
double y=a/b+c/b;
```

根据数学结合原则，上面的代码可以通过合并的方式减少代码中的除法运算，修改后的代码如下：

```
double x=a/(b*c);
double y=(a+c)/b;
```

同样，对于求模运算，也可以采用相应的方法来代替，如下面的示例代码：

```
a=a%8;
// 可以修改为：
a=a&7;
```

对于下面的表达式：

```
x=(x+y)%z;
```

可以通过如下方式来避免使用模操作符：

```
x+=y;
while(x>=z)
{
    x-=z;
}
```

通过上面的阐述，相信大家对如何减少使用除法与模运算有了初步了解。下面将详细讨论如何优化除法运算与求模运算。

建议 12-1：用倒数相乘来实现除法运算

何为倒数相乘？其实很简单，它的核心思想就是利用乘法来代替实现除法运算。例如，在 IA-32 处理器中，乘法指令的运算速度比除法指令要快 4 ~ 6 倍。因此，在某些情况下尽量使用乘法指令来代替除法指令。

那么，我们该如何利用乘法来代替实现除法运算呢？原理就是被除数乘以除数的倒数，用公式表现为：

$$x/y=x*(1/y)$$

例如，计算 10/5，可以根据公式 $x/y=x*(1/y)$ 这样来计算：

$$10/5=10*(1/5)=10*0.2=2$$

在实际应用中，一些编译器也正是基于这个原理才得以将除法运算转换为乘法运算的。现在来看一个除法运算示例，如代码清单 2-1 所示。

代码清单 2-1 除法运算示例

```
#include <stdio.h>
int main(void)
{
    int x = 3/2;
    float y = 3.0/2.0;
    printf("3/2 = %d\r\n3.0/2.0 = %1.1f\r\n",x,y);
    return 0;
}
```

代码清单 2-1 的运行结果如图 2-1 所示。

通过代码清单 2-1 可以看出，很明显没能充分考虑到浮点类型。另外，在 C 语言中，一般情况下 1 除以任何数其结果皆为 0。那么怎样才能解决这个问题呢？编译器采用了一种称为“定点运算”（fixed-point arithmetic）的方法。

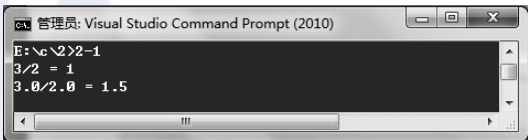


图 2-1 代码清单 2-1 的运行结果

那么何为定点运算，定点运算有什么特点呢？

前面已经阐述过，由于计算机表示实数时为了在固定位数内能表示尽量精确的实数值，分配给表示小数部分的位数并不是固定的，也就是说“小数点是浮动的”，因此计算机表示的实数数据类型也称为浮点数。

相对于“小数点是浮动的”来讲，定点运算根据字面意思来理解就是“小数点是固定的”。有了定点运算，表示小数时不再用阶码（exponent component，即小数点在浮点数据类型中的位置），而是要保持小数点的位置固定不变。这和硬件浮点数机制截然不同，硬件浮点数机制是由硬件负责向整数部分和小数部分分配可用的位数。有了这种机制，浮点数就可以表示很大范围的数——从极小的数（在 0 ~ 1 的实数）到极大的数（在小数点前有数十个 0）。这种小数的定点表示法有很多优点，尤其能极大地提高效率。当然，作为代价，同样也必须承受随之而来的精度上的损失。

对于定点数表示法（fixed-point），相信大家并不陌生。所谓定点格式，即约定机器中所有数据的小数点位置是固定不变的。在计算机中通常采用两种简单的约定：将小数点的位置固定在数据的最高位之前（即定点小数），或者固定在最低位之后（即定点整数）。

其中，定点小数是纯小数，约定的小数点位置在符号位之后、有效数值部分的最高位之前。若数据 x 的形式为 $x=x_0x_1x_2\cdots x_n$ （其中 x_0 为符号位， x_1, \cdots, x_n 是数值的有效部分，也称

为尾数， x_1 为最高有效位)，则在计算机中的表示形式为：

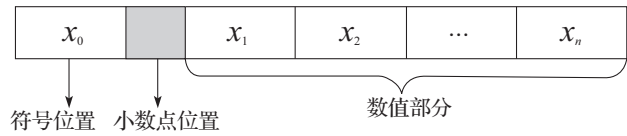


图 2-2 定点小数的表现形式

一般说来，如果最末位 $x_n=1$ ，前面各位都为 0，则数的绝对值最小，即 $|x|_{\min}=2^{-n}$ ；如果各位均为 1，则数的绝对值最大，即 $|x|_{\max}=1-2^{-n}$ 。因此定点小数的表示范围是：

$$2^{-n} \leq |x| \leq 1-2^{-n}$$

定点整数是纯整数，约定的小数点位置在有效数值部分最低位之后。若数据 x 的形式为 $x = x_0x_1x_2\cdots x_n$ （其中 x_0 为符号位， x_1, \cdots, x_n 是尾数， x_n 为最低有效位），则在计算机中的表示形式为：

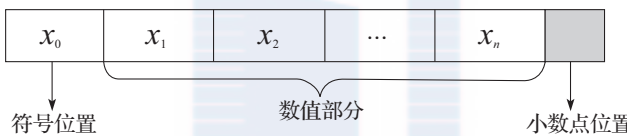


图 2-3 定点整数的表现形式

由此可知，定点整数的表示范围是：

$$1 \leq |x| \leq 2^n-1$$

当数据小于定点数能表示的最小值时，计算机将它作 0 处理，称为下溢；当数据大于定点数能表示的最大值时，计算机将无法表示，称为上溢，上溢和下溢统称为溢出。

当计算机采用定点数表示时，对于既有整数又有小数的原始数据，需要设定一个比例因子，数据按该比例缩小成定点小数或扩大成定点整数再参加运算。在运算结果中，根据比例因子，将数据还原成实际数值。若比例因子选择不当，往往会使运算结果产生溢出或降低数据的有效精度。

建议 12-2：使用牛顿迭代法求除数的倒数

在上一小节，我们阐述了如何使用倒数相乘（ $x/y=x*(1/y)$ ）的方法来实现除法运算。然而，对于如何能够快速有效地取倒数，牛顿迭代法（Newton's method）是最佳方案。

对于牛顿迭代法，相信学过高等数学的读者并不陌生，它又称为牛顿-拉夫逊方法（Newton-Raphson method），它是牛顿在 17 世纪提出的一种在实数域和复数域上近似求解方程的方法，它将非线性方程线性化，从而得到迭代序列的一种方法。

对于方程 $f(x)=0$ ，设 x_0 为它的一个近似根，则函数 $f(x)$ 在 x_0 附近截断高次项可用一阶泰勒多项式展开为如下形式：

$$f(x)=f(x_0)+f'(x_0)(x-x_0) \quad (1)$$

这样,由式(1)我们可以将 $f(x)=0$ 转化为如下形式:

$$f(x_0)+f'(x_0)(x-x_0)=0 \quad (2)$$

在这里,我们设 $f'(x) \neq 0$,则有:

$$x=x_0-\frac{f(x_0)}{f'(x_0)} \quad (3)$$

取 x 作为原方程新的近似根 x_1 ,再代入方程,如此反复,于是就产生了迭代公式:

$$x_{n+1}=x_n-\frac{f(x_n)}{f'(x_n)} \quad (4)$$

有了迭代公式(4)之后,现在我们继续来看如何用牛顿迭代公式来求倒数,即求除数 a 的倒数 $1/a$ 。

这里我们设 $f(x)=\frac{1}{x}-a$,式中 x 为 a 的倒数,方程 $f(x)=0$ 为一非线性方程。现在把 $f(x)=0$ 代入牛顿迭代序列式(4)中,就可以得出求倒数的公式,如下所示:

$$\begin{aligned} x_{n+1} &= x_n - \frac{f(x_n)}{f'(x_n)} \\ &= x_n - \frac{\frac{1}{x_n} - a}{\left(\frac{1}{x_n} - a\right)'} \\ &= x_n \times (2.0 - a \times x_n) \end{aligned} \quad (5)$$

在式(5)中, x_n 为第 n 次迭代的近似根。

如式(5)所示,用牛顿迭代法求倒数,每次迭代需要一次减法与两次乘法,所用的迭代次数决定最终的计算速度和精度。迭代次数越多,则精度越高。但迭代次数越多,速度也越慢,因此实际运用时应综合考虑速度和精度两方面的因素,选择合适的迭代次数。

其实,牛顿迭代法在程序中应用得非常广泛,如最常用的开方、开方求倒数等。在Quake III源码中,在game/code/q_math.c文件中就有一个函数Q_rsqrt,它的作用是将一个数开平方后取倒,其运行效率也非常高。如代码清单2-2所示。

代码清单 2-2 Q_rsqrt 函数的实现

```
float Q_rsqrt( float number )
{
    long i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y = number;
    i = * ( long * ) &y;
    i = 0x5f3759df - ( i >> 1 );
    y = * ( float * ) &i;
```

```

// 第一次迭代
y = y * ( threehalfs - ( x2 * y * y ) );
// 第二次迭代
//y = y * ( threehalfs - ( x2 * y * y ) );
return y;
}

```

从代码清单 2-2 中可以看出，程序首先猜测出一个接近 $1.0/\sqrt{\text{number}}$ 的近似值，然后两次使用牛顿迭代法进行迭代（实际只需要使用一次）。这里需要特别注意的是 `0x5f3759df` 这个值，因为通过执行语句“`0x5f3759df - (i >> 1)`”，得出的值出人意料地接近 $1/\sqrt{\text{number}}$ 的值，因此，我们只需要一次迭代就可以求得近似解，或许这就是数学的神奇。

建议 12-3：用减法运算来实现整数除法运算

我们知道，减法运算比除法运算要快得多。因此，对整数除法运算来说，如果知道被除数是除数很小的倍数，那么可以使用减法运算来代替除法运算。例如，对于下面的示例代码：

```

unsigned int x=300;
unsigned int y=100;
unsigned int z=x/y;

```

我们可以将“`z=x/y`”表达式修改成如下形式：

