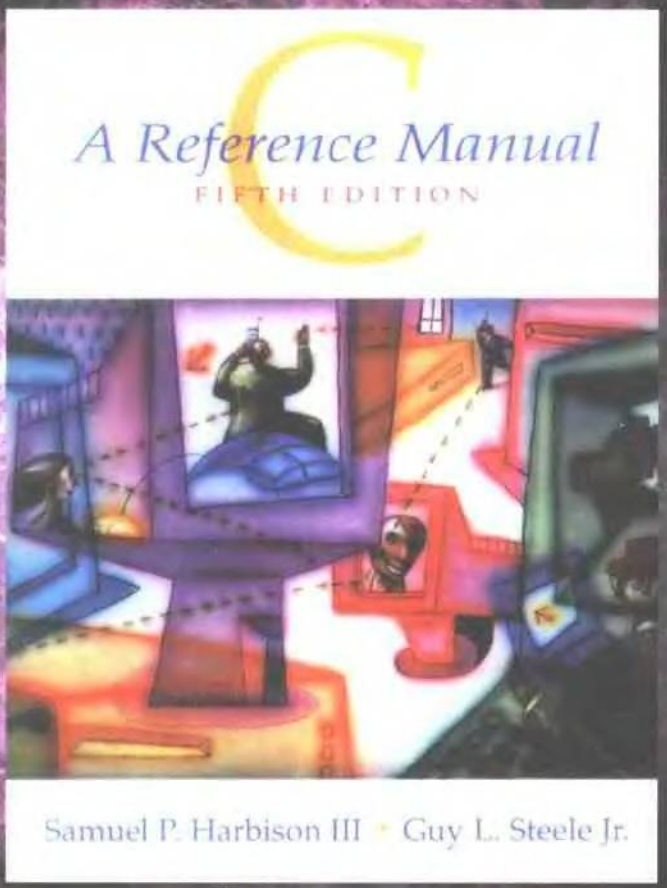
计 算 机 科 学 丛 书

原书第5版

C语言参考手册

（美） Samuel P. Harbison III Guy L. Steele Jr. 著 邱仲潘 等译



C: A Reference Manual
Fifth Edition

机械工业出版社
China Machine Press

PEARSON Prentice Hall

这本畅销的权威参考手册对C语言的基本概念和运行库提供了完整的描述，同时还强调了以正确性、可移植性和可维护性为根本出发点的良好的C语言编程风格，被国外众多高校广泛采用为教材或教学参考书。本书描述了C语言各个版本的所有细节，是C语言编程人员和实现者惟一必备的参考手册。最新的第5版经过修订和更新，融入了最新C语言标准ISO/IEC 9899:1999的完整描述，包括强大的语言扩展和新的函数库。

Web站点www.CAReferenceManual.com中包含了本书较长示例的源代码、对C语言争论点的深入讨论、最新ISO/IEC语言标准修订以及其他重要C语言资源的链接。

本书作为参考手册，提供了非常详细、清晰的C语言描述：

- 标准C语言（1999）：是标准C语言的新版本，支持复数类型与布尔类型、变长数组、精确浮点数编程和具有可移植性与国际化的新的库函数
- 标准C语言（1989）：当前大多数编程人员使用的C语言版本
- 传统C语言：1990年之前常用的版本，还有几百万行代码正在使用
- C++兼容C语言：可以同时在C语言与C++中使用的代码
- 所有C语言版本的完整运行库

作者简介

Samuel P. Harbison III 于卡内基-梅隆大学获得计算机科学博士学位，现任Carlow学院的计算机科学系副教授。他曾就职于德州仪器和Tartan公司，还曾经担任C++程序设计语言标准化国际工作组的主席。他的研究领域涉及程序设计语言和软件开发工具。

Guy L. Steele Jr. 于MIT获得计算机科学和人工智能博士学位，曾任卡内基-梅隆大学计算机科学系副教授，还曾就职于Tartan实验室和Thinking Machines公司，1994年加入SUN公司，主要从事并行算法、实现策略、软件支持等方面的研究以及Java语言规范的制定。他曾是X3J11(C语言)标准委员会、X3J3(Fortran)标准委员会成员，现在还担任X3J13(Common Lisp)标准委员会的主席。鉴于他在Lisp语言词法方面的贡献，1988年ACM授予他Grace Murray Hopper奖。他于1990年被选为美国人工智能学会会士，于1994年被选为ACM会士。他还曾任1990年ACM图灵奖评审委员会的主席。

PEARSON Prentice Hall www.PearsonEd.com Pearson Education

封面设计 刘永青

ISBN 7-111-12219-4
9 787111 122197

HZ BOOKS 华章图书

网上购书：www.china-pub.com
北京市西城区百万庄南街1号 100037
读者服务热线：(010)68995259，68995264
读者服务信箱：hzedu@hzbook.com
http://www.hzbook.com

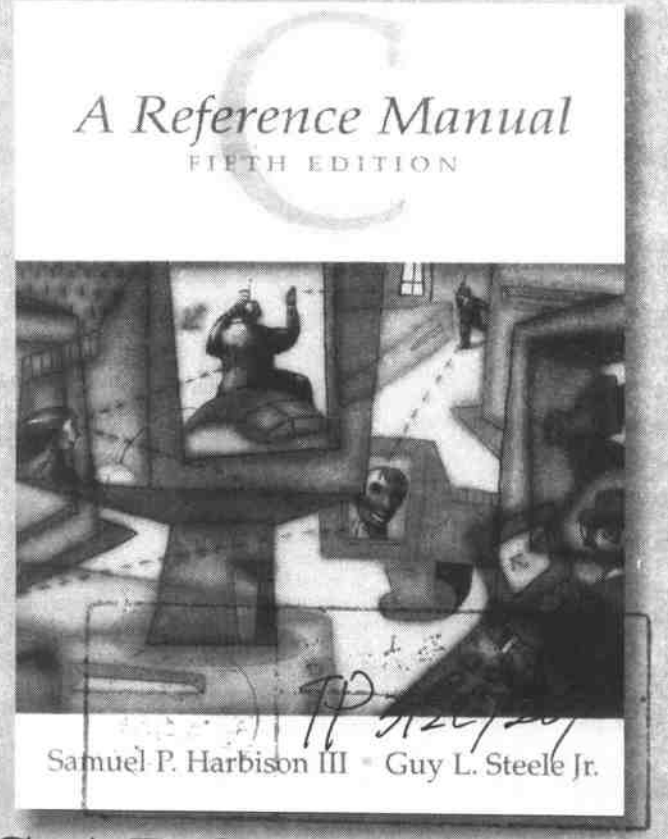
ISBN 7-111-12219-4/TP · 2696
定价：39.00 元

计 算 机 科 学 丛 书

原书第5版

C语言参考手册

(美) Samuel P. Harbison III Guy L. Steele Jr. 著 邱仲潘 等译



C: A Reference Manual
Fifth Edition

机械工业出版社
China Machine Press

本书是经典C语言参考手册的最新版，在强调正确性、可移植性和可维护性的基础上，对C语言的具体细节、运行库以及C语言编程风格做了完整、准确的描述。

本书涵盖了传统C语言、C89、C95、C99等所有C语言版本的实现，同时讨论了C++与C语言兼容的部分。全书自上而下介绍了C语言的词法结构、预处理器、声明、类型表达式、语句、函数和运行库，是所有C语言编程人员必备的参考书。

本书版权登记号：图字：01-2002-3650

图书在版编目（CIP）数据

C语言参考手册（原书第5版）/（美）哈比逊（Harbison, S. P.Ⅲ），斯蒂尔（Steele, G. L. Jr.）著；邱仲潘等译. -北京：机械工业出版社，2003.8
（计算机科学丛书）
书名原文：C: A Reference Manual, Fifth Edition
ISBN 7-111-12219-4

Ⅰ.C… Ⅱ.①哈… ②斯… ③邱… Ⅲ.C语言－程序设计－技术手册 Ⅳ.TP312-62

中国版本图书馆CIP数据核字（2003）第039007号

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：庞 燕
北京瑞德印刷有限公司印刷·新华书店北京发行所发行
2003年8月第1版第1次印刷
787mm × 1092mm 1/16 · 25.75印张
印数：0 001-5 000册
定价：39.00元

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭橥了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短、从业人员较少的现状下，美国等发达国家在其计算机科学发展的几十年间积淀的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章图文信息有限公司较早意识到“出版要为教育服务”。自1998年开始，华章公司就将工作重点放在了遴选、移译国外优秀教材上。经过几年的不懈努力，我们与Prentice Hall，Addison-Wesley，McGraw-Hill，Morgan Kaufmann等世界著名出版公司建立了良好的合作关系，从它们现有的数百种教材中甄选出Tanenbaum，Stroustrup，Kernighan，Jim Gray等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及度藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专诚为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍，为进一步推广与发展打下了坚实的基础。

随着学科建设的初步完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都步入一个新的阶段。为此，华章公司将加大引进教材的力度，在“华章教育”的总规划之下出版三个系列的计算机教材：除“计算机科学丛书”之外，对影印版的教材，则单独开辟出“经典原版书库”；同时，引进全美通行的教学辅导书“Schaum's Outlines”系列组成“全美经典学习指导系列”。为了保证这三套丛书的权威性，同时也为了更好地为学校和老师们服务，华章公司聘请了中国科学院、北京大学、清华大学、国防科技大学、复旦大学、上海交通大学、南京大学、浙江大学、中国科技大学、哈尔滨工业大学、西安交通大学、中国人民大学、北京航空航天大学、北京邮电大学、中山大学、解放军理工大学、郑州大学、湖北工学院、中国国家信息安全测评认证中心等国内重点大学和科研机构在计算机的各个领域的著名学者组成“专

家指导委员会”，为我们提供选题意见和出版监督。

这三套丛书是响应教育部提出的使用外版教材的号召，为国内高校的计算机及相关专业的教学度身订造的。其中许多教材均已为M. I. T.，Stanford，U.C. Berkeley，C. M. U. 等世界名牌大学所采用。不仅涵盖了程序设计、数据结构、操作系统、计算机体系结构、数据库、编译原理、软件工程、图形学、通信与网络、离散数学等国内大学计算机专业普遍开设的核心课程，而且各具特色——有的出自语言设计者之手、有的历经三十年而不衰、有的已被全世界的几百所高校采用。在这些圆熟通博的名师大作的指引之下，读者必将在计算机科学的宫殿中由登堂而入室。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证，但我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。教材的出版只是我们的后续服务的起点。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

电子邮件：hzedu@hzbook.com
联系电话：（010）68995264
联系地址：北京市西城区百万庄南街1号
邮政编码：100037

专家指导委员会

（按姓氏笔画顺序）

尤晋元	王　珊	冯博琴	史忠植	史美林
石教英	吕　建	孙玉芳	吴世忠	吴时霖
张立昂	李伟琴	李师贤	李建中	杨冬青
邵维忠	陆丽娜	陆鑫达	陈向群	周伯生
周克定	周傲英	孟小峰	岳丽华	范　明
郑国梁	施伯乐	钟玉琢	唐世渭	袁崇义
高传善	梅　宏	程　旭	程时端	谢希仁
裘宗燕	戴　葵			

前　言

本书作为一本C语言参考手册，对C语言的基本概念和运行库提供了完整和准确的描述，同时还强调了以正确性、可移植性和可维护性为根本出发点的良好的编程风格。

我们希望读者已经了解基本编程概念，并且很多读者已经可以用C语言熟练编程。为了保持参考手册的格式，我们自下而上介绍C语言的词法结构、预处理器、声明、类型、表达式、语句、函数和运行库。书中包括许多交叉引用，使读者可以从任何地方入手。

第5版完整地介绍了最新的国际C语言标准ISO/IEC 9899:1999(C99)，明确指出语言本身和库函数中哪些特性是C99新增的，指出C99与原有C89标准的不同之处。这是目前惟一一本适用于所有主流C语言版本的参考书：包括传统C语言、1989年C标准、1995年对C89的修改与补充以及当前的C99标准。本书还介绍了标准C语言和标准C++的原始C语言子集。尽管C99中有许多新的信息，但我们没有对本书的章节组织做很大的修改，这样就可以使熟悉旧版的读者能够顺利找到所要的材料。

本书最初源于我们在Tartan公司的工作——为从微机到大型机的一系列计算机开发C语言编译器系列。我们要求编译器文档齐全，提供精确和有用的错误诊断信息并能产生有效的目标代码。一个经过某一编译器正确编译的C语言程序应能在硬件差别允许的前提下，在所有其他编译器中正确编译。

1984年，尽管C语言已经非常普及，但还没有一本书能够非常精确地介绍C语言，以便指导我们设计新的编译器。同样，当时的文档对编程人员和客户也不够精确，人们希望利用编译器比采用当时已经习惯的方法可以更彻底地分析C语言程序。本书特别注意影响程序清晰度、目标代码有效性和不同环境中程序移植性的语言特性。

Web站点

欢迎读者访问本书的Web站点**CAReferenceManual.com**，其中包括了书中示例的代码、更深入的讨论、澄清的问题以及更多C语言资源的链接。

致谢

在第5版准备过程中，特别感谢原NCITS J11主席Rex Jaeschke、芬兰赫尔辛基的Antoine Trux以及爱迪生设计集团创始人Steve Adamczyk的帮助。

对于本书的以前版本给予过帮助的人员包括Jeffrey Esakov、Alan J.Filipski、Frank J. Wagner、Debra Martin、P. J. Plauger以及Steve Vinoski。其他提供过帮助的人员包括Aurelio Bignoli、Steve Clamage、Arthur Evans、Jr.,Roy J. Fuller、Morris M. Kessan、George V.Reilly、Mark Lan、Mike Hewett、Charles Fischer、Kevin Rodgers、Tom Gibb、David Lim、Stavros Macrakis、Steve Vegdahl、Christopher Vickery、Peter van der Linden和Dave Wilson。还要感谢Michael Angus、Mady Bauer、Larry Breed、Sue Broughton、Alex Czajkowski、Robert Firth、David Gaffney、Steve Gorman、Dennis Hamilton、Chris Hanna、Ken Harrenstien、Rex Jaeschke、

Don Lindsay、Tom MacDonald、Peter Nelson、Joe Newcomer、Keyin Nolish、David Notkin、Peter Plamondon、Roger Ray、Larry Rosler、David Spencer以及Barbara Steele。

本书最初的一些示例程序参考了下列著作中的算法：

- Beeler, Michael, Gosper, R. William, and Schroeppel, Richard, *HAKMEM*, AI Memo 239 (Massachusetts Institute of Technology Artificial Intelligence Laboratory, 1972年2月);
- Bentley, Jon Louis, *Writing Efficient Programs*(Prentice-Hall, 1982);
- Bentley, Jon Louis, "Programming Pearls" (monthly column appearing in *Communications of the ACM* beginning August 1983);
- Kernighan, Brian W., and Ritchie, Dennis M., *The C Programming Language*(Prentice-Hall, 1978);
- Knuth, Donald E., *The Art of Computer Programming* Volumes 1-3(Addison-Wesley, 1968, 1969, 1973, 1981);
- Sedgewick, Robert, *Algorithms*(Addison-Wesley, 1983).

感谢这些作者的灵感。

还有一点需要说明，Guy Steele先生工作繁忙，无法在新版本中参与更多工作。虽然书中仍然处处体现出他对C语言精辟的分析，但新版本中的任何新问题，都不能再由他负责。

《C语言参考手册》一书已经超过17岁，感谢所有读者多年来对她的关爱。

Sam Harbison
于宾夕法尼亚州匹兹堡市
harbison@CAReferenceManual.com

参加本书翻译的人员有：周阳生、刘文红、邹能东、彭振庆、黄志坚、李耀平、郭王旋。刘文琼、邱冬全、邱燕等完成了本书的录入工作，刘云昌、刘联昌对手写稿与打印稿进行了校对。

目　录

第二部分 C语言库

第三部分 附 录

第一部分 C 语 言

第1章 简 介

20世纪70年代初，Dennis Ritchie在贝尔实验室设计了C语言，它的前身可以追溯到1960年的ALGOL 60语言，1963年剑桥的CPL语言，1967年Martin Richard的BCPL语言，以及1970年贝尔实验室Ken Thompson的B语言。尽管C语言是一种通用编程语言，但通常用于系统编程。特别值得一提的是，著名的UNIX操作系统最初就是用C语言写成的。

C语言的普及有许多原因。它是个小巧、高效而强大的编程语言，具有丰富的运行库，而且不使用很多的隐藏机制就可以对计算机进行精确控制。经过10多年的标准化之后，编程人员已经习惯了C语言。一般来说，很容易用C语言编写可以在不同国家用不同语言的不同计算系统之间移植的程序。而且，现有的大量遗留C语言代码正在被修改和扩展。

从20世纪90年代末期开始，虽然C语言慢慢被“大哥”C++取代，但它仍然有许多忠实追随者，在不需要C++特性或不接受C++的开销的场合，C语言仍然非常流行。

C语言经受了时间的检验，仍不失为一种熟练的编程人员用来迅速有效地工作的编程语言。几百万行的代码充分证明了这种语言的优势。

1.1 C语言的演变

1984年我们编写本书第1版时，C语言已经被广泛使用，但还没有关于这个语言的正式标准和精确描述。事实上的标准是当时正在使用的C语言编译器。1989年，C语言建立了国际标准，1994年作了修订，1999年又作了一次重大修改。

仅仅改变语言的定义并不能自动改变全世界已有的几百万行C语言代码。我们努力使本书与时俱进，这样编程人员在遇到各种C语言方言时都能以本书作为参考手册。

1.1.1 传统C语言

最初的C语言描述出现在Brian Kernighan与Dennis Ritchie（通常合称为“K&R”）的著作《*The C Programming Language*》第1版（Prentice-Hall，1978）。此书出版后，这个语言不断有细小的演变，增加或删除了一些特性。我们把20世纪80年代初公认的C语言定义称为传统C语言，是标准化之前的方言。当然，各个C语言提供商也对传统C语言进行了各种扩展。

1.1.2 标准C语言（1989）

1982年，美国国家标准协会（ANSI）认识到标准化将有助于C语言在商业化编程中的普及，因此成立了一个委员会来为C语言及其运行库制定标准。这个委员会，即X3J11（现为NCITS J11）的主席是Jim Brodie，它制定了一个标准并在1989年被正式采用，即美国国家标准X3.159-1989或称作“ANSI C”。

考虑到编程活动是国际化的，因此完成ANSI C语言之后，成立了一个国际标准化组织

ISO/IEC JTC1/SC22/WG14，在PJ.Plauger的领导下，只作了少量编辑性修改，即把ANSI标准变成国际标准ISO/IEC 9899:1990。此后，ISO/IEC标准被ANSI采用，人们把这个公共标准称为“标准C语言”。由于这个标准后来又有了改变，因此我们称其为“标准C语言（1989）”或简称“C89”。

传统C语言到C89的改变包括：

- 增加了真正的标准库
- 新的预处理器命令与特性
- 函数原型允许在函数声明中指定参数类型
- 一些新关键字，包括**const**、**volatile**与**signed**
- 宽字符、宽字符串和多字节字符
- 对约定规则、声明和类型检查的许多小改动与澄清

1.1.3 标准C语言（1995）

作为对C语言标准的正常维护工作，WG14对C89作了两处技术修订（缺陷修复）和一个补充（扩展）。总的来看，尤其是通过增加新的库函数，以上这些工作对C语言标准进行了相对合适的修改，得到的结果我们称之为“C89增补1”或“C95”。C95对C89所作的改变包括：

4

- 3个新的标准库头文件**iso646.h**、**wctype.h**与**wchar.h**
- 几个新的记号和宏，用于替换一些国家的字符集中没有的运算符和标点符号
- **printf/scanf**系列函数的一些新格式代码
- 大量新函数和一些类型与常量，用于多字节字符和宽字符

1.1.4 标准C语言（1999）

ISO/IEC标准都需要经常进行审查和更新。1995年，WG14开始对C语言标准作更大的修订，最终于1999年完成并获批准。新标准ISO/IEC 9899:1999或“C99”取代原有的标准（及所有修订与补充），成为正式标准C语言。提供商根据新标准更新各自的C语言函数库和编译器。

C99在C89/C95语言和库函数中增加了许多新特性，包括：

- 复数运算
- 扩展整数类型，包括长标准类型
- 变长数组
- 布尔类型
- 对非英语字符集更好的支持
- 对浮点数类型更好的支持，包括所有类型的数学函数
- C++风格的注释（**//**）

C99的改动比C95更大，包括语言的改变和函数库的扩展。C99标准文档比C89文档大得多。但是，改变还是“本着C语言精神”进行的，语言的基本性质没有改变。

1.1.5 标准C++语言

C++是20世纪80年代初由AT&T贝尔实验室的Bjarne Stroustrup设计的，现在其地位已大大超过C语言，成为主流编程语言。大多数C语言实现实际上是C/C++实现，编程人员可以自己选择要使用哪一种语言。C++本身也已经标准化为ISO/IEC 14882:1998或称“标准C++语言”。C++在C语言之上作了许多改进，适合开发大型应用程序，包括对类型检查的改进和面向对象编程的支持。但是，C++也是最复杂的编程语言之一，对于粗心的编程人员来说，会陷入很多陷阱。

标准C++语言可以大致看成标准C语言的超集。由于C标准和C++标准的制定采用不同规范，因此它们之间不能以协调的方式互相适应。而且，C语言一直使自己有别于C++，例如，它不接受C++中类这种数据类型的“简化”版本。

可以用标准C和C++语言的公共子集（有人称为原始C语言，Clean C）编写C语言代码，使代码既可以作为C程序编译，也可以作为C++程序编译。由于C++的规则通常比标准C语言更严格，因此原始C语言通常是良好的可移植子集。编写原始C语言程序时要考虑的主要改变包括： 5

- 原始C语言程序要使用函数原型，C++中不允许旧式声明
- 原始C语言程序要在名称中避免C++保留字，如**class**与**virtual**

还有另外一些规则和差别，但通常不会造成问题。本书介绍如何编写标准C语言代码，并使其能在C++编译器中被接受。我们不准备讨论标准C语言没有提供的C++特性（当然，这些特性几乎包括了C++的一切优秀之处）。

1.1.6 本书内容

本书描述C语言的三大变形：传统C语言，C89，C99。书中提出C89增补1增加的特性，同时描述C/C++的原始C语言子集。我们还会介绍如何编写“良好的”C语言程序，即具有可读性、可移植性和可维护性的程序。

正式的“标准C语言”是C99，但我们通常所说的标准C语言是指C89中延用到C99的特性和概念。只在C99中才有的特性会被标识出来，以便于使用C89实现的编程人员可以避免使用这些特性。

1.2 使用C语言的何种方言

使用C语言的何种方言取决于可用的C语言实现产品和对代码要求的可移植性。可以选择：

1. C99，标准C语言当前版本，具有所有最新特性，但有些实现产品可能还不支持C99（这种现状会迅速改变）。
2. C89，标准C语言的上一版本。大多数最新C语言程序和大多数C语言实现产品都基于这个C语言版本，通常包括C89增补1。
3. 传统C语言，主要在维护早期C语言程序时会用到。
4. 原始C语言，与C++兼容。

C99通常与C89向上兼容，C89通常与传统C语言向上兼容。但是，很难编写向下兼容的C语言代码。例如，以函数原型为例，在标准C语言中是可选的，在传统C语言中是禁止的，而在C++中是必需的。好在可以用C语言预处理器根据使用的不同实现产品改变代码，甚至根据标准C语言是否包括C89增补1扩展而改变代码。因此，C语言程序可以和所有方言保持兼容。第3章将介绍如何用预处理器完成这项工作，例子见3.9.1节。

如果不受编译器或现有C语言代码体的限制，无疑要使用标准C语言作为基本语言。标准C语言编译器几乎随处可得，自由软件基金会的GNU C(**gcc**)是免费的标准C语言编译器（提供了许多扩展功能）。 6

1.3 C语言编程概述

大多数读者已经熟悉C语言等高级语言编程，但这里还是准备简单介绍一些C语言编程的知识。

C语言程序包括一个或几个源文件或翻译单元，各包含整个C语言程序的一部分，C程序通

常由若干外部函数组成。公共声明通常放在头文件中，用特殊的**#include**命令放进源文件中（见3.4节）。一个外部函数应命名为**main**（见9.9节），程序从这个函数开始执行。

C语言编译器独立处理每个源文件，并将C语言程序文本翻译成计算机可识别的指令。编译器“理解”C语言程序并分析其正确性。如果编程人员编写了编译器能够检查的错误，则编译器将发出出错消息。当不出现错误时，编译器的输出通常称为目标代码或目标模块。

编译完所有源文件之后，目标模块将被提供给连接程序。连接程序解决模块之间的引用，增加标准运行库中的函数，并检测一些程序错误，例如没有定义某个需要的函数。连接程序通常不是专门用于C语言的，每个计算机系统有一个标准连接程序，供许多不同语言写成的程序使用。连接程序产生一个可执行程序，然后可以调用或运行这个程序。尽管大多数计算机系统都要经过这些步骤，但编程人员看到的可能不同。在Microsoft公司Visual Studio之类集成环境中，这个过程可能是完全隐藏的。本书不关注建立C语言程序的细节，读者可以查阅自己的计算机系统和编程文档。

例　假设程序**aprogram**包括两个C语言源文件**hello.c**与**startup.c**。文件**hello.c**包含下列代码：

```
#include <stdio.h> /* defines printf */
void hello(void)
{
    printf("Hello!\n");
}
```

由于**hello.c**中包含的功能（函数**hello**）要在程序其他部分使用，因此我们生成头文件**hello.h**声明这些功能，其中包含下列语句：

```
extern void hello(void);
```

7

文件**startup.c**包含主程序，其功能是调用函数**hello**：

```
#include "hello.h"
int main(void)
{
    hello();
    return 0;
}
```

在UNIX系统中，编译、连接和执行程序只要两步：

```
% cc -o aprogram hello.c startup.c
% aprogram
```

第一行编译和连接两个源文件，增加需要的标准库函数，将可执行程序写入文件**aprogram**中。然后第二行执行程序，打印下列结果：

```
Hello!
```

其他非UNIX实现可能使用不同命令。现代编程环境向编程人员提供了集成的图形化界面，在这种环境中建立C语言应用程序时，只要从菜单中选择命令或点击图形按钮即可。　□

1.4　符合性

C语言程序和C语言实现产品都可以符合标准C语言。使C语言程序严格符合标准C语言的条

件是程序只使用那些在标准中描述过的特性和库函数。程序的操作一定不能依赖于标准C语言中指出为未说明的、未定义的或仅在某一特定实现中定义的C语言特征。Perennial公司和Plum Hall公司提供的标准C语言测试组件为建立实现产品与标准之间的符合性提供了帮助。

符合实现分为两种——*宿主实现*和*独立实现*。C语言实现成为符合宿主实现的条件是它接受任何严格符合的程序；符合独立实现的条件是它只接受任何不使用库函数的且严格符合的程序，而不接受那些头文件**float.h**、**iso646.h**(C95)、**limits.h**、**stdarg.h**、**stdbool.h**(C99)、**stddef.h**和**stdint.h**(C99)中提供的程序。第10章会列出这些头文件的内容。独立符合性是为了使C语言实现适用于嵌入式系统或其他只提供最基本运行环境支持的目标环境，例如，某一系统可能不包含文件系统。

符合程序是符合实现接受的程序，因此，符合程序可能依赖于一些符合实现的不可移植的、依赖于特定实现定义的特性，而严格符合程序则不能依赖于这些特性（从而获得最大的可移植性）。

符合实现可能提供一些扩展，但不改变任何严格符合程序的含义。这样就使实现可以增加库程序和定义自己的**#pragma**指令，但不引入新的保留标识符或改变标准库函数的操作。 8

编译器提供商不断提供客户熟悉的非符合扩展，可以通过特殊开关启用或关闭这些扩展。

1.5 语法符号

本书用统一的风格表示C语言程序形式。说明C语言语法时，终端符号用固定字体排印，在程序中的表示同书写形式完全一样。非终端符号用斜体字排印，以字母开头，后面可以加上0个或多个字母、数字或连字符：

expression argument-list declarator

引入语法定义时，在非终端符号名后加一个冒号，然后一行或几行可选项出现在随后的行中：

character :
    printing-character
    escape-character

如果冒号后面出现one of字样，则表示后面一行或几行中的终端符号都是可选的：

digit : one of
    0 1 2 3 4 5 6 7 8 9

定义中的可选成分以在终端符号或非终端符号后面附加下标opt的方式表示：

enumeration-constant-definition :
    enumeration-constant enumeration-initializer$_{opt}$

initializer :
    expression
    { *initializer-list* ,$_{opt}$ }

9

第2章 词法元素

本章介绍C语言的词法结构，即C语言源文件中可以出现的字符及其如何构成词法单元或记号。

2.1 字符集

C语言源文件是从一个字符集中选择的字符组合的序列。C语言程序用ISO/IEC 10646基本拉丁字符集中的下列字符编写：

1. 52个大小写拉丁字母

```
A B C D E F G H I J K L M N O P Q R S T
U V W X Y Z a b c d e f g h i j k l m
n o p q r s t u v w x y z
```

2. 10个数字

```
0 1 2 3 4 5 6 7 8 9
```

3. 空格

4. 水平制表符（HT）、垂直制表符（VT）和换页符（FF）

5. 29个特殊字符及其正式名称（见表2-1）

源程序还要用某种方式分行，为此可以使用字符、字符序列或源字符集之外的机制（如记录结束符）。

表2-1 特殊字符

字符	名称	字符	名称	字符	名称
!	感叹号	+	加号	"	引号
#	数字号	=	等号	{	左花括号
%	百分号	~	波浪号	}	右花括号
^	折音符	[	左方括号	,	逗号
&	和号	]	右方括号	.	句号
*	星号	'	撇号	<	小于号
(	左括号	|	竖线	>	大于号
_	下划线	\	反斜杠	/	除号
)	右括号	;	分号	?	问号
-	连字符	:	冒号		

有些国家的本国字符集不包括表2-1的所有特殊字符。C89（增补1）定义了三字符组与记号重拼，使C语言程序可以用ISO 646-1083不变代码集编写。

C语言源程序中有时还使用其他字符，包括：

1. 格式符，如退格符（BS）和回车符（CR）。

2. 其他基本拉丁字符，包括$(美元号)、@（商务中的AT）和'（重音符）。

格式符作为空格对待，不影响源程序。其他特殊字符只能放在注释语句、字符型常量、字符串型常量和文件名中。

参考章节 基本拉丁字符集 2.9；字符型常量 2.7.3；注释语句 2.2；字符编码方式 2.1.3；字符转义码 2.7.6；执行字符集 2.1.1；字符串型常量 2.7.4；记号重拼 2.4；三字符组 2.1.4

2.1.1 执行字符集

C语言程序执行期间解释的字符集不一定与编写C语言程序的字符集相同，执行字符集中的字符可以表示为源字符集中的对等字符或以反斜杠（\）开头的特殊字符转义序列。

除了前面介绍的标准字符之外，执行字符集还应包括：

1. null（空）字符，应编码为数值0。
2. 换行符，作为行末标志。
3. 警报、退格和回车符。

12

null（空）字符表示字符串结尾，换行符在输入/输出期间将字符流分行（在编程人员看来，换行符好像在执行环境的文本流中实际存在，但运行库实现可以任意模拟。例如，换行符可以在输出中转换成记录结束符，记录结束符可以在输入中转换成换行符）。

和源字符集一样，执行字符集通常要包括退格符、水平制表符、垂直制表符、换页符和回车符等格式符。可以在源程序中用特殊转义序列表示这些字符。

在同一计算机上编译和执行C语言程序时，这些源字符集和执行字符集是相同的。但有时程序是交叉编译的，即在一台计算机（主机）上编译，在另一台计算机（目标计算机）上执行。编译器计算涉及字符的常量表达式编译值时，要使用目标计算机的编码方式，而不是源计算机编码方式。

参考章节 字符型常量 2.7.3；字符编码方式 2.1.3；字符集 2.1；常量表达式 7.11；转义符 2.7.5；文本流 第15章

2.1.2 空白符与行终结符

在C语言源程序中，空格、行末符、水平制表符、垂直制表符和换页符统称为空白符（下面介绍的说明语句也是空白符）。这些字符在编译时忽略，除非用来分隔相邻记号或放在字符型常量、字符串型常量和`#include`文件名中。空白符可以将C语言程序排版成易读的格式。

行末符或字符序列表示源程序行结束。在有些实现中，回车符、换页符和垂直制表符等格式符同时也终止源代码行，称为*行终结符*。行终结符对识别预处理器控制行非常重要。行终结符后面的字符是下一行第一个字符。如果第一个字符是行终结符，则终止另一个空行，等等。

源代码行可以在下一行继续，只要在第一行末尾加上反斜杠（\）或标准C语言三字符组`??/`。删除反斜杠和行末标志可得到长的*逻辑源代码行*。这个规则在预处理器命令行和字符串型常量中都有效，是最实用的和可移植的。标准C语言和许多非标准实现将其一般化以适用于任何源程序行。这种源代码行拼接从概念上讲是发生在预处理和C语言程序词法分析之前，但在三字符组处理以及多字节字符序列转换成源字符集之后。

13

例 标准C语言中可以把记号分离到多行中。代码行

```
if (a==b) x=1; el\
se x=2;
```

等价于代码行

```
if (a==b) x=1; else x=2;
```

□

如果实现把任何非标准源字符当作空白符或行终结符，则应分别像空格和行末符一样处理。标准C语言建议实现把所有这类字符转换成某种传统表示方法，作为读取源程序时的第一个操作。但是，编程人员不要过分相信这种自动转换，应谨慎对待，例如期望换页符前面的反斜杠能被自动删除是不现实的。

大多数C语言实现对续行拼接前后的源行最大长度有一定限制，C89要求实现允许至少509个字符的逻辑源代码行，而C99允许4095个字符。

参考章节　字符型常量　2.7.3；预处理器词法规则　3.2；源字符集　2.1.1；字符串型常量　2.7.4；记号　2.3；三字符组　2.1.4

2.1.3 字符编码方式

计算机（执行）字符集中的每个字符都有一些约定编码方式，即计算机上的数字化表示。这个编码方式很重要，因为C语言将字符转换成整数，这些整数值是各字符的约定编码。前面列出的所有标准字符都要有惟一的非负整数编码。

一个常见的C语言编程错误是将一种编码方式误认为另一编码方式。

例　C语言表达式`'Z'-'A'+1`计算`Z`与`A`的编码之差再加1，得到字母表中的字符数。事实上，在ASCII字符集编码中结果为26，但在EBCDIC编码中，字母表不是连续编码，因此结果为41。□

参考章节　源字符集和执行字符集　2.1.1

2.1.4 三字符组

标准C语言中包括一组三字符组，使C语言程序可以只用ISO 646-1083不变代码集编写，这是七比特ASCII代码集的子集，是许多非英语国家字符集公用的代码集。三字符组以两个连续问号开头，见表2-2。标准C语言还提供一些记号的重拼（2.4节）和定义一些运算符的宏替换的头文件 14 `<iso646.h>`，但与三字符组不同的是，这些替换无法在字符串型常量和字符型常量中识别。

表2-2　ISO三字符组

三字符组	替换	三字符组	替换
`??(`	`[`	`??)`	`]`
`??<`	`{`	`??>`	`}`
`??/`	`\`	`??!`	`|`
`??'`	`^`	`??-`	`~`
`??=`	`#`		

源程序中三字符组的转换发生在词法分析（转换为记号）之前和识别字符串型常量和字符型常量中的转义符之前。标准C语言只能识别以上9个三字符组，所有其他字符序列（如`??&`）不进行转换。新的转义字符`\?`可用于防止对类似于三字符组的字符序列进行解释。

例　如果字符串中要包含通常解释为三字符组的三字符序列，则要用反斜杠转义字符引出至少一个三字符组字符。因此，字符串常量`"What?\?!"`实际表示包含字符`What??!`的字符串。

要编写包含一个反斜杠的字符串常量，就要编写两个连续反斜杠（第一个引出第二个），然后每个反斜杠可以转换成三字符组对等项目。因此，字符串常量`"??/??/"`表示包含一个反斜杠的字符串。□

参考章节　字符集　2.1；转义符　2.7.5；`iso646.h`　11.9；字符串拼接　2.7.4；记号重拼　2.4

2.1.5 多字节字符与宽字符

为了适应包含大量字符的非英语字母表，标准C语言引入了宽字符和宽字符串。要在面向字节的外部世界中表示宽字符和宽字符串，就要引入多字节字符的概念。C89增补1可以处理多字节字符与宽字符。

宽字符和宽字符串 宽字符是扩展字符集元素的二进制表示，具有整数类型**wchar_t**，在头文件**stddef.h**中声明。C89增补1增加了整数类型**wint_t**，应能表示所有**wchar_t**类型的值加上另一个独特的记为**WEOF**的非宽字符值。标准C语言没有对宽字符指定任何编码方式，但 15
数值0保留为“null宽字符”。可以用特殊常量语法指定宽字符常量（见2.7.3节）。

例 宽字符通常占用16位，因此**wchar_t**可以在32位计算机上表示为**short**或**unsigned short**。如果**wchar_t**为**short**，而数值 -1不是有效宽字符，则**wint_t**可以是**short**，**WEOF**可以是 -1。但**wint_t**通常是**int**与**unsigned int**。

如果实现者选择不支持扩展字符集（这在美国的C语言提供商中很常见），则**wchar_t**可以定义为**char**，“扩展字符集”与正常字符集相同。 □

宽字符串是以null宽字符结尾的宽字符连续序列。*null*宽字符是用0表示的宽字符。除了null宽字符和**WEOF**之外，标准C语言没有指定扩展字符集的编码方式。可以用特殊常量语法指定宽字符串常量（见2.7.4节）。

多字节字符 C语言程序把宽字符作为一个单元操纵，但大多数外部介质（如文件）和C语言源程序是基于字节字符。熟悉扩展字符集的编程人员提出了*多字节编码方式*作为映射字节字符序列与宽字符序列的区域设置特定方式。

*多字节字符*是源字符集或执行字符集中宽字符的表示（可以各有不同编码方式），因此多字节字符串是正常C语言字符串，但字符可以解释为一系列多字节字符。多字节字符的形式和多字节字符与宽字符之间的映射是由实现者定义的。这个映射是在编译时对宽字符常量和宽字符串常量进行的，标准库中提供了运行时进行这种映射的函数。

多字节字符可以使用*状态相关编码方式*，这时多字节字符的解释可能依赖于前面出现的多字节字符。通常，这种编码方式利用*shift*字符（多字节字符中的控制符）改变当前和后续字符的解释。多字节字符序列中的当前解释称为编码的*转换状态*（或shift状态）。开始转换多字节字符序列时总是有一个独特的*初始转换状态*（或shift状态），通常在转换结束时返回。

例 编码方式A是例子中使用的假想编码方式，是状态相关的，有两个shift状态：“上”和“下”。字符↑将shift状态变为“上”，字符↓将shift状态变为“下”。下状态是初始状态，所有非shift字符具有正常解释。上状态中每个多字节字符包括一对字母数字字符，以我们没有指定的方 16
式定义宽字符。

下列字符序列分别包含编码方式A中的3个多字节字符，从初始shift状态开始：

```
abc    ab↑e3    ↑ab↓b↑23    ↓a↓b↓c
```

最后一个字符串中的shift字符不是严格必需的。如果允许冗余shift序列，则多字节字符可能变得非常长（如↓↓...↓**x**）。除非知道多字节字符序列开始时的shift状态，否则无法分析**abcdef**之类的序列，它可能表示3个或6个宽字符。

序列**ab↑?x**在编码方式A中是无效的，因为在上状态中出现了非字母数字字符。序列**a↑b**

是无效的，因为最后一个多字节字符过早结束了。 □

多字节字符还可以使用状态无关编码方式，这时多字节字符的解释不依赖于前面的多字节字符（但可能要从头开始检查多字节字符序列，找到字符串中间多字节字符的开头）。例如，C语言转义符的语法（见2.7.5节）表示**char**类型的状态无关编码方式，因为反斜杠（\）改变后面一个或几个字符的解释，变成一个**char**类型的值。

例 编码方式B是例子中使用的另一个假想编码方式，是状态无关的，用一个特殊字符'∇ 改变后面的非null字符含义。下列字符序列分别包含编码方式B中的3个多字节字符：

```
abc    ∇a∇b∇c    ∇∇∇∇∇∇    a ∇bc
```

序列 ∇'∇ ∇ 在编码方式B中无效，因为末尾没有非null字符。 □

标准C语言对多字节字符有一定的限制：

1. 编码方式应表示标准字符集中的所有字符。
2. 在初始shift状态时，标准字符集中的所有单字节字符表示正常解释，不影响shift状态。
3. 包含全部0的字节是null字符，不管shift状态如何。任何多字节字符都不能在第二个字符和后续字符中使用包含全部0的字节。

这些规则保证多字节序列可以像正常C语言字符串一样处理（例如，多字节序列中不包含嵌入null字符），没有特殊多字节编码的C语言字符串可以像多字节序列一样得到解释。

多字节字符的源用法和执行用法 多字节字符可以出现在注释语句、标识符、预处理器头文件名、字符串型常量和字符型常量中。每个注释语句、标识符、预处理器头文件名、字符串型常量和字符型常量应以初始shift状态开始和结束，要包括多字节字符的有效序列。源程序物理表示中的多字节字符识别并转换成源字符集之后再进行任何词法分析、预处理和续行拼接。

17

例 日文编辑程序可以允许在字符串常量和注释中加入日文字符。如果文本写入字节流文件，则日文字符要转换成多字节序列，这是标准C语言实现可以接受和理解的。 □

处理过程中，字符型和字符串型常量中的字符转换成执行字符集，然后再解释为多字节序列。因此，可以在建立多字节字符时使用转义符（见2.7.5节）。注释语句在这个阶段之前已经从程序中删除，因此多字节注释语句中的转义符没有意义。

例 **如果源字符和执行字符集相同，'a'在执行字符集中的值为141_8，则字符串型常量"'∇ aa"**包含两个多字节字符，在编码方式B中为"'∇ \141\141"。 □

参考章节 字符型常量 2.7.3；注释 2.2；多字节转换 11.7，11.8；字符串型常量 2.7.4；**wchar_t** 11.1；**WEOF** 11.1；宽字符 2.7.3；宽字符串 2.7.4；**wint_t** 11.1

2.2 注释

标准C语言中有两种编写注释的方式。传统上，注释语句以/*开头，以*/结束。注释可以包含任意多的字符，并总被作为空白符处理。

从C99开始，注释也可以用//开始，延伸到（但不包括）下一个行终结符。这个改变可能破坏旧的C语言程序，但通常不会发生这种情况，读者可以想想什么情况下会发生这种问题。

注释无法在字符型常量或字符串型常量和其他注释中识别。C语言实现不检查注释语句内容，

除非识别（和越过）多字节字符与行终结符。

例 下列程序包含4个有效C语言注释语句： 18

```
// Program to compute the squares of
// the first 10 integers
#include <stdio.h>
void Squares( /* no arguments */ )
{
   int i;
   /*
      Loop from 1 to 10,
      printing out the squares
   */
   for (i=1; i<=10; i++)
      printf("%d //squared// is %d\n",i,i*i);
}
```

□

编译器在预处理之前删除注释语句，因此注释语句中的预处理命令无法识别，注释语句中的行终结符无法终止预处理命令。

例 下面两个**#define**命令具有相同效果：

```
#define ten (2*5)

#define ten /* ten:
               one greater than nine
            */ (2*5)
```

□

标准C语言中指定，为了进一步转换C语言程序，所有注释语句转换成一个空格字符，但一些旧的实现不插入任何空白符，这会影响预处理器工作，见3.3.9节介绍。

有些非标准C实现支持可嵌套注释语句，这种注释语句要求每个/*都有相应的*/。这个功能不是标准实现，编程人员不能依赖于这一功能。要让程序能在标准与非标准两种注释语句实现中都被接受，则注释语句中不能再包含/*字符序列。

例 要让编译器忽略C语言程序的大部分，最好把要删除的部分放在预处理器命令

```
#if 0
…
#endif
```

中，而不是在文本前后插入/*和*/。这样就可以避免担心其中源文本是否包含/*格式注释语句。□

参考章节 **#if**预处理器命令 3.5.1；预处理器词法规则 3.2；空白符 2.1 19

2.3 记号

C语言程序中的字符组合按照本章其余部分介绍的规则聚合成词法记号。记号共分5类：运算符、分隔符、标识符、关键字和常量。

编译器从左向右收集字符，总是尽量建立最长的记号，即使结果并不构成有效的C语言程序。相邻记号可以用空白符或注释语句分开。为了防止混淆，标识符、关键字、整型常量和浮点型

常量总是与后面的标识符、关键字、整数常量和浮点数常量分开。

预处理器采用稍有不同的记号规则。特别地，标准C语言预处理器把#和##作为记号，它们在传统C语言中是无效的。

例

字符组合	C语言记号
`forwhile`	`forwhile`
`b>x`	`b,>,x`
`b->x`	`b,->,x`
`b--x`	`b,--,x`
`b---x`	`b,--,-,x`

在第4个例子中，字符序列`b--x`是无效C语言语法。组合成记号后的`b,-,-,x`是有效语法，但不允许这种记号组合。 □

参考章节 注释 2.2；常量 2.7；标识符 2.5；预处理器记号 3.2；关键字 2.6；记号合并 3.3.9；空白符 2.1

2.4 运算符与分隔符

表2-3列出了C语言中的运算符与分隔符（标点符号）记号。为了帮助编程人员使用没有某些美语或英语字符的I/O设备，替换拼写`<%`、`%>`、`<:`、`:>`、`%:`与`%:%:`分别等价于分隔符`{`、`}`、`[`、`]`、`#`、`##`。除了这些重拼之外，头文件`iso646.h`中定义了展开某些运算符的宏。

在传统C语言中，复合赋值运算符被认为是两个分开的记号（一个运算符和一个等号），可以用空白符分开。在标准C语言中，运算符是单个记号。

20 **参考章节** 复合赋值运算符 7.9.2；`iso646.h` 11.9；预处理器记号 3.2；三字符组 2.1.4

表2-3 运算符与分隔符

记号类别	记　号
简单运算符	`! % ^ & * - + =`
	`~ | . < > / ?`
复合赋值运算符	`+= -= *= /= %=`
	`<<= >>= &= ^= |=`
其他复合运算符	`-> ++ -- << >>`
	`<= >= == != && ||`
分隔符	`( ) [ ] { } , ; : ...`
替换记号拼写	`<% %> <: :> %: %:%:`

2.5 标识符

标识符（或名称）是大小写拉丁字母、数字和下划线组成的序列。标识符不能以数字开头，不能与关键字的拼写相同。

从C99开始，标识符还可以包含通用字符名（见2.9节）和其他实现定义的多字节字符。通用字符不能以数字作为标识符开头，而应为字母式字符，并且不能是分隔符。具体清单见C99标

准（ISO/IEC 9899:1999，Annex D）和ISO/IEC TR 10176-1998。

identifier:
identifier-nondigit
identifier identifier-nondigit
identifier digit

identifier-nondigit:
nondigit
universal-character-name
other implementation-defined characters

nondigit: one of
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m
n o p q r s t u v w x y z
_

digit: one of
0 1 2 3 4 5 6 7 8 9

拼写完全一致的两个标识符是相同的，包括所有字母的大小写。就是说，标识符**abc**与**aBc**是不同的。

21

除了避免关键字的拼写问题之外，C语言编程人员还要保证关键字不会与标准库中使用的名称重复，包括当前标准和标准中的“未来库说明”部分。标准C语言还保留所有以下划线开头加上大写字母或另一个下划线的所有标识符，C语言编程人员要避免使用这些标识符，因为C语言实现有时用这些标识符扩展标准C语言或用于其他内部用途。

C89要求实现允许的最小标识符长度为31个有效字符，C99增加到63个有效字符。按照这一要求每个通用字符名序列或多字节序列被看成一个单一字符。

例 在标准之前的实现中，标识符长度限制为8个字符，因此标识符**countless**与**countlessone**是相同的标识符。长名称可以提高程序的清晰性，从而减少错误。利用下划线和混合大小写能使长标识符更易读：

```
averylongidentifier
AVeryLongIdentifier
a_very_long_identifier
```

□

外部标识符用存储类**extern**声明，可能还有其他拼写限制。这些标识符要由其他限制更严的软件处理，如调试程序或连接程序。C89要求外部标识符的最小长度为6个字符，不考虑字符大小写。C99增加到31个字符，区分字符大小写，但允许把通用字符名当作6个字符（最大为**\U0000FFFF**）或10个字符（**\U00010000**以上）。即使在C99之前，大多数实现也允许至少31个字符的外部名。

例 C语言编译器允许长内部标识符而目标计算机要求短外部名称时，可以用预处理器隐藏这些短名称。下列代码中，一个外部错误处理函数使用隐晦的短名称**eh73**，但此函数有个更可

读的名称error_handler，为此要把error_handler变成预处理器宏，扩展成名称eh73。

```
#define error_handler eh73
extern void error_handler();
…
error_handler("nil pointer error");
```

□

有些编译器可以用前面指定的字符之外的字符构成标识符，例如美元号（$），使程序可以访问一些计算系统提供的特殊非C语言库函数。

参考章节　#define命令　3.3；外部名称　4.2.9；关键字　2.6；多字节序列　2.1.5；保留库标识符　10.1.1；通用字符名　2.9

22

2.6 关键字

表2-4所示的标识符是标准C语言中的关键字，不能作为普通标识符，但可以作为宏名，因为所有预处理发生在识别这些关键字之前。关键字_Bool、_Complex、_Imaginary、inline与restrict是C99中增加的。

表2-4　C99关键字

auto	_Bool①	break	case
char	_Complex①	const	continue
default	restrict①	do	double
else	enum	extern	float
for	goto	if	_Imaginary①
inline	int	long	register
return	short	signed	sizeof
static	struct	switch	typedef
union	unsigned	void	volatile
while			

① 这些关键字是C99中增加的，C++中没有保留。

除了这些关键字之外，标识符asm与fortran是常用语言扩展。编程人员可以把头文件iso646.h(and、and_eq、bitand、bitor、compl、not、not_eq、or、or_eq、xor与xor_eq)中定义的宏当作保留字，这些标识符在C++中是保留字。

例　下列代码中可以使用与关键字拼写相同的宏。定义允许在非标准编译器建立的程序中使用void。

```
#ifndef __STDC__
#define void int
#endif
```

□

参考章节　_Bool　5.1.5；C++关键字　2.8；_Complex　5.2.1；#define命令　3.3；标识符　2.5；#ifndef命令　3.5；inline 9.10；<iso646.h>头文件　11.5；restrict 4.4.6；__STDC__ 11.3；void类型说明符　5.9

预定义标识符

尽管不是关键字，但C99中引入了预定义标识符的概念，并定义了一个预定义标识符：

__func__。与预定义宏不同的是，预定义标识符可以采用正常的标识符作用域规则。和关键字一样，预定义标识符不能由编程人员定义。 23

C99实现中隐式声明__func__标识符，就像在每个函数定义的花括号后面出现下列声明一样：

```
static const char __func__[] = "function-name";
```

调试工具可以利用这个标识符打印所包括的函数名，例如：

```
if (failed) printf("Function %s failed\n", __func__);
```

在内存较紧张的目标机器中转换C语言程序时，如果以上功能在运行时不是必需的，那么C语言实现就应当尽量避免使用这些字符串。

参考章节 函数定义 9.1；预定义宏 3.3.4；作用域 4.2.1

2.7 常量

常量的词法类包括4种不同常量：整型、浮点型、字符型和字符串型。

constant:
- *integer-constant*
- *floating-constant*
- *character-constant*
- *string-constant*

这种记号在其他语言中称为字面值，区别于数值是常数的对象（即数值不变），但不属于词法独立类。C语言中后一种对象的例子是枚举常量，其词法类为标识符。本书对两种情况都使用传统C语言术语“常量”。

每个常量具有数值和类型，下面几节介绍不同常量的格式。

参考章节 字符型常量 2.7.3；枚举常量 5.5；浮点数型量 2.7.2；整型常量 2.7.1；字符串型常量 2.7.4；记号 2.3；数值 7.3

2.7.1 整型常量

整型常量可以用十进制、八进制或十六进制方法指定。

integer-constant:
- *decimal-constant* *integer-suffix*$_{opt}$
- *octal-constant* *integer-suffix*$_{opt}$
- *hexadecimal-constant* *integer-suffix*$_{opt}$ 24

decimal-constant:
- *nonzero-digit*
- *decimal-constant* *digit*

octal-constant:
- **0**
- *octal-constant* *octal-digit*

hexadecimal-constant:
- **0x** *hex-digit*
- **0X** *hex-digit*
- *hexadecimal-constant* *hex-digit*

digit: one of

```
    0  1  2  3  4  5  6  7  8  9

nonzero-digit : one of
    1  2  3  4  5  6  7  8  9

octal-digit : one of
    0  1  2  3  4  5  6  7

hex-digit : one of
    0  1  2  3  4  5  6  7  8  9
    A  B  C  D  E  F  a  b  c  d  e  f

integer-suffix :
    long-suffix  unsigned-suffix_opt
    long-long-suffix  unsigned-suffix_opt                    (C99)
    unsigned-suffix  long-suffix_opt
    unsigned-suffix  long-long-suffix_opt                    (C99)

long-suffix : one of
    l  L

long-long-suffix : one of                                    (C99)
    ll  LL

unsigned-suffix : one of
    u  U
```

下列规则确定整型常量的基数：

1. 如果整型常量以**0x**或**0X**开头，则是十六进制表示，用字符**a**到**f**（或**A**到**F**）表示10到15。
2. 如果以数字**0**开头，则是八进制表示。
3. 否则是十进制表示。

25

整型常量可以用一个后缀字母指定其类型的最小长度：

- 字母**l**或**L**表示常量类型**long**
- 字母**ll**或**LL**表示常量类型**long long**(C99)
- 字母**u**或**U**表示**unsigned**类型（**int**、**long**或**long long**）

unsigned后缀与**long**或**long long**后缀可以按任意顺序组合。小写字母**l**很容易和数字**1**混淆，应避免在后缀中使用。

如果不发生溢出，则整型常量的值总是非负数。如果前面出现负号，则是对常量使用的一元运算符，而不是常量的一部分。

整型常量的实际类型取决于长度、基数、后缀字母和C语言实现确定的类型表示精度。确定类型的规则很复杂，在标准化前的C语言、C89和C99中各不相同。表2-5列出了所有规则。

如果整型常量的值超过表2-5中相应组的最后一种类型可以表示的最大整数，则结果是未定义的。在C99中，实现可能对这些大常量指定一个扩展整型类型，按照表中的符号规则处理（如果所有标准选择都是带符号的，则扩展类型应为带符号；如果所有标准选择都是无符号的，则扩展类型应为无符号的；否则有无符号均可接受）。在C89中，表示整型类型的信息在头文件

limits.h中提供。在C99中，文件**stdint.h**与**inttypes.h**还包含其他信息。

为了演示一些微妙的整型常量，假设类型**int**使用16位对二的补码表示，类型**long**使用32位对二的补码表示，类型**long long**使用64位对二的补码表示。表2-6列出了一些有意义的整型常量以及它们的实际数值、类型（习惯上和根据标准C语言规则）和存储这个常量的实际C语言表示。

这个表中很值得一提的是，$2^{15} \sim 2^{16}-1$范围内的整数写成十进制常量时为正值，而写成八进制或十六进制常量时为负值（转换成类型**int**）。尽管有这些异常现象，但编程人员很少对整型常量的值感到奇怪，因为即使类型有问题，常量表示还是相同的。

C99用**stdint.h**文件中定义了一些宏**INTN_C**、**UINT*N*_C**、**INTMAX_C**与**UINTMAX_C**，提供对整型常量的长度与类型的可移植性控制。

表2-5 整型常量类型

常 量	原始的C语言①	C89①	C99①②
dd...d	int long	int long unsigned long	int long long long
0*dd...d* 0X*dd...d*	unsigned long	int unsigned long unsigned long	int unsigned long unsigned long long long unsigned long long
dd...d U 0*dd...d* U 0X*dd...d* U	无	unsigned unsigned long	unsigned int unsigned long unsigned long long
dd...d L	long	long unsigned long	long long long
0*dd...d* L 0X*dd...d* L	long	long unsigned long	long unsigned long long long unsigned long long
dd...d UL 0*dd...d* UL 0X*dd...d* UL	无	unsigned long	unsigned long unsigned long long
*dd...d*LL	无	无	long long
0*dd...d* LL 0X*dd...d* LL	无	无	long long unsigned long long
dd...d ULL 0*dd...d* ULL 0X*dd...d* ULL	无	无	unsigned long long

① 所选类型是能表示常量而不溢出的相应组中第一种类型。

② 如果列出的类型都不够大，则可以用扩展类型（如果存在）。

例 如果**long**类型使用32位对二的补码表示，则下列程序确定有效规则：

```
#define K 0xFFFFFFFF /* -1 in 32-bit, 2's compl. */
#include <stdio.h>
```

```
int main()
{
   if (0<K) printf("K is unsigned (Standard C)\n");
   else printf("K is signed (traditional C)\n");
   return 0;
}
```
□

表2-6　对整型常量指定类型

C语言常量表示	实际值	传统类型	标准C语言类型	实际表示
0	0	int	int	0
32767	$2^{15}-1$	int	int	0x7FFF
077777	$2^{15}-1$	unsigned	int	0x7FFF
32768	2^{15}	long	long	0x00008000
0100000	2^{15}	unsigned	unsigned	0x8000
65535	$2^{16}-1$	long	long	0x0000FFFF
0xFFFF	$2^{16}-1$	unsigned	unsigned	0xFFFF
*65536	2^{16}	long	long	0x00010000
0x10000	2^{16}	long	long	0x00010000
2147483647	$2^{31}-1$	long	long	0x7FFFFFFF
0x7FFFFFFF	$2^{31}-1$	long	long	0x7FFFFFFF
2147483648	2^{31}	long①	unsigned long	0x80000000
			C99:long long	
0x80000000	2^{31}	long①	unsigned long	0x80000000
4294967295	$2^{32}-1$	long①	unsigned long	0xFFFFFFFF
			C99:long long	0x00000000FFFFFFFF
0xFFFFFFFF	$2^{32}-1$	long①	unsigned long	0xFFFFFFFF
4294967296	2^{32}	未定义	未定义	0x0
			C99:long long	0x0000000100000000
0x100000000	2^{32}	未定义	未定义	0x0
			C99:long long	0x0000000100000000

① 这个类型无法准确表示这个值。

参考章节　整型转换规则　6.2.3；扩展整型　5.1.4；整型　5.1；**INTMAX_C**　21.5；**INTN_C**　21.3；**limits.h**　5.1.1；溢出　7.2.2；**stdint.h**　第21章；一元负号运算符　7.5.3；无符号整数　5.1.2

2.7.2　浮点型常量

浮点型常量可以写成带小数点的、带符号指数的或同时包含两者的十进制数。标准C语言可以用后缀字母（浮点型后缀）指定**float**与**long double**类型的常量。如果不用后缀，则常量类型为**double**。

floating-constant :
　　decimal-floating-constant
　　hexadecimal-floating-constant　　*(C99)*

decimal-floating-constant :
　　digit-sequence exponent $floating\text{-}suffix_{opt}$
　　dotted-digits $exponent_{opt}$ $floating\text{-}suffix_{opt}$

digit-sequence :

digit
digit-sequence *digit*

dotted-digits:
digit-sequence .
digit-sequence . *digit-sequence*
. *digit-sequence*

digit: one of
0 **1** **2** **3** **4** **5** **6** **7** **8** **9**

exponent:
e *sign-part*$_{opt}$ *digit-sequence*
E *sign-part*$_{opt}$ *digit-sequence*

sign-part: one of
+ **-**

floating-suffix: one of
f **F** **l** **L**

如果不发生溢出，则浮点型常量的值总是非负数。如果前面出现负号，则是对常量采用的一元运算符，而不是常量的一部分。如果浮点型常量无法准确表示，则实现可能选择最接近的可表示值V或V周围较大或较小的可表示值。如果浮点型常量的值太大或太小，无法表示，则结果无法预测，这时有些编译器会向编程人员发出问题警报，但大多数编译器只是自作主张，用其他一些便于表示的值来替换。在标准C语言中，对于浮点数的限制记录在**float.h**头文件中。**math.h**中定义了特殊浮点型常量，例如无限大和NaN（非数字）。

在C99中，复数浮点型常量用包含了**complex.h**中定义的虚数常量**_Complex_I**(或**I**) 的浮点型常量表达式来表示。

例 下面是有效十进制浮点型常量：**0.**、**3e1**、**3.14159**、**.0**、**1.0E-3**、**1e-3**、**1.0**、**.00034**、**2e+9**。下列浮点型常量在标准C语言中是有效的：**1.0f**、**1.0e67L**、**0E1L**。**1.0+1.0*I**是C99复数常量的例子（如果包含了**complex.h**头文件）。 □

C99允许用十六进制方法表示浮点型常量，而旧版C语言中只有十进制浮点型常量。十六进制格式用字母**p**分开小数与指数部分，因为习惯用字母**e**可能与十六进制数字混起来。二进制指数是表示2的指数的带符号十进制数（不是十进制浮点型常量中的10的指数，也不是16的指数）。

hexadecimal-floating-constant: *(C99)*
hex-prefix *dotted-hex-digits* *binary-exponent* *floating-suffix*$_{opt}$
hex-prefix *hex-digit-sequence* *binary-exponent* *floating-suffix*$_{opt}$

hex-prefix:
0x
0X

dotted-hex-digits:
hex-digit-sequence .
hex-digit-sequence . *hex-digit-sequence*
. *hex-digit-sequence*

hex-digit-sequence:
　　hex-digit
　　hex-digit-sequence　*hex-digit*

binary-exponent:
　　p　*sign-part*$_{opt}$　*digit-sequence*
　　P　*sign-part*$_{opt}$　*digit-sequence*

如果**FLT_RADIX(float.h)**不等于2，则可能无法准确表示十六进制浮点型常量。如果无法准确表示，则指定的值要正确舍入到最接近的可表示值。

参考章节　**complex.h**　23.2；**double**类型　5.2；**float.h**　5.2；上溢与下溢　7.2.2；浮点型长度　5.2；一元负号运算符　7.5.3

2.7.3 字符型常量

字符型常量把一个或几个字符放在一对撇号中。可以用特殊转义机制编写不方便或无法直接在源程序中输入的字符或数字值。标准C语言允许在字符型常量前面加上字母**L**来指定宽字符常量。

character-constant:
　　' *c-char-sequence* **'**
　　L' *c-char-sequence* **'**　　*(C89)*

c-char-sequence:
　　c-char
　　c-char-sequence　*c-char*

c-char:
　　any source character except the apostrophe (**'**), backslash (\), or newline
　　escape-character
　　universal-character-name　　*(C99)*

通过使用转义符可以将撇号、反斜杠和换行符加入到字符型常量中，见2.7.5节。对源程序中一些
30 不易读的字符（如格式符）最好使用转义符。C99可以在字符型常量中使用通用字符名（见2.9节）。

没有在前面加上字母**L**的字符型常量为**int**类型，这种字符型常量通常是单个字符或转义码（见2.7.7节），其常量值为执行字符集中相应字符的整数编码。计算得到的整数值就像从**char**类型的对象转换过来一样。例如，如果**char**类型为8位带符号类型，则字符常量**'\377'**进行符号扩展，得到**-1**。下列情况下的字符型常量是由实现定义的：

1. 执行字符集中没有相应字符。
2. 常量中出现多个执行字符。
3. 数字转义的值在执行字符集中没有表示。

例　下面是一些单字符常量及其ASCII编码的十进制值。

字　符	取　值	字　符	取　值
'a'	**97**	**'A'**	**65**
' '	**32**	**'?'**	**63**
'\r'	**13**	**'\0'**	**0**
'"'	**34**	**'\377'**	**255**
'%'	**37**	**'\23'**	**19**
'8'	**56**	**'\\'**	**92**

□

标准C语言宽字符常量用前缀字母L指定，其类型为**wchar_t**，是头文件**stddef.h**中定义的整型类型，目的是让C语言编程人员能够表示**char**类型无法表示的长字母表中的字符（如日文字母）。宽字符常量通常包括字符序列和转义码，一起构成一个多字节字符。多字节字符与相应宽字符的映射是由实现定义的，对应于**mbtowc**函数，该函数在运行时进行转换。如果多字节字符使用shift状态编码，则宽字符常量要以初始shift状态开始和结束。如果宽字符常量包含多个宽字符，则其数值由实现定义。

多字符常量 整型和宽字符常量可以包含字符序列，将这个序列映射执行字符集之后，可能还有多个执行字符。这种常量的含义是由实现定义的。

较早的实现中的一个约定是把四字节整型常量表示为4个宽字符常量，如**'gR8t'**。这种用法是不可移植的，因为有些实现可能不允许，而且不同实现的整数长度和字节顺序（即把字符组装成单词的顺序）可能不同。 31

例 在四字节整数和从左向右组装的ASCII实现中，**'ABCD'**的值为41424344_{16}（其中**'A'**的值为**0x41**，**'B'**的值为**0x42**等等）。而如果使用从右向左组装，则**'ABCD'**的值为44434241_{16}。□

参考章节 ASCII字符 附录A；字节顺序 6.1.2；字符编码方式 2.1；**char**类型 5.1.3；转义符 2.7.5；格式符 2.1；**mbtowc**函数 11.7；多字节字符 2.1.5；**wchar_t** 11.1

2.7.4 字符串型常量

字符串型常量是双引号中的字符序列（可能是空的）。可以用字符常量所用的转义机制表示字符串中的字符。标准C语言允许在字符串型常量前面加上**L**前缀来指定宽字符串常量。

string-constant:
　　" *s-char-sequence*$_{opt}$ "
　　L" *s-char-sequence*$_{opt}$ "　　*(C89)*

s-char-sequence:
　　s-char
　　s-char-sequence s-char

s-char:
　　any source character except the double quote ",
　　backslash \, or newline character
　　escape-character
　　universal-character-name　　*(C99)*

通过使用转义符可以将双引号、反斜杠和换行符加入到字符串型常量中，见2.7.5节。对源程序中一些不易读的字符（如格式符）最好使用转义字符。C99可以在字符串型常量中使用通用字符名（见2.9节）。

例 下面列出5个字符串常量：

```
""
"\""
"Total expenditures: "
"Copyright 2000 \
Texas Instruments. "
"Comments begin with '/*'.\n"
```

32

第4个字符串等于"**Copyright 2000 Texas Instruments.**"，**0**和**T**之间不包含

换行符。 □

对每个n字符的非宽字符串常量，运行时静态分配$n+1$个字符的内存块，其中前n个字符是字符串中的字符，最后一个字符是null字符'\0'。这个内存块是字符串常量的值，类型为**char [n+1]**。同样，宽字符串常量变成n个宽字符加上一个宽null字符，类型为**wchar_t [n+1]**。

例 **sizeof**操作符返回操作数的长度，而**strlen**函数（见13.4节）返回字符串中的字符数。因此，**sizeof("abcdef")**的返回值是7而不是6，**sizeof("")**的返回值是1而不是0而**strlen("abcdef")**的返回值是6，**strlen("")**的返回值是0。

如果字符串型常量不是地址运算符**&**或**sizeof**操作符的参数，也不是字符数组的初值，则要进行普通数组转换，将字符串从字符数组变成指向字符串中第一个字符的指针。

例 声明**char *p = "abcdef";**将指针**p**初始化成存储7个字符**'a'**、**'b'**、**'c'**、**'d'**、**'e'**、**'f'**与**'\0'**的内存块地址。

单字符字符串型常量的值与字符型常量的值大不相同。声明**int X=(int)"A";**得到的结果是将**X**初始化成指向包含**'A'**和**'\0'**两个字符的内存块的指针（如果这个指针能表示为**int**类型），而声明**int Y = (int)'A';**则把**Y**初始化为**'A'**的字符编码（在ISO 646编码方式中为**0x41**）。 □

存储字符串型常量 不能修改保存字符串型常量字符的内存，因为这个内存可能是只读的，即物理上是防止修改的。有些函数（如**mktemp**）要接收就地修改的字符串指针，此时不要向这些函数传递字符串型常量，而要将这个字符串型常量的内容初始化到一个非**const**字符数组中，然后传递数组第一个元素的地址。

例 考虑下列3个声明：

```
char p1[]= "Always writable";
char *p2 = "Possibly not writable";
const char p3[] = "Never writable"; /* Standard C only */
```

p1、**p2**与**p3**的值都是字符数组的指针，但其可写性不同。赋值语句**p1[0]='x'**总是可行的，**p2[0]='x'**可能可行，也可能造成运行错误，而**p3[0]='x'**总是会造成编译错误，这是由**const**的含义决定的。 □

33

不要认为所有字符串型常量都存放在不同地址，标准C语言允许实现对包含相同字符的两个字符串型常量使用同一存储空间。

例 下面的简单程序区别字符串的不同实现。如果在只读内存中分配字符串型常量，则对**string1[0]**的赋值语句可能造成运行错误。

```
char *string1, *string2;
int main() {
    string1 = "abcd"; string2 = "abcd";
    if (string1==string2) printf("Strings are shared.\n");
    else printf("Strings are not shared.\n");
    string1[0] = '1'; /* RUN-TIME ERROR POSSIBLE */
    if (*string1=='1') printf("Strings writable\n");
    else printf("Strings are not writable\n");
    return 0;
}
```

□

字符串的续行 字符串型常量通常写在一个源程序行中。如果字符串太长，在一行中放不下，则除包含该字符串的最后一行以外的其他行都可以用反斜杠结尾，这样，反斜杠和行终结符被忽略，使字符串型常量可以写在多行中。一些较早的实现可能从续行中删除前面的空白符，但这么做是不正确的。

标准C语言自动接合相邻的字符串型常量和相邻的宽字符串常量，它会在最后一个字符串末尾放置一个null字符。因此，除了标准C语言程序中的\续行机制外，也可以把长字符串分解为多个字符串。在C99中，宽字符串和正常字符串常量也可以这样接合，得到宽字符串常量，而C89中则不允许这样。

例 标准C语言和标准化前的C语言编译器都接受对**s1**进行的字符串初始化操作，但只有标准C语言接受对**s2**进行字符串初始化的方式。

```
char s1[] = "This long string is acc\
eptable to all C compilers.";
char s2[] = "This long string is permissible "
            "in Standard C.";
```

□

通过在字符串型常量中放置转义序列**\n**即可在字符串中插入换行符（即执行字符串中的行终结符），注意不要与字符串型常量中的续行符混起来。

宽字符串 前面冠以**L**字母的字符串型常量是标准C语言宽字符串常量，类型为**wchar_t**的数组，表示扩展执行字符串中的宽字符序列，适用于日语之类的语言。宽字符串常量中的字符 34
是多字节字符串，映射到实现定义方式中的宽字符序列（**mbstowcs**函数在运行时完成类似功能）。如果多字节字符用shift状态编码，则宽字符串常量应以初始shift状态开始和结束。

参考章节 数组类型 5.4；**const**类型指定符 4.4.4；数组类型版本 6.2.7；转义符 2.7.5；初值 4.6；**mbstowcs**函数 11.8；**mktemp**函数 15.16；多字节字符 2.1.5；指针类型 5.3；预处理器词法约定 3.2；**sizeof**运算符 7.5.2；**strlen**函数 13.4；空白符 2.1；普通一元转换 6.3.3；**wchar_t** 11.1；通用字符名 2.9

2.7.5 转义符

转义符可以在字符型和字符串型常量中表示源程序中很难或无法直接输入的字符。转义符有两类："字符转义"可以表示特定格式和特殊字符，"数字转义"可以用字符的数字编码指定字符。C99中还包括通用字符名。

escape-character：
　　\ *escape-code*
　　universal-character-name *(C99)*

escape-code：
　　character-escape-code
　　octal-escape-code
　　hex-escape-code *(C89)*

character-escape-code：one of
　　n t b r f
　　v \ ' "
　　a ? *(C89)*

octal-escape-code:
　　octal-digit
　　octal-digit octal-digit
　　octal-digit octal-digit octal-digit

hex-escape-code:
　　x *hex-digit*
　　hex-escape-code hex-digit　　*(C89)*

下面几节将介绍这些转义符的含义。

如果反斜杠后面的字符不是八进制数字、字母x或前面列出的字符转义符，则结果是未定义的（在传统C语言中，忽略反斜杠）。在标准C语言中，反斜杠后面的所有小写字母保留用于未来
35 的语言扩展，大写字母可以用于实现特定扩展。

参考章节　通用字符名　2.9

2.7.6 字符转义符

字符转义符以独立于目标计算机字符集的方式表示一些常用特殊字符。表2-7列出了反斜杠后面的字符及其含义。

表2-7　字符转义符

转义符	说　明	转义符	说　明
a①	警报（如铃声）	**v**	垂直制表符
b	退格符	****	反斜杠
f	换页符	**'**	单引号
n	换行符	**"**	双引号
r	回车符	**?**①	问号
t	水平制表符		

① 标准C语言增加的。

代码**\a**通常映射成警报（铃声）或外部设备上的其他可听信号（如ASCII control-G，取值为7）。**\?**在少数情况下，例如可能与三字符组混淆时用于取代问号。

引号(")放在字符常量中时，前面可以不带反斜杠，撇号(')放在字符串常量中时，前面可以不带反斜杠。

例　下列程序用于计算输入中的行数（实际上是换行符个数），以此为例显示字符转义符的用法。函数**getchar**返回下一个输入的字符，直至达到输入结束，此时**getchar**返回
36 **stdio.h**中宏定义**EOF**的值。

```
#include <stdio.h>
int main(void) /* Count the number of lines in the input. */
{
    int next_char;
    int num_lines = 0;
    while ((next_char = getchar()) != EOF)
        if (next_char == '\n')
            ++num_lines;
    printf("%d lines read.\n", num_lines);
    return 0;
}
```

□

参考章节 字符型常量 2.7.3；**EOF** 15.1；**getchar**函数 15.6；**stdio.h** 15.1；字符串型常量 2.7.4；三字符组 2.1.4

2.7.7 数字转义符

数字转义符可以把执行字符集中的字符表示为八进制或(标准C语言中的)十六进制值编码值。最多可以出现3个八进制位或任意个十六进制位，但标准C语言对正常字符型常量禁止**unsigned char**范围以外的值，对宽字符常量禁止**wchar_t**范围以外的值。例如，在ASCII编码方式中，字符**'a'**可以写成**'\141'**或**'\x61'**，字符**'?'**可以写成**'\77'**或**'\x3F'**。null字符用于终止字符串，总是写成**\0**。在执行字符集中没有对应字符的数字转义符的值是由实现定义的。

例 下列代码段显示了数字转义符的用法。变量**inchar**的类型为**int**。

```
for (;;) {
    inchar = receive();
    if (inchar == '\0') continue;          /* Ignore */
    if (inchar == '\004') break;           /* Quit */
    if (inchar == '\006') reply('\006');   /* ACK */
    else reply('\025');                    /* NAK */
    }
```

□

编程人员使用数字转义符时有两点需要注意。第一，使用数字转义符可能依赖于字符编码方式，因此是不可移植的。最好把转义符隐藏在宏定义中，便于修改。

```
#define NUL '\0'
#define EOT '\004'
#define ACK '\006'
#define NAK '\025'
```

37

第二，数字转义符的语法是独特的，八进制转义符在用完3个八进制位之后或遇到第一个非八进制位时终止，因此字符串**"\0111"**包含两个字符**\011**和**1**，字符串**"\090"**包括3个字符**\0**、**9**和**0**。十六进制转义序列由于其长度任意也可能遇到终止问题。为了终止字符串中的标准C语言十六进制转义，可以把字符串分段。

```
"\xabc"      /* 这个字符串包含一个字符。 */
"\xab" "c"   /* 这个字符串包含两个字符。 */
```

一些非标准C语言实现提供的十六进制转义序列和八进制转义序列一样只允许固定的十六进制位数。

参考章节 字符型常量 2.7.3；**#define** 3.3；宏定义 3.3；null字符 2.1；字符串型常量 2.7.4；执行字符集 2.1

2.8 C++兼容性

本节介绍C语言与C++之间的词法差别。

2.8.1 字符集

C++标准支持标准C语言中的记号重拼和三字符组，但在标准化前的C++实现中不常见。C和C++都支持通用字符名的相同语法，但只有C语言明确允许标识符中使用其他实现定义的字符(人们希望C++能在实现时提供这种扩展)。

2.8.2 注释语句

C99注释语句可以在C++中接受，反之亦然。在C99之前，标准C语言不能用字符//引入注释语句，因此C语言中的字符序列//*在C++中可能有不同解释（具体解释留作练习）。

2.8.3 运算符

C++中增加了3个新的复合运算符：

```
.*    ->*    ::
```

由于这些记号组合在标准C语言程序中无效，因此不会影响从C向C++的可移植性。

2.8.4 标识符与关键字

38

表2-8列出的标识符是C++关键字，而不是C语言关键字。但是，标准C语言保留关键字**wchar_t**，C99的标准库中保留关键字**bool**、**true**和**false**。

表2-8　其他C++关键字

asm	export	private	throw
bool	false	protected	true
catch	friend	public	try
class	mutable	reinterpret_cast	typeid
const_cast	namespace	static_cast	typename
delete	new	template	using
dynamic_cast	operator	this	virtual
explicit			wchar_t

2.8.5 字符型常量

C语言中单字符常量的类型为**int**，而C++中的类型为**char**。多字符常量是实现定义的，在两种语言中均为**int**类型。实际上，这样并没有什么差别，因为按照通常转换规则，C++字符型常量会升级为**int**类型。但是，**sizeof('c')**在C++中为**sizeof(char)**，而在C语言中为**sizeof(int)**。

2.9 字符集、指令集和编码方式

最初设计C语言时，还没有很好地了解国际化、多语种编程群体的需求。标准C语言为了满足这个群体要求而对C语言进行扩展。本节简要介绍标准C语言为了对非英语用户更友好而做的工作和存在的问题。

指令集与ASCII　每种文化都以一组可打印字母或符号的字符集进行书面交流作为自己的基础。对于美式英语，这个字符集包括通常的52个大小写字母、十进制数和一些标点符号，共100多个字符，用称为ASCII的七位编码方式指定特定二进制值（由美式英语编程人员和计算机厂家指定）。这些编码字符出现在标准键盘上，应用于C语言定义等地方。

但是，其他文化采用不同字符集。例如，英国说英语的人群更常使用 而不用$，而七比特

39

ASCII中不包含这个符号。俄语和希伯莱语使用完全不同的字母表，中文/日文/朝鲜文的字符集中有几千个符号。今天的编程人员希望编写的C语言程序能够用多种语言读写文本，包括他们自己的母语，他们还希望在程序中使用母语的注释和变量名。这样编写的程序可以移植到其他字符集中，而不至于变成无效程序（当然，要阅读梵文注释，除非您懂梵文，而且您的计算机能显示梵文字符）。

这个问题是被逐渐认识的，当时已经出现几种部分解决方案，并且至今仍然在使用。例如，

ISO 646-1083不变代码集是ASCII的子集，是许多非英语字符集之间共用的集合，人们已经提出一些方法替换这个不变集中没有的C语言字符，如{、}、[、]与#。

ISO/IEC 10646 ISO/IEC 10646标准（及其增补）定义了字符集的一般解决方案，即通用多八位组编码字符集（UCS）。它定义了四字节（或4八位组）编码UCS-4，在表示了全世界所有文字中的所有字符之外还有很多空余的编码。UCS-4的16位子集称为基本多语种平面（UCS-2），由头两字节为0的UCS-4编码组成。UCS-2可以表示各主要语言字符集，包括大约两万个中文/日文/朝鲜文表意字符。但是，16位通常是不够的，而超过32位的值又很难在计算机上操纵，因此定义了UCS-4。

Unicode字符集标准最初是由Unicode联盟(**www.unicode.org**)建立的十六比特编码方式。Unicode 3.0与ISO/IEC 10646标准完全兼容，而旧版Unicode只与UCS-2兼容。Unicode的Web站点有字符编码技术的详细介绍。

UCS-4、UCS-2与Unicode字符集标准与ASCII兼容。高8位为0的十六比特字符就是八比特扩展ASCII字符集，现在称为Latin-1字符集。原始的七比特ASCII字符集现在称为基本拉丁字符集，是高9位为0的UCS-2字符集。

宽字符与多字节字符 超过传统8位表示的字符称为宽字符。但是，八比特（或七比特）字符不容易根除。许多计算机和遗留应用程序都基于八比特字符，人们提出了许多用八比特（或七比特）字符序列表示大字符集和宽字符的方案，称为多字节编码或多字节字符。宽字符都使用定长表示，而多字节字符通常对某些字符使用一个字节，对另一些字符使用两个字节，对其他一些字符使用3个字节，等等。一个或几个八比特字符作为转义或shift字符以表示多字节序列的开始。

今天在标准C语言中看到的是处理明显ASCII变形（三字符组和两字符组）的方法、处理完全现代宽字符环境的方法、处理I/O期间多字节字符序列的方法以及最新的按可移植方式表示适用于任何字符集的C语言程序的方法（标识符中的通用字符和特定区域字符等技术的组合）。 40

通用字符名 C99引入了一种表示法，可以在字符型常量、字符串型常量和标识符中指定UCS-2与UCS-4字符。其语法如下：

universal-character-name:
    \u *hex-quad*
    \U *hex-quad hex-quad*

hex-quad:
    hex-digit hex-digit hex-digit hex-digit

每个*hex-quad*是4个十六进制数，可以表示十六比特字符的值。*hex-quad*的值在ISO/IEC 10646标准中指定为4位，通用字符用8位“短标识符”表示。**\u***nnnn*指定的字符与**\U0000***nnnn*指定的字符相同。

C语言不允许短标识符小于00A0的通用字符名（**0024**($)、**0040**(@)与**0060**(`)除外），也不允许短标识符是介于**D800**到**DFFF**范围中的通用字符。这些是控制符，包括DELETE和为UTF-16保留的字符。用记号合并生成通用字符名的结果是未定义的。

参考章节 标识符与通用字符名 2.5；记号合并 3.3.9

2.10 练习

1. 下列哪些是词法记号？

(a) 关键字　　(e) 三字符组
(b) 注释　　(f) 宽字符串常量
(c) 空白符　　(g) 括号
(d) 十六进制常量

2. 假设用标准C语言编译器处理下列源字符串，每一源字符串中的哪些字符串将被作为C语言记号序列处理？每个源字符串中有多少个记号？（不必考虑有些记号序列能否在有效C语言程序中出现）。

(a) `X++Y`　　(f) `x**2`
(b) `-12uL`　　(g) `"X??/"`
(c) `1.37E+6L`　　(h) `B$C`
(d) `"String ""FOO"""`　　(i) `A*=B`
(e) `"String+\"FOO\""`　　(j) `while##DO`

3. 从下列C语言程序段中消除所有注释。

```
/**/*/*"*/*/*"//*//**/*/
```

4. 标准C语言编译器要对输入程序进行下列操作，这些操作正确的顺序是什么？

把字符组合成记号
删除解释
转换三字符组
41　处理续行

5. 下面是一些不良程序标识符，它们的问题在哪里？

(a) `pipesendintake`　　(d) `077U`
(b) `Const`　　(e) `SYS$input`
(c) `1O`

6. 用标准C语言编写一些简单代码段，这些代码段将因为下列原因在C++中无效或具有不同解释：

(a) C89中不支持//风格的注释
(b) 常量类型
42　(c) 关键字冲突

第3章 C语言预处理器

C语言预处理器是个简单的宏处理器，从概念上处理C程序源文本之后再让编译器正确读取源程序。在C语言的有些实现产品中，预处理器实际上是一个独立的程序，它读取原始的源文件并输出新的经过预处理的源文件，以作为C语言编译器的输入。在另一些实现产品中，一个程序一次性对源文件进行预处理和编译。

3.1 预处理器命令

预处理器用特殊的预处理器命令行控制，这些命令行以字符#开头，不包含预处理器命令的行称为源程序文本行。表3-1列出了预处理器命令。

预处理器通常从源文件中删除所有预处理器命令行，并按预处理器命令指示对源文件进行其他转换，如扩展源程序文本行中出现的宏调用。然后，得到的经过预处理的源文本成为有效C语言程序。

预处理器命令的语法完全独立于C语言中其余部分的语法（但有一定相似之处）。例如，宏定义可以扩展成语法化的不完整的代码段，只要这个段在调用宏的所有上下文中有意义（即正确完成）。

表3-1 预处理器命令

命 令	含 义	参考章节
`#define`	定义预处理器宏	3.3
`#undef`	取消预处理器宏定义	3.3.5
`#include`	插入另一源文件中的文本	3.4
`#if`	根据常量表达式值有条件地包括一些文本	3.5.1
`#ifdef`	根据是否定义宏名有条件地包括一些文本	3.5.3
`#ifndef`	根据与`#ifdef`相反的测试有条件地包括一些文本	3.5.3
`#else`	上述`#if`、`#ifdef`、`#ifndef`或`#elif`测试失败时包括一些文本	3.5.1
`#endif`	终止条件文本	3.5.1
`#line`	提供编译器消息的行号	3.6
`#elif`①	上述`#if`、`#ifdef`、`#ifndef`或`#elif`测试失败时根据另一常量表达式值包括一些文本	3.5.2
`defined`②	预处理器函数，在一个名称定义为预处理器宏时为1，否则为0，在`#if`与`#elif`中使用	3.5.5
#运算符②	将宏参数换成包含参数值的字符串常量	3.3.8
##运算符②	从两个相邻记号生成一个记号	3.3.9
`#pragma`②	对编译器指定实现相关信息	3.7
`#error`②	将编译错误换成指定消息	3.8

① 最初不属于C语言，但现已在ISO和非ISO实现中很常见。

② 标准C语言中增加的命令。

3.2 预处理器词法规则

预处理器不分析源文本，而是把源文本分解为记号，以便找到宏调用。预处理器词法规则

不同于编译器，预处理器识别出正常的C语言记号，并把C语言中无法识别为有效字符的其他字符也作为“记号”。这样就使预处理器可以识别出文件名、是否存在空白符和行末标志位置。

以#开始的行是预处理器命令，#后面是命令名。标准C语言允许同一源行的#前后出现空白符，但有些早期编译器不允许。一行中惟一的非空白符为#的情况，在标准C语言中称为null指令，相当于空行处理，较早的实现可能有不同的处理方法。

44 命令名后面的内容可以包含适当的命令参数。如果预处理器命令不带参数，则命令行中命令名后面应为空，但可以保留空白符或注释语句。许多ISO之前的编译器指令忽略所需参数后面的所有字符，这样可能造成移植性问题。预处理器命令参数通常要进行宏替换。

预处理器命令行在宏扩展之前识别，因此，如果宏扩展之后像预处理器命令，则标准C语言和大多数其他C语言编译器的预处理器无法识别这个命令（一些较早的UNIX实现可能会违背这个规则）。

例 下列代码的结果不是在编译的程序中包括文件**math.h**：

```
/* This example doesn't work as one might think! */
#define GETMATH #include <math.h>
GETMATH
```

面是把展开的记号序列传入和编译成（错误的）C语言代码：

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

□

2.1.2节曾介绍过，所有源行（包括预处理器命令行）都可以续行，只要在行末标志前面加一个反斜杠（\）。这一操作发生在扫描预处理器命令之前。

例 预处理器命令

```
#define err(flag,msg)  if (flag) \
   printf(msg)
```

相当于

```
#define err(flag,msg)  if (flag) printf(msg)
```

如果在行末标志前面加一个反斜杠（\），则下列两行

```
#define BACKSLASH \
#define ASTERISK *
```

将作为单个预处理器命令

```
#define BACKSLASH #define ASTERISK *
```

□

2.2节曾介绍过，预处理器把注释语句当作空白符，注释语句中的行终结符并不终止预处理器命令。

45 **参考章节** 注释　2.2；行终止与续行　2.1；记号　2.3

3.3 定义和替换

#define预处理器命令把名称（标识符）定义为预处理器的宏，称为*宏体*的记号序列与这个名称相关联。程序源文本或某些其他预处理器命令的参数中识别到宏名时，将视为对宏的调用，用宏体的拷贝替换这个宏名。如果宏定义为接受参数，则用宏名后面的实际参数替换宏体

中的正式参数。

例 如果宏**sum**用下列语句定义两个参数：

```
#define sum(x,y)  x+y
```

则预处理器将源程序行

```
result = sum(5,a*b);
```

使用简单文本替换为

```
result = 5+a*b;
```

□

由于预处理器不区别保留字与其他标识符，因此原则上可以用C语言保留字作为预处理器宏，但这是个不好的编程习惯。注释语句、字符串常量与字符常量和**#include**文件名中的宏名无法识别。

3.3.1 对象式宏定义

#define命令有两种形式，取决于左括号与要定义的宏名是否靠在一起。简单对象式形式没有左括号：

#define *name sequence-of-tokens*$_{opt}$

对象式宏没有参数，只是按名称调用。源程序文本中遇到宏名时，将其换成宏体（相关联的*sequence-of-tokens*，可能是空的）。**#define**命令语法不要求在定义的名称后面放上等号或任何其他特殊分隔记号，宏名后面就是宏体。

对象式宏在程序中引入命名常量时特别有用，使表格长度之类的“幻数”可以在一个位置编写，然后在其他任意位置按名称引用，这样，以后要改变数字时更加容易。

对象式宏的另一个重要用途是分隔外部定义函数名与变量名中的实现相关限制。2.5节举了一个例子。 46

例 下面是一些典型的宏定义：

```
#define BLOCK_SIZE 0x100
#define TRACK_SIZE (16*BLOCK_SIZE)
#define EOT '\004'
#define ERRMSG "*** Error %d: %s.\n"
```

一个常见的编程错误是包括多余的等号：

```
#define NUMBER_OF_TAPE_DRIVES = 5  /* Probably wrong. */
```

这是个有效定义，但把**NUMBER_OF_TAPE_DRIVES**定义为“**=5**”而不是“**5**”。如果编写下列代码段：

```
if (count != NUMBER_OF_TAPE_DRIVES) …
```

则会展开如下：

```
if (count != = 5) …
```

这在语法上是无效的。同样，还要避免多余的分号。

```
#define NUMBER_OF_TAPE_DRIVES 5 ;  /* Probably wrong. */
```

□

参考章节 复合赋值运算符 7.9.2；运算符与分隔符 2.4

3.3.2 函数式宏定义

更复杂的函数式宏定义声明正式参数名，放在括号中，用逗号分隔：

#define *name*(*identifier-list*$_{opt}$) *sequence-of-tokens*$_{opt}$

其中*identifier-list*是逗号分隔的参数名列表。C99中省略号（...）可以放在*identifier-list*之后，表示可变参数表，见3.3.10节介绍。目前只考虑固定参数表。

左括号与宏名之间不能有空格。如果左括号与宏名之间有空格，则定义变成不带参数的宏和以左括号开始的宏体。

正式参数名应为标识符，参数不能重名，宏体中不一定要提到这些参数名（但通常都会提到）。函数式宏可以采用空参数表（即0个正式参数），这种宏可以模拟不带参数的函数。

函数式宏取多个实际参数作为正式参数。调用宏时，要写入名称、左括号、每个正式参数
47 的实际参数记号序列和右括号。实际参数记号序列用逗号分开（调用不带正式参数的函数式宏时，要提供空的实际参数表）。调用宏时，宏名和左括号之间或实际参数中可以出现空白（一些早期的有缺陷的预处理器实现不允许实际参数记号表延伸到多行，除非用 \ 续行）。

实际参数记号序列可以包含括号，但要正确嵌套和平衡；可以包含逗号，但每个逗号出现在一组括号中（这个限制可以避免与分隔实际参数的逗号混淆）。宏参数中也可以出现花括号和下标方括号，但不能包含逗号，也不需要平衡。字符型常量与字符串型常量记号中的括号和逗号不计入括号平衡和分隔实际参数的逗号。

在C99中，宏的参数可以是空的，即不包含记号。

例 下面的宏将两个参数相乘：

```
#define product(x,y)  ((x)*(y))
```

在下列语句中调用两次：

```
x = product(a+3,b) + product (c, d);
```

product宏的参数可以是函数（或宏）调用，函数参数表中的逗号不影响宏参数的分析：

```
return product( f(a,b), g(a,b) ); /* OK */
```

例 **getchar**宏的参数表是空的：

```
#define getchar()  getc(stdin)
```

调用时，提供空参数表（**getchar**、**stdin**与**EOF**在标准头文件**stdio.h**中定义）：

```
while ((c=getchar()) != EOF) …
```

例 我们还定义一个取任意语句为参数的宏：

```
#define insert(stmt)  stmt
```

则下列调用是正确的：

```
insert( {a=1; b=1;} )
```

48 但如果将两个赋值语句改为包含两个赋值表达式的一条语句

```
insert( {a=1, b=1;} )
```

则预处理器会报告说插入的宏参数太多。要解决这个问题，可以改写如下：

```
insert( {(a=1, b=1);} )
```

例 定义语句上下文中使用的函数式宏比较复杂。下列宏交换两个参数**x**与**y**中的值，假设其数

值类型可以换算为**unsigned long**，并且换算回原类型后不发生改变，且不涉及**_temp**标识符。

```
#define swap(x, y) { unsigned long _temp=x; x=y; y=_temp; }
```

问题是，我们习惯于在**swap**之后放一个分号，

```
if (x > y) swap(x, y); /* Whoops! */
else x = y;
```

就好像**swap**是个实际函数一样，这会出错，因为扩展后的语句包括多余的分号（见8.1节）。我们把扩展语句放在另一行，可以更清楚地说明这个问题：

```
if (x > y) { unsigned long _temp=x; x=y; y=_temp; }
;
else x = y;
```

要避免这个问题，一个聪明的办法是将宏体定义为**do-while**语句，这样可以吸收多余的分号（见8.6.2节）：

```
#define swap(x, y) \
   do { unsigned long _temp=x; x=y; y=_temp; } while (0)
```

□

遇到函数式宏调用时，处理参数之后用宏体的处理拷贝替换整个宏调用。参数处理的过程如下。实际参数记号字符串与相应正式参数名相关联，然后建立宏体拷贝，将每个正式参数名换成与其相关联的实际参数记号序列拷贝。然后用这个宏体拷贝替换宏调用。用宏体的处理拷贝替换宏调用的全过程称为*宏扩展*，宏体的处理拷贝称为宏调用的*扩展*。

例 下列宏定义可以方便地通过循环计数某一数值范围内所有数。

```
#define incr(v,low,high)  \
   for ((v) = (low); (v) <= (high); (v)++)
```

49

要打印1到20的整数立方表，可以编写如下程序：

```
#include <stdio.h>
int main(void)
{
   int j;
   incr(j, 1, 20)
      printf("%2d %6d\n", j, j*j*j);
   return 0;
}
```

调用宏**incr**时扩展成产生下列循环：

```
for ((j) = (1); (j) <= (20); (j)++)
```

正确地使用括号可以保证编译器不会错误地解释这么复杂的实际参数（见3.3.6节）。 □

参考章节 **do**语句 8.6.2；语句语法 8.1；**unsigned long** 5.1.2；空白符 2.1.2

3.3.3 重新扫描宏表达式

扩展宏调用之后，宏调用的扫描恢复到扩展开头，这样可以识别扩展中的宏名，以便进一步进行宏替换。处理命令和定义宏名时，不对**#define**命令的任何部分进行宏替换，也不对宏体中的宏名进行替换。只有对特定宏调用扩展宏体之后，才识别宏体中的宏名。

扫描宏调用时，也不对函数式宏调用实际参数记号字符串中的宏名进行宏替换。假设相应的正式参数实际在宏体中出现一次或几次，那么只有在重新扫描扩展时，才在实际参数记号字符串中识别宏名（从而造成实际参数记号字符串在扩展中出现一次或几次）。

例　对于定义

```
#define plus(x,y)   add(y,x)
#define add(x,y)    ((x)+(y))
```

调用

```
plus(plus(a,b),c)
```

50 扩展如下：

步骤	结　果
1.（初始）	plus(plus(a,b),c)
2.	add(c,plus(a,b))
3.	((c)+(plus(a,b)))
4.	((c)+(add(b,a)))
5.（最后）	((c)+(((b)+(a))))

□

标准C语言不重新扩展自己的扩展中出现的宏，无论是直接包含在其中还是通过某种中间嵌套宏扩展序列包含的。这样就使编程人员可以根据原有的定义重新定义函数。较早的C语言预处理器传统上不能发现这种递归，会继续扩展宏，直到某个系统错误使其停止。

例　下列宏改变平方根函数的定义，以不同于正常方式的方式处理负数参数：

```
#define sqrt(x)   ((x)<0 ? sqrt(-(x)) : sqrt(x))
```

尽管它多次对参数求值，但这个宏在标准C语言中能够按预期目的工作，而在较早编译器中则会出现错误。同样问题会出现在下面的宏定义中。

```
#define char unsigned char
```

□

关于用宏跟踪函数调用的有趣例子参见7.4.3节。

3.3.4 预定义宏

标准C语言预处理器要求定义某些对象式宏（见表3-2）。每个预定义宏的名称以两个下划线字符开头和结尾，这些预定义宏不能被取消定义(**#undef**)或由编程人员重新定义。

__LINE__与**__FILE__**宏可以打印某种错误消息；**__DATE__**与**__TIME__**宏可以记录何时发生编译。**__DATE__**与**__TIME__**的值在整个编译过程中保持常量。**__LINE__**与**__FILE__**宏的值由实现建立，但可以用**#line**指令改变（见3.6节）。C99预定义标识符**__func__**（见2.6.1节）的作用与**__LINE__**相似，但实际上是个块作用域变量，而不是宏，它提供所在函数名。

__STDC__与**__STDC_VERSION__**宏可以编写与标准C语言和非标准C实现兼容的代码。

51 **__STDC_HOSTED__**宏是C99中引入的，用于区别宿主实现与独立实现。其余C99宏表示实现的浮点数和宽字符工具是否遵循其他相关国际标准（建议遵循，但不是必须的）。