```

unsigned int x=300;
unsigned int y=100;
unsigned int z=0;
while (x>=y)
{
    x-=y;
    ++z;
}

```

这里使用减法来代替除法运算，虽然代码看起来不是很直观，但是在运行效率上确实要快许多。当然，具体效率也要取决于被除数与除数的倍数。如果倍数比较大，那么相应的循环次数就会增多，采取这种方法就得不偿失了。

建议 12-4：用移位运算实现乘除法运算

用移位运算来实现乘除法运算的方法，相信大家并不陌生，实际上有很多 C 编译器都能够自动地做好这个优化。通常，如果需要乘以或除以 2^n ，都可以用移位的方法代替。

例如：

```

a=a*2;
b=b/2;

```


可以修改为如下形式:

```
a=a<<1;
b=b>>1;
```

其中,除以2等价于右移1位,乘以2等价于左移1位。同理,除以4等价于右移2位,乘以4等价于左移2位;除以8等价于右移3位,乘以8等价于左移3位,以此类推。

其实,利用上面的原理,只要是乘以或除以一个整数,均可以用移位运算的方法来得到结果,例如:

```
a=a*5;
```

可以将其分解为 $a*(4+1)$, 即 $a*4+a*1$ 。由此,我们就可以很简单地得到下面的程序表达式:

```
a=(a<<2)+a
```

建议 12-5: 尽量将浮点除法转化为相应的整数除法运算

有时候,如果不能够在代码中避免除法运算,那么尽量使除数和被除数是无符号类型的整数。实际上,有符号的除法运算执行起来比无符号的除法运算更加慢,因为有符号的除法运算要先取得除数和被除数的绝对值,再调用无符号除法运算,最后再确定结果的符号。

同时,对于浮点除法运算,可以先将浮点除法运算转化为相应的整数除法运算,最后对结果进行相应处理。例如,可以将浮点除法运算的分子和分母同时放大相同的倍数,就可以将浮点除法运算转换成相同功能的整数除法运算。

建议 13: 保证除法和求模运算不会导致除零错误

我们知道,除法运算与求模运算都可能会导致除零错误。因此,我们在做除法与求模运算的时候应该避免除数为零的情况发生,示例代码如下:

```
unsigned int x;
unsigned int y;
unsigned int result;
/* 初始 x,y,result */
if(y==0)
{
}
else
{
    result=x/y;
}
```

除此之外,当被除数等于有符号整数类型的最小值(负值)且除数等于-1时,也可能会

产生溢出。这种情况应该尽量避免，示例代码如下：

```
signed long x;
signed long y;
signed long result;
/* 初始 x,y,result*/
if((y==0) || ((x==LONG_MIN) && (y== -1)))
{
}
else
{
    result=x/y;
}
```

同理，求模运算也应该采用相同的方法来避免除零错误的发生，示例代码如下：

```
signed long x;
signed long y;
signed long result;
/* 初始 x,y,result*/
if((y==0) || ((x==LONG_MIN) && (y== -1)))
{
}
else
{
    result=x%y;
}
```

建议 14：适当地使用位操作来提高计算效率

我们知道，程序中的所有数据在计算机内存中都是以二进制的形式进行存储的，数据的位是可以操作的最小数据单位，位操作就是直接对整数在内存中的二进制位进行操作。因此，在理论上，我们可以通过“位运算”来完成所有的运算和操作，从而有效地提高程序运行的效率。

C 语言中提供了 &（与）、|（或）、^（异或）、~（取反）、>>（右移）、<<（左移）6 种位操作符。我们可以在程序中合理地使用这些位操作符号来提高程序的运行效率，例如，建议 12-4 中介绍的利用移位运算来提高乘法与求模运算。对于下面的示例代码：

```
int x=0;
int y=0;
x = 257 /8;
y = 456 % 32;
```

我们可以通过位操作符将其修改成如下形式：

```
int x=0;
int y=0;
```

```
x = 257 >>3;
y = 456 - (456 >> 4 << 4);
```

这样就可以使程序在性能上得到一定提升。

建议 14-1: 尽量避免对未知的有符号数执行位操作

在 C 语言中,如果在未知的有符号数上执行位操作,很可能会导致缓冲区溢出,从而在某些情况下导致攻击者执行任意代码,同时,还可能会出现出乎意料的行为或编译器定义的行为。

下面来看一个简单的示例,如代码清单 2-3 所示。

代码清单 2-3 在未知的有符号数上执行位操作示例

```
#include <stdio.h>
int main (void)
{
    int x=0;
    int y=0x80000000;
    char buf[sizeof("128")];
    x=sprintf(buf,"%u",y>>24);
    if(x==-1||x>=sizeof(buf))
    {
        // 错误处理
    }
    printf(buf);
    return 0;
}
```

在代码清单 2-3 中, $y \gg 24$ 的执行结果为 4294967168, 而 $\text{sizeof}(\text{buf})$ 的结果为 4。当我们将 $y \gg 24$ 的结果值转换为字符串“4294967168”时,超出了 buf 范围,所以结果值无法完全存储在 buf 中。因此,在执行语句“ $x=\text{sprintf}(\text{buf},\text{"\%u"},y \gg 24)$ ”时, sprintf 方法在进行写操作时就会越过 buf 的边界,从而产生缓冲区溢出。

如果在编译器 VC++ 中执行这段程序,将会产生如图 2-4 所示的错误报告。

在 C99 中,要修正这样的错误,最好利用 snprintf 方法来代替 sprintf 方法。因为 snprintf 方法最多从源串中复制 $n-1$ 个字符到目标串中,然后再从后面加一个 0。因此,如果目标串的大小为 n ,将不会产生溢出。

当然,如果将变量 y 声明成为无符号

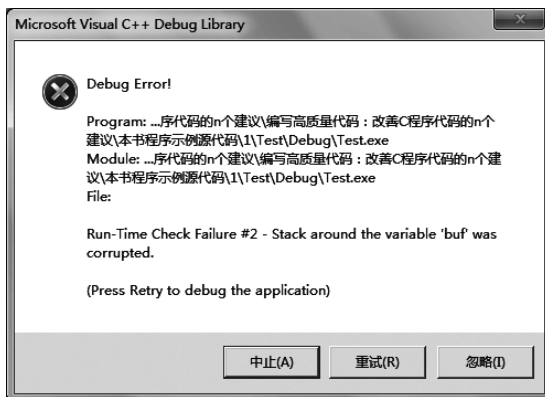


图 2-4 在 VC++ 中执行代码清单 2-3 的错误报告

类型，那么这种缓冲区溢出错误将不会发生，如代码清单 2-4 所示。

代码清单 2-4 代码清单 2-3 的改进示例

```
#include <stdio.h>
int main (void)
{
    int x=0;
    unsigned int y=0x80000000;
    char buf[sizeof("128")];
    x=sprintf(buf, "%u", y>>24);
    if(x== -1 || x>=sizeof(buf))
    {
        // 错误处理
    }
    printf(buf);
    return 0;
}
```

建议 14-2：在右移中合理地选择 0 或符号位来填充空出的位

在右移运算中，空出的位用 0 还是符号位进行填充呢？

其实答案由具体的 C 语言编译器实现来决定。在通常情况下，如果要进行移位的操作数是无符号类型的，那么空出的位将用 0 进行填充；如果要进行移位的操作数是有符号类型的，则 C 语言编译器实现既可选择 0 来进行填充，也可选择符号位进行填充。

因此，如果很关心一个右移运算中的空位，那么可以使用 `unsigned` 修饰符来声明变量，这样空位都会被设置为 0。同时，如果一个程序采用了有符号数的右移位操作，那么它就是不可移植的。

建议 14-3：移位的数量必须大于等于 0 且小于操作数的位数

如果被移位的操作数的长度为 n ，那么移位的数量必须大于等于 0 且小于 n 。因此，在一次单独的操作中不可能将所有的位从变量中移出。例如，一个 `int` 型的整数是 32 位，并且 n 是一个 `int` 型整数，那么 $n << 31$ 和 $n << 0$ 是合法的，但 $n << 32$ 和 $n << -1$ 是不合法的。因此，我们在进行移位运算的时候必须做相关测试。示例代码如下所示：

```
unsigned int x;
unsigned int y;
unsigned int result;
/* 初始 x,y,result*/
if(y>=sizeof(unsigned int) * CHAR_BIT)
{
    // 错误处理
}
else
```

```
{
    result=x>>y;
}
```

这里还需要说明的是,对于变量 x 与 y , C99 规定:

- 对于 $x \ll y$, 如果 x 是有符号类型的非负值, 并且 $x \ll y$ 的移位结果 $x * 2^y$ 可以用结果类型表示, 那么这个表达式就是结果值, 否则, 其行为是未定义的; 如果 x 是无符号类型, 则 $x \ll y$ 的移位结果为 $x * 2^y$, 是根据“结果类型可以表达的最大值加 1”进行求模运算得到的结果。需要注意的是, 尽管在 C99 中指定了无符号整数的取模行为, 无符号整数溢出还是常常导致出乎意料的值以及因此产生的潜在安全风险。
- 对于 $x \gg y$, 如果 x 是无符号类型或非负值的有符号类型, 那么 $x \gg y$ 的移位结果为 $x / 2^y$ 的商的整数部分; 如果 x 是有符号类型的负值, 那么 $x \gg y$ 的移位结果是由编译器所定义的。因此, 对一个带符号整数进行右移运算和将它除以 2 的某次幂不一定是等价的。要证明这一点很容易, 考虑 $(-1) \gg 1$ 的值, 它的执行结果不可能为 0, 而在大多数 C 语言编译器中 $(-1)/2$ 的结果都是 0。

建议 14-4: 尽量避免在同一个数据上执行位操作与算术运算

虽然位操作在很大程度上可以提高程序的执行效率, 但在同一个变量上执行位操作和算术运算会模糊程序员的意图, 削弱代码的可移植性与可读性, 还会导致安全审核员或代码维护人员难以确定应该执行什么检查以消除安全缺陷, 保证数据的完整性。示例代码如下:

```
unsigned int x=10;
x+=(x<<3)-5;
```

虽然上面的代码是一条合法的优化语句, 但是它的确严重地破坏了程序的可读性。因此, 建议采用下面的方式来书写代码:

```
unsigned int x=10;
x=x*9-5;
```

或许这时候有读者会问, 这样写代码不就降低了程序的执行效率吗?

其实不然, 有些优化编译器会比我们做得更好。以整数乘法为例, 如果目标系统有乘法指令, 硬件的乘法比自己用移位操作实现的乘法要快得多; 如果目标系统没有乘法指令, 编译器会自动用移位等操作来优化乘法, 示例代码如下:

```
unsigned int x=8;
unsigned int y=x*4;
unsigned int z=x/2;
```

上面的代码在 Microsoft Visual Studio 2010 集成开发环境 VC++ 的 Debug 模式下将生成如下汇编代码:

```
unsigned int x=8;
```

```

00DB139E  mov     dword ptr [x],8
          unsigned int y=x*4;
00DB13A5  mov     eax,dword ptr [x]
00DB13A8  shl     eax,2
00DB13AB  mov     dword ptr [y],eax
          unsigned int z=x/2;
00DB13AE  mov     eax,dword ptr [x]
00DB13B1  shr     eax,1
00DB13B3  mov     dword ptr [z],eax

```

从上面的汇编代码可以看出，编译器会自动在汇编代码中用移位操作来优化乘除法运算。因此，完全没有必要用手工进行这种优化，编译器会自动完成。我们应该把精力放在改进程序的算法上，一个好的算法可以使程序运行效率大大提高。当然，如果除数为 2 的幂，那么在进行除法运算时可以适当地采用移位算法来实现乘法。

除此之外，采用移位操作还需要注意不要超过该数据类型的精度范围（数据范围），示例代码如下：

```

#include <stdio.h>
int main (void)
{
    int x=-2147483647;
    x=x<<1;
    printf("%d \n",x);
    return 0;
}

```

在上面的代码中就要注意精度问题，在 32 位系统中，int 类型占 4 个字节，精度范围为“-2147483647 ~ 2147483647”。其中，数据“-2147483647”的原码为“11111111111111111111111111111111”，补码为“10000000000000000000000000000001”。现在将“10000000000000000000000000000001”左移 1 位，最高位的 1 没有了，最低位左移一位，得到的结果为 2。

建议 15：避免操作符混淆

在 C 语言中，有些操作符很相似，比如 = 与 ==、| 与 ||、& 与 && 等。在使用这些操作符时，一不小心就很容易造成混淆，给程序带来不必要的错误。因此，在编写代码时应该特别注意这些容易混淆的操作符，熟练地掌握它们之间的区别与用法。

建议 15-1：避免“=”与“==”混淆

在 C 语言中，最容易产生混淆的操作符要属“=”与“==”。其中，“=”并不等于符号，而是赋值操作符，如 x=3。除此之外，还可以在一个语句中向多个变量赋同一个值，即多重赋值。例如，在下面代码中把 0 同时赋给 x、y 与 z。

```
x = y = z = 0
```

相对于只有一个等号的赋值操作符,关系操作符中的等于操作符采用两个等号“==”来表示。正因如此,导致了一个潜在的问题:出于习惯,我们可能经常将需要等于操作符的地方写成赋值操作符,如下面的代码:

```
int x=10;
int y=1;
if(x=y)
{
    /* 处理代码 */
}
```

在上面的代码中,if语句看起来好像是要检查变量x是否等于变量y。实际上并非如此,此时if语句将变量y的值赋给变量x并检查结果是否非零。因此,虽然这里的x不等于y,但是y的值为1,if语句还是会返回真。

当然,当确实需要先对一个变量进行赋值之后再检查变量是否非零时,可以考虑显式给出比较符。示例代码如下:

```
int x=10;
int y=1;
if((x=y)!=0)
{
    /* 处理代码 */
}
```

这样,程序的可读性就得到了很大提高。

上面的示例代码详细地阐述了将等于操作符“==”误写成赋值操作符“=”所带来的严重后果。同理,将赋值操作符“=”误写成等于操作符“==”也会带来非常严重的后果。示例代码如下:

```
int x=0;
int y=-1;
if((x==y)<0)
{
    printf("y<0\n");
}
```

在上面的代码中,if语句的本意是将变量y的值赋给变量x,然后再判断变量x的值是否小于0。如果变量x的值小于0,就执行语句printf("y<0\n")。由于错误地将赋值操作符“=”误写成等于操作符“==”,所以无论变量y为何值,都不会执行语句printf("y<0\n")。原因是等于操作符“==”的结果只能是0或1,永远不会小于0。

除此之外,为了防止将等于操作符“==”误写成赋值操作符“=”,还可以在代码中采用如下形式:

```
int x=0;
```

```

if(0==x)
{
}

```

这样，就可以在一定程度上避免误写的发生。

建议 15-2：避免 “|” 与 “||” 混淆

在 C 语言中，“||” 是逻辑操作符（或），它的操作数是布尔型，即只有 “0”（表示 false）和 “1”（表示 true）两个数值。C 语言规定，在逻辑运算中，所有非 0 的数值都被看成 1 处理。

而 “|” 是位操作符（或），其操作数是位序列。位序列可以是字符型、整型与长短整型等（通常情况下选择无符号整型）。在位运算中，相应的位之间进行逻辑运算，因此，从逻辑上讲，位运算过程包含多个逻辑运算过程。

下面通过代码清单 2-5 来了解两者之间的区别。

代码清单 2-5 “|” 与 “||” 运算符操作示例

```

#include <stdio.h>
int main (void)
{
    unsigned int x = 0x1101;
    unsigned int y = 0x1100;
    /* 逻辑操作 */
    printf("sizeof(x || y): %d\n", sizeof(x || y));
    if(x||y)
    {
        printf("x || y : %d(True) \n", x||y);
    }
    else
    {
        printf("x || y : %d(False) \n", x||y);
    }
    /* 位操作 */
    printf("sizeof(x | y): %d\n", sizeof(x | y));
    printf("x | y : %x \n", x|y);
    return 0;
}

```

在代码清单 2-5 中，因为变量 x 与变量 y 都不为 0，所以执行语句 if(x||y) 返回 1。而当执行 x|y（即 1101|1100）时，相应的位之间逐一位地进行逻辑运算（或），因此所得到的结果为 1101。代码清单 2-5 的执行结果如图 2-5 所示。



图 2-5 在 VC++ 中执行代码清单 2-5 的运行结果

建议 15-3: 避免 “&” 与 “&&” 混淆

同建议 15-2 相似, “&&” 是逻辑操作符 (与), 它的操作数是布尔型; 而 “&” 是位操作符 (与), 其操作数是位序列。它们之间的区别如代码清单 2-6 所示。

代码清单 2-6 “&” 与 “&&” 运算符操作示例

```
#include <stdio.h>
int main (void)
{
    unsigned int x = 0x1101;
    unsigned int y = 0x1100;
    /* 逻辑操作 */
    printf("sizeof(x && y): %d\n", sizeof(x && y));
    if(x&& y)
    {
        printf("x && y : %d(True) \n", x&&y);
    }
    else
    {
        printf("x && y : %d(False) \n", x&&y);
    }
    /* 位操作 */
    printf("sizeof(x & y): %d\n", sizeof(x & y));
    printf("x & y : %x \n", x&y);
    return 0;
}
```

在代码清单 2-6 中, 因为变量 x 与变量 y 都不为 0, 所以执行语句 “if(x&&y)” 时返回 1。而当执行 “x&y” (即 1101&1100) 时, 相应的位之间逐一进行逻辑运算 (与), 因此所得到的结果为 1100。代码清单 2-6 的执行结果如图 2-6 所示。

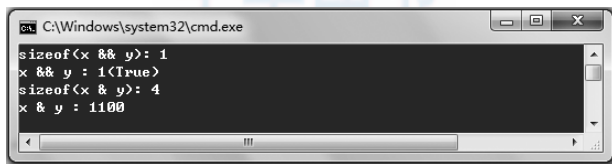


图 2-6 在 VC++ 中执行代码清单 2-6 的运行结果

建议 16: 表达式的设计应该兼顾效率与可读性

C 语言中的表达式相对其他语言来讲要复杂得多, 本节将重点讨论运算符的优先级及设计表达式的相关注意事项。

建议 16-1: 尽量使用复合赋值运算符

C 语言不但提供了最基本的赋值运算符 “=”, 而且为了简化程序并提高编译效率, 还

允许在赋值运算符“=”之前加上其他运算符，这样就构成了复合赋值运算符（即 /=、*=、%=、+=、-=、<<=、>>=、&=、^= 与 |=）。示例代码如下：

```
a/=b;      /* 等价于 a=a/b; */
a*=b;      /* 等价于 a=a*b; */
a%=b;      /* 等价于 a=a%b; */
a+=b;      /* 等价于 a=a+b; */
a-=b;      /* 等价于 a=a-b; */
a<<=b;     /* 等价于 a=a<<b; */
a>>=b;     /* 等价于 a=a>>b; */
a&=b;      /* 等价于 a=a&b; */
a^=b;      /* 等价于 a=a^b; */
a|=b;      /* 等价于 a=a|b; */
a+=2+1;    /* 等价于 a=a+(2+1); */
a/=2*10-5; /* 等价于 a=a/(2*10-5); */
```

对于表达式 a/b 与 $a=a/b$ ，有读者会问，它们两者之间究竟有什么区别呢？

答案很简单，对于 $a=a/b$ ， a 需要求值两次；而对于 $a/=b$ ， a 仅需要求值一次。一般来说，这种区别对程序的运行没有多大影响。编译器一般会进行优化，使两种表达式的运行效果一样。

当然，也有例外的情况，这时编译器无法进行优化。当表达式作为函数的返回值时，函数就会被调用两次，例如：

```
int f(int x)
{
    return x;
}
int main (void)
{
    int a[10] = {0};
    int x = 2;
    a[f(x) + 1] += 1;
    a[f(x) + 1] = a[f(x) + 1] + 1;
    return 0;
}
```

在上面的代码中，由于语句“ $a[f(x) + 1] = a[f(x) + 1] + 1$ ”是调用函数，编译器并不能确定每次函数调用都返回相同的值，所以编译器对此也无能为力，无法进行优化，因此只好乖乖地调用两次函数；而语句“ $a[f(x) + 1] += 1$ ”则只调用一次函数。

上面的代码在 Microsoft Visual Studio 2010 集成开发环境 VC++ 的 Debug 模式下将生成如下汇编代码：