表3-2 预定义宏

宏	值
__LINE__[①]	当前源程序行的行号，表示为十进制整型常量
__FILE__[①]	当前源文件名，表示为字符串型常量
__DATE__	转换的日历日期，表示为"Mmm dd yyyy"形式的字符串型常量，Mmm是由asctime产生的
__TIME__	转换的时间，表示为"hh:mm:ss"形式的字符串型常量，是由asctime产生的
__STDC__	编译器为ISO兼容实现时为十进制整型常量1
__STDC_VERSION__	如果实现符合C89增补1，则这个宏的值为199409L；如果实现符合C99，则这个宏的值为199901L；否则数值是未定义
__STDC_HOSTED__	（C99）实现为宿主实现时为1，实现为独立实现时为0
__STDC_IEC_559__	（C99）浮点数实现符合IEC 60559标准时定义为1，否则数值是未定义
__STDC_IEC_559_COMPLEX__	（C99）复数运算实现符合IEC 60559标准时定义为1，否则数值是未定义
__STDC_ISO_10646__	（C99）定义为长整型常量，yyyymmL表示wchar_t值符合ISO 10646标准及其指定年月的修订补充，否则数值未定义

① 这些宏在非ISO实现中也很常用。

实现还经常定义其他宏用于传递环境信息，如进行程序编译工作的计算机类型。具体定义哪些宏值是由实现决定的，但UNIX实现习惯上预定义unix。与内置宏不同的是，这些宏可以取消定义。标准C语言要求特定实现的宏名以下划线开头，加上大写字母或另一个下划线（unix宏不符合这个要求）。

例 预定义宏可以在某种错误消息中使用：

```
if (n != m)
   fprintf(stderr,"Internal error: line %d, file %s\n",
      __LINE__, __FILE__ );
```

其他实现定义的宏可以分隔主机或特定目标代码。例如，Microsoft Visual C++定义__WIN32为1： 52

```
#ifdef __WIN32
   /* Code for Win32 environment */
#endif
```

__STDC__与__STDC_VERSION__宏可以编写与标准C语言和非标准C实现兼容的程序：

```
#ifdef __STDC__
  /* Some version of Standard C */
#if defined(__STDC_VERSION__) && __STDC_VERSION__>=199901L
   /* C99 */
#elif defined(__STDC_VERSION__) && __STDC_VERSION__>=199409L
   /* C89 and Amendment 1 */
#else
   /* C89 but not Amendment 1 */
#endif
#else /* __STDC__ not defined */
  /* Not Standard C */
#endif
```

□

参考章节 **asctime**函数 20.3；复数运算 第23章；**fprintf** 15.11；独立实现与宿主实现 1.4；**#ifdef**预处理器命令 3.5.3；**#if**预处理器命令 3.5.1；取消宏定义 3.3.5；**wchar_t** 24.1

3.3.5 取消宏定义与重新定义宏

#undef命令可以取消宏定义：

#undef *name*

这个命令使预处理器忘记由*name*定义的任何宏，对当前没有定义的名称，取消定义时不会出错。一个名称被取消定义之后，可以对它进行全新的定义（用**#define**）。**#undef**命令中不进行宏替换。

标准C语言和许多C语言的其他实现中都允许宏的良性重新定义。良性重新定义就是新定义的每一个记号都与现有定义的记号相同。重新定义时要在与原定义相同的地方包括空白符，但特定空白符字符可以不同。编程人员应避免利用良性重新定义。一种好的编程风格是在一个地方定义所有程序实体，包括宏（一些较早的C语言实现不允许任何一种重新定义）。

例 下列定义中对**NULL**的重新定义方式是可以接受的，但对于**FUNC**的两个重新定义都是无效的（第一个重新定义包括了原定义中没有的空白符，第二个重新定义改变了原定义中的两个记号）。

53

```
# define NULL 0
# define FUNC(x) x+4
# define NULL /* null pointer */ 0
# define FUNC(x) x + 4
# define FUNC(y) y+4
```

□

例 编程人员因为合理的原因不知道某个宏在前面是否有定义时，可以用**#ifndef**命令测试前面是否有定义以避免重新定义：

```
#ifndef MAXTABLESIZE
#define MAXTABLESIZE 1000
#endif
```

这个方法对调用C语言编译器的命令中允许宏定义的实现特别有用。例如，下列C语言的UNIX调用提供**MAXTABLESIZE**宏的初始定义为**5000**。然后C语言编程人员可以通过以下命令对定义进行检查：

```
cc -c -DMAXTABLESIZE=5000 prog.c
```

□

尽管标准C语言中不允许，但一些较早的预处理器实现仍然坚持处理**#define**与**#undef**以维护定义堆栈。用**#define**重新定义一个名称时，旧的定义压入堆栈中，然后新定义替换旧定义。用**#undef**取消定义时，放弃当前定义，恢复到最近的定义（如有）。

参考章节 **#define**命令 3.3；**#ifdef**与**#ifndef**命令 3.5.3

3.3.6 宏扩展中的优先级错误

宏完全通过记号文本替换进行操作。只有在宏扩展过程之后才把宏体分析成声明、表达式或语句，这样就可能因为不小心而产生奇怪的结果。作为规则，最安全的方法是总是把宏体中出现的每个参数放在括号中。整个宏体如果语法只是个表达式，也应放在括号中。

例 考虑下列宏定义：

```
#define SQUARE(x) x*x
```

SQUARE取一个参数表达式并产生计算这个参数平方的新表达式。例如，**SQUARE(5)**扩展成**5*5**。但是，表达式**SQUARE(z+1)**扩展成**z+1*z+1**，分析成**z+(1*z)+1**而不是我们所要的**(z+1)*(z+1)**。要避免这个问题，可以将**SQUARE**定义如下：

```
#define SQUARE(x) ((x)*(x))
```

54

外层括号是必要的，以防止对**(short)SQUARE(z+1)**等形式的表达式进行错误解释。 □

参考章节 转换表达式 7.5.1；表达式优先级 7.2.1

3.3.7 宏参数的副作用

宏还可能因为副作用造成问题。由于宏的实际参数可能是文本复制，因此可能执行多次，实际参数中的副作用也就可能发生多次。相反，真正的函数调用则只求值参数表达式一次（这是宏调用要模拟的），因此表达式的任何副作用只发生一次。使用宏时，要小心避免这类问题。

例 对上例中的**SQUARE**宏，还有一个函数**square**，完成与之基本相同的工作：

```
int square(int x) { return x*x; }
```

SQUARE宏可以求整数或浮点数的平方，而**square**函数只能求整数平方。运行时调用函数要比使用宏慢一些。但这些差别比起副作用问题就算小问题了。在下列程序段中：

```
a = 3;
b = square(a++);
```

变量**b**取得数值9，变量**a**最终为数值4。而在下列程序段中：

```
a = 3;
b = SQUARE(a++);
```

变量**b**取得数值12，变量**a**最终为数值5。因为第二段程序展开如下：

```
a = 3;
b = ((a++)*(a++));
```

此处的12和5只是一种可能的取值，因为标准C语言实现可能用不同方法求值表达式**((a++)*(a++))**，见7.12节。 □

参考章节 自增运算符++ 7.4.4

3.3.8 将记号转换为字符串

标准C语言中有一种机制可以将宏参数（扩展之后）转换为字符串型常量。在此之前，编程人员要利用许多C语言预处理器中的漏洞以不同方式达到相同结果。

在标准C语言中，宏定义中出现的**#**记号被当作一元“字符串化”运算符，后面为宏正式参数名。宏扩展期间，**#**和正式参数名换成相应的包含在字符串引号当中的实际参数。生成字符串时，记号参数表中的每个空白序列换成一个空格符，任何嵌入引号和反斜杠前面加上一个反斜杠以保留其在字符串中的含义。参数开头和末尾的空白符忽略，因此空参数扩展为空字符串""（即使逗号之间有空白符）。

55

例 考虑**TEST**宏的标准C语言定义：

```
#define TEST(a,b) printf( #a "<" #b "=%d\n", (a)<(b) )
```

语句**TEST(0,0xFFFF);TEST('\n',10);**展开如下：

```
printf("0" "<" "0xFFFF" "=%d\n", (0)<(0xFFFF) );
```

```
printf("'\\n'" "<" "10" "=%d\n", ('\n')<(10) );
```

拼接相邻字符串之后，变成：

```
printf("0<0xFFFF=%d\n", (0)<(0xFFFF) );
printf("'\\n'<10=%d\n", ('\n')<(10) );
```

□

许多非标准C语言编译器替换字符串型与字符型常量中的宏正式参数，但标准C语言禁止这一操作。

例　在这些非标准C语言实现中，**TEST**宏可能写成如下：

```
#define TEST(a,b) printf( "a<b=%d\n", (a)<(b) )
```

扩展**TEST(0,0xFFFF)**可以模拟字符串化的结果：

```
printf("0<0xFFFF=%d\n", (0)<(0xFFFF) );
```

但是，扩展**TEST('\n',10)**几乎肯定会丢失附加的反斜杠，使**printf**函数的输出由于无法预料的行终结符而出现错误：

```
printf("'\n'<10=%d\n", ('\n')<(10) );
```

□

非ISO实现中空白符的处理也随不同编译器而不同，为此，只有在标准C语言实现中才能使用这个特性。

3.3.9 宏扩展中的记号合并

标准C语言中合并记号形成新记号时，由宏定义中的合并运算符##控制。重新扫描更多宏之前，宏替换表中任何运算符##中间的两个记号合并成一个记号。因此必须有这种记号：##不能
56 放在替换表开头或末尾。如果合并之后得不到有效记号，则结果是未定义的。

例

```
#define TEMP(i)  temp ## i
TEMP(1) = TEMP(2 + k) + x;
```

预处理之后变成：

```
temp1 = temp2 + k + x;
```

□

上例中，扩展**TEMP()+x**时可能出现一个奇妙的情形。宏定义是有效的，但##右边没有要组合的记号（除非遇到+，但这不是我们所要的）。要解决这个问题，应把正式参数**i**看成是专为##扩展的特殊的“空”记号。这样，**TEMP()+x** 扩展的结果将如我们预料的那样为**temp+x**。

不能用记号拼接产生通用字符名。

和宏参数转换成字符串时一样（见3.3.8节），编程人员可以利用许多非标准C语言实现中的漏洞获得这种合并功能。尽管C语言的原始定义明确地把宏体描述成记号序列，而不是字符序列，但许多C语言编译器把宏体当作字符序列一样扩展和重新扫描。这在编译器处理注释语句时更加明显，编译器将注释语句完全删除而不是换成空格，因此一些巧妙编写的程序就可以利用这个特性。

例　考虑下列例子：

```
#define INC ++
#define TAB internal_table
#define INCTAB table_of_increments
```

```
#define CONC(x,y) x/**/y
CONC(INC,TAB)
```

标准C语言将**CONC**体解释为两个记号**x**和**y**，用空格分开（注释语句变成空格）。调用**CONC(INC,TAB)**扩展为两个记号**INC TAB**。但有些非标准C语言实现直接删除注释语句，然后重新扫描宏体中的记号，这样就把**CONC(INC,TAB)**扩展为一个记号**INCTAB**：

步骤	标准C语言扩展	可能的非标准扩展
1	CONC(INC,TAB)	CONC(INC,TAB)
2	INC/**/TAB	INC/* */TAB
3	INC TAB	INCTAB
4	++ internal_table	table_of_increments

□

参考章节 自增运算符++ 7.5.8；通用字符名 2.9

57

3.3.10 宏中的可变参数表

在C99中，函数式宏的最后一个参数或惟一正式参数可以是省略号，表示宏接受可变数目的参数表：

#define *name*(*identifer-list*, ...) *sequence-of-tokens*$_{opt}$
#define *name*(...) *sequence-of-tokens*$_{opt}$

调用这种宏时，实际参数至少要和*identifier-list*中的标识符数一样多。尾部参数（包括任何分隔逗号）合并成一个预处理记号序列，称为可变参数。宏定义替换表中出现的标识符**__VA_ARGS__**被看作是合并的可变参数的宏参数，即用多余参数表替换**__VA_ARGS__**，包括逗号分隔符。**__VA_ARGS__**只能出现在参数表中包括省略号的宏定义中。

具有可变参数的宏常用于与带可变数目参数的函数接口，如**printf**。利用字符串化运算符#，也可以将参数表变成一个字符串，而不必把参数放在括号中。

例 下列指令生成**my_printf**宏，可以将参数写入出错信息或标准输出：

```
#ifdef DEBUG
#define my_printf(...) fprintf(stderr, __VA_ARGS__)
#else
#define my_printf(...) printf(__VA_ARGS__)
#endif
```

可以通过以下方法使用这个宏：

```
my_printf("x = %d\n", x);
```

□

例 对下列定义：

```
#define make_em_a_string(...) #__VA_ARGS__
```

下列调用：

```
make_em_a_string(a, b, c, d)
```

展开成字符串：

```
"a, b, c, d"
```

□

58

3.3.11 其他问题

一些非标准C语言实现不对宏定义和调用进行严格的错误检查，包括允许宏调用之后的文本

完成宏体中不完整的记号。某些实现缺乏错误检查并不能使对这种缺陷的巧妙利用变得合法。标准C语言再三强调宏体应为形式合理的记号序列。

例　下列代码段来自一种非标准C语言实现：

```
#define FIRSTPART "This is a split
...
printf(FIRSTPART string."); /* Yuk! */
```

预处理之后得到源文本如下：

```
printf("This is a split string.");
```

□

3.4 文件包含

#include预处理器命令可以处理指定源文本文件的全部内容，就像这些内容放在**#include**命令的位置一样。**#include**命令在标准C语言中有下列3种形式：

```
# include < h-char-sequence >
# include " q-char-sequence "
# include   preprocessor-tokens
```

(标准C语言)

h-char-sequence(*h*字符序列)

除>和行末符以外的任何字符序列

q-char-sequence(*q*字符序列)

除"和行末符以外的任何字符序列

preprocessor-tokens（预处理器记号）

任何不以<或"开头的C语言记号序列或不能解释为记号的非空白字符

在**#include**的前两种形式中，分隔符之间的字符应为某种实现定义格式的文件名。在>或"之后只能是空白符。**#include**的前两种形式是所有C语言编译器都支持的，文件名可以使用标准C语言中的三字符组替换和源行续行，但不能进行其他字符处理。

在**#include**的第三种形式中，预处理器记号要进行正常的宏扩展，结果应匹配前两种形式之一（包括引号或尖括号）。这种**#include**形式比较少见，非标准编译器中可能没有实现或以不同方式实现。

59

例　下面是使用**#include**的第三种形式的一种方法：

```
#if some_thing==this_thing
#   define IncludeFile "thisname.h"
#else
#   define Includefile <thatname.h>
#endif
...
#include Includefile
```

这个样式可以用于本地化定制，但编程人员如果希望与早期编译器兼容，则要把**#include**命令放在上面介绍的方法的**#define**命令位置：

```
#if some_thing==this_thing
#   include "thisname.h"
```

```
#else
#   include <thatname.h>
#endif
```
□

文件名语法是与实现相关的，但标准C语言要求所有实现允许**#include**中的文件名包括字母与数字（以字母开头），加一个点号和一个字母。C99允许点号之前最多8个字母与数字，但C89只允许点号之前最多5个字母与数字。这种形式的文件名应映射一个实现定义文件。

引号分隔的文件与尖括号分隔的文件差别在于C语言实现用不同方法定位。这两种形式都在一组（可能不同的）实现定义的位置中寻找文件。通常，下列形式：

#include <*filename*>

根据实现定义搜索规则在某个标准位置寻找文件。这些标准位置通常包含实现自己的头文件，如**stdio.h**。而下列形式：

#include "*filename*"

也搜索标准位置，但通常先搜索一些本地位置，如编程人员的当前目录。通常，实现在C语言之外有一些标准方式用于指定一组搜索这些文件的位置。一般意义上说，"…"形式引用编程人员编写的头文件，而<…>形式引用标准实现文件。

60

事实上，标准C语言把**stdio.h**之类的标准头文件当作特例处理。标准C语言要求实现能够识别出<>分隔格式的**#include**命令中出现的标准库头文件名，但不要求这些名称指定真正的文件名。这些标准库头文件名可以作为特例处理，我们假设其内容是C语言实现所知道的。为此，标准将其称为*标准头*而不是*标准头文件*。本书两种称法都用。

包括文件可以包含**#include**命令，这种**#include**嵌套允许的深度随实现而定，但标准C语言要求支持至少8级（C99中要求支持15级）。包括文件的位置可能影响嵌套文件的搜索规则。

例 假设编译文件系统目录**/near**中的C语言程序**first.c**。

first.c文件包含下列行，指定**second.h**在目录**/far**中：

```
// In /near/first.c
#include "/far/second.h"
```

头文件**second.h**包含下列行，没有指定目录：

```
// In /far/second.h
#include "third.h"
```

实现是选择原工作目录中的**/near/third.h**文件，还是选择包括文件所在目录中的**/far/third.h**文件呢？有些UNIX C编译器会寻找**/far/third.h**文件。原始的C语言描述要求找到**/near/third.h**文件。大多数实现允许编程人员为那些没有指定目录的包括文件设定一个搜索目录顺序列表。
□

参考章节 字符串型常量 2.7.4；三字符组 2.1.4

3.5 条件编译

预处理器条件命令允许预处理器根据计算条件处理或删除源文本行。

3.5.1 #if、#else与#endif命令

#if、**#else**与**#endif**预处理器条件命令可以根据条件在编译时包括或排除源文本行，其

使用方法如下：

```
#if constant-expression
    group-of-lines-1
#else
    group-of-lines-2
#endif
```

61

*constant-expression*准备进行宏替换，要求值为常量算术值。第7.11.1节将介绍表达式限制。一个行组中可能包含多行任意类型的文本，甚至包括别的预处理器命令行或不包括任何行。**#else**命令及后面的行组可以省略，这相当于包括了其后紧跟空行组的**#else**命令。每个行组还可以包括一组或多组**#if-#else-#endif**命令。

如前所示的一组命令进行处理时，一个行组进行编译，而放弃另一行组。首先，求值**#if**命令中的常量表达式，如果其值不是0，则*group-of-lines-1*进行编译，*group-of-lines-2*放弃（如有）。否则*group-of-lines-1*放弃，如果有**#else**命令，则*group-of-lines-2*进行编译，但如果没有**#else**命令，则没有行要进行编译。3.5.4和7.11节详细介绍**#if**命令中可以使用的常量表达式。

放弃的行组不在预处理器中处理，不进行宏替换，并且忽略预处理器命令。一个例外是，在放弃行组中，要识别**#if**、**#ifdef**、**#ifndef**、**#elif**、**#else**与**#endif**命令以用于计数，这是为了维护条件编译命令的正确嵌套。这一识别过程还表明要扫描放弃行，并且将其分解成记号，字符串型常量和注释语句也要能被识别并被恰当分隔。

如果**#if**或**#elif**的常量表达式中出现未定义宏名，则换成整数常量0，即只要定义的宏名具有常量数学非0值命令“**#ifdef** *name*”与“**#if** *name*”就具有相同效果。我们认为这里用**#ifdef**或分隔运算符更明确，但标准C语言也支持**#if**的这个用法。

参考章节 **defined** 3.5.5；**#elif** 3.5.2；**#ifdef** 3.5.3

3.5.2 #elif命令

#elif命令在标准C语言和较新的ISO前编译器中都被使用，非常方便，可以简化一些预处理器条件，具体用法如下：

```
#if constant-expression-1                    (或#ifdef或#ifndef)
    group-of-lines-1
#elif constant-expression-2
    group-of-lines-2
...
#elif constant-expression-n
    group-of-lines-n
#else
    last-group-of-lines
#endif
```

62

处理这个命令序列时，至多有一个行组进行编译，所有其他行组被放弃。首先，求值**#if**命令中的*constant-expression-1*，如果其值不是0，则*group-of-lines-1*进行编译，到相应**#endif**为止的所有其他行组放弃；如果其值为0，则求值第一个**#elif**命令中的*constant-expression-2*，如果其值不是0，则*group-of-lines-2*进行编译。一般情况下，按顺序求值*constant-expression-i*，直到有一个常量表达式产生非0值，然后预处理器传递非0常量表达式所在命令后面的行组进行编译，忽略命令集中的任何其他常量表达式，放弃所有其他行组。如果所有常量表达式均为0，又

有**#else**命令，则预处理器传递**#else**命令后面的行组进行编译，但如果没有**#else**命令，则不传递任何行组进行编码。**#elif**命令中使用的常量表达式与**#if**命令中可以使用的常量表达式相同，见3.5.4和7.11节详细介绍。

在放弃行组中，要识别**#if**、**#ifdef**、**#ifndef**、**#elif**、**#else**与**#endif**命令以用于计数，这是为了维护条件编译命令的正确嵌套。

宏替换在**#elif**命令后面的命令行中进行，因此可以在常量表达式中使用宏调用。

例 尽管**#elif**命令在适当时候很方便，但可以只用**#if**、**#else**与**#endif**命令实现它的功能，下面举一个例子。

使用#elif	不用#elif
#if *constant-expression-1* 　*group-of-lines-1* **#elif** *constant-expression-2* 　*group-of-lines-2* **#else** 　*last-group-of-lines* **#endif**	**#if** *constant-expression-1* 　*group-of-lines-1* **#else** **#if** *constant-expression-2* 　*group-of-lines-2* **#else** 　*last-group-of-lines* **#endif** **#endif**

□

3.5.3 #ifdef与#ifndef命令

#ifdef与**#ifndef**命令可以测试一个名称是否定义为预处理器宏。命令

```
#ifdef name
```

如果定义了*name*（即使宏体是空的），则等价于命令

63

```
#if 1
```

如果没有定义*name*或用**#undef**命令取消了*name*的定义，则其等价于命令

```
#if 0
```

#ifndef命令的含义与**#ifdef**的含义正好相反，在没有定义*name*时为真，在定义了*name*时为假。

注意 **#ifdef**与**#ifndef**只测试名称是否用**#define**定义过（或用**#undef**取消定义），而不管在C语言程序文本的声明中出现的宏名是否被编译过（有些C语言实现允许用特殊编译器命令行参数定义宏名）。

例 C语言程序中有几种使用**#ifdef**与**#ifndef**命令的方法。首先，可以实现预处理器时间枚举类型，一组符号中只定义一个。例如，假设我们要用一组名称VAX、PDP11与CRAY2表示运行程序的计算机，则可以定义所有这些名称，将其中一个定义为1，其余定义为0：

```
#define VAX 0
#define PDP11 0
#define CRAY2 1
```

然后可以选择编译与机器相关源代码如下：

```
#if VAX
    VAX-dependent code
#endif
```

```
#if PDP11
    PDP11-dependent code
#endif
#if CRAY2
    CRAY2-dependent code
#endif
```

但习惯方法只定义一个符号：

```
#define CRAY2 1
    /* None of the other symbols is defined. */
```

然后用条件命令测试是否定义了每个符号：

```
#ifdef VAX
    VAX-dependent code
#endif
#ifdef PDP11
    PDP11-dependent code
#endif
#ifdef CRAY2
    CRAY2-dependent code
#endif
```

□

例　**#ifdef**与**#ifndef**命令的另一个用法是提供宏的默认定义。例如，只有当一个名称没有其他定义的时候，库文件可能会为它提供一个定义：

```
#ifndef TABLE_SIZE
#define TABLE_SIZE 100
#endif
...
static int internal_table[TABLE_SIZE];
```

程序可能只是包括下列文件：

```
#include <table.h>
```

□

这时**TABLE_SIZE**的定义为**100**，都在库文件中和**#include**之后，程序也可能先提供一个显式定义：

```
#define TABLE_SIZE 500
#include <table.h>
```

这时**TABLE_SIZE**的定义为**500**。

□

一个常见的编程错误是用**#if name**而不是**#ifdef name**或**#if defined(name)**测试名称是否已定义。这个错误形式通常能够工作，因为预处理器会把**#if**表达式中没有定义的任何名称替换成具有常量0的宏。因此，如果**name**没有定义，则3种形式是等价的。但如果**name**定义为数值0，则即使定义了名称，**#if name**的值仍为false。同样，如果**name**定义的值不是有效表达式，则**#if name**会产生错误。

参考章节　**#define**　3.3；**defined**运算符　3.5.5；**#include**　3.4；预处理器词法规则　3.2；**#undef**　3.3

3.5.4　条件命令中的常量表达式

7.11.1节描述了**#if**与**#elif**命令中可以使用的表达式，包括整型常量和所有整数算术运算

符、关系运算符、位运算符和逻辑运算符。

C99强制用目标计算机上最大的整数类型（在**stdint.h**中定义的**intmax_t**或**uintmax_t**）进行所有预处理器运算。过去，标准C语言不要求翻译程序具有目标计算机的算术属性。

参考章节 **intmax_t** 21.5; **uintmax_t** 21.5

65

3.5.5 defined运算符

defined运算符可以在**#if**与**#elif**表达式中使用，但不能在其他地方使用。下列各种形式的表达式在预处理器中定义了*name*时求值为1，否则求值为0：

```
defined name
defined( name )
```

例 **defined**命令使编程人员可以编写

```
#if defined(VAX)
```

而不是

```
#ifdef VAX
```

defined运算符用起来可能更方便，因为它可以建立复杂表达式如下：

```
#if defined(VAX) && !defined(UNIX) && debugging
...
```

□

3.6 显式的行编号

#line预处理器命令告诉C语言编译器，源程序是由另一工具产生的，并且表示了源程序中位置与产生C语言源程序的原用户编写的文件的行位置的对应关系。**#line**命令有两种形式，下列形式表示下面的源行是从原用户编写的文件*filename*的第*n*行派生的，*n*应为十进制数序号：

```
# line n " filename "
```

下列形式表示下面的行是从上次**#line**命令中提到的原用户编写文件的第*n*行派生的：

```
# line n
```

最后，如果**#line**命令不符合上述两种形式，则解释如下：

```
# line preprocessor-tokens
```

宏替换对参数记号序列进行，结果要符合上述两种**#line**命令形式之一。

66

#line命令提供的信息用于设置预定义宏**__LINE__**与**__FILE__**的值，否则其行为是未说明的，编译器可以将其忽略。通常，这个信息也用于诊断消息。一些产生C语言源文本的工具用**#line**命令使错误消息可以和这个工具的输入文件相联系，而不是和实际C语言源文件相联系。

一些C语言实现允许预处理器独立于编译器其他部分而使用，而有时预处理器是另一个独立程序，产生中间文件，然后由实际编译器处理。在这种情况下，预处理器可能在中间文件中产生新的**#line**命令，然后编译器要识别这些事件，但不识别任何其他预处理器命令。预处理器是否产生**#line**命令取决于实现。同样，预处理器是否传递、修改和删除输入中的**#line**命令也取决于实现。

较早的C语言版本允许用#作为**#line**命令的同义词，采用下列形式：

```
# n filename
```

这个语法已经过时，在标准C语言中不允许，但许多实现仍然支持以保证其兼容性。

参考章节　**__FILE__**　3.3.4；**__LINE__**　3.3.4

3.7 杂注指令

#pragma命令是标准C语言新增的。这个命令名后面可以放任何记号序列：

 # pragma *preprocessor-tokens*

C实现可以用**#pragma**指令增加新的预处理器功能或向编译器提供实现定义信息。**#pragma**命令后面的信息没有任何限制，实现应忽略不理解的信息。**#pragma**命令的参数要进行宏扩展。

显然，两个不同实现可能对同一信息进行不一致的解释，因此最好根据使用的编译器有条件地使用**#pragma**命令。

例　下列代码检查发出**#pragma**命令之前正在使用的适当的编译器(**tcc**)、计算机和符合标准的实现：

67

```
#if defined(_TCC) && defined(__STDC__) && defined(vax)
#pragma builtin(abs),inline(myfunc)
#endif
```

□

参考章节　**defined**　3.5.3；内存模型　6.1.5；**#if**　3.5.1

3.7.1 标准杂注

在C99中，有些杂注引入特定含义。为了加以区别，所有标准杂注前面都要加上记号**STDC**（在所有由实现定义的宏扩展之前，）并且其后的记号不能是扩展的宏，即下列指令是实现定义杂注：

```
#pragma FENV_ACCESS ON
```

而下列指令指定C99**FENV_ACCESS**杂注：

```
#pragma STDC FENV_ACCESS ON
```

如果标准杂注名前面没有加上记号，则实现会发出警告，因为这是常见的错误。

C99定义的几个标准杂注是**FP_CONTRACT**、**FENV_ACCESS**和**CX_LIMITED_RANGE**。它们取一个参数作为开关：

on-off-switch:
　　ON
　　OFF
　　DEFAULT

参数**DEFAULT**将杂注设置为初始默认值(on或off)，每个标准杂注都指定一个默认值（有时指定为实现定义）。

参考章节　**CX_LIMITED_RANGE**　23.2；**FENV_ACCESS**　22.2；**FP_CONTRACT**　22.5

3.7.2 标准杂注的位置

标准杂注要遵循一定的位置规则，使杂注更容易处理并允许杂注嵌套。标准杂注可能出现在两个地方：翻译单元顶层任何外部声明之前或复合语句开头所有显式声明和语句之前。

放在顶层时，杂注一直保持有效，直到翻译单元结束或遇到同一杂注的另一个实例。第二

个杂注可能是顶层的另一个杂注，取代第一个杂注，也可能是复合语句中的杂注。

放在复合语句开头时，这个杂注保持有效，直到复合语句的词法结束或复合语句中遇到同一杂注的另一个实例。第二个杂注可能在同一复合语句开头，取代第一个杂注，也可能是内层 [68] 复合语句中的杂注。包含标准杂注的复合语句结束时，杂注恢复遇到复合语句之前的状态，即标准杂注按正常变量作用域规则嵌套，只是可以在同一作用域层中指定多次。

参考章节 作用域 4.2.1

3.7.3 _Pragma运算符

C99增加了**_Pragma**运算符，使杂注功能更加灵活。扩展宏之后，下列形式的运算符表达式：

_Pragma("*string-literal*")

就像字符串字面值内容（删除外层引号，将\"变成"，将\\变成\之后）是**#pragma**指令中出现的预处理器记号一样。例如，下列表达式：

```
_Pragma("STDC FENV_ACCESS ON")
```

就像在该位置增加下列杂注一样：

```
#pragma STDC FENV_ACCESS ON
```

#pragma可以单独放在一行，其预处理器记号不随宏扩展，而**_Pragma**可以放在其他表达式中间，可以通过宏扩展产生。

3.8 错误指令

#error指令是标准C语言新增加的。命令名后面可以加上任何记号序列：

 # error *preprocessor-tokens*

#error指令产生编译错误消息，包括要进行宏扩展的参数记号。

例 **#error**指令可以用于探测编程人员的不一致性和预处理期间违反限制的情形。例如：

```
#if defined(A_THING) && defined(NOT_A_THING)
#error Inconsistent things!
#endif
#include "sizes.h"   /* defines SIZE */
…
#if (SIZE % 256) != 0
#error "SIZE must be a multiple of 256!"
#endif
```

[69]

在第一个**#error**例子中，我们不用字符串常量。第二个例子中，我们使用字符串常量，因为输出消息中不希望扩展记号**SIZE**。 □

参考章节 **defined** 3.5.3; **#if** 3.5.1

3.9 C++兼容性

C++使用C89预处理器，因此C语言与C++之间差别不大。

预定义宏

宏**__cplusplus**是C++实现中预定义的，可以在C和C++环境中的源文件中使用。这个名称不符合预定义宏的标准C语言拼写规则，但与现有C++实现兼容。在标准C语言中，它的值为

199711L之类的版本号。

__STDC__是否在C++环境中定义取决于实现。标准C语言与C++有一定差别，因此不知道是否要定义**__STDC__**。

表3-2列出的C99中独有的宏在C++中没有。

例　为了与传统C语言、标准C语言和C++兼容，要以下列方式测试环境：

```
#ifdef __cplusplus
   /* It's a C++ compilation */
#else
#ifdef __STDC__
   /* It's a Standard C compilation */
#else
   /* It's a non-Standard C compilation */
#endif
#endif
```

70 如果知道您的C语言实现符合标准C语言，则可以简化成

```
#if defined(__cplusplus)
   /* It's a C++ compilation */
#else
   /* It's a Standard C compilation */
#endif
```

□

参考章节　**__STDC__**　3.3.4；**__STDC_VERSION__**　3.3.4

3.10 练习

1. 下列哪些标准C语言宏定义可能是错误的？为什么？哪些定义可能在传统C语言中出错？

(a) `#define ident (x) x`　　(c) `#define PLUS +`

(b) `#define FIVE = 5;`　　(d) `#define void int`

2. 下面是一些宏定义和调用。传统C语言和标准C语言如何扩展每个宏调用？

	定义	调用
(a)	`#define sum(a,b) a+b`	`sum(b,a)`
(b)	`#define paste(x,y) x/**y`	`paste(x,4)`
(c)	`#define str(x) # x`	`str(a book)`
(d)	`#define free(x)x ? free(x) : NULL`	`free(p)`

3. 下面是两个头文件和一个C语言程序文件。如果对程序文件采用C语言预处理器处理，会产生什么结果？

```
/* File blue.h */
int blue = 0;
#include "red.h"
```

```
/* File red.h */
#ifndef __red__
#define __red__
#include "blue.h"
int red = 0;
#endif
```

```
/* File test.c */
#include "blue.h"
#include "red.h"
```

4. 一个朋友提供下列宏定义，要把数字参数加倍。这个宏错在哪里？改写宏，使其能够正确操作。

```
#define DBL(a)    a+a
```

5. 下列标准C语言程序段中，哪个是**M(M)(A,B)**的扩展？

```
#define M(x)      M ## x
#define MM(M,y)   M = # y
M(M)(A,B)
```

6. 编写一个标准预处理器指令序列，使标准C语言程序在宏**SIZE**没有定义时或虽然定义而数值不在1~10之间时编译失败。

7. 举一个字符序列的例子，使其构成预处理器的一个记号，但不是C编译器的记号。

8. 下列程序段错在哪里？

```
if (x != 0)
   y = z/x;
else
   # error "Attempt to divide by zero, line " __LINE__
```

第4章 声 明

C语言中声明一个名称就是把一个标识符与某个C语言对象相关联，如变量、函数或类型。C语言中可以声明的名称包括：

- 变量
- 函数
- 类型
- 类型标志
- 结构成员与联合成员
- 枚举常量
- 语句标号
- 预处理器宏

除了语句标号和预处理器宏之外，所有标识符都在C语言声明中声明。变量、函数、类型放在声明的声明符中，类型标志、结构成员与联合成员和枚举常量在声明的某种类型说明符中声明。语句标号在C语言函数中出现时声明，而预处理器宏用**#define**预处理器命令声明。

C语言中的声明很难描述，原因有几个。第一，它们涉及一些异常语法，初学者不容易理解，例如，声明

```
int (*f)(void);
```

声明了一个函数的指针，这个函数不取参数，返回一个整数值。

第二，声明的许多抽象属性在C语言中比其他编程语言中更复杂，如作用域与生存期。介绍实际声明语法之前，我们要先在4.2节中介绍这些属性。

最后，C语言声明的有些方面要先了解C语言类型系统之后才能理解，C语言类型系统放在第5章介绍。类型标志、结构成员与联合成员和枚举常量也要在第5章介绍，但为了完整起见，本章会介绍这些声明的一些属性。

参考章节 枚举类型 5.5；**#define**预处理器命令 3.3；语句标号 8.3；结构类型 5.6；类型说明符 4.4；联合类型 5.7

4.1 声明组织

声明可以在C语言程序的多个地方出现，可能影响声明的属性。C语言源文件或翻译单元包括函数、变量等项目的顶层声明序列。每个函数有参数声明和体，而体中又可能包含各种块，包括复合语句。块可能包含内部声明序列。

下面是声明的基本语法。函数定义见第9章介绍。

declaration（声明）：
    declaration-specifiers initialized-declarator-list ;

declaration-specifiers（声明说明符）：

storage-class-specifier declaration-specifiers$_{opt}$
type-specifier declaration-specifiers$_{opt}$
type-qualifier declaration-specifiers$_{opt}$
function-specifier declaration-specifiers$_{opt}$ *(C99)*

initialized-declarator-list（初始化声明符列表）：
initialized-declarator
initialized-declarator-list , *initialized-declarator*

initialized-declarator（初始化声明符）：
declarator
declarator = *initializer*

声明说明符中至多可以出现一个存储类说明符和一个类型说明符，但一个类型说明符可能包括多个记号（如**unsigned long int**）。在C99中，类型说明符是必需的。每个类型限定符可以在声明说明符中至多出现一次。C99函数说明符**inline**只能在函数声明中出现一次。在这些限制中，类型说明符、存储类说明符、函数说明符和类型限定符可以在声明说明符中按任何顺序出现。

例 习惯上先写存储类说明符，然后是类型限定符，最后是类型说明符。下列声明中，**i**和**j**具有相同类型和存储类，但**i**的声明方式更好。 74

```
unsigned volatile long extern int const j;
extern const volatile unsigned long int i;
```

参考章节 声明符 4.5；表达式 第7章；函数定义 第9章；初值 4.6；语句 第8章；存储类说明符 4.3；类型说明符与类型限定符 4.4

4.2 术语

本节介绍几个描述声明时使用的术语。

4.2.1 作用域

声明作用域就是声明有效的C语言程序文本区域。在C语言中，标识符可能有表4-1所示的6个作用域之一。

表4-1 标识符作用域

类 型	声明有效的区域
顶层标识符	从声明点（4.2.3节）延伸到源程序文本结束
函数定义中的正式参数	从声明点延伸到函数体结束
函数原型中的正式参数[①]	从声明点延伸到原型结束
块（本地）标识符	从块中的声明点延伸到块结束
语句标号	包括所在的整个函数体
预处理器宏	从声明的**#define**命令延伸到源程序文本结束或第一个取消定义的**#undef**命令

① 这是标准C语言增加的。

函数定义和块中的非预处理器标识符（包括正式参数）通常称为块作用域或本地作用域。原型中的标识符为原型作用域。语句标号具有函数作用域，所有其他标识符具有文件作用域。

块通常是个复合语句。在C99中，还有与选择和迭代语句相关联的C隐式块。

每个标识符的作用域局限于所在的C语言源文件。但是，有些标识符可以声明为外部作用域，这时可以连接两个或多个文件中同一标识符的声明，见4.8节。

参考章节　**#define**预处理器命令　3.3；外部名称　4.8；原型　9.2；**#undef**预处理器命令　3.3

75

4.2.2 有效性

标识符的声明在一些上下文中有效，在这个上下文中使用标识符时，约束到相应声明（即标识符与这个声明相关联）。声明可能在整个作用域中有效，但也可以用作用域和有效性与第一个声明重叠的另一声明隐藏。

例　下列程序中，声明**foo**为整数变量，被声明**foo**为浮点数变量的内部声明隐藏。外部**foo**只在**main**函数体中隐藏。

```
int foo = 10; /* foo defined at the top level */
int main(void)
{
    float foo; /* this foo hides the outer foo */
    …
}
```

□

在C语言中，块开头的声明可以隐藏块外的声明。要让一个声明隐藏另一声明，声明的标识符必须相同，要属于同一个重载类，要声明两个不同作用域，其中一个作用域包含另一作用域。

在标准C语言中，函数定义中正式参数声明的作用域与函数体块开头声明的标识符作用域相同。但是，一些较早的C语言认为参数作用域包括块作用域。

例　下列**x**的重新声明在标准C语言中是错误的，但一些较早的C语言允许，使一些复杂的编程错误不被发现。

```
int f(x)
    int x;
{
    long x = 34;      /* invalid? */
    return x;
}
```

□

参考章节　块　8.4；重载类　4.2.4；参数声明　9.3；顶层声明　4.1

4.2.3 向前引用

标识符通常不能在完全声明之前使用。准确地说，我们定义标识符的声明点为包含标识符词法记号的声明符结尾处。声明点之后可以使用这个标识符。下例中，整型变量**intsize**可以在声明点之后的初始化代码中使用，因此可以初始化为自己的长度：

76

```
static int intsize = sizeof(intsize);
```

如果标识符在声明完成之前使用，则发生了向前引用。3种情况下C语言允许向前引用：

1. 语句标号可以在成为标号之前在**goto**语句中使用，因为它的作用域包括整个函数体：

```
if (error) goto recover;
…
recover:
    CloseFiles();
```

2. 可以声明不完整的结构、联合、数组或枚举类型、使它们在被完整定义之前用于某种用途（见5.6.1节）。
3. 函数可以独立于定义而声明，可以用一个声明，也可以在函数调用中出现作为隐式声明（见4.7和5.8节）。C99不允许函数调用隐式声明函数。

例 本例显示无效向前引用。编程人员要用**typedef**声明定义一个自引用结构。这里行中最后出现**cell**时是声明点，因此在结构中使用**cell**是无效的。

```
typedef struct { int Value; cell *Next; } cell;
```

要声明这种类型，正确的方法是使用结构标志**S**，在第一次出现时定义，然后在后面的声明中使用：

```
typedef struct S { int Value; struct S *Next; } cell;
```

□

此后还会介绍隐式声明（4.7节）和重复声明（4.2.5节）。

参考章节 重复声明 4.2.5；函数类型 5.8；**goto**语句 8.10；隐式声明 4.7；指针类型 5.3；结构类型 5.6

4.2.4 名称重载

在C语言和其他编程语言中，同一标识符可能会同时和多个程序实体相关联，这时称为名称重载，使用名称的上下文决定有效的关联。例如，标识符可能既表示变量名，又表示结构标志名。在表达式中使用时，使用变量关联；而在类型说明符中使用时，使用标志关联。 77

C语言中的名称有5种重载类（也称为命名空间），见表4-2。

表4-2 重 载 类

重 载 类	包括的标识符
预处理器宏名	由于预处理过程逻辑上发生在编译之前，因此预处理器使用的名称独立于C语言程序中任何名称
语句标号	命名的语句标号属于语句的一部分，语句标号的定义总是跟一个冒号（不是**case**标号的一部分），语句标号总是放在保留字**goto**之后
结构标志、联合标志与枚举标志	这些标志属于结构说明符、联合说明符与枚举类型说明符的一部分，出现时总是放在保留字**struct**、**union**或**enum**前面
成员名（标准C语言中的成员）	成员名在与每个结构和联合类型相关联的命名空间中分配，同一标识符可以同时作为多个结构或联合中的成员名。成员名定义总是放在结构或联合的类型说明符中。成员名使用时总是跟在选择运算符.与->之后
其他名称	其他名称包括变量、函数、**typedef**名称和枚举常量，属于另一个重载类

这些重载规则与原先C语言中的定义稍有不同。首先，语句标号最初与原标识符在同一命名空间。第二，所有结构成员名与联合成员名放在同一个命名空间，而不是每种类型一个命名空间。

用几个关联重载名称时，每个关联有自己的作用域，可以由独立于其他关联的另一声明隐藏。例如，如果一个标识符同时作为变量与结构标志，则内部块可以重新定义变量关联而不改变标志关联。

C++把结构标志与联合标志归为“其他”命名空间（见4.9.2节）。

参考章节 成员名 5.6.3；重复定义 4.2.5；枚举标志 5.5；**goto**语句 8.10；选择运算符 7.4.2；语句标号 8.10；结构标志 5.6；结构类型说明符 5.6；**typedef**名称 5.10；联

合标志5.7；联合类型说明符　5.7

4.2.5 重复声明

同一块或顶层中（在同一重载类内）两次声明同一名称是无效的。这种声明称为冲突。

78 **例**　下面两个**howmany**声明是冲突的，但**str**声明不是（因为它们不在同一重载类内）。

```
extern int howmany;
extern char str[10];
typedef double howmany();
extern struct str {int a, b;} x;
```

□

禁止重复声明的规则有两个例外。第一，同一名称可以有多个外部（引用）声明，只要每个实例中对名称指定相同类型。这个例外反映了这样的思想：两次声明同一个外部库函数是有效的。

第二，如果一个标识符声明为外部标识符，则声明后面可以在程序中进行名称定义（4.8节），只要这个定义对名称指定与外部声明相同的类型。这个例外使用户可以对变量和函数产生有效的向前引用。

例　我们定义两个函数**f**和**g**，它们相互引用。通常，在**g**中使用**f**时是无效的向前引用。但在**g**定义前面加上**f**的外部声明后，编译器就可以得到编译**g**所要的**f**信息（如果没有**f**的初始声明，则一遍编译器无法在编译**g**时知道**f**返回**double**类型的值）。

```
extern double f(double z);

double g(double x, double y)
{
    … f(x-y) …
}

double f(double z)
{
    … g(z, z/2.0) …
}
```

□

参考章节　定义与引用声明　4.8；**extern**存储类　4.3；向前引用　4.2；，重载类　4.2；**static**存储类　4.3

4.2.6 重复有效性

由于C语言作用域规则指定名称作用域从声明点开始，而不是从所在块头开始，因此可能在同一块的不同部分引用两个互不冲突的声明。

例　下列代码中，**B**块引用两个**i**变量，外部块中声明的整型**i**用于初始化变量**j**，然后声明
79 一个浮点型变量**i**，隐藏第一个**i**。

```
{
   int i = 0;
   …
   B: {
         int j = i;
         float i = 10.0;
         …
      }
}
```

初始化变量j时引用的i具有歧义性。用哪个i呢？大多数编译器能按我们的明显意图做，B块中第一次使用i时与外部定义相关联，然后i的重新定义在块的其余部分隐藏第一个i。这是标准C语言规则。这种用法是不良编程风格，应该避免。 □

4.2.7 生存期

变量与函数不同于类型，在运行时具有存在性，即要分配存储空间。这些对象的*生存期*是保持分配存储空间的时间。标准C语言将其称为*存储期间*。

如果分配存储空间在程序开始执行之前进行，而且保持到程序终止，则对象具有*静态生存期*。在C语言中，所有函数都具有静态生存期，顶层声明中声明的所有变量也都有静态生存期。块中声明的变量是否具有静态生存期取决于具体声明。

如果对象在进入块或函数时生成，在退出块或函数时删除，则称为*本地生存期*。如果具有本地生存期的变量有一个初始化语句，则变量在每次生成时初始化。正式参数具有本地生存期，块开头声明的变量是否具有本地生存期取决于具体声明。C语言中把具有本地生存期的变量称为*自动变量*。

最后，C语言中的数据对象还可能具有*动态生存期*，编程人员可以随意地显式创建和删除对象。但是，动态对象要通过**malloc**之类的特殊库程序生成，不属于C语言的一部分。

参考章节 **auto**存储类 4.3；*初始化语句* 4.6；**malloc**函数 16.1；**static**存储类 4.3；*存储分配函数* 16.1

4.2.8 初值

分配变量存储空间不一定对这个存储空间建立初始内容。C语言中大多数变量声明可以有初始化语句，这个表达式可以在分配存储空间时设置变量的初值。如果本地变量没有指定初始化语句，则分配空间之后其数值是不确定的（静态变量默认初始化为0）。

一定要记住，静态变量只初始化一次，即使程序在这个变量的作用域之外执行，静态变量也保留这个值。

80

例 下列代码中，在块开头声明两个变量L和S，两者都初始化为0。两个变量都是本地作用域的，但S为静态生存期，而L为本地（自动）生存期。每次进入块时，这两个值加1，打印新值。

```
{
    static int S = 0;
    auto int L = 0;
    L = L + 1;
    S = S + 1;
    printf("L = %d, S = %d\n", L, S);
}
```

打印结果如何？如果块执行多次，则输出如下：

```
L = 1, S = 1
L = 1, S = 2
L = 1, S = 3
L = 1, S = 4
...
```

□

在块开头声明的自动变量初始化时有一个危险的特性。如果块正常进入，即控制从块开头开始，则能保证只对变量初始化一次，即控制流进入块开头时。通过使用语句标号和**goto**语句，

可以跳到块中间，这时就不能保证将自动变量初始化。事实上，大多数标准实现和非标准实现都不对自动变量初始化。对于**switch**语句，通常会跳到**switch**体中**case**或**default**标号的语句，因此第一个这类标号之前的自动变量并不初始化。

例　下面变量**sum**的初始化过程在**goto**语句将控制转移到标号**L**时可能不发生。这样就使**sum**的开始值不确定。

```
goto L;
…
{
    static int vector[10] = {1,2,3,4,5,6,7,8,9,10};
    int sum = 0;
L:
    /* Add up elements of "vector". */
    for ( i=0; i<10; i++ ) sum += vector[i];
         printf("sum is %d", sum);
}
```

□

81

参考章节　**goto**语句　8.10；变量初始化　4.6；存储类　4.3；**switch**语句　8.7

4.2.9 外部名称

作用域与有效性的一个特例是外部标识符，也称为具有外部连接的标识符。对于同一C语言程序的所有文件中的同一个外部标识符，其所有实例被强制要求引用同一个对象或函数，要在每个文件中用兼容类型声明，否则结果无法确定。

外部名称要显式或隐式地用**extern**声明，但并不是所有用**extern**声明的名称都是外部名称。外部名称通常在C语言程序顶层声明，因此具有文件作用域。但非标准实现对块中声明的外部名称采用不同处理方法。

例　许多C语言编译器能够接受下列程序段，它在块中声明外部名称，然后在块外使用：

```
{
    extern int E;
    …
}
E = 1;
```

根据正常的块作用域规则，声明在块外无效，但许多C语言实现隐式对**E**指定文件作用域，因此这个代码块可以顺利编译。标准C语言要求声明具有块作用域，但没有指定上述程序段无效。技术上，这种情况下的实现行为是未定义的，因此符合实现可以接受这个程序。我们认为编程人员应该把这个代码段看成是编程错误，即使编译器能够接受它，即使它的运行行为是正确的。□

如果同一文件或同一程序中不同文件有两个外部声明对同一标识符指定不兼容类型，则一定会产生错误。

例　下列程序源文件中对**x**的两个声明不冲突，但其在运行时的行为是未定义的：

```
int f() { extern int X; return X; }
double g() { extern double X; return X; }
```

□

参考章节　外部名规则　2.5；外部名定义与引用　4.8；作用域　4.2.1；类型兼容性　5.11；有效性　4.2.2

4.2.10 编译名称

前面主要介绍了变量与函数，其在运行时具有存在性。但是，作用域和有效性规则也适用于运行时不具有存在性的与对象相关联的标识符：**typedef**名称、类型标志和枚举常量。声明这些标识符时，其作用域与同一地址定义的变量作用域相同。宏和标号也是编译名称，但其作用域与同一地址定义的变量的作用域不同。 82

参考章节 枚举常量 5.5；作用域 4.2.1；结构类型 5.6；**typedef**名称 5.10；有效性 4.2.2

4.3 存储类说明符与函数说明符

我们要继续介绍声明的组成部分：存储类说明符、类型说明符与限定符、函数说明符、声明符和初始化语句。

存储类说明符确定所声明对象的生存期（除了**typedef**是个特例）。声明中最多可以出现一个存储类说明符。习惯上把存储类说明符放在声明中的类型说明符与限定符之前。

storage-class-specifier : one of
auto extern register static typedef

表4-3列出了存储类的含义，注意并不是所有存储类都在每个声明上下文中都被允许。

表4-3 存储类说明符

说 明 符	用 法
auto	只在块内的变量声明[①]中允许，表示变量具有本地（自动）生存期（这是默认，因此C语言程序中很少看到**auto**说明符）
extern	出现在顶层或块中的外部函数与变量声明中，表示声明的对象具有静态生存期，连接程序知道其名称，参见4.8节
register	可以用于本地变量或参数声明，作用相当于**auto**，只不过它要向编译器提供提示说该对象会被频繁使用，应在采取一种最小化访问时间的分配方式
static	可以放在函数与变量声明中。在函数定义时，其只用于指定函数名而不将函数导出到连接程序。在函数声明中，其表示文件后面会定义声明的函数，存储类为**static**。在数据声明中，总是表示定义的声明不导出到连接程序。用这个存储类声明的变量具有静态生存期（而**auto**则指定本地生存期）
typedef	表示声明定义新的数据类型名，而不是函数与变量声明。数据类型名出现在变量声明中出现变量名的地方，数据类型本身是变量名要指定的类型（见5.10节）

① C99允许在块中任何地方声明。较早版本的C语言只能在第一条语句之前声明。 83

标准C语言允许对任何类型变量与参数使用**register**存储类说明符，但不允许显式（用**&**运算符）或隐式（在计算数组下标时把数组名转换成指针）计算这种对象的地址。许多非标准C语言编译器具有不同行为：

- 可能只限于标量类型对象使用**register**存储类说明符。
- 可能允许对**register**对象使用**&**。
- 可能隐式加宽用**register**声明的小对象（例如，把声明**register char x**当作**register int x**）。

实现可以把**register**存储类说明符当作与**auto**说明符相同。但是，编程人员可以对函数中大量使用的一个或两个变量使用**register**存储类说明符以提高性能。对太多声明使用**register**存储类说明符通常效率不高，反而不利。在最新编译器中使用**register**存储类说明符通常效果不大，因为这些编译器已经在需要时将变量分配到寄存器。

参考章节　地址运算符**&** 7.5.6；正式参数声明 9.3；初始化语句 4.6；下标 7.4.1；顶层声明 4.1；**typedef**名称 5.10

4.3.1 默认存储类说明符

如果声明中没有提供存储类说明符，则根据表4-4所示的上下文设定一个存储类说明符。

表4-4　默认存储类说明符

声明位置	声明类型	默认存储类说明符
顶层	所有	**extern**
函数参数	所有	无（如“非**register**”）
块中	函数	**extern**
块中	非函数	**auto**

顶层声明中省略存储类说明符不同于提供**extern**，见4.8节介绍。作为良好的编程风格，编程人员应在块中声明外部函数时提供存储类说明符**extern**。**auto**存储类很少在C语言程序中看到，它通常是默认值。

参考章节　块 8.4；参数声明 9.3；顶层声明 4.1，4.8

4.3.2 存储类说明符举例

84

下面显示堆排序算法的实现方法，本书不准备介绍其工作细节。

例　这个算法把数组看成二叉树，元素**b[k]**的两个子树为元素**b[2*k]**与**b[2*k+1]**。这里的堆（heap）是一棵树，每个节点包含的数值不小于该节点的后代节点包含的数值。

```
#define SWAP(x, y) (temp = (x), (x) = (y), (y) = temp)

static void adjust (int v[], int m, register int n)
/* If v[m+1] through v[n] is already in heap form,
   this puts v[m] through v[n] into heap form. */
{
   register int *b, j, k, temp;
   b = v - 1; /* b is "1-origin", customary in heapsort,
                 i.e., v[j] is the same as b[j-1] */
   j = m;
   k = m * 2;
   while (k <= n) {
      if (k < n && b[k] < b[k+1]) ++k;
      if (b[j] < b[k]) SWAP(b[j], b[k]);
      j = k;
      k *= 2;
   }
}

/* Sort v[0]…v[n-1] into increasing order. */
void heapsort(int v[], int n)
```

```
{
    int *b, j, temp;
    b = v - 1;
    /* Put the array into the form of a heap. */
    for (j = n/2; j > 0; j--) adjust(v, j, n);
    /* Repeatedly extract the largest element and
       put it at the end of the unsorted region. */
    for (j = n-1; j > 0; j--) {
        SWAP(b[1], b[j+1]);
        adjust(v, 1, j);
    }
}
```

辅助函数**adjust**不一定要在外部有效，因此声明为**static**。**adjust**函数的速度对排序性能至关重要，因此其局部变量的存储类为**register**以向编译器提供提示。正式参数**n**也在**adjust**函数中重复引用，因此用存储类**register**指定。**adjust**的另外两个正式参数默认为非**register**。

主函数为**heapsort**，应允许排序包用户访问，因此使用默认存储类**extern**。函数**heapsort**的局部变量不影响性能，因此指定默认存储类**auto**。 □ 85

4.3.3 函数说明符

函数说明符是C99中增加的。

function-specifier: (C99)
inline

函数说明符**inline**只能在函数声明中出现，这种函数称为内联函数。函数说明符可以多次出现而不改变含义。使用**inline**提示C实现，函数调用应尽量快。

内联函数的详细规则见第9章。

参考章节 内联函数 9.10

4.4 类型说明符与限定符

类型说明符对所声明的程序标识符的数据类型提供一定信息。其他类型信息由声明符指定。类型说明符还可以定义（作为副作用）类型标志、结构成员名与联合成员名和枚举常量。

类型限定符**const**、**volatile**与**restrict**指定类型的其他属性，这些属性只在通过左值访问该类型对象时与类型有关：

type-specifier:
enumeration-type-specifier
floating-point-type-specifier
integer-type-specifier
structure-type-specifier
typedef-name
union-type-specifier
void-type-specifier

type-qualifier:
const
volatile
restrict (C99)

例　下面是类型说明符的一些例子：

```
void                    union { int a; char b; }
int                     enum {red, blue, green}
unsigned long int       char
my_struct_type          float
```

□

类型说明符将在第5章详细介绍，特定类型说明符的介绍也放到第5章，下面几节将介绍一些与类型说明符有关的一般问题。

参考章节　声明符　4.5；枚举类型说明符　5.5；浮点数类型说明符　5.2；整数类型说明符　5.1；左值　7.1；结构类型说明符　5.6；类型限定符　4.4.3；**typedef**名称　5.10；联合类型说明符　5.7；**void**类型说明符　5.9

4.4.1 默认类型说明符

最初，C语言允许省略变量声明和函数定义中的类型说明符，这时默认类型说明符为**int**。但现代C语言认为这是不良编程风格，事实上C99把它当作错误处理。较早的编译器没有实现**void**类型，因此函数定义中省略类型说明符的目的是告诉读者这个函数不返回数值（但编译器要假设其返回数值）。

例　在标准之前的C语言中，经常看到如下函数定义：

```
/* Sort v[0]...v[n-1] into increasing order. */
sort(v, n)
    int v[], n;
{
    ...
}
```

现代C语言用**void**类型声明这些函数：

```
/* Sort v[0]...v[n-1] into increasing order. */
void sort(int v[], int n)
{
    ...
}
```

例　使用没有实现**void**类型的编译器时，最好自己定义**void**，然后显式使用，而不是完全省略类型说明符：

```
/* Make "void" be a synonym for "int". */
typedef int void;
```

至少我们知道有一个编译器实际保留标识符**void**，但没有实现。对这个编译器，下列预处理器定义：

```
#define void int
```

是极少数可以用保留字作为宏名的情形。

□

例　声明语法（见4.1节）要求声明包含存储类说明符、类型说明符、类型限定符及三者的组合，这个要求可以避免语言中的语法歧义性。如果所有说明符和限定符都用默认，则声明

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

变成

```
f();
```

语法上等于由函数调用构成的语句。我们认为最好的编程风格是始终包括类型说明符并让存储类说明符保持默认（至少为**auto**）。 □

例 对LALR(1)文法迷要说的最后一点是：函数定义中可以同时省略类型说明符和存储类说明符，这在C语言程序中很常见，例如：

```
main() { … }
```

这里没有语法歧义性，因为函数声明中的声明符后面要加上逗号或分号，而函数定义中的声明符后面是左花括号。 □

参考章节 声明 4.1；函数定义 9.1；**viod**类型说明符 5.9 □

4.4.2 忽略声明符

下面介绍声明与类型说明符的一个微妙之处。结构、联合或枚举定义的类型指定符定义新类型或枚举常量。如果只要定义类型，则可以从声明中省略所有声明符，只编写类型说明符。标准C语言中的声明要有声明符，然后或者定义结构标志或联合标志，或者定义枚举常量。在传统C语言中，无意义的声明通常被忽略。

例 下列声明只有一个类型说明符，定义新结构类型**S**，具有成员**a**和**b**：

```
struct S { int a, b; }; /* Define struct S */
```

后面可以只用说明符引用这个类型：

```
struct S x, y, z; /* Define 3 variables */
```

但是，下列声明在标准C语言中无意义，是非法的：

```
struct { int a, b; };                /* no tag */
int ;                                /* no declarator */
static struct T { int a, b; }; /* extra storage class */
```

88

第一种情况没有结构标志，因此无法在后面的程序中引用。第二种情况，声明根本无效。第三种情况，提供的存储类说明符会被忽略。也许您以为下列形式的声明会使**x**与**y**的存储类为**static**，其实不然。

```
struct T x, y;
```

□

参考章节 枚举类型 5.5；声明符 4.5；结构类型 5.6；类型说明符 4.4；联合类型 5.7

4.4.3 类型限定符

类型限定符**const**与**volatile**是C89中增加的，**restrict**是C99中增加的。用这些限定符的任何组合声明的标识符称为限定类型的，因此每个未限定类型共有7种可能的限定版本（类型限定符的顺序并不重要）。这7种限定版本之间及其与未限定类型之间互不兼容。如果声明中同一限定符多次出现，则在C99中会忽略多余的限定符，而在C89中会出错。

类型限定符规定类型的其他属性只在通过左值（指示符）访问该类型对象时有关。如果所在上下文中需要数值而不是指示符，则从这个类型中删除限定符。即在表达式L=R中，等号右边操作数的类型总为未限定类型，即使用类型限定符声明，而左操作数则保持限定，因为它是

在左值表达式上下文中使用的。

除了在声明顶层出现外，类型限定符还可以放在指针声明符和（C99中的）数组声明符中。

例　如果C语言编译器不支持类型限定符，则可以提供下列宏定义，使类型限定符不会造成编译失败。当然，这时类型限定符也不起作用。

```
#ifndef __STDC__
#define const /*nothing*/
#define volatile /*nothing*/
#define restrict /*nothing*/
#endif
```

□

参考章节　**#ifndef**　3.5.3；**__STDC__**　11.3；数组声明符　4.5.3；指针声明符　4.5.2；类型兼容性　5.11

4.4.4　const类型限定符

const限定类型的左值表达式不能用来修改对象，即这种左值不能作为赋值表达式的左操作数或递增与递减运算符的操作数，目的是用**const**限定符指定数值不变的对象，让C语言编译器保证编程人员不改变这个值。

89

例　下列声明指定**ic**是个常量整数，数值为37：

```
const int ic = 37;
…
ic = 5; /* Invalid */
ic++;   /* Invalid */
```

□

const限定符还可以放在指针声明符中，使其可以同时声明“常量指针”和“常量数据的指针”：

```
int * const const_pointer;
const int *pointer_to_const;
```

这个语法容易引起混淆，例如，常量指针与常量整数的类型限定符**const**放在不同位置。使用**typedef**名称时，上例中的常量指针**const_pointer**可以声明如下：

```
typedef int *int_pointer;
const int_pointer const_pointer;
```

这样就使**const_pointer**像是常量**int_pointer**的指针，但其实不是，它还是**int**（非常量）的常量指针。事实上，由于类型说明符和类型限定符的顺序并不重要，因此最后一个声明也可以写成：

```
int_pointer const const_pointer;
```

可以改变类型为常量数据的指针的变量，但所指的对象不能改变。要产生这种类型的表达式，可以对**const**限定类型的值采用地址运算符**&**。为了保护常量数据的完整性，只能用显式转换将“常量*T*的指针”类型的值赋予“*T*的指针”类型的对象。

例

```
const int *pc; /* pointer to a constant integer */
int *p, i;
const int ic;
pc = p = &i;     /* OK */
pc = &ic;        /* OK */
```

90

```
*p = 5;          /* OK */
*pc = 5;         /* Invalid */
pc = &i;         /* OK */
pc = p;          /* OK */
p = &ic;         /* Invalid */
p = pc;          /* Invalid */
p = (int *)&ic;  /* OK */
p = (int *)pc;   /* OK */
```

□

const的语言规则不是牢不可破的，只要编程人员狠下功夫，也能绕过或越过这些规则。例如，可以将常量对象的地址传递给没有原型的外部函数，这个函数可以修改常量对象。但是，实现可以在只读存储空间中分配**const**限定类型的静态对象，因此改变对象会造成运行错误。

例 下列程序段演示了绕过**const**限定符的危险：

```
const int * pc;
int * p;
const int ic = 0;
...
pc = &ic;        /* OK */
p = (int *)pc;   /* Valid, but dangerous */
*p = 5;          /* Valid, but may cause a run-time error */
```

□

最后，顶级声明的类型限定符为**const**而没有显式存储类时，在C语言中被当作**extern**存储类。

参考章节 赋值表达式 7.9；自增与自减表达式 7.4；指针声明符 4.5.2

4.4.5 volatile类型限定符与序列点

volatile类型限定符告诉标准C语言实现，某些对象可以用实现不能控制的方式改变数值。易失对象（即用**volatile**限定类型的lvalue表达式访问的任何对象）不能参与优化，假设这些优化没有隐藏的副作用。

更准确地说，标准C语言在C语言程序中引入了序列点的概念。序列点不会出现在一个大表达式所包含的部分表达式完成时，而只能出现在包含于大表达式的所有表达式完成之后，即可以存在于表达式语句结束时，**if**、**switch**、**while**与**do**语句的控制表达式之后，**for**语句的每个控制表达式之后，逻辑AND(**&&**)、逻辑OR(||)、条件(**?:**)与逗号(,)运算符的第一个操作数之后，**return**语句表达式之后以及初始化语句之后。其他序列点可能出现在完整声明符末尾，函数调用中求值所有参数之后，返回库函数之前，与**printf/scanf**转换指定符相关联的操作之后以及调用供**bsearch**与**qsort**使用的比较函数时。

91

跨序列点不能优化易失对象的引用与修改，但序列点之间可以进行优化。源代码中其他引用与修改是C语言标准允许的。但根据我们的经验，编程人员喜欢实现能够按指定方式引用与修改易失对象。编程人员很容易从易失对象中拷贝数值以促进优化。

例 下列程序段中**j**在循环之前赋值：

```
extern int f(int);
auto int i,j;
...
i = f(0);
while (i) {
```

```
    if (f(j*j)) break;
}
```

如果变量i在其生存期期间不再使用，则传统C语言实现允许将这个程序段改写如下：

```
if (f(0)) {
    i = j*j;
    while( !f(i) ) ;
}
```

i的第一个赋值被清除，i作为临时变量复用，保存j*j的值，其在循环体外求值一次。如果i和j声明如下，则不允许以上优化：

```
auto volatile int i,j;
```

但是，可以将循环写成如下形式，消除if语句控制表达式末尾序列点之前的一个j引用：

```
i = f(0);
while (i) {
    register int temp = j;
    if (f(temp*temp)) break;
}
```

□

指针声明符的新语法允许声明“volatile的指针”类型，这种指针的引用可以优化，但其所指的对象的引用不可以优化，只能用显式转换将“volatile *T*的指针”类型的值赋予“*T*的指针”类型的对象。

92

例 下面是volatile对象的有效与无效用法的例子：

```
volatile int * pv;
int *p;
pv = p;         /* OK */
p = pv;         /* Invalid */
p = (int *)pv; /* OK */
```

□

volatile最常见的用法是对特殊内存地址提供可靠的访问，计算机硬件或中断处理器之类的异步进程使用这些特殊内存地址。

例 下面是个典型例子。计算机上有3个特殊硬件地址：

地　址	用　途
0xFFFFFF20	输入数据缓冲区
0xFFFFFF24	输出数据缓冲区
0xFFFFFF28	控制寄存器

程序可以读取而不能写入控制寄存器和输入数据缓冲区。控制寄存器的倒数第3个有效位是输入可用位（input available），在外部源数据到达时设置为1，程序读取输入数据缓冲区中的数据时自动将其设置为0（然后缓冲区内容变成未定义，直到输入可用位再次变成1）。控制寄存器的倒数第2个有效位是输出可用位（output available），在外部设备准备接收数据时将其设置为1，程序将数据放到输出数据缓冲区时，这个位自动设置为0，并写出数据。控制位为0时将数据放进输出数据缓冲区会出现无法预测的结果。

函数**copy_data**将数据从输入拷贝到输出，直到遇到输入值0，然后返回拷贝的字符数。这个函数没有提供溢出和其他错误条件的处理方法。

```
typedef unsigned long datatype, controltype, counttype;

#define CONTROLLER \
    ((const volatile controltype * const) 0xFFFFFF28)
#define INPUT_BUF \
    ((const volatile datatype * const) 0xFFFFFF20)
#define OUTPUT_BUF \
    ((volatile datatype * const)  0xFFFFFF24)
#define input_ready ((*CONTROLLER) & 0x4)
#define output_ready ((*CONTROLLER) & 0x2)
counttype copy_data(void)
{
    counttype count = 0;
    datatype temp;
    for(;;) {
        while (!input_ready) ;     /* Wait for input */
        temp = *INPUT_BUF;
        if (temp == 0) return count;
        while (!output_ready) ;    /* Wait to do output */
        *OUTPUT_BUF = temp;
        count++;
    }
}
```

□

93

参考章节 **bsearch** 20.5；转换规范 15.8.2，15.11.2；声明符 4.5；初始化语句 4.6；指针声明符 4.5.2；**qsort** 20.5

4.4.6 restrict类型限定符

类型限定符**restrict**是C99中增加的，它只能用于限定对象指针或不完整类型，作为C语言编译器的非别名提示。即目前只有指针是访问其指向对象的惟一方式。破坏这个假设会造成不确定行为。“目前”指在某些情况下，函数或块中可以从原始的限制限定指针生成别名，只要在函数或块结束之前删除这些别名即可。C99标准提供了**restrict**的准确数学定义，但下面介绍一些常见情形。

1. 假设用**restrict**声明的文件作用域指针只用于访问所指对象，这样可以声明一个全局指针，并在运行时用**malloc**对其进行初始化。

```
extern double * restrict ptr;
...
void initialize(void)
{
   ptr = my_malloc( sizeof(double) );
}
```

2. 限制指针是函数参数时，假设该指针是函数执行开始时访问其所指向对象的惟一方式，因此不是从参数生成的其他指针不能修改对象。例如，**memcpy**函数（与**memmove**不同）要求源和目标内存区不重叠。在C99中，这一要求可以用函数原型表示：

94

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

```
void *memcpy(
    void * restrict s1,
    const void * restrict s2,
    size_t n);
```

3. 两个限制指针或一个限制指针和一个非限制指针可以引用同一对象，只要这个对象在限制指针存在期间不被修改。例如，下列函数将两个向量相加，将和存放在第3个向量中：

```
void add(int n, int * restrict dest,
    int * restrict op1, int * restrict op2)
{
    int i;
    for (i = 0; i < n; i++)
        dest[i] = op1[i] + op2[i];
}
```

如果**a**和**b**是长度为**N**的不相交数组，则可以调用**add(N,a,b,b)**，得到**op1**与**op2**指定的数组**b**，因为数组**b**没有被修改。当然，这要求知道**add**的实现方法，编程人员如果只看到**add**原型，则无法知道这种情况是否安全。

4. 结构成员可以是限制指针，即生成结构实例时，限制指针是引用指定对象的惟一方法。

restrict加进C语言之前，编程人员要靠不可移植的杂注开关或编译器开关进行某种指针优化，这样操作在对象一次只能由一个指针访问时是安全的。这些优化可以大大加快运行速度。

省略**restrict**并不改变程序的含义，C语言实现可以忽略**restrict**。本书的许多库函数原型都使用**restrict**限定符。使用C99之前实现的编程人员可以省略或不管**restrict**。

参考章节 **malloc** 16.1; **memcpy** 14.3

4.5 声明符

声明符引入声明的名称，并提供其他类型信息。过去的其他编程语言没有类似于C语言声明符的项目：

declarator:
    pointer-declarator
    direct-declarator

95

direct-declarator:
    simple-declarator
    (*declarator*)
    function-declarator
    array-declarator

下面几节将介绍不同类型的声明符。

4.5.1 简单声明符

简单声明符用于定义算术类型、枚举类型、结构类型以及联合类型的变量：

simple-declarator:
    identifier

假设*S*为类型说明符，*id*为任何标识符，则下列声明表示*id*为*S*类型，*id*称为简单声明符：

```
S id ;
```

例

声 明	x的类型
`int x;`	整型
`float x;`	浮点型
`struct S { int a; float b;} x;`	含有两个成员的结构

□

简单声明符可以在类型说明符提供所有类型信息的声明中使用，例如算术类型、枚举类型、结构类型、联合类型以及void类型，还有**typedef**名称表示的类型。指针类型、数组类型和函数类型要求使用更复杂的声明符。但是，每个声明符包括一个标识符，因此可以说声明符"包围"标识符。

参考章节 类型说明符 4.4；结构类型 5.6；**typedef**名称 5.10

4.5.2 指针声明符

指针声明符用于声明指针类型的变量。下列语法中的*type-qualifier-list*是标准C语言增加的，在较早的编译器中被省略：

pointer-declarator:
pointer direct-declarator

pointer:
* *type-qualifier-list*$_{opt}$
* *type-qualifier-list*$_{opt}$ *pointer*

type-qualifier-list: (C89)
type-qualifier
type-qualifer-list type-qualifier

96

假设*D*是包围标识符*id*任何声明符，声明"*S D*;"表示*id*的类型为"...*S*"，则声明

```
S *D ;
```

表示*id*的类型为"指向*S*的指针……"。指针声明符语法中可选的*type-qualifier-list*只能出现在标准C语言中。出现时，限定符适用于指针，而不适用于指针所指的对象。

例 在下表的3个**x**声明中，*id*是**x**，*S*是**int**，"……"分别是""、"数组"和"返回函数"。

声 明	x的类型
`int *x;`	指向**int**的指针
`int *x[];`	指向**int**的指针数组
`int *x();`	指向**int**的指针的返回函数

□

例 下列声明中，**ptr_to_const**是常量**int**的非常量指针，而**const_ptr**是非常量**int**的常量指针。

```
const int * ptr_to_const;
int * const const_ptr;
```

□

参考章节 数组声明符 4.5.3；**const**类型限定符 4.4.4；函数声明符 4.5.4；指针类型 5.3；类型限定符 4.4.3

4.5.3 数组声明符

数组声明符用于声明数组类型的对象：

array-declarator:
　　direct-declarator [*constant-expression*$_{opt}$]　　*(until C99)*
　　direct-declarator [*array-qualifier-list*$_{opt}$ *array-size-expression*$_{opt}$] *(C99)*
　　direct-declarator [*array-qualifier-list*$_{opt}$ *]　　*(C99)*

constant-expression:
　　conditional-expression

97

array-qualifier-list:
　　array-qualifier
　　array-qualifier-list array-qualifier

array-qualifier:
　　static
　　restrict
　　const
　　volatile

array-size-expression:
　　assignment-expression
　　*

假设*D*是包围标识符*id*的任何声明符，声明“*S D*;”表示*id*的类型为“...*S*”，则声明

S **(*D*) [*e*] ;**

表示*id*的类型为“*S*数组的……”（构造声明符时，可以根据优先顺序规则省略括号，见4.5.5节）。类型*S*不能是不完整数组类型或函数类型。

大多数情况下，方括号中出现整数常量表达式*e*，指定数组元素个数。这个数字应为大于0的整数值。C语言数组的基数为0，即声明**int A[3]**定义元素**A[0]**、**A[1]**与**A[2]**。高维数组声明为数组的数组（见5.4.2节）。

例 在下表的3个**x**声明中，*id*是**x**，*S*是**int**，“……”分别是“”、“指针”和“数组”。

声　明	x的类型
`int (x)[5];`	整数数组
`int (*x)[5];`	指向整数数组的指针
`int (x[5])[5];`	整数数组的数组
`int x[5][5];`	整数数组的数组（与上一声明相同）

□

数组声明符的方括号中也可以不出现整数常量表达式。数组声明可以有3种变形：不完整数组类型、变长数组以及在*array-declarators*中使用*array-qualifiers*（类型限定符和**static**）。

不完整数组类型　如果方括号中是空的，则声明符描述不完整数组类型。不能创建不完整数组

98　类型的对象，因为其长度是未知的。可以声明不完整数组类型的指针。下列情形可以省略数组长度：

1. 声明的数组是函数的正式参数。由于数组参数要转换为指针，因此不需要数组长度。如果数组是多维的，则只能省略最左边的维的长度。例如：

```
int f(int ary[]); /* array of unspecified length */
```

2. 声明符带有初始化语句，可以求出数组长度。处理初始化语句之后，类型不再是不完整数组类型。例如：

```
char prompt[] = "Yes or No?";
```

3. 声明不是定义的出现值，而是引用在其他地方定义的对象，然后就不再是不完整数组类型。

对于多维数组，只有最左边的维的长度可以省略。还可以声明不完整数组类型的指针。例如：

```
extern int matrix[][10]; /* incomplete type */

...
static int matrix[5][10]; /* no longer incomplete */
```

4. 在C99中，结构的最后一个成员可以是灵活数组成员，不声明长度。

声明n维数组时，要包括最后$n-1$维的长度，以便确定访问算法。

变长数组 在C99中，如果*array-declarator*方括号中的*array-size-expression*为*或非常量表达式，则这个声明符描述了一个变长数组。*只能在不属于函数定义的函数原型中的数组参数声明内出现。变长数组不是不完整的数组类型。5.4.5节将会介绍变长数组及其在函数原型中的用法。

数组限定符 C99中允许*array-declarator*方括号中使用*array-qualifier-list*，但只能在声明数组类型的函数参数时使用。见9.3节介绍。

参考章节 数组类型 5.4；赋值表达式 7.9；条件表达式 7.8；常量表达式 7.11；灵活数组成员 5.6.8；正式参数 9.3；初始化语句 4.6；引用与定义声明 4.8；类型限定符 4.4.3；变长数组 5.4.5

4.5.4 函数声明符

函数声明符声明或定义函数和声明以函数指针为成分的类型：

99

function-declarator:
    direct-declarator (*parameter-type-list*) (C89)
    direct-declarator (*identifier-list*$_{opt}$)

parameter-type-list:
    parameter-list
    parameter-list , ...

parameter-list:
    parameter-declaration
    parameter-list , *parameter-declaration*

parameter-declaration:
    declaration-specifiers declarator
    declaration-specifiers abstract-declarator$_{opt}$

identifier-list:
    identifier
    parameter-list , *identifier*

假设D是标识符*id*所在的任何声明符，声明“*S D*;”表示*id*的类型为“*...S*”，则声明

```
S (D)(P);
```

表示*id*的类型为“带有参数P的S的返回函数”。构造声明符时，可以根据优先级规则省略D两边的括号（见4.5.5节）。声明符语法中的*parameter-type-list*表示这个声明符采用标准C语言原型形式。没有列表时则表示为传统C语言原型形式，在传统C语言编译器和标准C语言编译器中都能接受。

例 下面是一些函数声明符的例子：

声　明	x的类型
`int x();`	未指定参数的函数，返回整数
`int x(double, float);`	指定一个双精度浮点数和另一个浮点数为参数的函数，返回整数（原型）
`int x(double d, float f);`	同上一声明
`int (*x)();`	指向未指定参数的函数的指针，返回整数
`int (*x[])(int,...);`	指向的函数指针数组，该函数以一整数开始且参数数目可变，返回整数（原型）
`int (* const x) (void)`	指向无参数函数的常量指针，返回整数

100

□

根据是否在函数定义、对象类型声明或函数类型声明中出现，函数声明符有几个限制。表4-5列出了函数声明符可能的形式，指明是标准C语言原型形式还是传统C语言形式，显示是否可以在函数定义中出现或在函数类型声明中出现，显示指定的参数信息。表中T_x指语法"*declaration specifiers declarator*"（即包括参数名x的参数类型声明）。T_z指

declaration-specifiers *abstract-declarator*$_{opt}$

即省略参数名的参数类型声明。

表4-5 函数声明符

语　法	形　式	出现位置	指定的参数
f()	传统C语言	函数定义	无参数
f()	传统C语言	函数类型声明	多个参数
f(x,y,...,z)	传统C语言	函数定义	固定参数①
f(void)	标准C语言原型	两者均可	无参数
$f(T_x,T_y,...,T_z)$	标准C语言原型	函数类型声明	固定参数
$f(T_x,T_y,...,T_z,...)$	标准C语言原型	函数类型声明	固定参数，加额外的参数②
f(T,x,T,y,...,T,z)	标准C语言原型	两者均可	固定参数
f(T,x,T,y,...,T,z,...)	标准C语言原型	两者均可	固定参数，加额外的参数②

① 在标准C语言之前，可以有其他未指定的参数。

② 额外的参数的个数与类型未指定。

函数的声明与函数的用法将在第9章详细介绍。变长参数表用**stdarg.h**或**varargs.h**头文件中的工具访问。

参考章节 抽象声明符 5.12；数组声明符 4.5.3；定义声明与引用声明 4.8；函数类型与函数声明 5.8；函数定义 9.1；指针声明符 4.5；**stdarg.h**与**varargs.h** 11.4

4.5.5 复合声明符

复合声明符可以构成更复杂的类型，如"指向返回整数的函数的5个元素的指针数组"，在这个声明中为**ary**类型：

```
int (*ary[5])();
```

101

声明符的惟一限制是结果类型应在C语言中有效。在C语言中无效的类型包括：

1. 任何包括**void**的类型，除了"返回**void**的函数"或（标准C语言中）"**void**的指针"。

2. “……的函数的数组”，数组可以包含函数指针，但不包含函数本身。

3. “返回……数组的函数”，函数可以返回数组指针，但不返回数组本身。

4. “返回……的函数的函数”，函数可以返回其他函数指针，但不返回函数本身。

复合声明符时，声明符表达式的优先顺序非常重要。函数与数组声明符的优先顺序高于指针声明符，因此“`*x()`”等价于“`*(x())`”（“返回值为……指针的函数”）而不是“`(*x)()`”（“指向返回值为……的函数的指针”）。可以用括号正确地组合声明符。早期C语言编译器的上限为6层声明符嵌套，而标准C语言编译器至少允许12层深度的嵌套。

尽管声明符可以任意复杂，但好的编程风格是将复杂的声明符分解成几个较简单的声明符。

例

声 明	x的类型
`int x();`	返回值为整数的函数
`int (*x)();`	指向返回值为整数的函数的指针
`void (*x)();`	指向不返回结果的函数的指针
`void *x();`	返回值为`void`的指针的函数

□

例 语句

```
int *(*(*(*x)())[10])();
```

可以替换成

```
typedef int *(*print_function_ptr)();
typedef print_function_ptr (*digit_routines)[10];
digit_routines (*x)();
```

变量`x`是函数指针，这个函数的返回值为指向包含10个元素的指针数组的指针，而指针数组中的指针指向返回值为指向整数的指针的函数。 □

例 声明符的语法的合理之处在于它模拟了使用所包括标识符的语法。为了说明声明和使用之间的对称性，如果使用下列声明：

```
int *(*x)[4];
```

则下列表达式类型为`int`：

```
*(*x)[i]
```

□

102

参考章节 数组类型 5.4；函数类型 5.8；指向`void`的指针 5.3.2；指针类型 5.3；`void`类型指定符 5.9

4.6 初始化语句

变量声明可以带有初始化语句用于指定变量在其生存期开始时的值。初始化语句的完整语法如下：

initializer:
    assignment-expression
    { *initializer-list* ,*opt* }

initializer-list:
    initializer

initializer-list , *initializer*
designation initializer　(C99)
initializer-list , *designation initializer*　(C99)

designation :
designator-list =

designator-list :
designator
designator-list designator

designator :
[*constant-expression*]
. *identifier*

花括号中可选的尾部逗号不影响初始化语句的含义。

C99允许指定初始化语句（见4.6.9节），编程人员可以命名要初始化的特定集合的成分。

特定声明允许的初始化语句取决于要初始化的对象类型和声明的对象是静态存储类还是自动存储类。表4-6列出了选项，我们将在后面几节详细介绍。外部对象声明只在定义声明时才能用初始化语句（见4.8节）。

初始化语句的形式（花括号中的初始化语句列表）应与要初始化的变量结构相匹配。语言定义中指定标量变量的初始化语句可选加上花括号，虽然这种花括号在逻辑上是不需要的。我们建议保留花括号，表示集合初始化。集合初始化语句缩写有特殊规则。

103

表4-6　初始化语句形式

存储类	类　型	初始化语句表达式	默认初始化语句
静态	标量	常量	0、0.0、false或null指针
静态	数组①或结构	花括号包含的常量	每个成分递归默认
静态	联合②	常量	第一个成分的默认
自动	标量	任意	无
自动	数组①②	花括号中的常量	无
自动	结构②	花括号中的常量或相同结构类型的单个非常量表达式	无
自动	联合②	常量或相同联合类型的单个非常量表达式	无

① 数组长度可以未知，由初始化语句确定。变长数组不能初始化。
② 适用于标准C语言，较早的C语言实现可能不允许初始化这些对象。

提示　C语言最初的初始化语句语法省略 =运算符，为了保持兼容性，一些当前C语言编译器也接受这种语法。这些编译器的用户在不小心省略声明中的逗号或分号时（如“`int a b;`”），会得到一个关于无效初始化语句的模糊的错误消息。标准C语言不支持这种过时语法。

下面几节介绍每种变量的特殊要求。

参考章节　自动与静态生存期　4.2；声明　4.1；外部对象　4.8；静态存储类　4.3

4.6.1　整数变量初始化语句

整数变量的初始化语句形式如下：

declarator = *expression*

初始化表达式的类型应允许对初始化变量进行简单赋值，适用常用的赋值转换。如果变量为静态存储类或外部存储类，则初始化语句表达式应为常量表达式；如果变量为自动存储类或寄存器存储类，则初始化语句表达式允许为任意表达式。静态整数的默认初始化语句为0。

例 下列代码段中，**Count**用常量表达式初始化，但**ch**用函数调用的结果初始化：

104

```
#include <stdio.h>
static int Count = 4*200;

int main(void)
{
    int ch = getchar();
    ...
}
```

□

参考章节 常量表达式 7.11；整型 5.1；静态与自动生存期 4.2；常用赋值转换 6.3.2

4.6.2 浮点数变量初始化语句

浮点数变量的初始化语句形式如下：

declarator = *expression*

初始化表达式的类型应允许对初始化变量进行简单赋值，适用常用的赋值转换。如果变量为静态存储类或外部存储类，则初始化语句表达式应为常量表达式；如果变量为自动存储类或寄存器存储类，则初始化语句表达式允许为任意表达式。

例

```
static void process_data(double K)
{
    static double epsilon = 1.0e-6;
    auto float fudge_factor = K*epsilon;
    ...
}
```

□

标准C语言显式允许初始化语句中使用浮点数常量表达式。一些较早的C语言编译器不支持复杂的浮点数常量表达式。

静态浮点数变量的默认初始化语句为0.0。这个值在目标计算机上可能不表示位数为0的对象。标准C语言编译器要将变量初始化为0.0的正确表示，但大多数较早的C语言编译器总是将静态存储类初始化为0位。

参考章节 算术类型 第5章；常量表达式 7.11；浮点型常量 2.7.2；浮点型 5.2；静态与自动生存期 4.2；一元负号运算符 7.5.3；普通赋值转换 6.3.2

4.6.3 指针变量初始化语句

指针变量的初始化语句形式如下：

declarator = *expression*

105

初始化表达式的类型应允许对初始化变量进行简单赋值，适用普通的赋值转换。如果变量是自动存储类的，则允许任何合适类型的表达式。

如果变量为静态存储类或外部存储类，则初始化语句表达式应为常量表达式。下列元素可

以构成作为指针类型*PT*（指向*T*的指针）初始化语句的常量表达式。

1. 数值为0的整型常量表达式或这样的数值转换成**viod *类型。这是null指针常量，通常用标准库中的名称**NULL**引用。

```
#define NULL ((void *)0)
double *dp = NULL;
```

2. 类型为“返回*T*的函数”的静态函数名或外部函数名转换成类型为“指向返回*T*的函数的指针”的常量。

```
extern int f();
static int (*fp)() = f;
```

3. 类型为“*T*的数组”的静态数组名或外部数组名转换成类型为“指向*T*的指针”的常量。

```
char ary[100];
char *cp = ary;
```

4. 类型为*T*的静态变量名或外部变量名采用&运算符得到类型为“指向*T*的指针”的常量。

```
static short s; auto short *sp = &s;
```

5. 类型为“*T*的数组”的静态数组或外部数组用常量表达式作为下标，采用&运算符得到类型为“指向*T*的指针”的常量。

```
float PowersOfPi[10];
float *PiSquared = &PowersOfPi[2];
```

6. 整型常量转换成指针类型，得到这个指针类型的常量，但这是不可移植的。

```
long *PSW = (long *) 0xFFFFFFF0;
```

并不是所有编译器都接受常量表达式的转换，但标准C语言中允许。

7. 字符串字面值作为指针类型变量的初值时得到类型为“指向**char**的指针”的常量。

```
char *greeting = "Type <cr> to begin ";
```

8. 第3项到第7项任何表达式与一个整型常量表达式的和或差。

```
static short s;
auto short *sp = &s + 3, *msp = &s - 3;
```

106

一般来说，指针类型的初值要求是整数，这个整数转换为指针类型或地址加（或减）一个整数常量。这个限制影响了大多数连接程序的功能。

静态指针默认初始化为null指针。在极少数情况下不用位为0的对象表示null指针时，标准C语言编译器指定要用正确的null指针值。大多数早期的C语言编译器只是把静态存储类初始化为0位。

参考章节　地址运算符&　7.5.6；数组类型　5.4；涉及指针的转换　6.2.7；函数类型　5.8；整型常量　2.7；指针声明符　4.5；指针类型　5.3；字符串型常量　2.7；普通赋值转换　6.3.2

4.6.4 数组类型变量初始化语句

如果I_i表达式是类型为*T*的对象的可允许的初始化语句，则：

$$\{\ I_0\ ,\ I_1\ ,\ \ldots,\ I_{n-1}\ \}$$

是类型为“*T*的*n*元素数组”的可允许的初始化语句。C99允许I_i为非常量表达式，但早期的C语言版本要求I_i为常量表达式。初值I_i初始化数组的元素*j*（基数为0）。多维数组采用相同模式，初始化语句按行列出（C语言中最后一维下标变化很快）。

例 一维数组初始化时列出各个元素：

```
int ary[4] = { 0, 1, 2, 3 };
```

多维数组初始化时初始化每个子数组：

```
int ary[4][2][3] =
          { { { 0, 1, 2}, { 3, 4, 5} },
            { { 6, 7, 8}, { 9, 10, 11} },
            { {12, 13, 14}, {15, 16, 17} },
            { {18, 19, 20}, {21, 22, 23} } };
```

结构的数组（见4.6.6节）可以用类似方法初始化：

```
struct {int a; float b;} a[3] = { {1, 2.5},
                                  {2, 3.9},
                                  {0, -4.0} };
```

□

静态数组或外部数组总是可以按照这样的方法初始化。标准C语言允许初始化自动数组，但C语言原始的定义中没有这个功能。数组初始化有一些特殊规则：

1. 初值的个数可能少于数组元素个数，这时其余元素初始化为默认初始化值（静态数组中使用的值）。如果初值个数多于元素个数，则产生错误。 107

例 声明

```
float ary[5] = { 1, 2, 3 };
int mat[3][3] = { {1, 2}, {3} };
```

等于声明

```
int ary[5] = { 1.0, 2.0, 3.0, 0.0, 0.0 };
int mat[3][3] = { {1, 2, 0},
                  {3, 0, 0},
                  {0, 0, 0} };
```

□

2. 数组边界不需要指定（例如，不完整数组类型），这时可以从初值长度得到数组边界。静态数组与自动数组初始化都可以这样。

例 声明

```
int squares[] = { 0, 1, 4, 9 };
```

等于声明

```
int squares[4] = { 0, 1, 4, 9 };
```

□

3. 可以用字符串字面值初始化“**char**数组”类型的变量。这时数组第一个元素初始化为字符串中第一个字符，等等。字符串的终止null字符'**\0**'在空间允许时或没有指定数组长度时存放在数组中。字符串可选放在花括号中。字符串太长，超过字符数组指定的长度时不算错误，但可能使读者不理解（在C++中是个错误）。

元素类型与**wchar_t**兼容的数组可以采用同样的方法用宽字符串字面值初始化。

例 声明

```
char x[5]  = "ABCDE";
char str[] = "ABCDE";
wchar_t q[5] = L"A";
```

等于声明

```
char x[5]   = { 'A', 'B', 'C', 'D', 'E' }; /* No '\0'! */
char str[6] = { 'A', 'B', 'C', 'D', 'E', '\0' };
wchar_t q[5]   = { L'A', L'\0', L'\0', L'\0', L'\0' };
```
□

108 4. 可以用字符串列表初始化字符指针的数组。

例 `char *astr[] = { "John", "Bill", "Susan", "Mary" };` □

5. 变长数组不能初始化。

参考章节 数组类型 5.4；字符型常量 2.7；字符类型 5.1.3；指针类型 5.3；字符串型常量 2.7；变长数组 5.4.5；宽字符串 2.7.4

4.6.5 枚举类型变量初始化语句

枚举类型变量的初始化语句形式如下：

declarator = *expression*

初始化表达式的类型应允许对初始化变量进行简单赋值，适用常用的赋值转换。如果变量为静态存储类或外部存储类，则初始化语句表达式应为常量表达式；如果变量为自动存储类或寄存器存储类，则初始化语句表达式允许为任意表达式。

例 好的编程风格应该是初始化表达式类型与初始化的变量为相同的枚举类型。例如：

```
static enum E { a, b, c } x = a;
auto enum E y = x;
```
□

参考章节 转换表达式 7.5.1；常量表达式 7.11；枚举类型 5.5；普通赋值转换 6.3.2

4.6.6 结构类型变量初始化语句

如果结构类型T有n个命名成员，类型为T_j，j=1，…n，如果I_j表达式是类型为T_j的对象的可允许的初始化语句，则：

$$\{ I_1 , I_2 , \ldots, I_n \}$$

是类型为T的可允许的初始化语句。未命名位字段的成员不参与初始化。C99允许I_j为非常量表达式，但早期的C语言版本要求I_j为常量。

例

```
struct S {int a; char b[5]; double c; };
struct S x = { 1, "abcd", 45.0 };
```
□

所有C语言编译器都能够初始化结构类型的静态变量与外部变量；结构类型的自动变量与寄
109 存器变量可以在标准C语言中初始化，两种形式都可以使用。第一，可以像静态变量一样使用花括号内的常量表达式列表。第二，可以用下列形式进行初始化：

declarator = *expression*

其中*expression*与初始化的变量具有相同类型。一些较早的C语言编译器存在缺陷，不允许初始化包含位字段的结构。

和数组初始化语句一样，结构初始化语句有一些特殊缩写规则。特别地，初值的个数可能少于结构成员个数，这时其余成员初始化为各自的默认初始化值。如果初值个数多于成员个数，则产生错误。

例 对于结构声明

```
struct S1 {int a;
           struct S2 {double b;
                      char c; } b;
           int c[4]; };
```

初始化形式

```
struct S1 x = { 1, {4.5} };
```

等效于初始化形式

```
struct S1 x = { 1, { 4.5, '\0' }, { 0, 0, 0, 0 } };
```

□

参考章节 位字段 5.6.5；常量表达式 7.11；结构类型 5.6

4.6.7 联合变量初始化语句

标准C语言允许初始化联合变量（而传统C语言则不允许）。静态联合变量、外部联合变量、自动联合变量或寄存器联合变量的初始化语句都要把常量表达式放在花括号中，作为联合中第一个成员类型对象的初始化语句。自动联合类型与寄存器联合类型的初始化语句也可以是相同联合类型的单个表达式。C99中，可以用指定符初始化第一个成员以后的成员。

例 下面是联合变量**x**与**y**的两种初始化语句形式：

```
enum Greek { alpha, beta, gamma };
union U {
          struct { enum Greek tag; int Size; } I;
          struct { enum Greek tag; float Size; } F;
        };
static union U x = {{ alpha, 42 }};
auto union U y = x;
```

□ 110

余下的C语言类型是函数类型和**void**类型。由于不能声明这些类型的变量，因此不存在初始化问题。

参考章节 指定初值 4.6.9；静态生存期 4.2；联合类型 5.7

4.6.8 省略花括号

C语言允许在某些情况下省略初值列表中的花括号，但为了清晰起见，通常要保留花括号。一般规则如下：

1. 初始化数组或结构类型的变量时，不能删除最外层花括号。
2. 除非初值列表包含要初始化的对象的正确元素个数，则可以删除花括号。

例 这些规则最常见的用法是在初始化多维数组时删除内层花括号：

```
int matrix[2][3] = { 1, 2, 3,   4, 5, 6 };
      /* same as: { {1, 2, 3}, {4, 5, 6} } */
```

□

有些C语言编译器处理初值列表时有时会允许过多或过少的花括号。我们建议尽量让初始化语句保持简单，并用花括号明确表示结构。

4.6.9 指定初值

C99允许在初值列表中指定要初始化的集合（结构、联合或数组）成分。指定初值与位置（非指定）初值可以在同一初值列表中混合。

在数组初值列表中，指定符的形式为[e]，其中常量表达式e用索引指定数组元素。如果数组长度没有指定，则允许任何非负索引，显式的初始化索引中最大的一个确定数组的最终长度。如果指定初值后面是个位置初值，则这个位置初值开始初始化指定元素后面的成分。这种形式中，列表中后面的值可能覆盖前面的值。

例　下列每个初始化之后都有个说明，提供所有元素得到的初值。

```
int a1[5] = { [2]=100, [1]=3 };
   /* {0, 3, 100, 0, 0} */
int a2[5] = { [0]=10, [2]=-2, -1, -3 };
   /* {10, 0, -2, -1, -3} */
int a3[] = { 1, 2, 3, [2]=5, 6, 7};
```

111

```
   /* {1, 2, 5, 6, 7} ; a3 has length 5 */
```

□

在结构的初值列表中，指定符的形式为.c，其中c是结构中的成员名。如果指定初值后面是个位置初值，则这个位置初值开始初始化指定元素后面的成分。这种形式中，列表中后面的值可能覆盖前面的值。

例　下列每个初始化之后都有个说明，提供所有元素得到的初值。

```
struct S {int a; float b; char c[4]; };
struct S s1 = { .c = "abc" };
   /* {0, 0.0, "abc"} */
struct S s2 = { 13, 3.3, "xxx", .b=4.5 };
   /* {13, 4.5, "xxx"} */
struct S s3 = { .c = {'a','b','c','\0'} };
   /* {0, 0.0, "abc"} */
```

□

在联合初值列表中，指定符的形式为.c，其中c是联合中的成员名。这样就可以通过联合的任一成员将其初始化，而不是只能通过第一个成员初始化。

例　下列每个初始化之后都有说明，提供所有元素得到的初值。

```
union U {int a; float b; char c[4]; };
union U u1 = { .c = "abc" };
   /* u1.c is "abc\0"; other components undefined */
union U u2 = { .a = 15 };
   /* u2.a is 15; other components undefined */
union U u3 = { .b = 3.14 };
   /* u3.b is 3.14; other components undefined  */
```