```
    a[f(x) + 1] += 1;
013F3763  mov     eax,dword ptr [ebp-3Ch]
013F3766  push    eax
013F3767  call    f (13F11DBh)
```

```

013F376C add     esp,4
013F376F lea     ecx,[ebp+eax*4-2Ch]
013F3773 mov     dword ptr [ebp-104h],ecx
013F3779 mov     edx,dword ptr [ebp-104h]
013F377F mov     eax,dword ptr [edx]
013F3781 add     eax,1
013F3784 mov     ecx,dword ptr [ebp-104h]
013F378A mov     dword ptr [ecx],eax
    a[f(x) + 1] = a[f(x) +1] +1;
013F378C mov     eax,dword ptr [ebp-3Ch]
013F378F push    eax
013F3790 call    f (13F11DBh)
013F3795 add     esp,4
013F3798 mov     esi,dword ptr [ebp+eax*4-2Ch]
013F379C add     esi,1
013F379F mov     ecx,dword ptr [ebp-3Ch]
013F37A2 push    ecx
013F37A3 call    f (13F11DBh)
013F37A8 add     esp,4
013F37AB mov     dword ptr [ebp+eax*4-2Ch],esi

```

从上面的汇编代码中可以很清楚地看到两者之间的区别,使用普通的赋值运算符会加大程序的开销,从而使程序的效率降低。因此,如果能确定表达式中不含具有副作用的元素(如上面的函数 f),那么应该尽量使用复合赋值运算符来代替普通的赋值运算符,不过一定要注意程序的可读性。

建议 16-2: 尽量避免编写多用途的、太复杂的复合表达式

C 语言中的复合表达式是指如 $a = b = c = 0$ 这样的表达式,它不仅书写简洁,还可以提高编译效率,所以在专业的 C 程序中经常可以看到。接下来看这样一个例子:

```

int a=10;
a+=a--a*=a;
printf("%d\n",a);

```

有过面试经历的读者看上面的代码应该比较眼熟,笔者也曾经见过多家企业将本题作为面试题,那么 a 的值究竟应该是多少呢?

由于赋值运算符是从右向左结合的,因此可以将表达式写成如下形式:

```

a+=a--a*=a;
等价于 a=a+(a--a*=a);
等价于 a=a+(a=a-(a*=a));
等价于 a=a+(a=a-(a=a*a));

```

根据表达式 $a=a+(a=a-(a=a*a))$,可以将其拆成如下 3 步分别进行计算:

```

a=a*a;    // a 等于 100
a=a-a;    // a 等于 0

```

```
a=a+a;    // a 还是等于 0
```

因此，表达式 `a+=a-=a*=a` 所得到的结果为 0。

从上面的示例可以看出，虽然复合表达式可以提高编译效率，但是太复杂的复合表达式就适得其反了。同时，还应该避免编写有多用途的复合表达式，例如：

```
d = (a = b + c) * e ;
```

该表达式既求 `a` 的值又求 `d` 的值，应该将其拆分为两个独立的语句，代码如下：

```
a = b + c ;
d = a * e ;
```

建议 16-3：尽量避免在表达式中使用默认的优先级

在 C 语言中，运算符的优先级如表 2-1 所示。

表 2-1 C 语言中运算符的优先级表

优先级	运算符	描述	结合方向
1	[]	数组下标，如 <code>a[10]</code>	从左到右
	()	圆括号	
	.	用于直接使用结构和联合，如 <code>employee.age</code>	
	->	用于结构和联合指针，如 <code>employee->age</code>	
2	+	正号	从右到左
	-	负号	
	(类型)	强制类型转换	
	++	自增	
	--	自减	
	*	指针取值	
	&	取地址	
	!	逻辑非	
	~	按位取反	
3	sizeof	取长度	从左到右
	/	除法	
	*	乘法	
4	%	取模（求余）	从左到右
	+	加法	
	-	减法	
5	<<	左移	从左到右
	>>	右移	
6	>	大于	从左到右
	>=	大于等于	
	<	小于	
	<=	小于等于	

(续)

优先级	运算符	描述	结合方向
7	==	等于	从左到右
	!=	不等于	
8	&	按位与	从左到右
9	^	按位异或	从左到右
10		按位或	从左到右
11	&&	逻辑与	从左到右
12		逻辑或	从左到右
13	?:	条件运算符，如 y=x>0?1:0	从右到左
14	=	赋值运算符	从右到左
	/=	除后赋值	
	*=	乘后赋值	
	%=	取模后赋值	
	+=	加后赋值	
	-=	减后赋值	
	<<=	左移后赋值	
	>>=	右移后赋值	
	&=	按位与后赋值	
	^=	按位异或后赋值	
	=	按位或后赋值	
15	,	逗号运算符，例如：表达式 1, 表达式 2, 表达式 3	从左到右

虽然 C 语言中的运算符都有自己的优先级别，但是为了提高程序的可读性，防止阅读程序时产生误解，防止因默认的优先级与设计思想不符而导致程序出错，我们应该尽量避免使用默认的优先级。如果代码行中的运算符比较多，应当用括号明确表达式的计算顺序，从而避免采用默认的运算符优先级。

来看下面的示例代码：

```
if ((a | b) && (a & c))
{
    /* 处理代码 */
}
if ((a | b) < (c & d))
{
    /* 处理代码 */
}
```

在上面的代码中，我们用括号来明确表达式的计算顺序，使程序看起来非常直观，具有良好的可读性。现在，我们采用默认的运算符优先级来改写上面的代码：

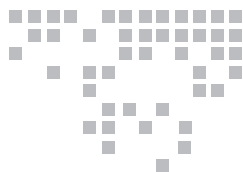
```
if (a | b && a & c)
{
    /* 处理代码 */
}
```

```
if (a | b < c & d)
{
    /* 处理代码 */
}
```

根据表 2-1 的运算符优先级原则，“a | b && a & c”等价于“(a | b) && (a & c)”，第一个 if 语句“if(a | b && a & c)”不会出错，但语句却不易理解；再来看第二个 if 语句“if(a | b < c & d)”，因为“<”运算符的优先级比“|”与“&”运算符高，所以“a | b < c & d”等价于“a | (b < c) & d”，这就造成了判断条件出错。

最后还需要说明的是，除了可以通过括号的方式来明确表达式的计算顺序，避免使用默认的运算符优先级，还需要遵循表达式简洁原则，尽量避免在表达式中把不同类型的操作符混合起来。





程序控制语句应该保持简洁高效

同其他程序设计语言一样，C 语言也提供三种基本流程控制结构：顺序结构、选择结构与循环结构。从执行方式上看，从第一条语句到最后一条语句完全按顺序来执行的，称为顺序结构；若在程序执行过程当中，根据用户的输入或中间结果去选择执行若干不同的任务，则称为选择结构；如果在程序的某处，需要根据某项条件重复地执行某项任务若干次或直到满足或不满足某条件为止，这就构成循环结构。

建议 17：if 语句应该尽量保持简洁，减少嵌套的层数

C89 指明，编译程序必须最少支持 15 层嵌套，C99 把限度提升到 127 层。实际上，多数编译器程序支持远大于 15 层嵌套的 if。虽然如此，但我们为了使 if 语句保持简洁与可读性，应该尽量减少嵌套的层数。

建议 17-1：先处理正常情况，再处理异常情况

我们在编写代码时，首要原则就是要使正常情况的执行代码清晰，确认那些不常发生的异常情况处理代码不会遮掩正常的执行路径。也就是说，我们应该把正常情况的处理放在 if 后面，而不要放在 else 后面。这样，不仅符合我们平时的逻辑思维习惯，同时这对代码的可读性和性能也很重要。例如，下面的代码是对学生的成绩及格与不及格进行判断：

```
if(grade>=60)
{
    /* 处理成绩及格的学生 */
}
```



```

}
else if(grade>=30&&grade<60)
{
    /* 处理成绩大于等于 30, 并且小于 60 的学生 */
}
else
{
    /* 处理成绩 30 以下的学生 */
}

```

这样的代码，不仅看起来很符合我们平时的逻辑思维习惯，而且 if 语句在做判断时，正常情况一般比异常情况发生的概率更大（否则就应该把异常和正常调过来了），即及格的学生多于不及格的学生。如果把执行概率更大的代码放到后面，也就意味着 if 语句将进行多次无谓的比较，如下面的代码所示：

```

if(grade<30)
{
    /* 处理成绩 30 以下的学生 */
}
else if(grade>=30&&grade<60)
{
    /* 处理成绩大于等于 30, 并且小于 60 的学生 */
}
else
{
    /* 处理成绩及格的学生 */
}

```

因为及格的学生总是多于不及格的学生，所以在上面的代码中，if 语句将进行多次无谓的比较，同时也难以理解。

建议 17-2：避免“悬挂”的 else

对于“悬挂”的 else 这个问题，或许大家都已经了解甚至熟知了。尽管如此，但在平时编码中还是会有许多菜鸟和老手在此为程序埋下 Bug 的种子，如下面的示例代码所示：

```

#include <stdio.h>
int main (void)
{
    int a=30;
    int b=31;
    int c=7;
    int x=0;
    if(a>b)
        if(b<c)
            x=1;
        else
            x=-1;
}

```

```

    printf("%d\n",x);
    return 0;
}

```

上面的代码虽然看起来比较简洁，但却给程序带来致命性错误，同时这种写作方式也是许多老手所崇拜的（if 语句后省略掉花括号 {}）。

在上面的这段代码中，我们的本意应该有两种主要情况： $a > b$ 与 $a \leq b$ 。如果满足条件 $a > b$ ，我们将继续判断条件 $b < c$ ，如果满足条件 $b < c$ ，则将 1 赋给变量 x ；如果满足条件 $a \leq b$ ，则直接将 -1 赋给变量 x 。

然而实际情况并非如此，这段代码所做的与我们的意图相去甚远。其根本原因就在于 C 语言中有这样一条规则：else 始终与同一对括号内最近的未匹配的 if 相结合。根据这条规则，代码中的 else 应该与 $\text{if}(b < c)$ 进行配对，而并非与 $\text{if}(a > b)$ 进行配对，所以变量 x 的原值没有变，依然是 0。

为了使大家能够更加清楚地看到“悬挂”的 else 所产生的原因以及带来的问题，我们可以通过给 if/else 语句加上 “{}”，写成如下等价形式：