□

嵌套集合可以用指示符按相应方式初始化。指示符可以接合，初始化更深层嵌套的元素。

例　下列每个初始化之后都有说明，提供所有元素得到的初值。

```
struct Point {int x; int y; int z; };
typedef struct Point PointVector[4];
PointVector pv1 = {
    [0].x = 1, [0].y = 2, [0].z = 3,
    [1] = {.x = 11, .y=12, .z=13},
    [3] = {.y=3} };
```

112

```
    /* {{1,2,3},{11,12,13},{0,0,0},{0,3,0}} */
    typedef int Vector[3];
    typedef int Matrix[3][3];
```

```
struct Trio {Vector v; Matrix m; };
struct Trio t = {
    .m={[0][0]=1, [1][1]=1, [2][2]=1},
    .v={[1]=42,43} };
    /* {{0,42,43},{{1,0,0},{0,1,0},{0,0,1}}} */
```
□

4.7 隐式声明

在C99之前，函数调用中使用的外部函数不需要事先声明。如果编译器发现标识符*id*后面是个左括号，而这个*id*没有事先声明，则最内层范围中隐式增加下列形式的声明：

```
extern int id();
```

C99实现在*id*没有事先声明为函数时会发出诊断，但通过隐式声明的方法可以继续工作。一些非标准实现可能在顶层而不是最内层声明标识符。

例 让函数默认声明不是良好的编程风格，可能造成错误，特别是不正确的返回类型。如果**malloc**（见16.1节）之类的指针返回函数可以隐式声明如下：

```
extern int malloc();
```

而不是用下列正确声明：

```
extern char *malloc();  /* returns (void *) in Standard C */
```

则调用**malloc**可能在类型**int**与**char ***的表示方法不相同时无法工作。假设**int**类型占用两个字节，指针类型占用4个字节，则当编译器遇到下列代码时：

```
int *p;
…
p = (int *) malloc(sizeof(int));
```

产生的代码将**malloc**返回的两字节值扩展为指针所要的四字节。结果只把**malloc**返回的地址低半部分赋予**p**，当分配到足够存储空间之后，**malloc**返回的地址大于**0xFFFF**，程序失败。□

4.8 外部名称

外部名称的一个重要问题是保证几个文件中同一外部名称的声明之间的一致性。例如，如果同一外部变量的两个声明指定不同初始化，会发生什么情形呢？由于类似原因，一定要区别一组文件 113
中一个外部名称的*定义声明*。然后同一名称的其他声明被看作是*引用声明*，即引用这个定义声明。

C语言中一个著名的缺陷是外部变量的定义声明与引用声明很难区别。一般来说，编译器用4个模型之一确定哪个顶层声明是定义声明。

4.8.1 初始化语句模型

顶层声明中存在初始化语句时，表示这个声明是定义声明时，其他声明是引用声明。C语言程序的所有文件之中只能有一个定义声明。这是标准C语言采用的模型，还有一个附加规则将在下节讨论。

4.8.2 省略存储类模型

在这个模型中，所有引用声明要显式地包括存储类**extern**，而每个外部变量的惟一定义声明中省略存储类。定义声明可以包括初始化语句，但不是必需的。一个声明中不能既有初始化语句又有存储类**extern**。

在标准C语言中，没有初始化语句或存储类的顶层声明称为*试探性定义*，即先将其作为引用

声明．但如果同一变量中没有其他声明有初始化语句，则把试探性定义变成真正的定义。

在C++中，有初始化语句时，忽略**extern**存储类。

4.8.3 公用模型

这种模型之所以称为“公用模型”，是因为它与FORTRAN编程语言中的**COMMON**块有关，**COMMON**块的多个引用合并成一个定义声明。无论是显式指定还是默认指定，定义和引用外部声明的存储类都是**extern**。构成程序的所有对象文件中每个外部名有许多声明，但只有一个声明具有初始化语句。连接时，同一标识符（在所有C语言目标文件中）的所有外部声明合并起来，产生一个定义声明，不一定与任何特定文件相关联。如果某一声明指定初值，则这个初值用于初始化数据对象（如果多个声明指定初值，则结果是未确定的）。

114 这个方案对编程人员最方便，对系统软件要求最高。

4.8.4 混合公用模型

这个模型介于公用模型和省略存储类模型之间，在许多UNIX版本中使用。

1. 如果省略**extern**且具有初始化语句，则发出符号的定义。如果构成程序的所有文件中有多个这种定义，则会在连接时或连接之前出错。
2. 如果省略**extern**而没有初始化语句，则发出类似于FORTRAN中**COMMON**样式的定义。同一标识符可以有多个这种定义共存。
3. 如果有**extern**，则声明是对其他地方所定义名称的引用，这种声明不能有初始化语句。

如果没有给外部变量提供显式的初始化语句，则将初始化值视为整数常量0对变量进行初始化。

4.8.5 总结与建议

表4-7显示了不同外部引用模型的顶层声明。为了维护与大多数编译器的兼容性，建议采用下列规则：

1. 每个外部变量有一个定义点（在源文件中），定义声明时，省略存储类**extern**，但要包含初始化语句：

```
int errcnt = 0;
```

2. 在引用其他地方定义的外部变量的每个源文件或头文件中，使用存储类**extern**而不包含初始化语句：

```
extern int errcnt;
```

表4-7　顶层声明解释

顶层声明	初始化语句模型	省略存储类模型（和C++）	公用模型	混合公用模型	标准C语言模型
int x;	引用	定义	定义或引用	定义或引用	引用①
int x = 0;	定义	定义	定义	定义	定义
extern int x;	引用	引用	定义或引用	引用	引用
extern int x = 0;	定义	（无效）	定义	（无效）	定义

115 ① 如果文件中没再遇到定义声明，则这就成为定义声明。

除了定义/引用的差别外，外部名一定要在构成程序的所有文件中声明为相同类型。C语言编译器无法验证不同文件中的声明是否一致，不一致行为会在运行时产生错误。UNIX系统C语言编译器通常提供一个**lint**程序，可以检查多个文件的声明是否一致，还有一些适用于UNIX

系统与Windows系统的进行一致性声明检查的商业化产品。

4.8.6 未引用的外部声明

尽管C语言没有要求，但习惯上忽略从来不引用的外部变量声明和函数声明。例如，如果程序中出现声明“**extern double fft();**”，而从未使用过函数**fft**，则不向连接程序传递名称为**fft**的外部引用。因此，函数**fft**不会装入程序中，避免无谓地占用空间。

4.9 C++兼容性

4.9.1 作用域

C++中**struct**定义与**union**定义是作用域，即类型声明只出现在这些定义中，在定义之外不可见，而标准C语言中则可以在这些作用域外有效（5.6.3节）。为了保持兼容，只要把类型声明移到结构以外（有些C++实现方法可以在不发生歧义时允许这种错误）。

例 下列代码中，在结构**s**中定义结构**t**，但在结构之外引用，这在C++中是无效的。

```
struct s {
    struct t {int a; int b;} f1; /* define t here */
} x1;
struct t x2; /* use t here; OK in C, not in C++ */
```

□

参考章节 作用域 4.2.1；结构成员 5.6.3

4.9.2 标志名称与typedef名称

结构标志名称与联合标志名称不能作为**typedef**名称，除非具有相同标志类型。在C++中，标志名称隐式声明为**typedef**名称和标志（但后续同一作用域中的同名变量声明或函数声明可以将其隐藏）。这可能造成诊断，偶尔也会引起不同行为。

例 下面的一些例子在C语言或C++中引起诊断。

116

```
typedef struct n1 {…} n1;          /* OK in both C and C++ */
struct n2 {…}; typedef double n2; /* OK in C, not in C++ */
struct n3 {…}; n3 x; /* OK in C++, not in C */
```

但是，标志名称也可以作为变量名或函数名面不造成混淆。下列声明序列在C语言和C++中都是可以接受的，但最好避免一些容易引起混淆的声明：

```
struct n4 {…};
int n4;
struct n4 x;
```

C++中内部作用域中的**struct**标志声明可以隐藏外部作用域中的变量声明，这样就可能使C语言程序的含义发生改变而不发出警告。下列代码中，表达式**sizeof(ary)**引用C语言中的数组长度，而在C++中引用的是**struct**类型的长度：

```
int ary[10];
…
void f(int x)
{
    struct ary { … };/* In C++, this hides previous ary */
    …
    x = sizeof(ary);  /* Different meanings in C and C++! */
}
```

□

关于C++中**typedef**重新定义的兼容性，见5.13.2节。

参考章节　命名空间　4.2.4；**typedef**重新定义　5.10.2

4.9.3 类型的存储类说明符

不要在类型声明中放置存储类说明符。这样的操作在传统C语言中是被忽略的，而在C++和标准C语言中是无效的。

例
```
static struct s {int a; int b;};   /*无效*/
```
□

参考章节　类型的存储类　4.4.2

4.9.4 const类型限定符

具有**const**类型限定符而没有显式存储类的顶层声明在C++中被认为是**static**存储类，而在C语言中被认为是**extern**存储类。为了保持兼容，应检查顶层**const**声明，提供显式的存储类。

在C++中，字符串型常量隐式地限定为**const**，而C语言中则不然。

例　下列声明在C语言和C++中具有不同含义：

117
```
const int c1 = 10;
```
但下列声明在C语言和C++中具有相同含义：
```
static const int c2 = 11;
extern const int c3 = 12;
```
除了引用外部定义常量的**const**声明外，其他**const**声明均在C++中都要有初始化语句。□

参考章节　**const**类型限定符　4.4.4

4.9.5 初始化语句

在C++中，用字符串字面值初始化定长字符数组（或用宽字符串直接数初始化**wchar_t**数组）时，数组中要有整个字符串的足够空间，包括终止null字符。

例
```
char str[5] = "abcde"; /* valid in C, not in C++ */
char str[6] = "abcde"; /* valid in both C and C++ */
```
□

4.9.6 隐式声明

C++和C99中不允许函数隐式声明（见4.7节）。所有函数都要先声明后使用。

参考章节　隐式声明　4.7

4.9.7 定义声明与引用声明

在C++中，顶层变量没有试探性定义。C语言中的试探性定义在C++中被看成实际定义，即下列声明序列在标准C语言中有效，而在C++中会造成重复定义错误：
```
int i;
…
int i;
```
例　不能生成相互递归的静态初始化变量，这个规则也适用于静态变量。
```
struct cell {int val; struct cell *next;};
static struct cell a; /* tentative declaration */
static struct cell b = {0, &a};
```

```
static struct cell a = {1, &b};
```

118

对于全局变量，这不成问题：第一个**static**可以换成**extern**，第二个和第三个**static**可以删除（C++中可以生成相互递归的静态初始化变量。但与C语言不兼容）。 □

参考章节 结构类型引用 5.6.1；试探性定义 4.8.2

4.9.8 函数连接

从C++中调用C语言函数时，函数要声明具有“C”连接，详见第10章介绍。

例 如果C++程序中要调用C实现编译的函数**f**，则要将C++声明写成：

```
/* This is a C++ program. */
extern "C" int f(void);/* f is a C, not C++, function */
```

□

4.9.9 无参函数

在C++中，声明带有空参数表的函数被视为无参函数，而C语言中这种函数表示未指定参数，即C++中声明**int f()**等价于C语言中声明**int f(void)**。

4.10 练习

1. 下面是静态函数**P**的定义，如果前面没有调用**P**，则**P(6)**的值为多少？如果是第二次调用，**P(6)**的值又应为多少？

```
static int P(int x)
{
    int i = 0;
    i = i+1;
    return i*x;
}
```

2. 下列程序段显示了包含名称**f**的不同声明的一个块。这些声明是否冲突？如果冲突，请删除声明，直至程序有效，尽可能多地保留不同的声明。

```
{
    extern double f();
    int f;
    typedef int f;
    struct f {int f,g;};
    union f {int x,y;};
    enum {f,b,s};
    f: ...
}
```

119

3. 下列程序段声明3个变量**i**，类型分别为**int**、**long**与**float**。指出每个变量分别在哪些行进行了声明和使用？

```
1 int i;
2 void f(i)
3     long i;
4 {
5     long l = i;
6     {
7         float i;
8         i = 3.4;
```

```
 9      }
10      l = i+2;
11  }
12  int *p = &i;
```

4. 编写表示下列内容的C语言声明，要求函数声明使用原型。

（a）P是一个外部函数，没有参数，不返回结果。

（b）i是一个本地整型变量，被大量使用，应优化速度。

（c）LT是“字符指针”类型的同义词。

（d）Q是一个外部函数，带两个参数，不返回结果。第一个参数i是整数，第二个参数cp是字符串。字符串不允许被修改。

（e）R是一个外部函数，只有一个参数p为函数指针，指向带有一个32位整数参数i，且返回指向double类型数值的指针的函数。R返回一个整数值。假设类型long为32位宽。

（f）STR是一个静态未初始化字符串，可修改，最多保存10个字符，不包括终止null符。

（g）STR2是一个字符串，初始化为字符串字面值，即INIT_STR2宏的值。初始化之后，字符串不能被修改。

（h）IP是指向整数的指针，初始化为变量i的地址。

5. 矩阵m的声明为int m[3][3];第一个下标指定行号，第二个下标指定列号。写出一个m的初始化语句f，使矩阵的第一列值全为1，第二列值全为2，第三列值全为3。

6. 对下列声明：

```
const int * cip;
int * const cpi;
int i;
int * ip;
```

允许下面哪些赋值？

```
(a)  cip = ip;
(b)  cpi = ip;
(c)  *cip = i;
(d)  *cpi = i;
```

7. 使用C99指定初值，编写3×3的int元素矩阵identity的声明和初始化语句。这个初始化语句对元素identity[1][1]、identity[2][2]与identity[3][3]赋值1，其余所有元素赋值为0。

120

8. 对两个结构标志为left与right的结构编写C语言声明。left结构包含一个double字段data和一个指向right结构的指针link。right结构包含一个int字段data和一个指向left结构的指针link。

9. 刚刚购买C99编译器，要用它重新编译现有软件。软件在较早的C89编译器中能够顺利编译，但C99编译器中报告一些问题。对下列报告的错误，指出可能产生错误的原因。

(a) C99编译器拒绝一个函数调用，称这个函数未定义。

(b) C99编译器拒绝本地声明register i;。

10. 在C语言程序中，假设fm定义为函数式宏，om定义为对象式宏（3.3节）。如果程序还包含下列本地变量声明：

```
int fm;
int om;
```

121

这些声明与宏之间会不会有任何冲突？讨论编译程序时会发生什么情况。

第5章 类　型

类型是一组数值和对这些数值的一组操作。例如，整型数值包括一些指定范围的整数和对这些数值的一组操作，包括加、减、不等性测试等等。浮点型数值包括与整数表示形式不同的数值，以及一组不同的操作：浮点数加、减、不等性测试等等。

我们说变量或表达式类型为T是指它的值被限制在T域。变量类型是通过变量声明建立的，表达式类型由表达式运算符定义指定。C语言提供了大量内部类型，包括几种类型的整数、浮点数、指针、枚举、数组、结构、联合和函数。

可以把C语言类型组织成表5-1所示的类别。整型包括所有形式的整数、字符和枚举。算术类型包括整型和浮点型。标量类型包括算术类型与指针类型。函数类型是返回数值的函数。组合类型包括数组与结构。联合类型是用**union**类型说明符生成的。**void**类型没有数值也没有操作。

_Bool、**_Complex**与**_Imaginary**类型是C99中新增加的。布尔类型（**_Bool**）是无符号整数类型，而6个复数类型是浮点型。C99还进一步把算术类型分为域：6个复数类型在复数域中，所有其他算术类型在实数域中，是实数类型。

本章介绍C语言的所有类型。对于每种类型，我们指出这种类型的对象如何声明，这种类型的数值范围和有关这种类型的长度与表示的任何限制，以及对这种类型定义了哪些操作。

参考章节 数组类型 5.4；布尔类型 5.15；字符类型 5.13；复数类型 5.21；声明 4.1；枚举类型 5.5；浮点型 5.2；函数类型 5.8；整型 5.1；指针类型 5.3；结构类型 5.6；联合类型 5.7；**void**类型 5.9

表5-1 C语言类型与类别

C语言类型	类型分类		
short int long long long（带符号与无符号）	整型	算术类型①	标量类型
char（带符号与无符号）			
_Bool ②			
enum {...}			
float, double, long double	浮点型		
float _Complex, double _Complex, long double _Complex, float _Imaginary, double _Imaginary, long double _Imaginary ②			
T *	指针类型		
T [...]	数组类型		组合类型
struct {...}	结构类型		
union {...}	联合类型		
T (...)	函数类型		
void	void类型		

① 除了复数类型之外的所有其他算术类型又被归类为实数类型。

② C99中新增的，**_Imaginary**是可选的。

5.1 整数类型

C语言与大多数编程语言相比，提供更多整数类型和运算符。这种不同反映了大多数计算机的字长和算术运算符种类的不同，这样就允许C语言程序与底层硬件之间保持密切的一致性。C语言整数类型可以用来表示：

1. 带符号或无符号整数值，可以进行通常的算术运算与关系型运算。
2. 位向量，具有非、与、或、异或和左移与右移运算。
3. 布尔值，0表示false，所有非零值表示true，整数1为标准的true值。
4. 字符，用计算机上的整数编码表示。

枚举类型也是整型或整数式类型，将在5.5节介绍。

标准C语言要求实现使用整数的二进制编码，因为许多低级C语言运算只有用二进制表示方
124 式才能在计算机上以可移植的方式描述。

可以把整数类型分为4类：带符号类型、无符号类型、布尔类型和字符型。每一类有特定类型说明符集用以声明这个类型的对象：

integer-type-specifier:
　　signed-type-specifier
　　unsigned-type-specifier
　　character-type-specifier
　　bool-type-specifier　　*(C99)*

5.1.1 带符号整数类型

C语言向编程人员提供了4种标准带符号整数类型，类型指定符按字长的递增顺序分别为**short**、**int**、**long**与**long long**。类型**signed char**是第5种带符号整数类型，放在5.1.3节介绍。C99引入了**long long**类型和扩展整数类型（见5.1.4节）。

每个类型可以用几种等价方式命名，下列语法显示了4种带符号整数类型的等价名称：

signed-type-specifier:
　　short 或 **short int** 或 **signed short** 或 **signed short int**
　　int 或 **signed int** 或 **signed**
　　long 或 **long int** 或 **signed long** 或 **signed long int**
　　long long 或 **long long int** 或 **signed long long** 或
　　　signed long long int

关键字**signed**是C89中新增的，为了与早期的C语言兼容，可以将其省略。**signed**惟一可能影响程序含义的时候是和类型**char**以及位字段连在一起在结构中使用时，这时带符号整数与“普通”整数（即不带**signed**的整数）之间存在差别。

标准C语言指定了大多数整数类型的最小精度。类型**char**至少应为8位，类型**short**至少应为16位，类型**long**至少应为32位，类型**long long**至少应为64位（即C99要求有64位整数类型和完整64位算术运算集）。整数类型的实际范围记录在**limits.h**文件中。

带符号整数类型所表示数值的精度范围不仅取决于表示时所用位数，还取决于使用的编码方法。到目前为止，最常使用的整数二进制编码方法是*对二的补码表示法*（twos-complement-notation），用n位表示的带符号整数的数值范围为-2^{n-1}~$2^{n-1}-1$，编码方式如下：

1. 字的高位（最左边）是符号位。如果符号位为1，则这个数为负数，否则为正数。

2. 正数采用正常二进制编码序列：

$0 = 000...0000_2$

$1 = 000...0001_2$ 125

$2 = 000...0010_2$

$3 = 000...0011_2$

$4 = 000...0100_2$

...

n位字中省略符号位之后有$n-1$位表示正整数，可以表示$0 \sim 2^{n-1}-1$的整数。

3. 要取得负整数，将字中的所有位取反，然后在结果中加1。这样，要得到整数 -1，从1（$00...0001_2$）开始，对每一位取反（$11...1110_2$），然后加1（$11...1111_2 = -1$）。

4. 最大负数$10...0000_2$或-2^{n-1}没有对应的正整数，这个数的反数还是这个数。

另外两种二进制整数编码方式其一是对一的补码表示法（ones-complement-notation），只是求字中所有位的反数；其二是带符号数表示法（sign magnitude notation），对于负数只是对其符号位求反。这些方法的取值范围为$-(2^{n-1}-1) \sim 2^{n-1}-1$，比对二的补码表示法少一个负数，有两个表示0的数（正0和负0）。这3种表示法表示正整数的方法完全一致，并且3种表示法在标准C语言中都可以接受。

标准C语言要求实现在**limits.h**头文件中记录整数类型的实际范围，并指定了所有符合ISO标准的实现中每种整数类型的最大可表示范围。表5-2定义了**limits.h**中必须定义的符号。实现可以将其换成自己的值，但绝对值不能小于表中所示的值，而且应有相同正负号。因此，符合ISO标准的实现无法用8位表示**int**类型，严格符合的C语言程序也不能用**short**类型表示数值 -32 768（为了适应对一的补码表示法表示二进制整数的计算机）。使用非ISO实现的编程人员可以针对自己的实现生成**limits.h**文件。这里记录的范围与使用填充位之后得到的类型长度不一定相同（见6.1.6节）。

C89增补1增加了符号**WCHAR_MAX**与**WCHAR_MIN**，用来表示**wchar_t**类型的最大值与最小值。但是，这些符号是在**wchar.h**头文件中定义的，而没有在**limits.h**中定义。C99增加了新的头文件**stdint.h**，包含了其他整数类型的范围。

例 下面是带符号整数的一些典型声明例子：

```
short i, j;
long int l;
static signed int k;
```

为了尽量保证程序的可移植性，最好不要用**int**表示 -32 767 ~ 32 767以外的整数。如果这个范围不够，可以采用**long**类型。一个好的编程风格是根据每个特定程序的需要用**typedef**定义特殊的整数类型。例如： 126

```
/* invdef.h Inventory definitions for the XXX computer. */
typedef short part_number;
typedef int   order_quantity;
typedef long  purchase_order;
```

C99中的最佳方案是使用某个扩展整数类型名，说明需要的精度。

```
/* invdef.h Inventory definitions for the XXX computer. */
#include <stdint.h>
typedef uint_least64_t part_number;      // at least 64 bits
```

```
typedef int_fast32_t    order_quantity;  // fast and 32 bits
typedef int32_t         purchase_order;  // exactly 32 bits
```

□

表5-2 limits.h中定义的值

名 称	最 小 值	含 义
CHAR_BIT	8	char类型的宽度，用位数表示
SCHAR_MIN	$-(2^7-1)$; -127	signed char的最小值
SCHAR_MAX	2^7-1; 127	signed char的最大值
UCHAR_MAX	2^8-1; 255①	unsigned char的最大值
SHRT_MIN	$-(2^{15}-1)$; -32 767	short int的最小值
SHRT_MAX	$2^{15}-1$; 32 767	short int的最大值
USHRT_MAX	$2^{15}-1$; 65 535	unsigned short的最大值
INT_MIN	$-(2^{15}-1)$; -32 767	int的最小值
INT_MAX	$2^{15}-1$; 32 767	int的最大值
UINT_MAX	$2^{16}-1$; 65 535	unsigned int的最大值
LONG_MIN	$-(2^{31}-1)$; -2 147 483 647	long int的最小值
LONG_MAX	$2^{31}-1$; 2 147 483 647	long int的最大值
ULONG_MAX	$2^{32}-1$; 4 294 967 295	unsigned long的最大值
LLONG_MIN	$-(2^{63}-1)$; -9 223 372 036 854 775 807	long long int的最小值
LLONG_MAX	$2^{63}-1$; +9 223 372 036 854 775 807	long long int的最大值
ULLONG_MAX	$2^{64}-1$; 18 446 744 073 709 551 615	unsigned long long的最大值
CHAR_MIN	SCHAR_MIN或0②	char的最小值
CHAR_MAX	SCHAR_MAX或UCHAR_MAX③	char的最大值
MB_LEN_MAX	1	任何支持区域设置的多字节字符的最大字节数

① UCHAR_MAX必须是$2^{CHAR_BIT}-1$。

② 如果char类型默认为带符号，则为SCHAR_MIN，否则为0。

③ 如果char类型默认为带符号，则为SCHAR_MAX，否则为UCHAR_MAX。

例 C语言中可以用任何整数类型表示布尔值，0表示false，所有非零表示true。布尔表达式为假时值为0，否则值为1。例如，`i = (a<b)`在a小于b时对整型变量i赋值1，在a不小于b时对变量i赋值0。同样，下列语句：

127

```
if (i) statement1; /* Do this if i is nonzero */
else statement2;   /* Do this if i is zero */
```

在i为非0（真）时执行$statement_1$，在i为0时执行$statement_2$。

C99引入了真正的布尔类型_Bool，还引入头文件stdbool.h，其中定义了更方便的类型名bool和布尔值true与false。这些名称与C++中的布尔类型一致，但与C语言中传统上使用的宏名FALSE与TRUE不同：

```
#include <stdbool.h>;
bool b;
```

```
...
b = (x < y) && (y < z);
if (b) ...
```

无法访问C99的编程人员可以方便地定义**bool**、**true**与**false**。 □

参考章节 结构中的位字段 5.6.5；**_Bool**类型 5.1.4；声明 4.1；扩展整数类型 5.1.4；整数常量 2.7.1；**signed**类型说明符 5.1.1；**stdbool.h** 11.3；**stdbool.h** 第21章；类型转换 第6章；**typedef** 5.10

5.1.2 无符号整数类型

对每个带符号整数类型，都有相应的无符号类型，占用相同的存储空间，但具有不同的整数编码。无符号类型通过在对应的带符号类型说明符前面加上**unsigned**关键字设定（代替**signed**关键字，如果有的话）。

unsigned-type-specifier（无符号类型说明符）：

unsigned short int$_{opt}$
unsigned int$_{opt}$
unsigned long int$_{opt}$
unsigned long long int$_{opt}$ *(C99)*

每种情况下，关键字**int**是可选的，但建议加上。选择无符号类型时包括和基于带符号整型相同的考虑。C99引入类型**unsigned long long int**与**_Bool**，都是无符号型，我们将分别介绍。

所有无符号类型都使用直接的二进制方法，不管带符号类型使用对二的补码表示法、对一的补码表示法还是带符号数表示法。符号位当作普通数据位处理。因此，n位字可以表示$0 \sim 2^n - 1$的整数。大多数计算机很容易用带符号表示法或无符号表示法解释字中的值。例如，使用对二的补码表示法时，位模式$11...1111_2$（长度为n位）可以表示 -1（带符号表示法）或$2^n - 1$（无符号表示法）。$0 \sim 2^{n-1} - 1$的整数在带符号表示法或无符号表示法中具有相同的表示。标准C语言 128
实现中整数类型的具体范围都记录在头文件**limits.h**中。

整数是否采用带符号表示法会影响对其进行的运算。所有无符号整数的算术运算都是根据算术模2^n的求模规则进行的。例如，在最大无符号整数上加1得到0，这种溢出已经严格定义了。

带符号整数与无符号整数混合运算的表达式强制采用无符号运算。6.3.4节会介绍要进行的转换，第7章将介绍参数为无符号时每个运算符的作用。

例 这些转换可能令人吃惊。例如，由于无符号整数总是非负数，因此有人会认为下列测试应该总为真：

```
unsigned int u;
...
if (u > -1) ...
```

但它的结果却总是假！（带符号）-1要转换成无符号整数之后再进行比较，得到最大无符号整数，因此u的值不可能大于该整数。 □

C语言原始的定义只提供一个无符号类型**unsigned**。大多数非标准C语言实现都提供了完整范围的无符号类型。

参考章节 **_Bool**类型 5.1.5；整数转换 6.2.3；常量 2.7；**limits.h** 5.1.1；带符号类型 5.1.1

5.1.3 字符类型

C语言中的字符类型是整型，也就是说，字符类型的值是整数而且可以在整型表达式中使用：

character-type-specifier (字符类型符):
```
char
signed char
unsigned char
```

字符类型有3种形式：带符号字符类型、无符号字符类型和普通字符类型。每种形式占用相同的存储空间，但可能表示不同的值。带符号字符类型表示法和无符号字符类型表示法与带符号整数类型表示法和无符号整数类型表示法相同。普通字符类型在类型说明符中没有**signed**与**unsigned**关键字。**signed**关键字是标准C语言新增的，因此在不认识这个关键字的C语言实现中只有两种字符类型：无符号字符类型和普通字符类型。C语言中用字符数组表示“字符串”。

129

例 下面是涉及字符的一些典型声明。

```
static char greeting[7];        /* a 7-character string */
char *prompt; /* a pointer to a character */
char padding_character = '\0'; /* a single character */
```
□

字符类型的表示取决于目标计算机的字符处理设备和字符串处理设备的性质。字符类型有一些特殊特性，使其不同于普通带符号类型和无符号类型。例如，普通**char**类型可能是带符号类型、无符号类型或两种类型的混合。为了提高效率，C语言编译器可以随意用下列两种方式处理**char**类型：

1. **char**类型可以是等价于**signed char**的带符号整型类型。
2. **char**类型可以是等价于**unsigned char**的无符号整型类型。

在一些标准化之前的实现中，**char**类型是个伪无符号整型，即可能只包含非负值，但进行普通一元转换时，如同带符号类型一样处理。

例 如果需要真正的无符号字符类型，则可以指定**unsigned char**类型。如果需要真正的带符号字符类型，则可以指定**signed char**类型。如果**char**类型使用8位对二的补码表示法，并给出下列声明：

```
unsigned char uc = -1;
signed char sc = -1;
char c = -1;
int i = uc, j = sc, k = c;
```

则在所有标准C语言实现中，**i**的值应为255，**j**的值应为 -1。但是，**k**的值是255还是 -1将由具体的C语言实现决定。如果实现不能识别关键字**signed**或不允许**unsigned char**，则会遇到歧义的普通字符的问题。
□

字符的符号性是个重要问题，因为标准I/O库程序通常从文件中返回字符，会在遇到文件结束时返回负值（标准头文件的**EOF**宏中指定这个负值，通常是 -1）。编程人员应把这些函数当作返回**int**类型的值的函数来对待，因为**char**类型可能是无符号的。

例 下列程序要把字符从标准输入流复制到标准输出流，直到从**getchar**返回文件结束标志。前3个定义通常在标准头文件**stdio.h**中提供：

```
extern int getchar(void);
extern void putchar(int);
#define EOF (-1) /* Could be any negative value */
void copy characters(void)
{
    char ch; /* Incorrect! */
    while ((ch = getchar()) != EOF)
        putchar(ch);
}
```

130

但如果char类型是无符号的，则这个函数无法工作。为了演示，我们假设char类型用8位表示，int类型用16位表示，使用对二的补码算术。这样，getchar返回 -1时，赋值语句ch=getchar()对ch赋值255（-1的低8位）。然后循环测试变成255!=-1。如果char类型是无符号的，则通常转换会使-1变成无符号整数类型，得到无符号类型比较255!=65535，求值为true。这样，循环永不终止。只要把ch的声明变成“int ch;”，就可以避免所有问题。 □

例 为了提高可读性，这里可以定义一个“伪字符”类型。例如，下面改写copy_characters，使用新的character类型来表示原来用int类型表示的字符：

```
typedef int character;
…
void copy_characters(void)
{
    character ch;
    while ((ch = getchar()) != EOF)
        putchar(ch);
}
```

□

字符的另一模糊之处是长度。在上面的例子中，我们假设它占用8位，这个假设几乎总是有效，但仍然无法确定它的数值范围是0~255还是-127（或-128）~127。有些计算机可能使用9位或32位，因此编程人员要小心对待。标准C语言要求实现在头文件limits.h中记录字符类型的数值范围。

参考章节 位字段 5.6.5；字符型常量 2.7.3；字符集 2.1；EOF 15.1；getchar 15.6；整数类型 5.1；整数转换 6.2.3；limits.h 5.1.1

5.1.4 扩展整数类型

在C99中，除了标准整数类型外，实现还包括其他扩展整数类型。每个扩展带符号整数类型都有相应的无符号类型。对这些类型选择的关键字要以两个下划线开头或一个下划线加一个大写字母开头（这种标识符是标准C语言保留使用的）。这些扩展类型被认为是整数类型，适用于标准整数类型的所有语句也适用于这些扩展整数类型。扩展整数类型可以通过第21章介绍的stdint.h与inttypes.h C99头文件访问。

131

标准整数转换也适用于扩展整数类型，具体的规则将在第6章讨论转换阶时进行说明。

参考章节 转换阶 6.3.3；带符号整数类型 5.1.1

5.1.5 布尔类型

C99引入了无符号整数类型_Bool，只能保存数值0和1（分别表示“false”与“true”）。虽然在布尔上下文中也可以使用其他整数类型（例如，作为条件表达式中的测试条件），但如果C语言实现支持C99，那么使用_Bool类型更清晰。将任何标量值转换为_Bool类型时，所有非0

值变成1，0值变成0。

头文件**stdbool.h**定义**bool**宏为**_Bool**的同义词，定义**false**与**true**分别为0和1。名称**bool**不是关键字，因为较早的C语言程序中可能还有用户定义类型的名称也为**bool**。涉及**_Bool**类型的转换将与其他整数的转换及其进位一起介绍。

参考章节　整数转换　6.2.3；整数进位　6.3.3；**stdbool.h**　11.3

5.2 浮点数类型

C语言的浮点数有两种：单精度类型和双精度类型，或称为**float**类型和**double**类型。标准C语言增加了**long double**类型，C99又增加了3种复数浮点数类型（见5.2.1节）。非复数浮点数类型也称为实数浮点数类型。

floating-point-type-specifier（浮点数类型说明符）：

float	
double	
long double	*(C89)*
complex-type-specifier	*(C99)*

早期实现中存在类型说明符**long float**与**double**是同义词，但这种类型不太普及，标准C语言中已经将其删除。

例　下面是浮点数类型的一些典型声明：

```
double d;
static double pi;
float coefficients[10];
long double epsilon;
```

□

float、**double**与**long double**的用法与**short**、**int**与**long**的用法相似。在标准C语言之前，实现要求把所有**float**类型的值转换成**double**类型之后再进行运算（见6.3.4节），因此使用**float**类型不一定比使用**double**类型更有效。在标准C语言中，可以直接用**float**类型进行运算，C99中有一组全面支持**float**类型的库函数。

C语言没有指定浮点数类型使用的长度以及不同浮点数类型之间的差别。编程人员可以假设**float**类型中表示的值是**double**类型中表示的值的子集，而**double**类型中表示的值又是**long double**类型中表示的值的子集。一些C语言程序假设**double**类型可以精确表示所有**long**类型的值，即把**long**类型的对象转换为**float**类型的对象，然后再转换回**long**类型的对象，得到的是准确的原始的**long**类型的值。尽管这个假设通常成立，但不能依赖于这个假设。

标准C语言要求在头文件**float.h**中记录实数浮点数类型的特征。表5-3列出了必须定义的符号。名称以**FLT**开头的符号表示**float**类型，名称以**DBL**开头的符号表示**double**类型，名称以**LDBL**开头的符号表示**long double**类型。每种符号显示了允许值，即范围的最小要求和浮点数类型的精度。

大多数算术运算和逻辑运算都适用于浮点数类型操作数，包括算术求反与逻辑求反，加、减、乘、除，关系运算与相等性测试，逻辑AND和逻辑OR，赋值运算以及与所有算术类型的相互换算。

实数浮点数x用带符号数表示，没有隐藏位，可以写成如下形式：

$$x = s \times b^e \times \sum_{k=1}^{p} f_k \times b^{-k}, e_{min} \leqslant e \leqslant e_{max}$$

其中

s 是符号（± 1）

b 是进制基数（通常为2、8、10、16）

e 是取值在e_{min}与e_{max}之间的指数值

p 是b进制的有效位数

f_k 是有效数字，$0 \leqslant f_k < b$

规格化（normalized）浮点数在x为非0时$f_1 > 0$。次规格化（subnormal）浮点数是个非0值，$e = e_{min}$且$f_1 = 0$。非规格化（un-normalized）浮点数是个非0值，$e > e_{min}$且$f_1 = 0$（次规格化浮点数太小，无法规格化；非规格化浮点数是可以规格化的，但由于一些原因而没有规格化）。

浮点数类型可以包括特殊的非浮点数值：无穷大和*NaN*（Not-a-Number）。算术表达式中传递静态*NaN*（quiet NaN）时不发生异常，包含NaN的表达式结果为NaN。当表达中遇到信号*NaN*（signaling NaN）时会产生异常。无穷大与NaN可以有符号，NaN还可以有不同变形。C99扩展了标准库，允许输入和输出这些特殊值，并提供了生成和测试这些值的库函数（见17.13和17.14节）。 133

表5-3 float.h定义的值

名 称	最 小 值	含 义
FLT_RADIX[①]	2	进制基数b
FLT_ROUNDS[①]	无	舍入方式：-1：不确定；0：向0舍入；1：最近舍入；2：向正无穷大舍入；3：向负无穷大舍入[②]
FLT_EVAL_METHOD[③]	无	-1：不确定；0：仍然保持原类型的精度与取值范围；1：float类型与double类型统一用double类型；long double类型保持不变；2：都用long double类型
FLT_EPSILON	10^{-5}	b^{1-p}，最小x，（$x > 0.0$），使$1.0 + x > 1.0$，表中所示为允许的最大值
DBL_EPSILON	10^{-9}	
LDBL_EPSILON	10^{-9}	
FLT_DIG	6	精度小数位数
DBL_DIG	10	
LDBL_DIG	10	
FLT_MANT_DIG	无	p是b进制的有效位数
DBL_MANT_DIG		
LDBL_MANT_DIG		
DECIMAL_DIG[③]	10	表示支持的最宽浮点数类型所需的小数位数，等于$1 + p_{max} \log_{10} b$（b不是10的指数时）
FLT_MIN	10^{-37}	最小规格化正数
DBL_MIN	10^{-37}	

（续）

名　称	最 小 值	含　义
LDBL_MIN	10^{-37}	
FLT_MIN_EXP DBL_MIN_EXP LDBL_MIN_EXP	无	e_{min}，最小负整数 x，使 b^{x-1} 在规格化浮点数类型取值范围内
FLT_MIN_10_EXP	-37	最小x，使10^x在规格化浮点数类型取值范围内
DBL_MIN_10_EXP	-37	
LDBL_MIN_10_EXP	-37	
FLT_MAX	10^{+37}	最大可表示的有限数
DBL_MAX	10^{+37}	
LDBL_MAX	10^{+37}	
FLT_MAX_EXP DBL_MAX_EXP LDBL_MAX_EXP	无	e_{max}，最大整数x，使b^{x-1}在可表示的有限浮点数取值范围内
FLT_MAX_10_EXP	37	最大x，使10^x在可表示的有限浮点数取值范围内
DBL_MAX_10_EXP	37	
LDBL_MAX_10_EXP	37	

① FLT_RADIX与FLT_ROUNDS适用于所有3种浮点数类型。

134 ② 其他值的含义由不同的实现定义。

③ C99中新增的。

例　IEEE二进制浮点数算术标准（ISO/IEEE Std 754-1985）指定了许多微处理器使用的常用浮点数表示法。这个标准中32位单精度和64位双精度浮点数类型的模型为（调整为标准C语言表示规则）：

$$x_{\text{float}} = s \times 2^e \times \sum_{k=1}^{24} f_k \times 2^{-k} \qquad -125 \leqslant e \leqslant +128$$

$$x_{\text{double}} = s \times 2^e \times \sum_{k=1}^{53} f_k \times 2^{-k} \qquad -1021 \leqslant e \leqslant +1024$$

表5-4列出了对应于这些类型的**float.h**值。**float**类型的浮点数常量用标准C语言后缀**F**表示。IEEE的支持选项在标准C语言中是可选的。 □

参考章节　浮点数常量　2.7.2；浮点数转换　6.2.4；浮点数表示　6.1.1；NaN相关函数　17.14，17.15

表5-4　IEEE浮点数特征

名　称	FLT_*name*值	DBL_*name*值
RADIX	2	不适用
ROUNDS	实现定义的	不适用

（续）

名 称	FLT_*name*值	DBL_*name*值
EPSILON	1.19209290E－07F或0X1P-23F（C99）	2.2204460492503131E－16或0X1P－52（C99）
DIG	6	15
MANT_DIG	24	53
DECIMAL_DIG[1]	17	17
MIN	1.17549435E－38F或0X1P 126F（C99）	2.2250738585072014E－308或0X1P 1022（C99）
MIN_EXP	－125	－1021
MIN_10_EXP	－37	－307
MAX	3.40282347E+38F或0X1.fffffeP127F(C99)	1.7976931348623157E+308或0X1.fffffffffffffP1023（C99）
MAX_EXP	128	1024
MAX_10_EXP	38	308

① 这个名称没有FLT_或DBL_。

复数浮点数类型

C99中增加了6个复数浮点数类型：**float _Complex**、**double _Complex**、**long double _Complex**、**float _Imaginary**、**double _Imaginary**以及**long double _Imaginary**。复数类型是浮点数和算术类型。非复数算术类型称为*实数类型*。独立实现产品不需要实现任何复数类型，纯虚数**_Imaginary**类型即使在宿主实现产品中也是可选的。 135

complex-type-specifier（复数浮点数类型说明符）： *(C99)*

```
float _Complex
double _Complex
long double _Complex
```

关键字**_Complex**是为了避免与现有程序中的用户定义类型**complex**发生冲突。关键字**_Complex**前面的类型说明符指定*对应实数类型*（corresponding real type）。头文件**complex.h**中定义了**complex**宏，作为**_Complex**的同义词，因此没有遗留问题的编程人员可以使用简化的名称。

每个复数类型表示为对应实数类型的二元数组，每个元素与这种数组有相同的对齐要求。第一个元素表示复数的实数部分，第二个元素表示复数的虚数部分。

C99实现可以选择支持纯虚数类型**float _Imaginary**、**double _Imaginary**与**long double _Imaginary**，这些也是复数类型，但只用一个对应实数类型的元素表示。这些类型为有些复数运算提供了方便，但还不太适合作为标准的正式部分。

复数（或虚数）值至少有一个无限的部分，即使另一部分为NaN。要得到有限复数，两个部分都应为有限（不能是无限或NaN）。复数（或虚数）值在两个部分都是0时为0。

参考章节 复数转换 6.2.4；**complex.h**头文件 第23章；普通二进制转换 6.3.4

5.3 指针类型

对任何类型*T*，可以建立指向*T*的指针。根据*T*是对象类型或函数类型，相应的指针类型称为*对象指针*（object pointer）或*函数指针*（function pointer）。指针类型的值是类型为*T*的对象或函

数的地址。4.5.2节介绍了指针类型的声明。

例

```
int *ip; /* ip: a pointer to an object of type int */
char *cp; /* cp: a pointer to an object of type char */
int (*fp)(); /* fp: a pointer to a function returning
                an integer */
```

□

136

指针类型使用的两个最主要运算符是地址运算符&和间接访问运算符*，前者生成指针值，后者访问指针所指的对象。

例　下例中，指针**ip**指定为变量**i**的地址（**&i**）。赋值之后，表达式***ip**引用的就是**i**所指的对象：

```
int i, j, *ip;
ip = &i;
i = 22;
j = *ip;    /* j now has the value 22 */
*ip = 17;   /* i now has the value 17 */
```

□

指针类型的其他运算包括赋值运算、减法运算、关系运算与相等性测试、逻辑AND和逻辑OR、整数加减运算以及指针类型与整数类型之间的转换。

指针长度是由实现定义的，有时随所指对象类型不同而不同。例如，数据指针可能比函数指针更短或更长（见6.1.5节）。指针长度与任何整数类型的长度之间不必有任何关系，但我们通常假设**long**类型至少和其他指针类型一样长。在C99中，使用**intptr_t**。

在标准C语言中，指针类型可以用类型限定符**const**、**volatile**与**restrict**(C99)进行限定。指针类型的限定符（如有）可能影响指针类型的运算与转换以及指针类型允许的优化操作。

参考章节　地址运算符&　7.5.6；数组与指针　5.4.1；赋值运算符　7.9；转换表达式　7.5.1；指针转换　6.2.7；**if**语句　8.5；间接访问运算符　*7.5.7；**intptr_t**　21.5；指针声明符　4.5.2；类型限定符　4.4.3

5.3.1 通用指针

通用指针可以转换成任何对象指针类型，这是底层编程中偶尔要用到的。在传统C语言中，习惯上用类型**char** *表示通用指针，将这些通用指针转换成适当类型之后再取消引用，详见6.2节，其中将详细讨论指针转换。这样使用**char** *类型的问题是，编译器无法检查编程人员是否总是正确地转换指针类型。

标准C语言引入类型**void** *作为通用指针，同时还保留了与类型**char** *相同的表示方法，以兼容早期的实现，但它对通用指针的处理方法与早期的实现不同。通用指针不能用*或下标运算符取消引用，也不能作为加减运算的操作数。指向对象或不完整类型（而不是指向函数类型）的任何指针都可以转换为**void** *类型，然后再转换为原类型，值保持不变。**void** *类型既不是对象指针，也不是函数指针。

137

例　下面是一些指针声明和转换例子：

```
void *generic_ptr;
int *int_ptr;
char *char_ptr;
```

```
generic_ptr = int_ptr;       /* OK */
int_ptr = generic_ptr;       /* OK */
int_ptr = char_ptr;          /* Invalid in Standard C */
int_ptr = (int *) char_ptr; /* OK */
```
□

通用指针在函数原型中使用时还提供了其他灵活性。函数的正式参数能接受任何类型的数据指针时，正式参数应声明为**void** *类型。如果正式参数声明为任何其他指针类型，则实际参数必须与正式参数具有相同类型，因为标准C语言中不同指针类型是不能赋值兼容的。

例 **strcpy**函数复制字符串，因此要求参数的类型为**char** *：

```
char *strcpy(char *s1, const char *s2);
```

但**memcpy**可以取任何类型的指针，因此使用**void** *类型：

```
void *memcpy(void *s1, const void *s2, size_t n);
```
□

参考章节 赋值兼容性 6.3.2；**const**类型说明符 4.4；**memcpy**函数 14.3；**strcpy**函数 13.3

5.3.2 null指针与无效指针

C语言中每个指针类型都有一个特殊值，称为null指针。它不同于该类型的每个有效指针，而是等价于null指针常量，可以转换成其他指针类型的null指针，在布尔上下文中使用时，取值为false。C语言中的null指针常量是取值为0的任何整数常量表达式或可以转换成类型**void** *的表达式。传统上标准头文件中把**NULL**宏定义为null指针常量，标准C语言在**stddef.h**中定义，较早的实现在**stdio.h**中定义。

通常，将所有位设置为0来表示所有的null指针，但这不是必需的。事实上，不同指针类型的null指针可以有不同表示方法。如果null指针不是表示为0，则实现要竭尽全力保证在不同的指针类型之间正确地转换null指针和null指针常量。

例 语句

```
if (ip) i = *ip;
```

138

是下列语句的缩写形式：

```
if (ip != NULL) i = *ip;
```
□

一个好的编程风格是保证指针在没有指定有效对象或函数时取值为**NULL**。

有时可能不小心生成无效指针（invalid pointer），这个指针值不是null，但又没有指定有效对象或函数。产生无效指针通常是因为声明了指针变量却又没有将其初始化为**NULL**或某个有效指针。无效指针的任何用法在标准C语言中都是未定义的，包括与**NULL**比较、将其作为参数传入函数或将其赋值给另一指针。产生无效指针也可能是因为把任意整数值转换成指针类型、收回为指针所指的对象分配的存储空间（例如使用**free**函数释放存储空间）或用指针运算在数组范围之外生成指针。无效指针取消引用时可能造成运行错误。

考虑到与指针运算一起使用，C语言中还要求定义数组中最后一个元素的后一个对象的地址，尽管这个地址取消引用时仍然可能无效。这个要求便于用指针表达式遍历数组。

例 下列循环使用数组范围之外的地址，但并没有取消引用这个地址：

```
int array[N]; /* last object address is &array[N-1] */
int *p;
```

```
...
for (p = &array[0]; p < &array[N]; p++)
   ...
```
□

这个要求可能会限制一些使用不连续编址体系结构的目标计算机上的实现，将数组的最大长度减少一个对象。在这种计算机中，不能对不在连续内存区内的指针进行运算，编程人员只有通过分配数组保证内存连续。

参考章节 **free**函数 16.1；整数常量 2.7.1；指针运算 7.6.2；**stddef.h**函数 11.1；**void** *类型 5.3.1

5.3.3 指针注意事项

许多C语言编程人员以为所有指针类型（实际上是所有地址）具有统一表示方法。在常见的字节寻址计算机上，所有指针通常是简单字节地址，占用一个字。在这些计算机上进行指针类型与整数类型之间的转换时不需要改变表示方法，不会丢失信息。

139 事实上，C语言没有这么严格的要求。6.1节将详细讨论这个问题，这里简要介绍如下：

1. 指针通常与**int**类型长度不同，有时与**long**类型长度不同，有时其长度是个编译器选项。在C99中，**intptr_t**是能够容纳对象指针的整型类型。
2. 字符和**void** *指针可能比其他类型的指针大，可以使用不同于其他类型的指针的表示方法。例如，可以用高顺序位，这些位在其他类型的指针中通常为0。
3. 函数指针与数据指针可能有大不相同的表示方法，包括不同长度。

编程人员总是在指针类型之间转换时使用显式转换，一定要保证函数的指针参数具有函数所要的正确类型。在标准C语言中，可以用**void** *作为通用对象指针，但没有通用函数指针。

参考章节 转换 7.5.1；**intptr_t** 21.5；**malloc**函数 16.1；指针转换 6.2.7

5.4 数组类型

如果*T*为除**void**、不完整数组类型或函数类型以外的任何C语言类型，则可以声明“*T*的数组”类型。这个类型的值是*T*类型的元素序列。所有数组起始下标为0。4.5.3节介绍了数组声明符的语法和含义，包括不完整数组类型和变长数组类型。

例 声明为**int A[3];**的数组包括元素**A[0]**、**A[1]**与**A[2]**。下列代码中，声明一个整数数组（**ints**）和一个指针数组（**ptrs**），指针数组中的每个指针设置为等于整数数组中相应整数的地址：

```
int ints[10], *ptrs[10], i;
for (i = 0; i < 10; i++)
   ptrs[i] = &ints[i];
```

数组的内存长度（从**sizeof**运算符看）总是等于数组的元素个数乘以元素的存储长度。 □

参考章节 数组声明符 4.5.3；**sizeof**运算符 7.5.2；存储单元 6.1.1；结构类型 5.6；变长数组 5.4.5

5.4.1 数组与指针

在C语言中，“*T*的数组”与“*T*的指针”类型之间具有密切对应关系。首先，表达式中出现

数组标识符时，标识符类型要从“T的数组”转换成“T的指针”，标识符的值要转换成数组中第一个元素的指针，这是普通一元转换规则之一。这个转换规则的惟一例外是用数组标识符作为 140
sizeof运算符或地址运算符**&**的操作数时，**sizeof**返回整个数组的长度，而**&**返回指向数组的指针（而不是指向第一个元素的指针的指针）。

例 下面第二行中，数值**a**转换成数组中第一个元素的指针：

```
int a[10], *ip;
ip = a;
```

相当于下列语句：

```
ip = &a[0];
```

sizeof(a)的值为**sizeof(int)*10**，而不是**sizeof(int *)**。 □

第二，数组下标用指针运算定义，即表达式**a[i]**的定义与***((a)+(i))**相同，其中**a**用普通一元转换规则变为**&a[0]**。这种关于下标的定义还意味着**a[i]**与**i[a]**相同，任何指针都可以像数组一样标注下标。编程人员要保证指针指向正确的元素数组。

例 如果**d**的类型为**double**，**dp**为指向**double**对象的指针，则表达式

```
d = dp[4];
```

只在**dp**当前指向**double**数组的元素时才定义，并要求所指元素后面至少有4个数组元素。 □

参考章节 地址运算符**&** 7.5.6；加法运算符**+** 7.6.2；数组声明符 4.5.3；数组转换 6.3.3；间接访问运算符* 7.5.7；指针类型 5.3；**sizeof**运算符 5.4.4，7.5.2；下标 7.4.1；普通一元转换 6.3.3

5.4.2 多维数组

多维数组声明为“数组的数组”，例如：

```
int matrix[12][10];
```

其声明**matrix**为12×10的**int**元素数组。这个语言没有限制多维数组的维数。数组**matrix**也可以用两步声明，使这个结构更清晰：

```
typedef int vector[10];
vector matrix[12];
```

即**matrix**是12个元素的数组，其中每个元素又是一个包含10个元素的**int**数组。**matrix**的类型为**int[12][10]**。多维数组元素按以行为主的顺序存放，即只有最后一个下标不同的元素相邻存放。 141

多维数组将数组转换成指针的方法和将单维数组转换成指针的方法一样。

例 数组**int t[2][3]**的元素（按地址递增顺序）存储如下：

```
t[0][0], t[0][1], t[0][2], t[1][0], t[1][1], t[1][2]
```

表达式**t[1][2]**扩展成***(*(t+1)+2)**，按下列步骤求值：

1. 表达式**t**是一个2×3数组，转换成指向第一个3元素子数组的指针。
2. 表达式**t+1**是第二个3元素子数组的指针。
3. 表达式***(t+1)**是第二个3元素整数子数组，转换成指向这个子数组中第一个整数的指针。

4. 然后，表达式***(t+1)+2**转换为指向第二个3元素整数子数组中的第三个整数的指针。

5. 最后，***(*(t+1) +2)**是第二个3元素整数子数组中的第三个整数**t[1][2]**。 □

一般来说，任何类型为“*T*的*i*×*j*×…×*k*数组”的表达式*A*立即转换成“*T*的*j*×…×*k*数组的指针”。

参考章节 加法运算符+ 7.6.2；数组声明符 4.5.3；间接访问运行符* 7.5.7；指针类型 5.3；下标 7.4.1

5.4.3 数组边界

分配数组存储区时，要知道数组长度。但由于通常不检查声明的数组边界中指定的下标，因此声明另一模块中定义的外部一维数组或声明作为函数正式参数的一维数组时（见4.5.3节），通常可以省略长度（即，使用不完整数组类型）。

例 下列函数**sum**返回外部数组**a**的前**n**个元素的和，其数组边界没有指定：

```
extern int a[];

int sum(int n)
{
    int i, s = 0;
    for (i = 0; i < n; i++)
        s += a[i];
    return s;
}
```

142 数组可以作为参数传递如下：

```
int sum(int a[], int n)
{
    int i, s = 0;
    for (i = 0; i < n; i++)
        s += a[i];
    return s;
}
```

参数**a**可以声明为**int *a**而不改变函数体，这样声明可以更准确地反映实现，但意图表达的不是很明确。 □

使用多维数组时，需要对除了第一维以外的所有其他维指定边界，以便进行正确的地址运算：

```
extern int matrix[][10]; /* ?-by-10 array of int */
```

如果不指定这种边界，则声明出错。在C99中，数组（包括多维数组）可以是变长的。

参考章节 数组声明符 4.5.3；定义与引用声明 4.8；间接访问运算符* 7.5.7；省略数组边界 4.5；指针类型 5.3；下标 7.4.1；变长数组 5.4.5

5.4.4 运算

数组值可以直接进行的运算只有应用**sizeof**运算符和地址运算符**&**的运算。对于**sizeof**运算，数组要有边界，结果为数组占用的存储单元数。对*T*类型的*n*元数组，**sizeof**运算的结果总是**sizeof** *T*的*n*倍。而**&**运算的结果是指向数组（第一个元素）的指针。

在其他上下文中，例如为数组标记下标时，数组值实际上作为指针处理，因此可以对数组

值采用指针运算。

参考章节 数组声明符 4.5.3；从数组转换成指针 6.2.7；指针类型 5.3；**sizeof**运算符 7.5.2；下标 7.4.1

5.4.5 变长数组

C99允许C语言编程人员使用变长数组，变长数组只有到运行时才能确定其长度。变长数组的声明与定长数组相似，只是数组长度用非常量表达式指定。遇到声明时，对长度表达式求值并生成这个长度的数组，长度应为正整数值。一旦生成，变长数组就不能再改变其长度。数组中的元素一直可以访问到分配的长度，访问到分配的长度之外时则会产生不确定行为。这种界外访问没有检查要求。包含声明的块完成时，删除这个数组。每个块开始时，都会分配一个新的数组。

143

先不考虑函数中的数组参数，变长数组应在块作用域中声明，不能是**static**存储类型或**extern**存储类型。只能将普通标识符声明为变长数组，而不能将结构成员或联合成员声明为变长数组。数组变量的作用域从声明点延续到所在的最内层块结束为止。数组生存期从声明开始一直延续到离开数组作用域为止。在C99中，离开数组作用域的操作包括完成这个块，跳出这个块或跳回声明之前的块地址。实现者可在处理声明的过程中，在执行堆栈中对数组分配空间。

可变修改类型包括变长数组和其他包括变长数组的类型，如变长数组指针。只有不带连接的块作用域范围内的普通标识符才能声明为可变修改类型。这样就留下了一个漏洞：可以用可变修改类型（而不是变长数组）作为**static**块作用域标识符的类型。这时，尽管块执行时保持**static**标识符的值，但每次进入块时，嵌入其中的变长数组都可能改变它的维度。

例 下列代码段中，**a**和**b**是变长数组，指针**c**具有可变修改类型：

```
int a_size;
...
void f(int b_size)
{
    int c_size = b_size + a_size;
    int a[a_size++];
    int b[b_size][b_size];
    static int (*c)[5][c_size];
    ...
}
```

□

变长数组的限制简化了C99中的实现，但仍然保留了它的大部分作用。如果不加限制，则会出现大量复杂细节与交互。结构可能要对可变修改类型成员提供隐藏类型描述符。在文件作用域中声明变长数组要求C语言接受在运行时处理复杂的顶层声明的开销（C++和其他语言有这种机制，但这不符合C语言的精神）。

如果在**typedef**声明中使用变长数组，则长度表达式求值一次，即在遇到**typedef**声明时求值，而不是每次使用新类型名时都求值。

例

```
/* Assume n has the value 5 here */
typedef int[n] vector;
n += 1;
```

144
```
vector a;
int b[n];
```

变量**a**是5个元素的整数数组，在遇到**typedef**声明时反映**n**值。相反，**b**是6个元素的整数数组，因为遇到**b**声明时会改变**n**值。 □

变长数组参数　变长数组或可变修改类型可以作为函数参数类型。如果数组长度也是参数，则根据C语言词法的作用域规则，其必须先出现。

调用具有变长数组参数的函数时，数组参数的维长应符合函数定义中声明的数组参数，否则结果不确定。

例　第一个函数定义是正确的，第二个可能非法，或者要用其他变量**r**与**c**的值来计算**a**的维度。

```
void f( int r, int c, int a[c][r] ) {...} /* OK */
void f( int a[c][r], int r, int c ) {...}
   /* NO: r, c are not visible to a[c][r] */
```
□

在函数原型声明中，而不是部分函数定义中，可以用**[*]**指定变长数组维度。这种在函数原型的括号中出现的非常量表达式等价于**[*]**。

例　下列函数原型都是兼容的。尽管第三个原型表达了正方形数组，但编译时不检查这个限制。最后一个原型表示可以省略变长数组最内层的维度（或惟一维度）。

```
void f(int, int [*][*]);
void f(int n, int [*][m]);
void f(int n, int [n][n]);
void f(int, int [][*]);
```
□

参考章节　数组声明符　4.5.3；函数原型　9.2；**sizeof**运算符　7.5.2

5.5 枚举类型

声明枚举类型的语法如下：

enumeration-type-specifier（枚举类型说明符）：
　　enumeration-type-definition
　　enumeration-type-reference

enumeration-type-definition（枚举类型定义）：
145
　　enum *enumeration-tag*$_{opt}$ { *enumeration-definition-list* }
　　enum *enumeration-tag*$_{opt}$ { *enumeration-definition-list* , } *(C99)*

enumeration-type-reference（枚举类型引用）：
　　enum *enumeration-tag*

enumeration-tag :
　　identifier

enumeration-definition-list :
　　enumeration-constant-definition
　　enumeration-definition-list , *enumeration-constant-definition*

enumeration-constant-definition :

enumeration-constant
enumeration-constant = *expression*

enumeration-constant:
identifier

C语言中的枚举类型是用称为枚举常量的标识符表示的一组整数值。枚举常量在定义类型时说明，类型为**int**。每个枚举类型表示为实现定义的整数类型并与这个整数类型兼容。这样，进行类型检查时，枚举类型只作为一个整数类型。在C语言允许表达式的上下文中，就可以使用枚举类型常量或枚举类型值（C++中则不然，见5.13.1节）。

为了方便起见，C99允许在*enumeration-definition-list*末尾放上逗号。

例 下列声明：

```
enum fish { trout, carp, halibut } my_fish, your_fish;
```

生成一个枚举类型，取值为**trout**、**carp**与**halibut**，并声明两个枚举类型变量**my_fish**与**your_fish**，可以用下列赋值语句赋值：

```
my_fish = halibut;
your_fish = trout;
```

□

枚举类型的变量和枚举类型的其他对象可以在包含枚举类型定义的同一声明中声明，也可以在后面通过“枚举类型引用”提及枚举类型的声明中进行声明。

例 声明

```
enum color { red, blue, green, mauve }
   favorite, acceptable, least_favorite;
```

等价于两个声明

146

```
enum color { red, blue, green, mauve } favorite;
enum color acceptable, least_favorite;
```

又等价于4个声明

```
enum color { red, blue, green, mauve };
enum color favorite;
enum color acceptable;
enum color least_favorite;
```

枚举标志**color**使枚举类型可以在定义之后引用。尽管另一个声明

```
enum { red, blue, green, mauve }
   favorite, acceptable, least_favorite;
```

声明了相同的变量和枚举常量，但由于没有枚举标志，因此无法在后面声明时引入更多这个类型的变量。

□

枚举标志与结构标志和联合标志属于同一重载类，其作用域与源程序中同一位置声明的变量作用域相同。

定义为枚举常量的标识符是与变量、函数和typedef名称同一重载类的成员，其作用域与源程序中同一位置定义的变量作用域相同。

例 下列代码中，**shepherd**声明为枚举常量，隐藏前面的整型变量**shepherd**声明。但

是，浮点数变量**collie**的声明会造成编译错误，因为**collie**已经在同一作用域中声明为枚举常量。

```
int shepherd = 12;
{
    enum dog_breeds {shepherd, collie};
        /* Hides outer declaration of the name "shepherd" */
    float collie;
        /* Invalid redefinition of the name "collie" */
}
```
□

枚举类型通过与枚举常量相关联的整数值实现，因此枚举类型数值的赋值与比较可以用整数赋值与比较实现。整数值用下列方式与枚举常量相关联：

1. 可以在类型定义中用下列语句显式地将整数值与枚举常量相关联：

```
enumeration-constant = expression
```

147 表达式应为整数类型的表达式，包括涉及前面定义的枚举常量的表达式，例如：

```
enum boys { Bill = 10,
            John = Bill+2,
            Fred = John+2 };
```

2. 如果不指定明确的值，则第一个枚举常量将与数值0相关联。
3. 没有明确地与整数值进行关联的后续枚举常量将与比上一枚举常量相关联的值大1的整数值相关联。

任何能表示为**int**类型的带符号整数值都可以和枚举常量相关联。可以随意选择正整数和负整数，甚至可以将同一整数和两个不同的枚举常量相关联。

例　对于下列声明：

```
enum sizes { small, medium=10, pretty_big, large=20 };
```

small、**medium**、**pretty_big**与**large**的值分别为0、10、11、20。尽管下列定义是有效的：

```
enum people { john=1, mary=19, bill=-4, sheila=1 };
```

但其效果是使表达式**john == sheila**成立，这是不够直观的。
□

尽管建议用结构类型和联合类型作为枚举类型定义的形式，并进行严格类型检查，但事实上标准C语言中的枚举类型（本书提供的定义）不过是通过一种更易读的方法定义整型常量而已。作为风格，我们建议编程人员把枚举类型与整数类型区分开，不要把枚举类型与整型表达式不加转换而混合使用。事实上，一些UNIX C编译器实现枚举类型的弱类型化形式，要先进行类型转换之后才允许枚举类型与整数之间进行某种转换。

参考章节　转换表达式　7.5.1；标识符　2.5；重载类　4.2.4；作用域　4.2.1

5.6 结构类型

C语言中的结构类型相当于其他编程语言中的记录（record）类型，是分量（components）（也称成员（members）或字段（fields））的集合名，成员可以取不同类型。结构可以定义为包含相关数据对象。

structure-type-specifier（结构类型说明符）：
structure-type-definiuon
structure-type-referernce

structure-type-definition（结构类型定义）：
struct *structure-tag*$_{opt}$ { *field-list* }

148

structure-type-reference（结构类型引用）：
struct *structure-tag*

structure-tag :
identifier

field-list :
component-declaration
field-list component-declaration

component-declaration :
type-specifier component-declarator-list ;

component-declarator-list :
component-declarator
component-declarator-list , *component-declarator*

component-declarator :
simple-component
bit-field

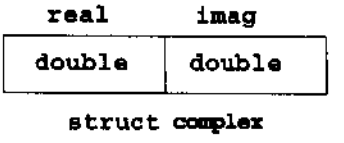
simple-component :
declarator

bit-field :
declarator$_{opt}$: *width*

width :
constant-expression

例 编程人员要实现复数时（C99之前）可以定义**complex**结构，将实数与虚数部分分别保存为**real**与**imag**成员。第一个声明定义了新类型，第二个声明声明了两个变量**x**与**y**：

```
struct complex {
    double real;
    double imag;
};
struct complex x,y;
```



struct complex

可以编写**new_complex**函数，生成这个类型的新对象。注意，用选择运算符（.）访问这个结构的成员： 149

```
struct complex new_complex(double r, double i)
{
    struct complex c;
    c.real = r;
    c.imag = i;
    return c;
}
```

也可以定义这个类型的运算，如**complex_multiply**：

```
struct complex complex_multiply( struct complex a,
                                 struct complex b )
{
    struct complex product;
    product.real = a.real * b.real - a.imag * b.imag;
    product.imag = a.real * b.imag + a.imag * b.real;
    return product;
}
```

□

例　声明

```
struct complex { double real, imag; } x, y;
```

等价于下面两个声明：

```
struct complex { double real, imag; };
struct complex x, y;
```

□

5.6.1 结构类型引用

使用语法类*structure-type-definition*和*union-type-definition*的类型说明符（5.7节）引入了与其他类型不同的新类型定义。如果定义中存在结构标志，则结构标志与一个新标志相关联，可以在后续的结构类型引用中使用。

定义（和类型标志，如果有）的作用域是从声明点开始到说明符所在最内层块的末尾结束。新定义显式地覆盖（隐藏）了所在块中的任何类型标志定义。

如果不需要结构长度，则可以在同一作用域或所在作用域中使用前面没有定义过的语法类*structure-type-reference*和*union-type-reference*的类型说明符（5.7节），包括声明：

1. 结构指针

2. typedef名称，作为结构的同义词

使用这种说明符引入类型的“不完整”定义和最内层块使用的类型标志。要让这个定义完
150 整，同一作用域中就要在后面加上*structure-type-definition*或*union-type-definition*。

作为特例，声明中没有声明符时，*structure-type-reference*或*union-type-reference*隐藏所有包含它的作用域中的任何类型标志定义并建立不完整类型。

例　下面内部块中正确定义了两个自引用结构：

```
{
    struct cell;
    struct header { struct cell   *first; … };
    struct cell   { struct header *head;  … };
    …
}
```

第一行的“不完整”定义“**struct cell;**”要隐藏所有包括作用域中的任何类型标志**cell**的定义。第二行的**struct header**定义自动隐藏任何包括定义，用**struct cell**定义指针是有效的。第三行**struct cell**的定义完成**cell**的信息。 □

不完整类型声明也存在于*structure-type-definition*或*union-type-definition*中，从第一次引用新标志到定义完成为止。这样就使一个结构类型可以包括自己的指针（如图5-1）。

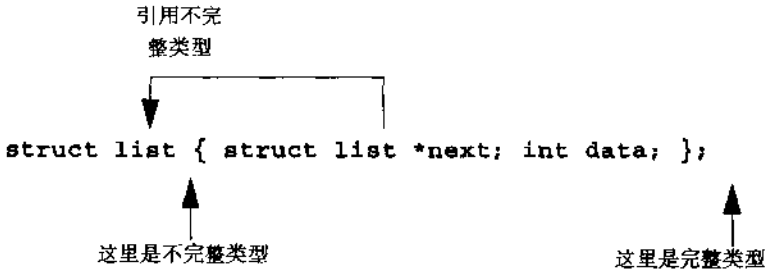


图5-1 声明中的不完整结构类型

参考章节 声明 4.1；声明符 4.5；重复有效性 4.2.2；作用域 4.2.1；选择运算符. 7.4.2；类型等价 5.11

5.6.2 结构运算

结构运算随编译器不同而不同。所有C语言编译器都对结构提供选择运算符.和->，较新的编译器还允许对结构赋值、将结构作为参数传递给函数以及从函数返回结构（对较早的编译器，赋值要一个成员一个成员地进行，而且只能从函数返回或向函数传递结构指针）。 151

两个结构不能比较相等性。结构类型的对象是其他类型的成员序列。由于目标计算机可能将某些数据对象限制在某些地址边界，因此结构对象可能包含“空穴”，一些存储单元不属于结构中的任何成员。这些空穴使逐位进行比较的相等性测试无法实现，而逐个成员进行比较的相等性测试可能成本太高（当然，编程人员可以自己编写逐个成员进行比较的相等性测试函数）。

在可以对结构采用一元地址运算符&以获得指向结构的指针的任何场合中，也可以对结构成员采用地址运算符&以取得结构成员的指针。指针有可能指向结构中间。这个规则不适用于定义为位字段的成员。定义为位字段的成员通常不在机器的可寻址边界上，因此可能无法建立指向位字段的指针。C语言没有提供指向位字段的指针。

参考章节 地址运行符& 7.5.6；赋值 7.9；位字段 5.6.5；相等运算符== 7.6.5；选择运算符.与-> 7.4.2；类型等价 5.11

5.6.3 成员

结构成员可以是非可变修改类型的任何对象。结构不能包含自身实例，但可以包含指向自身实例的指针。

在C99中，结构成员不能有可变修改类型。结构的最后一个成员可能是不完整数组类型，这时称为*灵活数组成员*（flexible array member）（见5.6.8节）。

例 下列声明是无效的：

```
struct s {
```

```
    int a;
    struct S next; /* invalid! */
};
```

下列声明是有效的：

```
struct S {
    int a;
    struct S *next; /* OK */
};
```

□

结构成员名在与结构类型相关联的特殊重载类中定义，即一个结构中的成员名应当惟一，但可以和其他结构中的成员名相同，可以和变量名、函数名以及类型名相同。

152

例 考虑下列声明序列：

```
int x;
struct A { int x; double y; } y;
struct B { int y; double x; } z;
```

标识符**x**有3个不冲突的声明：一个是整型变量，一个是结构**A**的整型成员，一个是结构**B**的浮点数成员。这3个声明分别在下列表达式中使用：

```
x
y.x
z.x
```

□

如果一个成员中定义结构标志，则这个标志的作用域延续到定义该结构的块末尾（如果在顶层定义包括结构，则内部标志也在顶层定义）。

例 在下列声明中：

```
struct S {
    struct T {int a, b; } x;
};
```

□

标志**T**从出现时开始定义，到定义**S**的作用域结束。

历史备注 C语言原始的定义指定所有结构中的所有成员从相同的重载类分配，因此两个结构不能有同名成员（例外是成员具有相同类型和在结构中具有相同相对位置时）。这种解释已经过时，但早期文档和一些早期编译器的实际实现中可能遇到这种情形。

参考章节 灵活数组成员 5.6.8；不完整数组类型 5.4；重载类 4.2.4；作用域 4.2.1；可变修改类型 5.4.5

5.6.4 结构成员存储布局

大多数编程人员并不关心结构成员存储。但是，C语言可以让编程人员控制结构成员的存储。C语言编译器被限制成按严格的顺序为成员分配递增的内存地址，第一个成员从结构开始地址开始。

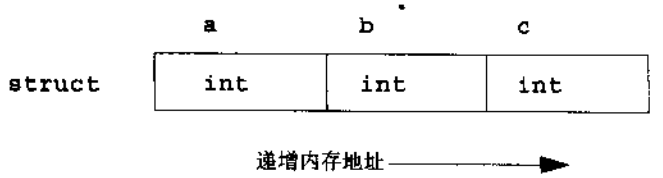
例 结构

```
struct { int a, b, c; };
```

153 与结构

```
struct { int a; int b, c; };
```

的结构成员布局没有什么不同，两者都按递增内存地址先放**a**，再放**b**，最后放**c**，如下图所示：



□

对指向同一结构中不同成员的两个指针**p**和**q**，要使**p<q**，当且仅当**p**所指的成员的声明比**q**所指的成员的声明在结构声明中早出现。

例

```
struct vector3 { int x, y, z; } s;
int *p, *q, *r;
...
p = &s.x;
q = &s.y;
r = &s.z;/* At this point p < q, q < r, and p < r. */
```

□

连续成员之间和最后一个成员之后可能出现空穴或填充区，使成员在内存中适当对齐。这种空穴中出现的位模式是不确定的，不同结构可能不同，同一结构在不同时间也可能不同。**sizeof**运算符返回的值包括填充区所占的空间。有些实现提供杂注或开关来控制结构成员的存储。

5.6.5 位字段

C语言允许编程人员将整型成员存储在比编译器通常允许的空间更小的空间内。这些整型成员称为*位字段*（bit field），指定位字段时在成员声明符后面加上冒号和一个常量整数表达式，表示字段的宽度（位）。

例 下列结构有3个成员**a**、**b**、**c**，分别占用4位、5位和7位：

```
struct S {
    unsigned a:4;
    unsigned b:5, c:7;
};
```

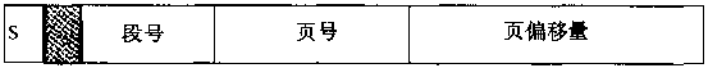
□ 154

*n*位的位字段可以表示0～2^n-1范围的无符号整数值和-2^{n-1}～$2^{n-1}-1$范围的带符号整数值，假设带符号整数用对二的补码表示。C语言原始的定义只允许**unsigned**类型的位字段，但标准C语言允许**unsigned int**类型、**signed int**类型或**int**类型的位字段，分别称为*无符号位字段*、*带符号位字段*和*普通位字段*。和普通字符一样，普通位字段可以无符号或带符号。一些C语言实现允许任何整数类型的位字段，包括**char**类型。C99允许**_Bool**类型的位字段。

位字段通常用于机器相关程序中，这样的程序强制数据结构对应于固定硬件表示。虽然成员（特别是位字段）存储成结构的具体方式是与实现相关的，但每种实现中的方式都是可预测的。根据本节稍后讨论的规则，位字段在结构中的存储应尽量紧凑。因此，位字段的使用通常是不可移植的。编程人员应从实现的文档中了解是否需要按特定方式在内存中设计结构的布局，

然后验证C语言编译器是否实际按所需方式存储成员。

例　下面的例子用位字段生成匹配预定义格式的结构。下列布局把32位字当作假想计算机上的虚拟地址。字中包含段号字段、页号字段和页偏移量字段，加上一个管理位和一个未设置位。



字段宽度（位）　1　1　6　8　16

为了复制这个布局，首先要知道计算机存储位字段时是从左向右还是从右向左，即是用“高位存储法”还是“低位存储法”（见6.1.2节）。如果位字段从右向左存储，则相应结构定义如下：

```
typedef struct {
    unsigned Offset     : 16;
    unsigned Page       : 8;
    unsigned Segment    : 6;
    unsigned UNUSED     : 1;
    unsigned Supervisor : 1;
} virtual_address;
```

相反，如果位字段从左向右存储，则相应结构定义如下：

```
typedef struct {
    unsigned Supervisor : 1;
    unsigned UNUSED     : 1;
    unsigned Segment    : 6;
    unsigned Page       : 8;
    unsigned Offset     : 16;
} virtual_address;
```

155　□

普通整型位字段的符号性与普通字符的符号性相同，即普通整型位字段可以实际实现为无符号类型或带符号类型（5.1.3节）。要分别实现无符号位字段或带符号位字段来保存无符号值或带符号值。

例　考虑下列标准C语言声明在采用对二的补码为编码方式的计算机中的效果：

```
struct S { unsigned ubf:3;
           signed sbf:3;
           int bf:3; } x = { -1, -1, -1 };
...
int i = x.ubf;
int j = x.sbf;
int k = x.bf;
```

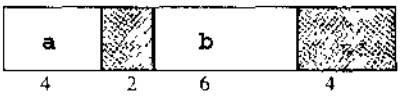
i的值应为7，j的值应为 -1，但k的值可能是7或 -1。　□

编译器可以随意限制位字段最大长度并指定位字段不能越过某个地址边界。这些对齐限制通常与目标计算机的自然字长有关。如果字段相对计算机的自然字长过长，则编译器会发出相应错误消息。如果字段跨过字的边界，则可能移到下一个字。

结构中还可以包括一个无名位字段，提供相邻成员之间的填充。无名位字段不能引用，运行时无名位字段的内容是无法预测的。

例 下列结构将成员a放在结构的前4位，随后是2位填充，然后在接着的6位中放成员b（假设基本字长为16位，则结构末尾的4位未用，见5.6.7节）

```
struct S {
   unsigned a : 4;
   unsigned   : 2;
   unsigned b : 6;
};
```



□

对无名位字段指定长度0具有特殊含义——表示存储前一位字段的区中不能再放置更多的位字段。这里的区（area）指实现定义的存储单元。 156

例 下列结构中，成员b在成员a后面的自然地址边界上开始（如16位）。新结构比旧结构多占一倍的存储空间：

```
struct S {
   unsigned a : 4;
   unsigned   : 0;
   unsigned b : 6;
};
```

□

位字段成员可能不能采用地址运算符&，因为许多计算机不能直接寻址任意长度的字段。

参考章节 地址运算符& 7.5.6；对齐限制 6.1.3；**_Bool**类型 5.1.4；字节顺序 6.1.2；枚举类型 5.5；带符号类型 5.1.1；无符号类型 5.1.2

5.6.6 移植性问题

依赖于存储策略是危险的，原因有几个。第一，不同计算机对数据类型的对齐限制不同。例如，一些计算机上的4字节整数要从4的倍数的字节边界开始，而有些计算机上的整数则向最近的字节边界对齐。

第二，位字段宽度的限制不同。一些计算机的字长为16位，这限制了字段的最大长度和字段无法越过的边界。而一些计算机的字长为32位，等等。

第三，不同计算机把字段包装成字的方式不同，即字节顺序不同。在Motorola 68000系列计算机中，字符从左向右包装成字，从最高有效位到最低有效位。而在Intel 80x86系列计算机中，字符从右向左包装成字，从最低有效位到最高有效位。从上节的**virtual_address**例子可以看出，采用不同字节顺序的计算机需要不同的结构定义。

有两种情形好像可以使用位字段：

1. 要准确匹配预定义数据结构，以便在C语言程序中引用（这些程序无法移植）。
2. 要维护结构化数据的数组，其长度很大，要求成员紧凑存储，节省内存。

用C语言位运算符进行掩码和移位，可以实现位字段而不对字中的字节顺序敏感。

例 假设要访问**virtual_address**结构中的**Page**字段（见5.6.5节）。由于这个8位字段距离字的低顺序端16位，因此可以用下列代码访问： 157

```
unsigned V;     /* formatted as a virtual_address */
int Page;
...
Page = (V & 0xFF0000) >> 16;
```

这段代码等价于更可读的结构成员访问语句**Page=V.Page**，但掩码和移位方法不会对字中的字节顺序敏感，像**virtual_address**定义一样。下面演示了**V==0xb393352e (Page==0x93)**的掩码和移位运算：

```
10110011100100110011010100101110   V
00000000111111110000000000000000   0xFF0000
00000000100100110000000000000000   V & 0xFF0000
00000000000000000000000010010011   (V & 0xFF0000)>>16
```

□

可以用类似运算设置位字段的值。这两种访问方法的运行性能可能稍有不同。

参考章节　对齐限制　6.1.3；按位运算符　7.6.6；字节顺序　6.1.2；移位运算符　7.6.3

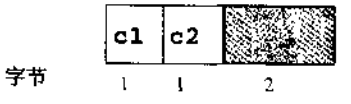
5.6.7 结构长度

结构类型对象的长度是表示该类型中所有成员所需的存储空间量，包括成员间和成员后面未用的填充空间。规则是结构要填充到该类型数组的元素可能占用的长度（对任何类型*T*，包括结构，*T*的*n*元素数组长度为*T*的长度的*n*倍）。另一种方法是，结构要在与开始时相同的对齐边界上结束，即如果结构从偶字节边界开始，则要到偶字节边界结束。结构类型的对齐要求至少和要求最严的成员的对齐要求一样严格。

例　在所有结构从4字节倍数的地址开始的计算机上，下列结构的长度为4的倍数（也许就是4），即使实际上只使用了两个字节：

```
struct s {
    char c1;
    char c2;
};
```

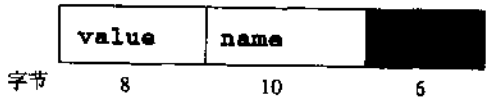
158



□

例　如果计算机上要求所有**double**类型对象的地址为8字节的倍数，则下列结构的长度可能是24字节，即使只声明18字节：

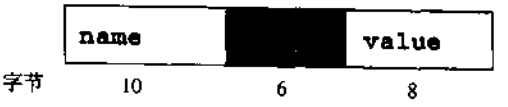
```
struct s {
    double value;
    char name [10];
};
```



结构末尾要填充6个单元，使结构的长度成为对齐要求8的倍数。如果不用填充，则这种结构的数组中不能保证所有结构的**value**成员正确对齐8的倍数地址。 □

例　对齐要求可能在结构中间造成填充。如果上例中逆转成员顺序，则长度保持24，但未用空间出现在成员之间，使数值成员可以对齐到相对于结构开头为8字节倍数的地址：

```
struct s {
    char name [10];
    double value;
};
```



结构类型对象要求地址为8的倍数，因此这种对象的数值成员总是正确对齐。 □

5.6.8 灵活数组成员

在C99中，结构的最后一个成员可能是不完整数组类型，这时称为*灵活数组成员*（flexible array member）。灵活数组成员是为了使一个历史悠久而不安全的C语言编程观念合法化，就是

结构长度可以在运行时改变。

要使用灵活数组成员，声明结构类型*S*，其最后一个成员是灵活数组成员*F*，元素类型为*E*。类型*S*不能只包含*F*，至少还要有另一个命名成员。例如：

```
struct S { int F_len; double F[]; }; /* E is double */
```

sizeof(*S*)的值定义为忽略成员*F*的结构长度，但这个长度要包括*F*之前要求的任何填充（要确定需要的填充量，假设*F*声明为相同元素类型的定长数组并使用在*F*之前需要的填充量）。

用类型*S*的lvalue访问数据对象时，可以把*F*看成定长数组，其长度*L*不会使*S*超过数据对象的长度，即如果数据对象的长度为*D*，则*L*是满足：**sizeof(*S*) + *L**sizeof(*E*) <= *D***条件的最大非负整数，可以引用*F*[**0**]、*F*[**1**]、...*F*[*L* – 1]。如果只是声明*S*类型的变量，则可能无法使用这个数组，因为数据对象（变量）没有容纳数组的空间（*D*等于**sizeof**(*S*)，因此*L*应为0）。即使没有数组的空间，也可以引用**&***F* [**0**]。 159

要用类型*S*访问大于自己的数据对象，可以声明指向*S*的指针，将一个大对象的地址赋值给它，或用联合使*S*与大对象共用存储区。

例 通常用灵活数组成员定义一个结构，保存变长向量和向量的长度：

```
struct Vec { int len; double vec[]; }
```

如果向量的长度固定，则可以静态声明：

```
#define N 20 /* Length of vector */
union{
    char data_object[sizeof(struct Vec) + N*sizeof(double)];
    struct S v;
} u = { .v = {N} }; /* C99 designated initializer */
```

如果向量的长度要到运行时才能确定，则可以用**malloc**分配空间：

```
struct Vec *p;
int n; /* length of vector */
...
p = malloc( sizeof(struct Vec) + n * sizeof (double));
p->len = n;
```

向量的使用方法如下：

```
for (i = 0; i < u.v.len; i++) u.v.vec[i] = 0.0;
for (i = 0; i < p->len; i++) p->vec[i] = 0.0;
```

在C99之前，必须要用如下的方式声明结构：

```
struct Vec { int len; double vec[1]; }
```

将**malloc**调用变成如下：

```
p = malloc( sizeof(struct Vec) + (len -1)*sizeof (double));
```

尽管这个代码通常能工作，但其行为仍然是未定义的。 □

5.7 联合类型

定义联合类型的语法与定义结构类型的语法大致相同： 160

union-type-specifier（联合类型说明符）：
　　union-type-definition
　　union-type-reference

union-type-definition（联合类型定义）：
　　union *union-tag*$_{opt}$ { *field-list* }

union-type-reference（联合类型引用）：
　　union *union-tag*

union-tag：
　　identifier

在联合中定义成员的语法和结构定义成员的语法中相同。在传统C语言中，联合不能包含位字段，但标准C语言取消了这个限制。

和结构类型与枚举类型一样，每个联合类型定义引入不同于其他联合类型的新联合类型。如果联合标志出现在定义中，那么它与一个新类型相关联，并可以在以后的联合类型引用中使用。联合类型允许向前引用和不完整定义的规则与结构类型的规则相同。

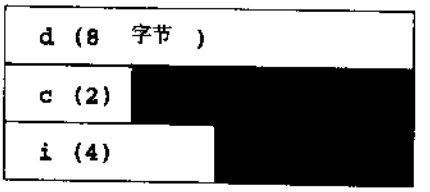
联合成员可以是非可变修改类型的任何对象。联合不能包含自身的实例，但可以包含指向自身实例的指针。和结构一样，联合成员名在与联合类型相关联的特殊重载类中定义，即一个联合中的成员名应当惟一，但可以和其他联合中的成员名同名，可以和变量名、函数名以及类型名同名。

5.7.1 联合成员存储布局

联合类型的每个成员从联合开头开始分配存储空间。联合中一次只能包含一个成员值。联合类型对象从任何被包含成员适合的存储对齐边界开始。

例 下面的联合有3个成员，有效地共用存储区：

```
union U {
    double d;
    char c[2];
    int i;
};
```



□

例 如果联合类型和对象定义如下：

```
static union U { …; int C; …; } object, *P = &object;
```

161 则下列两个等式成立：

```
(union U *) & (P->C) == P
&(P->C) == (int *) P
```

此外，不管C为何种类型的成员，不管C的前后是什么类型的成员，这两个等式总是成立。

参考章节 对齐限制 6.1.3

5.7.2 联合类型长度

联合类型对象的长度是表示该类型中所有成员所需的存储空间量，包括成员间和成员后面未用的填充空间。规则是联合要填充到该类型数组的元素可能占用的长度（对任何类型*T*，包括

联合，T的n元素数组长度为T长度的n倍）。另一种方法是，联合要在与开始时相同的对齐边界上结束，即如果联合从偶字节边界开始，则要到偶字节边界结束。

联合类型的对齐要求至少和要求最严的成员的对齐要求一样严格。

例 如果计算机上要求所有**double**类型对象的地址为8字节的倍数，则下列并的长度可能是16字节，即使只声明10字节：

```
union U {
    double value;
    char name [10];
};
```

末尾要填充6个单元，使联合的长度成为对齐要求8的倍数。如果不用填充，则这种联合的数组中不能保证所有联合的**value**成员正确对齐8的倍数地址。 □

5.7.3 使用联合类型

C语言的联合类型与其他语言中的“变体记录”相似。与结构类型一样，联合类型定义几个成员。但与结构类型不同的是，联合中一次只能包含一个成员值，从概念上讲，成员共用分配给联合的存储空间。如果联合的长度很大或者有大量的联合，则可以大大节省存储空间。 162

例 假设根据不同情形，对象可能是整数或浮点数，则可以定义**datum**联合：

```
union datum {
    int i;
    double d;
};
```

然后定义联合类型的变量：

```
union datum u;
```

要把整数存放在联合中，可以使用下列语句：

```
u.i = 15;
```

要把浮点数存放在联合中，可以为另一成员赋值：

```
u.d = 88.9e4;
```

□

引用联合的成员的前提只能是联合的上一次赋值是通过该成员进行的。C语言没有提供查询联合上一次赋值所用成员的方法，编程人员可以记住或对与联合相关联的显式数据标志进行编码。数据标志是与联合相关联的对象，保存联合中当前存储的成员的标记。数据标志和联合可以封装在一个公共结构中。

例 可以将联合

```
union widget { long count; double value; char name[10];} x;
```

换成

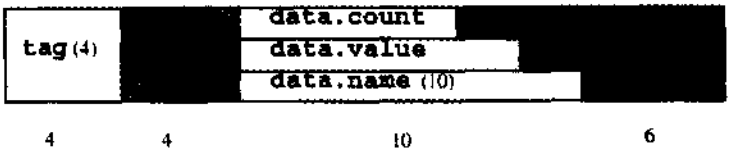
```
enum widget_tag { count_widget,
                  value_widget,
                  name_widget };

struct WIDGET {
```

```
    enum widget_tag tag;
    union { long count;
            double value;
            char name[10]; } data;
} x;
```

163

```
typedef struct WIDGET widget;
```

widget结构的长度是24字节，这里假设**double**类型的对象要在8字节边界上对齐。可能的存储布局如下：



与平常一样，如果**double**类型的对象可以放在4字节边界上，则**widget**的长度只有16字节。

要将一个整数值赋予联合，用下列语句：

```
x.tag = count_widget;
x.data.count = 10000;
```

要将一个浮点数值赋予联合，用下列语句：

```
x.tag = value_widget;
x.data.value = 3.1415926535897932384;
```

要赋值字符串，可以用**strncpy**库函数：

```
x.tag = name_widget;
strncpy(x.data.name, "Millard", 10);
```

下面的可移植函数可以区别联合的可能性。可以调用**print_widget**而不必考虑上一次赋值所用的成员：

```
void print_widget(widget w)
{
    switch(w.tag) {
        case count_widget:
            printf("Count %ld\n", w.data.count); break;
        case value_widget:
            printf("Value %f\n", w.data.value); break;
        case name_widget:
            printf("Name \"%s\"\n", w.data.name); break;
    }
}
```

□

尽管标准C语言没有对联合的存储布局提供什么保证，但对包括很多类似结构类型成员的联合提供了特殊保证。如果这些结构的类型都以相同的成员初始序列开始，则标准C语言保证这些初始序列恰好相互重叠。这样就可以把数据标志放在每个结构开始，用任何结构成员引用这个标志。

参考章节 转换表达式 7.5.1；枚举 5.5；重载 4.2.4；作用域 4.2.1；**switch**语句 8.7；**strncpy**函数 13.3；结构 5.6；**typedef** 5.10

164

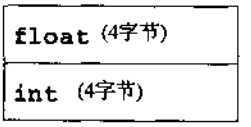
5.7.4 误用联合类型

如果引用联合的某一成员，但联合的上一次赋值不是通过该成员进行的，则是以不可移植

的方式使用联合。编程人员有时用这个方法深入访问C语言类型系统，了解计算机内部的一些底层数据表示方法。

例 要发现浮点数如何表示：

1. 用相同长度的浮点数和整数成员生成一个联合。



2. 对浮点数成员赋值。

3. 读取整数成员值并打印，例如打印成十六进制数。

下列函数就是这么做的，假设**float**与**int**类型具有相同长度：

```
void print_rep(float f)
{
    union { float f; int i } f_or_i;
    f_or_i.f = f;
    printf("The representation of %12.7e is %#010x\n",
        f_or_i.f, f_or_i.i );
}
```

调用**print_rep(1.0)**时，Motorola 68020工作站上的输出如下：

```
The representation of 1.0000000e+00 is 0x003f800000
```

注意，不能用转换运算发现底层表示方法。C语言中的转换运算将操作数变成新表示法中最接近的值，例如**(int) 1.0**的结果是1而不是**0x003f800000**。 □

5.8 函数类型

“返回*T*的函数”类型是函数类型，其中*T*可以是除“……数组”或“返回……的函数”以外的任何类型。换句话说，函数的返回值不能是数组或另一个函数，但可以返回指向数组或指向另一个函数的指针。

只能用两种方法引入函数。第一，函数定义可以生成函数，定义参数和返回值，提供函数体，9.1节提供了函数定义的更多信息。第二，函数声明可以引入在其他地方定义的函数对象的引用。 165

例 下面是**square**的函数定义：

```
int square(int x)
{
    return x*x;
}
```

如果**square**在其他地方定义，则下列声明引入其名称，使其可以调用：

```
extern int square(int);
```

□

外部函数声明可以引用另一C语言源文件中定义的函数或同一源文件在后面定义的函数（即向前引用）。

例 可以用向前引用生成相互递归的函数，如**f**与**g**：

```
extern int f(void);
…
int g(void) { … f(); …}
int f(void) { … g(); …}
```

静态函数也可以使用相同的声明样式：

```
static int f();
…
static int g() { … f(); …}
static int f() { … g(); …}
```

□

一些非标准C语言编译器可能不允许对静态函数进行这种向前引用。有时编译器允许第一个声明使用存储说明符**extern**，在遇到定义时将存储类变成**static**。

例

```
extern int f(void);  /* not really extern, see below... */
…
static int g(void) { … f(); …}
static int f(void) { … g(); …}  /* now, make f static */
```

□

这种编程方式不过是一种误导。标准C语言要求函数的第一个声明（事实上是任何标识符的第一个声明）就指定其为外部函数或静态函数，这样就可以在实现要求用不同方法处理静态函数与外部函数时一遍编译C语言程序。标准C语言没有明确地禁止“先**extern**后**static**”的风格，但没有指定其含义。

166 对函数类型表达式的运算仅限于将其变成函数指针和调用函数。

例 下列声明中，外部标识符**f**、**fp**与**apf**的类型分别是“返回**int**的函数”、“指向返回**int**的函数的指针”和“带**double**参数并返回**int**的函数的指针数组”：

```
extern int f(), (*fp)(), (*apf[])(double);
```

声明**apf**时包括函数的标准C语言原型。要在函数调用表达式中使用这些标识符，可以用下列语句：

```
int i,j,k;
...
i = f(14);
i = (*fp)(j, k);
i = (*apf[j])(k);
```

□

调用没有有效原型的函数时，实际参数采用某些标准转换，但不检查参数类型与个数，即使知道函数的正式参数类型与个数。具有有效原型的函数将参数转换成指定的参数类型。上例中，***apf[j]**指定的函数整数参数**k**转换成**double**类型。

在标准C语言和一些其他实现中，“指向函数的指针”类型的表达式可以在函数调用中使用，而不必显式地取消引用，这时上例中的调用**(*fp)(j,k)**可以写成**fp(j,k)**。

“返回……的函数”类型的表达式在函数调用中不能使用，因为地址运算符**&**的参数或**sizeof**运算符的参数立即转换为“指向返回……的函数的指针”类型（函数是**sizeof**的参数时，不进行转换将导致**sizeof**表达式无效，而不是得到指针长度）。惟一可以产生“返回*T*的函数”类型值的表达式是这种函数名和由一元间接访问运行符*****作用于“指向返回……的函数的指针”类型表达式时得到的间接访问表达式。

例 下列程序对**fp1**与**fp2**赋值相同的指针值：

```
extern int f();
int (*fp1)(), (*fp2)();
fp1 = f;   /* implicit conversion to pointer */
fp2 = &f;  /* explicit manufacture of a pointer */
```
□

调用函数所要的所有信息都包装在“指向返回……的函数的指针”类型对象中。尽管指向 167
函数的指针通常假设为内存中函数代码的地址，但有些计算机上函数指针实际指向调用函数所需的信息块。C语言编程人员通常不需要考虑这种表示问题，只有编译器实现者需要考虑。

参考章节 函数参数转换 6.3.5；函数调用 7.4.3；函数声明符 4.5.4；函数定义 9.1；函数原型 9.2；间接访问运算符* 7.5.7；**sizeof**运算符 7.5.2；普通一元转换 6.3.3

5.9 void类型

void类型没有数值，也没有运算：

void-type-specifier (void类型说明符)：
    void

void类型的作用包括：

- 作为函数返回类型，表示这个函数无返回值；
- 在转换表达式中显式地放弃一个值；
- 建立**void ***类型，这是通用数据指针；
- 代替函数声明符中的参数表，表示函数不带参数。

例 声明**write_line**用**void**作为返回类型和代替参数表。

```
extern void write_line(void);
...
write_line(); /* no value returned */
```

声明**write_line2**表示函数返回一个值，但调用通过转换为**void**类型显式地放弃返回值。

```
extern int write_line2(void);
...
(void) write_line2(...);    /* ignore returned value */
```
□

参考章节 类型转换 7.5.1；放弃表达式 7.13；**void *** 5.3.2

5.10 typedef名称

声明的存储类为**typedef**时，调用类型定义功能。

typedef-name：
    identifier 168

任何声明符中的标识符定义为类型名（**typedef**名称），类型是声明为正常变量声明的标识符类型。将一个名称声明为类型之后，它就可以出现在允许使用声明说明符的任何地方。这样就可以使用复杂类型的助记符缩写。

例 下面是一些声明：

```
typedef int *IP;        /* IP: "pointer to int" */
typedef int (*FP)( ); /* FP: "pointer to function
                         returning int" */
typedef int F(int);     /* F: "function with one int
                          parameter, returning int" */

typedef double A5[5]; /* A5: "5-element array of double" */
typedef int A[];        /* A: "array of int" */
```