```

#include <stdio.h>
int main (void)
{
    int a=30;
    int b=31;
    int c=7;
    int x=0;
    if(a>b)
    {
        if(b<c)
        {
            x=1;
        }
        else
        {
            x=-1;
        }
    }
    printf("%d\n",x);
    return 0;
}

```

现在，大家就能够很清楚地看到这个 else 到底与谁匹配了。由此可以看出，不论是菜鸟还是老手，在什么时候都不能够偷懒，还是老老实实地把 “{}” 加上，这样就不会让我们迷糊了。正确的写法如下面的代码所示：

```

#include <stdio.h>
int main (void)
{

```

```

int a=30;
int b=31;
int c=7;
int x=0;
if(a>b)
{
    if(b<c)
    {
        x=1;
    }
}
else
{
    x=-1;
}
printf("%d\n",x);
return 0;
}

```

现在，代码中的 `else` 可以正确地与第一个 `if(a>b)` 相结合了，即使它离第二个 `if(b<c)` 更近也是如此。因此，变量 `x` 的结果正是我们所期望的 `-1`。

除此之外，有些 C 程序员也通过使用宏定义的方式来能达到类似的效果，如下面的示例代码所示：

```

#include <stdio.h>
#define IF      {if(
#define THEN    ){
#define ELSE    else {
#define END      }
int main (void)
{
    int a=30;
    int b=31;
    int c=7;
    int x=0;
    IF a>b
        THEN IF b<c
            THEN x=1;
            END
        END
    ELSE
        x=-1;
    END
    END
    printf("%d\n",x);
    return 0;
}

```

虽然这样也可以避免“悬挂”的 `else` 这个问题，但是这样的代码却很难读懂，尤其会给

后来的代码维护人员带来麻烦，所以我们不建议大家这样做。

建议 17-3：避免在 if/else 语句后面添加分号 “;”

在 C 语言中，只有分号 “;” 组成的语句称为空语句。空语句是什么也不执行的语句，常常被用来作为空循环体。如果你不小心在 if/else 语句后面添加了分号 “;”，那么程序将很容易违背你的意愿，导致意外的运算结果，如下面的示例代码所示：

```
int main (void)
{
    int x=1;
    if(x<0);
    x++;
    printf("%d\n",x);
    return 0;
}
```

在上面的代码中，语句 `x++` 并不是在 “`if(x<0)`” 为真的时候才被调用，而是任何时候都会被调用，所以最后变量 `x` 的值为 2。这究竟是怎么回事呢？

其实，问题就出在 if 语句后面的分号 “;” 上。我们知道，在 C 语言中，分号预示着一语句的结尾。但值得注意的是，并不是每条 C 语言语句都需要分号作为结束标志。比如，if 语句的后面就并不需要分号，但如果你不小心添加了分号，编译器并不会提示出错。因为编译器会把这个分号解析成一条空语句，即上面的代码等价于下面的代码：

```
int main (void)
{
    int x=1;
    if(x<0)
    {
        ;
    }
    x++;
    printf("%d\n",x);
    return 0;
}
```

其实，这种手误性错误是我们很容易犯的，往往一不小心多写了一个分号，就会导致结果与预想的相差很远。因此，建议大家使用 `NULL` 来替代空语句，这样做可以明显地区分真正必需的空语句和不小心多写的分号造成的误解，如下面的示例代码所示：

```
int main (void)
{
    int x=1;
    if(x<0)
        NULL;
    x++;
}
```

```

    printf("%d\n",x);
    return 0;
}

```

建议 17-4：对深层嵌套的 if 语句进行重构

在代码中，如果某个函数的 if 语句嵌套太深，为了程序的可读性、可维护性与效率，我们应该尽量想办法进行重构，以减少 if 语句的嵌套层数，如下面的示例代码所示：

```

int main (void)
{
    int x=0;
    int a=1;
    int b=3;
    int c=5;
    int d=7;
    if(a<b)
    {
        x+=b;
        if(b<c)
        {
            x+=c;
            if(c<d)
            {
                x++;
                if(a>d)
                {
                    x=d;
                }
            }
        }
    }
    printf("%d",x);
    return 0;
}

```

在上面这个简单的示例代码中，我们使用了 4 层 if 嵌套。为了减少程序中 if 语句的嵌套层数，我们首先能够想到的办法就是可以通过重复检测条件中的某一部分来简化嵌套的 if 语句，如下面的示例代码所示：

```

int main (void)
{
    int x=0;
    int a=1;
    int b=3;
    int c=5;
    int d=7;
    if(a<b)
    {

```

```

        x+=b;
        if (b<c)
        {
            x+=c;
        }
    }
    if (a<b&& b<c&& c<d)
    {
        x++;
        if (a>d)
        {
            x=d;
        }
    }
    printf("%d",x);
    return 0;
}

```

通过上面的代码，我们可以清楚地看到通过重复检测条件中的某一部分来简化嵌套的 if 语句的办法并不能无偿地减少嵌套层数。同时，作为减少嵌套层数的代价，必须容忍使用一个更复杂的判断。也就是说，虽然这样减少了 if 语句的嵌套层数，但增加了一些复杂的判断，让我们有点得不偿失的感觉。

既然我们对通过重复检测条件中的某一部分来简化嵌套的 if 语句的办法不是很满意，那么我们还可以使用 if/break 块来简化 if 语句的嵌套层数。上面的示例代码采用 if/break 块重构后如下所示：

```

int main (void)
{
    int x=0;
    int a=1;
    int b=3;
    int c=5;
    int d=7;
    do{
        if (a>=b)
            break;
        x+=b;
        if (b>=c)
            break;
        x+=c;
        if (c>=d)
            break;
        x++;
        if (a<=d)
            break;
        x=d;
    } while(false);
    printf("%d",x);
}

```

```
    return 0;  
}
```

从上面的代码可以看出，相对于通过重复检测条件中的某一部分来简化嵌套的 if 语句的办法，if/break 块真正地实现了逻辑的扁平化，减少了 if 语句的嵌套层数，从而使代码看起来比较清晰，增加了程序的可读性。与此同时，该方法还不会增加复杂的条件判断，从而可以避免因为重构而导致的 if 条件出错。当然，你还可以使用它的另外两个相似的 if/return 和 if/goto 块来达到同样的效果，但这种方法唯一的缺陷就是破坏了程序的内聚性。

除了上面的两种方法之外，我们还可以通过把代码分割开来，把深层嵌套的 if 语句抽取出来放进单独的函数中。这样，不仅减少了 if 语句的嵌套层数，同时，一个好的函数名对代码也具有自我注解的作用，在一定程度上也可以提高程序的可读性。但这种方法与前面的方法一样，程序逻辑的复杂度依然存在，甚至更复杂。

因此，如有可能，我们应该选择完全重新设计深层嵌套的代码。在通常情况下，如果程序中存在比较复杂的逻辑代码，就说明你还没有充分地理解程序，从而无法简化它。所以对程序员来说，深层嵌套的 if 语句也是一个警告，它说明你要么想办法进行重构，要么重新设计该程序。

建议 18：谨慎 0 值比较

对于 0 值比较，看起来似乎很简单，但实际情况并非如此，笔者曾经见过许多面试的程序员对此题的回答模棱两可。下面，我们就来讨论一下如何正确地对各类型的数据进行 0 值比较。

建议 18-1：避免布尔型与 0 或 1 进行比较

布尔类型是计算机科学中的逻辑数据类型，它只提供两种原始值：true（真）和 false（假）。通常情况下，零值为“假”，任意非零值都是“真”。

在 C99 的标准中，增加了一个内置的布尔类型 _Bool，可以存储值 1（true）和 0（false）。同时，为了与 C++ 兼容，C99 还在 <stdbool.h> 文件中定义了宏 bool、true 与 false，从而使程序员写出 C 与 C++ 相互兼容的程序。

然而在 C99 之前（即 C89 中），C 语言的标准并没有提供布尔类型，但这并不意味着 C89 就不能表示布尔值的概念。其实，C 语言中的所有关系运算（>、>=、<、<=、== 与 !=）、逻辑运算（&&、|| 与 !）以及条件声明（if 与 while）等都以任意非零值代表 true（真），零值代表 false（假）。但是，任意非零值代表为 true（真），这样就会带来了一个严重的问题，因为 true 由一个特定的值来表示，然而 true 的值究竟是什么并没有一个统一的标准。例如，在 Visual C++ 中将 true 定义为 1，而在 Visual Basic 中则将 true 定义为 -1。

因此，我们将布尔类型的比较代码写成如下形式显然是不行的：


```
if( flag == 1 ) /* 表示 flag 为真 */
if( flag == 0 ) /* 表示 flag 为假 */
```

上面的代码虽然看起来是正确的，但不具备很好的可移植性。当然，我们也可以通过宏定义的形式来写成如下形式：

```
if( flag == true ) /* 表示 flag 为真 */
if( flag == false ) /* 表示 flag 为假 */
```

上面的代码虽然可读性较好，但同样也会因为 true 或 false 的不同定义值而出错。因此，正确的写法应该如下：

```
if( flag ) /* 表示 flag 为真 */
if( !flag ) /* 表示 flag 为假 */
```

这样就避免了上面的所有可能，并且使代码看起来也比较简洁。

建议 18-2：整型变量应该直接与 0 进行比较

相对于其他数据类型，整型变量与零值比较就简单多了。例如整型变量 i，它与零值比较的标准 if 语句如下：

```
if(i == 0)
if(i != 0)
```

切忌，千万不可用模仿布尔型变量的风格而写成如下形式：

```
if(i)
if(!i)
```

上面的代码很容易会让人误以为变量 i 是布尔型变量。

建议 18-3：避免浮点变量用 “==” 或 “!=” 与 0 进行比较

其实，关于浮点数的比较，早在建议 3-4 中就做了比较详细的介绍，本节将继续向大家做一些实用性的讲解。

我们知道，在 C 语言中，无论是 float 还是 double 浮点数类型的变量，都有其精度的限制。对于超出精度限制的浮点数，计算机会把它们的精度之外的小数部分截断。因此，原本就不相等的两个浮点数在计算机中就可能变成相等的，如下面的示例代码所示：

```
int main (void)
{
    float a=10.22222225;
    float b=10.22222229;
    if (a == b )
    {
        printf("a==b \n");
    }
}
```

```

    }
    else
    {
        printf("a!=b\n");
    }
    return 0;
}

```

在上面的代码中，从数学上讲，a 和 b 是不相等的，但是在 32 位计算机中它们却是相等的，因此程序的输出结果为“a==b”。由此可见，我们一定要避免将浮点类型的变量直接用“==”或“!=”与数字进行比较，而应该设法把它们转化成“>”或“<=”的形式。

如果两个同符号的浮点数之差的绝对值小于或等于某一个可接受的误差 EPSILON（即精度），就认为它们是相等的，否则就是不相等的。两个浮点数 x 与 y 是否相等的正确的比较方式如下面的代码所示：

```

if( fabs(x-y) <= EPSILON )    // x 等于 y
if( fabs(x-y) > EPSILON )    // x 不等于 y

```

同理，浮点数 x 和 0 是否相等的正确比较方式如下面的代码所示：

```

if( fabs(x) <= EPSILON )    // x 等于 0
if( fabs(x) > EPSILON )    // x 不等于 0

```

下面，为了加深大家的理解，我们来看一个完整的例子，如代码清单 3-1 所示。

代码清单 3-1 浮点数比较示例

```

#include <stdio.h>
#include <math.h>
#define EPSILON 0.000000001
int main (void)
{
    double a = 10.22222225;
    double b = 10.22222222;
    double c = 0.00000001;
    if ( fabs(a-b) <= EPSILON )
    {
        printf("a:%.12f == b:%.12f,
                精度为 %.12f \n",a,b,EPSILON);
    }
    else
    {
        printf("a:%.12f != b:%.12f,
                精度为 %.12f \n",a,b,EPSILON);
    }
    if (fabs(c) <= EPSILON)
    {
        printf("c:%.12f == 0, 精度为 %.12f \n",c,EPSILON);
    }
    else

```

```

{
    printf("c:%.12f != 0, 精度为 %.12f \n", c, EPSILON);
}
return 0;
}

```

代码清单 3-1 的运行结果如图 3-1 所示。



图 3-1 代码清单 3-1 的运行结果

建议 18-4：指针变量应该用 “==” 或 “!=” 与 NULL 进行比较

在 C 语言中，定义指针变量时一定要同时初始化该指针变量，如下面的示例代码所示：

```
int* p = NULL;
```

这里需要特别注意的是，尽管 NULL 的值与 0 相同，但是两者意义却不相同。因此，在我们将指针变量与 0 值做比较的时候，也应该直接用 “==” 或 “!=” 与 NULL 进行比较。例如，指针变量 p 与 0 值比较的标准 if 语句如下面的示例代码所示：

```
if( p== NULL )
if( p!= NULL )
```

这样通过将 p 与 NULL 显式进行比较，从而强调 p 是指针变量。如果我们直接将指针变量 p 与 0 值进行比较，就很容易让人误解 p 是整型变量，如下面的示例代码所示：

```
if( p==0 )
if( p!=0 )
```

同理，如果写成下面这种形式，就很容易让人误解 p 是布尔变量：

```
if( p )
if( !p )
```

建议 19：避免使用嵌套的 “?:”

在 C 语言中，“?:” 运算符是 if/else 语句的另外一种表示形式，其一般形式如下所示：

```
Exp1 ? Exp2 : Exp3
```

其中，Exp1、Exp2 与 Exp3 都是表达式。程序首先对 Exp1 求值，如果 Exp1 的值为真时，就对 Exp2 求值并将其求值结果作为整个问号表达式的值；否则，就对 Exp3 求值并将其求值

结果作为整个问号表达式的值。

由于问号表达式语法的简洁性，因此我们经常看到一些程序员在做判断时只用“?:”，而从不使用 if/else 语句。如下面的示例代码所示：

```
unsigned int f( int x)
{
    return x>=1?1:0;
}
```

很显然，相对于 if/else 语句，这样的代码看起来更加简洁明了。但是，如果我们嵌套使用“?:”，那就不一样了。如下面的示例代码所示：

```
unsigned int f(unsigned int x)
{
    return( (x<=1)?(1-x):(x==4)?2:(x+1) );
}
```

在上面的代码中，嵌套了两层“?:”，它实际等效于下面的代码：

```
unsigned int f(unsigned int x)
{
    unsigned temp;
    if(x <= 1)
    {
        if(x != 0)
        {
            temp = 0;
        }
        else
        {
            temp = 1;
        }
    }
    else
    {
        if(x == 4)
        {
            temp = 2;
        }
        else
        {
            temp = x + 1;
        }
    }
    return temp;
}
```

从上面的代码中可以看出，尽管嵌套使用“?:”可以使代码变得简洁，但代码的可读性却因此降低了不少。这个时候有的程序员或许会说，嵌套使用“?:”不仅使代码简洁，更重

要的是它比 if/else 语句能够产生更加高效的代码。但实际情况并非如此，有兴趣的读者不妨试一试下面的代码，对两者做个比较。

```
unsigned int f(unsigned int x)
{
    unsigned temp;
    if( x <= 1 )
    {
        temp = 0;
        if( x == 0 )
        {
            temp = 1;
        }
    }
    else
    {
        temp = 2;
        if( x != 4 )
        {
            temp = x + 1;
        }
    }
    return temp ;
}
```

其实，使用“?:”运算符所存在的问题是：由于它很简单，容易使用，看起来好像是产生高效代码的理想方法。因此，程序员就不再寻找更好的解决方法了。更糟糕的是，有些程序员会将 if/else 语句全部转换为“?:”来获得所谓的高效解决方案。而实际上“?:”并不比 if/else 语句好，它们所产生的汇编代码也基本相同。

因此，为了能够提高程序的执行效率，我们应该将时间花在寻求可替代的高效算法上，而不是一味地追求以某个稍微不同的方式来实现同一个算法上。例如，我们可以将上面的代码修改成下面这种更直接的实现方法：

```
unsigned int f(unsigned int x)
{
    assert( x >= 0 && x <= 4 );
    if( x == 1 )
    {
        return 0 ;
    }
    if( x == 4 )
    {
        return 2 ;
    }
    return x + 1 ;
}
```

甚至，我们还可以使用下面的列表方式来实现上面的示例程序，如下面的代码所示：

```
unsigned int f(unsigned int x)
{
    assert( x >= 0 && x <= 4 );
    static const unsigned temp[] = {1,0,3,4,2 };
    return temp[x] ;
}
```

建议 20：正确使用 for 循环

C 语言提供了三种类型的循环控制语句：for 循环语句、while 循环语句和 do/while 循环语句。其中，for 循环语句的一般形式为：

```
for ( < 初始化 >; < 条件表达式 >; < 增量 > )
{
    循环体语句 ;
}
```

一般情况下，初始化总是一个赋值语句，它用来为循环控制变量赋初值；条件表达式则是一个关系表达式，它决定什么时候退出循环；而增量定义循环控制变量每循环一次后按什么方式变化。这三个部分之间用分号“;”分割开来。

建议 20-1：尽量使循环控制变量的取值采用半开半闭区间写法

从功能上看，虽然半开半闭区间写法和闭区间写的功能是完全相同的，但相比之下，半开半闭区间写法更能够直观地表达意思，具有更高的可读性。下面，我们就通过示例代码看看两者之间的区别。

其中，闭区间的写法示例如下面的代码所示：

```
for( i=0;i<=n-1;i++ )
{
    /* 处理代码 */
}
```

在上面的代码中，i 值属于闭区间写法，即“ $0 \leq i \leq n-1$ ”，起点到终点的间隔为 $n-1$ ，循环次数为 n 。

半开半闭区间的写法示例如下面的代码所示：

```
for( i=0;i<n;i++ )
{
    /* 处理代码 */
}
```

在上面的代码中，i 值属于半开半闭区间写法，即“ $0 \leq i < n$ ”，起点到终点的间隔为 n ，循环次数为 n 。

从上面的两段示例代码中可以看出，尽管它们的功能是完全相同的，但相比之下，第二个程序示例（半开半闭区间写法）具有更高的可读性。因此，在 for 循环中，我们应该尽量使循环控制变量的取值采用半开半闭区间写法。

建议 20-2：尽量使循环体内工作量达到最小化

我们知道，for 循环随着循环次数的增加，会加大对系统资源的消耗。如果你写的一个循环体内的代码相当耗费资源，或者代码行数众多（一般来说循环体内的代码不要超过 20 行），甚至超过一显示屏，那么这样的程序不仅可读性不高，而且还会让你的程序的运行效率大大降低。这个时候，我们通常可以通过如下两种方法进行优化。

1) 重新设计这个循环，确认这些操作是否都必须放在这个循环里，并仔细考虑循环体内的语句是否可以放在循环体之外，从而使循环体内工作量最小化，提高程序的时间效率。如下面的示例代码所示：

```
for (i = 0; i < n; i++)
{
    tmp += i;
    sum = tmp;
}
```

很显然，在上面的代码中每执行一次 for 循环，就要执行一次“sum = tmp”语句来重新为变量 sum 进行赋值，这样的写法很浪费资源。因此，我们完全可以将“sum = tmp”语句放在 for 语句之后，如下面的示例代码所示：

```
for (i = 0; i < n; i++)
{
    tmp += i;
}
sum = tmp;
```

这样，“sum = tmp”语句只执行一次，不仅可以提高程序执行效率，而且程序也具有更高的可读性。

2) 可以考虑将这些代码改写成一个子函数，在循环中只调用这个子函数即可。

建议 20-3：避免在循环体内修改循环变量

在 for 循环语句中，我们应该严格避免在循环体内修改循环变量，否则很有可能导致循环失去控制，从而使程序执行违背我们的原意，如下面的示例代码所示：

```
for( i=0;i<10;i++ )
{
    i=10;
}
```


在上面的代码中，在循环体内对循环变量 *i* 进行赋值之后，for 循环中止执行，从而使程序执行违背我们的原意，更严重的情况会给程序带来灾难性的后果。

建议 20-4：尽量使逻辑判断语句置于循环语句外层

一般情况下，我们应该尽量避免在程序的循环体内包含逻辑判断语句。当循环体内不得已而存在逻辑判断语句，并且循环次数很大时，我们应该尽量想办法将逻辑判断语句移到循环语句的外层，从而使程序减少执行逻辑判断语句的次数，提高程序的执行效率。如下面的示例代码所示：