有了上述声明之后，就可以进行下列声明：

```
IP ip;       /* ip: pointer to an int */
IP fip();    /* fip: function returning a pointer to int */
FP fp;       /* fp: pointer to a function returning int */
F *fp2;      /* fp2: pointer to a function taking an
                int parameter and returning an int */

A5 a5;       /* a5: 5-element double array */
A5 a25[2];   /* a25: double [2][5]: a 2-element array
                of 5-element arrays of double */
A a;         /* a: array of int (with unspecified bounds) */
A *ap3[3];   /* ap3: 3-element array of pointers to
                arrays of int (with unspecified bounds) */
```

□

例 **typedef**名称不能和其他类型说明符一起使用：

```
typedef long int bigint;
unsigned bigint x;      /* invalid */
```

可以将类型限定符与**typedef**名称一起使用：

```
const bigint x;   /* OK */
```

□

用**typedef**存储说明符声明并不引入新类型，名称是类型的同义词，这个类型可以用其他方式指定。

例 下列声明：

```
typedef struct S { int a; int b; } s1type, s2type;
```

使类型说明符**s1type**、**s2type**以及**struct S**可以相互交换，指向同一类型。 □

169

尽管**typedef**只引入类型的同义词，可以用其他方式命名，但C语言实现可能想在内部保留声明的类型名，以便调试器和其他工具能按编程人员使用的名称引用这个类型。

在C99中，如果**typedef**声明包括变长数组类型，则在处理**typedef**声明时求值数组长度表达式，而不是在**typedef**名声明数组时求值。

例 下列代码段中，数组**a**是10个元素的整数数组，因为编译器是在遇到**typedef**时而不是声明**a**时约束**Array**类型的长度。

```
{
   int n = 10;
   typedef int Array[n];
   n = 25;
   Array a;
   ...
```

```
}
```
□

参考章节 类型兼容性 5.11；变长数组 5.4.5

5.10.1 函数类型的typedef名称

函数类型可以指定**typedef**名称，但函数不能从**typedef**名称继承其函数性，这就对函数**typedef**名称有一定限制。

例 下列声明使**DblFunc**成为“返回**double**的函数”的同义词：

```
typedef double DblFunc();
```

声明之后，**DblFunc**可以用正常的复合声明符规则声明函数类型的指针、函数类型的指针数组等等：

```
extern DblFunc *f_ptr, *f_array[];
```

根据类型声明的正常规则，编程人员不能声明无效类型，如函数数组：

```
extern DblFunc f_array[10];   /* Invalid! */
```

但是，不能用**DblFunc**定义函数。下列**fabs**定义会被拒绝，因为它好像定义了返回值为另一函数的函数：

```
DblFunc fabs(double x)
{
   if (x<0.0) return -x; else return x;
}
```

省略**fabs**后面的括号也不能解决这个问题，因为这里要列出参数。函数定义要按普通方式编写，就像**DblFunc**不存在一样：

```
double fabs(double x)
{
   if (x<0.0) return -x; else return x;
}
```

在标准C语言中，**typedef**名称可以包括函数原型信息（包含参数名）：

```
typedef double DFuncType( double x );
typedef double (*FuncPtr)( int, float );
```

本例中，**DfuncType**是函数类型，而**FuncPtr**是函数指针类型。 □

参考章节 函数声明符 4.5；函数定义 9.1；函数原型 9.2

5.10.2 重定义typedef名称

C语言指定**typedef**名称可以和普通标识符一样在内部块中重新定义。

例

```
typedef int T;
T foo;
...
{
   float T; /* New definition for T */
   T = 1.0;
   ...
}
```
□

170

一个限制是重新声明时不能省略类型说明符，假设默认类型为**int**。一些非ISO编译器在重新声明**typedef**名称时有问题，可能是因为**typedef**名称会给C语言语法带来压力。下面要介绍这个问题。

参考章节　C++中重定义**typedef**名称　5.13.2；名称作用域　4.2.1

5.10.3 实现注意事项

把普通标识符作为类型说明符会使C语言语法对上下文敏感，因此不符合LALR(1)文法。请看下列程序行：

```
A ( *B ) ;
```

如果**A**定义为**typedef**名称，则这一程序行表示声明变量**B**为“指向**A**的指针”类型（“***B**”周
171 围的括号忽略）。如果**A**不是类型名，则该程序行表示调用函数**A**，一个参数为***B**。这种歧义性是无法从语法上解决的。

C语言编译器基于分析器产生器YACC（如可移植C语言编译器）处理这个问题，把语义分析期间获得的信息反馈回词法分析中。所有C语言编译器都要在词法分析期间进行**typedef**处理。

5.11 类型兼容性

C语言中两个类型兼容是指它们属于同一类型或非常接近（在许多方面可以看成同一类型）。类型兼容性的概念是标准C语言中引入的，但主要是用更正式的方式表示传统C语言中使用的规则。还要增加一些规则来处理标准C语言中的特性，如函数原型和类型限定符。要让两个类型兼容，它们或者是同一类型，或者是具有某些属性的指针、函数或数组。下面几节将介绍具体规则。

与每两个兼容类型相关联的是复合类型（composite type），它是从两个兼容类型中产生的公共类型。这与普通二进制转换的方式很相似，先取两个整型类型，将其合并成一个公共结果类型再进行某些算术运算。我们将介绍从两个兼容类型产生的复合类型及类型兼容性的规则。

参考章节　数组类型　5.4；函数原型　9.2；函数类型　5.8；指针类型　5.3；结构类型　5.6；类型限定符　4.4.3；联合类型　5.7；普通二进制转换　6.3

5.11.1 一致类型

两个算术类型兼容当且仅当它们属于同一类型。如果一种类型可以用类型说明符的不同组合编写，则所有替换形式都是同一类型，即类型**short**与类型**short int**是相同的，但类型**unsigned int**与**short**类型是不同的；类型**signed int**与**int**类型相同（与**short**类型和**long**类型也相同），除非这些类型用作位字段类型；类型**char**、**signed char**与类型**unsigned char**是不同的。

任何两个相同类型都是兼容的，其复合类型与原类型是同一类型。在标准C语言中，任何类型限定符都会改变类型；类型**const int**不同于类型**int**，也不与其兼容。**typedef**定义中，声明为某一类型的名称是该类型的同义词，而不是新类型。

例　这些声明之后，**p**和**q**的类型是相同的，**x**和**y**的类型是相同的，但都不同于**u**类型，类型
172 **TS**与类型**struct S**是相同的，**u**、**w**与**y**的类型是相同的。

```
char * p, *q;
struct {int a, b;} x, y;
struct S {int a, b;} u;
typedef struct S TS;
```

```
struct S w;
TS y;
```
□

例 下列声明使类型**my_int**与类型**int**相同，类型**my_function**与类型“**float*()**”相同。

```
typedef int my_int;
typedef float *my_function();
```
□

例 下列声明使变量**w**、**x**、**y**与**z**具有相同的类型。

```
struct S { int a, b; } x;
typedef struct S t1, t2;
struct S w;
t1 y;
t2 z;
```
□

参考章节 整数类型 5.1；指针类型 5.3；结构类型 5.6；**typedef**名称 5.10

5.11.2 枚举兼容性

每个枚举类型定义产生一个新的整数类型。标准C语言要求每个枚举类型与表示这个枚举类型的由实现定义的整数类型兼容。同一程序中不同枚举类型的兼容整数类型可能不同。复合类型是枚举类型。同一源文件中任何两个不同枚举类型都不兼容。

例 下列声明中，类型**E1**与类型**E2**不兼容，但类型**E1**与类型**E3**兼容，因为它们是相同类型。

```
enum e {a,b} E1;
enum {c,d} E2;
enum e E3;
```
□

由于枚举类型通常作为整数类型处理，因此不论枚举兼容性如何，不同枚举类型的值可以任意混合。

例 兼容性规则的作用是，标准C语言会拒绝下面第二个函数声明，因为原型中的参数类型不符合第一个声明：

173

```
extern int f( enum {a,b} x);
extern int f( enum {b,c} x);
```
□

非标准实现有时把枚举类型看成与**int**完全兼容，并且相互间完全兼容。

参考章节 枚举类型 5.5

5.11.3 数组兼容性

两个具有相似限定的数组类型兼容当且仅当它们的元素类型是兼容的。如果两者都指定常量长度，则长度也要相同。但是，如果只有一个数组类型指定常量长度或两者都不指定常量长度，则两个类型是兼容的。两个兼容数组类型的复合类型是数组类型，具有复合元素类型和与之相同的类型限定。如果某个原类型指定常量长度，则复合类型具有常量长度，否则长度不定。如果两个数组用于需要兼容的上下文，则除非运行时维度相同，否则结果不确定。

例 下面的例子说明数组类型的兼容性，**e**是变长数组（C99）。

```
extern int a[];   /* compatible with b, c, and e; not d */
int b[5]; /* compatible with a          and e only */
int c[10]; /* compatible with a         and e only */
const int d[10]; /* not compatible with other types */
```

```
int e[n]; /* compatible with a, b, and c; not d */
```

d的类型与其他类型不兼容，因为它的元素类型**const int**与元素类型**int**不兼容。类型**a**和**b**的复合类型是**int [5]**。运行时，只有**a**的实际定义为长度5时，**a**和**b**相互代替才是定义良好的。□

参考章节 数组类型 5.4；数组声明符 4.5.3；类型限定符 4.4.3；变长数组 5.4.5

5.11.4 函数兼容性

要让两个函数类型兼容，它们应指定兼容的返回类型。如果两个函数类型都按传统（非原型）形式说明，即可满足条件。复合类型是一种传统形式函数类型，具有复合返回类型。

如果两个以原型形式声明的函数兼容，它们应满足下列条件：

1. 函数的返回类型要兼容

174

2. 参数个数和省略号的使用要相符
3. 相应参数的类型要兼容

参数名不一定要相同。复合类型是具有复合参数类型、相同的省略号用法以及具有复合返回类型的函数类型。

如果两个函数类型中只有一个采用原型形式，则要使两个函数类型兼容，它们应满足下列条件：

1. 函数的返回类型要兼容
2. 原型不包括省略号终止符
3. 原型中的每个参数类型*T*应与对*T*采用普通参数转换后得到的类型兼容

复合类型是原型形式的函数类型，具有复合返回值。

参考章节 函数原型 9.2；函数类型 5.8

5.11.5 结构和联合兼容性

在结构类型定义或联合类型定义中类型说明符的每一次出现都引入一个新的结构类型或联合类型，这些类型与同一源文件中任何其他类似类型既不相同，也不兼容。

作为结构类型、联合类型或枚举类型引用的类型说明符与引入到相应定义中的类型相同。类型标志将引用与定义相关联，从这个意义上说，类型标志可以看成是类型名。

例 下面**x**、**y**、**u**的类型各不同，但**u**和**v**的类型相同：

```
struct { int a; int b; } x;
struct { int a; int b; } y;
struct S { int a; int b; } u;
struct S v;
```

□

参考章节 枚举 5.5；结构 5.6；联合 5.7

5.11.6 指针兼容性

两个（类似限定）指针类型兼容的条件是这两个指针指向兼容类型。两个兼容指针类型的复合类型是复合类型的（类似限定）指针。

5.11.7 源文件之间的兼容性

尽管结构类型、联合类型或枚举类型定义引出新的（不兼容）类型，但一定要生成一个漏

175

洞，使同一程序的独立编译源文件之间能够相互引用。

例 假设头文件中包含下列声明：

```
struct S {int a,b;};
extern struct S x;
```

如果程序中两个源文件都导入这个头文件，则两个文件引用同一个变量**x**，其单一类型为**struct S**。但是，每个文件理论上包含不同结构类型的定义，只是在每个实例中刚好都称为**struct S**而已。 □

除非同一类型的两个声明兼容，否则标准C语言指出程序的运行行为是不确定的，因此：

1. 不同源文件中定义的两个结构类型或联合类型兼容的条件是其按相同顺序声明相同成员，每个对应成员具有兼容类型（包括位字段宽度）。在C99中，这个规则进一步严格化，要求结构标志或联合标志相同（或者都省略）。
2. 不同源文件中定义的两个枚举类型兼容的条件是其包含相同的枚举常量（任意顺序），各有相同值。

在这些情况下，复合类型是当前源文件中的类型。

参考章节 枚举类型 5.5；结构类型 5.6；联合类型 5.7

5.12 类型名与抽象声明符

C语言编程中有两种情形需要编写类型名而不声明这个类型的对象：编写类型转换表达式时和对一个类型采用**sizeof**运算符时。在这些情况下，可以使用通过抽象声明符建立的类型名（不要把类型名与“**typedef**名称”混淆）。

type-name（类型名）：
　　declaration-specifiers abstract-declarator$_{opt}$

abstract-declarator（抽象声明符）：
　　pointer
　　pointer$_{opt}$ *direct-abstract-declarator*

pointer：
　　* *type-qualifier-list*$_{opt}$
　　* *type-qualifier-list*$_{opt}$ *pointer*

type-qualifier-list： *(C89)*
　　type-qualifier
　　type-qualifier-list type-qualifier

direct-abstract-declarator：
　　(*abstract-declarator*)
　　direct-abstract-declarator$_{opt}$ [*constant-expression*$_{opt}$]
　　direct-abstract-declarator$_{opt}$ [*expression*] *(C99)*
　　direct-abstract-declarator$_{opt}$ [*] *(C99)*
　　direct-abstract-declarator$_{opt}$ (*parameter-type-list*$_{opt}$)

176

抽象声明符与普通声明符相似，其中的标识符换成空字符串，因此，类型名看上去像是省略了其中的标识符的声明。语法上，*declaration-specifiers*不能包括存储类说明符。只有标准C语言中允许*parameter-type-list*，用于原型形式类型声明。

抽象声明符替换的优先级和普通声明符中相同。

例

类 型 名	转　换
int	类型int
float *	float的指针
char (*)(int)	指向带int参数且返回值为char类型的函数的指针
unsigned *[4]	包含4个指向unsigned类型指针的数组
int (*(*)())()	返回“返回int的函数的指针”的函数的指针

□

类型名总是放在类型转换表达式或sizeof运算符语法中的括号内。如果类型名中的类型说明符是结构类型定义、联合类型定义或枚举类型定义，则标准C语言要求实现在此处定义新类型，包括类型标志（如有）。编程过程中使用这个特性是一种不良的编程风格（在C++中无效）。

例　假设遇到下面两条语句时没有定义struct S（完善的C语言实现应在遇到第一行时发出警告）。

```
i = sizeof( struct S {int a,b;}); /* OK, but strange */
j = sizeof( struct S ); /* OK, struct S is now defined */
```

□

177

参考章节　转换　7.5.1；函数原型　9.2；sizeof运算符　7.5.2

5.13 C++兼容性

5.13.1 枚举类型

一个好的编程习惯是一定要通过显式类型转换把枚举类型或枚举常量作为整数类型来使用。与C语言中不同的是，C++认为枚举类型之间是相互区别的，且不同于整数类型，但可以通过转换表达式相互转换。C++还允许枚举类型隐式转换为整数类型。

例

```
enum e {blue, red, yellow} e_var;
int i_var;
…
i_var = red;          /* valid in both C and C++ */
e_var = 1; /* valid in C, not in C++ */
i_var = (int) red; /* valid in both C and C++ */
e_var = (enum e) 3;   /* valid in both C and C++ */
assert(sizeof(blue) == sizeof(int));
   /* always succeeds in C; may fail in C++ */
```

□

参考章节　枚举类型　5.5

5.13.2 typedef名称

和C语言中一样，typedef名称可以在内部作用域中被重新声明为对象。但是，如果结构或联合中已经使用了原先的typedef名称，则C++不允许在结构或联合中（即作用域）再对这些名称重新声明，实际中很少发生这种情况。

例

```
typedef int INT;
struct S {
   INT i;
```

```
    double INT; /* OK in C, not C++; everywhere a bad idea*/
}
```

□

参考章节 重定义**typedef**名称 5.10.2

5.13.3 类型兼容性

C++没有C语言中类型兼容的概念。要进行更严格的类型检查，C语言允许是兼容类型，而C++则一定要求是一致类型。有时，C++会在类型不一致时发出诊断信息。但是，由于C++提供C语言的“布局兼容性”，因此即使包含未发现的不一致类型而具有标准C语言类型兼容性，C++程序照样可以正确工作。

参考章节 类型兼容性 5.11

178

5.14 练习

1. 下列数值的集合用什么C语言类型表示？假设主要优先考虑的是不同编译器或不同计算机之间的移植性，其次考虑减少空间使用量。

（a）一个5位数的美国邮政服务邮区编码

（b）由3位区号和7位本地号码构成的电话号码

（c）数值0和1

（d）数值－1、0和1

（e）字母字符或数值－1

（f）银行账号结余，以美元和美分为单位，最大为9 999 999.99

2. 一些常用计算机支持扩展字符集，包括普通ASCII字符和其他编码值在128～255之间的字符。假设类型**char**用8位表示。下列**is_up_arrow**函数当输入字符表示向上箭头时返回true，否则返回false。假设定义**UP_ARROW_KEY**中定义了目标计算机中向上箭头的正确编码值，那么这个函数能在不同的标准C语言编译器之间移植吗？如果不能，请改写并使之能够移植。

```
#define UP_ARROW_KEY  0x86
...
int is_up_arrow(char c)
{
   return c == UP_ARROW_KEY;
}
```

3. **vp**的类型为**void ***，**cp**的类型为**char ***，下列哪些赋值语句在标准C语言中有效？

（a）**vp = cp;**　　（c）***vp = *cp;**

（b）**cp = vp;**　　（d）***cp = *vp;**

4. **iv**的类型为**int [3]**，**im**的类型为**int [4][5]**，不用下标运算符，请改写下列表达式：

（a）**iv[i]**

（b）**im[i][j]**

5. 下列**f**函数返回哪个整数值？**return**语句中转换成**int**类型的转换表达式是否有必要？

```
enum birds {wren, robin=12, blue_jay};
int f()
{
   return (int) blue_jay;
}
```

6. 下面是结构类型及该类型变量的定义。写出一系列的语句，对结构中每个成员赋于一个
179 有效值。如果结构的两个成员共用存储单元，则只对其中的一个成员赋值。

```
struct S {
    int i;
    struct T {
        unsigned s: 1;
        unsigned e: 7;
        unsigned m: 24;
    } F;
    union U {
        double d;
        char a[6];
        int * p;
    } U;
} x;
```

7. 用5.6.5和5.7.3节介绍的格式对上题定义的结构画出两个示意图。假设计算机对**char**类型用8位进行字节寻址，对指针和**int**类型用32位进行字节寻址，对**double**类型用64位进行字节寻址。在第一个示意图中，假设计算机用高位存储法，位字段在32位字内从右到左存储；在第二个示意图中，假设计算机用低位存储法，位字段在32位字内从左到右存储。这两种情况下，假设编译器把位字段尽量紧凑存储（高位存储法与低位存储法的具体内容见6.1.2节）。

8. 编写“返回值为指向整数的指针的函数”类型的**typedef**定义。编写一个变量声明，该变量保存指向这种函数的指针，并编写一个这种类型的实际函数，尽量使用**typedef**定义。

9. 编写一个**stdbool.h**头文件，使编程人员能在C89实现中使用具有C99实现风格的布尔类型。是否有什么限制？

10. 改写5.7.3节的数据标志例子，包括**print_widget**函数，使**WIDGET**成为3个结构的联合，每一个结构都包含一个数据标志和一个数据值。改写的实现能否移植到其他的标准C语言符
180 合实现中？

第6章 转换与表示

大多数编程语言都想隐藏特定计算机上的语言实现细节。通常，C语言编程人员也不需要知道这些细节，但C语言的主要吸引力就在于编程人员可以深入到抽象语言层以下，发现程序和数据的底层表示方法。这个自由度是带有一定风险的：有些C语言程序人员可能进入抽象语言层以下，并在他们的程序中对数据表示方法进行了不可移植的假设。

本章有3个目的。第一，讨论数据表示与程序表示的一些特征，以及表示方法的选择如何影响C语言程序。第二，讨论将一种类型数值转换成另一种类型数值的细节，强调不同实现之间可以移植的C语言的特征。最后，介绍C语言的“普通转换规则”，这些转换在求值表达式时自动发生。

6.1 表示

本节讨论函数与数据的表示及表示方法的选择如何影响C语言程序和C语言实现。

6.1.1 存储单元与数据长度

C语言中除位字段之外的所有数据对象都在运行时在计算机内存中表示为整数个数的抽象存储单元。每个存储单元又是由固定数目的位构成，每个位可以用两个值表示，记作0和1。每个存储单元是惟一可寻址的，与**char**类型具有相同长度。存储单元中的位数是C语言中由不同的实现定义的，但要足够大，能够容纳基本字符集中的每个字符的编码。C语言标准还把存储单元称为字节，但字节通常指由8个位构成的存储单元。

根据定义，数据对象的长度就是这个数据对象占用的存储单元数。存储单元是一个字符占用的存储量，因此，**char**类型对象的长度为1。一个字符（字节）中位的数目由**limits.h**文件中的**CHAR_BIT**值指定。

由于一定类型的所有数据对象占用相同的存储量，因此也可以称一个类型的长度为该类型对象占用的存储单元数。可以用**sizeof**运算符确定一个数据对象或一个类型的长度。我们把长度更大的类型称为“更长”或“更大”，同样，我们把长度更小的类型称为“更短”或“更小”。标准C语言对整数类型和浮点数类型要求一定的最小范围，并提供了由实现定义的头文件**limits.h**与**float.h**来定义这两种类型的长度。

例 下列C99程序确定主要C语言数据类型的长度。为了与早期C语言兼容，**%3zd**中的长度修正符**z**应换成**size_t**（**sizeof**类型）的适当修正符：**long**用**l**，**int**不用任何修正符。

```
#include <stdio.h>
int main(void)
{
    printf("\tType sizes:\n");
    printf("char\tshort\tint\tlong\tllong\t"
           "float\tdouble\tldouble\n");
    printf("%3zd\t%3zd\t%3zd\t%3zd\t%3zd\t"
           "%3zd\t%3zd\t%3zd\n",
           sizeof(char), sizeof(short), sizeof(int),
```

```
            sizeof(long), sizeof(long long),
            sizeof(float), sizeof(double),
            sizeof(long double));
    return 0;
}
```

□

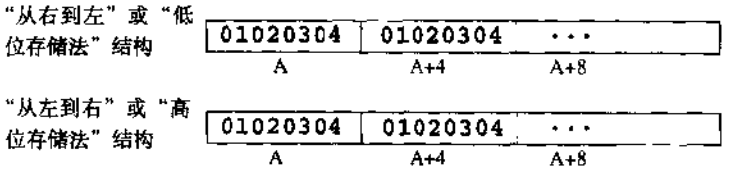
参考章节　字符类型　5.1.3；**float.h**　5.2；**limits.h**　5.1.1；最小整数长度　5.1.1；**sizeof**运算符　7.5.2；**stdio.h**标准I/O　第15章

6.1.2 字节顺序

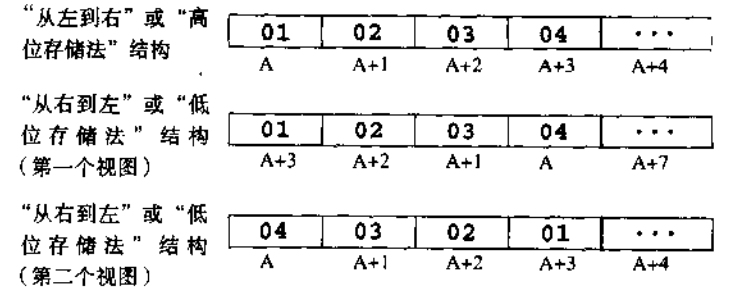
计算机的寻址结构决定指针如何命名不同长度的存储块。C语言的最普通的寻址模型中可以单独寻址计算机内存中每个字符，使用这种模型的计算机称为字节可寻址计算机。大块存储的
182 地址（如保存一个整数和浮点数类型数字的存储块）通常与大单元中第一个字符的地址相同。第一个字符是具有最低地址的字符。

即使在这个简单模型中，计算机的存储“字节顺序”也有所不同，大的存储块中的“第一个”字节不同。在“从右到左”或“低位存储法”结构中，包括Intel 80x86与Pentium微处理器中，用32位表示的整数的地址同时也是这个整数的正序字节地址（高地址字节中存储整数的高位）。而在“从左到右”或“高位存储法”结构中，包括Motorola 680x0微处理器中，用32位表示的整数的地址同时也是这个整数的逆序字节地址（高地址字节中存储整数的低位）。一些嵌入处理器可以根据总体系统需要配置成低位存储法或高位存储法。

例　Intel（低位存储法）和Motorola（高位存储法）体系结构都是字节寻址的，用8位字节和4字节字保存用32位表示的整数。下图显示了每种体系结构上的字顺序，每个字包含32位数值**0x01020304**。可以看出，这两个体系结构在这一层细节上是相同的。



如果看看字中各个字节的内容，则情况有所改变。对于高位存储法，字的地址是最左边（高位）字节的地址。由于字节地址从左向右增加，因此好像与前面写入这个字时相同。但对于低位存储法，字的地址是最右边（低位）字节的地址。这可以从两个方面看，或者字中的地址从右向左增加，或者将字节的顺序倒过来。下面显示了这种情形。



□

结构类型的成员按地址递增顺序分配，即根据计算机的字节顺序从左向右或从右向左分配。

由于位字段也是按字节顺序存储，因此可以按相同规则对存储块中的位编号。这样，在字节顺序为从左到右结构的计算机中，32位整数的最高位（最左边）的位编号为0，最低位编号为31。 183
而在字节顺序为从右到左结构的计算机中，32位整数的最低位（最右边）的位编号为0，最高位编号为31。在其中采用了特定字节顺序的程序是无法移植的。

例 下面的程序按无法移植的方式用联合确定计算机的字节顺序。联合与**long**类型对象具有相同长度，初始化后其低位字节包含1，所有其他字节包含0。在从右到左体系结构中，联合的字符类型成员**Char**与联合的**long**类型成员**Long**的低位字节重叠，而在从左到右体系结构中，联合的字符类型成员**Char**与联合的**long**类型成员**Long**的高位字节重叠：

```
#include <stdio.h>
union {
   long Long;
   char Char[sizeof(long)];
} u;

int main(void)
{
   u.Long = 1;
   if (u.Char[0] == 1)
      printf("Addressing is right-to-left\n");
   else if (u.Char[sizeof(long)-1] == 1)
      printf("Addressing is left-to-right\n");
   else printf("Addressing is strange\n");
   return 0;
}
```

□

6.1.3 对齐限制

一些计算机允许数据对象放在任何地址的存储区域中，不管数据类型如何。而有些计算机则对某些数据类型强加了对齐限制，要求这些类型的对象只占用一定地址。例如，字节寻址计算机通常要求32位（4字节）整数放在4的倍数地址上。这时，我们称这些整数的“对齐系数”为4。不遵守对齐限制可能造成运行错误或非预期的程序行为。即使没有对齐限制，在非对齐地址上使用数据也可能影响性能，因此，为了提高效率，C语言实现可以采用对齐数据。

C语言编程人员通常并没有意识到对齐限制，因为编译器会负责把地址放在适当的地址边界上。但是，C语言使编程人员能够把指针转换成不同类型，从而违反对齐限制。未初始化的指针也可能违反对齐限制。

一般来说，如果类型*S*的对齐要求至少和类型*D*一样严格（即*S*的对齐系数不少于*D*的对齐系数），则把“指向*S*类型的指针”转换成“指向*D*类型的指针”是安全的。这里的安全指得到的*D*类型的指针在用于读取与存储*D*类型对象时能够得到预期的结果，后面转换回原指针类型时能够 184
恢复原指针。一个推论是，任何数据指针都可以转换到**char** *或**void** *类型，然后能够安全地恢复原指针，因为这两种类型具有最不严格的对齐要求。

如果*S*类型的对齐要求没有*D*类型那么严格，则把“指向*S*类型的指针”转换成“指向*D*类型的指针”可能造成两种非预期行为。第一，用得到的指针读取或存储*D*类型对象时可能造成错误，使程序停止。第二，硬件或实现可能将目标指针“调整”为有效指针，通常强制其恢复最近一

次的有效地址，随后转换回原指针时，可能不能够恢复原指针。

参考章节　字节顺序　6.1.2；**malloc**函数　16.1；指针类型　5.3

6.1.4 指针长度

C语言中不要求整型长度很大，使它能够表示指针，但C语言编程人员通常假设**long**类型的长度足够表示指针，这在大多数计算机上是成立的。在C99中，头文件**inttypes.h**可能定义整型类型**intptr_t**与**uintptr_t**，保证整数类型的长度足够大，能用来表示指针。

尽管通常情况下函数指针长度比**void** *指针长度小，但也不完全是这样，见6.1.5节介绍。标准C语言把对象与函数指针之间的所有转换看成没有定义的转换。

参考章节　函数类型　5.8；指针转换　6.2.7；指针类型　5.3；类型长度　6.1.1

6.1.5 寻址模型

本节描述计算机内存设计对C语言编程人员和实现者的一些影响。

内存模型　一些小型专用微处理器设计成指针表示方式的选择涉及时间与空间的权衡，不一定适合所有程序。这些处理器可以利用长地址和短地址。较小的地址（一个段中的地址）更有效，但可以引用的内存量有限。大程序通常要访问多个段。

为了满足不同程序的需要，这些计算机的C语言编译器通常允许编程人员指定内存模型，在程序中权衡时间与空间。表6-1显示了早期PC机上C语言编译器支持的有代表性的内存模型。这些内存模型的变形仍然在一些数字信号处理器中存在。

表6-1　早期PC机上C语言编译器支持的有代表性的内存模型

内存模型名称	数据指针长度	函数指针长度	特　性
微型（tiny）	16位	16位	代码、数据和堆栈共同占用一个段
小型（small）	16位	16位	代码占用一个64KB段，数据和堆栈共用一个64KB段
中型（medium）	16位	32位	代码可以占用多段，数据和堆栈共同占用一段
袖珍（compact）	32位	16位	数据可以占用多段，代码和堆栈各占用一段
大型（large）	32位	32位	代码和数据都可以占用多段，堆栈只占用一段
巨型（huge）（32位平面）	32位	32位	同上，但一个数据项目的长度可以超过64KB

这里有几点值得注意。在所有内存模型中，代码与数据放在不同内存段中，有自己的地址空间。因此，数据指针与函数指针可以包含相同值，即使一个指向对象，一个指向函数。在袖珍与中型内存模型中，数据指针与函数指针具有不同长度。null指针常量**NULL**（见5.3.2节）是一个对象指针，使用时需要小心。表达式中**NULL**的简单用法，包括函数指针，能够正确转换，但将**NULL**作为函数指针参数传递时则可能因为没有原型无法正确工作。要消除这个问题，可以在标准C语言中认真地使用函数原型，使参数正确转换。

例　不熟悉分段体系结构的C语言编程人员可能认为只有数据指针与函数指针都是null指针时才可能包含相同的值，因此就可能错误地使用下列测试。这是行不通的，因为**cp**和**fp**可能指向不同的地址空间，偶尔会出现相同的非null值。

```
char *cp;
```

```
int (*fp)();
…
/* See if cp and fp are both null */
if ((int)cp == (int)fp) …  /* Incorrect!! */
```
□

例 在下列传统C语言例子中，函数**f**的行为在使用袖珍和中型内存模型时是不确定的，因为作为参数传递的null指针是个对象指针而不是函数指针，因此长度不正确。

```
extern int f(); /* no parameter information */
…
f(NULL);    /* This is NOT OK! */
…
int f( int (*fp)() ) { … }
```
□ 185 ~ 186

显式控制指针长度 除了对整个程序使用特定内存模型外，也可以指定特定函数或数据对象是否用“远”指针或“近”指针。这样，编程人员可以避免块间性能影响，但程序的移植性下降，维护难度增加。

例 一些分段结构的C语言编译器定义了变量与指针声明中可以使用的新关键字**__near**与**__far**。语法上，它们可以在标准C语言类型限定符可以使用的地方使用。这些关键字前有两个下划线，因为这些名称是为实现保留的（见10.1.1节）。

```
char __near near_char, *cp;
int __far (*fp)(), big_array[30000]
```

far指针占用32位，而**near**指针占用16位。声明为**far**的函数或数据对象可以放在远程段中，而声明为**near**的函数或数据对象应该放在“根”段中。使用这些语言扩展的编程人员在向没有用原型声明的函数传递指针时应格外小心。 □

数组寻址 不管计算机是否使用分段寻址模式，一些计算机可以在数组长度较小（通常不大于64KB）时更有效地访问数组元素。为了利用大数组，编程人员要提供特殊编译器选项或用某种方式指定大数组。

高难度计算机 尽管C语言已经在许多计算机上有效实现，但有些计算机上的数据和地址表示方法使C语言实现非常困难。如果计算机的自然字长不是自然字节长度的倍数，则可能遇到问题。假设（这是个真实的例子）计算机上的字长为36位，而用7位表示字符，每个字可以放五个字符，还有一位未用。所有非字符数据类型占用一个或几个整字。因为C语言编程要把任何数据结构映射到字符数组，所以这种内存结构对C语言实现者来说非常困难。要复制地址A中类型为T的对象，只要复制从A开头的**sizeof(**T**)**个字符即可。这种计算机上惟一的解决办法是用一些非标准位数（如9或36）表示字符，使其适合字长。这种表示方法会明显地影响性能。

如果字寻址计算机的基本可寻址存储单元大于一个字符，则也会发生类似的问题。在这些计算机上，可能有一种特殊的地址（字节指针）来表示字中的字符。如果有这种字节指针，则其可能大于非字符类型对象的指针，或使用指针中某些被忽略并在其他指针类型中通常被设置为0的位。C实现者要确定是否承担将所有指针表示为字节指针时增加的开销，是否只对**char ***类型（和标准C语言中**void ***类型）的对象使用大格式，或者是否用整字表示每个字符。字符指针长度不同时，要求C语言编程人员对指针转换要更加小心。 187

参考章节 数组类型 5.4；字符类型 5.1.3；函数参数转换 6.3.5；函数原型 9.2；指针类型 5.3；存储单元 6.1.1

6.1.6 类型表示

某种类型数值的表示方法是保存这个类型对象的存储区中的特定位模式，这个模式可以区别对象值与这个类型的其他值。类型的表示方法不一定使用对象中的所有位，有些位可能只是填充位，其数值是未定义的。例如，**short**数据类型可能只用16位，却要在32位字中存储。**sizeof**运算返回的长度中包括填充位。当填充位都被忽略时，范围或精度的说法更正确。

有时一个值在一个类型中可能有多种表示方法。例如，整数中+0和 0各有一种表示方法。实现可以随时随意选择这类等价表示方法。

一种类型的表示方法可能与另一种类型的表示方法不兼容，即使这些类型具有相同长度。如果把**long**类型的数值当作**float**类型的数值来访问，则结果是不确定的，甚至可能使程序停止。

用C99术语来说，对象的有效类型是对象当前使用其表示方法的类型。通常，数据对象（如变量）声明为一定类型，这总是它的有效类型，因此不存在问题。但有时（如使用**malloc**分配的对象时），对象没有声明类型。这时，对象的有效类型就是最后用于把数值存放到对象中的左值表达式的类型。后面访问对象时要使用与有效类型兼容的类型（或兼容类型的限定版本），否则结果是不确定的。将数值复制到没有声明类型的对象中（如用**memcpy**或引用存储对象的底层**char**类型值）时，目标会采用源对象的有效类型。

参考章节 左值 7.1；**malloc** 16.1；**memcpy** 14.3；限定类型 4.4

6.2 转换

下列情况下，C语言可以将一种类型的值转换成其他类型数值：

- 可以用转换表达式显式地将一个值转换成另一类型。
- 准备进行某种算术或逻辑运算时，操作数可以隐式地转换为另一类型。
- 一种类型的对象可以指定另一类型的地址（左值），产生隐式转换。
- 函数的实际参数可以隐式地转换为另一类型之后再进行函数调用。
- 函数的返回值可以隐式地转换为另一类型之后再进行函数返回。

一个对象的哪些类型可以转换是有一定限制的。此外，赋值的转换集不同于类型转换的转换集。

下面几节讨论可能的转换集，然后讨论有哪些转换可以在前面列出的每种情形中实际执行。

6.2.1 表示方法改变

一种类型的值转换成另一种类型可能涉及表示方法改变。例如，如果两个类型的长度不同，则要进行表示方法改变。整数转换成浮点数表示时，即使两者具有相同长度，也会发生表示方法改变。但是，**int**类型的值转换成**unsigned int**类型时，可能不需要表示方法改变。

有些表示方法改变很简单，只要放弃多余位或填上一些0位。而有些表示方法改变可能很复杂，如整数与浮点数表示方法之间的变化。对下面几节介绍的每种转换，我们都会介绍可能需要的表示方法改变。

6.2.2 普通转换

可以将一种类型的值转换成另一种相同或兼容类型的值，这时不会发生表示方法改变。5.11节介绍了类型相同或兼容的情形。

大多数实现不允许将结构类型转换成其他结构类型或将联合类型转换成其他联合类型，因为通常不允许转换成结构类型或联合类型。

6.2.3 转换成整数类型

标量类型（算术类型和指针）可以转换成整数类型。

布尔转换 在C99中，涉及_Bool类型的转换与只涉及其他整数类型的转换稍有不同。将算术值转换成_Bool类型时，如果原值为0，则转换值为0，否则为1。将指针类型转换成_Bool类型时，如果原值为null指针，则转换值为0，否则为1。将_Bool类型转换成算术类型时，结果是0或1，根据目标类型确定。本节其余部分均假设整型类型不是_Bool类型，除非另有说明。 189

从整型类型转换 除了_Bool类型之外，从整型类型转换到另一类型的一般规则是结果的算术值应尽量与源算术值相等。例如，如果无符号整数的值为15，则转换为带符号类型时，得到的带符号整数的值也应为15。

如果不可能用新类型表示对象原值，则有两种情况。如果结果类型是带符号类型，则转换溢出，结果值在技术上是不确定的。如果结果类型是无符号类型，则结果应为新类型的惟一值，等于2^n与原值的模，其中n等于结果类型的表示方法使用的位数。如果带符号整数用对二的补码方法表示，则在相同长度的带符号与无符号整数之间进行转换时，不需要改变表示方法。但如果带符号整数用其他方法表示，如对一的补码表示法或带符号数表示法，则需要改变表示方法。

将无符号整数转换成相同长度的带符号整数时，如果源值太大，无法用带符号方法表示，则转换溢出（即无符号整数的高顺序位为1）。但是，许多编程人员和程序希望转换悄悄完成，无需改变表示方法即可得到负值。

如果目标类型比源类型长，则结果类型中惟一不能表示源值的情况是负的带符号值变成长的无符号类型。这时转换是将源值先转换成与目标类型相同长度的长带符号整数，然后再转换成目标类型。

例 由于常量表达式-1的类型为int，因此：

```
((unsigned long) -1) == ((unsigned long) ((long) -1))
```

□

如果目标类型比源类型短，并且源类型和目标类型都是无符号类型，则可以进行转换，只要从源值中放弃多余的高顺序位。结果表示方法的位模式等于原表示方法的n个低顺序位，其中n是目标类型中的位数。这个放弃规则也适用于将对二的补码形式的带符号整数转换成更短的无符号类型的操作。如果带符号整数也采用对二的补码形式，则放弃规则也是将带符号整数或无符号整数转换成更短的带符号类型的几种可以接受的方法之一。注意，这个规则在发生溢出时并不保持数值的符号，但溢出的操作在任何情况下都是不确定的。如果带符号类型不是用对二的补码形式表示，则转换更加复杂。尽管C语言不要求带符号类型使用对二的补码形式，但使用对二的补码形式表示显然比较好。

如果目标类型为_Bool，则所有非0源值映射为1，只有源值0才转换成0。 190

从浮点数类型转换 从浮点数类型转换成整型值可能产生等于旧对象数值的值。如果浮点值的小数部分为非0值，则小数部分放弃，即转换时通常要把浮点数值截尾。

如果浮点数值无法用新类型表示，则结果是不确定的，例如数值太大或负浮点数值转换成无符号整数类型。上溢与下溢的处理由实现者决定。

从指针类型转换 如果源值为指针而目标类型不是**_Bool**，则把指针当作长度等于指针长度的无符号整数，然后用上面介绍的规则将这个无符号整数转换成目标类型。如果null指针不表示为数值0，则要在null指针转换为整数时显式地将其转换为0。

C语言编程人员习惯于假设将指针转换为**long**类型之后再恢复原类型时不会丢失信息。尽管这种假设通常成立，但不是C语言定义所要求的。在C99中，如果**stdint.h**文件中定义了**intptr_t**与**uintptr_t**类型，则分别是能够保存指针的带符号整数类型和无符号整数类型。问题是有些计算机上的指针表示可能比最大的整数类型更长。

参考章节 **_Bool**类型 5.1.5；字符类型 5.1.3；浮点数类型 5.2；整数类型 5.1；**intptr_t** 21.5；溢出 7.2.2；指针类型 5.3；**uintptr_t** 21.5；**stdint.h** 第21章；无符号类型 5.1.2；**void** *类型 5.3.1

6.2.4 转换成浮点数类型

只有算术类型能转换成浮点数类型。

从**float**类型向**double**类型或从**double**类型向**long double**类型转换时，结果应与源值为相同值。这可以看成是选择浮点数类型表示方法时的限制。

从**double**类型向**float**类型或从**long double**类型向**double**类型转换时，如果源值能在新值可表示的范围之内，结果应为最接近源值的两个浮点数值之一。源值向上或向下舍入是由实现者确定的。

如果源值在目标类型可以表示的数值范围之外（例如**double**类型值太大或太小，无法用**float**类型表示），则结果是不确定的，就像程序中的上溢与下溢行为一样。

从整数类型转换成浮点数类型时，如果整数值可以用浮点数类型表示，则结果就是与之等价的浮点数。如果整数值无法用浮点数准确表示，但在浮点数类型可表示的数值范围之内，则选择两个最接近的浮点数之一作为结果。如果源值在目标类型可以表示的数值范围之外，则结
191 果是不确定的。

复数浮点数类型（C99） 从一个复数类型转换成另一个复数类型时，实数浮点数成员与虚数浮点数成员按照实数浮点数转换规则分别转换。

将一个实数类型（整数或浮点数）转换成复数类型时，复数值的虚数部分为0（+0.0，如果可以表示）。实数类型转换成复数值的实数部分时采用正常的实数浮点数转换规则。

将复数类型转换成实数类型（整数或浮点数）时，虚数部分放弃，实数部分采用正常的实数浮点数转换规则。

_Imaginary类型（如有）是个复数类型，其实数部分总是0。实数类型与虚数类型的相互转换总是得到0，这是它们惟一的共同值。从**_Complex**类型转换为**_Imaginary**类型时，放弃实数部分；从**_Imaginary**类型转换为**_Complex**类型时，将实数部分设置为0。

参考章节 复数类型 5.2.1；浮点类型 5.2；整数类型 5.1；溢出 7.2.2

6.2.5 转换成结构类型或联合类型

不同结构类型之间或联合类型之间不允许进行转换。

参考章节 结构类型 5.6；联合类型 5.7

6.2.6 转换成枚举类型

转换成枚举类型的规则与转换成整数类型的规则相同。一些转换（如枚举与浮点类型之间）

可能是不好的编程风格。

参考章节 枚举类型 5.5

6.2.7 转换成指针类型

一般来说，指针和整数可以转换成指针类型。特殊情况下，数组与函数也可以转换成指针类型。

任何类型的null指针可以转换成任何其他类型的指针类型，仍然能够识别为null指针。转换时可能改变表示方法。

对任何类型的*S*与*D*，“*S*指针”类型的值可以转换成“*D*指针”类型。在标准C语言中，对象指针与函数指针不能相互转换。但是，表示方法改变或实现中的任何对齐限制都可能影响结果指针的行为。

整型常量0或数值为0的任何整型常量或转换成**void** *类型的常量都是null指针常量，并且可以转换成任何其他类型的指针类型。这种转换得到的null指针不同于任何有效指针。不同指针类型的null指针可能有不同的内部表示方法，null指针不一定要把所有位设置为0。 192

常量0以外的整数可以转换为指针类型，但结果是不可移植的。其目的是把指针看成无符号整数（与指针长度相同），这样就可以用标准整数转换将源类型变成目标类型。

要把“*T*数组”类型的表达式转换为“*T*指针”类型的值，可以替换数组第一个元素的指针。这是在普通一元转换时进行的（见6.3.3节）。

“返回*T*的函数”类型的表达式（即函数指定符）转换成“返回*T*的函数的指针”类型数值时，用函数指针替换。这是在普通一元转换时进行的（见6.3.3节）。

参考章节 对齐限制 6.1.3；数组类型 5.4；函数调用 7.4.3；函数说明符 7.1；整数类型 5.1；指针类型 5.3；**sizeof**运算符 7.5.2；普通一元转换 6.3.3

6.2.8 转换成数组类型或函数类型

任何类型都不能转换成数组类型或函数类型。

例 特别地，不能在数组类型之间和函数类型之间进行转换：

```
extern int f();
double d;
d = (( double () )f) ();      /* Invalid! */
d = (double) f();             /* OK */
d = (*(double (*)()) f)();
          /* Valid, but will have unexpected results */
```

在上述语句中，**f**的地址转换成返回**double**类型的函数指针，然后这个指针取消引用并调用函数。这是有效的，但存储在**d**中的结果值可能是无用数据，除非**f**实际定义为返回**double**类型值，而不是外部声明。 □

6.2.9 转换成void类型

任何值都可以转换成**void**类型。这种转换只能用于要放弃表达式值时，如表达式语句中。

例 将表达式转换成**void**类型的最常见用法是忽略函数调用结果。例如，调用**printf**函数，将信息写入标准输出流中，其返回一个出错指示，但这个出错指示通常被忽略。没有必要将结果转换成**void**类型，转换成**void**类型的目的就是要让读者知道编程人员故意要忽略函数调用结果。 193

```
(void) printf("Goodbye.\n");
```
□

参考章节　放弃表达式　7.13；表达式语句　8.2；**void**类型　5.9

6.3 普通转换

6.3.1 类型转换

本章前面介绍的所有转换都可以用类型转换表达式显式地进行，而不会发生错误。表6-2总结了可以进行的类型转换。注意标准C语言不允许函数指针直接转换成对象指针或对象指针直接转换成函数指针，但可以通过合适的整数类型进行转换。这个限制反映了对象指针与函数指针的表示方法可以不同。

表6-2　可以进行的类型转换

目标类型	允许的源类型
任何算术类型	任何算术类型
任何整数类型	任何指针类型
对象T的指针或(**void** *)	(a) 任何整数类型
	(b) (**void** *)
	(c) 对象Q的指针，对任意Q
	(d) 函数Q的指针，对任意Q①
函数T的指针	(a) 任何整数类型
	(b) 函数Q的指针，对任意Q
	(c) 对象Q的指针，对任意Q①
结构类型与联合类型	无，不允许转换
T的数组，或返回T的函数	无，不允许转换
void	任何类型

① 在标准C语言中不允许。

类型限定符的存在与否并不影响类型转换的有效性，有些转换可以越过类型限定符。允许的赋值转换限制更严。

标准C语言保证对象指针转换为**void** *和返回源类型时能保持源值，这对其他C语言实现中通过**char** *的转换通常成立。

参考章节　赋值转换　6.3.2；类型转换表达式　7.5.1；类型限定符　4.4.3；**void** *　5.3.1

195

6.3.2 赋值转换

在简单赋值表达式中，赋值运算符左边和右边的表达式类型应当相同，否则要把赋值运算符右边的值转换成左边的类型。表6-3列出了有效的转换，是类型转换的子集。除非另有指定，否则ISO类型限定符的存在与否并不影响转换的有效性，但赋值运算符左边不能使用**const**限定的左值表达式。

表6-3　允许的赋值转换

左边类型	允许的右边类型
任何算术类型	任何算术类型
_Bool（C99）	任何指针类型
结构类型或联合类型①	兼容的结构类型或联合类型

（续）

左边类型	允许的右边类型
(void *)②	(a) 常量0 (b) 对象T_1的指针③ (c) (void *)
对象T_1的指针②③	(a) 常量0 (b) 对象T_2的指针，其中T_1与T_2兼容 (c) (void *)
函数F_1的指针②	(a) 常量0 (b) 函数F_2的指针，其中F_1与F_2兼容

① 一些早期的C语言编译器不支持赋值结构类型和联合类型。

② 左边的引用类型应当有右边的引用类型的所有限定符。

③ 如果其他指针具有void *类型，则T_1可能是不完整类型（标准C语言）。

符合ISO标准的实现不允许其他没有显式类型转换表达式的转换，但传统C语言编译器几乎总是允许对混合指针类型的赋值，通常还允许对符合类型转换要求的任何类型赋值。

指针赋值的规则隐含了类型限定符的条件，因为可以用赋值越过限定。将指针赋值为_Bool类型过程中，在指针为null时赋值为0，否则赋值为1。

参考章节 赋值运算符 7.9.1；类型转换 6.3.1；兼容类型 5.11

6.3.3 普通一元转换

普通一元转换在进行运算之前决定是否转换和如何转换单个操作数，目的是将算术类型的大数简化为必须经过运算符处理的较小的数。一元运算符!、-、+、~与*的操作数自动采用普通一元转换，二元运算符<<与>>的操作数需要单独采用普通一元转换。 195

转换阶 C99中增加了标准整数类型，使实现能够扩展类型集，因此很难准确而简洁地描述这些隐式转换。C99标准引入了转换阶的概念，有助于解释转换，我们也采用这个概念。对于C89，只要忽略long long类型、_Bool类型和扩展整数类型即可。对传统C语言，见本节稍后的介绍。

转换阶是对每个整数类型指派的数字值，以指定其转换顺序。表6-4列出了标准整数类型的转换阶、表中没有显示枚举类型，但其与底层整数类型具有相同的阶。

表6-4 转换阶

转换阶	具有这种转换阶的类型
60	long long int, unsigned long long int (C99)
50	long int, unsigned long int
40	int, unsigned int
30	short, unsigned short
20	char, unsigned char, signed char
10	_Bool

转换阶使用的特定数字并不重要，但标准类型应有表中所示的相对数字顺序。这里没有选择连续数字，因为C语言实现可以在这个表中插入自己的扩展整数类型，其阶数可能在标准类型之间。扩展类型阶应符合下列规则：低于更高精度的类型的转换阶；低于任何相同精度的标准类型的转换阶；两个不同的带符号整数类型不能有相同阶；无符号类型与同一表示方法的带符

号类型应当具有相同阶。

对于上述转换阶，表6-5显示了它们的普通--元转换，采用表中第一个适用的转换。如果没有适用的转换规则，则不进行转换。整数采用的一元转换称为整数升级（integer promotion）。数组类型与函数类型的转换有时可以绕过，6.2.7节介绍了例外情况。

例　如果s是标准C语言中unsigned short类型变量，取值为1，则如果short类型的范围小于int类型范围，则表达式(-s)的类型为int，取值为－1，而如果short类型的范围等于int类型范围，则表达式(-s)的类型为unsigned，取值为一个大的正数。这是因为，在第一个实例
196 中，s升级为int类型之后再采用一元负号运算符，而第二个实例中s升级为类型unsigned。 □

表6-5　普通一元转换（采用第一个适用的转换）

操作数的类型	标准C语言转换	传统C语言转换
float	（无转换）	double
T的数组	T的指针	（与标准C语言相同）
返回T的函数	返回T的函数的指针	（与标准C语言相同）
阶大于或等于int的整数类型①	（无转换）	（与标准C语言相同）
阶小于int的带符号类型	int	（与标准C语言相同）
阶小于int的无符号类型，所有值可以用int类型表示	int	unsigned int
阶小于int的无符号类型，所有值无法用int类型表示	unsigned int	（与标准C语言相同）

① int signed int与unsigned int类型的位字段假设转换阶小于int类型，即其转换类型取决于所有值可以用int类型表示与否。

int、signed int与unsigned int类型的位字段假设转换阶小于int类型。

传统C语言实现用不同方法进行这些转换。首先，所有低转换阶的无符号类型转换成unsigned int类型，从而保留操作数的符号性，即使不保留其数值（编程人员要注意标准C语言转换，因为升级结果的符号性是与实现相关的，可能影响周围表达式的含义）。第二，float类型转换成double类型，减少所需的浮点数库函数，但可能会影响性能。这种权衡不再是强制的，但实现还可以继续进行升级。

数组转换与函数转换　普通一元转换指定数组类型值转换成数组第一个元素的指针，除非：

1. 数组是sizeof运算符或&地址运算符参数
2. 用字符串字面值初始化字符数组
3. 用宽字符串直接数初始化wchar_t类型数组

在C99中，任何数组类型的值都发生这个转换。在C99之前，只对数组类型的lvalue表达式进行转换。

例

```
char a[] = "abcd"; /* No conversion */
char *b = "abcd";  /* Array converted to pointer */
int i = sizeof(a); /* No conversion; size of whole array */
b = a + 1;         /* Array converted to pointer. *          /
```
197 □

普通一元转换指定函数指定符转换成函数指针，除非这个指定符是**sizeof**运算符或&地址运算符的操作数（如果是**sizeof**的操作数，则其无效）。

例

```
extern int f();
int (*fp)();
int i;
fp = f;          /* OK, f is converted to &f */
fp = &f;         /* OK, implicit conversion suppressed */
i = sizeof(fp);  /* OK, result is the size of the pointer */
i = sizeof(f);   /* Invalid */
```

□

参考章节 按位反运算符~ 7.5.5；扩展整数类型 5.1.4；函数调用 7.4.3；函数指定符 7.1；间接访问运算符* 7.5.7；初始化语句 4.6；逻辑非运算符! 7.5.4；lvalue 7.1；移位运算符<<与>> 7.6.3；**sizeof** 7.5.2；一元负号运算符 - 7.5.3；宽字符串 2.7.4

6.3.4 普通二元转换

两个值组合操作时，首先根据通常的普通二元转换转换成单个公共类型，这个公共类型通常是结果的类型。转换适用于大多数二元运算符的操作数和条件表达式的第二个和第三个操作数。普通一元转换和普通二元转换一起称为*普通算术转换*。

进行普通二元转换的运算符首先对每个操作数独立进行普通一元转换，将短值加宽，将数组和函数变成指针。然后如果某个操作数不是算术类型或两者具有相同算术类型，则不再进行转换，否则对两个操作数采用表6-6中第一个适用的转换。这个表假设两个操作数都不是复数，关于复数操作数的处理，见后面的讨论。

例 标准C语言规则与传统规则在同时遇到**long**操作数与**unsigned**操作数时有所不同（如果**long**严格大于**unsigned**）。下列程序确定发生何种转换：

```
unsigned int UI = -1;
long int LI = 0;
int main()
{
    if (UI < LI) printf("long+unsigned==long\n");
    else printf("long+unsigned==unsigned\n");
    return 0;
}
```

□ 198

表6-6 普通二元转换（采用第一个适用的转换）

其中一个操作数的类型[1]	另一个操作数的类型[1]	标准C语言转换	传统C语言转换
long double	任何实数类型	**long double**	不适用
double	任何实数类型	**double**	（与标准C语言相同）
float	任何实数类型	**float**	**double**
任何无符号类型	任何无符号类型	阶较大的无符号类型	（与标准C语言相同）
任何带符号类型	任何带符号类型	阶较大的带符号类型	（与标准C语言相同）
任何无符号类型	阶较小或相等的带符号类型	无符号类型	（与标准C语言相同）

（续）

其中一个操作数的类型[1]	另一个操作数的类型[1]	标准C语言转换	传统C语言转换
任何无符号类型	阶较大的带符号类型，能够表示所有无符号类型的值	带符号类型	带符号类型的无符号版本
任何无符号类型	阶较大的带符号类型，不能够表示所有无符号类型的值	带符号类型的无符号版本	（与标准C语言相同）
任何其他类型[2]	任何其他类型	（不转换）	（与标准C语言相同）

① 规则假设两个操作数已经采用普通一元转换。

② 关于复数操作数的处理，见下面介绍。

复数类型与普通二元转换　在C99中，普通二元转换要考虑复数类型。在混合实数/复数运算中，实数类型的操作数并不转换成复数类型，目的是为了提高性能，但是，要通过转换把两个操作数变成等价的浮点数精度。这样，运算在处理混合实数/复数操作数时就好像实数类型的操作数已经转换成复数类型（当然，实现也可以实际将实数类型的操作数转换成复数类型）。操作的结果类型是转换之后的复数操作数类型。

具体地说，如果两个操作数都是复数，则较短的操作数转换成较长操作数的类型，这就是结果类型，对应于组合两个浮点数实数操作时的情形。

如果一个操作数是复数而另一操作数是整数，则整数操作数转换成对应于复数类型的实数浮点数类型。例如，如果复数操作数的类型为**float_Complex**，则整数转换成**float**，结果是复数类型。

如果一个操作数是复数而另一操作数是实数浮点数类型，则在实数域或复数域中精度较低的类型转换成另一类型的精度。例如，组合**float**类型与**double_Complex**类型时，**float**类型操作数升级为**double**类型。组合**long double**类型与**double_Complex**类型时，
199 **double_Complex**类型升级为**long double_Complex**类型。

6.3.5 默认函数参数转换

如果表达式是不用原型控制的函数调用中的参数，或表达式显示为原型参数表中“**...**”部分的参数，则先转换表达式的值之后再传递到函数中。这个默认函数参数转换与普通一元转换相似，只是**float**类型的参数总是升级为**double**类型，即使在标准C语言中也是如此。

如果调用函数受到原型控制，则参数不一定进行普通整数升级，**float**类型的参数不一定升级为**double**类型。实现可以随意进行这些转换，但这些规则使实现可以优化调用序列。数组与函数要转换成指针。

在C99原型中，如果数组类型的正式参数在方括号中具有类型限定符列表*L*，则实际数组参数转换成表示类型的*L*限定指针。详见9.3节介绍。

float类型到**double**类型的参数转换有助于传统C语言的早期版本与标准C语言对库函数个数的控制，因为有了这一转换就不需要一个版本同时具有**float**类型和**double**类型。C99定义了**float**类型和**long double**类型以及**double**类型的完整数学函数集。

参考章节　数组限定符列表　4.5.3；函数调用　7.4.3；数学函数　第17章；原型　9.2；普通一元转换　6.3.3

6.3.6 其他函数转换

函数正式参数的声明类型及其返回值类型随函数参数转换做相应的调整，见9.4节介绍。

6.4 C++兼容性

赋值转换

在C++中，要通过类型转换把**void** *类型指针转换成另一种类型指针。C语言中也可以使用这种类型转换，但在赋值时不需要。

例 **malloc**函数返回新分配内存区的**void** *指针：

```
#include <stdlib.h>
char * cp;
const int SIZE = 10 * sizeof(char);
...
cp = malloc(SIZE);              /* OK in C, not C++ */
cp = (char *) malloc(SIZE);     /* OK in both */                    □ 200
```

只有未限定类型（不是**const**类型或**volatile**类型）对象的指针不必进行类型转换而转换成**void** *类型指针。

例

```
char * cp;
const char * const_cp;
void * vp;
...
vp = cp;                    /* valid in both C and C++ */
vp = const_cp;              /* valid in C, not in C++  */
vp = (void *) const_cp;     /* valid in both C and C++ */           □
```

参考章节 赋值转换 6.3.2

6.5 练习

1. 下表列出类型转换中使用的源类型和目标类型对。标准C语言中允许哪些转换？传统C语言中允许哪些转换？（对传统C语言，将**void**换成**char**）

	目标类型	源类型
(a)	**char**	**int**
(b)	**char ***	**int ***
(c)	**int (*f)()**	**int ***
(d)	**double ***	**int**
(e)	**void ***	**int (*f)()**
(f)	**int ***	**t *** （其中**typedef int t**）

2. 在上题的表中，标准C语言中允许哪一对进行赋值转换？传统C语言呢？（左边为目标类型，右边为源类型）

3. 对下列类型对采用传统C语言的普通二元转换时，结果类型是什么？标准C语言在哪些情况下具有不同结果？

(a) **char**与**unsigned**　　(d) **char**与**long double**

(b) **unsigned**与**long**　　(e) **int []**与**int ***

(c) **float**与**double**　　(f) **short ()**与**short(*) ()**

4. C语言实现中能否用**char**类型表示 -2147483648 ~ 2147483647的值？如果可以，那么在这个实现中**sizeof(char)**的值是多少？**int**类型的最小与最大范围是多少？

5. **sizeof(long double)**与**sizeof(int)** 之间要有什么关系？

6. 假设计算机A和B都是字节寻址的，字长为32位（4字节），但计算机A采用高位存储法，计算机B采用低位存储法。要想将整数128存放在计算机A的字中，然后传输到计算机B的字中，具体做法是，将A中字的第一个字节移到B中字的第一个字节，依此方法移动其他字节。传输完成后，计算机B的字中存储的是什么值？如果计算机B采用高位存储法，计算机A采用低位存储法，结果又将如何？

201

第7章 表 达 式

C语言中具有丰富的运算符，可以访问基础硬件提供的大多数运算。本章介绍表达式语法并介绍每个运算符的功能。

7.1 对象、左值与指定符

对象是可以检查和存放的内存区。*左值*（lvalue）是引用对象的表达式，可以检查或改变对象。赋值语句的左边只能使用左值表达式。非左值表达式也称为*右值*（rvalue），只能放在赋值语句右边。左值可以是对象或不完整类型，但不能是**void**类型。

由于标准C语言使用左值，因此左值不一定允许修改其所指的对象。如果左值为数组类型、不完整类型、**const**限定类型，或者具有结构类型或联合类型，其成员（递归适用于嵌套结构或联合）具有**const**限定类型，则会如此。*可修改的左值*（modifiable lvalue）表示左值允许修改其所指的对象。

*函数指定符*是函数类型的值，不是对象也不是左值。函数名是个函数指定符，函数指针取消引用的结果也是函数指定符。函数与对象在C语言中通常采用不同处理方法，我们要认真区别函数类型与对象类型、左值与函数指定符以及函数指针与对象指针。“指定对象的左值”一词是多余的，但有时为了强调了不包括函数指定符，我们还是使用这种说法。

表7-1 可以作为左值的非数组表达式

表达式	附加条件
名称	名称应为变量
e[*k*]	无
(*e*)	*e*应为左值
e.*name*	*e*应为左值
e->*name*	无
**e*	无
字符串型常量	无

表7-1列出了可以作为左值的C语言表达式，以及表达式成为左值时要满足的所有特殊条件。任何其他形式的表达式都不能产生左值，除字符串字面值之外，列出的其他表达式如果是“……数组”类型时也不能成为左值。不能成为左值的表达式包括：数组名、函数、枚举常量、赋值、类型转换和函数调用。

表7-2列出了要求某个操作数为左值的运算符。

表7-2 要求某个操作数为左值的运算符

运 算 符	要 求
&(一元)	操作数应为左值或函数名
++ --	操作数应为左值（前缀或后缀形式）
= += -= *= /= %= <<= >>= &= ^= |=	左操作数应为左值

参考章节　地址运算符　7.5.6；赋值表达式　7.9；类型转换表达式　7.5.1；成员选择　7.4.2；自减表达式　7.4.4, 7.5.8；枚举　5.5；函数调用　7.4.3；自增表达式　7.4.4,7.5.8；间接访问表达式　7.5.7；字面值　2.7,7.3.2；名称　7.3.1；字符串型常量　2.7.4；下标　7.4.1

7.2 表达式与优先级

本章介绍的表达式语法完整指定了C语言中运算符的优先级。表7-3按优先级从高到低列出了C语言中的运算符及其结合律。

203
~
204

表7-3　C语言运算符（按优先级从高到低）

记　号	运 算 符	类	优先级	结 合 律
名称，字面值	简单记号	主	16	无
a[*k*]	下标	后缀	16	从左到右
f(...)	函数调用	后缀	16	从左到右
.	直接选择	后缀	16	从左到右
->	间接选择	后缀	16	从左到右
++ --	自增、自减	后缀	16	从左到右
(*type name*) {*init*}	复合字面值（C99）	后缀	16	从左到右
++ --	自增、自减	前缀	15	从右到左
sizeof	长度	一元	15	从右到左
~	按位反	一元	15	从右到左
!	逻辑非	一元	15	从右到左
- +	算术负、正	一元	15	从右到左
&	地址	一元	15	从右到左
*	间接访问	一元	15	从右到左
(*type name*)	类型转换	一元	14	从右到左
* / %	乘法、除法、求余	二元	13	从左到右
+ -	加法、减法	二元	12	从左到右
<< >>	左移、右移	二元	11	从左到右
< > <= >=	关系	二元	10	从左到右
== !=	相等/不等	二元	9	从左到右
&	按位与	二元	8	从左到右
^	按位异或	二元	7	从左到右
|	按位或	二元	6	从左到右
&&	逻辑与	二元	5	从左到右
||	逻辑或	二元	4	从左到右
? :	条件	三元	3	从右到左
= += -= *= /= %= <<= >>= &= ^= |=	赋值	二元	2	从右到左
,	顺序求值	二元	1	从左到右

7.2.1 运算符优先级与结合律

C语言中每个表达式运算符都有一个优先级和结合律规则。如果不用括号显式地表示运算符的操作数组，则具有更高优先级的运算符将组合操作数。如果两个运算符具有相同优先级，则

操作数根据结合律为左结合或右结合，与左边或右边的运算符结合组成。具有相同优先级的所有运算符总是具有相同的结合律。 205

优先级和结合律规则确定表达式的含义，但不指定运行时大表达式或语句中的子表达式求值的顺序。7.12节将介绍求值的顺序。

例 下面是一些优先级和结合律规则的例子：

原表达式	等价表达式	等价原因
`a*b+c`	`(a*b)+c`	*的优先级比+高
`a+=b|=c`	`a+=(b|=c)`	+=与|=是右结合的
`a-b+c`	`(a-b)+c`	-与+是左结合的
`sizeof(int)*p`	`(sizeof(int))*p`	sizeof的优先级比类型转换高
`*p->q`	`*(p->q)`	->的优先级比*高

□

要总结结合律规则，除赋值运算符是右结合外，二元运算符都是左结合的。条件运算符是右结合的。一元与后缀运算符有时为右结合，但这只表明表达式`*x++`解释为`*(x++)`而不是`(*x)++`。我们更倾向于后缀运算符的优先级高于（前缀）一元运算符。

参考章节 赋值运算符 7.9；二元运算符 7.6；字符串连接 2.7.4；条件运算符 7.8；后缀运算符 7.4.4；一元正号运算符 7.5.3

7.2.2 溢出和其他算术异常

对C语言中的某些运算，如加法和乘法，运算的算术结果可能无法表示成所要结果类型的值（由普通转换规则确定）。这个条件称为溢出，包括上溢和下溢。

一般来说，C语言没有指定溢出的结果，一种可能是产生不正确的值（类型正确），另一种可能是程序终止执行，还有一种可能是发生某种机器相关中断或异常，程序按某种实现相关的方式探测。

对有些运算，C语言显式指定某些操作数值的结果是无法预测的，或总是产生一个值，但这个值对某些操作数值是无法预测的。如果除法运算符/或求余运算符%的除数为0，则结果是无法预测的。如果移位运算符<<或>>右边的操作数太大或是负数，则产生无法预测的值。 206

传统上，C语言的所有实现忽略带符号整数溢出的问题，结果是实现运算的机器指令产生的某个值（许多使用对二的补码表示带符号整数的计算机处理加法和减法的溢出时，就是产生真正对二的补码结果的低顺序位。无疑，许多现有C语言程序利用了这个事实，但这种代码在技术上是不可移植的）。浮点数上溢和下溢通常以机器支持的方便方式处理，如果机器体系结构提供多种处理异常浮点数条件的方式，则可能提供库函数，让C语言编程人员可以访问这些选项。

对无符号整数，C语言在溢出问题上相当明确。无符号整数的每个运算总是产生一个结果值，这是运算的真正数学结果的2^n模数（其中n是表示无符号结果的位数），这等于计算真正结果的低顺序n位值（对负结果为对二的补码结果，例如用小数减去大数时）。

例 假设无符号类型对象用16位表示，则无符号值4减无符号值7得到无符号值65533（即$2^{16}-3$），因为这个值是真正数学结果(-3)的2^{16}模数。 □

这个规则的一个重要后果是，无符号整数的运算保证能在两个实现之间完全可移植，只要这些实现使用的表示方法具有相同位数。利用更少的位数很容易模拟另一个实现的无符号算术。

参考章节　除法运算符/　7.6.1；浮点数类型　5.2；求余运算符%　7.6.1；移位运算符<<和>>　7.6.3；带符号类型　5.1.1；无符号类型　5.1.2

7.3 主表达式

主表达式有3种：名称（标识符）、字面值常量和括号表达式：

primary-expression（主表达式）：
　　identifier（标识符）
　　constant（直接数常量）
　　parenthesized-expression（括号表达式）

C语言中传统上把函数调用、下标表达式和成员选择表达式都看成主表达式，但我们把这些
207 放到下节后缀表达式中介绍。

7.3.1 名称

名称的值取决于它的类型。名称的类型由这个名称的声明确定（如有），见第4章介绍。

变量名声明为算术、指针、枚举、结构和联合类型时，求值为这个类型的对象，名称是个左值表达式。枚举常量名求值为相关联的整数值，不是左值。

例　3种颜色名是枚举常量。**switch**语句（见8.7节）根据参数**color**的值选择要执行的三条语句之一。

```
typedef enum { red, blue, green } colortype;

colortype next_color(colortype color)
{
    switch (color) {
        case red    : return blue;
        case blue   : return green;
        case green  : return red;
    }
}
```

□

数组名求值为这个数组，是左值，但不能修改。除非数组是**sizeof**的参数、地址运算符&的参数或要用字符串常量初始化的字符数组，否则数组值在普通一元转换中转换成指向数组中第一个对象的指针。

例　数组是**sizeof**的参数时，并不将数组名转换为指针，因此结果是数组长度，而不是指针长度。

```
extern void PrintMatrix();
int Matrix[10][10], total_length, row_length;

total_length = sizeof Matrix;
row_length = sizeof Matrix[0];
PrintMatrix(Matrix); /* pointer to first
                        element is passed */
```

□

函数名求值为这个函数，不是左值。除非函数名是**sizeof**的参数或地址运算符**&**的参数，否则这个名称在普通一元转换中转换成指向函数的指针。**&f**的结果是指向**f**的指针，而不是指向已经指向**f**的指针的指针，**sizeof(f)**无效。

例 本例用函数名作为另一函数的参数。 208

```
extern void PlotFunction(double (*f)(double),
                         double x0, double x1);

double fn(double x) { return x * x - x; }

int main(void)
{  ...
   PlotFunction(fn, 0.01, 100.0); /* fn converts to &fn */
   ...
}
```

□

名称作为表达式不能引用标号、**typedef**名称、结构成员名、联合成员名、结构标志、联合标志或枚举标志。这些地方使用的名称与表达式中的名称可以引用的名称属于不同命名空间。其中一些名称可以通过特殊结构在表达式中引用。例如，结构成员名和联合成员名可以用**.**或**->**选择运算符引用，**typedef**名称可以在转换时作为**sizeof**运算符的参数。

参考章节 数组类型 5.4；类型转换 7.5.1；枚举类型 5.5；函数调用 7.4.3；函数类型 5.8；左值 7.1；命名空间 4.2；**.**或**->**选择运算符 7.4.2；**sizeof**运算符 7.5.2；**typedef**名称 5.10；普通一元转换 6.3.3

7.3.2 字面值

字面值（词法常量）是个数字常量，在求值为表达式时得到这个常量值。除了字符串常量之外，字面值表达式不是左值。字面值及其类型和数值见2.7节介绍。

7.3.3 括号表达式

括号表达式包括左括号、表达式和右括号：

parenthesized-expression (括号表达式):
(*expression*)

括号表达式的类型与其中的表达式类型相同，不进行任何转换。括号表达式的值是其中的表达式值，只有在其中的表达式为左值时可以为左值。括号不一定强制特定求值顺序（见7.12节）。

括号表达式的目的是组合时隔离其中的表达式，避开运算符默认优先级或提高代码可读性。 209

例 `x1 = (-b + discriminant)/(2.0 * a);` □

参考章节 左值 7.1

7.4 后缀表达式

后缀表达式有6种：下标表达式、两种成员选择（直接选择与间接选择）表达式、函数调用表达式和后缀自增表达式与后缀自减表达式。

postfix-expression (后缀表达式):
　　primary-expression
　　subscript-expression
　　component-selection-expression
　　function-call
　　postincrement-expression
　　postdecrement-expression
　　compound-literal　　(C99)

C语言中传统上把函数调用、下标表达式和成员选择表达式都作为主表达式，但其语法与后缀表达式更接近。

7.4.1 下标表达式

下标表达式由一个后缀表达式、左方括号、表达式和右方括号组成。这个结构用于数组下标，后缀表达式（通常是数组名）求值为指向数组开始对象的指针，另一表达式求值为整数偏移量：

subscript-expression (下标表达式):
　　postfix-expression [*expression*]

在C语言中，按照定义，表达式$e_1[e_2]$准确等价于表达式*((e1)+(e2))。两个操作数采用普通二元转换，结果总是左值。间接访问运算符*的操作数应为一个指针，要使+运算符的结果成为指针，惟一的方法是其中一个操作数为指针，另一个操作数为整数。因此，对于$e_1[e_2]$，一个操作数为指针，另一个操作数为整数。习惯上，e_1是数组名，e_2是整数表达式，但也可以e_1是指针，或逆转操作数顺序。定义下标的结果是保证数组使用基数为0的索引。

210 要建立多维数组引用，就要组合下标运算符。

例

```
char buffer[100], *bptr = buffer;
int i = 99;
...
buffer[0] = '\0';       /* subscripting an array */
bptr[i-1] = bptr[0];    /* subscripting a pointer */
i[bptr]   = '\0';       /* unconventional subscripting */
```

以上程序分配给具有100个元素的数组**buffer**的第一个元素是**buffer[0]**，最后一个元素是**buffer[99]**。名称**buffer**与**bptr**都指向同一位置，即**buffer[0]**，是**buffer**数组的第一个元素，可以在下标表达式中以相同方式使用。但是，**bptr**是个变量（左值），因此可以指向其他某些位置，如

```
bptr = &buffer[6];
```

然后表达式**bptr[-4]**指向与表达式**buffer[2]**相同的位置（这演示了某些情况下负数下标是有意义的）。也可以通过赋值使**bptr**不指向任何位置：

```
bptr = NULL;   /* Store a null pointer into bptr. */
```

但是，数组名**buffer**不是左值，不能修改。作为指针，它总是指向同一个固定位置，就好像它使用了下列声明：

```
char * const buffer;
```

□

例 下列代码在10×10数组**matrix**的对角元素中存储1.0，在其他元素中存储0.0。

```
int matrix[10][10];
…
for (i = 0; i < 10; i++)
   for (j = 0; j < 10; j++)
      matrix[i][j] = ((i == j) ? 1.0 : 0.0);
```
□

在下标括号中使用逗号表达式是不良编程风格，因为这样会使熟悉其他编程语言的读者误以为是多维数组的下标。

例 表达式`commands[k=n+1, 2*k]`好像是引用二维数组**commands**，下标表达式为**k=n+1**与**2*k**，实际上C语言中的解释是引用一维数组**commands**，在**k**赋值为**n+1**之后，下标为**2*k**。如果确实需要逗号表达式（很少有这种情形），则应把它放在括号中，表示这是特例：

```
commands[(k=n+1, 2*k)]
```
□ 211

可以用指针和类型转换引用多维数组，就像一维数组一样，这样可以提高效率。记住，C语言中的数组按行存储。

例 下列代码设置相同的矩阵，对角元素中存储1.0，在其他元素中存储0.0。这个方法比较复杂，但比较快，它把二维数组当作具有相同元素个数的一维向量，从而简化下标，不需要使用嵌套循环。

```
#define SIZE 10
double matrix[SIZE][SIZE];
int i;
for (i = 0; i < SIZE*SIZE; i++)
   ((double *)matrix)[i] = 0.0;   /* zero all elements */
for (i = 0; i < SIZE*SIZE; i += (SIZE + 1))
   ((double *) matrix)[i] = 1.0; /* set diagonals to 1 */
```
□

参考章节 加法运算符+ 7.6.2；数组类型 5.4；逗号表达式 7.10；间接访问运算符* 7.5.7；整型 5.1；左值 7.1；指针类型 5.3

7.4.2 成员选择

成员选择运算符可以访问结构类型与联合类型的成员（字段）：

component-selection-expression (成员选择表达式):
direct-component-selection
indirect-component-selection

direct-component-selection (直接成员选择):
postfix-expression . *identifier*

indirect-component-selection (间接成员选择):
postfix-expression -> *identifier*

直接成员选择表达式由后缀表达式、点号和标识符组成。后缀表达式应为结构类型或联合类型，标识符应为这个类型的成员名。选择表达式的结果是结构或联合的命名成员。

如果结构或联合表达式是左值，则直接成员选择表达式的结果为左值（只有函数返回的结构或联合值才不是左值）。如果是左值结果，而且选择的成员不是数组，则这个结果可以修改。 212

例

```
struct S {int a,b;} x;
extern struct S f(); /* structure-returning function */
int i;
…
x = f();       /* OK */
i = f().a;     /* OK */
f().a = i;     /* Invalid; f() is not an lvalue */
```

最后一个赋值语句虽然有效，但不合理。函数f返回某个结构的拷贝，然后修改其中一个成员，最后在语句末尾放弃整个拷贝。 □

一些非标准C语言实现不允许函数返回结构。如果允许，有些也不允许函数调用采用选择运算符，把f().a看成错误。

如果点号前面的表达式具有类型限定符或成员具有类型限定符，则结果为两组限定符的联合。

例 下列赋值语句无效，因为x.a的类型为const int，const是从x继承的：

```
const struct {int a,b;} x;
…
x.a = 5;  /* Invalid */
```

□

间接成员选择表达式由后缀表达式、->运算符和名称组成。后缀表达式的值应为结构类型的指针或联合类型的指针，名称为这个结构类型的成员名或联合类型的成员名。结果是结构类型的命名成员或联合类型的命名成员，是左值，是可以修改的，除非是数组成员。表达式*e*→*name*根据定义与表达式(*e).*name*精确等价。

例 下列代码中，结构Point的两个成员都设置为0.0，演示这种对等性。

```
struct {float x, y; } Point, *Point_Ptr;
…
Point.x = 0.0;                /* Sets x to 0.0 */
Point_Ptr = &Point;
Point_Ptr->y = 0.0;   /* Sets y to 0.0 */
```

□

如果->前面的表达式具有类型限定符或成员具有类型限定符，则结果为两组限定符的联合。

有些C语言实现允许在间接选择运算符左边使用null指针。对结果采用地址运算符&和将这个
213 结果转换成整型可以得到结构中成员的偏移量（字节）。标准中没有显式地允许或禁止这种做法，但这通常是可行的。

例

```
#define OFFSET(type,field)  \
   ((size_t)&((type *)0)->field)
```

这个OFFSET宏类似于stddef.h文件中的offsetof宏。 □

参考章节 地址运算符& 7.5.6；间接访问运算符* 7.5.7；左值 7.1；offsetof宏 11.1；size_t 13.1；结构类型 5.6；类型限定符 4.4.3；联合类型 5.7

7.4.3 函数调用

函数调用由后缀表达式、左括号、逗号分开的参数表达式序列（可能是空的）和右括号组成：

function-call (函数调用):
postfix-expression (*expression-list*$_{opt}$)

expression-list :
assignment-expression
expression-list , *assignment-expression*

对于类型*T*，函数表达式类型在经过普通一元转换之后应为“返回*T*的函数的指针”。函数调用的结果类型为*T*，不是左值。如果*T*为**void**，则函数调用不产生结果，不能在要求函数调用产生结果的上下文中使用。*T*不能是数组类型。

在标准化之前的编译器中，函数表达式的类型应为“返回*T*的函数”，因此函数指针要显式地取消引用。如果**fp**是函数指针，则所指的函数只能用**(*fp)(...)**调用。但如果**fp**为正常参数时，可以写成**fp(...)**。

要进行函数调用，就要先求值函数表达式和参数表达式，求值的顺序没有指定。

然后，如果函数调用由标准C语言原型控制（见9.2节），则参数表达式的值转换成原型中指定的相应正式参数类型。如果不能进行这种转换，则调用出错。如果函数的参数个数是变化的，则多余参数根据普通参数转换规则进行转换（见6.3.5节），而不再进一步检查多余参数。

如果函数调用不是由原型控制，则参数表达式只是根据普通参数转换规则进行转换，编译器不再进一步检查。这是因为，没有原型时，编译器没有外部函数正式参数的任何信息。 214

实际参数进行求值和转换之后，被复制到所调用函数的正式参数中，这样，所有参数都按数值传递。在被调函数中，正式参数名是左值，但赋值正式参数只改变正式参数的拷贝值，不影响作为左值的任何实际参数。

例 下列函数**square**返回参数的平方：

```
double square(double y) { y = y*y; return y; }
```

假设**x**是**double**类型的变量，数值为4.0，则进行函数调用**square(x)**时，函数返回数值16.0，但**x**的值仍然保持4.0。在**square**中赋值**y**只改变正式参数的拷贝值。 □

被调函数可以改变调用者的数据，当且仅当数据对函数独立有效（例如全局变量中）或调用者将数据指针作为参数传入函数。传递指针时，复制的是指针，而不是所指的对象。因此，间接通过指针进行的改变可以传递回调用者。

例 下面的函数**swap**在提供对象指针作为参数时，交换两个整数对象的值：

```
void swap(int *xp, int *yp)
{
    int t = *xp;
    *xp = *yp;
    *yp = t;
}
```

如果**a**是所有元素均为0的整型数组，而**i**是数值为4的整型变量，则调用**swap(&a[i],&i)**之后，**i**的值为0，**a[4]**的值为4。 □

C语言总是把数组类型的正式参数和实际参数转换成指针。因此，改变函数中的数组正式参数会影响实际参数，尽管看起来不太明显。

例 下列函数**f**有一个数组参数：

```
void f(int a[10])
{
    a[4] = 12;  /* changes caller's array */
}
```

如果**vec**是整型数组，则调用**f(vec)**把**vec[4]**设置为12。数组参数中的维数10没有意义，**a**
215 也可以声明为**int a[]**。 □

如果在要放弃数值的上下文中调用的函数返回类型不是**void**，则编译器会发出警告。但是**printf**之类的非**void**函数也经常要放弃返回值，因此许多编程人员认为这种警告是无害的。

例 可以通过类型转换明确表示放弃数值的意图，例如调用**strcat**函数如下：

```
(void) strcat(word, suffix);
```
□

逗号表达式可以作为函数的参数，只要把其成员放在括号中，以免解释为另一参数。

例 假设C语言程序中要跟踪函数**f**的所有调用。如果**f**取一个参数，则下列宏在每个**f**调用之前插入**tracef**调用：

```
#define f(x) (tracef(__FILE__, __LINE__), f((x)))
```

如果调用**f**显示为函数参数，如**g(f(y))**，则**g**的参数是个逗号表达式。 □

参考章节 参数一致性 9.6；逗号运算符 7.10；放弃表达式 7.13；函数类型 5.8；函数原型 9.2；间接访问运算符* 7.5.7；左值 7.1；宏扩展 3.3.3；指针类型 5.3；**printf** 15.11；**strcat** 13.1；普通参数转换 6.3.5；**void**类型 5.9

7.4.4 后缀自增运算符与后缀自减运算符

后缀运算符**++**与**--**分别将操作数递增和递减，同时将原值作为结果产生。它们是产生副作用的运算符：

postincrement-expression (后缀自增表达式):
　　postfix-expression **++**

postdecrement-expression (后缀自减表达式):
　　postfix-expression **--**

这两个运算符的操作数都是可修改的左值，可以是任何实数算术或指针类型。常数1在**++**运算中与操作数相加，在**--**运算中与操作数相减，修改操作数。结果是递增或递减之前的旧操作数值。运算结果不是左值。操作数和常量1先进行普通二元转换之后再进行加法或减法，将修改
216 后的数值存储回操作数时，进行普通赋值转换。结果类型是转换前左值操作数的类型。

例 如果**i**和**j**是整型变量，则语句**i=j--;**可以改写成下列两条语句：

```
i = j;
j = j-1;
```
□

如果发生溢出，而操作数是带符号整数或浮点数，则这些运算符可能产生无法预测的后果。对无符号类型的最大可表示值进行自增运算将得到0，而对无符号类型的最小可表示值进行自减

运算也将得到0。

如果操作数是指针，是某个*T*类型的“*T*指针”类型，则++的结果是把指针移到所指对象下一个对象，就像*T*类型对象数组中将指针移到下一个元素（在字节寻址计算机上，就是让指针前进**sizeof(*T*)**个字节）。同样，--的效果就是将指针后退，就像*T*类型对象数组中将指针移到上一个元素。这两种情况下，表达式的值是修改之前的指针。

例 扫描数组或字符串元素时，经常使用后缀自增运算符，下面的例子计算字符串中的字符数：

```
int string_length(const char *cp)
{
    int count = 0;
    while (*cp++) count++;
    return count;
}
```
□

参考章节 加法 7.6.2；数组类型 5.4；赋值转换 6.3.2；浮点数类型 5.2；整数类型 5.1；左值 7.1；溢出 7.2.2；指针类型 5.3；标量类型 第5章；带符号类型 5.1.1；减法 7.6.2；无符号类型 5.1.2；普通二元转换 6.3.4

7.4.5 复合字面值

C99引入了复合字面值，表示集合类型的未命名常量。复合字面值包括括号中的类型名和花括号中的初始化表达式列表。初始化表达式列表后面还可选加上逗号。

compound-literal（复合字面值）：

(*type-name*) { *initializer-list* ,$_{opt}$ } *(C99)*

复合字面值生成指定类型的未命名对象，返回这个对象的左值。类型名可以指定任何对象类型或未知长度的数组类型。复合字面值中不能使用变长数组类型，因为它们无法初始化。结构、联合、数组和枚举类型在复合字面值中特别有用。初始化表达式列表的格式和含义与同一类型与生存期的对象声明中允许的初始化表达式相同。特别地，复合字面值的未初始化成员初始化为0（见4.6节）。 217

const类型限定符可以在复合字面值类型名中生成只读字面值，否则复合字面值是可以修改的。如果两个复合字面值具有相同类型和数值，则实现可以随意对其复用相同存储空间，其地址可以相同，像重复字符串字面值时一样。

例 让**Temp1**指向一个可修改字符串，让**Temp2**指向一个只读字符串。

```
char *Temp1 = (char []){"/temp/XXXXXXXX"};
char *Temp2 = "/temp/XXXXXXXX";
```

函数**POW2**通过表格查找计算2的指数。

```
inline int POW2(int n)
{
    assert( n >= 0 && n <= 7 );
    return (const int []){1, 2, 4, 8, 16, 32, 64, 128}[n];
}
```

DrawTo带有一个以数值形式传递的点结构，面**DrawLine**接收两个点的地址。

```
DrawTo( (struct Point){.x=12, .y=n+3} );
DrawLine( &(struct Point){x,y}, &(struct Point){-x,-y} );
```
□

如果复合字面值出现在文件顶部，则未命名对象的生存期是静态的——在整个程序执行期间存在。这时初始化表达式列表只能包含常量值。如果复合字面值出现在函数中，则具有最内层块的自动生存期与作用域。取得复合字面值的地址时，其生存期非常重要，编程人员要保证，离开字面值的作用域之后，这个地址未被使用。

每次进入所在块时，分配复合字面值，但不离开作用域而重复执行复合字面值只是对存储体进行必要的重新初始化。这种重复执行只能发生在用**goto**语句构造的循环中，因为在任何迭代语句中，复合字面值应在迭代体的作用域中，而每次迭代时都要重新进入一个新的作用域。

例 下列循环在**ptrs**中填入一个项目的指针，***(ptrs[i])== 4**。

```
int * ptrs[5]; int i = 0;
again:
   ptrs[i] = (int [1]){i};
   if (++i<5) goto again;
```

218 下列代码在**ptrs**中填入不同数组的指针，***(ptr[i])== i**。

```
int * ptrs[5]; int i = 0;
ptrs[i] = (int [1]){i++}; }
ptrs[i] = (int [1]){i++}; }
ptrs[i] = (int [1]){i++}; }
ptrs[i] = (int [1]){i++}; }
ptrs[i] = (int [1]){i++}; }
```

下列代码在**ptrs**中填入未定义（悬而未决）的指针，因为循环迭代结束时去配每个字面值数组：

```
int *ptrs[5];
for(int i=0; i<5; i++) { ptrs[i] = (int [1]){i}; }
```

□

参考章节 初始化表达式 4.6；变长数组 5.4.5

7.5 一元表达式

下面几节介绍几种一元表达式：

cast-expression（转换表达式）：
　　unary-expression
　　(*type-name*) *cast-expression*

unary-expression（一元表达式）：
　　postfix-expression
　　sizeof-expression
　　unary-minus-expression
　　unary-plus-expression
　　logical-negation-expression
　　bitwise-negation-expression
　　address-expression
　　indirection-expression
　　preincrement-expression
　　predecrement-expression

一元运算符的优先级低于后缀表达式，但高于所有二元和三元运算符。例如，表达式***x++**解释为***(x++)**，而不是**(*x)++**。

参考章节 二元表达式 7.6；后缀表达式 7.4；优先级 7.2.1；一元加法运算符 7.5.3

7.5.1 类型转换

类型转换表达式由左括号、类型名、右括号和操作数表达式组成。语法参见前面显示的一元表达式语法。 219

类型转换使操作数值转换成括号中指定的类型。任何允许的转换（见6.3.1节）都可以在类型转换表达式中调用，结果不是左值。

例

```
extern char *alloc();
struct S *p;
p = (struct S *) alloc(sizeof(struct S));
```
□

C语言的一些实现错误地忽略某些只具有“缩小”数值作用的类型转换。

例 假设类型**unsigned short**用16位表示，而类型**unsigned**用32位表示，则表达式

```
(unsigned)(unsigned short)0xFFFFFF
```

数值应为**0xFFFF**，因为类型转换**(unsigned short)**造成数值**0xFFFFFF**截尾成16位，然后类型转换**(unsigned)**将这个值加宽回到32位。有缺陷的编译器无法实现这个截尾效果，产生的代码不加改变面直接传递数值**0xFFFFFF**。同样，对于表达式

```
(double)(float)3.1415926535897932384
```

有缺陷的编译器无法产生将π的近似值精度减少到**float**类型的代码，而是不加改变而直接传递双精度值。 □

为了在使用非标准编译器时保证最大的可移植性，编程人员应截尾数值，将其存放在变量中，或对整数进行显式掩码运算（如用二进制按位与运算符**&**），而不是依赖于缩小转换。

参考章节 按位与运算符 7.6.6；类型转换 第6章；类型名 5.12

7.5.2 sizeof运算符

sizeof运算符可以取得一个类型或数据对象的长度：

sizeof-expression :
sizeof (*type-name*)
sizeof *unary-expression*

sizeof表达式有两种形式：**sizeof**运算符加带括号的类型名或**sizeof**运算符加操作数表达式。运算结果是整数值常量，不能是左值。在标准C语言中，**sizeof**的结果是头文件**stddef.h**中定义的无符号整数类型**size_t**。传统C语言实现通常用**int**或**long**作为结果类型。按照C语言优先级，**sizeof(long)-2**解释为**(sizeof(long))-2**而不是**sizeof((long)(-2))**。 220

对括号类型名采用**sizeof**运算符得到指定类型的对象长度，即该对象类型占用的内存量（存储单元数），包括内部或尾部的填充。根据定义，**sizeof**运算符作用于任何字符类型时得到1。类型名可以是不完整数组类型（没有显式长度）、函数类型或**void**类型。

对表达式进行**sizeof**运算得到的结果与对这个表达式的类型名进行**sizeof**运算得到的结果相同。**sizeof**运算符使表达式确定类型时不必进行任何普通转换，这样就可以用**sizeof**取得数组总长度，而不必将数组名转换成指针。但是，如果表达式中包含进行普通转换的运算符，则会在确定类型时考虑这些转换。**sizeof**的操作数可以是不完整数组类型或函数类型，但如果

sizeof运算符作用于声明为数组或函数类型的正确参数名，则返回的值是按正常规则将正式参数转换成这些类型时取得的指针类型长度。在标准C语言中，**sizeof**运算符的操作数不能是结构或联合对象中指定位字段的左值，但有些非标准实现允许这种做法，返回值为声明的成员类型长度（忽略位字段指定）。

例 下面是一些应用**sizeof**运算符的例子。假设**short**类型对象占用2个字节，**int**类型对象占用4个字节。

表 达 式	数 值
`sizeof(char)`	1
`sizeof(int)`	4
`short s; ... sizeof(s)`	2
`short s; ... sizeof(s+0)`	4(加法的结果类型为**int**)
`short sa[10];... sizeof(sa)`	20
`extern int ia[];...sizeof(ia)`	非法(类型不完整)

□

如果表达式采用**sizeof**运算符，则在编译时分析表达式以确定类型，但不求值这个表达式。如果**sizeof**的参数是类型名，则作为副作用，可以声明一个类型。

如果**sizeof**表达式中出现变长数组，而数组长度值影响**sizeof**表达式值，则总是完全求值数组长度表达式，包括副作用。如果数组长度值不影响**sizeof**结果，则没有确定是否求值长
221 度表达式。

例 下面语句中，**j**不递增，但**n**递增。函数调用**f(n)**不一定执行，因为**sizeof**表达式只计算变长数组的指针长度，不依赖于数组长度。

```
size_t z = sizeof(j++);
size_t x = sizeof(int [n++]);
size_t y = sizeof(int (*)[f(n)]);
```

语句`sizeof(struct S {int a,b;})`可以在标准C语言中生成新类型，但这不是良好的编程风格。源文件可以在后面引用这个类型（这在C++中是无效的）。

参考章节 数组类型 5.4；C++兼容性 7.15；函数类型 5.8；**size_t** 11.1；存储单元 6.1.1；类型名 5.12；无符号类型 5.1.2；普通二元转换 6.3.4；变长数组 5.4.5；**void**类型 5.9

7.5.3 一元负号运算符与一元正号运算符

一元负号运算符计算操作数的相反数，一元正号运算符（在标准C语言中引入）得到操作数的值：

unary-minus-expression (一元负号表达式):
　　- *cast-expression*

unary-plus-expression (一元正号表达式): *(C89)*
　　+ *cast-expression*

这两个运算符的操作数可以是任何算术类型，进行普通一元转换。结果的类型升级，而且不是左值。

一元负号表达式**-***e*是**0-(***e***)**的缩写，这两个表达式进行相同的计算。这个计算在操作数为

带符号整数或浮点数，并且发生溢出时可能产生无法预测的后果。对于无符号整数操作数k，结果总是无符号，等于$2^n - k$，其中n是表示结果的位数。由于结果是无符号数，因此不能是负数。这好像很奇怪，但需要注意的是，对于任何无符号整数x，$(-x)+x$等于0，而对于任何$-x$有良好定义的带符号整数x，$(-x)+x$也等于0。

一元正号表达式**+***e*是**0+(***e***)**的缩写。

参考章节 浮点数类型 5.2；整数类型 5.1；左值 7.1；溢出 7.2.2；减法运算符 7.6.2；无符号类型 5.1.2；普通一元转换 6.3.3

7.5.4 逻辑非运算符

一元运算符**!**计算操作数的逻辑反数。这个操作数可以是任何标量类型：

222

logical-negation-expression:
    ! *cast-expression*

操作数进行普通二元转换。一元运算符**!**的结果是**int**类型，如果操作数为0（对指针为null,对浮点数值为0.0），则结果为1，否则结果为0。结果不是左值。表达式**!(x)**等价于**(x)==0**。

例

```
#define assert(x,s) if (!(x)) assertion_failure(s)
...
assert(num_cases > 0, "No test cases.");
average = total_points/num_cases;
```

使用**assert**宏可以防止除数为0的问题，这是别的方法很难做到的。**assertion_failure**是个函数，接受一个字符串并将其作为消息向用户报告。标准头文件**assert.h**中有类似的**assert**宏。 □

参考章节 **assert** 19.1；相等运算符**==** 7.6.5；浮点数类型 5.2；整数类型 5.1；左值 7.1；指针类型 5.3；标量类型 第5章；普通二元转换 6.3.3

7.5.5 按位反运算符

一元运算符**~**对操作数按位取反：

bitwise-negation-expression:
    ~ *cast-expression*

操作数进行普通二元转换，操作数可以是任何整型类型。**~***e*的二进制表示中每一位都是操作数*e*的逆，结果不是左值。

例 如果**i**是16位整数，取值为**0xF0F0**(1111000011110000_2)，则**~i**的值为**0x0F0F**(0000111100001111_2)。 □

由于不同实现可能对带符号整数使用不同表示方法，因此对带符号操作数采用按位反运算符**~**的结果可能无法移植。为了得到可移植代码，建议只对无符号操作数采用按位反运算符**~**。对无符号操作数*e*，如果*e*的转换类型为**unsigned**，则**~***e*的值为**UINT_MAX** $-$ *e*；如果*e*的转换类型为**unsigned long**，则**~***e*的值为**ULONG_MAX** $-$ *e*。**UINT_MAX**与**ULONG_MAX**的值在标准C语言头文件**limits.h**中定义。

参考章节　整型类型　5.1；**limits.h**　5.1.1；左值　7.1；带符号类型　5.1.1；无符号类
223 型　5.1.2；普通一元转换　6.3.3

7.5.6 地址运算符

一元运算符**&**返回操作数的指针：

address-expression：
　　& *cast-expression*

&的操作数应为函数指定符或指定一个对象的左值。如果是左值，则对象不能用存储类**register**声明或是位字段。如果**&**的操作数类型为*T*，则结果类型为“*T*的指针”。**&**的操作数不采用普通转换，结果不能是左值。

函数指定符采用地址运算符得到函数的指针。由于普通转换规则将函数指定符转换成指针，因此函数通常不需要**&**运算符。事实上，一些标准C语言之前的实现不允许函数使用**&**运算符。

例

```
extern int f();
int (*fp)();
…
fp = &f;    /* OK; & yields a pointer to f */
fp = f;     /* OK; usual conversions yield a pointer to f */
```
□