```
for (i = 0; i < n; i++)
{
    if (condition)
    {
        DoSomething();
    }
    else
    {
        DoOtherthing();
    }
}
```

在上面的代码中，每执行一次 for 循环，都要执行一次 if 语句判断。当 for 循环的次数很大时，执行多余的判断不仅会消耗系统的资源，而且会打断循环“流水线”作业，使得编译器不能对循环进行优化处理，降低程序的执行效率。因此，我们可以通过将逻辑判断语句移到循环语句的外层的方法来减少判断的次数，如下面的代码所示：

```
if (condition)
{
    for (i = 0; i < n; i++)
    {
        DoSomething();
    }
}
else
{
    for (i = 0; i < n; i++)
    {
        DoOtherthing();
    }
}
```

虽然上面的代码没有前面的看起来简洁，但却使程序执行逻辑判断语句减少 *n*-1 次，在 for 循环次数很大时，这种优化显然是值得的。

最后还需要注意的是，循环体中的判断语句是否可以移到循环体外，要视程序的具体情况而定。一般情况下，与循环变量无关的判断语句可以移到循环体外，而有关的则不可以。

建议 20-5：尽量将多重循环中最长的循环放在最内层，最短的循环放在最外层

在多重 for 循环中，如果有可能，应当尽量将最长的循环放在最内层，最短的循环放在最外层，以减少 CPU 跨切循环层的次数。如下面的示例代码所示：

```
for (i=0; i<100; i++)
{
    for ( j=0; j<5; j++ )
    {
        /* 处理代码 */
    }
}
```

为了提高上面代码的执行效率，我们可以依照这条建议将上面的代码修改为如下形式：

```
for ( j=0; j<5; j++ )
{
    for (i=0; i<100; i++)
    {
        /* 处理代码 */
    }
}
```

这样，既不会失去程序原有的可读性，同时也提高了程序的执行效率。

建议 20-6：尽量将循环嵌套控制在 3 层以内

有研究数据表明，当循环嵌套超过 3 层，程序员对循环的理解能力会极大地降低。同时，这样程序的执行效率也会很低。因此，如果代码循环嵌套超过 3 层，建议重新设计循环或将循环内的代码改写成子函数。

建议 21：适当地使用并行代码来优化 for 循环

在实际编程中，尽量把长的有依赖的代码链分解成几个可以在流水线执行单元中并行执行的没有依赖的代码链，如下面的示例代码所示：

```
double num[100];
double sum=0;
int i=0;
for (i=0;i<100;i++)
{
    sum += num[i];
}
```

很显然，在上面的代码中要执行 100 次 for 循环语句。然而，对于这样的代码，我们可以使用分解成多路的形式进行优化。在这里，我们选择将上面的程序分解成 4 路，即使用 4

段流水线浮点加法，浮点加法的每一个段占用一个时钟周期，从而保证最大的资源利用率，如下面的示例代码所示：

```
double num[100];
double sum=0;
double sum1=0;
double sum2=0;
double sum3=0;
double sum4=0;
int i=0;
for(i=0;i<100;i+=4)
{
    sum1 += num[i];
    sum2 += num[i+1];
    sum3 += num[i+2];
    sum4 += num[i+3];
}
sum = sum4+sum3+sum2+sum1;
```

最后还需要说明的是，由上面的代码可以看出，因为浮点数的精确度问题，在一些情况下，这些优化可能会导致意料之外的结果。但在大部分情况下，最后结果可能只有最低位存在错误。因此，对计算结果正确性的影响不大。

建议 22：谨慎使用 do/while 与 while 循环

前面已经说过，在 C 语言中，循环控制语句除 for 循环语句之外，还提供另外两种循环控制语句：while 循环语句和 do/while 循环语句。在实际应用中，for 循环语句的使用频率最高，while 循环语句其次，do/while 循环语句很少用。

建议 22-1：无限循环优先选用 for(;;)，而不是 while(1)

在 C 语言中，最常用的无限循环语句主要有两种：while (1) 和 for(;;)。从功能上讲，这两种语句的效果完全一样。那么，我们究竟该选择哪一种呢？

其实，从 while 和 for 的语义上来看，显然 for(;;) 语句运行速度要快一些。按照 for 的语法规则，两个分号“;”分开的是 3 个表达式。现在表达式为空，很自然地被编译成无条件的跳转（即无条件循环，不用判断条件）。如代码 for(;;) 在 Microsoft Visual Studio 2010 集成开发环境 VC++ 的 Debug 模式下将生成如下汇编代码：

```
for( ; ; )
00931451  jmp             main+41h (931451h)
```

相比之下，while 语句就不一样了。按照 while 的语法规则，while() 语句中必须有一个表达式（这里是 1）判断条件，生成的代码用它进行条件跳转。即 while 语句 () 属于有条件

循环，有条件就要判断条件是否成立，所以其相对于 `for(;;)` 语句需要多几条指令。如代码 `while (1)` 在 Microsoft Visual Studio 2010 集成开发环境 VC++ 的 Debug 模式下将生成如下汇编代码：

```
while(1)
011A1451  mov     eax,1
011A1456  test    eax,eax
011A1458  je      main+55h (11A1465h)
011A1463  jmp     main+41h (11A1451h)
```

根据上面的分析结果，很显然，`for(;;)` 语句指令少，不占用寄存器，而且没有判断、跳转指令。当然，如果从实际的编译结果来看，两者的效果常常是一样的，因为大部分编译器都会对 `while (1)` 语句做一定的优化。但是，这还需要取决于编译器。因此，我们还是应该优先选用 `for(;;)` 语句。

建议 22-2：优先使用 for 循环替代 do/while 与 while 循环

在 C 语言中，`while` 循环与 `do/while` 循环的区别在于：`while` 循环语句先测试控制表达式的值，再执行循环体，如下面的示例代码所示：

```
unsigned int i=0;
while (i<1000)
{
    i++;
    /* 处理程序 */
}
```

相比之下，`do/while` 循环语句则先执行循环体，再测试控制表达式的值，如下面的示例代码所示：

```
unsigned int i=1000;
do
{
    i--;
    /* 处理程序 */
}
while (i>0);
```

如果控制表达式的值一开始为假，则 `while` 循环语句的循环体一次都不执行，而 `do/while` 循环语句的循环体仍然要执行一次再跳出循环。

在实际开发环境中，无论是 `do/while` 与 `while` 循环，还是 `for` 循环，它们之间都是可以相互替换的。但从代码的可读性而言，建议优先选用 `for` 循环。尤其面对多层循环嵌套，`for` 循环的代码相比之下就更易读懂了。当然，如果在循环的次数不明确的情况下，还是要使用 `do/while` 和 `while` 循环。

建议 23：正确地使用 switch 语句

相对于 if 语句而言，switch 语句可以更方便地应用于多个分支的控制流程。C89 指明，一个 switch 语句最少可以支持 257 个 case 语句，而 C99 则要求至少支持 1023 个 case 语句。然而，在实际开发环境中，为了程序的可读性与执行效率，应该尽量减少 switch 语句中的 case 语句。

除此之外，switch 语句与 if 语句不同的是，switch 语句只能够测试是否相等，因此，case 语句后面只能是整型或字符型的常量或常量表达式；而在 if 语句中还能够测试关系与逻辑表达式。

建议 23-1：不要忘记在 case 语句的结尾添加 break 语句

在 switch 语句中，每个 case 语句的结尾不要忘记添加 break 语句，否则将导致多个分支重叠。当然，除非有意使多个分支重叠，这样可以免去 break 语句。下面我们来看一个实际示例，如代码清单 3-2 所示。

代码清单 3-2 switch 示例

```
#include <stdio.h>

void print_week(unsigned int day);

void print_week(unsigned int day)
{
    switch(day)
    {
        case 1:
            printf("Monday\n");
            break;
        case 2:
            printf("Tuesday\n");
            break;
        case 3:
            printf("Wednesday\n");
            break;
        case 4:
            printf("Thursday\n");
            break;
        case 5:
            printf("Friday\n");
            break;
        case 6:
            printf("Saturday\n");
            break;
        case 7:
            printf("Sunday\n");
            break;
```

```
        default:
            printf("error\n");
            break;
    }
}
int main (void)
{
    print_week(3);
    return 0;
}
```

在代码清单 3-2 中，在 `print_week` 函数中通过 `switch` 语句实现根据数字输出星期名称的功能。执行代码清单 3-2，程序将输出 “Wednesday”。

现在，如果将 `case 1 ~ case 4` 的 `break` 语句去掉，如代码清单 3-3 所示，程序会输出什么结果呢？

代码清单 3-3 switch 去掉 break 示例

```
void print_week(unsigned int day)
{
    switch(day)
    {
        case 1:
            printf("Monday\n");
        case 2:
            printf("Tuesday\n");
        case 3:
            printf("Wednesday\n");
        case 4:
            printf("Thursday\n");
        case 5:
            printf("Friday\n");
            break;
        case 6:
            printf("Saturday\n");
            break;
        case 7:
            printf("Sunday\n");
            break;
        default:
            printf("error\n");
            break;
    }
}
int main (void)
{
    print_week(2);
    return 0;
}
```

在代码清单 3-3 中，由于 case 1 ~ case 4 缺少 break 语句，因此将导致多个分支重叠，其运行结果如图 3-2 所示。

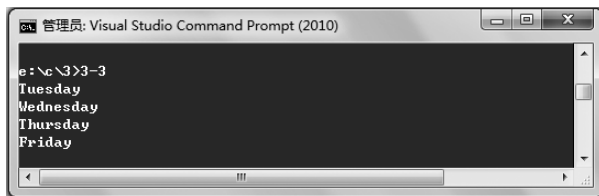


图 3-2 代码清单 3-3 的运行结果

建议 23-2：不要忘记在 switch 语句的结尾添加 default 语句

在 switch 语句中，default 语句主要用于检查默认情况，或者处理错误情况，如下面的示例代码所示：

```
default:
    printf("error\n");
    break;
```

如果在 switch 语句中去掉 default 语句，那么 switch 语句将失去对默认情况与错误情况的处理能力。所以，奉劝大家不要偷懒，老老实实把每一种情况都用 case 语句来完成，而把真正对默认情况的处理交给 default 语句来完成。即使程序真的不需要 default 处理，也应该保留此语句：

```
default:
    break;
```

这样做并非画蛇添足，可以避免令人误以为你忘记了 default 处理。

建议 23-3：不要为了使用 case 语句而刻意构造一个变量

在实际编程应用中，switch 中的 case 语句应该只用于处理简单的、容易分类的数据。如果数据并不简单，却为了使用 case 语句而刻意构造一个变量，那么这种变量很容易令我们得不偿失。因此应该严格避免这种变量，并使用 if/else if/else 结构来处理这类程序，如下面的示例代码所示：

```
char ch = c[0];
switch (ch)
{
    case 'a':
        f1();
        break;
    case 'b':
```



```

        f2();
        break;
    case 'c':
        f3();
        break;
    default:
        break;
}

```

在上面的程序中，字符变量 `ch` 的值是取字符数组 `c[]` 的第一个字符，与 `case` 语句中的常量值逐一进行比较。很显然，这种方法存在一个严重的问题。

例如，如果字符数组 `c[]` 中存储的是“ab”字符串，那么 `c[0]` 会取第一个字符“a”与 `case` 语句进行匹配，因此会匹配到第一个 `case` 语句，并调用 `f1()` 函数。然而，如果字符数组 `c[]` 中存储的是其他以字符 `a` 开头的字符串（比如“abc”“abcd”“abcde”等），因为 `c[0]` 始终会取第一个字符的关系，因此它们同样会匹配第一个 `case` 语句而调用 `f1()` 函数。其他的 `case` 语句同理。很显然，这并不是我们想要的结果。

由此可见，当为了使用 `case` 语句而刻意构造一个变量时，真正的数据可能不会按照我们所希望的方式映射到 `case` 语句。因此，我们应该严格避免为了使用 `case` 语句而刻意构造一个变量，并使用 `if/else if/else` 结构来处理这类程序，如下面的示例代码所示：

```

if(0 == strcmp("ab",c))
{
    f1();
}
else if(0 == strcmp("bc",c))
{
    f2();
}
else if(0 == strcmp("cd",c))
{
    f3();
}
else
{
}

```

建议 23-4：尽量将长的 `switch` 语句转换为嵌套的 `switch` 语句

有时候，当一个 `switch` 语句中包括很多个 `case` 语句时，为了减少比较的次数，可以把这类长 `switch` 语句转为嵌套 `switch` 语句，即把发生频率高的 `case` 语句放在一个 `switch` 语句中，作为嵌套 `switch` 语句的最外层；把发生频率相对低的 `case` 语句放在另一个 `switch` 语句中，放置于嵌套 `switch` 语句的内层。

例如，下面的代码把发生频率相对较低的情况放置于默认的 `case` 语句内。

```

void print_week(unsigned int day)

```

```
{
    switch(day)
    {
        case 1:
            printf("Monday\n");
            break;
        case 2:
            printf("Tuesday\n");
            break;
        case 3:
            printf("Wednesday\n");
            break;
        case 4:
            printf("Thursday\n");
            break;
        case 5:
            printf("Friday\n");
            break;
        default:
            switch(day)
            {
                case 6:
                    printf("Saturday\n");
                    break;
                case 7:
                    printf("Sunday\n");
                    break;
                default:
                    printf("error\n");
                    break;
            }
    }
}
```

在上面的代码中，假设 case 6 与 case 7 不经常发生，因此将它们放置到嵌套 switch 语句的最内层。从表面看，虽然这样损失了程序的一定可读性，但当 case 语句很多，并且确实有些 case 语句发生的频率比较低时，这种解决方案还是可取的。

建议 24：选择合理的 case 语句排序方法

对于 switch 中的 case 语句排序问题，如果 case 语句很少时，可以忽略这个问题。但是，如果 case 语句很多时，那就需要好好考虑这个问题了。一般而言，可以选择下面 3 个建议进行合理排序。

建议 24-1：尽量按照字母或数字顺序来排列各条 case 语句

通常情况下，如果所有 case 语句没有明显的重要性差别，并且发生的频率都差不多，那

么可以按 A-B-C 或 1-2-3 等顺序来排列 case 语句。这样做不仅可以提高代码的可读性，而且可以很容易找到某条 case 语句，如上面的代码清单 3-2 所示。

建议 24-2：尽量将情况正常的 case 语句排在最前面

如果 switch 中存在多个情况正常的 case 语句，同时又存在多个情况异常的 case 语句。那么应该尽量将情况正常的 case 语句排在最前面，而将情况异常的 case 语句排在最后面。同时，做好相应的注释，如下面的示例代码所示：

```
switch(i)
{

    /* 正常情况开始 */
case 0:
    /* 处理代码 */
    break;
case 1:
    /* 处理代码 */
    break;
    /* 正常情况结束 */

    /* 异常情况开始 */
case -1:
    /* 处理代码 */
    break;
case -2:
    /* 处理代码 */
    break;
    /* 异常情况结束 */

default:
    break;
}
```



建议 24-3：尽量根据发生频率来排列各条 case 语句

如果能够预测出每条 case 语句大概的发生频率，就可以将执行频率最高的 case 语句放在最前面，而将执行频率较低的 case 语句放在最后面。这样不仅可以适当提高程序的性能，而且便于调试代码。因为执行频率最高的代码可能也是调试的时候要单步执行次数最多的代码。如果放在后面，找起来可能会比较麻烦，而放在前面则方便快速找到。

建议 25：尽量避免使用 goto 语句

自从提倡结构化程序设计以来，goto 语句就成为业界争议最大的语句。不论各家对 goto

语句的意见是好是坏，总结前人的经验，还是应该尽量避免在程序中使用 goto 语句，其原因主要有以下两方面。

首先，由于 goto 语句可以灵活跳转，如果不加限制，它的确会破坏结构化设计风格，如下面的示例代码所示：

```
A:
    /* 处理代码 */
    goto B;
    /* 处理代码 */
    goto C;
B:
    /* 处理代码 */
    goto A;
    /* 处理代码 */
    goto C;
C:
    /* 处理代码 */
    goto A;
    /* 处理代码 */
    goto B;
```

很显然，上面的示例代码已经能够说明问题了，随着标签数量增多，将给代码的可读性、可调试性与可维护性带来一场灾难。

其次，若不加限制地使用 goto 语句，该语句可能跳过变量的初始化、重要的计算等语句，从而给程序带来灾难性的错误与潜在的安全隐患，如下面的示例代码所示：

```
char *p = NULL;
/* 处理代码 */

goto state;

/* 变量 sum 被 goto 跳过 */
int sum = 0;
/* 指针 p 被 goto 跳过，没有分配内存 */
p = (char *)malloc(40 * sizeof(char));
if (p == NULL)
{
    /* 处理代码 */
}
/* 处理代码 */

state:

/* 使用 p 指向的内存里的值的代码 */
/* 使用变量 num */
```

如上面代码所示，如果编译器不能发现此类错误，则每用一次 goto 语句都可能导致程序出现灾难性的错误与潜在的安全隐患。

当然，如果遇到下列情况，goto 语句还是有其自身优势的。

```

for(...)
{
    for(...)
    {
        for(...)
        {
            /* 使用 goto 语句跳出循环，执行其他的语句 */
            goto A;
        }
    }
}

A:
/* 处理代码 */

```

在上面的代码中，如果使用 break 语句，则只能跳出单层的循环；如果使用 return 语句，则会跳出整个函数，无法继续执行其他的代码。因此，这里可以使用 goto 语句。其实，如果陷入很深层次的循环中想要跳出最外层的循环，用 goto 直接跳出比用 break 一层循环一层循环跳出要好得多。

建议 26：区别 continue 与 break 语句

在 C 语言中，continue 语句和 break 语句的区别如下。

(1) 对于 continue 语句

它只结束本次循环，而不是终止整个循环的执行。也就是说，在 while 循环、do/while 循环和 for 循环中，continue 语句将跳过循环体中剩余的语句而强制执行下一次循环，即结束本次循环，跳过循环体中下面尚未执行的语句，接着进行下一次是否执行循环的判定，如下面的示例代码所示：

```

int main(void)
{
    unsigned int i=0;
    for( i=0;i<20;i++)
    {
        if (i%2==0)
            continue;
        printf("%4d",i);
    }
    printf("\n");
    return 0;
}

```

在上面的代码中，为了演示 continue 语句的作用，利用 continue 语句输出 0 到 19 之间不能被 2 整除的数。其中，当 i 能被 2 整除时，将执行 continue 语句，结束本次循环，并跳

过尚未执行的 `printf("%4d",i)` 语句，接着执行下一次循环与判断语句 `if (i%2==0)`。只有 `i` 不能够被 2 整除时才执行 `printf("%4d",i)` 语句来输出结果，如图 3-3 所示。



图 3-3 `continue` 示例代码的运行结果

(2) 对于 `break` 语句

相对于 `continue` 语句，`break` 语句则是结束整个循环过程，不再判断执行循环的条件是否成立。也就是说，在分支结构程序设计中用 `break` 语句可以跳出 `switch` 语句块，继续执行 `switch` 下面的语句。而在 `while` 循环、`do/while` 循环和 `for` 循环中，`break` 语句用来终止本层循环，继续执行该循环外的语句。

现在，如果将上面示例代码中的 `continue` 语句修改成 `break` 语句结果会是什么呢？如下面的示例代码所示：

```
int main(void)
{
    unsigned int i=0;
    for( i=0;i<20;i++)
    {
        if (i%2==0)
            break;
        printf("%4d",i);
    }
    printf("\n");
    return 0;
}
```

其实，从代码中可以看出，当 `for` 循环执行第一次循环时（即 `i` 的值为 0），表达式 `0%2` 的值为 0，因此，`if (i%2==0)` 语句返回真，从而执行 `break` 语句，终止整个 `for` 循环，最后程序什么都不输出。

最后还需要注意的是，`break` 语句不能用于循环语句和 `switch` 语句之外的任何其他语句中。在循环语句中，`break` 语句与 `continue` 语句一般与 `if` 语句一起使用。