地址运算符产生的函数指针在整个C语言程序执行期间有效。地址运算符产生的对象指针只在对象存储体保持分配期间有效。如果**&**的操作数是指定静态生存期变量的左值，则指针在整个程序执行期间有效。如果操作数指定自动变量，则指针在变量声明所在块活动期间有效。如果操作数指定动态分配的对象（如用**malloc**分配），则指针在内存显式释放之前有效。

标准C语言中地址运算符的作用有些不同于传统C语言。在标准C语言中，地址运算符作用于“*T*的数组”类型的左值时得到“*T*的数组的指针”类型的值，而许多标准化之前的编译器把**&a**看成与**a**相同，即**a**的第一个元素的指针。这两种解释是相互不一致的，但标准规则与**&**的解释更一致。

224 **例**　在下列标准C语言程序段中，**p**的所有赋值等价，**i**的所有赋值等价。

```
int a[10], *p, i;
…
p = &a[0];  p = a;   p = *&a;
i = a[0];   i = *a;i = **&a;
```
□

参考章节　数组类型　5.4；函数指定符　7.1；函数类型　5.8；左值　7.1；指针类型　5.3；**register**存储类　4.3

7.5.7 间接访问运算符

间接访问运算符*通过指针进行间接访问。**&**与*运算符是互逆的：如果*是变量，则表达式***&x**与**x**相同。

indirection-expression：
　　* *cast-expression*

操作数应为指针，如果是“*T*的指针”类型，则对于一些限定类型的*T*，结果类型为*T*（具有相同限定）。如果指针指向一个对象，则结果是引用这个对象的左值。如果指针指向一个函数，

则结果是函数指定符。

例

```
int i,*p;
const int *pc;
...
p   = &i;    /* p now points to variable i */
*p  = 10; /* sets value of i to 10 */
pc  = &i; /* pc now points to i, too */
*pc = 10;    /* invalid, *pc has type 'const int' */
```

□

间接访问运算符的操作数进行普通一元转换。惟一相关的转换是数组与函数指定符转换为指针。因此，如果f是函数指定符，则表达式*&f与*f是等价的。对于后者，f通过普通转换规则转换成&f。

对无效或null指针采用间接访问运算符的效果是未定义的。在有些实现中，null指针取消引用会使程序终止，而在有些实现中，则相当于null指针指定的内存块具有无法预测的内容。

参考章节 数组类型 5.4；函数指定符 7.1；函数类型 5.8；左值 7.1；指针类型 5.3；普通一元转换 6.3.3

7.5.8 前缀自增运算符与前缀自减运算符

前缀运算符++与--分别将操作数递增和递减，同时将修改后的值作为结果产生。它们是产生副作用的运算符（这些运算符还有后缀形式）：

225

preincrement-expression (前缀自增):

++ *unary-expression*

predecrement-expression (前缀自增):

-- *unary-expression*

这两个运算符的操作数都是可修改的左值，可以是任何实数算术或指针类型。常数1在++运算中与操作数相加，在--运算中与操作数相减。两种情况下，结果都存放回左值中，这个结果成为新的操作数值。操作数和常量1先进行普通二元转换之后再进行加法或减法，将修改后的数值存储回操作数时，进行普通赋值转换。结果类型是转换前左值操作数的类型。

如果操作数是指针，是某个*T*类型的“*T*指针”类型，则++的结果是把指针移到所指对象下一个对象；就像*T*类型对象数组中将指针移到下一个元素（在字节寻址计算机上，就是让指针前进**sizeof**(*T*)个字节）。同样，--的效果就是将指针后退，就像*T*类型对象数组中将指针移到上一个元素。

例 下列**strrev**函数在第二个参数中复制第一个参数的逆拷贝：

```
int strrev( const char *s1, char *s2 )
{
    const char *p = s1;
    while (*p++); /* Locate end of first string. */
    --p;          /* Overshot: back up to the null. */
    /* Now copy the characters in reverse order. */
    while (p > s1)
        *s2++ = *--p;
    *s2 = '\0';  /* Terminate the result string. */
}
```

□

如果发生溢出，而操作数是带符号整数或浮点数，则这些运算符可能产生无法预测的后果。对无符号类型的最大可表示值进行自增运算将得到0，而对无符号类型的最小可表示值进行自减运算也将得到0。

表达式++*e*等价于*e*+=1，--*e*等价于*e*-=1。如果不使用前缀自增运算符与自减运算符产生的值，则前缀与后缀形式的效果相同，即语句*e*++;等价于++*e*;，语句*e*--;等价于--*e*;。

参考章节 加法 7.6.2；数组类型 5.4；赋值转换 6.3.2；复合赋值 7.9.2；表达式语句 8.2；浮点数类型 5.2；整数类型 5.1；左值 7.1；溢出 7.2.2；指针类型 5.3；后缀自增运算符与自减运算符 7.4.4；标号类型 第5章；带符号类型 5.1.1；减法 7.6.2；无符号类型 5.1.2；普通二元转换 6.3.4

226

7.6 二元运算符表达式

二元运算符表达式是用二元运算符分开的两个表达式。这里的二元指两个操作数，而与二进制没有什么关系。二元表达式及其操作数类型按优先级由高到低如表7-4所示。所有运算符都是左结合的。

表7-4 二元运算符表达式

表达式类型	运 算 符	操 作 数	结 果
乘法表达式（multiplicative-expression）	*/	算术	算术
	%	整数	整数
加法表达式（additive-expression）	+	算术	算术
		指针+整数或整数+指针	指针
	-	算术	算术
		指针 - 整数	指针
		指针 - 指针	整数
移位表达式（shift-expression）	<< >>	整数	整数
关系表达式（relational-expression）	< <= >= >	算术或指针	0或1
判等表达式（equality-expression）	== !=	算术或指针	0或1
按位与表达式（bitwise-and-expression）	&	整数	整数
按位异或表达式（bitwise-xor-expression）	^	整数	整数
按位或表达式（bitwise-or-expression）	|	整数	整数

对本节介绍的每个二元运算符，要先求值两个操作数（但没有特定顺序）之后再进行运算。

参考章节 求值顺序 7.12；优先级 7.2.1

7.6.1 乘法运算符

3个乘法运算符*（乘法）、/（除法）和%（求余）具有相同优先级，都是左结合的：

multiplicative-expression:
　　cast-expression
　　multiplicative-expression mult-op cast-expression

mult-op : **one of**
　　* / %

227

参考章节 优先级 7.2.1

乘法 二元运算符*表示乘法，每个操作数可以是任意算术类型。操作数进行普通二元转换，

结果类型是转换的操作数类型。运算的结果不是左值。对整型操作数，进行整数乘法；对浮点数操作数，进行浮点数乘法。

如果发生溢出，而转换后的操作数是带符号整数或浮点数，则乘法运算符可能产生不可预测的后果。如果操作数为无符号整数，则结果为真实数学结果的2^n模数，其中n是表示无符号结果的位数。

参考章节 算术类型 第5章；浮点数类型 5.2；整数类型 5.1；左值 7.1；求值顺序 7.12；溢出 7.2.2；带符号类型 5.1.1；无符号类型 5.1.2；普通转换 6.3.4

除法 二元运算符/表示除法，每个操作数可以是任意算术类型。操作数进行普通二元转换，结果类型是转换的操作数类型。运算的结果不是左值。

对浮点数操作数，进行浮点数除法。对整型操作数，如果无法整除，则小数部分放弃（截尾到0）。在C99之前，C实现可以在操作数为负数时选择截尾到靠近0或远离0。负操作数的**div**与**ldiv**库函数总是良定义的。

如果发生溢出，而转换后的操作数是带符号整数或浮点数，则除法运算符可能产生不可预测的后果。注意如果用对二的补码形式表示带符号整数，则最大可表示负数除以 -1。可能产生溢出，得到的数学结果是无法表示的正值。操作数为无符号整数时不会发生溢出。

除数为0（整数或浮点数）的后果是未定义的。

参考章节 算术类型 第5章；**div** 17.1；浮点数类型 5.2；整数类型 5.1；**ldiv** 17.1；左值 7.1；溢出 7.2.2；带符号类型 5.1.1；无符号类型 5.1.2；普通转换 6.3.4

求余 二元运算符%计算第一个操作数除以第二个操作数的余数。每个操作数可以是任意整数类型。操作数进行普通二元转换，结果类型是转换的操作数类型。运算的结果不是左值。库函数**div**、**ldiv**与**fmod**也计算整数或浮点数的余数。

如果**a/b**是可表示的，则**(a/b)*b+a%b**总是等于**a**，因此余数运算的行为与整数除法相同。上节曾介绍过，在C99之前，一个操作数为负数时，除法运算符的行为是与实现相关的，使求余运算符的行为也变成是实现相关的。

例 下列**gcd**函数利用欧几里得算法计算最大公倍数，得到能整除**x**和**y**的最大整数。 228

```
unsigned gcd(unsigned x, unsigned y)
{
   while ( y != 0 ) {
      unsigned temp = y;
      y = x % y;
      x = temp;
   }
   return x;
}
```

□

如果两个操作数的除法产生溢出，则求余运算符也会出现无法预测的后果。注意如果用对二的补码形式表示带符号整数，则最大可表示负数除以 -1可能产生溢出，得到的数学结果是无法表示的正值，这时虽然余数（0）是可表示的，但也会出现无法预测的后果。

操作数为无符号整数时不会发生溢出。

参考章节 **div**, **ldiv** 17.1；**fmod** 17.3；整型类型 5.1；左值 7.1；溢出 7.2.2；带符号类型 5.1.1；无符号类型 5.1.2；普通二元转换 6.3.4

7.6.2 加法运算符

加法运算符类别有两个运行符+（加法）和-（减法），具有相同优先级，都是左结合的：

additive-expression:
　　multiplicative-expression
　　additive-expression add-op multiplicative-expression

add-op: one of
　　+ -

加法　二元运算符+表示加法。操作数进行普通二元转换。操作数可以都是算术类型，也可以一个是对象指针，一个是整数。不允许其他类型的操作数。运算的结果不是左值。

如果操作数是算术类型，则结果类型是转换的操作数类型。对整型操作数，进行整型加法；对浮点数操作数，进行浮点数加法。

将指针p与整数k相加时，p所指的对象假设为位于该对象的数组内或是该数组中最后一个对象的后一个对象，结果是数组中这个对象的指针，对象离p所指的对象隔k个对象。例如，$p+1$指向p所指的对象的后一个对象，$p+(-1)$指向p所指的对象的前一个对象。如果指针p或$p+k$不在数组之内或没有紧跟数组后面，则行为是未定义的。p不能是函数指针或**void ***类型。

229

例　假设计算机是字节寻址的，**int**类型分配4字节。假设**a**是10个整数的数组，从地址**0x100000**开始。假设**ip**为整数的指针，指定为数组**a**的第一个元素的地址。最后，假设**i**为整数变量，当前保存的值为6。对于下列情形：

```
int *ip, i, a[10];
ip = &a[0];
i = 6;
```

ip+i的值是多少？由于**int**类型分配4字节，因此表达式**ip+i**变成**0x100000+4*6**或**0x100018**(24_{10}即18_{16})。 □

例　多维与变长数组指针（C99）的实现方法差不多：

```
int n = 3; int m = 5;
double rect[n][m];
double (*p)[m];
p = rect; /* same as p = &rect[0]; */
p++; /* now p == &rect[1] */
```

标识符**p**指向**double [m]**类型对象，是5个双精度浮点数数组，相当于矩阵**rect**的一行。表达式**p++**将**p**向前移到**rect**中下一行，移动**5*sizeof(double)**个存储单元。 □

如果发生溢出，而转换后的操作数是带符号整数或浮点数，或者其中一个操作数为指针，则加法运算符可能产生不可预测的后果。如果操作数均为无符号整数，则结果为真实数学结果的2^n模数，其中n是表示无符号结果的位数。

参考章节　数组类型　5.4；浮点数类型　5.2；整数类型　5.1；左值　7.1；多维数组　5.4.2；求值顺序　7.12；溢出　7.2.2；指针表示　5.3.2；指针类型　5.3；标号类型　第5章；带符号类型　5.1.1；无符号类型　5.1.2；普通二元转换　6.3.4；变长数组　5.4.5

减法　二元运算符-表示减法。操作数进行普通二元转换。操作数可以都是算术类型，也可以左操作数是对象指针，右操作数是整数，或者两个都是兼容对象类型的指针（忽略任何类型

限定符)。运算的结果不是左值。

如果操作数是算术类型，则结果类型是转换的操作数类型。对整型操作数，进行整型减法；对浮点数操作数，进行浮点数减法。

例 一个无符号整数减去另一个无符号整数的结果总是无符号类型，因此不能是负数。但是，无符号数符合下列等价条件：

```
(a+(b-a)) == b
```

230

和

```
(a-(a-b)) == b
```

从指针减去整数同将指针与整数相加的运算处理方式类似。将指针p与整数k相减时，p所指的对象假设为位于该对象的数组内或是该数组中最后一个对象的后一个对象，结果是数组中这个对象的指针，对象离p所指的对象隔$-k$个对象。例如，$p-(-1)$指向p所指的对象的后一个对象，$p-1$指向p所指的对象的前一个对象。如果指针p或$p-k$不在数组之内或没有紧跟数组后面，则行为是未定义的。p不能是函数指针或**void***类型。

对两个相同类型的指针p和q，$p-q$的结果是整数k，k加q得到p。差的类型是**stddef.h**文件中定义的带符号整数类型**ptrdiff_t**(在标准化前的C语言中，根据实现不同，这个类型可以是**int**或**long**)。只有两个指针指向同一数组中的对象或数组的最后一个对象的后一个对象时，减法结果才是良定义的并且是可移植的。差值k是所指的两个对象下标的差。如果指针p或$p-q$在数组之外，则其行为是未定义的。p和q不能是函数指针或**void***类型。

如果发生溢出，而转换后的操作数是带符号整数或浮点数，或者其中一个操作数为指针。则减法运算符可能产生不可预测的后果。如果操作数为无符号整数，则结果为真实数学结果的2^n模数，其中n是表示无符号结果的位数。

参考章节 数组类型 5.4；浮点数类型 5.2；整数类型 5.1；左值 7.1；溢出 7.2.2；指针表示 5.3.2；指针类型 5.3；**ptrdiff_t** 11.1；标号类型 第5章；带符号类型 5.1.1；类型兼容性 5.11；类型限定符 4.4.3；无符号类型 5.1.2；普通二元转换 6.3.4

7.6.3 移位运算符

二元运算符**<<**表示向左移位，二元运算符**>>**表示向右移位，两者具有相同优先级，都是左结合的：

shift-expression :
    additive-expression
    shift-expression shift-op additive-expression

shift-op : one of
    << >>

移位运算的每个操作数应为整数类型。两个操作数分别进行普通一元转换，结果的类型与转换后的左操作数的类型相同(标准化之前的C语言对两个操作数进行普通二元转换)。移位运算的结果不是左值。

231

第一个操作数是要进行移位操作的数，第二个操作数指定第一个操作数移动的位数。移动方向由使用的移位运算符控制。**<<**表示向左移位，移到边界之外的多余位放弃，缺少的位补0。而**>>**表示向右移位，**>>**运算符移到左边的位取决于转换的左操作数类型，如果是无符号(或带

符号的非负数），则左边移入0；如果是带符号的负数，则实现者可以选择补0或把左操作数最左边的位移入。因此，左操作数为负的带符号值，而右操作数为非0值时，移位运算符>>的应用是不可移植的。

如果右操作数是负数，则移位运算符的结果值是未定义的，因此指定负的移动位数不一定使<<向右移，使>>向左移。如果右操作数的值大于或等于转换后左操作数数值的宽度（以位数为标准），则结果值也是未定义的。如果右操作数为0，则不发生移位，结果值等于转换后左操作数数值。

例 可以利用运算符的优先级和结合律编写不易理解但却格式整齐的表达式，如

```
b << 4 >> 8
```

如果b是16位无符号值，则这个表达式取出中间8位。和平常一样，最好在可能引起混淆时使用括号，如

```
(b << 4) >> 8
```

例 可以用无符号移位运算计算二进制算法中两个整数的最大公倍数。这个方法比欧几里得算法更复杂，但可能更快，因为一些C语言实现求余运算的速度较慢，特别是对无符号操作数。

```
unsigned binary_gcd(unsigned x, unsigned y)
{
    unsigned temp;
    unsigned common_power_of_two = 0;
    if (x == 0) return y; /* Special cases */
    if (y == 0) return x;

    /* Find the largest power of two
       that divides both x and y. */
    while (((x | y) & 1) == 0) {
        x = x >> 1; /* or: "x >>= 1;" */
        y = y >> 1;
        ++common_power_of_two;
    }
    while ((x & 1) == 0) x = x >> 1;

    while (y) {
        /* x is odd and y is nonzero here. */
        while ((y & 1) == 0) y = y >> 1;
        /* x and y are odd here. */
        temp = y;
        if (x > y) y = x - y;
        else y = y - x;
        x = temp;
        /* Now x has the old value of y, which is odd.
           y is even, because it is the difference of two odd
           numbers; therefore it will be right-shifted
           at least once on the next iteration. */
    }
    return (x << common_power_of_two);
}
```

□

参考章节 整数类型 5.1；左值 7.1；优先级 7.2.1； 带符号类型 5.1.1；无符号类型 5.1.2；普通一元转换 6.3.3

7.6.4 关系运算符 □

二元运算符<、<=、>与>=可以用于操作数比较：

relational-expression :
    shift-expression
    relational-expression relational-op shift-expression

relational-op : one of
    `<  <=  >  >=`

关系运算的操作数进行普通二元转换。操作数可以都是实数（非复数）算术类型，都是兼容类型的指针或都是兼容不完整类型的指针。指针类型的任何类型限定符不影响比较。关系运算的结果总是`int`类型，取值为0或1，并且关系运算的结果不是左值。

运算符<测试小于关系，运算符<=测试小于或等于关系，运算符>测试大于关系，运算符>=测试大于或等于关系。如果指定关系对特定操作数值成立，则结果为1；如果指定关系对特定操作数值不成立，则结果为0。

标准C语言中的浮点数算术实现可能包括NaN之类的值，这是无法排序的。在关系表达式中使用这类值时可能造成“无效”异常，此时关系的值为false。17.16节介绍的函数比内部运算符能更好地处理这种情形。

对于整型操作数，进行整型比较（带符号或无符号）。对于浮点数操作数，进行浮点数比较。对指针操作数，结果取决于所指的两个对象地址空间的相对位置，要定义结果，所指的对象应 233
位于相同数组或结构中，这时“大于”指数组的索引值较大或结构的成员在成员表稍后声明。作为数组的特例，指向数组边界外面的那个对象也是良定义的，其大于数组内所有对象的指针。同一联合参数的所有成员指针相等。

例 可以编写`3<x<7`之类的表达式，但其含义与普通数学不等式不同，这是左结合的，相当于`(3<x)<7`。由于`(3<x)`的结果为0或1，都小于7，因此`3<x<7`总是为1。要表示普通数学不等式含义，可以用逻辑与运算符，例如`3<x && x<7`。 □

例 在混合类型中使用关系运算符时要小心。例如下列表达式：

```
-1 < (unsigned) 0
```

也许有些读者会以为这个表达式总是得到1（真），因为－1小于0。但是，经过普通二元转换之后，－1变成大的无符号值，然后再进行比较，而这个无符号值不可能小于0，因此这个表达式总是得到0（假）。 □

一些早期的实现允许指针和整数之间进行关系比较，其实是非法的。这些早期的实现可能进行带符号或无符号比较。

参考章节 算术类型 第5章；数组类型 5.4；按位与运算符& 7.6.6；兼容类型 5.11；浮点数类型 5.2；不完整类型 5.4，5.6.1；整数类型 5.1；逻辑与运算符&& 7.7； 左值 7.1；NaN 5.2；指针类型 5.3；优先级 7.2.1；带符号类型 5.1.1；类型限定符 4.4.3；无符号类型 5.1.2；普通二元转换 6.3.4

7.6.5 判等运算符

二元运算符==和!=比较操作数的相等性：

equality-expression:
　　relational-expression
　　equality-expression equality-op relational-expression

equality-op: one of
　　`==` `!=`

判等运算允许的操作数有多种：

1. 两个操作数均可以是算术类型，包括复数类型。

234

2. 两个操作数均可以是兼容类型的指针，或都是`void *`类型。
3. 一个操作数是对象或不完整类型的指针，另一个`void *`类型。第一个操作数要转换成`void *`类型。
4. 一个操作数是指针，另一个是null指针常量（整型常量0）。

对于指针操作数，所指的类型有没有类型限定符并不影响比较的合法性和结果。算术操作数进行普通二元转换。判等运算的结果总是`int`类型，取值为0或1，且不是左值。

对于整型操作数，进行整型比较。对于浮点数操作数，进行浮点数比较。只有满足下列条件时指针操作数才能判定为相等：

1. 两个指针指向同一对象或函数。
2. 两个指针都是null指针。
3. 两个指针指向同一数组对象中最后一个元素的后一个元素。

运算符==测试等于关系；运算符!=测试不等于关系。如果指定关系对特定操作数值成立，则结果为1；如果指定关系对特定操作数值不成立，则结果为0。

对于复数操作数（C99），实数和虚数部分都要相等时，这个复数操作数才相等。如果一个操作数为实数，一个操作数为复数，则比较时实数操作数先转换成复数类型。两个操作数通过普通二元转换得到相同精度。

结构类型和联合类型不能比较相等性，即使这些关系可以赋值。对齐限制在结构类型和联合类型中造成的间隙使其中可能包含任意值，要补偿这个间隙，会对相等性比较和修改结构类型与联合类型的所有运算带来巨大的开销。

二元判等运算符具有相同优先级（都低于<、<=、>与>=的优先级），都是左结合的。

例　表达式`x==y==7`并不具有普通数学意义上的含义。根据左结合律，它解释为`(x==y)==7`。由于`(x==y)`取值为0或1，都不等于7，因此`x==y==7`总是得到0。要表示普通数学不等式含义，可以用逻辑与运算符，例如

```
x==y && y==7
```

□

例　在按位运算符中，存在和按位与运算符和按位或运算符相配的按位异或运算符，但在逻辑运算中不存在和逻辑与运算符和逻辑或运算符相配的逻辑异或运算符。!=运算符可以作为逻辑异或运算符：如果`a<b`与`c<d`只有一个成立，则表达式`a<b != c<d`的结果为1，否则为0。如果某

235

个操作数可能取0和1以外的值，则可以对两个操作数采用一元运算符!，表达式`!x != !y`在`x`与`y`中只有一个为非0时得到1，否则结果为0。同样，==可以作为逻辑等于（EQV）运算符。 □

例 一个常见的C语言编程错误是该用==运算符（判等）时写成=运算符(赋值)。另外几种编程语言用=进行相等性比较。作为一种风格，如果测试表达式的值是否为0时需要使用赋值表达式，则最好显式写成“!=0”、明确意图。例如，下列循环不知是正确还是包含输入错误。

```
while (x = next_item()) {
/* Should this be "x==next_item()" ?? */
   …
}
```

如果原形式正确，则可以用下列方式明确意图：

```
while ((x = next_item()) != 0) {
   …
}
```

□

参考章节 对齐限制 5.6.4，6.1.3；按位运算符 7.6.6；兼容类型 5.11；逻辑运算符 7.5.4，7.7；左值 7.1；null指针 5.3.2；指针类型 5.3；优先级 7.2.1；赋值运算符= 7.9.1；类型限定符 4.4.3；普通二元转换 6.3.4；**void*** 5.3.1

7.6.6 按位运算符

二元运算符&、^和 |分别指定按位与、按位异或和按位或功能，都是左结合的，用不同优先级确定表达式求值顺序。操作数应为整型，并进行普通二元转换。结果类型为转换的操作数类型，且不是左值：

bitwise-or-expression (按位或表达式):
    bitwise-xor-expression
    bitwise-or-expression | *bitwise-xor-expression*

bitwise-xor-expression (按位异或表达式):
    bitwise-and-expression
    bitwise-xor-expression ^ *bitwise-and-expression*

bitwise-and-expression (按位与表达式):
    equality-expression
    bitwise-and-expression & *equality-expression*

这些运算符结果中的每一位等于两个转换的操作数相应位的布尔函数。

- &函数在两个参数都是1时得到1，否则得到0。
- ^函数在一个参数为1、另一个参数为0时得到1，否则得到0。
- |函数在两个参数中有一个参数是1时得到1，否则得到0。

下面是其真值表。

a	b	a&b	a^b	a|b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	0	1

每个按位运算符是具有交换律和结合律的，编译器可以按照7.12节的限制重新排列包含运算符对象的表达式。

236

为了得到可移植代码，建议只对无符号类型使用按位运算符。在大部分采用对二的补码表示法表示带符号数的计算机中，对带符号操作数进行按位运算不会造成问题，但在其他计算机上可能失败。

编程人员要格外小心，不要用按位运算符&和|代替逻辑与运算符&&和逻辑或运算符||。按位运算符只有在参数为没有副作用的布尔值（0或1）时才能与相应逻辑运算符得到相同结果。另外，按位运算符总是求值两个操作数，而逻辑运算符在能够用左操作数确定最后表达式结果时就不再求值右操作数。

例　如果**a**为2，**b**为4，则**a&b**为0（false），而**a&&b**为1（true）。 □

7.6.7 整数集合示例

下面介绍“整数集合”程序包的使用、声明和定义。它用按位运算符将集合实现为位向量。示例包括样本程序（**testset.c**）、测试程序输出、包头文件（**set.h**）和包中函数的实现（**set.c**）。

参考章节　整数类型　5.1；逻辑运算符&&与||　7.7；　左值　7.1；求值顺序　7.12；关系运算符　7.6.4；　带符号类型　5.1.1；无符号类型　5.1.2；普通二元转换　6.3.4

237

```
#include "set.h"
int main(void)
{
    print_k_of_n(0, 4);
    print_k_of_n(1, 4);
    print_k_of_n(2, 4);
    print_k_of_n(3, 4);
    print_k_of_n(4, 4);
    print_k_of_n(3, 5);
    print_k_of_n(3, 6);
    return 0;
}
```

SET包的使用示例文件**testset.c**

```
All the size-0 subsets of {0, 1, 2, 3}:
{}
The total number of such subsets is 1.

All the size-1 subsets of {0, 1, 2, 3}:
{0}  {1}  {2}  {3}
The total number of such subsets is 4.

All the size-2 subsets of {0, 1, 2, 3}:
{0, 1}  {0, 2}  {1, 2}  {0, 3}  {1, 3}  {2, 3}
The total number of such subsets is 6.

All the size-3 subsets of {0, 1, 2, 3}:
{0, 1, 2}  {0, 1, 3}  {0, 2, 3}  {1, 2, 3}
The total number of such subsets is 4.

All the size-4 subsets of {0, 1, 2, 3}:
{0, 1, 2, 3}
The total number of such subsets is 1.

All the size-3 subsets of {0, 1, 2, 3, 4}:
{0, 1, 2}  {0, 1, 3}  {0, 2, 3}  {1, 2, 3}
```

```
{0, 1, 4}  {0, 2, 4}  {1, 2, 4}  {0, 3, 4}
{1, 3, 4}  {2, 3, 4}
The total number of such subsets is 10.

All the size-3 subsets of {0, 1, 2, 3, 4, 5}:
{0, 1, 2}  {0, 1, 3}  {0, 2, 3}  {1, 2, 3}
{0, 1, 4}  {0, 2, 4}  {1, 2, 4}  {0, 3, 4}
{1, 3, 4}  {2, 3, 4}  {0, 1, 5}  {0, 2, 5}
{1, 2, 5}  {0, 3, 5}  {1, 3, 5}  {2, 3, 5}
{0, 4, 5}  {1, 4, 5}  {2, 4, 5}  {3, 4, 5}

The total number of such subsets is 20.
```

SET包：文件**testset.c**的输出 238

```
/* set.h
A set package, suitable for sets of small integers in the
range 0 to N-1, where N is the number of bits in an unsigned
int type. Each integer is represented by a bit position; bit
i is 1 if and only if i is in the set. The low-order bit is
bit 0. */

#include <limits.h> /* defines CHAR_BIT */

/* Type SET is used to represent sets. */

typedef unsigned int SET;

/* SET_BITS: Maximum bits per set. */

#define SET_BITS              (sizeof(SET)*CHAR_BIT)

/* check(i): True if i can be a set element. */

#define check(i)            (((unsigned) (i)) < SET_BITS)

/* emptyset: A set with no elements. */

#define emptyset ((SET) 0)

/* add(s,i): Add a single integer to a set. */

#define add(set,i)          ((set) | singleset (i))

/* singleset(i): Return a set with one element in it. */

#define singleset(i)(((SET) 1) << (i))

/* intersect: Return intersection of two sets. */

#define intersect(set1,set2 )((set1) & (set2))

/* union: Return the union of two sets. */

#define union(set1,set2     )((set1) | (set2))

/* setdiff: Return a set of those elements in set1 or set2,
   but not both. */

#define setdiff(set1,set2   )((set1) ^ (set2))

/* element: True if i is in set. */

#define element(i,set       )(singleset((i)) & (set))
```

SET包：头文件**set.h**(1/2) 239

```
/* forallelements:  Perform the following statement once for
   every element of the set s, with the variable j set to
   that element. To print all the elements in s, just write
   int j;
   forallelements(j, s)
      printf("%d ", j);
*/

#define forallelements(j,s) \
   for ((j)=0; (j)<SET_BITS; ++(j)) if (element((j),(s)))

/* first_set_of_n_elements(n): Produce a set of size n whose
   elements are the integers from 0 through n-1.  This
   exploits the properties of unsigned subtractions. */

#define first_set_of_n_elements(n)(SET)((1<<(n))-1)

/* next_set_of_n_elements(s): Given a set of n elements,
   produce a new set of n elements.  If you start with the
   result of first_set_of_n_elements(k), and then at each
   step apply next_set_of_n_elements to the previous result,
   and keep going until a set is obtained containing m as a
   member, you will have obtained a set representing all
   possible ways of choosing k things from m things. */

extern SET next_set_of_n_elements(SET x);

/* printset(s): Print a set in the form "{1, 2, 3, 4}". */

extern void printset(SET z);

/* cardinality(s): Return the number of elements in s. */

extern int cardinality(SET x);

/* print_k_of_n(k,n): Print all the sets of size k having
   elements less than n.  Try to print as many as will fit
   on each line of the output.  Also print the total number
   of such sets; it should equal n!/(k!(n-k)!)
   where n! = 1*2*...*n. */

extern void print_k_of_n(int k, int n);
```

240

SET包：头文件**set.h**(2/2)

```
#include <stdio.h>
#include "set.h"

int cardinality(SET x)
{

/* The following loop body is executed once for every 1-bit
   in the set x.  Each iteration, the smallest remaining
   element is removed and counted. The expression (x & -x)
   is a set containing only the smallest element in x, in
   twos-complement arithmetic. */

    int count = 0;
    while (x != emptyset) {
```

```
        x ^= (x & -x);
        ++count;
    }
    return count;
}

SET next_set_of_n_elements(SET x)
{
/* This code exploits many unusual properties of unsigned
   arithmetic.  As an illustration:
     if x == 001011001111000, then
     smallest            == 000000000001000
     ripple              == 001011010000000
     new_smallest        == 000000010000000
     ones                == 000000000000111
     the returned value == 001011010000111
   The overall idea is that you find the rightmost
   contiguous group of 1-bits. Of that group, you slide the
   leftmost 1-bit to the left one place, and slide all the
   others back to the extreme right.
   (This code was adapted from HAKMEM.) */

    SET smallest, ripple, new_smallest, ones;
    if (x == emptyset) return x;
    smallest     = (x & -x);
    ripple       = x + smallest;
    new_smallest = (ripple & -ripple);
    ones         = ((new_smallest / smallest) >> 1) - 1;
    return (ripple | ones);
}
```

SET包：文件**set.c**(1/2) 241

```
void printset(SET z)
{
    int first = 1;
    int e;
    forallelements(e, z) {
        if (first) printf("{");
        else printf(", ");
        printf("%d", e);
        first = 0;
    }
    if (first) printf("{"); /* Take care of emptyset */
    printf("}");            /* Trailing punctuation */
}


#define LINE_WIDTH 54

void print_k_of_n(int k, int n)
{
    int count = 0;
```

```
    int printed_set_width = k * ((n > 10) ? 4 : 3) + 3;
    int sets_per_line = LINE_WIDTH / printed_set_width;
    SET z = first_set_of_n_elements(k);
    printf("\nAll the size-%d subsets of ", k);
    printset (first_set_of_n_elements(n));
    printf(":\n");
    do {       /* Enumerate all the sets. */
       printset(z);
       if ((++count) % sets_per_line) printf (" ");
       else printf("\n");
       z = next_set_of_n_elements(z);
    }while ((z != emptyset) && !element(n, z));
    if ((count) % sets_per_line) printf ("\n");
    printf("The total number of such subsets is %d.\n",
       count);
}
```

SET包：文件**set.c**(2/2)

7.7 逻辑运算符表达式

逻辑运算符表达式用逻辑运算符**&&**或**||**分隔两个表达式。在其他语言中，这些运算符也称为条件与运算符和条件或运算符，因为能够用左操作数确定最后表达式结果时就不再求值右操作数。

logical-or-expression (逻辑或表达式):
　　logical-and-expression
　　logical-or-expression || *logical-and-expression*

logical-and-expression (逻辑与表达式):
　　bitwise-or-expression
　　logical-and-expression && *bitwise-or-expression*

逻辑运算符接受任何标量类型的操作数，两个操作数的类型之间没有联系，各自独立进行普通一元转换。逻辑运算的结果为**int**类型，取值为0或1，不是左值。

逻辑与运算符　**&&**先完全求值左操作数，如果左操作数等于0（从**==**运算符意义上），则不求值右操作数，结果为0。如果左操作数不等于0，则求值右操作数。如果右操作数为0，则结果为0，否则结果为1。

逻辑或运算符　**||**先完全求值左操作数，如果左操作数不等于0（从**!=**运算符意义上），则不求值右操作数，结果为1。如果左操作数等于0，则求值右操作数。如果右操作数不为0，则结果为1，否则结果为0。

例　赋值语句**r = a && b**等价于

```
if (a == 0) r = 0;
else {
   if (b == 0) r = 0;
   else r = 1;
}
```

赋值语句**r = a || b**等价于

```
if (a != 0) r = 1;
else {
    if (b != 0) r = 1;
    else r = 0;
}
```

□ 242 ↓ 243

例 下面是一些逻辑运算符的例子。

a	b	a&&b	b求值否?	a || b	b求值否?
1	0	0	是	1	否
0	34.5	0	否	1	是
1	"Hello\n"	1	是	1	否
'\0'	0	0	否	0	是
&x	y=2	1	是	1	否

□

这两个逻辑运算符在语法上都是左结合的，但这和编程人员的关系不大，因为这些运算符在词法上是完全结合的，没有其他相同优先级的运算符，运算符&&的优先级比||高，但逻辑表达式直接使用括号通常能使程序更易读。

例 表达式

```
a<b || b<c && c<d || d<e
```

等价于更明确的形式

```
a<b || (b<c && c<d) || d<e
```

参考章节 按位运算符&和| 7.6.6；判等运算符==和!= 7.6.5；左值 7.1；指针类型 5.3；优先级 7.2.1；标量类型 第5章；普通一元转换 6.3.3

7.8 条件表达式

?与:运算符引入条件表达式，其优先级低于二元表达式，而且是右结合的：

conditional-expression (条件表达式):
    logical-or-expression
    logical-or-expression ? *expression* : *conditional-expression*

条件表达式有3个操作数，第一个操作数应为标量类型，第二个和第三个操作数可以是不同类型，进行普通一元转换（如果求值）。结果类型取决于第二和第三个操作数的类型。表7-5列出了传统C语言中允许的条件表达式的操作数类型，表7-6列出了标准C语言允许的条件表达式的操作数类型。条件表达式针对第一个和第三个操作数是右结合的，其结果不是左值，但一些标准化之前的C语言编译器将这个值转化为左值。 244

表7-5 传统C语言中允许的条件表达式的操作数类型

一个操作数类型	另一个操作数类型	结果类型
算术类型	算术类型	普通二元转换后的类型
结构类型或联合类型①	同一结构类型或联合类型	结构类型或联合类型
指针类型	同一指针类型或0	指针类型

① 这些操作类类型在一些标准化前的编译器中不允许。

表7-6 标准C语言允许的条件表达式的操作数类型

一个操作数类型	另一个操作数类型	结果类型
算术类型	算术类型	普通二元转换后的类型
结构类型或联合类型	兼容结构类型或联合类型	结构类型或联合类型
void类型	**void**类型	**void**类型
类型T_1限定或未限定版本的指针	类型T_2限定或未限定版本的指针（如果T_1与T_2兼容）	复合指针类型①
T类型的指针②	限定或未限定的**void** *	**void** *①
任何指针类型	null指针常量	指针类型①

① 结果所指的类型具有两个操作数所指的类型的所有限定符。
② T应为对象或不完整类型。

条件表达式的执行过程如下：

1. 完全求值第一个操作数，并用0测试。
2. 如果第一个操作数不为0，则求值第二个操作数，将其值转换成结果类型，成为条件表达式的值。不求值第三个操作数。
3. 如果第一个操作数为0，则求值第三个操作数，将其值转换成结果类型，成为条件表达式的值。不求值第二个操作数。

例 表达式**r=a?b:c**等价于：

```
if (a != 0) r = b;
else r = c;
```

表达式

```
a ? b : c ? d : e ? f : g
```

解释为

245

```
a ? b : (c ? d : (e ? f : g))
```

□

例 在这个例子中，条件表达式的嵌套好像很有用，**signum**函数根据变量为正、负或0返回1、－1或0。

```
int signum(int x){ return (x > 0) ? 1 : (x < 0) ? -1 : 0; }
```

更复杂的情形最好用一个或几个if语句完成。作为一种风格，最好把条件表达式的第一个操作数放在括号中，但这不是必需的。 □

参考章节 算术类型 第5章；数组类型 5.4；浮点数类型 5.2；整数类型 5.1；左值 7.1；指针类型 5.3；优先级 7.2.1；标量类型 第5章；带符号类型 5.1.1；结构类型 5.6；联合类型 5.7；无符号类型 5.1.2；普通二元转换 6.3.4；普通一元转换 6.3.3；**void**类型 5.9

7.9 赋值表达式

赋值表达式包括赋值运算符分开的两个表达式，是右结合的。运算符=称为简单赋值运算符，其他都是复合赋值运算符：

assignment-expression (赋值表达式):

conditional-expression
unary-expression assignment-op assignment-expression

assignment-op : one of
`= += -= *= /= %= <<= >>= &= ^= |=`

赋值运算符的优先级相同，都是右结合（C语言中的所有其他二元运算符都是左结合的）。

例 表达式`x*=y=z`等价于`x*=(y=z)`，而不是`(x*=y)=z`；同样，表达式`x=y*=z`等价于`x=(y*=z)`，而不是`(x=y)*=z`。

赋值运算符的右结合律使多个赋值表达式具有“明显”的解释，即表达式`a=b=d+7`相当于`a=(b=(d+7))`，因此先将`d+7`的值赋予`b`，然后再赋予`a`。 □

每个赋值运算符的左操作数为可修改的左值，修改这个左值时在其中存储新值。运算符用不同方法计算新值。赋值表达式的结果不是左值。

参考章节 可修改的左值 7.1；优先级 7.2.1

246

7.9.1 简单赋值

一个等号=表示简单赋值。右操作数的值转换成左操作数的类型，然后存放在左操作数中。表7-7列出了允许的操作数类型。

表7-7 赋值操作数

左操作数类型	右操作数类型
算术类型	算术类型
结构类型或联合类型	兼容结构类型或联合类型
T的指针	T'的指针，其中T'与T兼容
`void *`类型	T的指针①
T的指针①	`void*`类型
任何指针	null指针常量

① 在标准C语言中，T应为对象或不完整类型。

C语言原始的定义不允许赋值结构类型和联合类型，一些早期的编译器可能仍然有这个限制。

在标准C语言中，还有与操作数的类型限定符相关的限制。首先，左操作数不能为`const`限定类型。其次：

1. 如果操作数为算术类型，则可以为限定类型或非限定类型。
2. 如果操作数为结构类型或联合类型，则应为限定类型或非限定类型的兼容类型。例如，它们应有相同限定的成员。
3. 如果操作数为对象指针或函数指针，则应为限定类型或非限定类型的兼容类型指针，左操作数所指的类型应当具有右操作数所指的类型的所有限定符。这样就可以防止把`const int *`指针赋予`int *`指针，避免因此修改常量整数。
4. 如果一个操作数是限定或非限定版本的`void *`类型，则另一操作数应为对象指针或不完整类型的指针。左操作数所指的类型应当具有右操作数所指的类型的所有限定符。原因同上。

赋值运算符的结果类型等于左操作数的未转换和未限定类型。赋值运算的结果是存放在左操作数中的值，不是左值。如果两个操作数都是算术类型，则用普通赋值转换将右操作数转换

成左操作数类型之后再赋值。

简单赋值运算符不能把一个数组的整个内容复制到另一数组。数组名不是可修改的左值，
247 因此不能放在赋值表达式左边。另外，赋值表达式右边的数组名要通过普通转换转换成第一个元素的指针，因此赋值时只复制指针，而不是复制数组内容。

例　可以用=运算符将数组地址复制到指针变量中。

```
int a[20], *p;
…
p = a;
```

本例中，a是整数数组，p是“整数的指针”类型。赋值使p指向a数组（的第一个元素）。

要复制整个数组，可以在结构类型或联合类型中嵌入数组，因为可以用简单赋值复制整个结构类型或联合类型：

```
struct matrix {double contents[10][10]; };
struct matrix a, b;
…
{
    /* Clear the diagonal elements. */
    for (j = 0; j < 10; j++)
        b.contents[j][j] = 0;
    /* Copy whole 10x10 array from b to a. */
    a = b;
}
```

□

简单赋值运算符的实现假设内存中右边值和左边的对象不共用存储空间（除非完全共用存储空间，如赋值语句x = x）。如果发生共用存储空间，则赋值行为是未定义的。

参考章节　算术类型　5.1，5.2；数组类型　5.4；普通赋值转换　6.3.2；左值　7.1；null 指针　5.3.2；指针类型　5.3；结构类型　5.6；类型兼容性　5.11；联合类型　5.7

7.9.2 复合赋值

复合赋值运算符可以简单地看成表达式“*a op=b*”等价于“*a = a op b*”，表达式*a*只求值一次。操作数允许的类型取决于使用的赋值运算符，见表7-8。

更准确地说，求值*op*=的左右操作数，其中左操作数要为可修改的左值。对两个操作数值采用运算符*op*表示的运算，包括运算符进行的任何“普通转换”。然后在进行普通赋值转换之后将得到的值存放在左操作数指定的对象中。

对复合赋值运算符，和简单赋值运算符一样，结果类型等于左操作数的未转换和未限定类型。结果是存放在左操作数中的值，不是左值。

表7-8　复合赋值表达式操作数允许的类型

赋值运算符	左操作数类型	右操作数类型
*= /=	算术类型	算术类型
%=	整数类型	整数类型
+= -=	算术类型	算术类型
+= -=	指针类型	整数类型

（续）

赋值运算符	左操作数类型	右操作数类型
<<= >>=	整数类型	整数类型
&=	整数类型	整数类型
^=	整数类型	整数类型
|=	整数类型	整数类型

在最早的C语言版本中，复合赋值运算符写成相反形式，等号放在运算前面，这样会产生语法歧义。例如，`x=-1`可以解释为`x= (-1)`或`x=x-1`。新的形式消除了这些问题。一些非标准C语言编译器继续支持早期的形式，以保证兼容性，并且把`x=-1`视为`x=x-1`，除非等号和负号之间加一个空格。

参考章节 算术类型 第5章；赋值转换 6.3.2；浮点数类型 5.2；整数类型 5.1；指针类型 5.3；带符号类型 5.1.1；无符号类型 5.1.2；普通二元转换 6.3.4；普通一元转换 6.3.3

7.10 顺序表达式

逗号表达式包括逗号分开的两个表达式。逗号运算符的语法是左结合的，但对编程人员关系不大，因为运算符在词法上刚好是完全结合的。注意，逗号表达式在C语言表达式语法树的开头：

comma-expression：
　　assignment-expression
　　comma-expression , *assignment-expression*

expression：
　　comma-expression

先要完全求值逗号运算符的左操作数，不一定产生任何值。如果产生一个值，则放弃这个值。然后求值右操作数。逗号表达式的结果类型与数值等于右操作数经过普通一元转换后的类型与数值。逗号表达式的结果不是左值。这样，语句“`r=(a,b,...,c);`”（注意括号是必需的）等价于“`a;b;...r=c;`”。差别在于可以在表达式上下文中使用逗号运算符，如在循环控制表达式中。

248 ~ 249

例 在`for`语句中，逗号运算符可以把几个赋值表达式合并成一个表达式，在一个循环中初始化或单步执行几个变量：

```
for( x=0, y=N; x<N && y>0; x++, y--) …
```

□

逗号运算符是可结合律，可以在一个表达式中包括多个逗号分开的表达式，子表达式按顺序求值，最后一个表达式的值成为整个表达式的值。

例 滥用逗号运算符可能引起混淆，在某些情况下，它会与逗号的其他用法发生冲突。例如，表达式

```
f(a, b=5,2*b, c)
```

总被认为是用4个参数调用函数`f`一样。参数表中的任何逗号表达式都要放在括号中：

```
f(a, (b=5,2*b), c)
```

□

其他要把逗号运算符放在括号中的上下文包括结构与联合声明符表中的字段长度表达式、

枚举声明符表中的枚举值表达式以及声明与初始化程序中的初始化表达式。逗号还用作预处理器宏调用中的分隔符。

逗号运算符保证操作数按从左向右顺序求值，而逗号的其他用法则没有这个保证。例如，函数调用中的参数表达式不一定按从左向右顺序求值。

参考章节 放弃表达式 7.13；枚举类型 5.5；**for**语句 8.6.3；函数调用 7.4.3；初始化表达式 4.6；左值 7.1；宏调用 3.3；结构类型 5.6；联合类型 5.7

7.11 常量表达式

在几种上下文中，C语言允许编写在编译时要求值为常量的表达式。每种上下文对表达式允许的形式有不同限制。常量表达式有3类：

1. 预处理器常量表达式，用作**#if**与**#elif**预处理器控制语句中的测试值。
2. 整型常量表达式，用作数组边界、结构中的位字段长度、显式枚举值和**switch**语句的**case**标号中的值。
3. 初始化程序常量表达式，作为静态与外部变量和（C99之前的）集合类型自动变量的初始化程序。

250

常量表达式不能包含赋值、自增、自减、函数调用和逗号表达式，除非放在**sizeof**运算符的操作数中。此外，只要符合下面几节中介绍的表达式类的限制条件，还可以出现任何字面值或运算符。这些限制是标准C语言要求的，传统实现对具体情形可能要求更松一些。

7.11.1 预处理器常量表达式

预处理器常量表达式在编译时求值，有比较严格的限制。这种表达式应为整型，可能只涉及整型常量、字符型常量和特殊**defined**运算符。在C99中，所有算术都是用宿主计算机中定义的类型完成的，根据操作数的符号性等价于目标类型**intmax_t**或**uintmax_t**。这些类型在**stdint.h**中定义，至少64位。在C99之前，标准C语言只要求用宿主计算机中定义的**long**或**unsigned long**类型完成所有运算符，这在宿主计算机与目标计算机有巨大差别时可能遇到问题。

预处理器常量表达式不能进行任何环境性查询，除非引用**float.h**、**limits.h**、**stdint.h**等文件中定义的宏，不允许类型转换，也不允许**sizeof**运算符。预处理器无法访问任何程序变量，即使用**const**限定符声明。

例 下列代码错误地检查目标计算机上的**int**类型是否大于16位：

```
#if 1<<16
/* Target integer has more than 16 bits (NOT!)*/
...
#endif
```

事实上，以上代码只是测试宿主计算机上的**long**类型表示（C89）或目标计算机上的**intmax_t**类型表示（C99）。测试目标类型长度的正确方式如下：

```
#include <limits.h>
...
#if UINT_MAX > 65535
   /* target integer has more than 16 bits */
   ...
#endif
```

□

预处理器要识别字符常量中的转义序列，但可以在字符常量转换成整数时使用源字符集或目标字符集。这样，表达式'\n'或'z'-'a'在if语句中的预处理器表达式中可能有不同值。编程人员使用源字符集或目标字符集不同的编译器时要注意这个问题。 251

宏扩展之后，如果预处理器常量表达式包含任何其他标识符，则分别换成常量0。这可能不是良好的规则，因为存在这种标识符通常表示编程错误。测试预处理器中是否定义了一个名称的更好方法是使用defined运算符或#ifdef与#ifndef命令。

编译器还可以接受其他形式的预处理器常量表达式，但使用这些扩展的程序无法移植。

参考章节 类型转换表达式 7.5.1；字符常量 2.7.3；字符集 2.1；defined运算符 3.5.5；枚举常量 5.5；转义符 2.7.5；float.h 5.2；#ifdef与#ifndef 3.5.3；intmax_t 21.5；limits.h 5.1；sizeof运算符 7.5.2；stdint.h 第21章

7.11.2 整型常量表达式

整型常量表达式，用作数组边界、结构中的位字段长度、显式枚举值和switch语句case标号中的值。整型常量表达式应为整型，可以包括整数常量、字符型常量和枚举常量。可以使用sizeof运算符和任何操作数。可以使用类型转换表达式，但只能把算术类型转换成整型（除非在sizeof的操作数中）。浮点数常量只能作为转换的中间操作数或在sizeof的操作数中。

不在预处理器命令中的常量表达式应当像在目标计算机上一样求值，包括字符型常量的值。

编译器还可以接受其他形式整型常量表达式，包括转换成整型的更一般的浮点数表达式，但使用这些扩展的程序无法移植。一些标准化之前的编译器不允许在常量表达式中进行任何类型的转换。编程人员如果要考虑这些编译器的移植性，最好避免在常量表达式中进行转换。

参考章节 位字段 5.6.5；类型转换表达式 7.5.1；枚举类型 5.5；浮点数常量 2.7.2；sizeof运算符 7.5.2；switch语句 8.7

7.11.3 初始化程序常量表达式

初始化程序常量表达式可以包括算术常量表达式和地址常量表达式。

算术常量表达式包括整型常量表达式，但也可以包括一般的浮点数常量（而不只是转换成整数或在sizeof中）和可以转换成任何算术类型（包括浮点数类型）的类型转换表达式。如果编译时在常量表达式中求值浮点数表达式，则实现使用的表示方法可能提供比目标环境更高精度或更大范围。因此，编译时浮点数表达式的值与程序执行期间求值时可能有所不同。这个规则反映了准确模拟外部浮点数实现的困难。除此之外，表达式应和目标计算机上一样求值。 252

地址常量表达式可以是null指针常量，例如(void *)0，也可以是静态或外部对象与函数的地址或静态或外部对象的地址加减一个整型常量表达式。建立地址时，可以使用地址运算符（&）、间接访问运算符（*）、下标运算符（[]）和成员选择运算符（.与->），但不能访问任何对象的值。也可以使用转换成指针类型的类型转换表达式。

编译器可以随意接收其他形式的初始化程序常量表达式，如更复杂的涉及多个地址的地址表达式，但使用这些扩展的程序无法移植。

标准C语言指出，实现可以随意在运行时进行初始化，也可以在编译时避免浮点数算术。但是，执行任何访问初始化变量的代码之前，可能很难进行这种初始化。

例 下面p和pf的初始化程序是地址常量表达式的例子。

```
static int a[10];
static struct { int f1, f2; } s;
extern int f();
int i = 3;
…
int *p[] = { &i, a, &a[0],
             (int *)((char *)&a[0]+sizeof(a)),
             &s.f2 };
int (*pf)() = &f;
```

□

参考章节　地址运算符& 7.5.6；数组类型 5.4；初始化程序 4.6；**sizeof**运算符 7.5.2；结构类型 5.6

7.12 求值顺序

一般来说，编译器可以重新布置表达式求值的顺序。这种重新布置可能包括按不是从左向右的顺序求值函数调用参数或二元运算符的两个操作数。二元运算符+、*、&、^与|是完全结合的和可交换的，编译器可以利用这个假设。例如，编译器求值**(a+b)+(c+d)**时，可以像写成**(a+b)+(b+c)**一样（假设所有变量具有相同的算术类型）。

交换性和结合性的假设对无符号操作数的&、^与|运算总是成立，但对于不同的带符号整数类型的表示方法，其对带符号操作数的&、^与|运算不一定总是成立。交换性和结合性的假设对*运算和+运算也可能不成立，因为不能确定所写的表达式顺序是否造成溢出。尽管如此，编译器仍可
253 以利用这个假设。涉及这些运算的任何表达式的重新布置都不能改变操作数的隐式类型转换。

例　要控制求值顺序，编程人员可以使用临时变量赋值。但是，一个好的优化编译器可能重新布置计算，例如：

```
int temp1, temp2;
…
/* Compute q=(a+b)+(c+d), exactly that way. */
temp1 = a+b;
temp2 = c+d;
q = temp1 + temp2;
```

□

例　本例中，两个表达式不是等价的，编译器不能将一个换成另一个，虽然其中一个是通过另一个重新布置而得到的。

```
(1.0 + -3) + (unsigned) 1; /* Result is -1.0 */
1.0 + (-3 + (unsigned) 1); /* Result is large */
```

第一个赋值语句很简单，产生所要的结果。第二个赋值语句产生大结果，因为普通二元转换使带符号值 – 3转换成大的无符号值$2^n - 3$，其中n是表示无符号整数的位数。然后加上无符号值1，结果转换成浮点数表示并加1.0，得到$2^n - 1$的浮点数表示。这个结果可能是编程人员所要的，也可能不是，但编译器必须保证不能通过加法重新布置使表达式更混乱。 □

根据语言定义，编译器可以同样自由地重新布置浮点数表达式。但是，浮点数表达式求值的顺序根据操作数的特定值可能对结果精度产生巨大影响。由于编译器无法预测操作数值，因此数字分析师希望编译器总是按所写的方式求值浮点数表达式，使编程人员能够控制求值顺序。

求值函数调用中的实际参数时，参数和函数表达式的求值顺序没有指定，但效果上好像是先选择一个参数，完全求值，然后选择另一个参数，完全求值，一直到求值完所有参数。二元表达式运算符的每个操作数和表达式**a[i]**中的**a**与**i**具有相同的限制与自由度。

例 本例中，变量**x**是字符指针的数组，被看作字符串数组。变量**p**是字符指针的指针，被看作字符串的指针。**if**语句的目的是确定**p**所指的字符串（称为**s1**）和后一个字符串（称为**s2**）是否相等（并将指针**p**移到数组中这两个字符串之外）。

```
char *x[10], **p=x;
…
if ( strcmp(*p++, *p++) == 0 ) printf("Same.");
```

254

当然，同一表达式中同一变量有两个副作用是不好的编程风格，因为副作用的顺序是未定义的，但编程人员有理由相信副作用的顺序并不重要，因为这两个字符串可以按任何顺序传递到**strcmp**。 □

序列点

在标准C语言中，如果连续序列点之间对一个对象修改多次，则结果是未定义的。*序列点*是程序执行序列中发生前面执行的所有副作用的点，此后不再发生任何副作用。下列条件出现序列点：

- 完整表达式结束时，即初始化表达式、表达式语句、**return**语句中的表达式和条件迭代或**switch**语句中的控制表达式（包括**for**语句中的每个表达式）之后
- **&&**、**||**、**?:**或逗号运算符的第一个操作数之后
- 函数调用中求值参数和函数表达式之后

根据这个规则，表达式**++i*++i**的值是未定义的，上例的**strcmp**也一样。

参考章节 加法运算符**+** 7.6.2；二元运算符 7.6；按位与运算符**&** 7.6.6；按位或运算符**|** 7.6.8；按位异或运算符**^** 7.6.7；逗号运算符 7.10；条件表达式**?:** 7.8；条件语句 8.5；表达式语句 8.2；函数调用 7.4.3；初始化表达式 4.6；迭代语句 8.6；逻辑与**&&**和逻辑或**||** 7.7；乘法运算符***** 7.6.1；**return**语句 8.9；**strcmp**函数 13.2；普通二元转换 6.3.4

7.13 放弃值

可以在3种上下文中只使用表达式而不用其数值：

1. 表达式语句
2. 逗号表达式第一个操作数
3. **for**语句的初始化表达式与自增表达式

在这些上下文中，我们放弃表达式的值。

放弃没有副作用的表达式的值时，编译器认为发生了编程错误并发出警告。副作用产生的运算包括赋值和函数调用。如果放弃表达式的主运算符没有副作用，则编译器也可能发出警告消息。 255

例

```
extern void f();
f(x);       /* These expressions do not */
i++;        /* justify any warning about */
```

```
a = b; /* discarded values.  */
```

以下这些语句虽然有效，但可能引起警告消息：

```
extern int g();
g(x);              /* The result of g is discarded. */
x + 7;             /* Addition has no defined side effects. */
x + (a *= 2);      /* The result of the last operation to be
                      performed, "+", is discarded. */
```

编程人员要避免放弃值的警告，可以使用转换成**void**类型的类型转换表达式，表示故意要放弃这个值：

```
extern int g();
(void) g(x); /* Returned value is purposely discarded */
(void)(x + 7); /* This is pretty silly, but presumably
                  the programmer has a purpose. */
```

□

放弃函数调用的值时，编译器通常不会发生警告消息，因为不返回结果的函数被声明为“返回**int**的函数”类型。尽管标准C语言向编译器提供了更多信息，但制造商可能想保持与旧代码兼容，因此不会发生警告消息。

如果编译器认为放弃表达式的主运算符没有副作用，则编译器可能不对这个运算符产生代码（这时其操作数成为放弃值，并且可能递归地受到相同的处理）。

参考章节　赋值　7.9；类型转换　7.5.1；逗号运算符　7.10；**for**语句　8.6.3；函数调用　7.4.3；表达式语句　8.2；**void**类型　5.9

7.14 优化内存访问

作为一般规则，编译器可以随意生成等价于所编写程序行为的任何代码，编译器显式地取得重新安排代码的一定自由度，见7.12节。如果编译器认为放弃表达式的主运算符没有副作用，则编译器可能不对这个运算符产生代码，见7.13节。

例　有些编译器还可以重新组织代码，不一定按程序指定的顺序对内存访问那么多次。例如，如果某个数组元素被引用多次，则编译器可能巧妙地安排这个元素只被读取一次，从而提高速
256 度。事实上，它把代码

```
int x,a[10];
…
x = a[j] * a[j] * a[j];    /* Cube the table entry. */
```

改写成代码

```
int x,a[10];
register int temp;
…
temp = a[j];
x = temp * temp * temp;  /* Cube the table entry. */
```

对大多数应用程序，包括几乎所有可移植应用程序，这种优化技术是件好事，因为这样不会改变实际计算行为而能把程序速度提高好几倍。但是，编写中断处理器和C语言中的其他某些机器相关程序时可能遇到问题。这时，编程人员要使用标准C语言类型限定符**volatile**控制某些内存访问。

参考章节 `volatile` 4.4.5

7.15 C++兼容性

sizeof表达式的改变

在C++中，不能在表达式中声明类型，如在类型转换表达式和`sizeof`表达式中。另外，一些`sizeof`表达式的值因为作用域改变和字符字面值类型而在C语言和C++中有所不同。

例

```
i = sizeof(struct S { … }); /* OK in C, not in C++ */
```

□

例 有时`sizeof(T)`的值可能不同，`T`可能重定义。

`sizeof('a')`在C语言中为`sizeof(int)`，而在C++中为`sizeof(char)`。

枚举常量`e`的`sizeof(e)`值在C语言中为`sizeof(int)`，而C++中可能不同。 □

参考章节 字符字面值 2.8.5；枚举类型 5.13.1；作用域差别 4.9.2；`sizeof`表达式 7.5.2

257

7.16 练习

1. 下列表达式哪些在传统C语言中有效？对有效的表达式，表达式为什么类型？假设`f`类型为`float`，`i`的类型为`int`，`cp`的类型为`char *`，`ip`的类型为`int *`。

(a) `cp+ 0x23`　　(f) `f==0`

(b) `i+f`　　(g) `!ip`

(c) `++f`　　(h) `cp && cp`

(d) `ip[i]`　　(i) `f%2`

(e) `cp?i:f`　　(j) `f+=i`

2. 假设`p1`与`p2`的类型为`char *`。不用自增与自减运算符，试改写下面两条语句：

(a) `*++p1=*++p2;`

(b) `*p1--=*p2--;`

3. 位掩码是个整数，由指定的二进制0和1序列组成。编写一个产生下列位掩码的宏。如果宏参数是常量，则结果也应为常量。可以假设整数使用对二的补码表示法，但宏不能依赖于整数使用多少位表示或计算机使用高位存储法或低位存储法。

(a) `low_zeroes(n)`，一个字的低*n*位都是0，其余位都是1。

(b) `low_ones(n)`，一个字的低*n*位都是1，其余位都是0。

(c) `mid_zeroes(width,offset)`，一个字的低*offset*位都是1，接着的*width*位都是0，其余位都是1。

(d) `mid_ones(width,offset)`，一个字的低*offset*位都是0，接着的*width*位都是1，其余位都是0。

4. `j++==++j`是否是有效表达式？`j++&&++j`呢？如果`j`以0值开始，每个表达式的结果如何？

5. 下表列出简单赋值表达式的左边与右边类型对，标准C语言中允许哪些组合？

	左边类型	右边类型
(a)	`short`	`signed short`
(b)	`char *`	`const char *`
(c)	`int (*)[5]`	`int (*)[]`
(d)	`short`	`const short`
(e)	`int (*)()`	`signed (*)(int x, float d)`
(f)	`int*`	`t *` (当`typedef int t`)

6. 如果变量x类型为`struct{int f;}`，变量y单独定义类型`struct{int f;}`，`x=y`在标准C语言中是否有效？

258

第8章 语　句

C语言提供了大多数代数编程语言中常见的各种语句，包括条件语句、循环语句和goto语句。我们先介绍一些一般语法规则，然后介绍各种语句。

statement (语句):
　　expression-statement
　　labeled-statement
　　compound-statement
　　conditional-statement
　　iterative-statement
　　switch-statement
　　break-statement
　　continue-statement
　　return-statement
　　goto-statement
　　null-statement

conditional-statement (条件语句):
　　if-statement
　　if-else-statement

iterative-statement (迭代语句):
　　do-statement
　　while-statement
　　for-statement

8.1 语句的一般语法规则

尽管熟悉ALGOL式语言的编程人员熟悉C语言语句，但仍然有一些语法差别容易造成混淆和错误。

和Pascal与Ada语言一样，分号通常出现在C语言的连续语句之间。但在C语言中，分号不是语句分隔符，而只是某些语句的语法构成部分。惟一不需要用分号终止的C语言语句是复合语句（或块），放在花括号中（**{}**），而不是放在**begin**与**end**关键字之间。

```
a = b;
{ b = c; d = e; }
x = y;
```

C语言语句的另一个规则是条件和迭代语句中的“控制”表达式要放在括号中。控制表达式后面没有“then”、“loop”或“do”之类的特殊关键字，表达式后面紧跟其他语句：

```
if (a<b) x=y;
while (n<10) n++;
```

最后，其他语言中的赋值语句在C语言中是赋值表达式，可以放在更复杂的表达式内，也可以放上分号，成为独立语句：

```
if ((x=y)>3) a=b;
```

参考章节 赋值表达式 7.9；复合语句 8.4；条件语句 8.5；迭代语句 8.6

8.2 表达式语句

只要在表达式后面放上分号就构成表达式语句。

expression-statement(表达式语句):
    expression ;

执行这类语句时，对表达式求值，然后放弃结果值（如有）。

表达式语句只在求值表达式能产生副作用时才有用，如对变量赋值或进行输入与输出。通
260 常，用在这类语句中的表达式为赋值、自增或自减运算以及函数调用表达式。

例

```
speed = distance / time;  /* assign a quotient */
++event_count;            /* Add 1 to event_count.*/
printf("Again?");         /* Call the function printf.*/
pattern &= mask;          /* Remove bits from pattern */
(x<y) ? ++x : ++y;        /* Increment smaller of x and y */
```

最后一条语句虽然有效，但用**if**语句可以表述得更清晰，如：

```
if (x < y) ++x;
else ++y;
```

□

如果表达式没有副作用并且其值可以放弃（见7.13节），则编译器不必求值表达式或其局部。

参考章节 赋值表达式 7.9；放弃表达式 7.13；表达式 第7章；函数调用 7.4.3；自增表达式 7.5.8, 7.4.4

8.3 标号语句

可以用标号标记任何语句，使控制通过**goto**或**switch**语句转移到这些语句。标号有3种，命名标号可以放在任何语句中，和**goto**语句一起使用。**case**标号和**default**标号只能放在**switch**语句体中的语句内：

labeled-statement(标号语句):
    label : *statement*

label:
    named-label
    case-label
    default-label

标号不能单独出现，而要和语句相联系。如果标号单独出现（如在复合语句末尾），则它将被连接到null语句。在C99中，语句和声明可以混合，但标号不能直接用于声明，而要连接到声

明前面的null语句。

命名标号将在介绍**goto**语句时详细介绍。**case**标号或**default**标号将在介绍**switch**语句时介绍。

参考章节 **goto**语句 8.10；null语句 8.11；**switch**语句 8.7

261

8.4 复合语句

复合语句是由花括号中0个或多个声明和语句列表共同构成。在C99中，语句和声明可以混合，早期C语言中则要求把声明放在语句之前。

compound-statement (复合语句):
{ *declaration-or-statement-list*$_{opt}$ }

declaration-or-statement-list :
declaration-or-statement
declaration-or-statement-list declaration-or-statement

declaration-or-statement :
declaration
statement

复合语句可以放在能够使用语句的任何地方，它建立一个新作用域或块，从而影响其中的任何声明或复合字面值。执行复合语句时，通常一次一个按顺序处理每个语句和声明。执行最后一个声明或语句之后，复合语句的执行停止。可以在到达结尾之前用**goto**、**return**、**continue**或**break**语句跳出复合语句。也可以用**goto**或**switch**语句跳转到复合语句中的某个标号，而不是从第一句开始进入复合语句。复合语句中的跳入和跳出可能影响其中的声明，见本节下面部分的介绍。

参考章节 **auto**存储类 4.3；**break**与**continue**语句 8.8；声明 第4章；**goto**语句 8.10；**register**存储类 4.3；**return**语句 8.9；作用域 4.2.1

复合语句中的声明

复合语句或其他块中声明的标识符称为块级标识符，这种声明称为块级声明。块级标识符的作用域从声明点延伸到块末尾。标识符在这个作用域中有效，除非由内部块中的同一标识符声明隐藏。在块中声明标识符通常是良好的编程习惯，因为限制变量作用域能使程序更容易理解。

块中不用存储类指定符时，声明的标识符假设为存储类**extern**（函数类型标识符）或**auto**（所有其他情形）。块中声明函数类型标识符时，只能使用存储类**extern**。

如果用存储类**extern**在块中声明变量或函数，则不分配存储空间，不允许初始化表达式。这个声明引用其他地方定义的外部变量或函数，可能在同一源文件中，也可能在不同源文件中。

如果块中用存储类**auto**或**register**声明变长数组以外的变量，则每次进入块时分配一个未定义值，每次退出块时收回。这样，变量的生存期延伸到整个块，而不是只从声明点开始。如果 262
变量声明中有初始化表达式，则每次在执行流中遇到声明时，求值初始化表达式并初始化变量。这通常只发生一次，但C99中可能发生多次，例如**goto**语句将控制从复合语句转移回到声明之前的位置。如果用**goto**或**switch**语句跳转到复合语句中声明后面的位置，则可能不求值初始化表达式，变量值仍然未定义。自动块级标识符的值不从块的上次执行传递到下次执行中。

C99中，块中声明的变长数组不像其他自动变量一样在进入块时分配存储空间，而是在遇到

声明和求值长度表达式时分配存储空间，在控制离开这个块时收回。因此，其生存期和作用域是相同的。变长数组不能初始化，不能从作用域之外跳到数组的作用域中（即声明之后），但可以从作用域中跳到声明之前的位置。这时，收回数组的存储空间再（可能用新的长度）重新分配存储空间。块中所有变长数组采用后分配先收回的原则，因此可以在过程调用堆栈中进行分配。

如果在块中用存储类**static**声明一个变量，则其在程序执行之前分配一次，就像任何其他静态变量一样。如果声明中有初始化表达式，则该表达式（应为常量）只在程序执行之前求值一次，变量在复合语句上次执行与下次执行之间保持数值不变。在C99中，初始化表达式也应为常量。

例　下列代码段如果将语句**L**：作为从复合语句外部跳转的目标，则无法工作，因为变量**sum**没有初始化。此外，要检查函数整个函数体才能判断这些跳转是否发生。

```
{
        extern int a[100];
        int i, sum = 0;
        …
    L:
        for (i = 0; i < 100; i++)
            sum += a[i];
        …
}
```

□

例　未标号复合语句作为**switch**语句体时不能按正常方式执行，只能通过把控制转移到其中的标号语句而执行。因此，在这种复合语句开头并不初始化**auto**或**register**变量，如果存在这样的初始化表达式，一定是错误的。

263

```
switch (i) {
          int sum = 0; /* ERROR! sum is NOT set to 0 */
case 1: return sum;
default: return sum+1;
}
```

□

参考章节　**auto**存储类　4.3；**extern**存储类　4.3；**goto**语句　8.10；初值　4.2.8；初始化表达式　4.6；**register**存储类　4.3；作用域　4.2.1；**static**存储类　4.3；**switch**语句　8.7；变长数组　5.4.5；有效性　4.2.2

8.5 条件语句

条件语句有两种形式：一种有**else**子句，一种没有**else**子句。C语言在**if**语句语法中不使用**then**关键字。

conditional-statement（条件语句）：
　　if-statement
　　if-else-statement

if-statement：
　　if (*expression*) *statement*

if-else-statement:
if (*expression*) *statement* **else** *statement*

对每种形式的**if**语句，首先求值括号中的表达式。如果这个值为非0（见8.1节），则执行括号后面的语句。如果控制表达式的值为0，而且有**else**子句，则执行**else**后面的语句；如果控制表达式的值为0，而没有**else**子句，则执行条件语句后面的语句。

在C99中，整个**if**语句构成一个块作用域，像子语句一样，但它们不是复合语句。这样可以限制使用复合字面值或类型名时作为副作用生成的对象作用域与类型作用域。

参考章节 复合字面值 7.4.5；控制表达式 8.1；类型名 5.1.2

8.5.1 多路条件语句

多路判定可以表示为一系列级联**if-else**语句，除最后一个**if**语句以外的每个**if**语句都有自己的**else**从句。这种系列可以看成如下： 264

```
if (expression1)
   statement1
else if (expression2)
   statement2
else if (expression3)
   statement3
...
else
   statementn
```

例 下面是一个三路判定：函数**signum**在参数小于0时返回－1，参数大于0时返回1，否则返回0：

```
int signum(int x)
{
   if (x > 0) return 1;
   else if (x < 0) return -1;
   else return 0;
}
```

试与7.8节使用条件表达式的**signum**版本比较。 □

switch语句处理特定类型的多路判定，其中表达式值与一组固定常量比较。

参考章节 **switch**语句 8.7

8.5.2 悬而未决的else问题

一个条件语句可能包含另一个条件语句，从而产生歧义。有时可能很难看出一个**else**子句属于哪一个条件语句。为了解决歧义性问题，习惯上认为**else**子句总是属于最靠近它的最内层的**if**语句。

例 为了演示这种歧义性，本例故意按错误的方式缩排：

```
if ((k >= 0) && (k < TABLE_SIZE))
   if (table[k] >= 0)
      printf("Entry %d is %d\n", k, table[k]);
else printf("Error: index %d out of range.\n",k);
```

265

不细心的读者可能以为**else**是外层**if**语句的替换选项，即下列测试为假时，应打印错误消息：

```
(k >= 0) && (k < TABLE_SIZE)
```

但如果把最后错误消息的文本改成如下：

```
else printf("Error: entry %d is negative.\n", k);
```

则可以看出编程人员希望在**table[k]>=0**为false时执行**else**部分。上述代码的第二种解释能够得到期望的结果，而第一种解释不行。要让第一种解释能够实现，就要引入复合语句：

```
if (k >= 0 && k < TABLE_SIZE) {
   if (table[k] >= 0)
      printf("Entry %d is %d\n", k, table[k]);
}
else printf("Error: index %d out of range.\n", k);
```

为了减少混淆，第二种解释也可以引入复合语句：

```
if (k >= 0 && k < TABLE_SIZE) {
   if (table[k] >= 0)
      printf("Entry %d is %d\n", k, table[k]);
   else printf("Error: entry %d is negative.\n",k);
}
```

□

如果**if**语句控制的语句总是放在花括号中，则可以消除混淆。但是，这样的规则可能会在程序中产生很多不必要的花括号。一种介于两者之间的良好编程风格是在**if**语句控制的语句不是表达式和null语句时才将其放在花括号中。

参考章节　复合语句　8.4；表达式语句　8.2；null语句　8.11

8.6 迭代语句

C语言中迭代语句有3种：

iterative-statement (迭代语句):
　　while-statement
　　do-statement
　　for-statement

while语句在每次执行一个语句之前测试退出条件。**do**语句在每次执行一个语句之后测试退出条件。**for**语句提供特殊语法，可以方便地初始化或更新一个或几个控制变量，同时测试退出条件。迭代语句中嵌入的语句也称为迭代语句体。

在C99中，每个迭代语句构成自己的块作用域，像子语句一样，但它们不是复合语句。这样可以限制使用复合字面值或类型名时作为副作用生成的对象作用域与类型作用域。

266　**参考章节**　复合字面值　7.4.5；控制表达式　8.1；类型名　5.1.2

8.6.1 while语句

C语言不在**while**语句中使用**do**关键字：

while-statement:
　　while (*expression*) *statement*

while语句执行时，首先求值控制表达式。如果结果为真（非0），则执行语句。然后整个

过程重复，再次求值控制表达式，如果结果仍为真，则再次执行语句。由于语句或表达式的副作用，使控制表达式的值不断改变。

如果控制表达式求值为假（0）或**return**、**goto**或**break**语句将控制转移到**while**语句体之外，则**while**语句执行完毕。**continue**语句也可以修改**while**语句的执行情况。

例 下列函数用**while**循环求出整数**base**的指数**exponent**，其中**exponent**为非负整数（不检查溢出）。这里使用的方法是重复求基数的平方，将指数译码成二进制形式以决定何时将结果与基数相乘。

可以注意到，**while**循环维持不变的条件，直到得到正确答案是**result**乘以**base**的**exponent**次幂。最终**exponent**为0，这个条件表示**result**具有正确值。

```
int pow(int base, int exponent)
{
    int result = 1;
    while (exponent > 0) {
        if ( exponent % 2 ) result *= base;
        base *= base;
        exponent /= 2;
    }
    return result;
}
```
□

例 **while**循环可以用null语句体：

```
while ( *char_pointer++ );
```

在这行代码中，++运算符将字符指针向前移动，直到遇到null字符，然后指针指向null后面的字符。这是寻找字符串结尾的简洁方法（注意测试表达式解释为***(char_pointer++)**而不是**(*char_pointer)++**，否则会把**char_pointer**所指的字符编码值加1）。□ 267

例 另一个常见方法是用两个指针复制字符串：

```
while ( *dest_pointer++ = *source_pointer++ );
```

这行代码的作用是复制字符串，直到遇到null字符并复制，然后终止复制操作。当然，在这行代码中，编程人员有理由相信目标区域足够大，能够容纳要复制的所有字符。□

参考章节 **break**与**continue**语句 8.8；控制表达式 8.1；**goto**语句 8.10；null语句 8.11；**return**语句 8.9

8.6.2 do语句

do语句与**while**语句不同之处在于，**do**语句至少将语句体执行一次，而**while**语句可能一次也不执行其语句体：

do-statement:
do *statement* **while** (*expression*) ;

do语句先执行嵌入语句，然后求值控制表达式。如果结果为真（非0），则整个过程重复，仍执行嵌入语句，然后再求值控制表达式。如果结果仍为真（非0），则整个过程继续重复。

如果控制表达式求值为假（0）或**return**、**goto**或**break**语句将控制转移到**do**语句体之

外，则**do**语句执行完毕。**continue**语句也可以修改**do**语句的执行情况。

C语言**do**语句类似于Pascal语言中的“repeat-until”语句，其不同之处在于它在控制表达式求值为false时终止执行，而Pascal语言中的“repeat-until”语句则是在控制表达式求值为true时终止执行。C语言在这方面更加一致：C语言中的所有迭代结构(**while**、**do**和**for**)均在控制表达式求值为false时终止执行。

例　下列程序段读取和处理字符，在处理完换行符之后停止：

```
int ch;
do
   process( ch = getchar());
while (ch != '\n');
```

要得到同样效果，可以把计算移到**while**语句的控制表达式中，但这样做显得意图不太明确：

```
int ch;
while( ch = getchar(ch),
       process(ch),
       ch != '\n' ) /*empty*/ ;
```

□

268

例　可以编写以null语句作为语句体的**do**语句：

```
do ; while (expression);
```

但这个循环通常写成**while**语句：

```
while (expression);
```

□

参考章节　**break**与**continue**语句　8.8；控制表达式　8.1；**goto**语句　8.10；null语句　8.11；**return**语句　8.9；**while**语句　8.6.1

8.6.3 for语句

C语言中的**for**语句比大多数其他语言中的“递增与测试”语句更加一般化。在介绍**for**语句执行方法之后，我们将举例说明**for**语句的用法：

for-statement:
　　for *for-expressions statement*

for-expressions:
　　(*initial-clause*$_{opt}$; *expression*$_{opt}$; *expression*$_{opt}$)

initial-clause:
　　expression
　　declaration　　*(C99)*

for语句包括**for**关键字和包含在括号中的3个用分号分开的表达式，然后是语句。括号中的3个表达式都是可选的，可以省略，但两个分号和外面的括号是必要的。

通常，第一个表达式初始化循环变量，第二个表达式测试循环是继续还是终止，第三个表达式更新循环变量（例如递增）。但是，原则上这些表达式可以进行在**for**控制结构框架中有用的任何计算。**for**语句执行方法如下：

1. 如果*initial-clause*是个表达式，则求值这个表达式并放弃其数值。如果*initial-clause*是个

声明（C99），则初始化声明的变量。如果*initial-clause*不存在，则不进行任何操作。

2. 如果存在第二个表达式，则当作控制表达式求值。如果结果为0，则**for**语句执行完毕，否则（如果数值不是0或省略第二个表达式）转入第3步。

3. 执行**for**语句体。

4. 如果存在第三个表达式，则求值这个表达式并放弃其数值。

5. 转回第2步。

269

如果控制表达式求值为假（0）或**return**、**goto**或**break**语句将控制转移到**for**语句体之外，则**for**语句执行完毕。在**for**语句中执行**continue**语句将使执行转入第4步。

C99中**for**语句构成自己的块作用域，像子语句一样，但它们不是复合语句。这样可以限制使用复合字面值或类型名时作为副作用生成的对象作用域与类型作用域。另外，**for**语句中的第一个表达式可以换成声明，可以声明和初始化一个或多个循环控制变量。这种变量的作用域延伸到**for**语句结束，包括循环控制中的第二个和第三个表达式。编写**for**语句时，经常需要这种控制变量，限制其范围能使C语言编译器更好地进行优化。

参考章节 **break**与**continue**语句 8.8；复合字面值 7.4.5；控制表达式 8.1；放弃表达式 7.13；**goto**语句 8.10；**return**语句 8.9；类型名 5.12；**while**语句 8.6.1

8.6.4 for语句应用

例 通常，**for**语句的第一个表达式用于初始化变量，第二个表达式以某种方式测试变量，第三个表达式按照某个目标修改变量。例如，要打印0~9的整数及其平方，可以编写下列代码：

```
int j;
...
for (j = 0; j < 10; j++)
   printf("%d %d\n", j, j*j);
```

这里，第一个表达式用于初始化变量**j**，第二个表达式测试变量是否到达10（如果是，则循环终止），第三个表达式递增**j**。

在C99中，变量**j**可以在循环中声明，因此其作用域限制在循环中：

```
for (int j = 0; j < 10; j++)
   printf("%d %d\n", j, j*j);
```

□

例 C语言可以用两种常见方法编写永不停止的循环（也称为“do forever循环”）：

```
for (;;) statement

while (1) statement
```

可以在循环体中用**break**、**goto**或**return**语句终止循环。 □

例 前面为演示**while**语句而用的**pow**函数可以用**for**循环改写如下：

270

```
int pow(int base, int exponent)
{
   int result = 1;
   for (; exponent > 0; exponent /= 2) {
      if ( exponent % 2 )
```

```
            result *= base;
        base *= base;
    }
    return result;
}
```

这个形式强调循环是由**exponent**变量控制的，通过不断用2除，使其逐步趋向0。注意循环变量**exponent**仍然要在**for**语句之外声明。**for**语句不包括任何变量的声明。一个常见的编程错误是忘记声明**for**语句中使用的变量**i**和**j**之类，结果发现程序其他地方的变量**i**和**j**被循环修改了。 □

例 下面是个简单排序程序，使用插入排序算法：

```
void insertsort(int v[], int n)
{
    register int i, j, temp;
    for (i = 1; i < n; i++) {
        temp = v[i];
        for (j = i-1; j >= 0 && v[j] > temp; j--)
            v[j+1] = v[j];
        v[j+1] = temp;
    }
}
```

外层**for**循环从1（包括）到**n**（不包括）计算**i**，每一步处理时，元素**v[0]**到**v[i-1]**已经排序，而元素**v[i]**到**v[n-1]**还没有排序。内层循环**j**的取值从**i-1**开始递减，将数组元素一步一步上移，直到找到插入**v[i]**的正确位置（因此称为插入排序）。这个算法不适合很大的未排序数组，因为在最坏情况下，排序次数是**n*n**（即为$O(n^2)$）的比例。 □

例 可以把插入排序从$O(n^2)$改进到$O(n^{1.25})$，只要在前两个循环之外增加第三重循环，在**insertsort**使用常量1的几个地方引入**gap**。下列排序函数使用**shell**排序算法，与Kernighan和Ritchie的著作《*The C Programming Language*》中的**shell**排序的例子相似，但这
271 里进行了3处修改，其中两处是Knuth与Sedgewick提出的（参见前言），使速度更快：

```
void shellsort(register int v[], int n)
{
    register int gap, i, j, temp;
    gap = 1;
    do (gap = 3*gap + 1); while (gap <= n);
    for (gap /= 3; gap > 0; gap /= 3)
        for (i = gap; i < n; i++) {
            temp = v[i];
            for (j=i-gap; (j>=0)&&(v[j]>temp); j-=gap)
                v[j+gap] = v[j];
            v[j+gap] = temp;
        }
}
```

改进之处包括：(1)在原先的**shell**函数中，**gap**值从**n/2**开始，每次遍历外循环时除以2。而在这个版本中，**gap**初始化时寻找不大于**n**的序列（1、4、13、40、121、...）中的最小值，每次

遍历外循环时除以3。这样可以使排序加快20%～30%（这样选择**gap**作为初始值比用**n**作为初始值更合适）。(2)内循环中的赋值从3次减少到一次。(3)增加了**register**和**void**存储类。在有些实现中，**register**声明可以大大提高性能（有的高达40%）。 □

例 **for**语句不一定只能用来计数整数值。下面的例子扫描链接的结构链，其循环变量是指针：

```
struct intlist {
    struct intlist *link;
    int data;
};

void print_duplicates(struct intlist *p)
{
    for (; p; p = p->link) {
        struct intlist *q;
        for (q = p->link; q; q = q->link)
            if (q->data == p->data) {
                printf("Duplicate data %d", p->data);
                break;
            }
    }
}
```

结构**intlist**用于实现一个记录链表，每个记录包含一些数据。对于这种链表，函数**print_duplicates**打印表中每个冗余记录的数据。第一个**for**语句用正式参数**p**作为循环变量，在给定的表中向下扫描。遇到null指针时，循环终止。对每个记录，内层**for**语句检查其后面的所有记录，用相同方式在表中扫描指针**q**。 □ 272

参考章节 指针类型 5.3；**register**存储类 4.3；选择运算符**->** 7.4.2；结构类型 5.6；**void**类型 5.9

8.6.5 多个控制变量

有时**for**循环可以有多个控制变量。这种情况下，逗号运算符特别有用，因为可以用它将几个赋值表达式合并为一个表达式。

例 下列函数修改链接指针，逆转链表：

```
struct intlist { struct intlist *link; int data; };

struct intlist *reverse(struct intlist *p)
{
    struct intlist *here, *previous, *next;
    for (here = p, previous = NULL ;
         here != NULL ;
         next = here->link, here->link = previous,
           previous = here, here = next) /*empty*/ ;
    return previous;
}
```

□

例 下列函数**string_equal**接受两个字符串，如果它们相等，则返回1，否则返回0：

```
int string_equal(const char *s1, const char *s2)
{
    char *p1, *p2;
    for (p1=s1, p2=s2; *p1 && *p2; p1++, p2++)
        if (*p1 != *p2) return 0;
    return *p1 == *p2;
}
```

for语句在两个字符串中并行扫描两个指针变量。表达式**p1++**，**p2++**使两个指针都移到下一个字符。如果发现字符串中的差别，则用**return**语句终止执行整个函数，返回0。如果在某个字符串中遇到null字符（由表达式***p1 && *p2**确定），则循环正常终止，第二个**return**语句确定两个字符串是否在同一位置以null字符结束（如果用表达式***p1**而不用***p1 && *p2**，则函数仍然正常工作，而且运行速度更快一些，但没有这么对称）。 □

参考章节 **break**语句与**continue**语句 8.8；逗号运算符 7.10；指针类型 5.3；选择运算符**->** 7.4.2；结构类型 5.6

273

8.7 switch语句

switch语句是基于控制表达式数值的多路分支语句，用法和Pascal与Ada语法中的“case”语句相似，但其实现方法更像FORTRAN的“computed goto”语句：

switch-statement:
　　switch (*expression*) *statement*

case-label:
　　case *constant-expression*

default-label:
　　default

关键字**switch**后面的控制表达式应为整型，进行普通一元转换。**case**关键字后面的表达式应为整型常量表达式（见7.11.2节）。**switch**语句中嵌入的语句也称为**switch**语句体，通常是复合语句，但这并不是必须的。

case标号与**default**标号属于所在的最内层**switch**语句。**switch**语句体中的任何语句（或语句体本身）可以标上**case**标号或**default**标号。事实上，同一语句可以标上多个**case**标号与**default**标号。**case**标号与**default**标号只能出现在**switch**语句体中，同一**switch**语句中的两个**case**标号不能包含相同数值的常量表达式。任何一个**switch**语句至多有一个**default**标号。**switch**语句的执行方法如下：

1. 求值控制表达式。
2. 如果控制表达式的值等于**switch**语句中某个**case**标号中的常量表达式值，则程序控制转移到这个**case**标号表示的点，就像**goto**语句一样。
3. 如果控制表达式的值不等于任何**case**标号，但这个**switch**语句有一个**default**标号，则程序控制转移到这个**default**标号表示的点。
4. 如果控制表达式的值不等于任何**case**标号，又没有**default**标号，则不执行**switch**语

句体中的语句，程序控制转移到**switch**语句后面的语句。

比较控制表达式与**case**表达式时，**case**表达式转换成控制表达式的类型（进行普通一元转换）。

控制表达式与每个**case**表达式进行比较的顺序没有定义，实现比较的方法可能依赖于**case**表达式的个数与数值。编程人员通常假设**switch**语句实现为**if**语句的序列，与**case**表达式具有相同顺序，但事实不一定如此。 274

控制转移到**case**标号与**default**标号时，执行通过连续语句继续，忽略遇到的其他任何**case**标号或**default**标号，直到到达**switch**语句结尾或控制通过**goto**、**return**、**break**及**continue**语句转移到**switch**语句之外。

尽管标准C语言允许控制表达式为任何整数类型，但有些早期的编译器不允许使用**long**或**unsigned long**类型。标准C语言还允许实现限制**switch**语句中的**case**标号个数。C89中的极限为257个，C99中的极限为1023个，足够处理典型（8位）**char**类型中的所有值。

在C99中，如果任何可变修改类型的对象在任何**case**标号或**default**标号中有效，则这个对象的作用域要包括整个**switch**语句。可变修改类型的对象作用域不能只包括**switch**语句的一部分，除非这个作用域完全在**case**标号或**default**标号作用域之内。换句话说，不能在包含可变修改类型的对象的块中插入**case**标号或**default**标号。

参考章节 **break**语句与**continue**语句 8.8；常量表达式 7.11；**goto**语句 8.10；整型 5.1；标号语句 8.3；**return**语句 8.9；可变修改类型 5.4.5

switch语句的用法

通常，**switch**语句体是个复合语句，其内层顶层语句具有**case**标号与**default**标号。注意，**case**标号与**default**标号并不改变程序控制流程，这些标号不能阻止执行过程。可以在**switch**语句体中用**break**语句终止其执行。

例

```
switch (x) {
   case 1: printf("*");
   case 2: printf("***");
   case 3: printf("*****");
   case 4: printf("*******");
}
```

在上述**switch**语句中，如果**x**值为2，则打印9个星号。因为**switch**语句将控制转移到具有表达式2的**case**标号，执行参数为"**"的**printf**调用。然后执行参数为"***"的**printf**调用，最后执行参数为"*****"的**printf**调用。如果每次调用**printf**之后要终止执行**switch**体，则要使用**break**语句如下： 275

```
switch (x) {
   case 1: printf("*");
           break;
   case 2: printf("***");
           break;
   case 3: printf("*****");
           break;
   case 4: printf("*******");
```

```
            break;
    }
```

尽管本例中最后一个**break**语句在逻辑上是不必要的，但这样做是好的编程风格，能够防止后面在**switch**语句中增加第5个**case**语句时出现程序错误。 □

建议对**switch**语句的样式采取下列简单规则：语句体总是复合语句，属于**switch**语句的所有标号应放在复合语句中的顶层语句中（就像**goto**语句采用的样式准则）。此外，除第一个**case**标号与第一个**default**标号以外的每个**case**标号与**default**标号前面要放上两个项目之一：或者用**break**语句终止上一**case**中的代码，或者用一个注释语句表示上一段代码要延续下来。

尽管这是好的编程风格，但C语言定义并不要求语句体一定为复合语句，也不要求**case**标号与**default**标号只能放在复合语句中的顶层语句上，或**case**标号与**default**标号要按某些特定顺序出现或放在不同语句中。

例 下列代码段中，注释语句告诉读者**fatal**后面故意不用**break**语句：

```
...
case fatal:
    printf("Fatal ");
    /* Drops through. */
case error:
    printf("Error");
    ++error_count;
    break;
...
```

例 下面的**switch**语句本意是使这个简单程序段尽量高效：

```
if (prime(x)) process_prime(x);
else process_composite(x);
```

276 函数**prime**在参数是素数时返回1，在参数是合数时返回0。程序测试表明，**prime**大多数时候都是用小整数调用，为了避免调用**prime**的开销，代码改成用**switch**语句处理小整数，用**default**标号处理大整数。压缩代码之后，结果如下：

```
switch (x)
    default:
    if (prime(x))
        case 2: case 3: case 5: case 7:
            process_prime(x);
    else
        case 4: case 6: case 8: case 9: case 10:
            process_composite(x);
```

为了达到提高程序效率的目的，**switch**语句的形式就变得如此古怪。 □

8.8 break语句与continue语句

break语句与**continue**语句可以在**switch**语句的循环中改变控制流程。在达到相同目的的前提下，使用这两个语句比使用**goto**语句具有更好的编程风格：

break-statement:
    break;

continue-statement:
    continue;

执行**break**语句使所在的最内层**while**、**do**、**for**或**switch**语句执行终止。程序控制立即转移到所终止语句之外的点。在没有迭代或没有**switch**语句的场合出现**break**语句是错误的。

continue语句终止所在的最内层**while**、**do**或**for**语句体的执行。程序控制立即转移到语句体的末尾，所影响的迭代语句从重新求值循环测试的点（对**for**语句为自增运算表达式）开始执行。在没有迭代语句的场合出现**continue**语句是错误的。

continue语句与**break**语句不同，没有与**switch**语句的交互。**continue**语句可以放在**switch**语句中，但只影响所在的最内层迭代语句，不影响**switch**语句。

例 **break**语句与**continue**语句可以用**goto**语句来解释。下面是**break**语句与**continue**语句影响的语句：

277

```
while ( expression ) statement
do statement while ( expression );
for (expression1; expression2; expression3) statement
switch ( expression ) statement
```

没没所有这些语句改写如下：

```
  while (expression) {statement C:;} B:;}
  do {statement C:;} while (expression); B:;}
{ for (expression1; expression2; expression3) {statement C:;} B:;}
{ switch (expression) statement B:; }
```

其中C是所在函数中的标号，这样，这些语句体中的任何**break**语句等价于“**goto B;**”，这些语句体中的任何**continue**语句等价于“**goto C;**”（**switch**除外，不允许）。这里假设循环体不包含另一个包含**break**或**continue**的循环。 □

例 **break**语句经常在两种重要情境中使用：终止处理**switch**语句中的特定**case**标号和提前终止循环。第一种用法见8.7节介绍**switch**时的例子，第二种用法见这个例子，在数组中填入输入字符，在数组已满或输入结束时终止：

```
#include <stdio.h>
static char array[100];
int i, c;
...
for (i = 0; i < 100; i++) {
    c = getchar();
    if (c == EOF) break;    /* Quit if end-of-file. */
    array[i] = c;
}
/* Now i is the actual number of characters read. */
```

注意**break**语句是如何处理异常情况的。正常情况最好在循环测试中处理。 □

例 下面的例子在永久循环中使用**break**语句，目的是尽量有效地寻找长度为**N**的数组**a**中

[278] 最小的元素。这里假设数组可以临时修改：

```
int temp = a[0];
register int smallest = a[0];
register int *ptr = &a[N];  /* just beyond end of a */
...
for (;;) {
    while (*--ptr > smallest) ;
    if (ptr == &a[0]) break;
    a[0] = smallest = *ptr;
}
a[0] = temp;
```

大多数工作都是在一个紧凑的**while**循环中完成的，其中在数组中向下扫描指针**ptr**，跳过大于目前所找到的最小元素的元素（如果元素的顺序随机，则找到一个较小的元素之后，大多数元素都会比它大，因此都会跳过）。**while**循环不能超出数据前端，因为当前最小的元素存放在第一个数组元素中。**while**循环完成时，如果扫描已经到达数组前端，则**break**语句终止外循环，否则更新**smallest**和**a[0]**，再次进入**while**循环。 □

例 下面采用更简单更明显的方法完成上例：

```
register int smallest = a[0];
register int j;
...
for (j = 1; j < N; ++j)
    if (a[j] < smallest) smallest = a[j];
```

这个版本显然更容易理解。但是，每次对循环迭代时，要显式检查(**j<N**)是否超出数组前端，而不是用更巧妙的代码进行隐式检查。在效率要求很高的情况下，可以使用更复杂的代码，但通常应使用更简单更明显的方法。 □

参考章节 **do**语句 8.6.2; **for**语句 8.6.3; **goto**语句 8.10; **switch**语句 8.7; **while**语句 8.6.1

8.9 return语句

return语句终止当前函数，可能返回一个值：

return-statement :
    return *expression*$_{opt}$;

[279] 执行**return**语句时，终止当前函数，程序控制返回到调用点。

如果**return**语句中没有出现表达式，则函数的返回类型在C99中应为**void**，否则这个语句无效。C89中允许非**void**函数中省略表达式，但指出，如果函数调用要求返回数值，则其行为是未定义的。

如果**return**语句中有表达式，则函数的返回类型不能为**void**，否则这个语句无效。返回表达式进行转换，好像赋值为函数的返回类型。如果不能进行这种转换，则这个**return**语句无效。

如果程序控制到达函数体末尾而没有遇到**return**语句，则等于执行了没有表达式的**return**语句。如果函数具有非**void**返回类型，则其行为是未定义的。

例 许多编程人员对**return**语句的表达式加上括号，这不是必要的。他们也许是沿习了在

switch、**if**、**while**之类后面的表达式中加上括号的习惯。

```
int twice(int x) { return (2*x); }
```
□

参考章节 放弃值 7.13；函数调用 7.4.3；函数定义 9.1；函数返回类型一致性 9.8

8.10 goto语句

goto语句可以将控制转移到函数中任何语句：

goto-statement：
goto *named-label* **;**

named-label：
identifier

goto关键字后面的标识符应与当前函数中某个语句的命名标号相同。执行**goto**语句会把程序控制立即转移到函数中标号所指的点，执行这个命名标号的语句。

参考章节 标号语句 8.3

使用goto语句

C语言允许**goto**语句将控制转移到函数中任何语句，但某些类型的分支可能使程序混乱，可能阻碍编译器优化。为此，建议不要从**if**语句之外分支到**if**语句的“then”或“else”从句中；不要在“then”从句与“else”从句之间相互分支；不要从**switch**语句或迭代语句之外分支到这些语句体中；不要从复合语句之外分支到复合语句中。这些分支类型不仅要在使用**goto**语句时避免，也要在**switch**语句中使用**case**标号与**default**标号时避免。从复合语句之外分支到复合语句中会绕过复合语句开头声明的任何变量的初始化。一种好的编程风格是尽量用**break**、**continue**与**return**语句代替**goto**语句。 280

例 尽管格外小心，有时还是要用到**goto**语句。下例中，要从二维数组**a**中搜索数值**v**。如果找到，则用**goto**语句分支到双嵌套循环之外，保留循环变量**i**和**j**的值。

```
#include <stdio.h>
int i, j, v, a[N][M];
…
for (i=0; i++; i<N)
  for (j=0; j++; j<M)
    if (a[i][j] == v) goto found;
printf("a does not contain %d\n", v);
…
found:
  printf("a[%d][%d]==%d\n", i, j, v);
```
□

参考章节 **break**语句与**continue**语句 8.8；控制表达式 8.1；**if**语句 8.5；标号语句 8.3；**return**语句 8.9；**switch**语句 8.7

8.11 null语句

null语句只有一个分号：

null-statement:

;

null语句主要用于两种情形。第一种情况下，用null语句体作为迭代语句(**while**、**do**或**for**)的语句体。第二种情况下，终止复合语句的右花括号前面要用一个标号。右花括号前面不能直接放标号，而只能将标号与一个语句相连接。

例 下列循环不需要循环体，因为所有工作都在控制表达式中完成：

```
char *p;
…
while ( *p++ ); /* 找到字符串的结束位置*/
```

281 □

例 标号**L**放在null语句上：

```
if (e) {
    …
    goto L; /* 终止这个if分支*/
    …
    L:;}
else …
```

□

参考章节 **do**语句 8.6.2; **for**语句 8.6.2; 标号语句 8.3; **while**语句 8.6.1

8.12 C++兼容性

8.12.1 复合语句

C++不能绕过带初始化的声明而跳转入复合语句。

例

```
goto L; /* 在C语言中是合法的，但是不提倡；在C++中是非法的*/
{
    int i = 10;
L:
    …
}
```

□

参考章节 跳转入复合语句 8.4.2

8.12.2 循环中的声明

C99可以在**for**循环的*initial-clause*中声明变量，其作用域延伸到循环体结束。这与标准C++一致。一些较早版本的C++将这个变量的作用域扩展到循环末尾之外的所在函数或复合语句中。

8.13 练习

1. 改写下列语句，不用**for**、**while**或**do**语句。

(a) `for(n=A;n<B;n++) sum+=n;`

(b) `while(a<b) a++;`

(c) `do sum+=*p; while (++p < q);`

282 2. 下列程序执行完毕后，j的值是多少？

```
{   int j=1;
    goto L;
    {
        static int i = 3;
    L:
        j = i;
    }
}
```

3. 下列程序执行完毕后，**sum**的值是多少?

```
int i,sum = 0;
for(i=0;i<10;i++) {
    switch(i) {
    case 0: case 1: case 3: case 5: sum++;
    default: continue;
    case 4: break;
    }
    break;
}
```

第9章 函 数

本章介绍函数的使用，详细介绍函数声明与定义、指定正式参数与返回类型以及函数的调用。本书前面已经提到函数的一些信息：如果4.5.4节介绍了函数声明符，5.8节介绍了函数类型与声明。

自从C语言原始定义以来，函数的描述越来越复杂。标准C语言引入了新的更好的函数声明方法，用函数原型指定函数参数的更多信息。提供原型时，调用函数的操作与不提供原型时不同。尽管原型形式和非原型形式单独看都很容易理解，但同一函数中混合这两种情形时发生的情况则要通过更复杂的规则来确定（C++中必须使用原型）。

函数原型的存在性可以用函数声明符语法确定（见4.5.4节）。简单地说，传统C语言中不使用原型时：

1. 函数参数进行自动升级（普通参数转换）之后再调用函数。
2. 不检查参数类型和个数。
3. 任何函数均可取可变个数的参数。

相反，使用原型时：

1. 函数参数像赋值一样转换为正式参数的声明类型。
2. 参数类型和个数应与声明匹配，否则程序出错。
3. 函数取可变个数的参数时应显式指定，未指定参数要进行默认参数转换。

C语言程序中是否使用原型是复杂的移植性问题。为了保持与非标准实现兼容，应避免使用原型；为了保持与C++兼容，必须使用原型。可以用条件编译指令包括两种形式，但这样也很危险。下面几节介绍原型形式和非原型形式的函数声明，并讨论一些可移植性选项。

9.1 函数定义

函数定义引入新函数并提供下列信息：

1. 函数的返回值类型（如有）。
2. 正式参数类型与个数。
3. 函数在所定义文件之外的有效性。
4. 调用函数时要执行的代码。

函数定义的语法如下。函数定义只能放在C语言源文件或翻译单元的顶层：

translation-unit (翻译单元):
    top-level-declaration
    translation-unit top-level-declaration

top-level-declaration (顶层声明):
    declaration
    function-definition

function-definition (函数定义):
function-def-specifier compound-statement

function-def-specifier (函数定义说明符):
declaration-specifiers$_{opt}$ *declarator declaration-list*$_{opt}$

declaration-list (声明表):
declaration
declaration-list declaration

第4章讨论了其他顶层声明的语法。在C99之前，如果函数定义*declaration-specifiers*$_{opt}$中没有类型说明符，则假设为**int**。而C99要求必须有类型说明符。

在*function-def-specifier*中，声明符要包含*function-declarator*，指定在括号前面的函数标识符。4.5.4节介绍了函数声明符的语法，现重复如下：

286

function-declarator:
direct-declarator (*parameter-type-list*) (C89)
direct-declarator (*identifier-list*$_{opt}$)

parameter-type-list:
parameter-list
parameter-list , ...

parameter-list:
parameter-declaration
parameter-list , *parameter-declaration*

parameter-declaration:
declaration-specifiers declarator
declaration-specifiers abstract-declarator$_{opt}$

identifier-list:
identifier
identifier-list , *identifier*

如果指定所定义函数名的函数声明符包括*parameter-type-list*，则这个函数定义是原型形式的，否则是非原型形式或传统形式的。原型形式要在声明符中声明参数名和类型，声明符后面的*declaration-list*是空的。作为良好的风格，所有C语言函数定义都要写成原型形式。

在标准化之前的传统形式中，参数名在声明符中列出，类型在声明符后面*declaration-list*$_{opt}$中按任意顺序指定。所有参数都要在*declaration-list*中声明，但C99之前，被忽略的参数声明默认为**int**类型。C99把传统形式函数定义当作过时特性，即将来可能不再支持。

例

```
int f(int i, long j) { … }          /* 原型形式 */
int f(i,j) int i; long j; { … }     /* 传统形式 */
```

□

*function-def-specifier*的形式有一些限制。函数定义中声明的标识符应当用定义的声明符部分指定函数类型。即声明符要包含一个*function-declarator*，指定在括号前面的函数标识符。标

识符不能从**typedef**名称继承其“函数性”。

函数返回类型不能是数组类型或函数类型。

声明符要指定函数的参数名。如果声明符采用原型形式，则*parameter-declarations*应包括声明符而不是*abstract-declarator*。如果声明符不采用原型形式，则要包括*identifier-list*，除非函数不取任何参数。为避免标识符表与参数表之间的歧义性，参数名不能与有效的**typedef**名称同
287 名（较早的编译器中通常没有这个限制）。

参数声明中惟一允许的存储类说明符是**register**。

只有非原型形式允许$declaration\text{-}list_{opt}$，只能包括参数标识符的声明。一些传统C语言编译器允许其他声明（如结构或类型预定义），但这类声明的含义存在问题，最好放在函数体中。

例 为了演示这些规则，下面列出有效的函数定义：

定　义	解　释
`void f()` `{...}`	f函数不带参数，不返回数值（传统形式）

`int g(x, y)`
`  int x, y;`
`{...}`
	g函数带两个参数，返回一个整数结果（传统形式）

`int h(int x, int y)` `{...}`	h函数带两个参数，返回一个整数结果（原型形式）
`int (*f(int x))[]` `{...}`	f函数带一个整数参数，返回整数数组的指针（原型形式）

下面是无效的函数定义并指出了原因，假设typedef名称T声明为“**typedef int T();**”。

定　义	解　释
`int (*q)() {...}`	q是指针而不是函数
`T r {...}`	r不能从**typedef**名称继承其“函数性”
`T s() {...}`	声明s为返回函数的函数
`void t(int, double)` `{...}`	t的参数名在声明符中没有出现

`void u(int x, y)`
`  int y;`
`{...}`
	参数声明只是部分地采用原型形式

□

函数定义中允许的存储类说明符只有**extern**与**static**。**extern**表示函数可以从其他文件中引用，即函数名导出到连接程序中。**static**表示函数不可以从其他文件中引用，即函数名
288 不导出到连接程序中。如果函数定义中不出现存储类，则假设为**extern**。无论如何，函数总是从定义点到文件结束时有效，特别地，它在函数体内有效。

参考章节 声明符 4.5；**extern**存储类 4.3；函数声明 5.8；初始化声明 4.1；**static**存储类 4.3；类型指定符 4.4

9.2 函数原型

函数原型是用原型语法（*parameter-type-list*）写成的函数声明和函数定义。和传统函数声明

一样，函数原型声明函数返回类型。但与传统函数声明不同的是，函数原型还声明函数的正式参数类型和个数。所有现代C语言代码都要用原型编写。C99把较早的非原型形式看作过时形式。

根据函数有无参数、参数个数固定或参数个数不定的条件，函数原型分为3种基本形式：

1. 不带参数的函数要在参数类型表中包括一个类型说明符**void**。在函数定义中，空参数表与**void**的含义相同，但这是一种过时写法，应尽量避免。

例

```
extern int random_generator(void);
static void do_nothing(void) { }   /* void is optional */
```
□

2. 具有固定参数个数的函数在参数类型表中指定这些参数的类型。如果原型出现在函数声明中，则还可以包括参数名（我们认为这些参数名有助于记录函数）。参数名应出现在函数定义中。

例

```
double square(double x) { return x*x; }
extern char *strncpy(char *, const char *, size_t);
```
□

3. 参数个数或类型不定的函数先和前面一样指定固定参数的类型，然后加上逗号和省略号（...）。至少要有一个固定参数，否则无法用**stdarg.h**中的标准库函数引用这个参数表。

例 下面是参数个数不定的函数声明。参数名按照标准库要求的为实现者保留的方式进行拼写。

```
extern int fprintf( FILE *__file,
   const char *__format, ... );
```
□ 289

例 原型可以在任何函数声明符中使用，包括构成较复杂类型时。标准C语言声明**signal**（见19.6节）如下。

```
void (*signal(int sig, void (*func)(int siga)))(int siga);
```

声明**signal**函数带两个参数：**sig**是整数，**func**是一个**void**函数的指针，这个函数带一个整数参数**siga**。函数**signal**返回与第二个参数相同类型的指针（即取一个整数参数的**void**函数的指针）。**signal**声明更巧妙的写法如下：

```
typedef void sig_handler(int siga);
sig_handler *signal(int sig, sig_handler *func);
```

但是，实际定义信号处理函数时，根据函数定义规则，不能使用类型预定义名称**sig_handler**，而要将类型重复：

```
void new_signal_handler(int sig a) {...}
```

可以在同一声明中对有些声明符使用原型，有些声明符不使用原型。如果声明**signal2**如下：

```
typedef void sig_handler2();  /* not a prototype */
sig_handler2 *signal2(int sig, sig_handler2 *func);
```

则**signal2**函数的第二个参数不用原型形式，但**signal2**仍然是原型形式。
□

参考章节 函数声明符 4.5.4；函数声明 5.8；函数定义 9.1；**void**类型 5.9

9.2.1 何时存在原型

为了预测如何进行函数调用，编程人员一定要知道调用的函数（或函数类型）是否由原型

控制。函数调用由原型控制的条件如下：

1. 函数（或函数类型）声明有效，声明采用原型形式。

2. 函数定义有效，这个定义采用原型形式。

注意，任何函数原型的有效性是必需的，也许还有其他非原型声明或定义是有效的。

如果同一函数（或函数类型）有两个或多个原型声明，或既有原型声明又有原型定义，则声明和定义应按5.11.4节的规则兼容。

290 **参考章节**　兼容类型与复合类型　5.11

9.2.2 混合原型声明与非原型声明

尽管我们不提供同一函数混合原型声明与非原型声明，但标准C语言规定了这两种声明兼容的条件（见5.11.4节）。

如果提供的参数不符合函数定义，则函数调用的行为是未定义的。在传统C语言中，编程人员要负完全责任，保证调用符合定义，C语言可以帮助用户把参数转换成更小的更好管理的类型集。在标准C语言中，利用原型声明，编译器可以在调用时检查参数是否符合原型。

根据函数声明符是否出现，函数调用可以分别由原型声明、传统声明以及实际函数定义控制。调用与定义可以在一个源文件中，也可以在多个源文件中。如果一些函数调用不是由原型声明控制，则编程人员要负责保证这些调用中的参数符合函数定义。

例　一般来说，有不同的原型分别与非原型声明兼容。例如，假设C语言程序中有如下非原型声明：

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

则下面是一些兼容或不兼容的原型声明。

原　型	是否与int f()兼容	原　因
`extern double f(void);`	否	参数表没问题，但返回类型不兼容
`extern int f(int, float);`	否	在普通参数转换下，float变成double，这两个类型不兼容
`extern int f(double x);`	是	转换时不改变参数类型
`extern int f(int i, ...);`	否	原型不能包含省略号
`extern int f(float *);`	是	参数是不被转换的指针

□

一般来说，一个原型匹配一个非原型函数定义，这个原型有时也称为函数的*Miranda*原型，因为只有它指向函数定义。

在标准C语言中，函数带不定个数的参数时要用原型控制，即任何标准化前的函数声明如果
291 带不定个数的参数（如**printf**），都要用原型改写之后才能在标准C语言实现中使用。

例　假设C语言程序中出现下列非原型定义：

```
int f(x,y)
    float x;
    int y;
{…}
```

则下面是一些与这个定义兼容或不兼容的原型声明。

原　型	是否与其兼容	原　因
`extern double f(double x, int y);`	否	参数表没问题，但返回类型不兼容

（续）

原 型	是否与其兼容	原 因
`extern int f(float, int);`	否	第一个参数类型应为**double**
`extern int f(float, int, ...);`	否	原型不能包含省略号
`extern int f(double a, int b);`	是	这是惟一兼容原型，参数名不重要

□

参考章节 兼容类型 5.11; **printf** 15.11

9.2.3 正确使用原型

原型进行的参数检查不是牢不可破的。在分为多个源文件的C语言程序中，编译器无法检查是否一个函数的所有调用都由一个原型控制，同一函数的所有原型兼容或所有原型都匹配函数定义。

但是，如果编程人员遵循一些简单规则，则可以在实际应用中消除漏洞：

1. 每个外部函数应在头文件中有单个原型声明。单个原型声明消除了同一函数出现不兼容原型的可能性。
2. 每个具有函数调用的源文件应包括原型所在的头文件。这保证函数的所有调用由同一原型控制，编译器可以在调用时检查参数。
3. 函数定义所在的源文件也应包括头文件，使编译器能检查原型与声明匹配，从而保证所有调用匹配定义。

函数定义不一定采用原型形式。

使用静态函数时也要遵循类似的规则。要保证静态函数的原型形式声明出现在函数的任何调用之前和函数定义之前。 292

9.2.4 原型与调用规则

本节对于编译器实现者非常有用，同时也使其他编程人员能够了解函数原型的规则。函数原型的一个优点是可以使编译器产生更有效的函数调用序列。

例 根据传统C语言规则，即使函数定义成带有**float**类型的参数，编译器也别无选择，只是先把参数转换成**double**类型，然后调用函数，再在函数内将参数转换回**float**类型，并将其存放在参数中。在标准C语言中，如果编译器看到原型控制的函数调用：

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

则编译器有权不把参数转换成**double**类型，只要在实现**f**定义时在另一端使用相应假设：

```
int f(float x) {…}
```

□

这里的微妙之处是编译器不一定要保持与不是原型控制的函数调用兼容，因为**f**的非原型声明（或定义）可能与所指定的原型兼容。因此，标准C语言没有定义调用**f**而没有有效原型时发生的情形。编译器可以传递寄存器中的参数，即使非原型规则是传递一个堆栈中的所有参数。

另一方面，如果原型声明可能是传统方式声明式定义的函数的Miranda原型，则编译器要使用兼容的调用规则。

例 调用下列某个声明控制的函数**g**可能要用兼容方式实现：

```
extern short g();
```

```
extern short g(int,double);  /* Could be g's Miranda */
```

换句话说，如果标准C语言编译器遇到下列函数调用：

```
process( a, b, c, d );
```

其中没有有效原型，实际参数类型如下：

```
short a;
struct {int a,b;} b;
float *c;
float d;
```

293

则要像使用下列原型时一样实现函数调用：

```
int process(int, struct {int a,b;}, float *, double);
```

□

这个规则并不实际建立一个可能影响后面调用的原型。如果程序后面或另一源文件中出现**process**的第二个调用，这时**process**的参数是3个**double**类型的值，则像使用下列原型时一样实现第二个函数调用：

```
int process( double, double, double );
```

尽管这两个调用可能在执行时不兼容。

总结起来，编译器可以依赖一个原型控制函数的所有调用，条件是它看到函数调用由一个原型控制，而这个原型：

1. 包括一个参数类型，与普通参数转换不兼容（**char**、**short**以及它们的无符号变体或**float**）。
2. 包括省略号，表示可变参数表。

由于**char**类型与**short**类型转换成**int**类型在大多数计算机上的成本都很小，因此第一个规则主要用于**float**类型的参数。

第二个规则表示编译器的标准调用规则不一定要像传统C语言中一样支持可变参数表。例如，标准编译器可以在自己的标准规则中对任何函数的前4个（固定）参数字使用寄存器，其余参数用堆栈传递。这个规则可能不适合传统C语言，因为有些函数根据堆栈中连续传递的所有参数采用可变参数。任何取可变参数表的传统C语言函数（如**printf**）都要改写成具有原型之后再用标准C语言实现编译。

原型声明中出现存储类**register**时，被忽略，即不能用**register**改变函数的调用规则，只能用其作为函数体中的提示。

9.2.5 与标准C语言和传统C语言的兼容性

标准C语言已经非常普及，所有C语言程序都建议使用原型。如果有些异常情形要求与不提供原型的实现兼容，则可以不用原型，以保持与标准和传统C语言的兼容性。但是，使用标准C语言编译器时，要放弃其他类型检查。下面是用**PARMS**宏解决这个问题的方法：

```
#ifdef __STDC__
#define PARMS(x) x
#else
#define PARMS(x) ()
#endif
```

294

然后不是用原型声明

```
extern int f(int a, double b, char c);
```

而是编写下列声明（注意双层括号）：

```
extern int f PARMS((int a, double b, char c));
```

用传统实现编译时，预处理器将该行扩展成：

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

但标准C语言实现将其扩展如下：

```
extern int f (int a, double b, char c);
```

PARMS宏在函数定义中不能正确工作，因此要用传统语法编写相应函数定义，这也是标准C语言能够接受的：

```
int f(a, b, c)
   int a; double b; long c;
{
   …
}
```

标准C语言中的传统定义不会引起问题，只要函数的原型声明出现在源文件之前。

参考章节 **__STDC__**预定义宏 3.3.4

9.3 正式参数声明

在函数定义中，正式参数声明可以用原型语法或传统语法。

参数声明中可以出现的惟一一种存储类指定符是**register**，提示编译器这个参数经常用到，函数开始执行后最好将其放在寄存器中。何种参数类型可以标为**register**存储类的正常限制在此也适用（见4.3节）。

在标准C语言中，正式参数的作用域与函数体顶层声明的标识符作用域相同，因此不能用函数体中的声明隐藏或重新声明。当前还有些C语言实现允许这种属于编程错误的重新声明。 295

例 在下列函数定义中，如果用标准符合编译器编译，则声明**double x;** 是个错误。但是，有些非标准编译器允许这种声明，因此使函数体中无法访问参数**x**：

```
int f(x)
   int x;
{
   double x;  /* hides parameter!? */
   …
}
```

□

在标准C语言中，参数可以声明为除**void**以外的任何类型。但是，如果参数声明为“返回*T*的函数”类型，它会隐式地改写成“返回*T*的函数的指针”类型，如果参数声明为“*T*的数组”类型，则会改写为“*T*的指针”类型。参数声明中的数组类型可以不完整。不管使用原型定义还是传统定义，都会进行这些调整，这是与调用时的默认参数转换并行的，见6.3.5节。编程人员大多数情况下不需要知道这些参数类型改变，因为在函数中使用这些参数时可以认为它们已经声明了类型。

例 函数**FUNC**定义如下：

```
void FUNC(int f(void), int (*g)(void), int h[], int *j)
{
   int i;
```

```
    i = f();    /* OK */
    i = g();    /* OK */
    i = h[3];   /* OK */
    i = j[3];   /* OK */
    …
}
```

假设对**FUNC**进行下列调用：

```
extern int a(void), b[20];
…
FUNC( a, a, b, b );
```

在**FUNC**中，表达式**f**等价于**g**，**h**等价于**j**。 □

一些标准化之前的实现拒绝“返回*T*的函数”类型的参数声明，要求显式声明为“返回*T*的函数的指针”类型。

C99扩展了声明正式数组参数的语法，数组声明符的顶层方括号（**[]**）中可以出现*array-qualifier-list*。数组限定符（类型限定符）**const**、**volatile**与**restrict**支持数组类型与指针类型的等价性，即参数声明

296

```
T A[qualifier-list e]
```

等价于

```
T * qualifier-list A
```

例 C99声明

```
extern int f(int x[const 10]);
extern int g(const y[10]);
```

函数**f**中参数**x**被视作类型为**int * const**（**int**的常量指针），而**g**中的参数**y**被视作类型为**const int ***（常量的**int**指针）。 □

C99中的数组方括号中也可以使用**static**数组限定符。这是对C语言实现的优化提示，断言实际数组参数为非null，在进入函数时具有声明的长度和类型。如果没有这个数组限定符，则可能传递null指针作为数组参数的实际参数，使实现很难知道能否在进入函数时安全地预取了输入参数数组的内容。

最后，对原型（而不是函数定义）中的C99正式数组参数声明，可以把长度换成星号，表示实际参数是个变长数组。原型声明中数组长度使用任何非常量表达式时都和星号一样处理。函数定义要提供长度的非常量表达式。

正式参数作为指定（或改写）类型的局部变量，复制传入函数的相应参数值。参数可以赋值，但赋值只改变局部参数值，而不改变调用函数中的参数。声明为函数类型或数组类型的参数名根据改写规则是个左值，尽管这些类型的标识符通常不是lvalue。

传统C语言实现中允许参数声明段包括**typedef**、结构、联合或枚举类型声明。在标准C语言中，参数声明段可以定义的惟一名称是正式参数名，所有正式参数名都要定义（C99之前，**int**类型参数的定义是可选的）。如果参数用原型语法声明，则参数声明段应为空。

例

```
int process_record(r)
```

```
    struct { int a; int b; } *r; /* not Standard C */
{
    ...
}
```

传统C语言中这样通常是不良编程习惯。如果声明涉及参数，则声明要移到函数之外，使调用者也能使用。如果声明不涉及参数，则声明应移到函数体中。 □ 297

参考章节 *array-qualifier-list* 4.5.3；枚举类型 5.5；函数声明符 4.5.4；函数原型 9.2；不完整数组类型 5.4；**register**存储类 4.3；存储类说明符 4.3；结构类型 5.6；**typedef** 5.10；联合类型 5.7；变长数组 5.4.5；**void**类型 5.9

9.4 调整参数类型

本节只适用于传统C语言和标准C语言中不使用函数原型的情况。不用原型时，函数参数值要进行某些转换（升级）。这些转换的目的是简化和规范函数参数，称为普通参数转换（升级），见6.3.5节介绍。调用者需要这些参数转换，因此C语言函数要先把参数值转换成声明参数类型之后再执行函数体。例如，如果函数**F**声明为带有**short**类型的参数**x**，调用**F**时指定**short**类型的值，则调用以下列事件序列实现：

1. 调用者加宽参数类型**short**，使其完成**int**类型值。
2. 将**int**类型值传递到**F**。
3. **F**将**int**值缩小到**short**类型。
4. **F**在参数**x**中存储**short**类型值。

好在转换开销不大，特别是对整数。转换影响的参数类型包括**char**、**short**、**unsigned char**、**unsigned short**与**float**。

例 编程人员要注意，一些标准化之前的编译器无法在进入函数时进行必要的缩小运算。例如下列函数的参数类型为**char**：

```
int pass_through(c)
    char c;
{
    return c;
}
```

一些编译器实现这个函数时认为函数所带的参数为**int**类型，如：

```
int pass_through(c)
    int c;
{
    return c;
}
```

298

这种不正确的实现方法使参数值无法缩小为**char**类型。这样，**pass_through(0x1001)**返回**0x1001**值而不是**1**。这个函数的正确实现方法如下：

```
int pass_through(anonymous)
    int anonymous;
{
```

```
    char c = anonymous;
    return c;
}
```

□

参考章节　数组类型　5.4；浮点数类型　5.2；函数参数转换　6.3.5；函数定义　9.1；函数原型　9.2；函数类型　5.8；整数类型　5.1；左值　7.1；指针类型　5.3

9.5 参数传递规则

C语言只提供了通过数值调用的参数传递方法，即实际参数值复制到被调函数局部的存储区中。例如，可以用正式参数名作为赋值语句左边，但这时只能改变参数的局部拷贝。如果编程人员要让被调函数改变实际参数，则要显式地传递参数地址。

例　下列函数**swap**无法正确工作，因为**x**和**y**通过数值传递：

```
void swap(x, y)
/* swap: exchange the values of x and y */
/* Incorrect version! */
    int x, y;
{
    int temp;
    temp = x; x = y; y = temp;
}
...
swap(a, b); /* Fails to swap a and b. */
```

299 这个函数的正确实现要求显式地传递参数地址：

```
void swap(x, y)
/* swap - exchange the values of *x and *y */
/* Correct version */
    int *x, *y;
{
    int temp;
    temp = *x; *x = *y; *y = temp;
}
...
swap(&a, &b); /* Swaps contents of a and b. */
```

□

参数的本地存储区通常用堆栈实现。但是，将参数压入到堆栈中的顺序在语言中没有指定，语言也不阻止编译器传递寄存器中的参数。可以对正式参数名采用地址运算符**&**（除非正式参数用存储类**register**声明），因此取得地址时，该参数应在可寻址存储体中（注意，正式参数的地址是实际参数拷贝的地址，而不是实际参数的地址）。

编写带有可变数目参数的函数时，编程人员要用**varargs**或**stdarg**函数提高移植性。

参考章节　地址运算符**&**　7.5.6；函数原型　9.2；存储类**register**　4.3；**stdarg**函数　11.4；**varargs**函数　11.4.1

9.6 参数一致性

Pascal与Ada之类大多数现代编程语言都要检查函数的正式参数与实际参数的一致性，即参

数个数和各个参数的类型应相符。这个检查在标准C语言中用原型声明函数时也要进行。

例 下例中函数**sqrt**的调用不是用原型控制，因此，C语言编译器不需要警告编程人员**sqrt**的实际参数类型为**long**，而正式参数声明为类型**double**（事实上，如果调用与定义在不同源文件中，则编译器无法发出这个警告）。函数只是返回不正确的数值：

```
double sqrt( x )      /* not a prototype */
    double x;
{
    ...
}
long hypotenuse(x,y)
    long x,y;
{
    return sqrt(x*x + y*y);
}
```

□

如果标准C语言中用原型控制调用，则实际参数要转换成相应的正式参数类型。只有无法进行这个转换或参数个数与正式参数个数不符时，C语言编译器才会拒绝这个程序。

例 在上述**sqrt**定义中增加原型之后，例子能够正确工作。**long**类型的参数在编程人员不知道的情况下转换成**double**类型：

```
double sqrt( double x )     /* prototype */
{
    ...
}

long hypotenuse(x,y)
    long x,y;
{
    return sqrt(x*x + y*y);
}
```

作为好的编程风格，建议用显式类型转换将参数转换成所要的参数类型，除非这个转换只是重复普通参数转换。即可以把上例中的返回语句改写如下：

```
return  sqrt( (double) (x*x + y*y) );
```

□

一些C语言函数（如**fprintf**）所带的参数个数和类型是可变的。在传统C语言中，**varargs**库函数演变成提供了编写这种函数的可靠方式，但其用法是不可移植的，因为不同实现使用的**varargs**形式稍有不同。在标准C语言中，生成了类似的库机制**stdarg**，提供可移植性与可靠性。使用**stdarg**的函数要用带省略号形式（“**,...**”）的原型进行声明之后再进行调用，这样可以使编译器有机会准备适合的调用机制。

参考章节 实际参数转换 9.4；函数参数转换 6.3.5；函数原型 9.2；**fprintf** 15.11

9.7 函数返回类型

函数可以定义成具有除“*T*的数组”和“返回*T*的函数”以外任何类型的返回值。这两种情形要通过返回数组指针或返回函数的指针来处理。返回类型没有像正式参数一样的自动改写机制。

函数返回的值由**return**语句中的表达式指定，这个语句使函数终止。9.8节将介绍控制这个表达式的规则。

函数返回的值不是左值（通过值返回），因此函数调用不能作为赋值运算符左边的最外层表达式。

例

```
f() = x;    /* Invalid */
*f() = x;   /* OK if f returns a pointer of suitable type */
f().a = x; /* Invalid--not an lvalue ( Section 7.4.2) */
```

□

参考章节　数组类型　5.4；函数调用　7.4.3；函数参数　9.4；函数类型　5.8；左值　7.1；指针类型　5.3；**void**类型　5.9

9.8 返回类型一致性

如果函数声明非**void**类型的返回类型*T*，则**return**语句中出现的任何表达式类型都要能够通过赋值转换成*T*类型，事实上，标准C语言和传统C语言的**return**语句中都发生这种转换。

例　在声明返回类型int的函数中，语句

```
return 23.1;
```

等价于

```
return (int) 23.1;
```

相当于

```
return 23;
```

□

如果函数声明返回类型为**void**，则在函数的任何**return**语句中提供表达式都是错误。在需要数值的上下文中调用这个函数也是错误。早期的编译器不提供**void**类型，它们通常对不返回数值的函数省略类型说明符：

```
process_something()      /* probably returns nothing */
{
    ...
}
```

还可以定义自己的**void**类型以提高可读性（见4.4.1节）。

如果函数具有非**void**的返回类型，则C89允许没有表达式的**return**语句，即“**return;**”
302 （C99不允许这种**return**语句，C++也不允许）。这个规则是为了向下兼容不提供**void**类型的编译器。如果函数的返回类型为非**void**，则执行无参数的**return**语句时，实际返回的值是未定义的。因此，最好不要在需要数值的上下文中调用这个函数。

参考章节　调整正式参数　9.4；默认类型说明符　4.4.1；左值　7.1；**return**语句　8.9；**void**类型　5.9

9.9 主程序

所有C语言程序都要定义一个外部函数**main**。这个函数是程序的入口点，即程序启动时执行的第一个函数。这个函数返回时，程序终止，返回值表示程序成功与否，像库函数**exit**中的

用法一样。如果到达**main**函数体末尾而没有遇到返回语句，则视为执行**return 0;**。

标准C语言允许**main**函数定义0个或两个参数：

```
int main(void) { ... }
int main() { ... } /* also OK, but not recommended */
int main( int argc, char *argv[] ) { ... }
```

不声明参数时，不从外部环境向主程序中传递信息，但可以用**getenv**或**system**之类库函数从外部环境取得信息。

在C99之前，**main**的返回类型通常省略，默认为**int**，但现在不再允许这样。

声明参数时，这些参数是由运行环境设置的，不是C语言编程人员能够直接控制的。参数**argc**是用户或另一程序调用程序时提供的程序参数或选项个数。参数**argv**是表示程序参数的字符串的指针向量。第一个字符串**argv[0]**是程序名，如果不提供名称，则**argv[0][0]**应为**'\0'**。第i个程序参数为**argv**[i]，其中i=1,...,**argc** − 1。标准C语言要求**argv[argc]**为null指针，但一些早期的实现不是如此。向量**argv**及其所指的字符串应当可以修改，其数值不能在程序执行期间由实现或宿主系统修改。如果实现不支持混合大小写字符串，则**argv**中存储的字符串应使用小写字母。

独立C语言环境和某些软件框架（如Microsoft Windows MFC）对启动C语言程序可能有特殊规则。

例 下列短程序打印程序名和程序参数。 303

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int i;
    printf("Name: %s\n", argv[0]);
    printf("Arguments: ");
    for( i=1; i<argc; i++)
        printf("%s ", argv[i]);
    printf("\n");
    return 0;
}
```

□

有些实现允许**main**函数使用第3个参数**char * envp[]**，指向“环境值（environment value）”的null终止向量，每个元素是一个**"name=value"**形式的null终止字符串的指针。如果环境指针不是**main**的参数，则可以在全局变量中找到。一些UNIX实现用全局变量**environ**保存环境指针。但是，更便于移植的方法是使用标准C语言的**getenv**函数访问环境。

例 假设**envp**保存环境指针，则下列代码打印环境内容：

```
for(i=0; envp[i] != NULL; i++)
    printf("%s\n", envp[i]);
```

□

参考章节 **exit** 16.5; **getenv** 16.6; **system** 16.7

9.10 内联函数

内联函数是C99中增加的，在函数声明或定义中出现函数说明符**inline**。**inline**说明符只

是对翻译器的一个提示，建议尽快调用内联函数。这个名称来源于被称为内联扩展（inline expansion）的编译器优化，把函数调用换成函数体的拷贝，从而消除函数调用的开销。C99之前的许多C语言翻译器由扩展C语言提供内联函数，C++也提供内联函数。内联扩展有3个重要原则：

1. 有效定义（Visible definition）。要扩展内联函数调用，翻译器要在翻译调用时知道函数的定义。在C99中，如果函数声明为**inline**，则这个翻译单元中应能访问函数的定义。
2. 自由选择（Free choice）。翻译器不一定非要进行内联扩展。如果内联函数有4个调用，则
304 可以扩展其中两个调用，另外两个采用正常函数调用。C语言程序不能依赖于内联扩展。
3. 含义相同（Same meaning）。翻译器扩展一个或几个内联调用时，要保证程序行为和采用正常函数调用时相同。内联扩展只是优化，而不能改变程序的含义。

任何静态函数都可以指定为**inline**，因为所有调用与定义都局限在一个翻译单元中。

外部函数则不同，因为外部函数通常是在一个单元中调用，在另一个单元中定义。由于发生内联声明时应能访问函数的定义，因此外部函数的内联声明只能出现在定义这个函数的翻译单元中。我们希望其他翻译单元能够“偷看”外部函数的定义，使翻译器能够在这些单元中扩展外部函数调用。

这种“偷看”称为内联定义（inline definition）。如果翻译单元中函数的所有顶层声明包括**inline**而不包括**extern**，则这个单元中这个函数的定义称为内联定义（必须有这种内联定义，其应为**inline**而不是**extern**）。内联定义不提供函数的外部定义，还要在其他某个翻译单元中提供函数的外部定义。内联定义只是进行外部调用的一种替换方法，翻译器可以用这个替换方法进行内联扩展。如果翻译器不用替换方法，则只是产生正常的函数调用，把内联定义当作正常的**extern**声明。如果内联定义和函数的单个外部定义不等价，则程序行为是未定义的。要想使用内联定义，可以改变头文件，将函数声明换成内联定义。

例　函数**square**返回参数的平方。头文件**square.h**提供所包括的任何翻译单元的内联定义。如果翻译器不扩展调用或要取得函数地址，则把内联定义当作正常的**extern**声明。翻译单元**square.c**包括内联定义，但还提供**extern**声明，使**square.h**中的定义成为外部函数定义。

```
// File: square.h
// Inline definition:
inline double square(double x) { return x*x; }

// File square.c
#include "square.h"
// Force an external definition using the inline code
extern inline square(double x);
```

□

标准库头文件通常不能利用标准函数的**inline**定义，因为有些情况下程序可以重新声明这些函数（宏）。但是，实现可以随意使用各自的不可移植内联机制，或按其他某种特别方式处理标准库函数。

如果外部内联函数包括静态对象的定义，则可能出错。很难把内联定义中出现的静态对象
305 与另一单元中外部定义中出现的静态对象相连接。因此，C99禁止任何非静态内联函数定义可修改的静态对象，并禁止包含具有内部连接的标识符的引用。可以定义常量静态时象，但每个内联定义可能生成自己的对象。

参考章节 **inline**函数说明符 4.3.3

9.11 C++兼容性

9.11.1 原型

为了与C++兼容，所有函数都要用原型声明。事实上，非原型形式在C++中具有不同含义，空参数列表在C++中表示函数不带参数，而在C语言中表示函数带未知个数的参数。

例

```
int f(); /* Means int f(void) in C++, int f(...) in C */
int g(void); /* Means the same in both C and C++ */
…
x = f(2);    /* valid in C, not in C++ */
```

□

9.11.2 参数类型声明与返回类型声明

不要在参数列表或返回类型声明中放置类型声明，这在C++中是不允许的。

例

```
struct s { … } f1(int i); /* OK in C, not in C++ */
void f2(enum e{…} x); /* OK in C, not in C++ */
```

□

9.11.3 返回类型一致性

在C++和C99中，要从具有非**void**返回类型的函数中返回适当类型的值。C89允许不返回数值，这样就保证了其向下兼容性。

例

```
int f(void)
{
   …
   return ; /* Valid but unpredictable in C;
               invalid in C++ */
}
```

□

参考章节 返回类型一致性 9.8

306

9.11.4 main函数

在C++中，**main**函数不能递归调用，也不能取得其地址。C++对程序启动有更多限制，因此实现可能把**main**当作特例处理。如果要操纵**main**函数，只要生成第二个函数，从**main**中调用，在程序中用它代替**main**。C++中，如果控制到达**main**的结尾，就好像隐式地执行了“**return 0;**”。

9.11.5 内联

C99中函数内联定义的规则不像C++那么严格，C++要求所有内联定义和外部定义完全相同，而不只是等价。C99允许某些翻译单元中的内联定义特殊化，编程人员负责保证等价。C++还要求所有翻译单元中内联声明内联函数，而C99没有这个要求。为了保证移植性，应遵守更严格的C++规则。

9.12 练习

1. 下列哪些声明可以作为标准C语言原型？

(a) `short f(void);`　　(d) `int f(i,j);`

(b) `int f();`　　(e) `int *f(float);`

(c) `double f(...);`　　(f) `int f(i) int i; {...}`

2. 函数的声明和定义如下，哪一对与标准C语言兼容？

	声明	定义
(a)	`extern int f(short x);`	`int f(x) short x; {...}`
(b)	`extern int f();`	`int f(short x) {...}`
(c)	`extern f(short x);`	`int f(short int y) {...}`
(d)	`extern void f(int x);`	`void f(int x,...) {...}`
(e)	`extern f();`	`int f(x,y) short x,y; {...}`
(f)	`extern f();`	`f(void) {...}`

3. 函数的声明和定义如下，请指出它们在标准C语言中调用是否有效，如果有效，每个实际参数采用哪种转换。假设`s`的类型为`short`，`ld`的类型为`long double`。

	声明	定义
(a)	`extern int f(int *x);`	`f(&s)`
(b)	`extern int f();`	`f(s,ld)`
(c)	`extern f(short x);`	`f(ld)`
(d)	`extern void f(short,...);`	`f(s,s,ld)`
(e)	`int f(x) short x; {...}`	`f(s)`
(f)	`int f(x) short x; {...};`	`f(ld)`

307 4. 下列程序段中，`P`的调用是否由原型控制？为什么？

```
extern void P(void);
…
int Q()
{
    extern P();
    P();
    …
}
```

5. 如果函数声明的返回类型为`short`，则下列哪些表达式类型可以放在`return`语句中并能在调用时产生可预测的值？

(a) `int`

(b) `long double`

(c) `void`（例如调用返回`void`的函数）

(d) `char *`

6. 下列`square`的宏定义与9.10节的内联版本有什么不同？

308

```
#define square(x) ((x)*(x))
```

第二部分 C语言库

第10章 库 简 介

标准C语言包括语言标准和一组标准库。这些库支持字符和字符串、输入与输出、数学函数、日期与时间转换、动态存储分配和其他特性。每个库中的功能（类型、宏、函数）在标准头文件中定义，要使用库中的功能，就要增加一个预处理器命令**#include**，引用这个库的头文件。

例 下列程序段中头文件**math.h**使程序能够访问余弦函数**cos**。

```
#include <math.h>
double x, y;
...
x = cos(y);
```

□

传统C语言的一些实现不对所有库函数使用头文件，因此有些要由编程人员声明。

对定义为函数的库功能，标准C语言允许实现提供除真正函数以外的同名函数式宏。宏可能提供简单函数的更快实现方法或可能调用不同名称的函数。宏会负责求值每个参数表达式一次，像函数一样。如果不管宏是否存在而确定要需要访问函数，则要按下例所示绕过宏。 311

例 假设担心**math.h**中已有名为**cos**的宏，则可以用下面两种方法引用基础函数。两者都利用宏名后面不能紧跟一个开括号的特点，避免扩展同名函数或宏**cos**。

```
#include <math.h>
double a, b, (*p)(double);
...
p= &cos; a = (*p)(b); /* calls function cos, always */
a = (cos)(b);          /* calls function cos, always */
```

也可以取消所有涉及到的宏的定义：

```
#include <math.h>
#undef cos
...
a = cos(b);            /* calls function cos, always */
```

□

参考章节 **#include** 3.4；*带参数的宏* 3.3.2；**#undef** 3.3.5

10.1 标准C语言函数

10.3节汇总标准库函数，对于每一个库头文件都列出了其中定义的函数名及描述这些函数的章节。如果要查找特定库函数名而不知道它在哪个头文件中，则可以从书后的索引中寻找这个名称。

在本书的各个章节中，函数都是以标准C语言形式来描述的。除了另有说明，否则可以从标

准C语言定义得到传统C语言库函数定义，只要进行如下改写：

1. 消除任何使用标准C语言类型的函数，如**long long**与**_Complex**，或消除标准C语言中新增的函数（C89或C99）。
2. 删除限定符**const**、**restrict**与**volatile**。删除数组声明符括号内使用的**static**。
3. 将类型**void ***换成**char ***，将**size_t**换成**int**。

标准C语言中的库功能和头文件有许多特殊之处，主要是为了保护实现的完整性。

1. 库名原则上是保留字。编程人员不能定义与标准库名称同名的外部对象。
2. 库头文件或文件名可以内置在实现中，但仍然要被包括之后才能访问其名称。即**stdio.h**不一定实际对应于名为“**stdio.h**”的**#include**文件。

312

3. 编程人员可以多次按任意顺序包括库头文件（传统C语言实现中则不然）。

例 下面的方法可以保证库头文件不被包括多次：

```
/* Header stddef.h */
#ifndef _STDDEF /* Don't try to redeclare */
#define _STDDEF 1
typedef int ptrdiff_t;
.. /* other definitions */
#endif
```

□

保留库标识符

除了2.6节所列的关键字之外，标准C语言还保留标准库中声明的标识符和标准C语言实现内部使用的其他一些标识符。一个好记的规则是：不要将标准库中定义的标识符用于任何其他用途，不要使用以下划线开头的标识符。这样就可以避免在不同标准C语言实现之间移动时发生命名冲突。下面列出更详尽的规则。

标识符类型	编程人员使用
具有外部连接的库标识符（如函数名**errno**）	宿主实现中不能复用于外部连接
具有文件作用域的库标识符和库宏	如果包括定义这些名称或宏的库头文件，则不能作为文件作用域名称或宏复用
以下划线开头加上一个大写字母或加上另一下划线的标识符	不能用于任何用途，C语言实现常用其作为扩展
以下划线开头的其他标识符	不能作为文件作用域名或标志

不能编写标准库函数的自定义版本。如果把**sqrt**函数换成自定义的函数，则可能因为有两个同名函数而造成连接错误。这个限制使C语言实现可以更自由地组装并在内部使用标准库函数。

10.2 C++兼容性

C++语言包括标准C语言运行库，但增加了几个C++特定库。增加的库都不用以“.h”结尾的名称，因此通常不会与C语言库发生冲突。

C++用不同规则调用函数，即一般来说，不能从C语言程序中调用C++函数，但C++提供了

从C++中调用C语言函数的方法。声明C语言函数时有两个要求： 313

1. 函数声明要使用标准C语言原型，因为C++要求原型。

2. 外部C语言要显式地标为具有C语言连接，即在C++的存储类**extern**后面加上字符串**"C"**。

例 如果在一个C语言函数中调用另一C语言函数，则应声明

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

但是，如果从C++中调用C语言函数，则声明如下：

```
extern "C" int f(void);
```

如果C++中要声明一组C语言函数，则可以对所有C语言函数采用连接规范：

```
extern "C" {
    double sqrt(double x);
    int f(void);
    …
}
```

□

对可能从C语言或C++调用的库编写头文件时，要选择是在头文件中指定C语言连接还是要求C++程序在包括头的文件中提供连接声明。

例 假设要从C语言或C++调用头文件**library.h**。第一种方法是在头文件中包括**extern "C"**声明（条件预编译**__cplusplus**宏），表示这是个C++程序。

```
/* File library.h */
#ifdef __cplusplus
extern "C" {
#endif
…
/* C declarations */
…
#ifdef __cplusplus
}
#endif
```

另一种方法是用正常C语言声明编写头文件，只是要求C++用户用**#include**命令包装连接声明：

```
extern "C" {
#include "library.h"
}
```

□ 314

调用C++出现之前编写的库时，要使用上面第二种方法。可以嵌套**extern "C"{}**声明，这样做不会产生问题。

参考章节 **__cplusplus**宏 3.9.1 315

10.3 库头文件与名称

10.3.1 assert.h

参见第19章。

assert **NDEBUG**

10.3.2 complex.h

参见第23章。这个头文件是C99中增加的。

cabs	catan	clog	csinf
cabsf	catanf	clogf	csinh
cabsl	catanh	clogl	csinhf
cacos	catanhf	complex	csinhl
cacosf	catanhl	_Complex_I	csinl
cacosh	catanl	conj	csqrt
cacoshf	ccos	conjf	csqrtf
cacoshl	ccosf	conjl	csqrtl
cacosl	ccosh	cpow	ctan
carg	ccoshf	cpowf	ctanf
cargf	ccoshl	cpowl	ctanh
cargl	ccosl	cproj	ctanhf
casin	cexp	cprojf	ctanhl
casinf	cexpf	cprojl	ctanl
casinh	cexpl	creal	CX_LIMITED_RANGE
casinhf	cimag	crealf	I
casinhl	cimagf	creall	imaginary
casinl	cimagl	csin	_Imaginary_I

10.3.3 ctype.h

参见第12章。

isalnum	isgraph	isupper
isalpha	islower	isxdigit
isblank	isprint	tolower
iscntrl	ispunct	toupper
isdigit	isspace	

10.3.4 errno.h

参见第11章。

EDOM	ERANGE
EILSEQ	errno

316

10.3.5 fenv.h

参见第22章。这个头文件是C99中增加的。

FE_ALL_EXCEPT	FE_TONEAREST	fegetround	fesetround
FE_DFL_ENV	FE_TOWARDZERO	feholdexcept	fetestexcept
FE_DIVBYZERO	FE_UNDERFLOW	FENV_ACCESS	feupdateenv
FE_DOWNWARD	FE_UPWARD	fenv_t	fexcept_t
FE_INEXACT	feclearexcept	feraiseexcept	
FE_INVALID	fegetenv	fesetenv	
FE_OVERFLOW	fegetexceptflag	fesetexceptflag	

10.3.6 float.h

参见表5-3。

DBL_DIG	DBL_MIN_EXP	FLT_MAX_EXP	LDBL_MANT_DIG
DBL_EPSILON	DECIMAL_DIG	FLT_MIN	LDBL_MAX
DBL_MANT_DIG	FLT_DIG	FLT_MIN_10_EXP	LDBL_MAX_10_EXP
DBL_MAX	FLT_EPSILON	FLT_MIN_EXP	LDBL_MAX_EXP
DBL_MAX_10_EXP	FLT_EVAL_METHOD	FLT_RADIX	LDBL_MIN
DBL_MAX_EXP	FLT_MANT_DIG	FLT_ROUNDS	LDBL_MIN_10_EXP
DBL_MIN	FLT_MAX	LDBL_DIG	LDBL_MIN_EXP
DBL_MIN_10_EXP	FLT_MAX_10_EXP	LDBL_EPSILON	

10.3.7 inttypes.h

参见第21章。这个头文件是C99中增加的。

CNiLEASTN	PRIoMAX	PRIxPTR	SCNuFASTN
imaxabs	PRIoN	PRIXPTR	SCNuLEASTN
imaxdiv	PRIoPTR	SCNdFASTN	SCNuMAX
imaxdiv_t	PRIuFASTN	SCNdLEASTN	SCNuN
PRIdFASTN	PRIuLEASTN	SCNdMAX	SCNuPTR
PRIdLEASTN	PRIuMAX	SCNdN	SCNxFASTN
PRIdMAX	PRIuN	SCNdPTR	SCNxLEASTN
PRIdN	PRIuPTR	SCNiFASTN	SCNxMAX
PRIdPTR	PRIxFASTN	SCNiMAX	SCNxN
PRIiFASTN	PRIXFASTN	SCNiN	SCNxPTR
PRIiLEASTN	PRIxLEASTN	SCNiPTR	strtoimax
PRIiMAX	PRIXLEASTN	SCNoFASTN	strtoumax
PRIiN	PRIxMAX	SCNoLEASTN	wcstoimax
PRIiPTR	PRIXMAX	SCNoMAX	wcstoumax
PRIoFASTN	PRIxN	SCNoN	
PRIoLEASTN	PRIXN	SCNoPTR	

317

10.3.8 iso646.h

参见11.5节。这个头文件是C89增补1中增加的。

and	bitor	not_eq	xor
and_eq	compl	or	xor_eq
bitand	not	or_eq	

10.3.9 limits.h

参见表5-2。

CHAR_BIT	LLONG_MAX	SCHAR_MAX	UINT_MAX
CHAR_MAX	LLONG_MIN	SCHAR_MIN	ULLONG_MAX
CHAR_MIN	LONG_MAX	SHRT_MAX	ULONG_MAX
INT_MAX	LONG_MIN	SHRT_MIN	USHRT_MAX
INT_MIN	MB_LEN_MAX	UCHAR_MAX	

10.3.10 locale.h

参见第20章。

LC_ALL	LC_MONETARY	lconv	setlocale
LC_COLLATE	LC_NUMERIC	localeconv	
LC_CTYPE	LC_TIME	NULL	

318

10.3.11 math.h

参见第17章。

acos
acosf
acosh
acoshf
acoshl
acosl
asin
asinf
asinh
asinhf
asinhl
asinl
atan
atan2
atan2f
atan2l
atanf
atanh
atanhf
atanhl
atanl
cbrt
cbrtf
cbrtl
ceil
ceilf
ceill
copysign
copysignf
copysignl
cos
cosf
cosh
coshf
coshl
cosl
double_t
erf
erfc
erfcf
erfcl
erff
erfl
exp
exp2
exp2f
exp2l
expf
expl
expm1
expm1f
expm1l
fabs
fabsf
fabsl
fdim
fdimf
fdiml
float_t
floor
floorf
floorl
fma
fmaf
fmal
fmax
fmaxf
fmaxl
fmin
fminf
fminl
fmod
fmodf
fmodl
FP_CONTRACT
FP_FAST_FMA
FP_FAST_FMAF
FP_FAST_FMAL
FP_ILOGB0
FP_ILOGBNAN
FP_INFINITE
FP_NAN
FP_NORMAL
FP_SUBNORMAL
FP_ZERO
fpclassify
frexp
frexpf
frexpl
HUGE_VAL
HUGE_VALF
HUGE_VALL
hypot
hypotf
hypotl
ilogb
ilogbf
ilogbl
INFINITY
isfinite
isgreater
isgreaterequal
isinf
isless
islessequal
islessgreater
isnan
isnormal
isunorderedldex
p
ldexpf
ldexpl
lgamma
lgammaf
lgammal
llrint
llrintf
llrintl
llround
llroundf
llroundllog
log10
log10f
log10l
log1p
log1pf
log1pl
log2
log2f
log2l
logb
logbf
logbl

319

logf
logl
lrint
lrintf
lrintl
lround
lroundf
lroundl
MATH_ERREXCEPT
math_
errhandling
MATH_ERRNO
modf
modff
modfl
NAN
nan
nanf
nanl
nearbyint
nearbyintf
nearbyintl
nexttafter
nexttafterf
nexttafterl
nexttoward
nexttowardf
nexttowardl
pow
powf
powl
remainder
remainderf
remainderl
remquo
remquof
remquol
rint
rintf
rintl
round
roundf
roundl
scalbln
scalblnf
scalblnl
scalbn
scalbnf
scalbnl
signbit
sin
sinf
sinh
sinhf
sinhl
sinl
sqrt
sqrtf
sqrtl
tan
tanf
tanh
tanhf
tanhl
tanl
tgamma
tgammaf
tgammal
trunc
truncf
truncl

320

10.3.12 setjmp.h

参见19.4节。

```
jmp_buf          longjmp          setjmp
```

10.3.13 signal.h

参见19.6节。

```
raise            SIG_ERR          SIGFPE           signal
sig_atomic_t     SIG_IGN          SIGILL           SIGSEGV
SIG_DFL          SIGABRT          SIGINT           SIGTERM
```

10.3.14 stdarg.h

参见11.4节。

```
va_arg           va_end           va_start
va_copy          va_list
```

10.3.15 stdbool.h

参见11.3节。

```
bool                                 false
__bool_true_false_are_defined        true
```

10.3.16 stddef.h

参见11.1节。

```
NULL             ptrdiff_t        wchar_t
offsetof         size_t
```

321

10.3.17 stdint.h

参见第21章。这个头文件是C99中增加的。

```
INT_FASTN_MAX    INTN_C           SIG_ATOMIC_MIN   UINTN_MAX
INT_FASTN_MIN    INTN_MAX         SIZE_MAX         uintN_t
int_fastN_t      INTN_MIN         UINT_FASTN_MAX   UINTPTR_MAX
INT_LEASTN_MAX   intN_t           uint_fastN_t     uintptr_t
INT_LEASTN_MIN   INTPTR_MAX       UINT_LEASTN_MAX  WCHAR_MAX
int_leastN_t     INTPTR_MIN       uint_leastN_t    WCHAR_MIN
INTMAX_C         intptr_t         UINTMAX_C        WINT_MAX
INTMAX_MAX       PTRDIFF_MAX      UINTMAX_MAX      WINT_MIN
INTMAX_MIN       PTRDIFF_MIN      uintmax_t
intmax_t         SIG_ATOMIC_MAX   UINTN_C
```

10.3.18 stdio.h

参见第15章。

```
BUFSIZ           fputs            printf           stderr
clearerr         fread            putc             stdin
EOF              freopen          putchar          stdout
fclose           fscanf           puts             TMP_MAX
feof             fseek            remove           tmpfile
ferror           fsetpos          rename           tmpnam
fflush           ftell            rewind           ungetc
fgetc            fwrite           scanf            vfprintf
fgetpos          getc             SEEK_CUR         vfscanf
```

fgets	getchar	SEEK_END	vprintf
FILE	gets	SEEK_SET	vscanf
FILENAME_MAX	_IOFBF	setbuf	vsnprintf
fopen	_IOLBF	setvbuf	vsprintf
FOPEN_MAX	_IONBF	size_t	vsscanf
fpos_t	L_tmpnam	snprintf	
fprintf	NULL	sprintf	
322 fputc	perror	sscanf	

10.3.19 stdlib.h

参见第16章。

abort	_Exit	MB_CUR_MAX	strtof
abs	EXIT_FAILURE	mblen	strtol
atexit	EXIT_SUCCESS	mbstowcs	strtold
atof	free	mbtowc	strtoll
atoi	getenv	NULL	strtoul
atol	labs	qsort	strtoull
atoll	ldiv	rand	system
bsearch	ldiv_t	RAND_MAX	wchar_t
calloc	llabs	realloc	wcstombs
div	lldiv	size_t	wctomb
div_t	lldiv_t	srand	
exit	malloc	strtod	

10.3.20 string.h

参见第13章。

memchr	size_t	strcspn	strpbrk
memcmp	strcat	strerror	strrchr
memcpy	strchr	strlen	strspn
memmove	strcmp	strncat	strstr
memset	strcoll	strncmp	strtok
NULL	strcpy	strncpy	strxfrm

10.3.21 tgmath.h

参见17.12节。这个头文件是C99中增加的。

acos	cproj	hypot	nexttoward
acosh	creal	ilogb	pow
asin	erf	ldexp	remainder
asinh	erfc	lgamma	remquo
atan	exp	llrint	rint
atan2	exp2	llround	round
atanh	expm1	log	scalbln
carg	fabs	log10	scalbn
cbrt	fdim	log1p	sin
ceil	floor	log2	sinh
cimag	fma	logb	sqrt
conj	fmax	lrint	tan
copysign	fmin	lround	tanh
cos	fmod	nearbyint	tgamma
323 cosh	frexp	nextafter	trunc

10.3.22 time.h

参见第18章。

asctime	ctime	mktime	struct tm
clock	difftime	NULL	time
clock_t	gmtime	size_t	time_t
CLOCKS_PER_SEC	localtime	strftime	

10.3.23 wchar.h

参见第24章。这个头文件是C89增补1中增加的。

btowc	putwchar	wcschr	wcstok
fgetwc	size_t	wcscmp	wcstol
fgetws	swprintf	wcscoll	wcstold
fputwc	swscanf	wcscpy	wcstoll
fputws	tm	wcscspn	wcstoul
fwide	ungetwc	wcsftime	wcstoull
fwprintf	vfwprintf	wcslen	wcsxfrm
fwscanf	vfwscanf	wcsncat	wctob
getwc	vswprintf	wcsncmp	WEOF
getwchar	vswscanf	wcsncpy	wint_t
mbrlen	vwprintf	wcspbrk	wmemchr
mbrtowc	vwscanf	wcsrchr	wmemcmp
mbsinit	WCHAR_MAX	wcsrtombs	wmemcpy
mbsrtowcs	WCHAR_MIN	wcsspn	wmemmove
mbstate_t	wchar_t	wcsstr	wmemset
NULL	wcrtomb	wcstod	wprintf
putwc	wcscat	wcstof	wscanf

10.3.24 wctype.h

参见第24章。这个头文件是C89增补1中增加的。

iswalnum	iswgraph	iswxdigit	wctype
iswalpha	iswlower	towctrans	wctype_t
iswblank	iswprint	towlower	WEOF
iswcntrl	iswpunct	towupper	wint_t
iswctype	iswspace	wctrans	
iswdigit	iswupper	wctrans_t	

第11章 标准语言补充

可以把某些标准C语言库看成是这个语言的一部分，这些库提供标准定义和参数化，使C语言程序更容易移植。独立实现即使不提供其他库，也要提供这些库。这些核心库包括头文件**float.h**、**iso646.h**、**limits.h**、**stdarg.h**、**stdbool.h**、**stddef.h**与**stdint.h**。第5章介绍了**float.h**与**limits.h**中的函数，第21章将介绍**stdint.h**中的函数。本章将介绍其他库。

本章还要介绍**errno.h**中的函数，但这个库不属于标准语言补充。头文件**stdlib.h**虽然名字叫做“标准库”，但是也不属于标准语言补充，它将在第16章介绍。

11.1 NULL、ptrdiff_t、size_t、offsetof

语法概要

```
#include <stddef.h>
#define NULL ...
typedef ... ptrdiff_t;
typedef ... size_t;
typedef ... wchar_t;
#define offsetof( type, member-designator )...
```

这些是头文件**stddef.h**中定义的函数。

传统上，宏**NULL**的值是null指针常量。许多实现将其定义为整型常量0或转换为类型**void ***的0。在标准C语言中，为了方便起见，许多不同头文件都定义了这个宏。

325

ptrdiff_t类型是实现定义的带符号整型，是两个指针相减所得到的类型，大多数实现用**long**表示这个类型。**size_t**类型是**sizeof**运算符得到的无符号整型，大多数实现用**unsigned long**表示这个类型。标准化之前的实现有时用带符号类型**int**表示**size_t**。**ptrdiff_t**与**size_t**的最小值与最大值在C99的头文件**stdint.h**中定义。

随着处理器性能的增强，内存长度加大，使32位指针无法适应。C语言实现可以用C99类型**long long**表示**ptrdiff_t**，用**unsigned long long**表示**size_t**。这对早期的C语言代码可能会造成问题，因为其中假设**sizeof(size_t) = sizeof(ptrdiff_t) = sizeof(long)**。

宏**offsetof**扩展一个整型常量表达式（类型为**size_t**），这是结构类型*type*中成员*member-designator*的偏移量（字节数）。如果成员是位字段，则结果是不可预测的。如果没有定义**offsetof**（在非标准实现中），则可以定义如下：

```
#define offsetof(type, memb) ((size_t) & ((type *) 0)-> memb)
```

如果实现不允许按这种方式使用null指针常量，则可以用预定义非null指针和从结构的基址中减去成员的地址的方法计算偏移量。

例 下列程序段结束时，**diff**的值为1，**size**与**offset**的值相等。对于**sizeof(int)**为

4的字节寻址计算机，**size**与**offset**都等于4。

```
#include <stddef.h>
struct s {int a; int b; } x;
size_t size, offset;
ptrdiff_t diff;
...

diff = &x.b - &x.a;
size = sizeof(x.a);
offset = offsetof(struct s,b);
```

□

类型**wchar_t**也在**stddef.h**中定义，但我们将在第24章介绍**wchar.h**头文件时介绍这一类型。

参考章节 将整数转换成指针 6.2.7；null指针 5.3.2；指针类型 5.3；**sizeof**运算符 7.5.2；**stdint.h** 第21章；指针减法 7.6.2；**wchar_t** 24.1

326

11.2 EDOM、ERANGE、EILSEQ、errno、strerror、perror

语法概要

`#include <errno.h>`
`extern int errno;`
or `#define errno ...`
`#define EDOM ...`
`#define ERANGE ...`
`#define EILSEQ ...`

`#include <stdio.h>` `void perror(const char *s)`
`#include <string.h>` `char *strerror(int errnum)`

这些函数在**errno.h**和其他头文件中定义，支持标准库中的错误报告。

外部变量**errno**保存库程序中实现定义的错误码，通常被定义为**errno.h**中以**E**开头的宏。所有错误码都是正整数，库程序不能消除**errno**。在标准C语言中，**errno**不能是变量，但可以是扩展成**int**类型的任何可修改的lvalue的宏。

例 可以定义**errno**如下：

```
extern int *_errno_func();
#define errno (*_errno_func())
```

□

例 **errno**的常见用法是在调用库函数之前先清零，随后再进行检查：

```
errno = 0;
x = sqrt(y);
if (errno) {
   printf("?sqrt failed, code %d\n", errno);
```

```
    x = 0;
}
```

□

C语言实现通常定义一组标准错误码，可以放在**errno**中。**errno.h**中定义的标准错误码包括：

EDOM　参数不在数学函数能接受的域中。例如**log**函数的参数不能为负数参数。

ERANGE　数学函数的结果超出范围；函数具有定义良好的数学结果，但无法表示，因为受到浮点数格式限制。例如用**pow**函数求一个大数的大指数。

327

EILSEQ　翻译多字节字符序列时遇到的编码错误。这个错误最终会由**mbrtowc**或**wcrtomb**发现，它们又被其他宽字符函数调用（C89增补1）。

函数**strerror**返回一个错误消息字符串的指针，其内容是由实现定义的，字符串不能修改，但可以在后续调用**strerror**函数时覆盖。

函数**perror**在标准错误输出流中打印下面的序列：参数字符串**s**、冒号、空格、包含**errno**中当前错误码的错误短消息和新行符。在标准C语言中，如果**s**是null指针或null字符的指针，则只打印错误短消息，而不打印前面的参数字符串**s**、冒号以及空格。

例　前面章节中介绍的**sqrt**例子可以使用**perror**改写如下：

```
#include <math.h>
#include <errno.h>
...
errno = 0;
x = sqrt(y);
if (errno) {
    perror("sqrt failed");
    x = 0;
}
```

如果调用**sqrt**失败，则输出如下：

```
sqrt failed: domain error
```

□

标准C语言没有规定，但有些系统中可以把对应于**errno**值的错误短消息存放在字符串指针向量中，通常称为**sys_errlist**，可以用**errno**中的值作为索引。变量**sys_nerr**包含了**sys_errlist**索引可以使用的最大整数，应检查这个最大整数，以保证**errno**中不包含非标准错误号。

328

参考章节　编码错误　2.1.5；**mbrtowc**　11.7；**wcrtomb**　11.7

11.3 bool、false、true

语法概要

```
#include <stdbool.h>

#define bool _Bool
#define false 0
#define true 1
#define __bool_true_false_are_defined 1
```

stdbool.h头文件是C99中增加的，只包含上述声明。这些布尔类型和数值名与C++中的一致。

尽管标准头文件中通常不允许定义**#undefine**宏，但C99允许编程人员定义**#undefine**，必要时还可以重新定义**bool**、**false**与**true**宏。

参考章节 _Bool类型 5.1.5

11.4 va_list、va_start、va_arg、va_end

语法概要

```
#include <stdarg.h>
typedef ... va_list;
#define va_start( a_list ap, type LastFixedParm) ...
#define va_arg( va_list ap, type) ...
void va_end(va_list ap);
void va_copy(va_list dest, va_list src);
```

stdarg.h头文件中包含的函数为编程人员提供了访问可变参数表的可移植方式，这是**fprintf**（隐式）和**vfprintf**（显式）等函数需要的。

C语言最初没有限制参数传入函数的方式，编程人员因此对计算机系统上的行为作了不可移植的假设。最后，传统C语言中出现**varargs.h**功能，提高了移植性，标准C语言在**stdarg.h**中采用了类似的定义。**stdarg.h**的用法不同于**varargs.h**，因为标准C语言允许在可变参数表之前放上固定数目的参数，而早期实现则要求把整个参数表看成变长的。

下面列出**stdarg.h**中定义的宏、函数和类型的含义。这个功能是按传统的风格处理的，没有对实现作太多假设：

va_list 这种类型声明局部状态变量，这里统一称为**ap**，用于遍历参数。

va_start 这个宏初始化状态变量**ap**，要先调用之后才能调用**va_arg**与**va_end**。在 329
传统C语言中，**va_start**将**ap**中的内部指针设置成指向传入函数的第一个参数，而在标准C语言中，**va_start**还带另一参数（最后一个固定参数名），将**ap**中的内部指针设置成指向传入函数的第一个可变参数。

va_arg 这个宏返回参数表中下一个参数的值，将内部参数指针（在**ap**中）移到下一个参数（如有）。下一个参数的类型（经过普通参数转换之后）要用*type*指定，使**va_arg**能够计算其在堆栈中的长度。调用**va_start**之后第一次调用**va_arg**返回第一个可变参数的值。

va_end 这个函数或宏在用**va_arg**读取所有参数之后调用。对**ap**和**va_alist**进行必要的整理操作。

va_copy （C99）这个宏在**dest**中复制**src**的当前状态，在参数表中生成第二个指针。然后可以独立对**src**与**dest**采用**va_arg**。**dest**中要像**src**中一样调用**va_end**

va_arg的宏调用使用的类型名*type*应写成后缀*能产生“*type*的指针”类型。

C99中新增的**va_copy(saved_ap,ap)**宏可以在用**va_arg(ap,*type*)**将指针移到列表中下一位时在参数表中保留一个指针。如果需要，可以用**va_arg(saved_ap,*type*)**返回前面的位置。

例　我们介绍如何在标准C语言中编写可变参数函数。下节介绍传统C语言中的实现。函数**printargs**带有不同类型的可变个数参数，在标准输出中打印其数值。**printargs**的第一个参数是整数数组，表示后面参数的个数和类型。这个数组用0元素终止。下面的例子显示**printargs**的用法。这个例子在标准C语言和传统C语言中都可用：

```
#include "printargs.h"
int arg_types[] = { INTARG, DBLARG, INTARG, DBLARG, 0 };
int main()
{
    printargs( &arg_types[0], 1, 2.0, 3, 4.0 );
    return 0;
}
```

330 **printargs**的声明和整数类型指定符的值放在文件**printargs.h**中。

```
/* file printargs.h; Standard C */
#include <stdarg.h>
#define INTARG      1      /* codes used in argtypep[] */
#define DBLARG 2
…
void printargs(int *argtypep, ...);
```

标准C语言中**printargs**的相应定义如下：

```
#include <stdio.h>
#include "printargs.h"
void printargs( int *argtypep, ...)     /* Standard C */
{   va_list ap; int argtype;
    va_start(ap, argtypep);

    while ( (argtype = *argtypep++) != 0 ) {
        switch (argtype) {
        case INTARG:
            printf("int: %d\n", va_arg(ap, int) );
            break;

        case DBLARG:
            printf("double: %f\n", va_arg(ap, double) );
            break;
        /* … */
        }
    } /*while*/
    va_end(ap);
}
```

□

传统函数：varargs.h

传统C语言语法概要

```
#include <varargs.h>
#define va_alist ...
#define va_dcl ...
typedef ... va_list;
void va_start( va_list ap );
type va_arg( va_list ap, type);
void va_end(va_list ap);
```

在传统C语言中，可变参数用头文件**varargs.h**实现，其中包含两个新宏，并改变了**va_start**定义：

va_alist 这个宏代替具有可变个数参数的函数定义中的参数表。

va_dcl 这个宏代替函数定义中的参数声明，后面不能加分号，以便允许空值。 331

va_start 这个宏初始化状态变量**ap**，应在调用**va_arg**与**va_end**之前调用。在传统C语言中，**va_start**将**ap**中的内部指针设置成指向传入函数的第一个参数；这个宏在传统C语言中比标准C语言版本少一个参数。

例 下面是传统C语言中**printargs**的声明：

```
/* file printargs.h; Traditional C */
#define INTARG      1      /* codes used in argtypep[] */
#define DBLARG 2
…
#include <varargs.h>
printargs( va_alist );
```

下面是传统C语言实现的**printargs**，与标准C语言中实现的差别在于函数参数表不同和调用**va_start**的方法不同。

```
#include <stdio.h>
#include "printargs.h"
printargs( va_alist          )/* Traditional C */
    va_dcl
{   va_list ap; int argtype, *argtypep;
    va_start(ap);

    argtypep = va_arg(ap, int *);
    while ( (argtype = *argtypep++) != 0 ) {
        switch (argtype) {
        case INTARG:
            printf("int: %d\n", va_arg(ap, int) );
            break;
        case DBLARG:
            printf("double: %f\n", va_arg(ap, double) );
            break;
        /* … */
```

```
        }
      }
      va_end(ap);
332 }
```

□

11.5 标准C语言运算符宏

语法概要

```
#include <iso646.h>
#define and       &&
#define and_eq    &=
#define bitand    &
#define bitor     |
#define compl     ~
#define not       !
#define not_eq    !=
#define or        ||
#define or_eq     |=
#define xor       ^
#define xor_eq    ^=
```

C89增补1增加了头文件**iso646.h**，其中包含的宏可以代替某些运算符记号。这些记号在受限源字符集（如ISO 646）中编写时可能不太方便。在C++中，这些标识符是关键字。

例　下面3个**if**语句具有相同效果：

```
#include <iso646.h>
...
if (p || *p==0)      *p ^= q;      /*customary */
if (p ??!??! *p==0) *p ??'= q;    /*ISO C trigraphs */
if (p or *p==0)     *p xor_eq q;  /*ISO C iso646.h */
```

□

C89增补1还提供了用外语字母表中更常用的记号重新拼写{以及}之类标点符号的方法。

333 **参考章节**　关键字　2.6；记号重拼　2.4.1；三字符组　2.1.4

第12章 字符处理函数

字符处理有两类函数：分类与转换。每个字符分类函数的名称以**is**开头，返回**int**类型的值，在参数为指定类时为非0值（真），否则为0（假）。每个字符转换函数的名称以to开头，返回**int**类型的值，表示一个字符或**EOF**。标准C语言保留以**is**和**to**开头的名称，以便今后在库中增加更多分类函数与转换函数。本章介绍的字符处理函数在库头文件**ctype.h**中声明。

C89增补1定义了对宽字符进行运算的分类与转换函数。这些函数的名称以**isw**与**tow**开头，其余部分与相应的字符处理函数对应。宽字符分类函数接受**wint_t**类型参数，返回**int**类型的值。转换函数在**wint_t**类型值之间映射。还有通用的分类函数**wctrans**与**iswctrans**和通用的转换函数**wctrans**与**towctrans**，而扩展字符集可能使用特殊分类方法。这些函数都在头文件**wctype.h**中定义。

负整数**EOF**不是实际字符的编码（宽字符用**WEOF**）。例如，**fgetc**（见15.6节）在到达文件末尾时返回**EOF**，因为这时没有要读取的实际字符。但有些实现中类型**char**可能是带符号的，因此出现非标准字符值时，**EOF**可能无法与实际字符区别开来（标准字符值总是非负值，即使类型**char**是带符号的）。这里介绍的所有函数对表示为**char**和**unsigned char**的所有值以及**EOF**值都能进行正确运算，但对其他整数值的运算未定义，除非特别说明。**wchar_t**中**WEOF**的作用与**char**类型中**EOF**的作用相同，但**WEOF**不能为负数。

标准C语言建立这些功能时考虑了需要支持多种区域设置的可能性，一般来说，它尽量不对字符编码和“字母”之类的概念作任何假设。这些函数的传统C语言版本与标准C语言的“C”区域设置大致相同，但删除了其中与ASCII相关的部分（如**isascii**与**toascii**）。

警告 有些非标准C语言实现允许**char**类型为带符号类型，还支持**unsigned char**类型，但字符处理函数无法正确处理所有用**unsigned char**类型表示的值。有时，这些函数甚至无法正确处理所有用**char**类型表示的值，而只能处理“标准”字符值和**EOF**。

参考章节 **EOF** 11.1；**WEOF** 11.1；宽字符 2.1.4；**wchar_t** 11.1；**wint_t** 11.1

12.1 isalnum、isalpha、iscntrl、iswalnum、iswalpha、iswcntrl

语法概要

```
#include <ctype.h>

int isalnum(int c);
int isalpha(int c);
int iscntrl(int c);
int isascii(int c); /* Common extension */
```

```
#include <wctype.h>

int iswalnum(wint_t c);
int iswalpha(wint_t c);
int iswcntrl(wint_t c);
```

isalnum函数测试**c**是否是字母数字字符，即是否是C语言区域设置中的下列字符之一：

```
0 1 2 3 4 5 6 7 8 9
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

这个函数的定义等价于：

```
isalpha(c) || isdigit(c)
```

isalpha函数测试**c**是否是字母字符，即是否是C语言区域设置中的下列字符之一：

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

336 在任何区域设置中，**islower(c)**或**isupper(c)**为真值时，这个函数为真；**iscntrl(c)**、**isdigit(c)**、**ispunct(c)**或**isspace(c)**为真值时，这个函数为假，其余是由不同实现定义的。

函数**iscntrl**测试**c**是否是“控制符”。如果使用标准128位ASCII字符集，则控制符的值为0~31（37_8或$1F_{16}$）和127（177_8或$7F_{16}$）。**isprint**函数（见12.4节）至少对标准ASCII实现是个补充函数。

函数**isascii**不属于标准C语言，但是属于C语言库的公用扩展，它测试**c**的值是否在0～127（177_8或$7F_{16}$）之间，这是标准128字符的ASCII字符集的范围。与传统C语言中大多数字符分类函数不同的是，**isascii**能对任何类型的**int**值进行正确操作（其参数的类型即使在传统C语言中也是**int**）。

在传统C语言中，这些函数带有**char**类型的参数，但返回**int**值。

例　下列函数**is_id**在参数字符串**s**为有效C语言标识符时返回**TRUE**，否则返回**FALSE**。要使这个函数正确工作，当前区域设置应为“C”：

```
#include <ctype.h>
#define TRUE 1
#define FALSE 0

int is_id(const char *s)
{
    char ch;
    if ((ch = *s++) == '\0') return FALSE; /*empty string*/
    if (!(isalpha(ch) || ch == '_')) return FALSE;
    while ((ch = *s++) != '\0') {
        if (!(isalnum(ch) || ch == '_')) return FALSE;
    }
    return TRUE;
}
```

□

宽字符函数

C89增补1定义的**wctype.h**头文件提供3个宽字符函数。

iswalnum函数等价于**iswalpha(c) || iswdigit(c)**。

iswalpha函数测试**c**是否是特定区域设置的宽字符字母。在任何区域设置中，这个函数在**iswlower(c)**或**iswupper(c)**为真时取值为真；在**iswcntrl(c)**、**iswdigit(c)**、

iswpunct(c)或**iswspace(c)**为真时取值为假，其余是由不同实现定义的。

如果**c**是特定区域设置的宽控制字符集成员，则函数**iswcntrl**返回非0值。宽控制字符不能是**iswprint**指定的宽打印字符（见12.4节）。

337

12.2 iscsym、iscsymf

非标准语法概要

```
#include <ctype.h>
int iscsym(char c);
int iscsymf(char c);
```

这些函数不在标准C语言中。**iscsym**函数测试**c**是否可以作为C语言的标识符字符，**iscsymf**测试**c**是否可以作为标识符的第一个字符。

iscsymf函数至少对52个大小写字母和下划线字符为真，**iscsym**还对至少10个十进制数字为真。这些函数根据不同实现中的定义，还可能对其他字符为真。

12.3 isdigit、isodigit、isxdigit、iswdigit、iswxdigit

语法概要

```
#include <ctype.h>
int isdigit(int c);
int isxdigit(int c)
```

```
#include <wctype.h>
int iswdigit(wint_t c);
int iswxdigit(wint_t c);
```

isdigit函数测试**c**是否是10个十进制数字之一。**isxdigit**函数测试**c**是否是22个十六进制数字之一，即：

```
0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f
```

在标准化之前C语言中，这些函数的参数类型为**char**，但返回**int**。还可能遇到非标准**isodigit**函数，测试**c**是否是8个八进制数字之一。

宽字符函数

iswdigit函数（C89增补1）测试**c**是否对应于一个十进制数字字符，**iswxdigit**函数测试**c**是否对应于一个十六进制数字字符。

338

12.4 isgraph、isprint、ispunct、iswgraph、iswprint、iswpunct

语法概要

```
#include <ctype.h>
int isgraph(int c);
```

```
int ispunct(int c);
int isprint(char c);
```

```
#include <wctype.h>
int iswgraph(wint_t c);
int iswpunct(wint_t c);
int iswprint(wint_t c);
```

isprint函数测试**c**是否是打印字符（即任何非控制符字符）。空格是打印字符。**isgraph**函数测试**c**是否是图形字符的代码，即除空格以外的任何打印字符。**isprint**与**isgraph**函数的差别仅在于空格符的处理，**isprint**在大多数实现中与**iscntrl**相反，但并不是每个标准C语言区域设置中都需要这样。在传统C语言中，这些函数取**char**类型的参数，但返回**int**值。

例　如果使用标准128字符的ASCII字符集，则打印字符的代码为**040**到**0176**，即空格加上下列字符：

```
! " # $ % & ' ( ) * + , - . /
0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
[ \ ] ^ _
` a b c d e f g h i j k l m n o p q r s t u v w x y z
{ | } ~
```

图形字符也一样，只是不包括空格符。 □

函数**ispunct**测试**c**是否是标点符号的代码，作为打印字符的标点符号既不是空格，也不是**isalnum**为真的字符。

例　如果使用标准128字符的ASCII字符集，标点符号为下列字符：

```
! " # $ % & ' ( ) * + , - . / : ; < = >
? @ [ \ ] ^ _ ` { | } ~
```

339 □

宽字符函数

iswprint函数（C89增补1）测试**c**是否是宽打印字符（即特定区域设置的宽字符，至少在显示设备上占一个位置，但不是宽控制字符）。

iswgraph函数等价于**iswprint(c) && !iswspace(c)**。**iswpunct**函数测试**c**是否是符合下列条件的特定区域设置宽字符：

```
iswprint(c) && !(iswalnum(c) || iswspace(c))
```

12.5 islower、isupper、iswlower、iswupper

语法概要

```
#include <ctype.h>
int islower(int c);
int isupper(int c);
```

```
#include <wctype.h>
int iswlower(wint_t c);
int iswupper(wint_t c);
```

在C区域设置中，**islower**函数测试**c**是否是26个小写字母之一，**isupper**函数测试**c**是否是26个大写字母之一。在其他区域设置中，这些函数可能对符合下列条件的其他字符返回真值：

```
!iscntrl(c) && !isdigit(c) && !ispunct(c) && !isspace(c)
```

在传统C语言中，这些函数带有**char**类型的参数，但返回**int**值。

宽字符函数

iswlower函数（C89增补1）测试**c**是否是符合下列条件的特定区域设置的宽字符或小写字母：

```
!iswcntrl(c) && !iswdigit(c) && !iswpunct(c) && !iswspace
```

iswupper函数（C89增补1）测试**c**是否是符合上述条件的特定区域设置的宽字符或大写字母。

340

12.6 isblank、isspace、iswhite、iswspace

语法概要

```
#include <ctype.h>
int isblank(int c);
int isspace(int c);
```

```
#include <wctype.h>
int iswspace(wint_t c);
```

isspace函数测试**c**是否是空白符的代码。在C区域设置中，**isspace**返回真值的字符包括制表符（'**\t**'）、回车符（'**\r**'）、换行符（'**\n**'）、垂直制表符（'**\v**'）、换页符（'**\f**'）和空格符（' '）。许多其他库函数用**isspace**作为空白符定义。

isblank函数测试**c**是否是文本行中分隔单词的字符的编码，包括标准空白符、空格（' '）和水平制表符（'**\t**'），并可能包括使**isspace**为真值的其他特定区域设置的字符。"C"区域设置中没有其他空白符。

C语言的有些实现提供**isspace**的变体**iswhite**。在传统C语言中，本节介绍的这些函数带**char**类型的参数，但返回**int**值。

宽字符函数

iswspace函数（C89增补1）测试**c**是否是满足下列条件的特定区域设置的宽字符：

```
!iswalnum(c) && !iswgraph(c) && !ispunct(c)
```

12.7 toascii

非标准语法概要

```
#include <ctype.h>
int toascii(int c);
```

非标准**toascii**函数接受任何整数值，将其缩小到有效ASCII字符范围（编码为0~127[177_8或$3F_{16}$]），放弃数值中除低7位以外的所有位。如果参数已经是有效ASCII字符代码，则结果等于

341 参数。

12.8 toint

非标准语法概要

```
#include <ctype.h>
int toint(char c);
```

非标准**toint**函数返回十六进制数字的权值：0到9分别表示字符**'0'**到**'9'**，而10到15分别表示字符**'a'**到**'f'**（或**'A'**到**'F'**）。如果参数不是十六进制数字，则函数行为是由实现定义的。

例　标准C语言中没有这个函数，但很容易实现。这个实现的例子假设目标编码系统中某些字符是连续的：

```
int toint( int c )
{
    if (c >= '0' && c <= '9') return c - '0';
    if (c >= 'A' && c <= 'F') return c - 'A' + 10;
    if (c >= 'a' && c <= 'f') return c - 'a' + 10;
    /* c is not a hexadecimal digit */
    return 0;
}
```

□

12.9 tolower、toupper、towlower、towupper

语法概要

```
#include <ctype.h>
int tolower(int c);
int toupper(int c);
```

```
#include <wctype.h>
wint_t towlower(wint_t c);
wint_t towupper(wint_t c);
```

如果**c**是大写字母，则**tolower**返回相应的小写字母。如果**c**是小写字母，则**toupper**返回相应的大写字母。对所有其他情况，返回参数原值。在有些区域设置中，大写字母可能没有相应的小写字母，或小写字母可能没有相应的大写字母，这时返回参数原值。

函数**towlower**与**towupper**在C89增补1中定义。如果**c**是宽字符，**iswupper(c)**为真，**d**是对应于**c**的宽字符，**iswlower(d)**为真，则**towlower(c)**返回**d**，**towupper(d)**返回**c**，
342 否则这两个函数返回参数原值。

在非标准实现中使用时，要注意参数不是大写字母时**tolower**返回的值以及参数不是小写字母时**toupper**返回的值。许多早期的实现只在参数为相应的大小写字母时才能正确工作。允许**tolower**与**toupper**具有更一般参数的实现可能对其提供更快的版本——**_tolower**宏与**_toupper**宏。这些宏要求更严格的参数，速度更快。这些函数的非标准C语言中语法如下：

```
#include <ctype.h>
int tolower(char c);
int toupper(char c);
#define _tolower(c) …
#define _toupper(c) …
```

例 如果C语言库中的**tolower**版本不能很好地处理任意参数，则下列函数**safe_tolower**与**tolower**相似，但能很好地处理任意参数。**safe_tolower**很难写成宏，因为参数要通过**isupper**、**tolower**和**return**语句求值多次：

```
#include <ctype.h>
int safe_tolower(int c)
{
    if (isupper(c)) return tolower(c);
    else return c;
}
```

□

12.10 wctype_t、wctype、iswctype

语法概要

```
#include <wctype.h>
typedef … wctype_t;
wctype_t wctype(const char *property);
int iswctype(wint_t c, wctype_t desc);
```

wctype或**iswctype**函数在C89增补1中定义，实现了可扩展的、特定区域设置的宽字符分类功能。

wctype_t类型应为标量，保存的值表示特定区域设置的宽字符分类。**wctype**函数构造一个**wctype_t**类型的值，表示宽字符类。类用字符串名**property**指定，针对当前区域设置的**LC_CTYPE**类别值。表12-1列出了所有区域设置允许的**property**字符串名及其含义。

iswctype函数测试**c**是否是**desc**值所表示的类成员。调用**iswctype**时**LC_CTYPE**类别的设置应与**wctype**函数确定**desc**值时的**LC_CTYPE**设置相同。 343

表12-1 **wctype**的property名称

property名称	指定的类别
"alnum"	iswalnum(c)为真
"alpha"	iswalpha(c)为真
"cntrl"	iswcntrl(c)为真
"digit"	iswdigit(c)为真
"graph"	iswgraph(c)为真
"lower"	iswlower(c)为真
"print"	iswprint(c)为真
"punct"	iswpunct(c)为真
"space"	iswspace(c)为真
"upper"	iswupper(c)为真
"xdigit"	iswxdigit(c)为真

例　表达式**iswctype(c,wctype("alnum"))**与**iswalnum(c)**对于任何宽字符**c**和任何区域设置具有相同的真值。其他属性设置及相应的分类函数也是如此。 □

参考章节　**LC_CTYPE**　11.5；区域设置　11.5

12.11 wctrans_t、wctrans

语法概要

```
#include <wctype.h>
typedef ... wctrans_t;
wctrans_t wctrans( const char *property );
wint_t towctrans( wint_t c, wctrans_t desc );
```

本节的函数是C89增补1定义的，实现可扩展的、特定区域设置的宽字符映射函数。

wctrans_t类型应为标量，保存的值表示特定区域设置的宽字符映射。**wctrans**函数构造一个**wctrans_t**类型的值，表示宽字符间的映射。映射用字符串名**property**指定，针对当前
344 区域设置的**LC_CTYPE**类别值。下面列出了所有区域设置允许的**property**字符串名及其含义（注意，属性名不同于函数名）。

property值	指定与下列函数相同映射
"tolower"	towlower(c)
"toupper"	towupper(c)

towctrans函数将**c**映射到**desc**指定的另一个宽字符。调用**towctrans**时，**LC_CTYPE**类别的设置应与**wctrans**确定**desc**值时的**LC_CTYPE**设置相同。

例　表达式**towctrans(c,wctrans("tolower"))**与**towlower(c)**对任何宽字符**c**和任何区域设置具有相同的真值。其他属性设置及相应的分类函数也是如此。 □

345 **参考章节**　**LC_CTYPE**　11.5；区域设置　11.5

第13章 字符串处理函数

习惯上，C语言中的字符串是以null字符（'\0'）结尾的字符数组。编译器在所有字符串常量后面自动添加一个多余的null字符，但编程人员要保证字符数组中生成的字符串以null字符结尾。本章介绍的所有字符串处理函数都假设字符串以null字符结尾。

除了终止null字符之外，字符串中所有字符统称为字符串的内容。空字符串不包含字符，表示为null字符的指针，注意，这与null字符指针（**NULL**）不同，这个指针根本不指向任何字符。

字符转换成目标字符串时，通常不测试目标是否溢出。编程人员要保证内存区的目标区域足够大，能够放下结果字符串，包括终止null字符。

本章介绍的大多数函数在库头文件**string.h**中声明，一些标准C语言转换函数是**stdlib.h**提供的。在标准C语言中，不发生修改的字符串参数通常声明为类型**const char ***而不是**char ***，表示字符串长度的整数参数或返回值类型为**size_t**而不是**int**。

C89增补1增加了与普通字符串函数并行的宽字符串函数。差别在于宽字符串函数所带的参数类型为**wchar_t ***而不是**char ***，宽字符串函数名把普通字符串函数开头的字母**str**改成**wcs**。宽字符串以宽null字符终止。比较宽字符串时，比较**wchar_t**元素的整数值。宽字符串不进行解释，不会发生编码错误。

其他字符串函数在内存函数（第14章）、**sprintf**（15.11节）和**sscanf**（15.8节）中提供。

参考章节 **wchar_t** 11.1；宽字符 2.1.4

13.1 strcat、strncat、wcscat、wcsncat

语法概要

```
#include <string.h>
char *strcat( char *dest, const char *src );
char *strncat( char *dest, const char *src, size_t n );
#include <wchr.h>
wchar_t *wcscat( wchar_t *dest, const wchar_t *src );
wchar_t *wcsncat( wchar_t *dest, const wchar_t *src, size_t n );
```

函数**strcat**将字符串**src**的内容添加到字符串**dest**末尾，返回**dest**值。终止**dest**的null字符（和内存中其后面的其他字符）被**src**中的字符和新的终止null字符覆盖。从**src**中复制字符，直到遇到**src**中的null字符。假设从**dest**开始的内存区足够大，能够放下这两个字符串。

wcscat与**strcat**相似，只是参数类型和结果类型与**strcat**的不同。

例 下列语句将3个字符串添加到**D**中，得到的**D**包含字符串**"All for one."**：

```
#include <string.h>
char D[20];
```

```
...
D[0] = '\0';        /* Set string to empty */
strcat(D,"All ");
strcat(D,"for ");
strcat(D,"one.");
```
□

strncat函数从**src**内容中将最多**n**个字符添加到**dest**末尾。如果复制**n**个字符之前遇到**src**中的null字符，则复制到这个null字符为止。如果在**src**的前**n**个字符中没有遇到null字符，则把前**n**个字符复制到**dest**末尾，并提供一个终止目标字符串的null字符，即总共写入**n**+1个字符。如果**n**为0或负数，则调用**strncat**函数无效。这个函数总是返回**dest**。在传统C语言中，**strncat**函数最后一个参数的类型为**int**。

wcsncat与**strncat**相似，只是参数类型和结果类型与**strncat**的不同。

348 如果内存中有字符串共用存储空间，则所有这些函数的行为都是未定义的。

13.2 strcmp、strncmp、wcscmp、wcsncmp

语法概要

```
#include <string.h>
int strcmp( const char *s1, const char *s2 );
int strncmp( const char *s1, const char *s2, size_t n );
```

```
#include <wchr.h>
int wcscmp( const wchar_t *s1, const wchar_t *s2 );
int wcsncmp( const wchar_t *s1, const wchar_t *s2, size_t n );
```

函数**strcmp**在词法上比较以null终止的字符串**s1**与以null终止的字符串**s2**的内容，返回一个**int**类型值，在**s1**小于**s2**时小于0，在**s1**等于**s2**时为0，在**s1**大于**s2**时大于0。

例　要检查两个字符串是否相等，可以求**strcmp**返回值的相反数：

```
if (!strcmp(s1,s2)) printf("Strings are equal\n");
else printf("Strings are not equal\n");
```
□

两个字符串内容一致时为相等。下列条件下，字符串**s1**在词法上小于**s2**：

1. 字符串在一定字符位置之前相同，而在第一个不同字符位置上，**s1**中的字符值小于**s2**中的字符值。
2. 字符串**s1**比字符串**s2**短，**s1**内容与**s2**中到**s1**长度为止的内容相同。

wcscmp（C89增补1）与**strcmp**相似，只是参数类型不同。

函数**strncmp**与**strcmp**相似，只是至多比较以null终止的字符串**s1**的前**n**个字符与以null终止的字符串**s2**的前**n**个字符。比较字符串时，如果字符串长度不到**n**个字符，则使用整个字符串，否则只比较前**n**个字符。如果**n**的值为0或负数，则把这两个字符串当成空字符串，因此是相等，返回0。在传统C语言中，参数**n**的类型为**int**。

wcsncmp与**strncmp**相似，只是参数类型与**strncmp**的不同。

函数**memcmp**（见14.2节）提供与**strcmp**相似的功能。函数**strcoll**（见13.10节）提供特

349 定区域设置的比较功能。

13.3 strcpy、strncpy、wcscpy、wcsncpy

语法概要

```
#include <string.h>
char *strcpy( char *dest, const char *src );
char *strncpy( char *dest, const char *src, size_t n );
```

```
#include <wchar.h>
wchar_t *wcscpy( wchar_t *dest, const wchar_t *src );
wchar_t *wcsncpy( wchar_t *dest, const wchar_t *src, size_t n );
```

函数**strcpy**将字符串**src**的内容复制到字符串**dest**，覆盖**dest**中原先的内容。复制整个**src**的内容并添加终止null字符，即使**src**比**dest**长。函数**strcpy**返回**dest**。

wcscpy（C89增补1）与**strcpy**相似，只是参数类型与**strcpy**的不同。

例 **strcat**函数（见13.1节）可以用**strcpy**与**strlen**（见13.4节）实现如下：

```
#include <string.h>
char *strcat(char *dest, const char *src)
{
    char *s = dest + strlen(dest);
    strcpy(s, src);
    return dest;
}
```

□

函数**strncpy**只把**n**个字符复制到**dest**。首先它从**src**复制**n**个字符。如果**src**在null字符之前不足**n**个字符，则填充null字符，直到写入**n**个字符。如果**src**中超过**n**个字符，则只把前**n**个字符复制到**dest**，因此只向**dest**传输**src**的截尾复本。由此可见，只有**src**长度（不计终止null字符）小于**n**时，**strncpy**才用null字符终止**dest**中的拷贝。如果**n**为0或负数，则调用**strncpy**函数无效。函数**strncpy**总是返回**dest**的值。在传统C语言中，参数**n**的类型为**int**。

wcsncpy（C89增补1）与**strncpy**相似，只是参数类型与**strncpy**的不同。

函数**memcpy**与**memccpy**（见14.3节）提供与**strcpy**相似的功能。如果内存中有字符串共用存储空间，则所有这些函数的行为都是未定义的。对于可能发生的共用存储空间的情形，标准C语言中提供了函数**memmove**与**wmemmove**（见14.3节）。 350

13.4 strlen、wcslen

语法概要

```
#include <string.h>
size_t strlen(const char *s);
```

```
#include <wchar.h>
size_t wcslen(const wchar_t *s);
```

函数**strlen**返回**s**中终止null字符之前的字符数。空字符串的第一个字符为null字符，因此长度为0。在C语言的一些早期实现中，这个函数称为**lenstr**。

wcslen（C89增补1）与**strlen**相似，只是参数类型与**strlen**的不同。

13.5 strchr、strrchr、wcschr、wcsrchr

语法概要

```
#include <string.h>
char *strchr( const char *s, int c );
char *strrchr( const char *s, int c );
```

```
#include <wchar.h>
wchar_t *wcschr( const wchar_t *s, wchar_t c );
wchar_t *wcsrchr( const wchar_t *s, wchar_t c );
```

本节的函数在以null字符终止的字符串**s**中搜索一个字符**c**。在标准C语言函数中，**s**的终止null字符算是字符串的一部分，即如果**c**是null字符（0），则函数返回**s**中终止null字符的位置。在标准C语言中，参数**c**的类型为**int**；而在传统C语言中，参数**c**的类型为**char**。这些函数的返回值为指向非**const**的指针，但事实上如果第一个参数指向**const**对象，则指定的对象为**const**。这时，将数值存放在返回指针所指的对象中会造成未定义行为。

函数**strchr**搜索字符串**s**中第一次出现的字符**c**。如果在字符串中找到字符**c**，则返回指向第一个**c**的指针。如果没有在字符串中找到字符**c**，则返回null指针。

wcschr（C89增补1）与**strchr**相似，只是参数类型和返回值类型与**strchr**的不同。

函数**strrchr**与**strchr**相似，只是返回指向最后一个字符**c**的指针。如果找不到这个字符，则返回null指针。

wcsrchr（C89增补1）与**strrchr**相似，只是参数类型和返回值类型与**strrchr**的不同。

传统C语言函数**strpos**和**strchr**相似，只是返回值类型为**int**，返回第一个**c**的位置，**s**
351 中的第一个字符位置为0。如果找不到该字符，则返回 -1。函数**strrpos**与**strpos**相似，只是返回最后一个**c**的位置。标准C语言没有提供**strrpos**与**strpos**。

函数**memchr**与**wmemchr**（见14.1节）提供与**strchr**与**wcschr**相似的功能。在C语言的有些实现中，**strchr**与**strrchr**分别称为**index**与**rindex**。C语言的有些实现还提供函数**scnstr**，是**strpos**的变体。

例　下列函数**how_many**用**strchr**计算字符串中指定的非null字符出现的次数。参数**s**重复更新，指向上一个找到的字符后面的字符串位置：

```
int how_many(const char *s, int c)
{
    int n = 0;
    if (c == 0) return 0;
    while(s) {
        s = strchr(s, c);
        if (s) n++, s++;
    }
    return n;
}
```

□

13.6 strspn、strcspn、strpbrk、strrpbrk、wcsspn、wcscspn、wcspbrk

语法概要

```
#include <string.h>

size_t strspn( const char *s, const char *set );
size_t strcspn( const char *s, const char *set );
char *strpbrk( const char *s, const char *set );
```

```
#include <wchar.h>

size_t wcsspn( const wchar_t *s, const wchar_t *set );
size_t wcscspn( const wchar_t *s, const wchar_t *set );
wchar_t *wcspbrk( const wchar_t *s, const wchar_t *set );
```

本节的函数搜索以null终止的字符串**s**中指定的字符，主要的依据是看其是否包括在另一个以null终止的字符串**set**中。第二个参数是个字符集，字符的顺序及是否重复均不重要。

函数**strspn**搜索字符串**s**中第一个不包含在字符串**set**中的字符，跳过那些**set**中包含的字符。返回的值是**s**中从第一个字符起全部由**set**中字符组成的段的最大长度。如果**s**的每个字符都在**set**中，则返回**s**的总长度（不计算终止null字符）。如果**set**是个空字符串，则**s**中的第一个字符就不在**set**中，因此返回0。 352

函数**strcspn**与**strspn**相似，只是搜索字符串**s**中第一个在字符串**set**中的字符，跳过不在**set**中的字符。

函数**strpbrk**与**strcspn**相似，只是返回指向找到的**set**中第一个字符的指针，而不是跳过的字符数。如果没有找到**set**中的字符，则返回null指针。

非标准函数**strrpbrk**与**strpbrk**具有相同语法，但返回指向在**s**中找到的最后一个包含**set**中的字符的指针。如果**s**中没有找到**set**中的字符，则返回null指针。

wcsspn、**wcscspn**与**wcspbrk**函数（C89增补1）与相应的**str**函数相似，只是参数类型和结果类型与那些**str**函数的不同。

有时**strspn**与**strcspn**也称为**notstr**与**instr**。

例 函数**is_id**确定输入字符串是否是有效C语言标识符。**strspn**函数检查所有字符串字符是否是字母、数字或下划线。如果是，则进行最终测试，保证第一个字符不是数字。比较下列方案与12.1节的方案：

```
#include <string.h>
#define TRUE (1)
#define FALSE (0)

int is_id(const char *s)
{
    static char *id_chars =
           "abcdefghijklmnopqrstuvwxyz"
           "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
           "0123456789_";
    if (s == NULL) return FALSE;
    if (strspn(s,id_chars) != strlen(s)) return FALSE;
    return !isdigit(*s);
}
```

□ 353

13.7 strstr、strtok、wcsstr、wcstok

语法概要

```
#include <string.h>
char *strtok( char *str, const char *set );
char *strstr( const char *src, const char *sub );
```

```
#include <wchar.h>
wchar_t *wcstok(
  wchar_t *str, const wchar_t *set,wchar_t **ptr );
wchar_t *wcsstr(
  const wchar_t *src, const wchar_t *sub );
```

函数**strstr**是标准C语言新增加的，找到字符串**src**中第一次出现字符串**sub**的位置，并返回指向第一次出现的子串的开头的指针。如果**src**中没有**sub**，则返回null指针。**wcsstr**（C89增补1）与**strstr**相似，只是参数类型和返回值类型与**strstr**的不同。

可以用**strtok**函数将字符串**str**分解为由字符串**set**中的字符分隔的记号。可以对每个记号调用**strtok**，在连续调用中改变**set**的值。第一次调用包括字符串**str**，后续调用传入null指针作为第一个参数，让**strtok**从上一记号末尾开始处理（用**strtok**寻找字符串中的更多记号时，不能修改原先的字符串**str**）。

更准确地说，如果**str**不是null，则**strtok**首先跳过**str**中能在**set**中找到的所有字符。如果**str**中的所有字符都在**set**中，则**strtok**返回一个null指针，内部状态指针设置为null指针，否则内部状态指针设置为指向**str**中第一个不在**set**中的字符，然后继续执行，就像**str**是null一样。

如果**str**和内部状态指针都是null，则**strtok**返回null指针，内部状态指针不变（处理返回所有记号之后的最后一次**strtok**调用）。如果**str**为null，但内部状态指针不是null，则函数从内部状态指针开头开始搜索第一个在**set**中的字符。如果找到这种字符，则用**'\0'**覆盖这个字符，**strtok**返回内部状态指针的值，内部状态指针调整为指向插入的null字符的后一个字符。如果没有找到这种字符，则**strtok**返回内部状态指针的值，并将内部状态指针设置为null。

标准C语言中的库函数不允许以任何编程人员能够探测到的方式改变**strtok**的内部状态。编程人员无需考虑使用**strtok**的库函数会干扰编程人员自己的函数。

wcstok（C89增补1）与**strtok**相似，只是参数类型和返回值类型与**strtok**的不同。另外，还增加了一个**ptr**参数间接指定一个指针，作为**strtok**的“内部状态指针”，即**wcstok**的
354 调用者提供一个内部状态的保持器。

如果**strstr**或**wcsstr**的第一个参数是常量字符串的指针，则返回值也是常量字符串的指针，但不声明为**const**的指针。

例　下列程序读取标准输入行并用**strtok**将其分解为单词——即空格、逗号、句号、问号和引号分开的字符序列，然后在标准输出中打印这些单词：

```
#include <stdio.h>
#include <string.h>
#define LINELENGTH 80
#define SEPCHARS " .,?\"\n"
```

```
int main(void)
{
    char line[LINELENGTH];
    char *word;

    while(1) {
        printf("\nNext line?  (empty line to quit)\n");
        fgets(line,LINELENGTH,stdin);
        if (strlen(line) <= 1) break;   /* exit program */
        printf("That line contains these words:\n");
        word = strtok(line,SEPCHARS);/* find first word */
        while (word != NULL) {
            printf("\"%s\"\n",word);
            word = strtok(NULL,SEPCHARS);/*find next word*/
        }
    }
}
```

程序执行的示例如下：

```
Next line?  (empty line to quit)
"My goodness," she said, "Is that right?"
That line contains these words:
"My"
"goodness"
"she"
"said"
"Is"
"that"
"right"

Next line?  (empty line to quit)
```

□

13.8 strtod、strtof、strtold、strtol、strtoll、strtoul、strtoull

参见16.4节。 355

13.9 atof、atoi、atol、atoll

参见16.3节。

13.10 strcoll、strxfrm、wcscoll、wcsxfrm

语法概要

```
#include <string.h>
int strcoll( const char *s1, const char *s2 );
size_t strxfrm(
  char *dest, const char *src, size_t len );
```

```
#include <wchar.h>
int wcscoll(const wchar_t *s1, const wchar_t *s2);
size_t wcsxfrm(
  wchar_t *dest, const wchar_t *src, size_t len);
```

strcoll与**strxfrm**函数提供特定区域设置的字符串排序功能。**strcoll**函数比较字符串**s1**与**s2**，返回一个整数，在字符串**s1**大于、等于和小于**s2**时分别为大于、等于和小于0。比较计算根据特定区域设置的字符串排序规则（**LC_COLLATE**与**setlocale**，见11.5节）进行。不同的是，**strcmp**与**wcscmp**函数（13.2节）总是用目标字符集(**char**或**wchar_t**)的正常排序序列比较两个字符串。

wcscoll（C89增补1）与**strcoll**相似，只是参数类型与**strcoll**的不同。

strxfrm函数（按稍后介绍的方式）将字符串**src**变成字符数组**dest**中存放的第二个字符串，假设长度至少为**len**个字符。存储字符串所需的字符数（不包括终止null字符）由**strxfrm**返回。这样，如果**strxfrm**返回的值大于或等于**len**，或**src**与**dest**在内存中共用存储空间，则**dest**的最终结果是未定义的。此外，如果**len**为0而**dest**为null指针，则**strxfrm**只是计算并返回对应于**src**的转换字符串长度。

strxfrm函数转换字符串，使**strcmp**函数可以对这个转换字符串确定正确的排序顺序。如果**s1**与**s2**为字符串，而**t1**与**t2**是**strxfrm**从**s1**与**s2**产生的转换字符串，则：

- **strcmp(t1, t2) > 0**，如果**strcoll(s1, s2) > 0**
- **strcmp(t1, t2) == 0**，如果**strcoll(s1, s2) == 0**
- **strcmp(t1, t2) < 0**，如果**strcoll(s1, s2) < 0**

356

wcsxfrm（C89增补1）与**strxfrm**相似，只是参数类型不同。函数**wcscmp**用于比较经过转换的宽字符串。

函数**strcoll**与**strxfrm**有不同的性能取舍。**strcoll**函数不要求编程人员提供多余存储空间，但可能在每次调用时在内部进行字符串变换。如果同一组字符串要进行许多比较，则使用**strxfrm**函数可能更有效。

例　下列函数**transform**用**strxfrm**函数生成对应于参数**s**的转换字符串。字符串的空间是动态分配的。

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

char *transform( char *s )
/* Return the result of applying strxfrm to s */
{
   char *dest; /* Buffer to hold transformed string */
   size_t length; /* Buffer length required */
   length = strxfrm(NULL,s,0) + 1;
   dest = (char *) malloc(length);
   strxfrm(dest,s,length);
   return dest;
}
```

357

□

第14章 内存函数

本章介绍的函数向C语言编程人员提供复制、比较与设置内存块的有效方式。在标准C语言中，这些函数属于字符串函数，放在库头文件**string.h**中声明。而在早期的实现中，它们在另一个头文件**memory.h**中声明。

内存块在标准C语言中用**void ***类型指针指定，而在传统C语言中用**char ***类型指针指定。在标准C语言中，内存解释为**unsigned char**类型的对象数组；而传统C语言中没有显式指定，可以用**char**或**unsigned char**。这些函数处理null字符时，和处理其他字符没有任何差别。

C89增补1增加了5个操纵宽字符数组的新函数，用**wchar_t ***类型指针指定。这些函数在头**wchar.h**中定义，其名称以字母**wmem**开头。宽字符的顺序就是整型类型**wchar_t**中整数的顺序。宽字符串不进行解释，因此不会发生编码错误。

参考章节 **wchar_t** 11.1；宽字符 2.1.4

14.1 memchr、wmemchr

语法概要

```
#include <string.h>
void *memchr( const void *ptr, int val, size_t len );
#include <wchar.h>
wchar_t *wmemchr( const wchar_t *ptr, wchar_t val, size_t len );
```

函数**memchr**搜索以**ptr**开头的第一个**len**个字符中第一次出现的**val**，返回包含**val**的第一个字符指针（如有），如果找不到这种字符，则返回null指针。每个字符**c**与**val**比较，相当于 359
(unsigned char) c == (unsigned char) val。参见**strchr**（13.5节）。这些函数的返回值为非**const**的指针，但事实上如果第一个参数指向**const**对象，则指定的对象为**const**。

wmemchr函数（C89增补1）搜索以**ptr**开头的第一个**len**个宽字符中第一次出现的**val**，返回所找到的宽字符指针。如果找不到这种宽字符，则返回null指针。

在传统C语言中，**memchr**的语法如下：

```
#include <memory.h>
char *memchr(char *ptr, int val, int len );
```

14.2 memcmp、wmemcmp

语法概要

```
#include <string.h>
int memcmp( const void *ptr1, const void *ptr2, size_t len );
```

```
#include <wchar.h>
int wmemcmp(
  const wchar_t *ptr1, const wchar_t *ptr2, size_t len );
```

memcmp函数比较从**ptr1**开头的前**len**个字符与从**ptr2**开头的前**len**个字符。如果第一个字符串在词法上小于第二个字符串，则**memcmp**返回负值；如果第一个字符串在词法上大于第二个字符串，则**memcmp**返回正值；否则**memcmp**返回0。参见**strcmp**（13.2节）。

wmemcmp函数（C89增补1）对宽字符数组进行相同的比较。宽字符的顺序就是整型类型**wchar_t**中整数的顺序。根据**ptr1**处的宽字符小于、等于或大于**ptr2**处的宽字符序列，分别返回负数、0和正数。

早期的C语言实现可能包括函数**bcmp**，也是比较两个字符串，但在其相同时返回0，否则返回非0值，不比较大于或小于。**bcmp**与**memcmp**的传统C语言语法如下：

```
#include <memory.h>
int bcmp( char *ptr1, char *ptr2, int len );
int memcmp( char *ptr1, char *ptr2, int len );
```

360

14.3 memcpy、memccpy、memmove、wmemcpy、wmemmove

语法概要

```
#include <string.h>
void *memcpy (void *dest, const void *src, size_t len);
void *memmove(void *dest, const void *src, size_t len);
```

```
#include <wchar.h>
wchar_t *wmemcpy(
  wchar_t *dest, const wchar_t *src, size_t len);
wchar_t * wmemmove(
  wchar_t *dest, const wchar_t *src, size_t len);
```

memcpy与**memmove**函数（标准C语言）都从**src**到**dest**复制**len**个字符并返回**dest**值。差别在于**memmove**能对共用内存区正确工作，即**memmove**好像先把源内存区复制到另一临时内存区，然后再复制回目标内存区（事实上，实现**memmove**时不需要临时内存区）。**memcpy**的行为在源和目标共用存储区时是未定义的，但有些**memcpy**版本能够实现复制到临时内存区的词法。在两个函数都存在时，编程人员认为**memcpy**更快。参见**strcpy**（13.3节）。

函数**wmemcpy**与**wmemmove**（C89增补1）和**memcpy**与**memmove**相似，只是对宽字符数组进行运算，都返回**dest**。

除了**memcpy**之外，早期的C语言实现可能包括函数**memccpy**与**bcopy**。**memccpy**函数也是从**src**到**dest**复制**len**个字符，但在复制数值为**val**的字符之后立即停止。如果复制了全部**len**个字符，则**memccpy**返回null指针，否则返回**dest**中**val**的拷贝后面的字符指针。函数**bcopy**与**memcpy**相似，但保留源操作数和目标操作数。这些函数的传统C语言语法如下：

```
#include <memory.h>
char *memcpy( char *dest, char *src, int len );
```

```
char *memccpy(char *dest, char *src, int val, int len);
char *bcopy( char *src, char *dest, int len );
```

361

14.4 memset、wmemset

语法概要

```
#include <string.h>
void *memset( void *ptr, int val, size_t len );
```

```
#include <wchar.h>
wchar_t *wmemset( wchar t *ptr, int val, size_t len );
```

memset函数将**val**复制到从**ptr**开始的每**len**个字符。**ptr**所指定字符的为**unsigned char**。函数返回**ptr**值。

wmemset函数（C89增补1）与**memset**相似，只是填充宽字符数组。

早期的C语言实现可能包括更严格的函数**bzero**，将0复制到从**ptr**开始的每**len**个字符。这些函数的传统C语言语法如下：

```
#include <memory.h>
char *memset( char *ptr, int val, int len );
void bzero( char *ptr,int len );
```

362

第15章 输入/输出函数

基于数据流的概念，C语言有丰富而有用的输入/输出函数，这些函数可能是文件或其他某种数据源或数据使用者，包括终端或其他物理设备。数据类型**FILE**（在**stdio.h**中和其他输入/输出函数一起定义）保存数据流的信息。**FILE**类型的对象用**fopen**生成，其指针（文件指针）作为本章介绍的大多数输入/输出函数的参数。

FILE对象中包括数据流中的当前位置（文件位置）、任何相关缓冲区的指针以及是否遇到错误或文件末尾的指示符。数据流通常缓存，除非与交互式设备相关联。编程人员可以用**setvbuf**函数对缓冲进行一定控制，但一般来说数据流可以高效实现，编程人员不必考虑其性能。

数据流有两种通用类型：文本流与二进制流。文本流由分成行的字符序列组成；每一行包括0个或多个字符和一个换行符（**'\n'**）。文本流如果只包括标准字符集中的字符，则是可移植的。特定C语言运行库实现中的硬件与软件成员可能用不同方法表示文本文件（特别是文件结束指示符），但运行库应把这些实现映射为标准版本。标准C语言要求实现支持至少254个字符的文本流行（包括终止换行符）。

二进制流是**char**类型的数据值序列。由于任何C语言数据值都可以映射成**char**类型的数据值数组，因此二进制流可以透明地记录内部数据。实现不必区别文本流与二进制流。

C语言程序开始执行时，会预定义并打开3种文本流：标准输入（**stdin**）、标准输出（**stdout**）和标准错误（**stderr**）。

参考章节 **foper** 15.2；**setvbuf** 15.3；标准字符集 2.1

宽字符输入与输出 C89增补1增加了宽字符输入与输出功能。新的宽字符输入与输出函数放在头文件**wchar.h**中，对应于旧的字节输入/输出函数，只不过底层的程序数据类型和数据流元素是宽字符（**wchar_t**）而不是字符（**char**）。事实上，这些宽字符输入与输出函数的实现可以将宽字符与外部媒介上的多字节序列相互转换，但这通常是对编程人员透明的。

C89增补1不是对宽字符输入与输出生成新的流类型，而是在现有文本流与二进制流中增加一个定向（orientation）。打开数据流之后以及对数据流进行任何输入/输出操作之前，数据流是没有方向的。根据第一个输入/输出操作是宽字符还是字节函数，数据流变成面向宽字符或面向字节。一旦对数据流定向之后，只有同一方向的I/O函数才能使用，否则结果是未定义的。可以用**fwide**函数（15.2节）设置与测试数据流定向。

如果文件的外部表示为多字节字符序列，则文件中的一些多字节字符序列规则可以有所放松：

1. 文件中的多字节编码可以包含嵌入的null字符。

2. 文件不一定要以初始转换状态开始和结束。

不同文件可以对宽字符使用不同的多字节编码。一个文件的多字节编码逻辑上属于内部转换状态，可以在首次关联内部转换状态时通过设置区域设置的**LC_CTYPE**类别而确定，而不能在调用第一个宽字符输入与输出函数之后设置。关联文件的转换状态（和编码规则）之后，

LC_CTYPE的设置不能再影响相关数据流的转换。

由于宽字符与多字节字符之间的转换可能有相关状态，因此每个面向宽字符的数据流有个相关联的隐藏**mbstate_t**对象。输入/输出期间的转换概念上是用隐藏转换状态通过调用**mbrtowc**或**wcrtomb**发生的。**fgetpos**与**fsetpos**函数用文件位置记录这种转换状态。宽字符输入/输出期间的转换可能因为编码错误而失败，这时**EILSEQ**存放在**errno**中。如果允许多个文件编码方式，则数据流的编码可能放在**mbstate_t**对象中或至少要和它一起记录。

参考章节 转换状态 2.1.5；**EILSEQ** 11.2；**fgetpos**与**fsetpos** 15.5；**mbrtowc** 11.7；**mbstate_t** 11.1；多字节字符 2.1.5；定向 15.2.2；**wcrtomb** 11.7；宽字符 2.1.5 364

15.1 FILE、EOF、wchar_t、wint_t、WEOF

语法概要

```
#include <stdio.h>

typedef ... FILE ...;
#define EOF (-n)
#define NULL ...
#define size_t ...
```

```
#include <wchar.h>

typedef ... wchar_t;
typedef ... wint_t;
#define WEOF ...
#define WCHAR_MAX ...
#define WCHAR_MIN ...
#define NULL ...
#define size_t ...
```

整个标准I/O库用**FILE**类型表示数据流的控制信息，用于读取面向字节和宽字符的文件。

EOF值习惯上表示文件结束，即输入数据已经用光。大多数传统实现中，**EOF**的值为-1，但标准C语言只要求用负的整型常量表达式表示。由于**EOF**有时还表示其他问题，因此最好用**feof**功能（15.14节）确定返回**EOF**时是否实际遇到文件末尾。**WEOF**宏（C89增补1）在宽字符I/O中使用，像**EOF**在字节I/O中的作用一样，其数值的类型为**wint_t**（不一定是**wchar_t**），不一定是负值。**WCHAR_MAX**是类型**wchar_t**可以表示的最大值，**WCHAR_MIN**是类型**wchar_t**可以表示的最小值。

为了方便起见，类型**size_t**和null指针常量**NULL**在头文件**stdio.h**与**wchar.h**中定义。在标准C语言中，它们还在**stddef.h**中定义。使用多个头文件是无害的。

参考章节 **wchar_t** 2.1.5，11.1；**wint_t** 2.1.5，11.1 365

15.2 fopen、fclose、fflush、freopen、fwide

语法概要

```
#include <stdio.h>
FILE *fopen(
```

```
    const char * restrict filename, const char * restrict mode);
int fclose(FILE * restrict stream);
int fflush(FILE * restrict stream);
FILE *freopen(
    const char * restrict filename,
    const char * restrict mode,
    FILE * restrict stream);

#define FOPEN_MAX …
#define FILENAME_MAX …
```

```
#include <wchar.t>

int fwide(FILE * restrict stream, int orient);
```

函数**fopen**带有文件名和存取方式两个参数，分别指定为字符串。文件名按实现指定的方式打开或建立文件，将其和一个数据流相关联（宏**FILENAME_MAX**的值是文件名的最大长度，如果没有实际最大值时，则是适当长度）。函数返回指向**FILE***类型的指针，用于区别数据流和其他输入/输出操作。如果发现错误，则**fopen**把错误码存放在**errno**中，返回一个null指针。可以同时打开的数据流个数没有规定，标准C语言中可以用宏**FOPEN_MAX**的值指定，至少为8（包括3个预定义流）。在C89增补1中，**fopen**返回的数据流没有定向，可以对其进行面向字节或面向宽字符的操作，但不能同时进行两种操作。

函数**fclose**按适当方式有序地关闭已打开的数据流，包括清空必要的内部数据缓冲区。函数**fclose**在发现错误时返回**EOF**，否则返回0。

例　下面一些函数打开和关闭正常文本文件。它们处理必要的错误条件和打印必要的诊断信
366 息，返回值匹配**fopen**和**fclose**的返回值：

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

FILE *open_input(const char *filename)
    /* Open filename for input; return NULL if problem */
{
    FILE *f;
    errno = 0;
    /* Functions below might choke on a NULL filename. */
    if (filename == NULL) filename = "";
    f = fopen(filename,"r");    /* "w" for open_output */
    if (f == NULL)
        fprintf(stderr,
                "open_input(\"%s\") failed: %s\n",
                filename, strerror(errno));
    return f;
}

int close_file(FILE *f)
    /* Close file f */
{
    int s  = 0;
    if (f == NULL) return 0; /* Ingore this case */
    errno = 0;
```

```
    s = fclose(f);
    if (s == EOF) perror("Close failed");
    return s;
}
```

□

函数**fflush**用于清空与输出或更新数据流参数相关的任何缓冲区。数据流保持打开。如果发现任何错误，则**fflush**返回**EOF**，否则返回0。**fflush**通常只用于异常情况，正常情况下由**fclose**与**exit**负责刷新输出缓冲区。

函数**freopen**带有文件名、存取方式和打开的数据流3个参数，首先像调用**fclose**一样关闭**stream**，但这时发生的任何错误均被忽略。然后用**filename**与**mode**打开新文件，就像调用**fopen**一样，只是新数据流与**stream**相关联，而不是取得**FILE** *类型的新值。函数**freopen**成功时返回**stream**，否则（如果打开失败）返回null指针。**freopen**的一个主要用途是将标准输入/输出数据流**stdin**、**stdout**与**stderr**之一重新关联到另一文件。根据C89增补1，**freopen**从流中删除前面的任何定向。

参考章节 **EOF** 15.1；**exit** 19.3；**stdin** 15.4

15.2.1 文件访问方式

表15-1是函数**fopen**与**freopen**允许的文件访问方式值。 367

表15-1 函数fopen与freopen允许的文件访问方式值

访问方式①	含 义
"r"	打开现有文件以便输入
"w"	生成新文件或截尾现有文件以便输出
"a"	生成新文件或添加到现有文件以便输出
"r+"	打开现有文件，从文件开头开始更新（读和写）
"w+"	生成新文件或截尾现有文件以便更新
"a+"	生成新文件或追加到现有文件以便更新

① 所有访问方式都可以加上字母**b**，表示数据流保存二进制数据，而不是字符数据。

打开文件进行更新时（访问方式字符串中有+），得到的流可能用于输入和输出。但是，如果输出操作之后要进行输入操作，则中间要调用**fsetpos**、**fseek**、**rewind**或**fflush**；如果输入操作之后要进行输出操作，则中间要调用**fsetpos**、**fseek**、**rewind**或**fflush**，或输入操作遇到文件结尾（这些操作清空所有内部缓冲区）。

标准C语言允许表15-1所列的任何类型后面加上字母**b**，表示数据流保存二进制数据，而不是字符数据（这个区别在UNIX中是模糊的，因为在UNIX中这两种文件用相同方法处理，其他操作系统则没这么方便）。标准C语言还允许任何“更新”文件类型采用二进制方式，**b**指定符可以放在数据流方式指定的+之前或之后。

在标准C语言中，访问方式字符串还可以在上述方式符之后包含其他字符。实现可以用这些补充指定数据流的其他属性，例如：

```
f = fopen("C:\\work\\dict.txt","r,access=lock");
```

表15-2列出了每种数据流访问方式的一些属性。

表15-2　fopen访问方式的一些属性

属　性	方　式					
	r	w	a	r+	w+	a+
命名文件应已经存在	是	否	否	是	否	否
现有文件内容丢失	否	是	否	否	是	否
可以从数据流读取	是	否	否	是	是	是
可以向数据流写入	否	是	是	是	是	是
368 从数据流末尾开始写入	否	否	是	否	否	是

15.2.2 文件定向

可以用**fwide**函数（C89增补1）设置与测试流定向。调用之后，根据**stream**为面向宽字符、面向字节或无定向，函数返回正值、负值或0。**orient**参数确定**fwide**是否首先设置定向。如果**orient**为0，则不设置定向，返回值反映调用时的定向。如果**orient**为正值，则**fwide**设置宽字符定向；如果**orient**为负值，则**fwide**设置字节定向。这些设置只有前面没有定向时才能成功，即刚刚用**fopen**或**freopen**打开；否则定向保持不变。

例　使用面向宽字符流时，最好用**fwide**确定调用**fopen**时的定向。下列函数按指定方式打开指定文件，并设置为指定区域设置中的面向宽字符流。如果成功，则函数返回文件指针，否则返回**NULL**。

```
FILE *fopen_wide(
    const char *filename, /* file to open */
    const char *mode,     /* mode for open */
    const char *locale)/* locale for encoding */
{
    FILE *f = fopen(filename, mode);
    if (f != NULL) {
        char *old_locale = setlocale(LC_CTYPE, locale);
        if (old_locale == NULL || fwide(f, 1) <= 0) {
            fclose(f);   /* setlocale or fwide failed */
            f = NULL;
        }
        /* return locale to its original value */
        setlocale(LC_CTYPE, old_locale);
    }
    return f;
}
```

□

使用的多字节编码方式（如有）在建立数据流的定向时确定。它受到建立定向时当前区域
369 设置的**LC_CTYPE**类别影响。

15.3 setbuf、setvbuf

语法概要

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

```
int setvbuf(
  FILE * restrict stream,
  char *b restrict uf,
  int bufmode,
  size_t size );
void setbuf(
  FILE * restrict stream,
  char * restrict buf );

#define BUFSIZ ...
#define _IOFBF ...
#define _IOLBF ...
#define _IONBF ...
```

这些函数允许编程人员在极少数默认缓冲无法满足要求的情况下控制数据流的缓冲策略。函数要在打开流之后和读取或写入任何数据之前调用。

函数**setvbuf**是从UNIX System V借鉴的更一般函数。第一个参数是要控制的数据流，第二个参数（如果不是null）是代替自动产生的缓冲区的字符数组，**bufmode**指定缓冲类型，**size**指定缓冲区长度。如果成功，则函数返回0，如果参数不合适或无法满足请求，则返回非0值。

宏**_IOFBF**、**_IOLBF**与**_IONBF**扩展成**bufmode**可以使用的值。如果**bufmode**为**_IOFBF**，则数据流完全缓冲；如果**bufmode**为**_IOLBF**，则写入换行符时或缓冲区满时刷新缓冲区；如果**bufmode**为**_IONBF**，则数据流不缓冲。如果请求缓冲而**buf**不是null指针，则**buf**指定的数组长度应为**size**字节，代替自动产生的缓冲区。常量**BUFSIZ**是缓冲区长度的“适当”值。

函数**setbuf**是**setvbuf**的简化形式。表达式

```
setbuf(stream,buf)
```

等价于表达式

```
((buf==NULL) ?
  (void) setvbuf(stream,NULL,_IONBF,0) :
  (void) setvbuf(stream,buf,_IOFBF,BUFSIZ))
```

参考章节 **EOF** 15.1; **fopen** 15.2; **size_t** 11.1

370

15.4 stdin、stdout、stderr

语法概要

```
#include <stdio.h>

#define stderr ...
#define stdin  ...
#define stdout ...
```

表达式**stdin**、**stdout**与**stderr**的类型为**FILE ***，数值在应用程序开始某些标准文本流之前确定。**stdin**指向输入流，是程序的“正常输入”，**stdout**指向输出流，是程序的“正常输出”，**stderr**所指的输出流是错误消息和其他意外输出。在交互式环境中，这3个数据流通常都与启动程序的终端相关联，除**stderr**之外，另外两个数据流都缓冲。

这些表达式通常不是lvalue，任何情况下都不能通过赋值而改变，但可以用**freopen**函数

（见15.2节）来改变。

例　表达式**stdin**、**stdout**与**stderr**通常定义为静态或全局数据流描述符的地址：

```
extern FILE __iob[FOPEN_MAX];
...
#define stdin (& __iob[0])
#define stdout (&__iob[1])
#define stderr (&__iob[2])
```

□

特别地，UNIX系统提供了在启动应用程序时将这些流与文件或其他程序相关联的方便方法，使其在根据某些标准规则使用时非常强大。

在C89增补1中，**stdin**、**stdout**与**stderr**在启动C语言程序时没有定向。因此，这些数据流可以调用**fwide**（见15.2节）用于宽字符输入/输出，也可以对其使用宽字符输入/输出
371 函数。

15.5 fseek、ftell、rewind、fgetpos、fsetpos

语法概要

```
#include <stdio.h>
int fseek(
  FILE * restrict stream, long int offset, int wherefrom);
long int ftell(FILE * restrict stream);
void rewind(FILE * restrict stream);
#define SEEK_SET 0
#define SEEK_CUR 1
#define SEEK_END 2
typedef ... fpos_t ...;
int fgetpos( FILE * restrict stream, fpos_t *pos );
int fsetpos( FILE * restrict stream, const fpos_t *pos );
```

本节的函数可以随机访问文本流与二进制流，通常数据流与文件相关联。

15.5.1 fseek与ftell

函数**ftell**带有打开输入/输出的数据流参数，返回数据流中的位置通常采用**fseek**的第二个参数适合的数值形式。对保存的**ftell**结果使用**fseek**函数可以将数据流位置复位到文件中调用**ftell**的位置。

对二进制文件，返回的值为当前文件位置之前的字符数。对文本文件，返回的值是由实现定义的。返回的值应能在**fseek**中使用，数值**0L**表示文件开头，而不一定是惟一开头。

如果**ftell**遇到错误，则返回**-1L**，并将**errno**设置为由实现定义的正值。由于**-1L**可能被误认为有效文件位置，因此要检查**errno**以确认错误。造成**ftell**失败的条件包括在连接终端的数据流中定位一个位置或报告无法表示为**long int**类型对象的位置。

函数**fseek**可以随机访问打开的**stream**。第二个参数指定文件位置：**offset**是个带符号长整数，对二进制数据流指定字符数，**wherefrom**是“定位代码”，表示从文件中哪个点开始测量**offset**。数据流的定位方法如下，如果成功，则**fseek**返回0，否则返回下一个非0值

(errno的值不变)。然后清除任何文件结尾指示符，取消ungetc的任何效果。标准C语言定义了表示wherefrom值的常量SEEK_SET、SEEK_CUR与SEEK_END，编程人员使用非标准实现时要使用指定的整数值或定义宏。 372

重新定位二进制文件时，新位置由下表指定：

wherefrom值	新 位 置
SEEK_SET或0	距离文件开头offset个字符
SEEK_CUR或1	距离文件当前位置offset个字符
SEEK_END或2	距离文件末尾offset个字符（负值表示在文件末尾之前，正值用未指定的内容扩展文件）

标准C语言不要求实现对二进制流的wherefrom值的SEEK_END提供“有意义的”支持。标准C语言对文本流允许下列有限的调用形式。

调用形式	定位（文本）数据流
fseek(stream, 0L, SEEK_SET)	在文件开头
fseek(stream, 0L, SEEK_CUR)	在同一位置（即调用无效）
fseek(stream, 0L, SEEK_END)	在文件末尾
fseek(stream, ftell-pos, SEEK_SET)	在上次对stream调用ftell返回的位置

这些限制认识到文本文件中的位置可能无法直接映射文件的内部表示方法。例如，一个位置可能要求记录号和记录中的偏移量（但标准C语言要求实现支持对文本文件调用fseek(stream, 0L, SEEK_END)，而不必对二进制流提供“有意义的”支持）。

根据C89增补1，对面向宽字符的数据流进行的文件定位操作应满足二进制或文本文件的所有限制。一般来说，ftell和fseek函数不够强大，无法支持面向宽字符的数据流，即使进行最简单的定位操作也不行，例如定位数据流开头和结尾。面向宽字符的数据流应使用下节介绍的fgetpos与fsetpos函数来定位。

函数rewind将数据流复位到开头。根据标准C语言定义，调用rewind(stream)等价于：

```
(void) fseek(stream, 0L, SEEK_SET)
```

15.5.2 fgetpos与fsetpos

函数fgetpos与fsetpos是标准C语言新增加的。因为处理太大的文件时，其位置无法用整数类型long int（像ftell与fseek中一样）表示，所以增加了这两个函数。 373

fgetpos函数将当前文件位置存放在pos所指的对象中，在成功时返回0，如果遇到错误，则返回非0值，在errno中存储实现定义的正值。

fsetpos函数根据*pos的值设置当前文件位置，这个值应为前面fgetpos在同一个数据流中返回的值。fsetpos函数取消ungetc或ungetwc的任何影响，它在成功时返回0，如果遇到错误，则返回非0值，在errno中存储实现定义的正值。

根据C89增补1，fgetpos与fsetpos使用的文件位置对象要包括与面向宽字符的数据流相关联的隐藏转换状态的表示（即mbstate_t类型的值）。这个状态和文件位置一起用于在重定位操作之后解释后面的多字节字符。

在面向宽字符的输出流中，用fsetpos设置输出位置之后再写入一个或几个多字节字符会

使文件中所有后面的多字节字符变成未定义。这是因为，输出可能部分地覆盖现有多字节字符或改变转换状态，使后面的多字节字符无法正确解释。

参考章节 **mbstate_t** 11.1; **ungetc** 15.6

15.6 fgetc、fgetwc、getc、getwc、getchar、getwchar、ungetc、ungetwc

语法概要

```
#include <stdio.h>

int fgetc(FILE *stream);
int getc(FILE *stream);
int getchar(void);
int ungetc(int c, FILE *stream);
```

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

wint_t fgetwc(FILE *stream);
wint_t getwc(FILE *stream);
wint_t getwchar(void);
wint_t ungetwc(wint_t c, FILE *stream);
```

函数**fgetc**取输入流作为参数，从输入流中读取下一个字符并将其作为**int**类型的值返回，内部数据流位置指示符前移。连续调用**fgetc**从输入流中返回连续字符。如果发生错误或数据流
374 到达文件末尾，则**fgetc**返回**EOF**。这时要用**feof**与**ferror**功能确定是否实际到达文件末尾。

函数**getc**与**fgetc**相似，只是**getc**通常实现为宏，更加高效。**stream**参数不能有任何副作用，因为它可能求值多次。

函数**getchar**等价于**getc(stdin)**。和**getc**一样，**getchar**也通常实现为宏。

根据C89增补1，函数**fgetwc**、**getwc**与**getwchar**与对应的面向字节函数相似，包括可能的宏实现，但它们的功能是从输入流中读取和返回下一个宽字符。返回**WEOF**表示错误或文件结尾，如果错误是编码错误，则在**errno**中存储**EILSEQ**。读取宽字符时要从多字节字符转换成宽字符，这就像用数据流的内部转换状态调用**mbrtowc**一样。

函数**ungetc**将字符**c**（转换成**unsigned char**）推回指定的输入流，在下次对这个数据流调用**fgetc**、**getc**与**getchar**函数时返回这个字符。如果推进几个字符，则其按相反顺序返回（后进先出）。**ungetc**在字符顺利推回时返回**c**，失败时返回**EOF**。数据流中成功采用文件定位命令（**fseek**、**fsetpos**或**rewind**）时放弃所有推回的字符。读取或放弃所有推回的字符后，文件位置和推回字符之前一样。

如果数据流是缓存的，上次对流进行**fseek**、**fopen**或**freopen**操作以来至少读取了数据流中的一个字符，则能保证推回一个字符。将**EOF**值作为字符推回数据流中不会对数据流产生影响，并返回**EOF**。调用**fsetpos**、**rewind**、**fseek**或**freopen**从数据流中删除所有推回字符内存区，不影响与数据流相关联的任何外部存储区。

函数**ungetc**适用于实现**scanf**之类的输入扫描操作。程序可以“向前看”下一个输入字符，读取这个字符并在不合适时将其推回（但**scanf**和其他库函数不允许编程人员抢先使用**ungetc**，即编程人员即使在调用**scanf**之类的函数之后，也要保证至少有一个推回字符）。

函数**ungetwc**（C89增补1）与**ungetc**相似。

参考章节 **EOF** 15.1；**feof** 15.14；**fseek** 15.5；**fopen** 15.2；**freopen** 15.2；**scanf** 15.8；**stdin** 15.4

375

15.7 fgets、fgetws、gets

语法概要

```
#include <stdio.h>
char *fgets(char *s, int n, FILE *stream);
char *gets(char *s);
```

```
#include <stdio.h>
#include <wchar.h>
wchar_t *fgetws(wchar_t *s, int n, FILE *stream);
```

函数**fgets**带有3个参数：字符数组开头的指针**s**、数字**n**和输入流。字符从输入流读取到**s**中，直到遇到换行符、到达文件末尾或已经读取**n** - 1个字符而没有遇到换行符或到达文件末尾。然后在字符读取之后在数组末尾加一个终止null字符。如果因为遇到换行符而终止输入，则换行符存放在数组中终止null字符前面。顺利完成时，返回参数**s**。

如果从输入流中读取任何字符之前遇到末尾，则**fgets**返回null指针，数组**s**的内容不变。如果输入操作期间发生错误，则**fgets**返回null指针，数组**s**的内容不能确定。应当用**feof**函数（15.14节）确定返回**NULL**时是否遇到了文件末尾。

函数**gets**从标准输入流**stdin**读取字符到字符数组**s**中。但与**fgets**不同的是，换行符终止输入时，**gets**放弃换行符，不把它放到**s**中。使用**gets**是很危险的，因为输入长度可能超过字符数组中可用的存储空间。函数**fgets**更安全，因为**s**中最多只能放进**n**个字符。

函数**fgetws**（C89增补1）与**fgets**相似，但它处理面向宽字符的输入流，并将宽字符存放在**s**中，包括末尾的null宽字符。没有对应于**gets**的宽字符函数，这是避免使用**gets**的另一暗示。

参考章节 **feof** 15.14；**stdin** 15.4

376

15.8 fscanf, fwscanf, scanf, wscanf, sscanf, swscanf

语法概要

```
#include <stdio.h>
int fscanf(
  FILE * restrict stream, const char * restrict format, ...);
int scanf(
  const char * restrict format, ...);
int sscanf(
  char *s, const char * restrict format, ...);
```

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

```
int fwscanf(
  FILE * restrict stream,
  const wchar_t * restrict format, ...);
int wscanf(
  const wchar_t *format, ...);
int swscanf(
  wchar_t *s, const wchar_t *format, ...);
```

函数**fscanf**分析格式化输入文本，从第一个参数指定的数据流读取字符并根据控制字符串格式转换字符序列。根据控制字符串内容，可能还需要其他参数。控制字符串后面的每个参数应为指针，从输入流读取的值转换之后存放在指针指定的对象中。

函数**scanf**与**sscanf**和**fscanf**相似。对于**scanf**，从标准输入流**stdin**读取字符。对于**sscanf**，则是从字符串**s**读取字符。**sscanf**读取到字符串**s**之外时，遇到文件末尾时的操作和**fscanf**与**scanf**一样。

输入操作可能因为输入流到达文件末尾或控制字符串与从输入流读取的字符之间发生冲突而提前终止。这些函数返回的值是由于某些原因造成操作终止之前顺利进行的赋值次数。如果发生冲突或进行赋值之前遇到文件末尾，则函数返回**EOF**。发生冲突时，造成冲突的字符保持未读，在下次输入操作中处理。

C89增补1定义了一组宽字符格式化输入函数，对应于**fscanf**、**scanf**与**sscanf**。新的**wscanf**系列函数使用宽字符控制字符串，要求输入为宽字符序列。外部文件中底层多字节序列的任何转换都是对编程人员透明的。下面的介绍中描述面向字节函数。除非另有说明，面向宽
377 字符的函数行为可以把“字符”或“字节”换成“宽字符”而得到。

C89增补1还扩展标准C语言格式字符串，允许在**s**、**c**和**[**转换操作中指定**l**长度指定符，表示相关参数是宽字符串或字符的指针。详见这些转换操作的描述。

15.8.1 控制字符串

控制字符串是期望的输入形式。在标准C语言中，对于**scanf**系列，这是个多字节字符序列，从初始shift状态开始和结束；对于**wscanf**系列，这是宽字符序列。可以理解为这些函数在控制字符串和输入流之间进行简单的匹配操作。控制字符串内容可以分为3类：

空白符　控制字符串中的空白符使空白符被读取和放弃。将遇到的第一个非空白输入字符作为要从输入流读取的下一个字符。注意，如果控制字符串中出现几个连续空白符，则效果上与只有一个空白符相同。这样，控制字符串中的任何连续空白符序列可以匹配输入流中的任何连续空白符序列，两者长度可以不一样。

转换说明　转换说明以**%**号开始，其余语法见稍后详细介绍。从输入流读取的字符数取决于转换操作。作为总原则，转换操作处理字符，一直到（a）到达文件末尾；（b）遇到空白符或另一个不适合的字符；（c）从转换操作读取的字符数等于指定的最大字段宽度。处理的字符通常经过转换（例如变成数字值）并存放在控制字符串后面的指针参数指定的位置。

其他字符　除空白符和**%**号以外的任何其他字符都要匹配输入流中下一个字符。如果不匹配，则发生冲突，终止转换操作，并使造成冲突的字符在输入流中保持未读，在下次输入操作中处理。

指针参数要有正确的个数和正确类型，符合控制字符串中的转换说明。如果参数太多，则

忽略多余参数；如果参数太少，则结果是未定义的。如果任何转换说明有错，则行为也是未定义的。在每个转换说明执行操作之后有一个序列点。

15.8.2 转换说明

转换说明以%号开始，然后要按下列顺序出现转换说明元素：

1. 可选赋值取消标志（*）。如果通常要进行赋值的转换操作中有这个标志，则按正常方式从输入流读取和处理字符，但不进行赋值，不占用指针参数。 378
2. 可选最大字段长度，表示为正的十进制整数。
3. 可选长度说明，表示为字符序列hh、h、l、ll、j、z、t或L。表15-3列出了这些字符序列适用的转换操作。hh、ll、j、z与t长度说明是C99新增加的。
4. 必要的转换操作（或转换说明符），用一个字符表示：a、c、d、e、f、g、i、n、o、p、s、u、x、%或[，只有[操作还要把后面的字符和]合起来作为一个转换说明。

fscanf转换指定的语法和含义与fprintf相似，但有一定差别。可以把fprintf与fscanf的控制字符串语法看成大致相似，但不要用一个函数的文档作为另一函数的指导。

例 下面是fprintf与fscanf转换说明的一些差别：

[转换操作是fscanf特有的。

fscanf不接受fprintf所接受的任何精度说明，也不接受fprintf所接受的标志字符-、+、空格、0与#。

显式说明的字符宽度在fprintf中是最小值，而在fscanf中是最大值。

fprintf允许用计算参数说明字段宽度，用星号表示字段宽度，而fscanf中的星号另有用途，是赋值取消标志，这也许是两者最明显的差别。 □

除了上述差别之外，所有转换操作跳过任何初始空白符之后再转换。这个初始空白符不计入最大字段长度。转换操作通常不跳过尾部的空白符。尾部的空白符（如终止输入行的换行符）保持未读，除非在控制字符串中显式匹配（这可能有问题，因为控制字符串中的空白符匹配输入中的许多空白符，从而可能读取到换行符以外）。

控制字符串中的直接字符匹配成功与否无法直接确定，也无法直接确定涉及赋值取消的转换操作成功与否。这些函数返回的值只反映了顺利进行的赋值次数。

转换操作很复杂，表15-3是简单小结，后面将详细介绍。 379

表15-3 输入转换（scanf fscanf sscanf）

转换字母	长度说明符	参数类型	输入格式
d	无	int *	[-|+]dd...d
i[1]	hh	char *	[-|+][0[x]]dd...d[2]
	h	short *	
	l	long *	
	ll[3]	long long *	
	j	intmax_t *	
	z	size_t *	
	t	ptrdiff_t *	
u	无	unsigned *	[-|+]dd...d

（续）

转换字母	长度说明符	参数类型	输入格式
o	hh	unsigned char *	[-|+]dd...d[2]
x	h	unsigned short *	[-|+][0x]dd...d[4]
	l	unsigned long *	
	ll[5]	unsigned long long *	
	j	uintmax_t *	
	z	size_t *	
	t	ptrdiff_t *	
c	无	char *	定宽字符序列，使用l时应为多字节
	l[1]	wchar_t *	
s	无	char *	非空白符序列，使用l时应为多字节
	l[1]	wchar_t *	
p[2]	无	void **	与fprintf中%p输出相似的字符序列
n[1]	无	int *	无，读取的字符数存放在参数中
	hh	char *	
	h	short *	
	l	long *	
	ll[5]	long long *	
	j	intmax_t *	
	z	size_t *	
	t	ptrdiff_t *	
a[5], f, e, g	无	float *	任何浮点数常量或十进制整数常量，前面可选加上+或-
	l	double *	
	L[3]	long double *	
[	无	char *	从扫描集得到的字符序列，使用l时应为多字节
	l[1]	wchar_t *	

① C89中新增的。
② 数字的进制由第一个数确定，像C语言常量一样。
③ 数字为八进制。
④ 数字为十六进制，不管是否有0x。
⑤ C99中新增的。

d转换　进行带符号十进制转换，使用一个参数，类型为**int ***、**short ***或**long ***，取决于长度说明。

380

读取的数字格式与**strtol**函数的输入相同（对**wscanf**为**wcstol**），**base**参数为10，即十进制数位序列，前面可选加上+或-。如果输入表示的值太大，无法表示为相应长度的带符号整数，则行为是未定义的。

i转换　进行带符号整数转换，使用一个参数，类型为**int ***、**short ***或**long ***，取决于长度说明。

读取的数字格式与**strtol**函数的输入相同（对**wscanf**为**wcstol**），**base**参数为0，即不带后缀的C语言整型常量，前面可选加上+或-、**0**（八进制）和**0x**（十六进制）前缀。如果输入

表示的值太大，无法表示为相应长度的带符号整数，则行为是未定义的。

u转换 进行无符号十进制转换，使用一个参数，类型为**unsigned ***、**unsigned short ***或**unsigned long ***，取决于长度说明。

读取的数字格式与**strtoul**函数的输入相同（对**wscanf**为**wcstoul**），**base**参数为10，即十进制数位序列，前面可选加上+或-。如果输入表示的值太大，无法表示为相应长度的带符号整数，则行为是未定义的。

o转换 进行无符号八进制转换，使用一个参数，类型为**unsigned ***、**unsigned short ***或**unsigned long ***，取决于长度说明。

读取的数字格式与**strtoul**函数的输入相同（对**wscanf**为**wcstoul**），**base**参数为8，即八进制数位序列，前面可选加上+或-。如果输入表示的值太大，无法表示为相应长度的带符号整数，则行为是未定义的。

x转换 进行无符号十六进制转换，使用一个参数，类型为**unsigned ***、**unsigned short***或**unsigned long***，取决于长度说明。

读取的数字格式与**strtoul**函数的输入相同（对**wscanf**为**wcstoul**），**base**参数为16，即十六进制数位序列，前面可选加上+或-。字符**0123456789abcdefABCDEF**是有效的十六进制数字。如果输入表示的值太大，无法表示为相应长度的带符号整数，则行为是未定义的。

一些非标准C语言实现接受字母**x**作为对等转换操作。 381

c转换 读取一个或几个字符。使用一个指针参数，应为类型**char ***，在有**l**长度说明时为**wchar_t ***。**c**转换操作不跳过初始空白符。转换适用于输入字符，取决于是否有**l**长度说明符以及是用**scanf**还是**wscanf**。表15-4列出可能的了**c**说明符的输入转换。

表15-4 c说明符的输入转换

函 数	长度说明符	参数类型	输 入	转 换
scanf	无	**char ***	字符	无，复制字符
	l	**wchar_t ***	多字节字符	转换为宽字符，就像一个或几个**mbrtowc**调用
wscanf	无	**char ***	宽字符	转换为多字节字符，就像一个或几个**wcrtomb**调用
	l	**wchar_t ***	宽字符	无，复制宽字符

如果不说明字段宽度，则读取一个字符，除非输入流在文件末尾，这时转换操作失败。字符值赋予下一个指针参数所指的地址。

如果说明字段宽度，则指针参数设定为指向字符数组开头，这个字段宽度指定要读取的字符数，如果读取字符之前遇到文件末尾，则转换操作失败。读取的字符存放在数组中的连续地址内。读取的字符末尾不添加另一终止null字符。

s转换 读取字符串。使用一个指针参数，应为类型**char ***（C89增补1），在有**l**长度说明时为**wchar_t ***。**c**转换操作总是跳过初始空白符。

字符一直读取，直到文件结束、遇到空白符（这时该字符保持未读）或（指定字段宽度时）读取最大字符数。如果在遇到任何非空白符之前遇到文件结束，则转换操作失败。转换适用于输入字符，取决于是否有**l**长度说明符和是用**scanf**还是**wscanf**（表15-5）。如果**scanf**使用**l**

说明符，则第一个空白符终止输入，这发生在输入字符解释为多字节字符之前。

存储的字符末尾总是加一个终止null字符。s转换操作在不说明最大字符宽度时很危险，因为输入长度可能超过字符数组的存储空间。

具有显式说明字段宽度的s操作不同于具有显式说明字段宽度的c操作。c操作不跳过空白符，读取指定的字符数（或宽字符数），除非遇到文件末尾。s操作跳过初始空白符，在读取一定的
382 非空白符字符数（或宽字符数）之后用空白符终止，并在存储的字符末尾加一个终止null字符。

表15-5　s说明符的输入转换

函　数	长度说明符	参数类型	输　入	转　换
scanf	无	char *	字符	无，复制字符
	l	wchar_t *	多字节字符	转换为宽字符，就像mbrtowc调用
wscanf	无	char *	宽字符	转换为多字节字符，就像wcrtomb调用
	l	wchar_t *	宽字符	无，复制宽字符

p转换　进行指针转换。使用一个参数，应为void **类型。读取的指针值格式是由实现指定的，但通常与printf系列中%p转换产生的格式相同。指针的解释也是由实现指定的，但如果写出指针之后要读回（在同一程序执行期间）这个指针，则读取的指针与写出的指针相比是相等的。p转换是标准C语言增加的。

n转换　不进行转换，不读取字符，而是把当前调用scanf系列函数所处理的字符和写入参数，根据说明的长度，参数类型为int *、short *或long*。n转换是标准C语言增加的。

a、f、e与g转换　进行带符号十进制浮点数转换。在C99中，a转换的输入与f、e、g相同。使用一个指针参数，根据说明的长度，参数类型为float *、double*或long double *。

读取的数字格式与strtod函数的输入相同（对wscanf为wcstod），即十进制或十六进制数位序列，前面可选加上+或-，还可以包含小数点和带符号指数部分（可以接受不带小数点的整数）。输入字符串INF、INFINITY、NAN与NAN(...)表示特殊的浮点数（忽略大小写）。接受十六进制浮点数输入的特性是C99中增加的。

读取的字符解释为浮点数表示，转换成说明长度的浮点数。如果不读取数字或遇到指数部分之前不读取数字，则数值为0。如果字母引入指数之后不读取数字，则指数部分假设为0。如果输入表示的值太大或太小，无法用相应长度的浮点数表示，则返回相应符号的数值
383 HUGE_VAL并在errno中存储ERANGE（对不符合标准C语言的实现，返回值和errno的设置无法预测）。如果输入表示的值不会太大或太小，但仍然无法准确表示为相应长度的浮点数，则进行某种截尾或舍入。

a、f、e、g转换操作是完全一致的，都可以接受任何样式的浮点数表示。有些实现还接受G和E作为浮点数转换字母。

%转换　输入中要有一个百分号。由于百分号表示转换说明的开头，因此要写两个%号才能匹配一个%号。不使用指针参数，赋值取消标志、字段长度和长度说明与%转换操作无关。

[转换　读取字符串，使用一个指针参数，应为类型char *，在有l长度说明时为wchar_t *。[转换操作不跳过初始空白符。转换说明表示输入字段中读取什么字符。[后面要放上多个字符的控制字符串，然后加上]号。到]号之前的所有字符都属于转换说明，称为*扫描集*（scanset）。如果[后面的字符是个^号，则具有特殊含义，表示否定标志，扫描集包括所有不在^号与]号之

间的字符。扫描集中的字符是数学意义上的集合。

初始[与终止]之间的任何[号被看成另一字符。同样，不紧跟在初始[后面的任何^号也被看成另一字符。在标准C语言中，如果初始[号后面紧跟着]号，则它属于扫描集，由下一个]终止转换说明。如果]号后面紧跟一个^号，则这个]不在扫描集中，由下一个]终止转换说明。早期的实现可能不支持在转换说明开头这样对]进行特殊处理。

例

转换说明	扫描集中包含
%[abca]	3个字符a、b与c
%[^abca]	除3个字符a、b与c以外的所有字符
%[[]	一个字符[
%[]]	一个字符]
%[,\t]	空格、逗号和水平制表符

□

读取字符，直到文件末尾、遇到不在扫描集中的字符（这个字符保持未读）或（说明字符宽度时）读取最大字符数。如果不用*取消赋值，则输入字符存放在参数指针说明的对象中，就像**s**转换操作一样，包括任何与多字节字符之间的转换（见表15-5）。然后在存储的字符后面添加一个终止null字符。长度说明与[转换操作无关。 384

和**s**转换一样，[转换操作在不说明最大字符宽度时很危险，因为输入长度可能超过字符数组的存储空间。

参考章节 **EOF** 15.1; **fprintf** 15.11; **stdin** 15.4

15.9 fputc、fputwc、putc、putwc、putchar、putwchar

语法概要

```
#include <stdio.h>

int fputc(int c, FILE *stream);
int putc(int c, FILE *stream);
int putchar(int c);
```

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

wint_t fputwc(wchar_t c, FILE *stream);
wint_t putwc(wchar_t c, FILE *stream);
wint_t putwchar(wchar_t c);
```

函数**fputc**的参数是一个字符值和一个输出流。它在输出流的当前位置写入字符，并将这个字符作为**int**类型的值返回。连续调用**fputc**时在输出流中连续写入某个字符。如果发生错误，则**fputc**返回**EOF**，而不是写入的字符。

函数**putc**与**fputc**相似，但通常用宏实现。参数表达式不能有任何副作用，因为它们可能多次求值。

函数**putchar**向标准输出流**stdout**写入一个字符。和**putc**一样，**putchar**通常用宏实现，

相当有效。调用**putchar(c)**等价于**putc(c, stdout)**。

C89增补1增加了宽字符函数**fputwc**、**putwc**与**putwchar**，对应于面向字节函数。遇到错误时返回数值**WEOF**。如果发生编码错误，则还在**errno**中存储**EILSEQ**。

385 **参考章节** **EOF** 15.1; **stdout** 15.4

15.10 fputs、fputws、puts

语法概要

```
#include <stdio.h>

int fputs(const char *s, FILE *stream);
int puts(const char *s);
```

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

int fputws(const wchar_t *s, FILE *stream);
```

函数**fputs**的参数为一个以null终止的字符串和一个输出流。将字符串中的所有字符写入输出流中，不包括null终止字符。如果发生错误，则**fputs**返回**EOF**，否则返回某个非负值。

函数**puts**与**fputs**相似，但字符总是写入标准输出流**stdout**。**s**中的字符写出之后，再写入一个换行符（不管**s**中是否包含换行符）。

一些非标准UNIX中的**fputs**实现有错误，在**s**是空字符串时无法确定返回值。编程人员可能会在这种边界情况中收到警报信息。

C89增补1增加了宽字符函数**fputws**，对应于面向字节函数。遇到错误时返回数值**EOF**（而不是**WEOF**）。如果发生编码错误，则还在**errno**中存储**EILSEQ**。

386 **参考章节** **EOF** 15.1; **stdout** 15.4

15.11 fprintf、printf、sprintf、snprintf、fwprintf、wprintf、swprintf

语法概要

```
#include <stdio.h>

int fprintf(
  FILE * restrict stream, const char * restrict format, ... );
int printf(
  const char * restrict format, ...);
int sprintf(
  char * restrict s,
  const char * restrict format, ...);
int snprintf(
  char * restrict s, size_t n,
  const char * restrict format, ...);      // C99
```

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

int fwprintf(
  FILE * restrict stream,
```

```
  const wchar_t * restrict format, ... );
int wprintf(
  const wchar_t * restrict format, ...);
int swprintf(
  wchar_t *s, size_t n, const wchar_t *format, ...);
```

函数**fprintf**进行输出格式化，将输出发送到第一个参数指定的数据流，第二个参数是格式控制字符串，其他参数取决于控制字符串。根据控制字符串的指示，产生一系列输出字符，这些字符发送到指定数据流。

函数**printf**与**fprintf**相似，但字符总是写入标准输出流**stdout**。

sprintf函数把输出字符存放在字符串缓冲区**s**中。输出控制字符串指定的所有字符之后，最后的null字符输出到**s**中。编程人员要负责保证**sprintf**的目标字符串存储区足够大，能够包含格式化操作产生的输出。但是，**swprintf**函数与**sprintf**不同，还包括写入输出字符串的最大宽字符数（包括终止null字符）。在C99中，**snprintf**可以计算非宽函数的字符数。

如果输出操作期间遇到错误，则这些函数的返回值为**EOF**，否则返回其他结果值。在标准C语言和大多数当前实现中，函数在没有遇到错误时返回发送到输出的字符数。对于**sprintf**，这个数不包括终止null字符（遇到错误时，标准C语言允许这些函数返回任何负值）。 387

C89增补1指定了这些函数的3个宽字符版本：**fwprintf**、**wprintf**与**swprintf**。这些函数的输出概念上是宽字符串，在转换运算符控制下将其他参数转换成宽字符串。我们把这些函数称为**wprintf**系列函数或**wprintf**函数，区别于原先面向字节的**printf**函数。C89增补1可以在**printf**与**wprintf**函数中对**c**和**s**转换运算符采用**l**长度说明符。

C99对十六进制浮点数转换引入**a**和**A**转换运算符，引入**hh**、**ll**、**j**、**z**与**t**长度说明符。

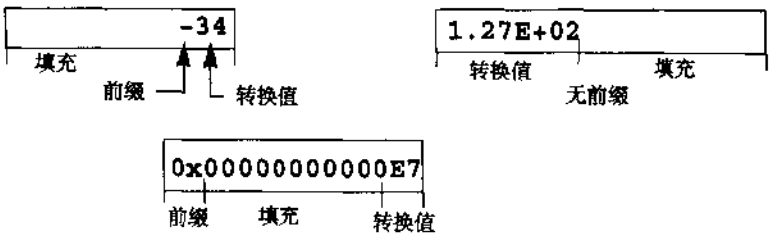
参考章节 **EOF** 15.1；十六进制浮点数格式 2.7.2；**scanf** 15.8；**stdout** 15.4；宽字符 2.1.4

15.11.1 输出格式

控制字符串就是要原样复制的文本，但其中还可以包含转换说明。在标准C语言中，控制字符串是以初始shift状态开始和结束的（未解释）多字节字符序列。在**wprintf**函数中，这是个宽字符串。

转换说明可能要求处理另外几个参数，得到的格式化转换操作中生成的输出字符没有显式地包含在控制字符串中。多余参数忽略，但太少参数的结果是无法预测的。如果任何转换说明有错误，则结果是无法预测的。输出转换说明与fscanf之类的输入转换说明相似，15.8.2节介绍了它们的差别。在每个转换说明的操作之后有一个序列点。

转换说明的字符序列或宽字符输出可以分为3个元素：*转换值*，反映转换参数的值；前缀（如有），通常是+、-或空格；*填充*，是空格序列或0序列，在必要时将输出序列宽度增加到指定的最小值。转换值前面总是放上前缀。根据转换说明，填充可能放在前缀前面、前缀与转换值之间或转换值之后。下图3个例子的方框中显示了转换说明控制的输出扩展。



388

15.11.2 转换说明

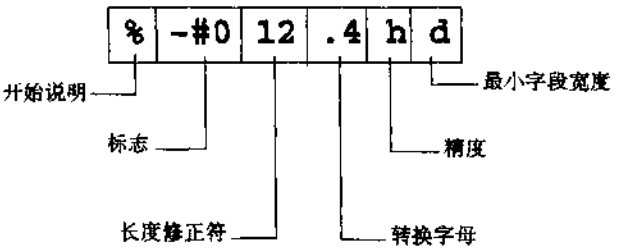
在下面的介绍中，字符、字母等都表示printf函数中的正常字符或字母（字节）和wprintf函数中的宽字符或字母。例如，在wprintf中，转换说明以宽字符百分号（%）开始。

转换说明以百分号（%）开始，然后依次出现下列元素：

1. 0个或多个标志字符（-、+、0、#或空格），修改转换操作的含义。
2. 可选最小字段宽度，表示为十进制整数常量。
3. 可选精度说明，表示为小数点加一个十进制整数。
4. 可选长度说明，表示为字母ll、l、L、h、hh、j、z或t。
5. 转换操作，字母a、A、c、d、e、E、f、g、G、i、n、o、p、s、u、x、X或%。

长度说明字母L与h和转换操作i、p、n是C89引入的，C99对十六进制浮点数转换引入a和A转换运算符，引入hh、ll、j、z与t长度说明符。

转换字母终止说明。转换说明%-#012.4hd的构成元素如下：



15.11.3 转换标志

可选标志字符调整主要转换操作的含义：

-	左对齐字段宽度中的值
0	用0而不用空格作为填充符
+	总是产生一个符号+或-
空格	总是产生一个符号-或空格
#	用主转换操作的变体

389

下面将详细地介绍标志字符的作用。

-标志　如果遇到-标志，则转换值左对齐字段宽度，即任何填充都放在转换值右边。如果没有负号，则转换值右对齐字段宽度。这个标志只在显式说明字段最小宽度而转换值小于字段最小宽度时才有效，否则数值在字段中填满，不进行填充。

0标志　如果有0标志，则在转换值左边要进行填充时，以0作为填充字符。0标志只在显式说明字段最小宽度而转换值小于字段最小宽度时才有效。在整型转换中，这个标志被精度说明覆盖。

如果没有0标志，则用空格作为填充符。如果要在转换值右边放上填充符，则总是用空格作为填充符，即使有-标志。

+标志　如果有+标志，则带符号转换的结果总是以一个符号开始，即正值前面显式放上+号（不管有没有+标志，负值前面总是放上-号）。这个标志只对转换操作a、A、d、e、E、f、g、G与i有效。

空格标志　如果有空格标志而带符号转换的结果中第一个字符不是符号（+或-），则在转换

值前面加一个空格。左边增加的这个空格独立于-标志控制下可能在左边或右边放上的任何填充符。如果一个转换说明中同时有空格和+标志，则忽略空格标志，因为+标志保证转换前面总是以符号开始。这个标志只对转换操作a、A、d、e、E、f、g、G与i有效。

#标志 如果有#标志，则使用主转换操作的替换形式。这个标志只对转换操作a、A、e、E、f、g、G、i、o、x与X有效。#标志所做的修改将在介绍相关转换操作时介绍。

15.11.4 最小字段宽度

可以说明可选最小字段宽度，表示为十进制整数常量。这个常量应为十进制数字的非空序列，开头不能是0（否则变成0标志）。如果转换值（包括前缀）得到的字符数少于说明的最小字段宽度，则用填充符将数值填充到说明宽度；如果转换值得到的字符数多于说明的最小字段宽度，则扩展字段长度，而不进行填充。

最小字段宽度也可以用星号说明，这时使用int类型的参数，说明最小字段宽度。说明负宽度的结果是无法预测的。 390

例 下面两个printf调用得到相同的结果：

```
int width=5, value;
…
printf("%5d", value);
printf("%*d", width, value);
```

□

15.11.5 精度说明

还可以指定一个可选精度说明，表示为点号加一个十进制整数。精度说明的作用如下：

1. d、i、o、u、x与X转换要打印的最少位数。
2. e、E与f转换小数点右边的位数。
3. g与G转换中的有效位数。
4. s转换中要从字符串写入的最大字符数。

如果有点号而没有整数，则这个整数设定为0，通常与省略精度说明具有不同效果。

精度也可以指定为点号加星号，这时使用int类型参数和说明精度。如果宽度与精度都用星号说明，则字段宽度完全放在精度参数前面。

15.11.6 长度说明

有些转换操作可以放上可选长度修正符，表示为字母ll、l、L、h、hh、j、z或t。

字母l用于转换操作d、i、o、u、x与X，表示转换参数的类型为long与unsigned long；用于n转换时，表示转换参数的类型为long *。在C89中，c转换也可以使用l修正符，表示转换参数的类型为wint_t；s转换也可以使用l修正符，表示转换参数的类型为wchar_t *。修正符l对a、A、e、E、f、F、g与G转换无作用，使用时要小心。

修正符ll用于转换操作d、i、o、u、x与X，表示转换参数的类型为long long int或unsigned long long int；用于n转换时，表示转换参数的类型为long long int *。修正符ll是C99中引入的。

字母h用于转换操作d、i、o、u、x与X，表示转换参数的类型为short或unsigned short。即尽管参数升级可以将参数转换成int或unsigned，但转换之前要先变为short或unsigned short类型。字母h用于n转换时，表示转换参数的类型为short *。修正符h是

391 C89中引入的。

字母**hh**用于转换操作**d**、**i**、**o**、**u**、**x**与**X**，表示转换参数的类型为**char**或**unsigned char**。即尽管参数升级可以将参数转换成**int**或**unsigned**，但转换之前要先变为**char**或**unsigned char**类型。字母**hh**用于**n**转换时，表示转换参数的类型为**signed char ***。修正符**hh**是C99中引入的。

字母**L**用于转换操作**a**、**A**、**e**、**E**、**f**、**F**、**g**与**G**，表示转换参数的类型为**long double**。**L**修正符是C89引入的。注意**long double**用**L**而不是**l**，因为**l**对这些操作无效。

修正符**j**用于转换操作**d**、**i**、**o**、**u**、**x**与**X**，表示转换参数的类型为**intmax_t**或**uintmax_t**；用于**n**转换时，表示转换参数的类型为**intmax_t ***。修正符**j**是C99中引入的。

修正符**z**用于转换操作**d**、**i**、**o**、**u**、**x**与**X**，表示转换参数的类型为**size_t**；用于**n**转换时，表示转换参数的类型为**size_t ***。修正符**z**是C99中引入的。

修正符**t**用于转换操作**d**、**i**、**o**、**u**、**x**与**X**，表示转换参数的类型为**ptrdiff_t**；用于**n**转换时，表示转换参数的类型为**ptrdiff_t ***。修正符**t**是C99中引入的。

15.11.7 转换操作

转换操作表示为一个字符**a**、**A**、**c**、**d**、**e**、**E**、**f**、**g**、**G**、**i**、**n**、**o**、**p**、**s**、**u**、**x**、**X**或**%**。说明转换确定允许的标志与长度字符、所要的参数类型及输出的样子。表15-6总结了转换操作，每个转换操作将在后面一一介绍。

表15-6 输出转换说明

转换	定义的标志 - + # 0 空格	长度修正符	参数类型	默认精度①	输　出
d、i②	- + 0 空格	无	int	1	dd...d
		h	short		-dd...d
		l	long		+dd...d
u	- + 0 空格	无	unsigned int	1	dd...d
		h	unsigned short		
		l	unsigned long		
o	- + # 0 空格	无	unsigned int	1	oo...o
		h	unsigned short		0oo...o
		l	unsigned long		
x、X	- + # 0 空格	无	unsigned int	1	hh...h
		h	unsigned short		0xhh...h
		l	unsigned long		0Xhh...h
f	- + # 0 空格	无	double	6	d...d.d...d
		l	double		-d...dd...d
		L	long double		+d...dd...d
e、E	- + # 0 空格	无	double	6	d.d...de+dd
		l	double		-d.d...dE-dd
		L	long double		
g、G	- + # 0 空格	无	double	6	同e、E或f
		l	double		

（续）

转换	定义的标志 - + # 0 空格	长度修正符	参数类型	默认精度①	输 出
		L	long double		
a、A③	- + # 0 空格	无	double	6	0xh.h...hp+dd
		l	double		-0xh.h...hp-dd
		L	long double		
c	-	无	int	1	c
		l④	wint_t		
s	-	无	char *	x	cc...c
		l④	wchar_t *		
p②	实现定义	无	void *	1	由实现定义
n②		无	int *	n/a	无
		h	short *		
		l	long *		
%		无	无	n/a	%

① 不说明时为默认精度。
② C89引入，转换i与d的输出等价。
③ C99引入。
④ C89增补1引入。

d与i转换 进行带符号十进制转换。如果不用长度修正符，则参数类型为**int**，使用长度修正符h时为参数类型**short**，使用长度修正符l时参数类型为**long**。标准C语言提供i操作符，保持与**fscanf**的兼容性，在输出中能够识别，保证统一性，与d操作符一致。

转换值包括十进制数序列，表示参数的绝对值。这个序列尽量短，但不能短于说明精度。转换值必要时可以在前面加0，满足精度说明，这些前导0独立于任何填充。如果精度为1（默认），则转换值没有前导0，除非参数为0，这时输出单个0。如果精度为0而参数为0，则转换值是空的（null字符串）。

前缀计算如下。如果参数是负值，则前缀为负号。如果参数是非负值并说明+标志，则前缀为加号。如果参数是非负值并说明空格而不是+标志，则前缀为空格。否则前缀为空。#标志与d和i转换无关。表15-7显示了d转换的例子。

u转换 进行无符号十进制转换。如果不用长度修正符，则参数类型为**unsigned**，使用长度修正符h时参数类型为**unsigned short**，使用长度修正符l时参数类型为**unsigned long**。

392
~
393

转换值包括十进制数序列，表示参数的绝对值。这个序列尽量短，但不能短于说明精度。转换值必要时可以在前面加0，满足精度说明，这些前导0独立于任何填充。如果精度为1（默认），则转换值没有前导0，除非参数为0，这时输出单个0。如果精度为0而参数为0，则转换值为空（null字符串）。前缀总为空的，+、空格和#标志与u转换无关。表15-8显示了u转换的例子。

表15-7 d转换例子

示例格式	示例输出 值=45	示例输出 值=-45
%12d	45	-45
%012d	000000000045	-00000000045

（续）

示例格式	示例输出 值=45	示例输出 值=-45
% 012d	000000000045	-00000000045
%+12d	+45	-45
%+012d	+00000000045	-00000000045
%-12d	45	-45
%- 12d	45	-45
%-+12d	+45	-45
%12.4d	0045	-0045
%-12.4d	0045	-0045

表15-8　u转换例子

示例格式	示例输出 值=45	示例输出 值=-45
%14u	45	4294967251
%014u	00000000000045	00004294967251
%#14u	45	4294967251
%#014u	00000000000045	00004294967251
%-14u	45	4294967251
%-#14u	45	4294967251
%14.4u	0045	4294967251
%-14.4u	0045	4294967251

o转换　进行无符号八进制转换。如果不用长度修正符，则参数类型为**unsigned**，使用长度修正符**h**时参数类型为**unsigned short**，使用长度修正符**l**时参数类型为**unsigned long**。

转换值包括八进制数序列，表示参数的绝对值。这个序列尽量短，但不能短于说明精度。转换值必要时可以在前面加0，满足精度说明，这些前导0独立于任何填充。如果精度为1（默认），则转换值没有前导0，除非参数为0，这时输出单个0。如果精度为0而参数为0，则转换值为空（null字符串）。如果有#标志，则前缀为0；如果没有#标志，则前缀为空。+标志与空格标志与o转换无关。表15-9显示了o转换的例子。

394

表15-9　o转换例子

示例格式	示例输出 值=45	示例输出 值=-45
%14o	55	37777777723
%014o	00000000000055	00037777777723
%#14o	055	037777777723
%#014o	00000000000055	00037777777723
%-14o	55	377777777723
%-#14o	055	037777777723
%14.4o	0055	37777777723
%-#14.4o	00055	037777777723

x与X转换　进行无符号十六进制转换。如果不用长度修正符，则参数类型为**unsigned**，使用

长度修正符h时参数类型为unsigned short，使用长度修正符l时参数类型为unsigned long。

转换值包括十六进制数序列，表示参数的绝对值。这个序列尽量短，但不能短于说明精度。x操作用0123456789abcdef作为数位，而X操作用0123456789ABCDEF。转换值必要时可以在前面加0，满足精度说明，这些前导0独立于任何填充。如果精度为1（默认），则转换值没有前导0，除非参数为0，这时输出单个0。如果精度为0而参数为0，则转换值为空（null字符串）。如果不指定精度，则默认为1。

如果有#标志，则前缀为0x（对x操作）或0X（对X操作）；如果没有#标志，则前缀为空。+标志与空格标志与x与X转换无关。表15-10显示了x与X转换的例子。

表15-10 x与X转换例子

示例格式	示例输出 值=45	示例输出 值=－45
%12x	2d	ffffffd3
%012x	00000000002d	0000ffffffd3
%#12x	0X2D	0XFFFFFFD3
%#012x	0X000000002D	0X00FFFFFFD3
%-12x	2d	ffffffd3
%-#12x	0x2d	0xffffffd3
%12.4x	002d	ffffffd3
%-#12.4x	0x002d	ffffffd3

395

c转换 参数打印为字符或宽字符。使用一个参数。+、空格、#标志和精度说明都与c转换操作无关。根据是否有l长度说明符和是否使用printf或wprintf，参数字符的转换方式不同，见表15-11。表15-11显示了c转换的例子。

表15-11 c说明符的转换

函 数	长度说明符	参数类型	转 换
printf	无	int	参数转换成unsigned char并复制到输出
	l	wint_t	参数转换成wchar_t，像wcrtomb①一样转换成多字节字符并输出
wprintf	无	int	参数像btowc一样转换成宽字符并输出
	l	wint_t	参数转换成wchar_t并复制到输出

① 字符转换之前，wcrtomb函数的转换状态设置为0。

表15-12 c转换例子

示例格式	示例输出 值='*'
%12c	*
%012c	00000000000*
%-12c	*

s转换 参数打印成字符串。使用一个参数。如果没有l长度说明符，则参数应为任何字符类型数组的指针。如果有长度说明符l，则参数类型为wchar_t *并指定宽字符序列。前缀总

为空。+、空格和#标志与s转换无关。

如果不说明精度，则转换值是字符串参数中的字符序列，到终止null字符或null宽字符为止，但不包括终止null字符或null宽字符。如果精度说明为*p*，则转换值是输出字符串中前*p*个字符或到终止null字符或null宽字符为止，但不包括终止null字符或null宽字符（取其中较短者）。提供精度说明时，参数字符串不一定要以null字符结尾，只要其中包含足够字符，能达到最大输出字符数。编写多字节字符时（如**printf**函数带有长度说明符**l**），不可能写入部分多字节字符，因
396 此写入的实际字节数可能少于*p*。

根据是否有**l**长度说明符和是否使用**printf**或**wprintf**，参数字符的转换方式不同，见表15-13。表15-14显示了**s**转换的例子。

表15-13　s说明符的转换

函　数	长度说明符	参数类型	转　换
printf	无	**char ***	参数字符串中的字符复制到输出
	l	**wchar_t ***	参数字符串中的宽字符转换成多字节字符，像使用**wcrtomb**①一样
wprintf	无	**char ***	参数字符串中的多字节字符转换成宽字符，像使用**mbrtowc**①一样
	l	**wchar_t ***	参数字符串中的宽字符复制到输出

① **awcrtomb**与**mbrtowc**函数的转换状态在转换第一个字符之前设置为0。后续转换使用前面字符修改的状态。

表15-14　s转换例子

示例格式	示例输出 值="zap"	示例输出 值="longish"
%12s	**zap**	**longish**
%12.5s	**zap**	**longi**
%012s	**000000000zap**	**00000longish**
%-12s	**zap**	**longish**

p转换　参数应为**void ***类型，按实现定义格式打印。对大多数计算机，可能与**o**、**x**或**X**转换产生的格式相同。这个转换操作符在标准C语言中提供，但并不常用。

n转换　如果不用长度修正符，则参数类型为**int ***，使用长度修正符**l**时参数类型为**long ***，使用长度修正符**h**时参数类型为**short ***。这个转换操作符除了输出字符，还将迄今为止输出的字符数写入指定整数。这个转换操作符在标准C语言中提供，但并不常用。

f与F转换　进行带符号十进制浮点数转换。使用一个参数，如果不用长度修正符，则参数类型为**double**，使用长度修正符**L**时参数类型为**long double**。如果提供**float**类型参数，则通过普通参数升级转换为**double**类型，因此可以用**%f**打印**float**类型的数字。

397 转换值包括十进制数字序列，可能带小数点，表示参数的近似绝对值。小数点前面至少有一个数字。精度说明了小数点后面的位数，如果精度为0，则小数点后面的位数为0。此外，除非使用#标志，否则不出现小数点。如果不说明精度，则默认为6。

如果浮点数值无法用产生的位数准确表示，则转换值应为准确浮点数值舍入到符合产生位数的十进制数结果（一些C语言实现并不能在所有情况下正确地舍入）。

在C99中，如果浮点数值表示无穷大，则使用**f**操作符的转换值为**inf**、**-inf**、**infinity**

或-infinity之一（具体由实现定义）。如果浮点数值表示NaN，则使用f操作符的转换值为nan、-nan、nan(...)或-nan(...)，其中省略号是由实现定义的字母、数字或下划线序列。F操作符用大写字母转换NaN和无穷大。#标志和0标志对NaN和无穷大的转换无效。

前缀的计算方法如下。如果参数是负值，则前缀为负号。如果参数是非负值并说明+标志，则前缀为加号。如果参数是非负值并说明空格标志而不是+标志，则前缀是一个空格。表15-15显示了f转换的例子。

表15-15 f转换例子

示例格式	示例输出 值=12.678	示例输出 值=-12.678
%10.2f	12.68	12.68
%010.2f	0000000012.68	00000012.68
% 010.2f	00000012.68	00000012.68
%+10.2f	+12.68	12.68
%+010.2f	+00000012.68	00000012.68
%-10.2f	12.68	12.68
%- 10.2f	12.68	12.68
%-+10.4f	+12.6780	12.6780

e和E转换 进行带符号十进制浮点数转换。使用一个参数，如果不用长度修正符，则参数类型为double，使用长度修正符L时参数类型为long double。和f转换一样，可以使用float类型参数。下面介绍e转换，E转换只是把字母e换成E。

转换值由一个十进制数，然后可能包括小数点和更多十进制数，随后是字母e，随后是正号或负号，最后还有至少两个十进制数共同组成。除非数值为0，否则字母e前面的部分表示1.0到9.99...的值，而字母e后面的部分表示指数值，是个带符号十进制整数。第一部分的值与以第二部分值作为10的指数得到的数相乘，得到参数的近似绝对值。所有值的指数位相同，是表示实现中浮点数范围的最大位数。表15-16显示了e与E转换的例子。 398

表15-16 e与E转换例子

示例格式	示例输出 值=12.678	示例输出 值=-12.678
%10.2e	1.27e+01	-1.27e+01
%010.2e	00001.27e+01	-0001.27e+01
% 010.2e	0001.27e+01	-0001.27e+01
%+10.2E	+1.27E+01	-1.27E+01
%+010.2E	+0001.27E+01	-0001.27E+01
%-10.2e	1.27e+01	-1.27e+01
%- 10.2e	1.27e+01	-1.27e+01
%-+10.2e	+1.27e+01	-1.27e+01

精度说明小数点后面的位数，如果不说明精度，则默认为6。如果精度为0，则小数点后面的位数为0。此外，除非使用#标志，否则不出现小数点。如果浮点数值无法用产生的位数准确表示，则转换值应为准确浮点数值舍入到符合产生位数的浮点数结果。无穷大或NaN值的表示

和f转换与F转换中一样。

g与G转换　进行带符号十进制浮点数转换。使用一个参数，如果不用长度修正符，则参数类型为double，使用长度修正符L时参数类型为long double。与f转换一样，可以使用float类型参数。下面介绍g转换，G操作也一样，只是g使用e转换，而G使用E转换。如果说明精度少于1，则使用精度1。如果不说明精度，则默认为6。

g转换从f或e转换开始，取决于转换的值。标准C语言规范指出，只有e转换得到的指数小于－4时或大于等于说明精度时才使用e转换。有些实现在转换得到的指数小于－3时或大于说明精度时使用e转换。

f或e转换的　转换值进一步修改，删除小数点后面的尾部0。如果结果的小数点后面没有数字，则删除小数点。如果有#标志，则不删除0和小数点。

前缀的确定与f或e转换相同。无穷大或NaN值的表示和f与F转换中一样。

a与A转换　a与A转换是C99新增加的。进行带符号十六进制浮点数转换。使用一个参数，如果不用长度修正符，则参数类型为double，使用长度修正符L时参数类型为long double。
399 与f转换一样，可以使用float类型参数。下面介绍a转换。A转换只是十六进制数前缀和指数字母使用大写（0X和P）。

转换值包括一个十六进制数，然后可能包括小数点和更多十六进制数，随后是字母p，然后是正号或负号，最后是一个或几个十进制数。除非数值为0或非规格化，否则前面的十六进制数为非0。字母p后面的部分表示二进制指数值，是带符号十进制整数。

精度说明小数点后面的十六进制位数，如果不说明，则用足够位数区别double类型的值（如果FLT_RADIX为2，则默认精度足以准确表示这个值）。如果精度为0，则小数点后面的位数为0。此外，除非使用#标志，否则不出现小数点。如果浮点数值无法用产生的位数准确表示，则转换值应为准确浮点数值舍入到符合产生位数的浮点数结果。

无穷大或NaN值的表示和f与F转换中一样。

%转换　打印一个百分号。由于百分号表示转换说明的开头，因此要写两个%号才能打印出一个%号。不使用参数，前缀是空白。

在标准C语言中。%转换不允许有任何标志字符、最小宽度、精度和长度说明符，完整的转换说明为%%。但是，有些C语言实现和任何其他转换操作一样进行填充，例如转换说明%05%在这些实现中打印出0000%。+标志、空格标志、#标志、精度和长度说明符与%转换操作无关。

例　下面几行程序称为奎因（quine），是一种自我复制程序，执行时会在标准输出中打印自身的拷贝（程序第一行太长，因此我们在%cmain()后面插入反斜杠和分行符，表示续行）。

```
char*f="char*f=%c%s%c,q='%c',n='%cn',b='%c%c';%cmain()\
{printf(f,q,f,q,q,b,b,b,n,n);}%c",q='"',n='\n',b='\\';
main(){printf(f,q,f,q,q,b,b,b,n,n);}
```

下面的程序看上去也像是个奎因，我们在";main()后面插入反斜杠和分行符，表示续行。读者可以看看，为什么它不是真正的奎因。

```
char*f="char*f=%c%s%c;main(){printf(f,34,f,34);}";main()\
400 {printf(f,34,f,34);}
```

□

15.12 v[x]printf、v[x]scanf

语法概要

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

int vfprintf(FILE * restrict stream,
  const char * restrict format, va_list arg);
int vprintf(
  const char * restrict format, va_list arg);
int vsprintf(char *s,
  const char * restrict format, va_list arg);
int vfscanf(FILE * restrict stream,
  const char * restrict format, va_list arg); // C99
int vscanf(
  const char * restrict format, va_list arg); // C99
int vsscanf(const char * restrict s,
  const char * restrict format, va_list arg); // C99
```

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

int vfwprintf(FILE * restrict stream,
  const wchar_t * restrict format, va_list arg);
int vwprintf(
  const wchar_t * restrict format, va_list arg);
int vswprintf(wchar_t * restrict s,
  size_t n, const wchar_t * restrict format, va list arg);
int vfwscanf(FILE * restrict stream,
  const wchar_t * restrict format, va_list arg); // C99
int vswscanf(const wchar_t * restrict s,
  const wchar_t * restrict format, va_list arg); // C99
int vwscanf(
  const wchar_t * restrict format, va_list arg); // C99
```

函数**vfprintf**、**vprintf**与**vsprintf**分别与**fprintf**、**printf**与**sprintf**相似，只是增加了参数，作为**varargs**（或**stdarg**）函数定义的可变参数表（见11.4节）。参数**arg**要用**va_start**宏进行初始化，后面可能还要调用**va_arg**。这些函数可以让编程人员定义自己的可变参数函数，使用格式化输出函数。这些函数不调用**va_end**函数。

C89增补1增加的**vfwprintf**、**vwprintf**与**vswprintf**函数分别与**fwprintf**、**wprintf**与**swprintf**相似。C99增加了相应输入函数**vfscanf**、**vscanf**与**vsscanf**及其宽字符版本**vfwscanf**、**vwscanf**与**vswscanf**。

401

例 假设要编写一个一般函数**trace**，打印函数名及其参数。任何要跟踪的函数先调用下列形式的**trace**函数：

```
trace(name,format,parm1,parm2,...,parmN)
```

其中**name**是要调用的函数名，**format**是打印参数**parm1**、**parm2**、...、**parmN**的格式字符串。例如：

```
int f(int x, double y) /* Trace this function. */
```

```
{
    trace("f","x=%d, y=%f", x, y);
    ...
}
```

下面是传统C语言中实现**trace**的一种方法：

```
#include <varargs.h>
#include <stdio.h>
void trace(va_alist)
    va_dcl
{
    va_list args;
    char *name;
    char *format;
    va_start(args);
    name = va_arg(args,char *);
    format = va_arg(args,char *);
    fprintf(stderr,"--> entering %s(", name);
    vfprintf(stderr, format, args);
    fprintf(stderr,")\n");
    va_end(args);
}
```

□

15.13 fread、fwrite

语法概要

```
#include <stdio.h>
size_t fread(
  void * restrict ptr, size_t element_size, size_t count,
  FILE * restrict stream);
size_t fwrite(
  const void * restrict ptr, size_t element_size, size_t count,
  FILE * restrict stream);
```

函数**fread**与**fwrite**分别输入和输出到二进制文件。两种情况下，**stream**是输入或输出
402 流，**ptr**是指向**count**元素数组的指针，每个元素是**element_size**个字符长。

函数**fread**从输入流读取最多**count**个指定长度的元素到指定数组中，返回实际读取的项目数，如果遇到文件末尾，则这个数可能小于**count**。如果遇到错误，则返回0。可以用**feof**或**ferror**函数确定返回0时是因为遇到错误还是遇到文件末尾。如果**count**或**element_size**为0，则不传输数据，返回0。

例　下列程序读取包含结构类型对象的输入文件，打印读取的对象数。程序用**exit**关闭输入文件：

```
/* Count the number of elements
   of type "struct S" in file "in.dat" */
#include <stdio.h>
static char *FileName = "in.dat";
struct S { int a,b; double d; char str[103]; };

int main(void)
```

```
{
    struct S buffer;
    int items_read = 0;
    FILE *in_file = fopen(FileName,"r");
    if (in_file == NULL)
    { fprintf(stderr,"?Couldn't open %s\n",FileName);
      exit(1); }

    while (fread((char *) &buffer,
           sizeof(struct S), 1, in_file) == 1)
       items_read++;

    if (ferror(in_file))
       { fprintf(stderr,"?Read error, file %s record %d\n",
          FileName,items_read+1); exit(1); }
    printf("Finished; %d elements read\n",items_read);
    return 0;
}
```

□

函数**fwrite**从指定数组写入**count**个长度为**element_size**的元素，返回实际写入的项目数，除非遇到错误，否则这个值等于**count**。

在传统C语言中，**element_size**参数类型为**unsigned**，**ptr**参数类型为**char***，**count**参数类型为**int**。

参考章节 **exit** 19.3；**feof**、**ferror** 15.14；**fseek**、**ftell** 15.5

403

15.14 feof、ferror、clearerr

语法概要

```
#include <stdio.h>

int feof(FILE *stream);
int ferror(FILE *stream);
void clearerr(FILE *stream);
```

函数**feof**的参数是个输入流。如果读取输入流时遇到文件末尾，则返回一个非0值，否则返回0。注意，即使流中没有更多要读取的字符，**feof**也不指示遇到文件末尾，除非要读取到最后一个字符之外。这个函数通常在输入操作失败之后使用。

函数**ferror**返回数据流的错误状态。如果读取或写入数据流期间发生错误，则**ferror**返回一个非0值，否则返回0。一个数据流发生错误之后，重复调用**ferror**时继续报告错误，除非用**clearerr**显式地重置错误条件。用**fclose**关闭数据流时也重置错误条件。

函数**clearer**重置数据流中的任何错误和文件末尾指示，后面再调用**ferror**时不再报告错误，除非发生另一错误。

15.15 remove、rename

语法概要

```
#include <stdio.h>

int rename(
```

```
    const char *oldname, const char *newname);
int remove(const char *filename);
```

remove函数删除指定文件，在操作成功时返回0，否则返回非0值。**filename**所指的字符串不改变。不同实现中“remove”与“delete”的细节可能不同，但程序不能打开已经删除的文件。如果文件打开或不存在，则**remove**的操作由实现定义。传统C语言中没有这个函数，而通常提供UNIX特定的**unlink**函数。

rename函数将名称**oldname**变成**newname**，操作成功时返回0，否则返回非0值。**oldname**与**newname**所指的字符串不改变。如果**oldname**文件打开或不存在，或**newname**文
404 件已经存在，则**rename**的操作由实现定义。

15.16 tmpfile、tmpnam、mktemp

语法概要

```
#include <stdio.h>
FILE *tmpfile(void);
char *tmpnam(char *buf);
#define L_tmpnam ...
#define TMP_MAX ...
```

tmpfile函数生成一个新文件并用**fopen**方式**"w+b"**（传统C中为**"w+"**）打开。如果操作成功，则返回新文件的文件指针，否则返回null指针。其目的是只在当前程序执行期间使用新文件。文件关闭或程序终止时，删除这个文件。将数据写入文件之后，编程人员可以用**rewind**函数重定位到文件开头进行读取。

函数**tmpnam**生成不与当前使用的其他文件名发生冲突的新文件名，然后编程人员可以用完全一般化的**fopen**方法打开这个名称的新文件。这样生成的文件不是临时文件，程序终止时并不自动删除这个文件。如果**buf**为**NULL**，则**tmpnam**返回新文件名字符串的指针；后面调用**tmpnam**时可以改变这个字符串。如果**buf**不是**NULL**，则要指向不少于**L_tmpnam**个字符的数组，**tmpnam**把新文件名字符串复制到这个数组中，并返回**buf**。如果**tmpnam**失败，则返回null指针。标准C语言定义**TMP_MAX**值为产生惟一名称的连续**tmpnam**调用次数，至少应为25。

传统C语言函数**mktemp**与**tmpnam**具有相同语法，但**buf**（模板）要指向尾部有6个**X**字母的字符串，其用其他字母或数字覆盖之后，建立惟一文件名。函数**mktemp**返回**buf**值。连续调用**mktemp**时要指定不同模板，保证惟一名称。UNIX实现通常用程序的进程标识代替**XXXXXX**。标准C语言中没有**mktemp**函数。

例　C语言中一个常见的不良编程习惯是

```
ptr = fopen(mktemp("/tmp/abcXXXXXX"),"w+");
```

如果字符串常量不可修改，则这个语句会失败，编程人员还无法引用文件名字符串。下列写法更好，效率也不会更低：

```
char filename[]="/tmp/abcXXXXXX";
ptr = fopen(mktemp(filename),"w+");
```
405

□

第16章 通用函数

本章介绍的函数在头文件**stdlib.h**中声明，可以分为几类：

- 存储分配
- 随机数生成
- 数字转换与整型算术
- 环境通信
- 搜索与排序
- 多字节、宽字符和字符串转换

16.1 malloc、calloc、mlalloc、clalloc、free、cfree

语法概要

```
#include <stdlib.h>
void *malloc(size_t size);
void *calloc(size_t elt_count, size_t elt_size);
void *realloc(void *ptr, size_t size);
void free(void *ptr);
```

malloc函数分配足够大的内存区来存储长度（用**sizeof**运算符计算）为**size**的对象，返回这个区域中第一个元素的指针，并保证对任何数据类型正确对齐。调用者可以用类型转换运算符将这个指针转换成另一指针类型。如果由于某种原因无法进行请求的分配，则返回null指针。如果请求的长度为0，则标准C语言函数返回一个null指针或不能用于访问对象的非null指针。分配的内存不进行任何初始化，因此调用者无法依赖于其内容。由于通过**malloc**分配的区域要保证对任何数据类型正确对齐，因此每个区域实际占用的内存块是对齐长度的位数，通常为4个或8个字节。

例 分配程序的调用者通常将结果指针赋予相应类型的变量。因此，我们假设**T**是某个要动态分配的对象类型，可能是结构、数组或字符。

```
T *NewObject(void)
{
   T *objptr = (T *) malloc(sizeof(T));
   if (objptr==NULL) printf("NewObject: failed!\n");
   return objptr;
}
```

标准C语言中并不严格要求类型转换(**T***)，因为**malloc**返回**void ***类型的指针，在赋值为**objptr**时可以进行隐式转换。在传统C语言中，**malloc**的返回类型为**char ***，隐式转换可能产生警告消息。为了保证C++兼容性，需要进行类型转换。 □

函数**calloc**分配足够大的内存区来存储**elt_count**个元素的数组，每个元素的长度（用

sizeof运算符计算）为**elt_size**。内存区按位清0，返回该区域第一个元素的指针。如果由于某种原因无法进行请求的分配或者**elt_count**或**elt_size**为0，则返回值与**malloc**相同。注意，内存按位清0与浮点数0或null指针的表示方法可能不同。

函数**realloc**所带参数为前面由某个标准函数分配的内存区的指针，在保留其内容的同时改变其长度。如果需要，把内容复制到新的内存区。**realloc**函数返回（可能是新的）内存区的指针。如果无法满足请求，则返回null指针，不影响旧的区域。如果**realloc**的第一个参数为null指针，则函数行为和**malloc**一样。如果**ptr**不是null而**size**为0，则**realloc**返回null指针或不能使用的指针（和**malloc**一样），旧的区域释放。如果新长度比旧长度小，则放弃旧区域末尾的一些旧内容。如果新长度比旧长度大，则所有旧内容保留，在末尾增加新的空间。新的空间不进行任何初始化，调用者要假设其包含无用单元信息。如果**realloc**返回的指针不同于第一个参数，则编程人员要假设旧的内存区释放，不能再用。

例　下面是**realloc**的典型用法，扩展**samples**指针指定的动态数组（这种数组的元素要
408 用下标表达式引用，数组的任何指针可以调用**realloc**失效）。

```
#include <stdlib.h>
#define SAMPLE_INCREMENT 100
int sample_limit = 0; /* Max size of current array */
int sample_count = 0; /* Number of elements in array */
double *samples = NULL; /* Will point to array */

int AddSample( double new_sample )
   /* Add an element to the end of the array */
{
   if (sample_count < sample_limit) {
      samples[sample_count++] = new_sample;
   } else {
      /* Allocate a new, larger array. */
      int new_limit = sample_limit + SAMPLE_INCREMENT;
      double *new_array =
         realloc(samples, new_limit * sizeof(double));
      if (new_array == NULL) {
         /* Can't expand; leave samples untouched. */
         fprintf(stderr,"?AddSample: out of memory\n");
      } else {
         samples = new_array;
         sample_limit = new_limit;
         samples[sample_count++] = new_sample;
      }
   }
   return sample_count;
}
```

□

函数**free**释放前面由**malloc**、**calloc**或**realloc**分配的内存区。函数**free**的参数应为指针，与前面由某个分配函数返回的指针相同。如果参数为null指针，则调用无意义。一块内存区域一经释放，就不能再用于任何用途。使用这个区域中的任何指针（垂悬指针）都会产生无法预测的结果。同样，存储区分配一次而释放多次也会产生无法预测的结果。

在内存受限的独立实现中，编程人员可以用**malloc**和其他函数直接控制可以分配的内存量。这种内存通常称为“堆”。在独立实现环境的许多C语言程序中，从来不用**malloc**函数，因此不需要堆。堆的长度如何指定是由实现定义的。

参考章节 赋值转换 6.3.2

409

传统存储分配函数

传统存储分配函数语法概要

```
char *malloc (unsigned size);
char *mlalloc(unsigned long size);
char *calloc (unsigned elt_count, unsigned elt_size);
char *clalloc(unsigned long elt_count, unsigned long elt_size);
void  free (char *ptr);
void  cfree(char *ptr);
char *realloc (char *ptr, unsigned size);
char *relalloc(char *ptr, unsigned long size);
```

在传统C语言实现中，通常没有声明这些函数的头文件，因此编程人员要声明这些函数。

存储分配函数的**size**参数原始类型为**unsigned int**。由于这个类型太小，无法表示大存储区，因此新版存储分配函数的**size**参数改用类型**unsigned long**。返回类型为**char ***，结果指针应显式转换成对象指针类型。

free的传统版本释放前面由**malloc**、**mlalloc**、**realloc**或**relalloc**分配的内存。而**cfree**函数释放前面由**calloc**或**clalloc**分配的内存。将null指针传递到传统**free**或**cfree**函数在传统实现中由实现定义其行为。

16.2 rand、srand、RAND_MAX

语法概要

```
#include <stdlib.h>
int rand(void);
void srand(unsigned seed);
#define RAND_MAX …
```

连续调用**rand**返回的整数值范围在0到**int**类型可以表示的最大正值之间，是伪随机数生成器的连续结果。在标准C语言中，**rand**范围的上限由**RAND_MAX**指定，至少为32767。

函数**srand**可以初始化调用**rand**时生成连续值的伪随机数生成器。调用**srand**之后，可以连续调用**rand**生成一系列伪随机数。如果用同一参数再次调用**srand**，则此后连续调用**rand**产生相同的一系列伪随机数。用户程序中，在调用**srand**之前连续调用**rand**与用参数1调用**srand**之后连续调用**rand**产生相同的伪随机数系列。

410

标准C语言库函数不会以任何可能影响用户看到的伪随机数系列的方式调用**rand**或**srand**。

16.3 atof、atoi、atol、atoll

语法概要

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

```
double     atof ( const char *str );
int        atoi ( const char *str );
long       atol ( const char *str );
long long atoll( const char *str );  // C99
```

这些函数将字符串**str**的初始部分转换成数字，这在许多UNIX实现中都有。在标准C语言中，它们提供兼容性，但用16.4节介绍的**strtox**函数更好。如果本节介绍的函数无法转换成输入字符串，则其行为是未定义的。

除了错误行为之外，这些函数用下列更一般的形式定义：

```
#include <stdlib.h>

double atof(const char *str) {
   return strtod(str, (char **) NULL);
}

int atoi(const char *str) {
   return (int) strtol(str, (char **) NULL, 10);
}

long atol(const char * str) {
   return strtol(str, (char **) NULL, 10);
}

long long atoll(const char * str) {
   return strtoll(str, (char **) NULL, 10);
}
```

411

16.4 strtod、strtof、strtold、strtol、strtoll、strtoul、strtoull

语法概要

```
#include <stdlib.h>

double strtod(
  const char * restrict str, char ** restrict ptr );
float strtof(
  const char * restrict str, char ** restrict ptr );
long double strtold(
  const char * restrict str, char ** restrict ptr );

long strtol(
  const char * restrict str, char ** restrict ptr, int base );
long long strtoll(
  const char * restrict str, char ** restrict ptr, int base );
unsigned long strtoul(
  const char * restrict str, char ** restrict ptr, int base );
unsigned long long strtoull(
  const char * restrict str, char ** restrict ptr, int base );
```

将字符串转换成数字的转换函数**strtod**与**strtol**来源于System V UNIX，现在已经被标准C语言中采用。**strtoul**函数加进C89中，提供完整性。C99中增加了**strtof**、**strtold**、

strtoll与**strtoull**函数。一般来说，这些函数比相应的**sscanf**函数提供了更好的转换控制。C99还有**strto[u]imax**函数（21.8节）。

对所有这些函数，**str**指向要转换的字符串，**ptr**（如果不是null）指向一个**char ***指针，函数将其设置成指向**str**中已经转换的字符串部分后面的第一个字符。如果**ptr**为null，则将其忽略。如果**str**以空白符开头（见**isspace**函数定义），则先跳过这些空白符再进行转换。

这些函数还有宽字符版本（见24.4和21.9节）。

浮点数转换 浮点数转换函数**strtod**、**strtof**与**strtold**希望要进行转换的数字除了包括可选的正负号以外，其后所跟的内容应符合下列内容之一：

1. 一系列十进制数，可能包含小数点及其后的如2.7.2节定义的可选指数部分。
2. 字符**0x**或**0X**加非空十六进制数系列，加如2.7.2节定义的可选二进制指数部分。
3. 字符串**INF**或**INFINITY**，忽略大小写。
4. 字符串**NAN**或**NAN(...)**，忽略大小写，省略号可以是任何字母、数字或下划线序列。

匹配这些模型之一的最长字符序列转换成一个浮点数，然后返回。返回类型取决于选择的函数。希望数字的格式不同于C语言自己的浮点数常量语法（2.7.2节），可能出现可选的正负号，不 412
需要小数点，小数点不一定是点号（取决于区域设置），不一定要有浮点数后缀（**f**、**F**、**l**或**L**）。

如果由于字符串不匹配希望的数字模型（或是空的）而无法进行转换，则返回0，***ptr**设置为**str**的值，**errno**设置为**ERANGE**。如果数字转换可能造成上溢，则返回**HUGE_VAL**、**HUGE_VALF**或**HUGE_VALL**（包括正确符号）。如果数字转换可能造成下溢，则返回0。对于上溢和下溢，都把**errno**设置为**ERANGE**。根据这个定义，无效数字与造成下溢的值无法区别，但可以用***ptr**中设置的值区分开来。一些传统实现在字符串不匹配希望的数字模型时将**errno**设置为**EDOM**。

C99中新增加了用**strtod**转换十六进制浮点数、无限大和NaN的功能。字符串**INF**与**INFINITY**解释为无限大。如果无限大无法在返回类型中表示，则把这些输入看成造成上溢。字符串**NAN**或**NAN(...)**表示NaN。如果NaN无法在返回类型中表示，则把这些输入看成无法转换。

如果区域设置不是“C”，则可以接受其他浮点数输入格式。

整数转换 整数转换函数**strtol**、**strtoll**、**strtoul**与**strtoull**将参数字符串的初始部分分别转换成**long int**、**long long int**、**unsigned long int**或**unsigned long long int**类型的整数。希望的数字格式随希望的基数（**base**）而改变，所有情况下都相同，可以包括可选的正负号。不能用整数后缀(**l**、**L**、**u**或**U**)。

如果**base**为0，则可选符号之后的数字格式应为十进制常量、八进制常量或十六进制常量。数字的基数可以从格式得到。如果**base**为2到36之间，则数值应为表示一个指定进制整数的字母与数字的非0序列。字母**a**到**z**（或**A**到**Z**）分别表示数值10到35。只能使用那些用于表示比**base**小的值的字母。作为特例，如果**base**为16，则数字可以在任何符号之后以**0x**或**0X**开头（将其忽略）。

如果无法进行转换，则返回0，***ptr**设置为**str**的值，**errno**设置为**ERANGE**。如果数字转换可能造成溢出，则返回**LONG_MAX**、**LONG_MIN**、**LLONG_MAX**、**LLONG_MIN**、**ULONG_MAX**或**ULLONG_MAX**。（取决于函数返回类型和数值符号），**errno**设置为**ERANGE**。

如果区域设置不是“C”，则可以接受其他整数输入格式。

参考章节 十进制常量 2.7；**errno** 11.2；浮点数常量 2.7；十六进制常量 2.7；

HUGE_VAL　第17章；整数常量　2.7；**isspace**函数　12.6；**LONG_MAX**、**LONG_MIN**、
413 **ULONG_MAX**　5.1.1；NaN　5.2；八进制常量　2.7；类型标志　2.7

16.5 abort、atexit、exit、_Exit、EXIT_FAILURE、EXIT_SUCCESS

语法概要

```
#include <stdlib.h>
#define EXIT_FAILURE ...
#define EXIT_SUCCESS ...
void exit (int status);
void _Exit(int status);                    // C99
void abort(void);
int atexit(void (*func)(void));
```

exit、**_Exit**与**abort**函数使程序终止，控制并不返回这些函数的调用者。

函数**exit**正常终止程序，并进行下列清理操作：

1.（仅对标准C语言）所有向**atexit**函数注册的函数按与注册相反的顺序调用，注册几次就调用几次。

2. 刷新打开的输出流，关闭所有打开的数据流。

3. 删除**tmpfile**函数生成的文件。

4. 控制返回宿主环境，提供状态值。

按照许多系统中的习惯，**status**值为0表示终止程序成功，用非0值表示各种异常终止。标准C语言中数值0和宏**EXIT_SUCCESS**的值表示终止成功，宏**EXIT_FAILURE**的值表示终止不成功，其他值的含义是由实现定义的。从函数**main**返回一个整数值相当于用这个值调用**exit**函数。

函数**_Exit**与**exit**函数不同之处在于既不调用**atexit**注册的退出处理器，也不调用**signal**注册的信号处理器。是否进行其他清理操作是由实现定义的，如关闭所有打开的数据流。**_Exit**是C99增加的，传统上有些实现用名为**_exit**的函数提供类似功能。

abort函数使程序异常终止，不调用向**atexit**注册的函数。**abort**是否引起清理操作是由实现定义的，向宿主系统返回的状态值也是由实现定义的，但应表示为“不成功”。在标准C语言和许多传统实现中，调用**abort**转换成可以捕获的特殊信号（标准C语言中为**SIGABRT**）。如果信号被忽略或处理器返回，则标准C语言实现仍然终止程序，而其他实现可能使**abort**函数返回调用者。断言失败（见19.1节）也会调用**abort**。

atexit函数是标准C语言中增加的，它注册一个函数，使得调用**exit**时或函数**main**返回时会调用这个函数。程序异常终止时（如用**abort**或**raise**终止），不调用注册的函数。实现应允许至少注册32个函数。如果注册成功，则**atexit**函数返回0，否则返回非0值。函数无法注销。
414 所有向**atexit**函数注册的函数按与注册相反的顺序调用，然后再由**exit**函数完成所有标准清理操作。每个函数不带参数调用，应具有返回类型**void**。注册函数不能引用任何不是自己定义的存储类为**auto**或**register**的对象（例如通过指针引用）。函数注册几次就会在此时调用几次。一些传统C语言实现用**onexit**实现类似功能。

例 本例中**main**函数打开一个文件，然后注册**cleanup**函数，在调用**exit**时关闭文件（事实上，**exit**关闭所有文件，但也许编程人员要先关闭这个文件）。

```
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
FILE *Open_File;

void cleanup(void) {
    if (Open_File != NULL) fclose(Open_File);
}

int main(void)
{
    int status;
    ...
    Open_File = fopen("out.dat","w");
    status = atexit(cleanup);
    assert(status == 0);
    ...
}
```

□

参考章节 **assert** 19.1; **fflush** 15.2; **atexit** 19.5; **main**函数 9.9; **raise** 19.6; **return**语句 8.9; **signal** 19.6; **tmpfile** 15.16; **void**类型 5.9

16.6 getenv

语法概要

```
#include <stdlib.h>
char * getenv( const char *name );
```

getenv函数的惟一参数是一个字符串的指针，按某种实现定义的方式解释作为执行环境认识的名称。函数返回另一个字符串的指针，是参数**name**的值。如果所指的名称没有值，则返回null指针。编程人员不能修改返回的字符串，但后续调用**getenv**可以将其覆盖。 415

在传统C语言中，可以用 **main**的第三个非标准参数**env**向**main**函数提供 (*name*,*value*)绑定集（见9.9节）。通常会有一个**setenv**函数或**getenv** 可以设置环境变量。

16.7 system

语法概要

```
#include <stdlib.h>
int system( const char *command );
```

函数**system**将字符串参数传递到操作系统的命令处理器（或shell）中，按实现定义的方式执行。**system**的行为和返回值是由实现定义的，但返回值通常是这个命令的完成状态。在标准C语言中，**system**可以用null参数调用。这时如果实现没有提供命令处理器，则返回0，如果实现提供命令处理器，则返回非0值。

exec系列函数

传统C语言语法概要

```
execl (char *name, char *argi, ..., NULL);
execlp(char *name, char *argi, ..., NULL);
execle(char *name, char *argi, ..., NULL, char *envp[]);
execv (char *name, char *argv[]);
execvp(char *name, char *argv[]);
execve(char *name, char *argv[], char *envp[]);
```

各种**exec**不属于标准C语言，而主要在UNIX系统中提供。所有情况下，它们执行文件**name**中的程序，把当前进程变成新进程，各个函数的差别在于新进程的参数如何提供：

1. 函数**execl**、**execlp**与**execlp**带有可变个数参数，最后一个是null指针。按照规则，第一个参数与**name**相同，即应为要执行的程序名。
2. 函数**execv**、**execvp**与**execve**以指向以null终止的向量的指针作为参数，像提供给**main**函数的参数一样。按照规则，**argv[0]**与**name**相同，即应为要执行的程序名。
3. 函数**execle**与**execve**还向新进程传递显式“环境”。参数**envp**是以null终止的字符串指针向量。每个字符串的形式为**"name=value"**（在其他**exec**版本中，调用进程的环境指针隐式传递到新进程）。

416

4. 函数**execlp**与**execvp**分别和**execl**与**execv**相似，只是运行**execlp**和**execvp**时，系统在通常包含命令的目录集中寻找文件（通常是环境变量**path**或**PATH**的值）。

启动新进程时，向**exec**提供的参数提供给新进程的**main**函数（9.9节）。

16.8 bsearch、qsort

语法概要

```
#include <stdlib.h>
void *bsearch(
  const void *key,
  const void *base,
  size_t count,
  size_t size,
  int (*compar)(const void *the_key, const void *a_value));
void qsort(
  void *base,
  size_t count,
  size_t size,
  int (*compar)(const void *element1, const void *element2) );
```

函数**bsearch**搜索**count**个元素的数组，其第一个元素是**base**所指的元素。字符中每个元素的长度是**size**。**compar**函数的参数是指向关键字的指针和指向一个数组元素的指针，返回的负值、0和正值分别表示关键字小于、等于和大于这个元素。搜索开始时，数组要按升序排序（根据**compar**）。**bsearch**返回数组中与**key**匹配的元素的指针，如果找不到这样的元素，则返回null指针。

函数**qsort**排序**count**个元素的数组，其第一个元素是**base**所指的元素。字符中每个元素

的长度是**size**。**compar**函数的参数是分别指向两个数组元素的指针，返回的－1、0和1分别表示第一个元素小于、等于和大于第二个元素。排序结束时，数组要按升序排序（根据**compar**）。

这些函数中，在每次调用**compar**之前和之后都有一个序列点。

例 下列函数**fetch**用**bsearch**函数搜索**Table**，这是个排序的结构数组。函数**key_compare**测试关键字的值。注意，**fetch**首先把关键字嵌入一个虚参(**key_elem**)，使 417
bsearch与**qsort**都可以使用**key_compare**（20.6节）：

```
#include <stdlib.h>
#define COUNT 100
struct elem {int key; int data; } Table[COUNT];

int key_compare(const void * e1, const void * e2)
{
    int v1 = ((struct elem *)e1)->key;
    int v2 = ((struct elem *)e2)->key;
    return (v1<v2) ? -1 : (v1>v2) ? 1 : 0;
}

int fetch(int key)
/* Return the data item associated with key in
  the table, or 0 if no such key exists. */
{
    struct elem *result;
    struct elem key_elem;
    key_elem.key = key;
    result = (struct elem *)
           bsearch(
               (void *) &key_elem, (void *) &Table[0],
               (size_t) COUNT, sizeof(struct elem),
               key_compare);
    if (result == NULL)
        return 0;
    else
        return result->data;
}
```

□

例 下列函数**sort_table**用**qsort**排序上例中的表，也用**key_compare**函数比较表中元素：

```
void sort_table(void)
/* Sorts Table according to the key values */
{
    qsort(
        (void *)Table,
        (size_t) COUNT,
        sizeof(struct elem),
        key_compare );
}
```

□

传统C语言中的bsearch与qsort

传统C语言中**bsearch**与**qsort**的语法如下： 418

```
char *bsearch(
   char *key,
```

```
    char *base,
    unsigned count,
    int size,
    int (*compar)(
       char *the_key,
       char *a_value));

void qsort(
    char *base,
    unsigned count,
    int size,
    int (*compar)(
       char *element1,
       char *element2));
```

16.9 abs、labs、llabs、div、ldiv、lldiv

语法概要

```
#include <stdlib.h>

int          abs(int           x);
long        labs(long int      x);
long long llabs(long long int x);          // C99
typedef …   div_t;
typedef …  ldiv_t;
typedef … lldiv_t;                         // C99
div_t     div(int       n, int       d);
ldiv_t   ldiv(long      n, long      d);
lldiv_t lldiv(long long n, long long d);   // C99
```

本节的函数是标准C语言**stdlib.h**中和传统C语言**math.h**中定义的整型算术函数。函数**abs**、**labs**和（C99中的）**llabs**返回参数的绝对值，差别在于参数类型与返回值类型各不相同。浮点数版本由**math.h**中的**fabs**函数提供，最大长度的整型版本由**inttypes.h**（参见21.7节）中的**imaxabs**提供。绝对值函数很容易实现，一些编译器把它们当作内部函数，这在标准C语言中是允许的。

3个除法函数**div**、**ldiv**与(C99中的)**lldiv**都同时计算**n**除以**d**的商和余数，差别在于参数类型与返回值类型各不相同。类型**div_t**、**ldiv_t**与（C99中的）**lldiv_t**是包含两个成员**quot**与**rem**（顺序未指定）的结构，成员类型分别为**int**、**long int**与**long long int**。返回的商**quot**等于**n/d**，余数**rem**等于**n%d**。**d**为0时或商与余数之一无法用返回类型表示时，函数的行为在其最有效的实现中也是未定义的（不一定是定义域错误）。最大长度的整型版本由**inttypes.h**中的**imaxdiv**提供。

419

之所以提供除法函数，是因为大多数计算机能够同时计算商与余数，因此使用这个函数（可以内联扩展）比分别使用**/**和**%**更快。

参考章节　**fabs**　17.2；**imaxabs**　21.7；**imaxdiv**　21.7

16.10 mblen、mbtowc、wctomb

语法概要

```
#include <stdlib.h>
typedef ... wchar_t;
#define MB_CUR_MAX ...
int mblen(const char *s, size_t n);
int mbtowc(wchar_t *pwc, const char *s, size_t n);
int wctomb(char *s, wchar_t wchar);
```

标准C语言可以处理扩展的特定区域设置的字符集，这些字符集非常大，无法在一个**char**类型对象中表示每个字符。对于这种字符集，标准C语言提供了内部和外部表示模式。在内部，扩展字符假设为适合宽字符，这些宽字符是由实现定义的整型类型**wchar_t**的对象。扩展字符的字符串（宽字符串）可以表示为**wchar_t[]**类型的对象。在外部，一个宽字符可以表示为正常字符的序列——对应于宽字符的多字节字符。2.1.5节介绍了多字节字符与宽字符，2.9节介绍了字符集与编码方式。

本节的字符转换函数在C89增补1中得到加强，增加了新的可重新启动的函数，包括**mbrlen**、**btowc**、**wctob**、**mbrtowc**与**wcrtomb**。新函数更加灵活，其行为说明更完整。它们在**wchar.h**中定义，我们在24.2节介绍这些函数。

16.10.1 编码方式与转换状态

本节介绍多字节字符与宽字符之间转换的一些特征。这些术语适合本章介绍的许多函数。

多字节字符与宽字符没有强制或排除某种表示方法，但正常字符序列和多字节字符序列都要用一个null字符**'\0'**作为终止符。多字节编码方式一般是状态相关的，用shift字符序列改变后续字符的含义。

420

本章中原始的标准C语言函数保留最近处理多字节字符时的内部转换状态信息。而C89增补1增加的新函数则提供显式类型**mbstate_t**，保存转换状态信息，使几个字符串可以并行处理。但是，如果新状态参数为null，则每个函数都使用自己的内部状态。其他标准库函数调用不允许影响这些内部shift状态。

当前区域设置中表示多字节字符的最大字节数由（非常量）表达式**MB_CUR_MAX**指定。大多数以多字节字符的指针**s**作为参数的函数还带一个整数参数**n**，指定**s**的最大字节数。无论如何，**n**不能大于**MB_CUR_MAX**，但可以更小以限制转换。

指定当前转换状态、多字节字符的指针**s**和长度**n**之后，有几种可能的情况：

1. **s**的前**n**个或更少字节可以组成一个有效多字节字符，对应于一个宽字符**wc**。转换状态要相应更新。如果**wc**刚好是个null宽字符，则我们说**s**生成null宽字符。
2. **s**中所有**n**个字节可以组成一个有效多字节字符的开头，但本身并不完整，无法计算相应的宽字符。这时**s**称为不完整多字节字符（如果**n**至少是**MB_CUR_MAX**，则**s**包含冗余shift字符时可能发生这种情形）。
3. **s**中的**n**字节可以组成一个无效多字节字符，即无法按当前编码方式建立有效或不完整的多字节字符。

改变区域设置的**LC_CTYPE**类别（见11.5节）可能改变字符编码方式，使shift状态不确定。**MB_CUR_MAX**值包括shift字符的足够空间。

参考章节 **mbstate_t** 11.1

16.10.2 长度函数

mblen函数检查**s**所指定字符串中最多**n**个字节，看这些字符是否表示与当前shift状态相对的有效多字节字符。如果是，则返回构成多字节字符的字节数，如果**s**无效或不完整，则返回 -1。如果**s**是null指针，则**mblen**返回一个非0值（如果多字节字符特定区域设置的编码方式是状态相关的），作为副作用，这个调用将任何内部状态重置为预定义的初始条件。

16.10.3 转换成宽字符

mbtowc函数根据内部转换状态将多字节字符**s**转换成宽字符，如果**pwc**不是null指针，则结
421 果存放在**pwc**指定的对象中。返回值是构成多字节字符的字节数，如果**s**无效或不完整，则返回 -1。如果**s**是null指针，则**mbtowc**返回一个非0值（如果多字节字符特定区域设置的编码方式是状态相关的），作为副作用，这个调用将任何内部状态重置为预定义的初始条件。

例　下面是用**mbtowc**函数实现的**mbstowcs**（16.11节）：

```
#include <stdlib.h>
size_t mbstowcs(wchar_t *pwcs, const char *pmbs, size_t n)
{
    size_t i = 0; /* index into output array */
    (void) mbtowc(NULL,NULL,0); /* Initial shift state */
    while (*pmbs && i < n) {
        int len = mbtowc(&pwcs[i++],pmbs,MB_CUR_MAX);
        if (len == -1) return (size_t) -1;
        pmbs += len;    /* to next multibyte character */
    }
    return i;
}
```

□

参考章节　**mbstate_t**　11.1；多字节字符　2.1.5；**size_t**　11.1；**NEOF**　11.1

16.10.4 转换宽字符

wctomb函数根据内部转换状态将宽字符**wc**转换成多字节字符，结果存放在**s**指定的字符数组中，至少应为**MB_CUR_MAX**个字符，并更新转换状态，不添加null字符。如果**wc**是有效字符编码，则返回**s**中存放的字符数，否则返回 -1。如果**s**是null指针，则**wctomb**返回一个非0值（如果多字节字符特定区域设置的编码方式是状态相关的），作为副作用，这个调用将任何内部状态重置为预定义的初始条件。

16.11 mbstowcs、wcstombs

语法概要

```
#include <stdlib.h>
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

本节介绍的标准C语言函数在宽字符串与多字节字符序列之间相互转换。这些函数的可重新
422 启动版本**mbsrtowcs**与**wcsrtombs**是C89增补1增加的，在**wchar.h**中定义，见24.3节。

16.11.1 转换成宽字符串

函数**mbstowcs**将以null终止的字符串**s**中的多字节字符转换成相应的宽字符序列，将结果存放在**pwcs**指定的数组中。**s**中的多字节字符要以初始shift状态开始，以null字符终止。到null终止

字符为止的每个多字节字符像调用**mbtowc**函数一样进行转换。宽字符数组中存放**n**个元素后、到达**s**结尾时（这时在输出中存储一个null宽字符）或发生转换错误时，转换停止。函数返回存放的宽字符个数（不包括null终止字符，如有），如果发生转换错误则返回 -1（转换成**size_t**）。

pwcs输出指针可能是null指针，这时不存储输出的宽字符，长度参数**n**被忽略。

如果已经向**pwcs**中写入**n**个输出的宽字符（**pwcs**不是null指针），则输入多字节字符串转换在null终止字符之前停止。这时**src**所指的指针设置为指向最后转换的多字节字符后面。转换状态更新（不一定是初始状态）并返回**n**。

如果发生转换错误，也会提前终止输入多字节字符串转换。这时**src**所指的指针设置为指向转换时发生错误的多字节字符。函数返回 -1（转换成**size_t**），**errno**中存储**EILSEQ**，转换状态为不确定。

16.11.2 转换宽字符串

函数**wcstombs**将以**pwcs**指定的值开头的宽字符序列转换成相应的多字节字符，将结果存放在**s**指定的字符数组中。每个宽字符像调用**wctomb**函数一样进行转换。输入宽字符序列要以null宽字符终止。输出多字节字符序列以初始shift状态开始。如果已经向**s**写入**n**个字符、遇到**pwcs**末尾（这时在输出中存储一个null字符）或发生转换错误时，转换停止。函数返回写入**s**的字符数（不包括null终止字符，如有），如果发生转换错误则返回 -1（转换成**size_t**）。

s输出指针可能是null指针，这时不存储输出的字节，长度参数**n**被忽略。

如果已经向**s**中写入**n**个输出字节（**s**不是null指针），则输入宽字符串转换在null终止字符之前停止。这时**src**所指的指针设置为指向最后转换的宽字符后面。转换状态更新（不一定是初始状态）并返回**n**。

如果发生转换错误，也会提前终止输入多字节字符串转换。这时**src**所指的指针更新为指向转换时发生错误的宽字符。函数返回 -1（转换成**size_t**），**errno**中存储**EILSEQ**，转换状态为不确定。 423

例 下列语句读取多字节字符串（**mbs**），将其转换成宽字符串（**wcs**），然后转换回多字节字符串（**mbs2**）。我们认为，如果转换函数完全填满目标数组，则会发生错误，因为这时转换的字符串不是以null终止的：

```
#include <stdlib.h>
#include <stdio.h>
#define MAX_WCS 100
#define MAX_MBS (100*MB_CUR_MAX)
wchar_t wcs[MAX_WCS+1];
char mbs[MAX_MBS], mbs2[MAX_MBS];
size_t len_wcs, len_mbs;

/* Read in multibyte string; check for error */
if (!fgets(mbs, MAX_MBS, stdin))
    abort();

/* Convert to wide character string; check for error */
len_wcs = mbstowcs(wcs, mbs, MAX_WCS);
if (len_wcs == MAX_WCS || len_wcs == (size_t)-1)
    abort();

/* Convert back to multibyte string; check for error */
len_mbs = wcstombs(mbs2, wcs, MAX_MBS);
if (len_mbs == MAX_MBS || len_mbs == (size_t)-1)
    abort();
```

□

参考章节 转换状态 2.1.5；多字节字符串 2.1.5；宽字符 2.1.5 424

第17章 数学函数

本章介绍的函数在库头文件**math.h**中声明。在标准C语言中，**stdlib.h**中还包含几个数学函数，而C99用**complex.h**声明复数数学函数。

下面介绍本章数学函数的一般规则：

参数类型 在C99之前，所有浮点数的C语言库运算只对**double**类型参数定义。即使使用**float**类型，这也是可行的，因为**float**参数会在调用之前自动转换为**double**参数。C99还对**float**与**long double**类型参数定义了一组并行数学函数，分别在原函数名后面加上后缀**f**和**l**。

每种浮点数参数用不同数学函数名使编程人员可以更好地控制性能与类型转换，但会减少程序移植性。例如，将变量类型从**double**变成**long double**之后，就要编辑许多函数名，否则会不知不觉遇到精度问题，因为**long double**参数要根据**double**函数原型转换成**double**类型。因此，C99在头文件**tgmath.h**（17.12节）中定义了通用类型宏。这些宏与原始的**double**类型库函数同名，根据参数类型调用相应同名函数，就像内置的加法与乘法运算符进行的操作一样。如果需要访问原函数，编程人员可以用**#undef**取消这些宏的定义（或不包括**tgmath.h**）。这些宏要加进C99实现中，因为无法用C语言编写通用类型宏。

错误处理 数学函数可以进行两种一般化错误处理，但早期的C语言实现可能不按一致方法处理。如果输入参数位于定义函数的定义域之外，或参数具有无限大和NaN之类特殊值，则发生定义域错误。这时**errno**（见11.2节）设置为数值**EDOM**，函数返回一个由实现定义的值，传统的错误返回值为0，但有些实现可能有更好的选择，如特殊的“非数字”值。

如果函数结果无法表示为函数返回类型的值，则发生范围错误。这时**errno**设置为**ERANGE**，函数应返回与正确结果具有相同符号的最大可表示浮点数值。在C89中，这是宏**HUGE_VAL**的值；C99中提供了**HUGE_VALF**与**HUGE_VALL**宏。C99可以灵活地控制哪种情形表示错误，应该停止；哪种情形是遇到无限大或NaN，可以继续。

如果函数结果太小，无法表示，则函数返回0，**errno**是否设置为**ERANGE**由实现确定。

17.1 abs、labs、llabs、div、ldiv、lldiv

这些函数在**stdlib.h**中定义（见16.9节）。

17.2 fabs

语法概要

```
#include <math.h>

double      fabs (double x);
float       fabsf(float x); // C99
long double fabsl(long double x);   // C99
```

fabs函数返回参数的绝对值。整型绝对值函数(**abs**、**labs**与**llabs**)在**stdlib.h**中定义。

参考章节 abs、labs、llabs 16.9；通用类型宏 17.12 426

17.3 ceil、floor、lrint、llrint、lround、llround、nearbyint、round、rint、trunc

语法概要

```
#include <math.h>            // All new to C99 except ceil, floor

double      ceil  (double x);
float       ceilf (float x);
long double ceill (long double x);

double      floor  (double x);
float       floorf (float x);
long double floorl (long double x);

double      nearbyint  (double x);
float       nearbyintf (float x);
long double nearbyintl (long double x);

double      rint  (double x);
float       rintf (float x);
long double rintl (long double x);

long int lrint (double x);
long int lrintf (float x);
long int lrintl (long double x);

long long int llrint  (double x);
long long int llrintf (float x);
long long int llrintl (long double x);

double      round  (double x);
float       roundf (float x);
long double roundl (long double x);

long int lround (double x);
long int lroundf (float x);
long int lroundl (long double x);

long long int llround  (double x);
long long int llroundf (float x);
long long int llroundl (long double x);

double      trunc  (double x);
float       truncf (float x);
long double truncl (long double x);
```

所有这些函数计算与浮点数参数接近的整数。尽管许多函数返回的值是整数，但其返回类型都设为浮点型，因为得到的整数可能太大，无法用整型表示。本节除**ceil**与**floor**以外的所有函数都是C99新增加的，都具有通用类型宏。具有浮点数返回类型的函数在参数为无限大时返回具有正确符号的无限大。

427

- **ceil**函数返回不小于**x**的最小整数。
- **floor**函数返回不大于**x**的最大整数。
- **round**函数返回最接近**x**的整数，如果**x**在两个整数中点上，则返回绝对值较大的整数（即远离0）。
- **trunc**函数返回最接近**x**的整数（靠近0方向），对正数为**floor(x)**，对负数为**ceil(x)**。

- **nearbyint**函数返回最接近**x**的整数，根据当前舍入方向（见**fenv.h**）。
- **lrint**与**llrint**函数与**nearbyint**相同，即返回整数类型的舍入值。如果舍入值无法表示为这个整型类型，则结果未定义。
- **rint**函数与**nearbyint**相同，但如果结果值不同于参数（即参数不是整数），则会发生“不精确”浮点数异常。

参考章节 舍入方向 22.4；通用类型宏 17.12

17.4 fmod、remainder、remquo

语法概要

```
#include <math.h>   // All new to C99 except fmod

double      fmod  (double x,      double y);
float       fmodf (float x, float y);
long double fmodl (long double x, long double y);

double      remainder  (double x, double y);
float       remainderf (float x, float y);
long double remainderl (long double x, long double y);

double      remquo (double x, double y, int *quo); C99
float       remquof(float x, float y, int *quo); C99
long double remquol(long double x, long double y, int *quo);
```

这些函数返回**x/y**的浮点数余数的近似值，即对于某一整数n的$r=x-n*y$的r近似值。差别在于n的选择，但所有情况下r的绝对值小于y的绝对值。除**fmod**以外的所有函数都是C99新增加的，都具有通用类型宏。

- **fmod**函数选择n为**trunc**(x/y)，即r与x符号相同。
- **remainder**与**remquo**函数选择n为**round**(x/y)，但如果x/y在两个整数中点上，则选择偶数整数。r与x符号可能不同。

428

remquo函数返回与**remainder**函数相同的值。此外，它们在***quo**中存储一个值，符号与x/y相同，大小为2^k与x/y整除商的模数。k是实现定义整数，大于或等于3。即***quo**设置为x/y整除商的低顺序位。这在某些参数简化计算中有用，不在C语言库的作用域之内。

如果y为0，则符合标准C语言的实现可能产生定义域错误或从这些函数返回0。在一些早期的C语言实现中，这种情况会返回x。尽管余数在数学上用x/y定义，但定义余数时x/y的值不一定要可表示。

函数**fmod**不要与函数**modf**（17.5节）混淆，后者用于取得浮点数的整数和小数部分。

参考章节 **round** 17.4；**trunc** 17.4；通用类型宏 17.12

17.5 frexp、ldexp、modf、scalbn

语法概要

```
#include <math.h>   // All new to C99 except frexp

double      frexp (double x, int *nptr);
float       frexpf(float x, int *nptr);
long double frexpl(long double x, int *nptr);

double      ldexp (double x, int n);
```

```
float       ldexpf(float x, int n);
long double ldexpl(long double x, int n);

double      modf (double x, double *nptr);
float       modff(float x, float *nptr);
long double modfl(long double x, long double *nptr);

double      scalbn (double x, int n);
float       scalbnf(float x, int n);
long double scalbnl(long double x, int n);

double      scalbln (double x, long int n);
float       scalblnf(float x, long int n);
long double scalblnl(long double x, long int n);
```

本节介绍的大部分函数都是C99新增加的，都具有通用类型宏。

frexp函数将浮点数**x**分为小数f和指数n，使f为0.0或$0.5 \le |f| < 1.0$，$f*2^n$等于**x**。返回小数部分f，并把指数n存放在**nptr**所指的位置。如果**x**为0，则两个返回值都是0。如果**x**不是浮点数，则结果未定义。

frexp函数的逆是**ldexp**函数，计算$x*2^n$的值，可能发生范围错误。 429

modf函数将浮点数**x**分为小数部分f和整数部分n，使$|f| < 1.0$而$f+n=$**x**。f和n与**x**具有相同符号。返回小数部分f，并把整数部分n存放在**nptr**所指的位置。名称**modf**是个助记符，其计算的值称为余数。函数**modf**不要与函数**fmod**（17.3节）混淆，后者计算一个浮点数除以另一个浮点数的余数。一些早期的C语言实现用不同方法定义**modf**，可以查看一下本地库文档。C99中，**modf**没有通用类型宏。

scalbn与**scalbln**函数将浮点数**x**乘以b^n，其中b是**FLT_RADIX**。这个计算比实际计算b^n并将浮点数**x**乘以b^n更有效。可能发生范围错误。

17.6 exp、exp2、expm1、ilogb、log、log10、log1p、log2、logb

语法概要

```
#include <math.h>     // All new in C99 except exp, log, log10

double      exp (double x);
float       expf(float x);
long double expf(long double x);

double      exp2 (double x);
float       exp2f(float x);
long double exp2l(long double x);

double      expm1 (double x);
float       expm1f(float x);
long double expm1l(long double x);

double      log (double x);
float       logf(float x);
long double logl(long double x);

double      log10 (double x);
float       log10f(float x);
long double log10l(long double x);

double      log1p (double x);
float       log1pf(float x);
```

```
long double log1pl(long double x);

double      log2 (double x);
float       log2f(float x);
long double log2l(long double x);

int ilogb (double x);
int ilogbf(float x);
int ilogbl(long double x);
```

430

本节介绍的大部分函数都是C99新增加的，都具有通用类型宏。

exp函数计算e^x，其中e是自然对数的底数。**exp2**函数计算2^x。**expm1**函数计算e^{x-1}（如果**x**很小，则**expm1(x)**比**exp(x)-1**）更精确。在所有情况下，大参数可能发生范围错误。C99之前只有**exp**函数。

log函数计算**x**的自然对数函数。**log10**函数计算底数为10的对数，**log2**函数计算底数为2的对数。如果**x**为负数，则发生定义域错误。如果**x**为0或接近0，则可能发生范围错误（接近$-\infty$）或返回数值$-\infty$而不发生错误。一些早期C语言实现把0当作定义域错误，并且可能把**log**函数称为**ln**。C99之前只有**log**与**log10**函数。

logb与**ilogb**函数从浮点数参数**x**的表示中取出指数。前面曾介绍过，字母**b**是标准模型中浮点数表示的基数，可以作为**float.h**中的**FLT_RADIX**。参数**x**不需要规格化。**logb**函数返回浮点数类型的（整数）指针，如果**x**为0，则可能发生定义域错误。**ilogb**函数返回整数指针，相当于把**logb**结果转换成**int**类型，只是有下列例外：如果**x**为0，则**ilogb**返回**FP_ILOGB**；如果**x**为∞或$-\infty$，则**ilogb**返回**INT_MAX**；如果**x**为NaN，则**ilogb**返回**FP_ILOGBNAN**。

431

参考章节　浮点数模型　5.2；**FLT_RADIX**　5.2；通用类型宏　17.12

17.7 cbrt、fma、hypot、pow、sqrt

语法概要

```
#include <math.h>       // All new in C99 except pow, sqrt

double      cbrt (double x);
float       cbrtf(float x);
long double cbrtl(long double x);

double      hypot (double x, double y);
float       hypotf(float x, float y);
long double hypotl(long double x, long double y);

double      fma (double x, double y, double z);
float       fmaf(float x, float y, float z);
long double fmal(long double x, long double y, long double z);

double      pow( double x, double y);
float       powf(float x, float y);
long double powl(long double x, long double y);

double      sqrt (double x);
float       sqrtf(float x);
long double sqrtl(long double x);
```

函数**pow**计算x^y。如果**x**为非0值而**y**为0，则结果为1.0。如果**x**为0而**y**为正值，则结果为0。如果**x**为负数而**y**不是整数或**x**为0而**y**为非正值，则可能发生定义域错误。函数**pow**也可能发生范围错误。

hypot函数计算x^2+y^2的平方根。C99要求其计算结果不出现计算sqrt(x*x+y*y)时产生的上溢和下溢。

fma函数计算(x*y)+z，好像是用无限精度计算，然后一次性将最后结果舍入为返回类型。

sqrt函数计算**x**的非负平方根。**x**为负数时可能发生定义域错误。

cbrt函数计算**x**的立方根。

参考章节 通用类型宏 17.12

17.8 rand、srand、RAND_MAX

这些函数在**stdlib.h**中定义（见16.2节）。 432

17.9 cos、sin、tan、cosh、sinh、tanh

语法概要

```
#include <math.h>

double      cos (double x);
float       cosf(float x); // C99
long double cosl(long double x);    // C99

double      sin (double x);
float       sinf(float x); // C99
long double sinl(long double x);    // C99

double      tan (double x);
float       tanf(float x); // C99
long double tanl(long double x);    // C99

double      cosh (double x);
float       coshf(float x); // C99
long double coshl(long double x);   // C99

double      sinh (double x);
float       sinhf(float x); // C99
long double sinhl(long double x);   // C99

double      tanh (double x);
float       tanhf(float x); // C99
long double tanhl(long double x);   // C99
```

cos函数计算**x**的三角余弦函数，**x**的值为弧度。不可能发生定义域错误和范围错误，但编程人员要注意结果对太大的**x**值没什么意义。

sin与**tan**函数计算**x**的三角正弦和正切函数，如果参数值接近π/2的奇数倍，则可能发生范围错误。**sin**与**tan**函数结果对太大的x值也没什么意义。

cosh、**sinh**与**tanh**函数计算**x**的双曲余弦、双曲正弦和双曲正切，如果**sinh**与**cosh**的参数绝对值很大，则可能发生范围错误。

参考章节 通用类型宏 17.12 433

17.10 acos、asin、atan、atan2、acosh、asinh、atanh

语法概要

```
#include <math.h>   // New in C99 except acos, asin, atan, atan2
```

```
double      acos (double x);
float       acosf(float x);
long double acosl(long double x);

double      asin (double x);
float       asinf(float x);
long double asinl(long double x);

double      atan (double x);
float       atanf(float x);
long double atanl(long double x);

double      atan2(double y, double x);
float       atan2f(float y, float x);
long double atan2l(long double y, long double x);

double      acosh (double x);
float       acoshf(float x);
long double acoshl(long double x);

double      asinh (double x);
float       asinhf(float x);
long double asinhl(long double x);

double      atanh (double x);
float       atanhf(float x);
long double atanhl(long double x);
```

acos函数计算**x**的三角反余弦函数主值，结果为弧度，在0到π之间（由于舍入误差的影响，这些函数的范围是近似的）。如果参数小于－1.0或大于1.0，则发生定义域错误。

asin函数计算**x**的三角反正弦函数主值，结果为弧度，在$-\pi/2$与$\pi/2$之间。如果参数小于－1.0或大于1.0，则发生定义域错误。

atan函数计算**x**的三角反正切函数主值，结果为弧度，在$-\pi/2$与$\pi/2$之间。不会发生定义域错误和范围错误。一些早期C语言实现把这个函数称为**arctan**。

atan2函数计算**y/x**的三角反正切函数主值，根据两个参数的符号确定象限信息。在正交坐标系中，结果是**x**轴与原点到(**x**,**y**)点所画直线之间的夹角。结果是弧度，在$-\pi$与π之间。如果**x**为0，则结果为$-\pi/2$与$\pi/2$，取决于**y**的符号。如果**x**与**y**都为0，则发生定义域错误。

434

函数**acosh**计算**x**的非负反双曲余弦。如果**x**<1，则发生定义域错误。

函数**asinh**计算**x**的反双曲正弦。

函数**atanh**计算**x**的反双曲正切。如果**x**<－1或**x**>1，则发生定义域错误。**x**为－1或1时则可能发生范围错误。

参考章节　通用类型宏　17.12

17.11　fdim、fmax、fmin

语法概要

```
#include <math.h>                          // All new in C99

double      fdim (double x, double y);
float       fdimf(float x, float y);
long double fdiml(long double x, long double y);

double      fmax (double x, double y);
```

```
float       fmaxf(float x, float y);
long double fmaxl(long double x, long double y);

double      fmin (double x, double y);
float       fminf(float x, float y);
long double fminl(long double x, long double y);
```

函数**fdim**计算**x**与**y**之间的正差值，即在$x>y$时返回$x-y$，$x\leqslant y$时返回+0。

fmax函数返回两个参数中较大的值（到$+\infty$），**fmin**函数返回两个参数中较小的值（到$-\infty$）。对于这两个函数，如果一个参数为数字，而一个参数为NaN，则返回这个数字。

参考章节 NaN 5.2；通用类型宏 17.12

17.12 通用类型宏

C99定义了一组通用类型宏，可以对使用数学函数或复数函数的C语言程序提高移植性。这些宏根据参数类型扩展成调用特定库函数。使用宏时，可以包括库头文件**tgmath.h**，其中包括库头文件**math.h**与**complex.h**。

表17-1列出了使用原型表示的通用类型宏，其中**T**表示通用类型**float**、**double**、**long double**、**float complex**、**double complex**或**long double complex**。**REAL(T)**表示与通用复数类型长度相同的实数类型。尽管大多数函数带一个通用参数，但有些函数可能带多个通用参数，有些函数还带其他特定（非通用）参数。这些参数类型不管通用类型如何，总是相同的。 435
表中还列出了根据参数类型实际调用的实数与复数函数。函数名根据C语言库函数中原始的**double**版本名采用一致规则：复数函数前面加上**c**字母，带有**float**或**float complex**参数的函数前面加上**f**字母，带有**long double**或**long double complex**参数的函数前面加上字母**l**。

例 实现可以对通用类型宏进行特殊处理，但作为例子，**sqrt**宏可以实现如下：

```
#define sqrt(x) \
  ((sizeof(x) == sizeof(float)) ? sqrt(x) :
   (sizeof(x) == sizeof(double)) ? sqrtf(x) : sqrtl(x))
```

□

如果用某个类型的通用参数调用表17-1中的通用类型宏，则用下列规则确定调用哪个函数。选择函数之后，所有参数转换成这个函数的相应类型，如果有函数原型，则按照参数转换的普通规则进行转换。

1. 如果任何通用参数的类型为**long double complex**，则调用**long double complex**版本函数。如果没有这种函数，则结果是未定义的。
2. 如果任何通用参数的类型为**double complex**，则调用**double complex**版本函数。如果没有这种函数，则结果是未定义的。
3. 如果任何通用参数的类型为**float complex**，则调用**float complex**版本函数。如果没有这种函数，则结果是未定义的。
4. 如果任何通用参数的类型为**long double**，则调用**long double**版本函数。如果没有这种函数而有**long double complex**版本函数，则调用这个复数函数。
5. 如果任何通用参数的类型为**double**，则调用**double**版本函数。如果没有这种函数而有**double complex**版本函数，则调用这个复数函数。
6. 否则调用**float**版本函数（要使用这个规则，所有通用参数都应为**float**类型）。如果没

436 有这种函数而有**float complex**版本函数，则调用这个复数函数。

表17-1 通用类型宏

通用类型宏（tgmath.h）	实数函数（math.h）	复数函数（complex.h）
T acos(T x)	acos, acosf, acosl	cacos, cacosf, cacosl
T acosh(T x)	acosh, acoshf, acoshl	cacosh, cacoshf, cacoshl
T asin(T x)	asin, asinf, asinl	casin, casinf, casinl
T asinh(T x)	asinh, asinhf, asinhl	casinh, casinhf, casinhl
T atan(T x)	atan, atanf, atanl	catan, catanf, catanl
T atan2(T y, T x)	atan2, atan2f, atan2l	
T atanh(T x)	atanh, atanhf, atanhl	catanh, catanhf, catanhl
T carg(T x)		carg, cargf, cargl
T cbrt(T x)	cbrt, cbrtf, cbrtl	
T ceil(T x)	ceil, ceilf, ceill	
REAL(T) cimag(T x)		cimag, cimagf, cimagl
T conj(T x)		conj, conjf, conjl
T copysign(T x, T y)	copysign, copysignf, copysignl	
T cos(T x)	cos, cosf, cosl	ccos, ccosf, ccosl
T cosh(T x)	cosh, coshf, coshl	ccosh, ccoshf, ccoshl
T cproj(T x)		cproj, cprojf, cprojl
REAL(T) creal(T x)		creal, crealf, creall
T erf(T x)	erf, erff, erfl	
T erfc(T x)	erfc, erfcf, erfcl	
T exp(T x)	exp, expf, expl	cexp, cexpf, cexpl
T exp2(T x)	exp2, exp2f, exp2l	
T expm1(T x)	expm1, expm1f, expm1l	
T fabs(T x)	fabs, fabsf, fabsl	cabs, cabsf, cabsl
T fdim(T x, T y)	fdim, fdimf, fdiml	
T floor(T x)	floor, floorf, floorl	
T fma(T x, T y, T z)	fma, fmaf, fmal	
T fmax(T x, T y)	fmax, fmaxf, fmaxl	
T fmin(T x, T y)	fmin, fminf, fminl	
T fmod(T x, T y)	fmod, fmodf, fmodl	
T frexp(T value, int *exp)	frexp, frexpf, frexpl	
T hypot(T x, T y)	hypot, hypotf, hypotl	
int ilogb(T x)	ilogb, ilogbf, ilogbl	
T ldexp(T x, int exp)	ldexp, ldexpf, ldexpl	

437

（续）

通用类型宏（tgmath.h）	实数函数（math.h）	复数函数（complex.h）
T lgamma(T x)	lgamma, lgammaf, lgammal	
long long int llrint(T x)	llrint, llrintf, llrintl	
long long int llround(T x)	llround, llroundf, llroundl	
T log(T x)	log, logf, logl	clog, clogf, clogl
T log10(T x)	log10, log10f, log10l	
T log1p(T x)	log1p, log1pf, log1pl	
T log2(T x)	log2, log2f, log2l	
T logb(T x)	logb, logbf, logbl	
long int lrint(T x)	lrint, lrintf, lrintl	
long int lround(T x)	lround, lroundf, lroundl	
无	modf, modff, modfl	
T nearbyint(T x)	nearbyint, nearbyintf, nearbyintl	
T nextafter(T x)	nextafter, nextafterf, nextafterl	
T nexttoward(T x, long double y)	nexttoward, nexttowardf, nexttowardl	
T pow(T x, T y)	pow, powf, powl	cpow, cpowf, cpowl
T remainder(T x, T y)	remainder, remainderf, remainderl	
T remquo(T x, Ty, int *quo)	remquo, remquof, remquol	
T rint(T x)	rint, rintf, rintl	
T round(T x)	round, roundf, roundl	
T scalbln(T x, long int n)	scalbln, scalblnf, scalblnl	
T scalbn(T x, int n)	scalbn, scalbnf, scalbnl	
T sin(T x)	sin, sinf, sinl	csin, csinf, csinl
T sinh(T x)	sinh, sinhf, sinhl	csinh, csinhf, csinhl
T sqrt(T x)	sqrt, sqrtf, sqrtl	csqrt, csqrtf, csqrtl
T tan(T x)	tan, tanf, tanl	ctan, ctanf, ctanl
T tanh(T x)	tanh, tanhf, tanhl	ctanh, ctanhf, ctanhl
T tgamma(T x)	tgamma, tgammaf, tgammal	
T trunc(T x)	trunc, truncf, truncl	

17.13 erf、erfc、lgamma、tgamma

语法概要

```
#include <math.h>           // All new in C99 but common in UNIX

double      erf (double x);
float       erff(float x);
long double erfl(long double x);

double      erfc (double x);
float       erfcf(float x);
long double erfcl(long double x);

double      lgamma (double x);
float       lgammaf(float x);
long double lgammal(long double x);

double      tgamma (double x);
float       tgammaf(float x);
long double tgammal(long double x);
```

erf函数计算误差函数：

$$\frac{2}{\sqrt{\pi}} \cdot \int_0^x e^{-t^2} dt$$

erfc函数计算**1-erf(x)**，即：

$$\frac{2}{\sqrt{\pi}} \cdot \int_x^\infty e^{-t^2} dt$$

lgamma函数计算数值**x**的gamma函数的自然对数：

$$\log|\Gamma(x)|$$

439

tgamma函数计算**x**数量的gamma函数：

$$\Gamma(x)$$

17.14 fpclassify、isfinite、isinf、isnan、isnormal、signbit

语法概要

```
#include <math.h>          // All new in C99

int fpclassify(real-floating-type x);
#define FP_INFINITE ...
#define FP_NAN ...
#define FP_NORMAL ...
#define FP_SUBNORMAL ...
#define FP_ZERO ...

int isfinite(real-floating-type x);

int isinf(real-floating-type x);

int isnan(real-floating-type x);
```

```
int isnormal(real-floating-type x);
int signbit(real-floating-type x);
```

本节的宏为通用类型，其参数由*real-floating-type*（实数浮点数类型）列出，可以是任何实数浮点数类型表达式。由于浮点数表达式可以用比实际词法类型更大的精度求值，因此这些宏负责将参数表达式转换为正确类型表示之后再进行检查。C语言标准指出，**long double**格式的规格化数字可能在**double**格式中次规格化，可能在**float**格式中变成0。

fpclassify宏返回**FP_INFINITE**、**FP_NAN**、**FP_NORMAL**、**FP_SUBNORMAL**或**FP_ZERO**值之一。每个宏是一个单独的整型常量表达式。其他以**FP_**开头，后面使用大写字母的分类宏可以在C语言实现中指定。

isfinite宏返回非0值的条件为参数既不是无限大也不是NaN，次规格化数是有限数。

isinf宏返回非0值的条件为参数是无限大（符号任意）。

isnan宏返回非0值的条件为参数是NaN。

isnormal宏返回非0值的条件为参数是规格化数，而对0、次规格化数、无限大和NaN则返回0。

signbit宏返回非0值的条件为参数是负数。

440

17.15 copysign、nan、nextafter、nexttoward

语法概要

```
#include <math.h>      // All new in C99

double      copysign (double x, double y);
float       copysignf(float x, float y);
long double copysignl(long double x, long double y);

double      nan (const char *tagp);
float       nanf(const char *tagp);
long double nanl(const char *tagp);

double      nextafter (double x, double y);
float       nextafterf(float x, float y);
long double nextafterl(long double x, long double y);

double      nexttoward (double x, long double y);
float       nexttowardf(float x, long double y);
long double nexttowardl(long double x, long double y);
```

本节介绍的函数操纵浮点数值。

copysign函数返回的**x**采用**y**的符号。

nan函数返回一个静态NaN，如果C语言实现提供静态NaN，则其内容由**tagp**指定的字符串表示，否则返回0。调用

```
nan("char-sequence")
nan("")
nan(NULL)
```

分别等价于调用

```
strtod("NAN(char-sequence)", (char **) NULL)
strtod("NAN()", (char **) NULL)
```

strtod("NAN", (char **) NULL)

调用nanf与nanl映射strtof与strtodl的相应调用。

nextafter函数返回y方向上x的下一个可表示浮点数值。如果没有这样的有限值，则可能发生范围错误。如果x与y相等，则返回y。注意，参数和返回值实际上转换成正式参数和返回类型，即使在宏实现中也是如此，因为准确的浮点数表示非常重要。

nexttoward函数与nextafter函数等价，只是y的类型总是long double。

441 **参考章节** 静态NaN 5.2; strtod 13.8

17.16 isgreater、isgreaterequal、isless、islessequal、islessgreater、isunordered

语法概要

```
#include <math.h>          // All new in C99
int isgreater(real-floating-type x, real-floating-type y);
int isgreaterequal(real-floating-type x, real-floating-type y);
int isless(real-floating-type x, real-floating-type y);
int islessequal(real-floating-type x, real-floating-type y);
int islessgreater(real-floating-type x, real-floating-type y);
int isunordered(real-floating-type x, real-floating-type y);
```

如果两个浮点数值中的一个或两个为NaN，则它们是无序的。对无序值使用C语言的比较运算符时，会产生“无效”浮点数异常。本节介绍的通用类型比较宏不会产生异常，因此在某些浮点数编程中很有用。如果C语言实现不会对比较运算符产生“无效”浮点数异常，则这些比较运算符与这些宏作用相同。

isunordered宏返回真值的条件为参数无序。

isgreater宏在参数无序时返回0，否则返回(x)>(y)。

isgreaterequal宏在参数无序时返回0，否则返回(x)>=(y)。

isless宏在参数无序时返回0，否则返回(x)<(y)。

islessequal宏在参数无序时返回0，否则返回(x)<=(y)。

islessgreater宏在参数无序时返回0，否则返回(x)<(y)||(x)>(y)（无需求值参数两次）。

442 **参考章节** NaN 5.2

第18章 时间和日期函数

本章介绍的函数使C语言编程人员可以获得并使用日历日期和时间以及处理器时间，即运行程序所用的处理器时间量。

可以用日历时间记录运行程序或打开文件的日期，或计算过去或未来的日期。日历时间用两种形式表示：时间函数返回的简单算术值；**gmtime**与**localtime**函数从简单算术值求出的分解的结构化形式。标准C语言函数**strftime**提供特定区域设置的格式。

处理器时间常用于衡量程序或部分程序的运行快慢。处理器时间表示为**clock**函数返回的算术值（通常是整型值）。

18.1 clock、clock_t、CLOCKS_PER_SEC、times

语法概要

```
#include <time.h>

typedef ... clock_t;
#define CLOCKS_PER_SEC ...
clock_t clock(void);
```

clock函数返回当前进程使用处理器时间的近似值。时间单位随实现而不同，通常以微秒为单位。标准C语言**clock**函数允许实现者随意使用任何算术类型**clock_t**表示处理时间。每秒的时间单位数（时钟滴答）用**CLOCKS_PER_SEC**宏定义。如果无法得到处理器时间，则返回数值 -1（转换成**clock_t**类型）。

编程人员要当心处理器时间被覆盖。例如，如果类型**clock_t**表示为32位，而**clock**返回的时间单位为微秒，则返回的时间在大约36分钟内覆盖开始值。

例 用**clock**函数定时标准C语言程序的方法如下：

```
#include <time.h>
clock_t start, finish;
...
start = clock();
process();
finish = clock();
printf("process() took %f seconds to execute\n",
       ((double) (finish - start)) / CLOCKS_PER_SEC );
```

转换成**double**类型的类型转换允许**clock_t**与**CLOCKS_PER_SEC**可以是浮点数或整数。□

在传统C语言中，**clock**的返回类型为**long**，但返回的值实际上是**unsigned long**类型，**long**是在这个语言中加入**unsigned long**之前使用的。计算处理器时间时总是用无符号算术。一些非标准实现中使用**times**函数而不是**clock**函数，其返回的结构化值报告处理器时间的各个成员，通常用1/60秒为单位。语法如下：

```
#include <sys/types.h>
#include <sys/times.h>
long clock(void);
void times(struct tms *);
struct tms { … };
```

例 可以用非标准**times**函数编写近似的标准C语言**clock**函数如下：

```
#include <sys/types.h>
#include <sys/times.h>
#define CLOCKS_PER_SEC 60
long clock(void)
{
    struct tms tmsbuf;
444 times(&tmsbuf);
    return (tmsbuf.tms_utime + tmsbuf.tms_stime);
}
```

旧的结构使用了**time_t**类型，这是个处理器时间单位，因此不同于标准C语言中定义的日历时间类型**time_t**。 □

参考章节 **time** 18.2; **time_t** 18.2

18.2 time、time_t

语法概要

```
#include <time.h>

typedef ... time_t;
time_t time(time_t *tptr);
```

标准C语言函数**time**返回当前日历时间，编码成**time_t**类型的值，可以是任何算术类型。如果参数**tptr**不是null，则返回值还存放在***tptr**中。如果遇到错误，则返回-1（转换成**time_t**类型）。

通常，**time**返回的值传入**asctime**或**ctime**函数，将其转换成可读形式，或传入**localtime**或**gmtime**中，转换成更容易处理的形式。可以用标准C语言函数**difftime**计算两个日历时间的间隔，在其他实现中，编程人员可能要使用**gmtime**分解的时间或用时间的习惯表示，即离过去某个日期的秒数（通常是距1970年1月1日）。

在传统实现中，用类型**long**代替**time_t**，但返回的值逻辑上为类型**unsignedlong**。发生错误时，返回**-1L**。在System V UNIX中，**errno**还设置为**EFAULT**。

参考章节 **asctime** 18.3; **ctime** 18.3; **difftime** 18.5; **errno** 11.2; **gmtime** 18.4; **localtime** 18.4

18.3 asctime、ctime

语法概要

```
#include <time.h>

char *asctime( const struct tm *ts );
char *ctime( const time_t *timptr );
```

asctime或**ctime**函数都返回可打印日期与时间字符串的指针，其形式如下所示：

```
"Sat May 15 17:30:00 1982\n"
```

445

asctime函数的惟一参数是结构化日历时间的指针，这种结构是**localtime**或**gmtime**函数从**time**返回的算术时间生成的。**ctime**函数取**time**所返回数值的指针，因此**ctime(tp)**等价于**asctime(localtime(tp))**。

大多数实现中（包括许多符合标准C语言的实现），函数返回静态数据区的指针，因此返回的字符串应先打印或复制（用**strcpy**）之后再调用这些函数。

在传统C语言中，**long**类型代替**time_t**。可以在头文件**sys/time.h**中找到这些函数。

例 许多程序需要打印当前日期与时间。下面用**time**和**ctime**函数打印：

```
#include <time.h>
#include <stdio.h>
time_t now;
...
now = time(NULL);
printf("The current date and time is: %s",ctime(&now));
```

□

参考章节 **gmtime** 18.4; **localtime** 18.4; **strcpy** 13.3; **struct tm** 18.4; **time** 18.2

18.4 gmtime、localtime、mktime

语法概要

```
#include <time.h>
struct tm { ... };
struct tm *gmtime( const time_t *t );
struct tm *localtime( const time_t *t );
time_t mktime( struct tm *tmptr );
```

函数**gmtime**与**localtime**将**time**返回的算术日历时间转换成**struct tm**类型的分解形式。**gmtime**函数转换GMT时间，而**localtime**函数转换本地时间，还要考虑时区和夏时制等因素。如果遇到错误，则函数返回unll指针。函数可以在UNIX系统和标准C语言之间移植。**struct tm**结构包括表18-1所示的字段。所有字段的类型为**int**。

在大多数实现中（包括许多标准实现），**gmtime**与**localtime**返回单一静态数据区的指针，在每次调用时覆盖、因此，返回的结构应先打印或复制之后再调用这些函数。

函数**mktime**（标准C语言）从参数**tmptr**指定的经过分解的本地时间构造**time_t**类型的值。

表18-1 **struct tm**结构的字段

名 称	单 位	范 围
tm_sec	分钟后的秒数	0..61[1]
tm_min	小时后的分钟数	0..59
tm_hour	从午夜算起的时间	0..23
tm_mday	当月第几天	1..31

（续）

名　称	单　位	范　围
tm_mon	从1月算起的月份	0..11
tm_year	从1900年起的年数	
tm_wday	从星期天算起的星期几	0..6
tm_yday	从1月1日算起的天数	0..365
tm_isdst	夏时制标志	>0为夏时制，0为不是夏时制，<0为不确定

① 允许最多两个闰秒（C89），但C99只要一个闰秒。

mktime忽略**tmptr->tm_wday**与**tmptr->tm_yday**的值。如果成功，则**mktime**返回新时间值，并调整***tmptr**内容，设置**tm_wday**与**tm_yday**成员。如果指定的日历时间无法表示为**time_t**类型值，则**mktime**返回－1（转换成**time_t**）类型。例子见18.5节。

传统C语言的时间函数语法如下：

```
#include <sys/time.h>
struct tm { … };
struct tm *gmtime(long *t);
struct tm *localtime(long *t);
```

18.5 difftime

语法概要

```
#include <time.h>
double difftime( time_t t1, time_t t0 );
```

difftime函数只在标准C语言中提供，从日历时间**t1**减去日历时间**t0**，返回**double**类型的差值（秒数）。编程人员不能假设日历时间编码为**time_t**类型的标量值（如微秒数），因此**difftime**只能用两个**double**类型的值相减。

447

例　下列函数返回1990年4月15日午夜与当前日期和时间之间相差的秒数。

```
#include <time.h>
…
double Secs_Since_Apr_15(void)
{
    struct tm Apr_15_struct = {0}; /* Set all fields to 0 */
    time_t Apr_15_t;
    Apr_15_struct.tm_year = 90;
    Apr_15_struct.tm_mon  = 3;
    Apr_15_struct.tm_mday = 15;
    Apr_15_t = mktime(&Apr_15_struct);
    if (Apr_15_t == (time_t)-1)
        return 0.0; /* error */
    else
        return difftime( time(NULL), Apr_15_t);
}
```

□

参考章节 `time_t` 18.2

18.6 strftime、wcsftime

语法概要

```
#include <time.h>
size_t strftime(
  char *s, size_t maxsize,
  const char *format,
  const struct tm *timeptr);
#include <wchar.h>
size_t wcsftime(
  wchar_t *s, size_t maxsize,
  const wchar_t *format,
  const struct tm *timeptr);
```

这些函数只在标准C语言中提供。和**sprintf**一样（见15.11节），**strftime**在多字节字符串**format**控制下将字符存放在参数**s**所指的字符数组中。但是，**strftime**只是格式化**timeptr**（18.4节）指定的一个日期和时间量，**format**中的格式代码与**sprintf**中有不同解释。**s**指定的数组中不能放超过**maxsize**个字符（包括终止null字符），返回存储的实际字符数（不包括终止null字符），如果**maxsize**不够大，放不下整个格式字符串，则返回0，放弃输出字符串内容的定义。**strftime**格式是特定区域设置的，使用**LC_TIME**类别（见**setlocale**，20.1节）。

C89增补1增加了**wcsftime**函数，可以将日期与时间格式化成宽字符串。这个函数与**wsprintf**相似（见15.11节）。

448

格式代码

format设置包括转换说明与其他多字节字符的任意混合。在格式化过程中，转换说明换成表18-2所示的其他字符，其他多字节字符直接复制到输出中。转换说明包括**%**号、可选的一个修正符**E**或**O**加上一个说明转换的字符。

表18-2 **strftime**格式代码

字 母	替 换	使用的timeptr字段
a	缩写星期名，在C语言区域设置中总是%A的前3个字母，如"Mon"（等等）	tm_wday
A	全名星期名，在C语言区域设置中为"Monday"（等等）	tm_wday
b	缩写月份名，在C语言区域设置中总是%B的前3个字母，如"Feb"（等等）	tm_mon
B	全名月份名，在C语言区域设置中为"February"（等等）	tm_mon
c	特定区域设置的日期与时间，在C语言区域设置中与%a %b %e %T %y相同	任意或全部

（续）

字　母	替　换	使用的timeptr字段
C	（C99中）年份的后两位（00～99）	tm_year
d	当月日期（01～31）	tm_mday
D	等价于%m/%d/%y	tm_mon、tm_mday、tm_year
e	当月日期（1～31），单位数前面用空格	tm_mday
F	ISO 8601日期格式%Y-%m-%d	tm_mon、tm_mday、tm_year
g	基于周的年份后两位（00～99）①	tm_year、tm_wday,tm_yday
G	基于周的年份（0000～9999）	tm_year、tm_wday、tm_yday
h	与%b相同	tm_mon
H	24时制时间，十进制整数（00～23）	tm_hour
I	12时制时间，十进制整数（01～12）	tm_hour
j	当年日期，十进制整数（001～366）	tm_yday
m	月份，十进制整数（01～12）	tm_mon
M	分钟，十进制整数（00～59）	tm_min
n	（C99）换成换行符	无
P	12时制的特定区域设置的上下午指定符，在C语言区域设置中为AM或PM	tm_hour
r	(C99)12时制时间，在C语言区域设置中为%I:%M:%S %p	tm_hour、tm_min、tm_sec
R	(C99)同%H:%M	tm_hour、tm_min,
S	秒数，十进制整数（00～60）②	tm_sec
t	(C99)换成水平制表符	无
T	(C99)ISO 8601时间格式：%H:%M:%S	tm_hour、tm_min, tm_sec
u	(C99)ISO 8601星期数（1～7），1为星期一	tm_wday
U	一年的周数(00～53)③	tm_year, tm_wday, tm_yday
V	(C99)ISO 8601周数(01～53)，用于基于周的年份	tm_year、tm_wday、tm_yday
w	星期数（0～6），0为星期日	tm_wday
W	一年的周数(00～53)④	tm_year、tm_wday、tm_yday
x	特定区域设置的日期，在C语言区域设置中为%m/%d/%y	任意或全部
X	特定区域设置的时间，在C语言区域设置中为%T	任意或全部
y	年份后两位（00～99）	tm_year
Y	世纪年份，十进制整数（如1952）	tm_year
z	(C99)ISO 8601与UTC的时区偏差，或无意义；-530表示与格林威治的时差为5小时30分钟	tm_isdst
Z	时区名或缩写，不知道时区时无意义；在C语言区域设置中是由实现定义的	tm_isdst
%	一个%	无

① 见基于周的年份的定义。
② 允许一个闰秒（60）。
③ 周数1为第一个星期日，上一天为第0周。
④ 周数1为第一个星期一，上一天为第0周。

修正符和许多转换字母是C99增加的。修正符**E**可以用于转换**c**、**C**、**x**、**X**、**y**和**Y**，说明使用区域设置的替换表示法（不指定）。修正符**O**可以用于转换**d**、**e**、**H**、**I**、**M**、**m**、**s**、**u**、**U**、**V**、**w**、**W**与**y**，说明使用区域设置的替换数字符号（不指定）。在C语言区域设置中，忽略修正符。 450

一些C99转换字母根据ISO 8601基于周的年份指定转换。在这种系统中，一周从星期一开始，一年第1周为1月4号所在的周（即第一周至少包含新年中的4天）。这样，1月1日、1月2日和1月3日可能属于上一年最后一周，12月29日、12月30日、12月31日也可能属于下一年第一周。例如，1999年1月2日星期六是1998年第53周。而**%U**与**%W**则在需要时引入“第0周”。

例 下面用**strftime**实现**asctime**（18.3节）。由于格式是特定区域设置的，因此输出字符串长度（包括null终止字符）很难预测（这里是**asctime**的输出）：

```
#include <time.h>
#define TIME_SIZE 80  /* hope this is big enough */
char *asctime2( const struct tm *tm )
{
    static char time_buffer[TIME_SIZE];
    size_t len;
    len = strftime( time_buffer, TIME_SIZE,
        "%a %b %d %H:%M:%S %Y\n", tm);
    if (len == 0)
        return NULL;  /* time_buffer is too short */
    else
        return time_buffer;
}
```

□ 451

第19章 控制函数

本章介绍的函数对C语言程序的标准控制流提供扩展，在头文件**assert.h**、**setjmp.h**与**signal.h**中提供。

19.1 assert、NDEBUG

语法概要

```
#include <assert.h>
#ifndef NDEBUG
void assert( int expression );
#else
#define assert(x) ((void)0)
#endif
```

assert宏带单个参数，可以是任何整数类型的值（许多实现允许任何标量类型）。如果这个值为0，而不定义**NDEBUG**宏，则**assert**在标准输出流中打印一个诊断消息，并调用**abort**函数（标准C语言）或**exit**函数（传统C语言）终止程序。**assert**函数总是用宏实现，源文件中要包括头文件**assert.h**才能使用这个宏。诊断消息包括参数文本、文件名(**__FILE__**)和行号(**__LINE__**)。C99实现还可以使用函数名(**__func__**)。

如果读取**assert.h**头文件时定义了**NDEBUG**宏，则通常把**assert**定义为空语句，将
453 **assert**功能关闭，不打印诊断消息，不求值**assert**的参数。

例 **assert**函数通常在程序开发期间用于验证某些条件在运行时是否为真。它向读程序的人提供可靠的文档，并对调试大有帮助。程序可操作时，可以很容易地关闭断言，避免运行开销。下例中，断言是比英文注释还好的文档说明，可以避免误解：

```
#include <assert.h>
int f(int x)
{
    /* x should be between 1 and 10 */    /* 1? */
    assert(x>0 && x<10);
    ...
}
```

□

参考章节 **abort** 19.3; **exit** 19.3; **__func__** 2.6.1; **__LINE__** 3.3.4

19.2 system、exec

参见16.7节。

19.3 exit、abort

参见16.5节。

19.4 setjmp、longjmp、jmp_buf

语法概要

```
#include <setjmp.h>
typedef ... jmp_buf;
int setjmp( jmp_buf env );
void longjmp( jmp_buf env, int status );
```

setjmp与**longjmp**函数实现基本形式的非本地跳转，可以处理异常或例外情形。这个函数比**signal**更好移植（见19.6节），但后者也已经加进标准C语言中。

setjmp宏在“跳转缓冲区”参数**env**中记录调用者的环境，**env**是个由实现定义的数组，并向调用者返回0（类型**jmp_buf**要实现为数组类型，使**env**的指针实际传递到**setjmp**）。

函数**longjmp**的参数为前面调用**setjmp**填充的跳转缓冲区和一个整型值**status**（通常为非0值）。调用**longjmp**的结果是程序再次从调用**setjmp**返回，这时返回**status**值。一些实现 454
（包括标准C语言）不允许**longjmp**产生从**setjmp**返回0的结果，如果用**status**值0调用**longjmp**，则会从**setjmp**返回1。

setjmp与**longjmp**函数的实现非常困难，编程人员最好别对其做太多假设。**setjmp**返回非0值时，编程人员可以假设调用**longjmp**时静态变量具有适当的值。**setjmp**所在函数本地的自动变量在标准C语言中保证有正确值的条件是具有**volatile**限定类型或最初调用**setjmp**与相应**longjmp**调用之间没有改变其数值。此外，标准C语言要求调用**setjmp**时是整个表达式语句（可能转换成**void**）、简单赋值表达式右边或作为**if**、**switch**、**do**、**while**或**for**语句的控制表达式，形式如下：

```
(setjmp(...))
(!setjmp(...))
(exp relop setjmp(...))
(setjmp(...) relop exp)
```

其中*exp*是整型常量表达式，*relop*是关系运算符或判等运算符。C89要求**longjmp**在非嵌套信号（中断）处理器中正确操作，但C99取消了这一要求。中断处理器应该被认为是由实现定义的。

如果**setjmp**没有设置**longjmp**的跳转缓冲区参数，或**setjmp**所在的函数在调用**longjmp**之前终止，则行为是未定义的。

例

```
#include <setjmp.h>
jmp_buf ErrorEnv;
...

int guard(void)
    /* Return 0 if successful; else longjmp code. */
{
    int status = setjmp(ErrorEnv);
    if ( status != 0) return status;  /* error */
```

```
    process();
    return 0;
}

int process(void)
{  …
455    if (error_happened) longjmp(ErrorEnv, error_code);
   …
}
```

longjmp函数在**process**函数中遇到错误时调用。**guard**函数是它的下一站，控制由**longjmp**函数转移。函数**process**应从**guard**中直接或间接调用，保证无法在**guard**返回之后调用**longjmp**，不会依赖于**longjmp**所在函数**process**中的局部变量值（这是个保守政策）。注意，要测试**setjmp**的返回值，确定返回是否由**longjmp**造成的。 □

19.5 atexit

参见16.5节。

19.6 signal、raise、gsignal、ssignal、psignal

语法概要

```
#include <signal.h>
#define SIG_IGN …
#define SIG_DFL …
#define SIG_ERR …
#define SIGxxx …
...
void (*signal( int sig, void (*func)(int) )) (int);
int raise( int sig );
typedef … sig_atomic_t;

/* Non-Standard extensions: */
int kill( int pid, intsig );
int (*ssignal( int softsig, int (*func)(int) )) (int);
int gsignal( int softsig );
void psignal( int sig, char *prefix );
```

信号是（潜在的）异步事件，可能要在用户程序或实现中进行特殊处理。信号用整数值指定，每个实现定义一组信号，放在头文件**signal.h**中，以字母**SIG**开头。信号的触发可以通过计算机的错误探测机制、用户程序中的**kill**或**raise**函数以及程序外部的操作。函数**ssignal**与**psignal**使用的软件信号是用户定义的，数值通常在1~15之间，否则它们像普通信号一样操作。

信号**sig**的信号处理器是个用户函数，在发出信号**sig**时调用。处理器函数要进行一些有用的操作，然后返回，通常使程序恢复中断所在点。处理器还可以调用**exit**。信号处理器是普通
456 C语言函数，带有一个参数，即发出的信号：

```
void my_handler(int the_signal) { … }
```

一些非标准实现可能向信号处理器传递其他参数，用于某些预定义信号。

函数**signal**将信号处理器与特定信号相关联。正常情况下，**signal**接收一个信号值和这个信号的信号处理器指针。如果关联成功，则**signal**返回上一信号处理器的指针，否则返回－1值（在标准C语言中为**SIG_ERR**）并设置**errno**。

例

```
void new_handler(int sig) { … }
void (*old_handler)();
…
/* Set new handler, saving old handler */
old_handler = signal( sig, &new_handler );
if (old_handler==SIG_ERR)
   fprintf("stderr,"?Could't establish new handler./n")
…
/* Restore old handler */
if (signal(sig,old_handler)==SIG_ERR)
   fprintf("stderr,"?Could't put back old handler.\n");
```

signal的函数参数和返回值还可以有两个特殊值**SIG_IGN**与**SIG_DFL**。**signal(sig, SIG_IGN)**形式的**signal**调用表示要忽略信号**sig**。**signal(sig, SIG_DFL)**形式的**signal**调用表示信号**sig**要接受缺省处理，通常是忽略某些信号和在遇到另一些信号时终止程序。

ssignal函数（在UNIX System V中）和**signal**相似，但只和**gsignal**一起用于用户定义软件信号。向**ssignal**提供的信号处理器返回一整数值成为**gsignal**的返回值。

raise或**gsignal**函数在当前进程中发出指定的信号（或软件信号）。**kill**函数使特定进程中发出指定的信号，其移植性较差。

对**signal**或**gsignal**建立的处理器发出信号时，处理器取得控制权。标准C语言（和大多数其他实现）或者在处理器取得控制权之前将相关处理器重置为**SIG_DFL**，或者用其他某种方法阻止信号，从而避免不必要的递归（这是否对**SIGILL**信号发生是由实现定义的，有历史和性能方面的原因）。信号处理器可能返回，这时执行在中断点继续，但有下列注意事项：

1. 如果信号是由**raise**或**gsignal**发出的，则这些函数返回其调用者。
2. 如果信号是由**abort**发出的，则标准C语言程序终止。其他实现可能返回**abort**的调用者。 457
3. 如果处理的信号是**SIGFPE**或另一实现定义的计算信号，则返回时的行为是未定义的。

信号处理器要避免调用**signal**以外的库函数，因为有些信号可能从库函数发出，而**signal**以外的库函数不一定是可再入的。

标准C语言定义表19-1所示的宏，作为某些标准信号。这些信号在C语言的许多实现版本中很常见。

表19-1 标准信号

宏 名	信号含义
SIGABRT	异常终止，如由**abort**函数造成
SIGFPE	错误算术运行，如除数为0
SIGILL	无效计算机指令造成错误

（续）

宏 名	信号含义
SIGINT	提示信号，如交互式用户按下特殊键
SIGSEGV	无效内存访问
SIGTERM	用户或另一程序的终止信号

psignal函数（标准C语言中不提供）在标准错误输出中打印字符串**prefix**（通常是程序名）和信号**sig**的简要描述。这个函数在需要调用**exit**或**abort**的处理器中很有用。

参考章节 **exit** 19.3; **longjmp** 19.4

19.7 sleep、alarm

非标准语法概要

```
void sleep( unsigned seconds );
unsigned alarm( unsigned seconds );
```

这些函数不属于标准C语言。**alarm**函数将内部系统定时器设置为指定秒数，返回原先定时器上的秒数。定时器超时时，程序中发出信号**SIGALRM**。如果**alarm**的参数为0，则调用的效果是取消任何前面的**alarm**请求。**alarm**函数可以从各种死锁情形中退出。

sleep函数使程序暂停指定秒数，然后**sleep**函数返回，执行继续。休眠通常用与**alarm**
458 相同的定时器实现。如果休眠时间超过**alarm**定时器中已有的时间，则处理**SIGALRM**信号之后，**sleep**立即返回。如果休眠时间少于**alarm**定时器中已有的时间，则**sleep**返回之前重置定时器，使**SIGALRM**信号能够如期收到。

实现通常在处理任何信号时终止**sleep**，有些实现提供缺眠秒数，作为**sleep**的返回值（**unsigned**类型）。

有些实现定义这些函数带有**unsigned long**类型的参数。

459 **参考章节** **signal** 19.6

第20章 区域设置函数

标准C语言是为国际化社区设计的，其成员使用不同字母表和不同的数字、币值、日期与时间格式规则。C语言标准允许实现根据不同情况调整运行库行为，同时保证一定的跨国移植性。

国家、文化和语言规则集称为区域设置，**locale.h**头文件中定义了区域设置相关的函数。区域设置影响数字与币值格式、字母表与排序顺序（见第12章的字符处理功能）和日期与时间格式。运行时可以从实现定义的区域设置集选择不同区域设置。标准C语言只定义了C语言区域设置，这一区域设置指定与原始C语言定义一致的最基本环境。

20.1 setlocale

语法概要

```
#include <locale.h>
#define LC_ALL ...
#define LC_COLLATE ...
#define LC_CTYPE ...
#define LC_MONETARY ...
#define LC_NUMERIC ...
#define LC_TIME ...
char *setlocale( int category, const char *locale );
```

setlocale函数改变运行库的区域设置特定特性。第一个参数**category**指定要改变的行为，**category**允许的值包括表20-1的宏，实现也可以定义以**LC_**开头的类型。

461

表20-1 预定义setlocale类别

名 称	影响的行为
LC_ALL	所有行为
LC_COLLATE	strcoll与strxfrm函数的行为
LC_CTYPE	字符处理函数（见第12章）
LC_MONETARY	localeconv返回的币值信息
LC_NUMERIC	localeconv返回的小数点和非币值信息
LC_TIME	strftime函数的行为

第二个参数**locale**是实现定义的字符串，指定**category**指定的行为规则所用的区域设置。**locale**的惟一预定义值为标准C语言区域设置"C"和""空字符串，习惯上表示实现定义的自然区域设置。运行库总是使用C语言区域设置，直到用**setlocale**显式改变为止。

如果**setlocale**的**locale**参数是null指针，则函数不改变区域设置，而是返回所指定类别当前区域设置名的字符串指针。这个名称使后而用**category**的相同值和用返回字符串作为**locale**值调用**setlocale**时变成用空**locale**调用**setlocale**时生效的行为。例如，编程人员要改变区域设置特定行为时，首先要用参数**LC_ALL**与**NULL**调用**setlocale**，取得当前区域

设置值，后面可以用其恢复前面的区域设置特定行为。返回的字符串不能改变，但后续调用**setlocale**可以将其覆盖。

如果**setlocale**的**locale**参数不是null，则**setlocale**改变当前区域设置并返回表示新区域设置名的字符串。如果**setlocale**因故无法满足请求，则返回null指针。返回的字符串不能改变，但后续调用**setlocale**可以将其覆盖。

例　下列函数**original_locale**返回当前区域设置描述，以便后面在需要时恢复。**setlocale**返回的字符串没有固定的最大长度，因此其空间要动态分配。

```
#include <locale.h>
#include <string.h>
#include <stdlib.h>
462 char *original_locale(void)
{
    char *temp, *copy;
    temp = setlocale(LC_ALL, NULL);
    if (temp == NULL) return NULL;  /* setlocale() failed */
    copy = (char *)malloc(strlen(temp)+1);
    if (copy == NULL) return NULL;  /* malloc() failed */
    strcpy(copy,temp);
    return copy;
}
```

下列代码用**original_locale**改变和恢复区域设置：

```
#include <locale.h>
extern char *original_locale(void);
char *saved_locale;
…
saved_locale = original_locale();
setlocale(LC_ALL,"");           /* Change to native locale */
setlocale(LC_ALL,saved_locale);/* Restore former locale   */
```

□

参考章节　**malloc**　16.1；**localeconv**　20.2；**strcoll**　13.10；**strcpy**　13.3；**strftime**　18.6；**strlen**　13.4；**strxfrm**　13.10

20.2 localeconv

语法概要

```
#include <locale.h>
struct lconv {…};
struct lconv *localeconv(void);
```

localeconv函数取得当前区域设置中格式化数字与币值的规则信息，使编程人员可以实现应用程序的特定转换与格式化程序，具有一定的区域设置间可移植性，避免不必要地在标准C语言中增加区域设置特定转换功能。**localeconv**函数返回**struct lconv**类型对象的指针，其成员至少要包括表20-2的内容。返回的字符串不能改变，但后续调用**localeconv**可以将其覆盖。在**structlconv**中，数值为空字符串的字符串成员和数值为**CHAR_MAX**的字符成员。解释为“不知道”。

例 下列函数利用**localeconv**以正确的小数点字符打印浮点数： 463

```
#include <locale.h>
#include <stdio.h>
…
void P(int int_part, int fract_part, int fract_digits)
{
    struct lconv *lconv = localeconv();
    char *pt = lconv->decimal_point;
    /* If *pt is the empty string, use "." */
    if (!*pt) pt = ".";
    printf("%d%s%0*d\n",
        int_part, pt, fract_digits, fract_part);
}
```

□

表20-2列出了**struct lconv**的其他内容，并在下面介绍。

数字分组 **structlconv**的**grouping**与**mon_grouping**成员是**char**类型的整数值序列。尽管它们描述成字符串，但字符串只是编码小整数序列的一种方式。序列中每个整数指定一个组中的位数。第一个整数对应于小数点左边第一组，第二个整数对应于更左边一组，等等。整数0（字符串末尾的null字符）表示重复上一组数；整数CHAR_MAX表示不进行进一步分组。习惯上的千分法表示为"\3"，第一组3位，后续组重复。字符串"\1\2\3\127"将**1234567890**组成**1234 567 89 0**（**CHAR_MAX**假设为127）。

符号位置 **struct lconv**的**p_sign_posn**与**n_sign_posn**成员分别描述**positive_sign**与**negative_sign**的位置。取值如下：

0 数字与**currency_symbol**放在括号中

1 符号字符串放在数字与**currency_symbol**前面

2 符号字符串放在数字与**currency_symbol**后面

3 符号字符串放在**currency_symbol**前面

4 符号字符串放在**currency_symbol**后面

表20-3和表20-4列出了币值格式的完整例子。表20-3显示了4个国家的典型币值格式，表20-4显示了表20-3所示格式的**struct lconv**成员值。 464

表20-2 **struct lconv**成员

类 型	名 称	用 法	C语言区域设置中的值
char *	decimal_point	小数点号（非币值）	"."
char *	thousands_sep	非币值数字分组分隔符	""
char *	grouping	非币值数字分组	""
char *	int_curr_symbol	三字符国际币值符号加上分隔国际币值符号与币值金额的字符	""
char *	currency_symbol	当前区域设置的本地币值符号	""
char *	mon_decimal_point	小数点号（币值）	""
char *	mon_thousands_sep	币值数字分组分隔符	""
char *	mon_grouping	币值数字分组	""
char *	positive_sign	非负币值量的符号符	""
char *	negative_sign	负币值量的符号符	""

（续）

类　型	名　称	用　法	C语言区域设置中的值
char	int_frac_digits	国际币值格式小数点右边的数字	CHAR_MAX
char	frac_digits	非国际币值格式小数点右边的数字	CHAR_MAX
char	p_cs_precedes	currency_symbol放在负币值前时为1，否则为0	CHAR_MAX
char	p_sep_by_space	currency_symbol与非负币值用空格分开时为1，否则为0	CHAR_MAX
char	n_cs_precedes	类似对于负值的p_cs_ precedes操作	CHAR_MAX
char	n_sep_by_space	类似对于负值的p_sep_by_space操作	CHAR_MAX
char	p_sign_posn	非负币值量的positive_sign定位（加上currency_symbol）	CHAR_MAX
char	n_sign_posn	负币值量的negative_sign定位（加上currency_symbol）	CHAR_MAX

表20-3 典型币值格式

国　家	格　式		
	正	负	国　际
意大利	L.1.234	-L.1.234	ITL.1.234
芬兰	F 1.234,56	F -1.234,56	MLG 1.234,56
挪威	Kr1.234,56	Kr1.234,56-	NOK 1.234,56
瑞士	SFrs.1,234.56	SFrs.1,234.56C	CHF 1,234.56

465

表20-4 struct lconv成员例子

成　员	意大利	芬　兰	挪　威	瑞　士
int_curr_symbol	"ITL."	"MLG"	"NOK"	"CHF "
currency_symbol	"L."	"F"	"kr"	"SFrs."
mon_decimal_point	""	","	","	"."
mon_thousands_sep	"."	"."	"."	","
mon_grouping	"\3"	"\3"	"\3"	"\3"
positive_sign	""	""	""	""
negative_sign	"-"	"-"	"-"	"C"
int_frac_digits	0	2	2	2
frac_digits	0	2	2	2
p_cs_precedes	1	1	1	1
p_sep_by_space	0	1	0	0
n_cs_precedes	1	1	1	1
n_sep_by_space	0	1	0	0
p_sign_posn	1	1	1	1
n_sign_posn	1	4	2	2

466

第21章 扩展整数类型

本章介绍的C99函数提供具有不同特征的整数类型的其他声明，在**stdint.h**与**inttypes.h**头文件中提供。**stdint.h**头文件包含一定长度整数类型的基本定义，是宿主与独立实现所必需的。**inttypes.h**头文件包括**stdint.h**并增加了可移植格式与转换函数，只在宿主实现中需要。

C语言精神是让实现选择标准类型的长度。但是，这种指导思想使可移植代码难以编写。本章介绍的函数解决了移植性问题，但这些头文件中的几个定义比较复杂。

参考章节 宿主实现与独立实现 1.4

21.1 一般规则

这些库包含大量类型、宏和函数，都是按普通方式构造的。本节介绍这个库的一般规则。

21.1.1 类型种类

库包含多种不同“种类”的整数类型和宏，有些用类型宽度*N*参数化。*N*应为非负十进制整数，前面没有0，以“位”来表示类型宽度。

例 准确长度8位整数类型称为**int8_t**与**uint8_t**（而不是**int08_t**与**uint08_t**）。最快整数类型宽度在8位以上时称为**int_fast8_t**与**uint_fast8_t**。“准确长度”与“最快”是两种不同“种类”的类型。 □

21.1.2 全部定义或全部不定义

定义哪些类型（如*N*的哪些值）有时是实现定义的。但是，如果定义某个值*N*的特定“种类”的类型，则该类型的带符号与无符号类型和所有宏及类型长度都要定义。如果特定种类和类型长度是可选的，实现不准备定义，则相关的类型和宏都不定义。

例 如果实现具有准确长度16位整数类型，则类型**int16_t**与**uint16_t**和宏**INT16_MIN**、**INT16_MAX**、**UINT16_MAX**,**PRId16**、**PRIi16**、**PRIo16**、**PRIu16**、**PRIx16**、**PRIX16**、**SCNd16**、**SCNi16**、**SCNo16**、**SCNu16**、与**SCNx16**都要定义。如果实现没有准确长度16位整数类型，则这些类型和宏都不定义。 □

21.1.3 最小限制与最大限制

...**MIN**与...**MAX**宏定义所定义类型的范围时，指定这些类型可表示的最小值与最大值，像标准类型在**limits.h**中的...**MIN**与...**MAX**宏一样。大多数情况下，范围的最小值由C99指定。

例 **int16_t**与**uint16_t**是准确长度16位整数类型，其范围如下：

```
#define INT16_MIN -32768
#define INT16_MAX 32767
#define UINT16_MAX 65535
```

□

参考章节　**limits.h**　表5-2

21.1.4 PRI...与SCN...格式字符串宏

PRI*cKN*与**SCN***cKN*宏是**printf**与**scanf**系列函数的格式控制字符串。*c*表示特定转换操作符：**d**、**i**、**o**、**u**、**x**或**X**。*K*表示“种类”的类型：空或者**LEAST**、**FAST**、**PTR**或**MAX**。*N*是宽度（位）。表21-1列出了相关的宏的完整的集合。

PRI...宏扩展成字符串字面值，包含**printf**转换操作符（**d**、**i**、**o**、**u**、**x**或**X**），前面加上特定种类与长度类型输出值的可选长度说明。**SCN**...宏扩展成字符串字面值，包含**scanf**转换操
468 作符（**d**、**i**、**o**、**u**、**x**或**X**），前面加上适合转换数字输入的可选长度说明，并将其存放在特定种类与长度类型指针指定的对象中。

例　至少64位宽的最小整数类型称为**int_least64_t**与**uint_least64_t**（种类*K*为**LEAST**）。如果这些类型分别定义为**long**与**unsigned long**，则可以在**inttypes.h**中看到下列定义：

```
#define PRIdLEAST64 "ld"
#define PRIiLEAST64 "li"
#define PRIoLEAST64 "lo"
#define PRIuLEAST64 "lu"
#define PRIxLEAST64 "lx"
#define PRIXLEAST64 "lX"
#define SCNdLEAST64 "ld"
#define SCNiLEAST64 "li"
#define SCNoLEAST64 "lo"
#define SCNuLEAST64 "lu"
#define SCNxLEAST64 "lx"
```

假设变量**a**的类型为**long**，**b**的类型为**int_least64_t**。下面两条语句显示了两种打印这些值的方式。第二种方式的移植性更好，不管**int_least64_t**指定什么整数类型，都能工作：

```
printf("a=%25ld\n", a); /* usual */
printf("b=%25" PRIdLEAST64 "\n", b); /* portable */
```

□

参考章节　**limits.h**　表5-2；**printf**转换　15.11.7；**scanf**转换　15.8.2

表21-1　整型的格式控制字符串宏（*N*=类型宽度（位））

	准确长度类型	最少长度类型	快速长度类型	指针长度类型	最大长度类型
带符号	**PRIdN**	**PRIdLEASTN**	**PRIdFASTN**	**PRIdPTR**	**PRIdMAX**
printf格式	**PRIiN**	**PRIiLEASTN**	**PRIiFASTN**	**PRIiPTR**	**PRIiMAX**
无符号	**PRIoN**	**PRIoLEASTN**	**PRIoFASTN**	**PRIoPTR**	**PRIoMAX**
printf格式	**PRIuN**	**PRIuLEASTN**	**PRIuFASTN**	**PRIuPTR**	**PRIuMAX**
	PRIxN	**PRIxLEASTN**	**PRIxFASTN**	**PRIxPTR**	**PRIxMAX**
	PRIXN	**PRIXLEASTN**	**PRIXFASTN**	**PRIXPTR**	**PRIXMAX**
带符号	**SCNdN**	**SCNdLEASTN**	**SCNdFASTN**	**SCNdPTR**	**SCNdMAX**
scanf格式	**SCNiN**	**SCNiLEASTN**	**SCNiFASTN**	**SCNiPTR**	**SCNiMAX**
无符号	**SCNoN**	**SCNoLEASTN**	**SCNoFASTN**	**SCNoPTR**	**SCNoMAX**
scanf格式	**SCNuN**	**SCNuLEASTN**	**SCNuFASTN**	**SCNuPTR**	**SCNuMAX**
469	**SCNxN**	**SCNxLEASTN**	**SCNxFASTN**	**SCNxPTR**	**SCNxMAX**

21.2 准确长度类型

语法概要

```
#include <stdint.h>                    // All C99
typedef ... intN_t
typedef ... uintN_t
#define  INTN_MIN   -2^(N-1)
#define  INTN_MAX   2^(N-1)-1
#define UINTN_MAX   2^N-1

#include <inttypes.h>
#define PRIcN "..."
#define SCNcN "..."
```

这些类型和宏定义的整数类型具有准确长度，没有填充位。...**MIN**与...**MAX**宏要显示准确值。

这些类型在**stdint.h**中是可选的，但如果实现恰好有8、16、32或64位的宽度，则要定义相应的类型和宏。实现还可以定义其他准确长度。

例 下列定义在许多字节寻址计算机上的C语言实现中可以遇到：

```
#include <limits.h> /* SCHAR_MIN, SCHAR_MAX, UCHAR_MAX */
typedef signed char              int8_t;
typedef unsigned char uint8_t;
typedef short int16_t;
typedef unsigned short         uint16_t;
typedef int int32_t;
typedef unsigned int           uint32_t;
typedef long long int int64_t;
typedef unsigned long long int uint64_t;
#define INT8_MIN SCHAR_MIN
#define INT8_MAX SCHAR_MAX
#define UINT8_MAX UCHAR_MAX
#define PRId8 "hhd"
#define SCNo64 "llo"
// etc.
```

随着今后计算机字长的增加，**long**可能称为**int64_t**，**longlongint**可能称为**int128_t**。 □ 470

21.3 最小长度类型

语法概要

```
#include <stdint.h>                    // All C99
typedef ...  int_leastN_t
typedef ... uint_leastN_t
#define  INT_LEASTN_MIN  -(2^(N-1)-1)
#define  INT_LEASTN_MAX  2^(N-1)-1
#define UINT_LEASTN_MAX  2^N-1
```

```
#define  INTN_C(constant)  ...
#define UINTN_C(constant)  ...

#include <inttypes.h>
#define PRIcLEASTN "..."
#define SCNcLEASTN "..."
```

这些类型和宏定义的整数类型是具有指定最小长度的最小值。...**MIN**与...**MAX** 宏要与所示值具有相同符号，至少有相同的数量级。由于这些类型是指定长度的最小类型，因此如果一定的N有准确长度类型（21.1节），则准确长度类型也应是相同*N*值的最小长度类型。

所有C99实现都要对*N*=8、16、32和64定义这些类型和宏。其他*N*值的定义是可选的，但如果提供，则要定义该*N*值的所有类型和宏。

例 32位字寻址计算机的C语言实现可能把**char**、**short**与**int**都定义为32位类型。这时准确长度类型**int8_t**与**int16_t**（及对应的无符号类型）不定义，而最小长度类型**int8_t**与**int16_t**定义为一个32位类型，如**int**。 □

INT*N***_C**宏带一个参数，是十进制、十六进制或八进制常量，扩展为**int_least***N***_t**类型的带符号整型常量，具有相同值。**UINT***N***_C**宏扩展成**uint_least***N***_t**类型的无符号整型常量。宏在常量后面增加适当的后缀字母。

例 如果**int_least64_t**定义为**long long int**，则**INT64_C(1)**为**1LL**，而
471 **UINT64_C(1)**为**1ULL**。 □

21.4 快速长度类型

语法概要

```
#include <stdint.h>                    // All C99
typedef ...  int_fastN_t
typedef ... uint_fastN_t
#define  INT_FASTN_MIN  -(2^(N-1)-1)
#define  INT_FASTN_MAX  2^(N-1)-1
#define UINT_FASTN_MAX  2^N-1

#include <inttypes.h>
#define PRIcFASTN "..."
#define SCNcFASTN "..."
```

这些类型和宏定义的整数类型是具有指定最小长度的最快类型。...**MIN**与...**MAX** 宏要与所示值具有相同符号，至少有相同的数量级。所有C99实现都要对*N*=8、16、32和64定义这些类型和宏。其他*N*值的定义是可选的，但如果提供，则要定义该*N*值的所有类型和宏。

要确定哪个类型最快，可能要实现者进行判断，不一定对一个类型的所有用途都最快。例如，标量算术的最快类型不一定是访问数组元素时的最快类型。

例 在针对32位算术进行优化的字节寻址计算机上，C语言实现可能选择在需要更少位时也推荐32位类型。下面是**stdint.h**中的一组定义，本例中只显示带符号类型：

```
typedef char int8_t;
typedef char int_least8_t;
typedef int int_fast8_t;

typedef short int16_t;
typedef short int_least16_t;
typedef int int_fast16_t;

typedef int int32_t;
typedef int int_least32_t;
typedef int     int_fast32_t;
```

□ 472

21.5 指针长度类型与最大长度类型

语法概要

```
#include <stdint.h>                    // All C99
typedef ... intptr_t;
typedef ... uintptr_t;
#define  INTPTR_MIN  -(2^15-1)
#define  INTPTR_MAX  2^15-1
#define UINTPTR_MA X 2^16-1

typedef ...  intmax_t;
typedef ... uintmax_t;
#define  INTMAX_MIN  -(2^63-1)
#define  INTMAX_MAX  2^63-1
#define UINTMAX_MA X 2^64-1
#define  INTMAX_C(constant) ...
#define UINTMAX_C(constant) ...

#include <inttypes.h>
#define PRIcPTR "..."
#define SCNcPTR "..."
#define PRIcMAX "..."
#define SCNcMAX "..."
```

类型**intptr_t**与**uintptr_t**分别是带符号和无符号整型类型，可以放置任何对象指针。如果*P*是**void ***类型的值，则P可以转换成**intptr_t**与**uintptr_t** ，然后再转换回**void ***，结果是原先的指针*P*。...**MIN**与...**MAX** 宏要与所示值具有相同符号，至少有相同的数量级。这些类型是可选的，因为也许没有这种整型类型（但通常有）。

类型**intmax_t**与**uintmax_t**分别是实现定义的最大带符号和无符号整型类型，所有C语言实现都要定义。由于C99实现允许提供扩展整型类型，因此**intmax_t**类型不一定是**longlongint**之类的标准C语言类型。...**MIN**与...**MAX** 宏要与所示值具有相同符号，至少有相同的数量级。

INTMAX_C宏带有一个参数，是十进制、十六进制或八进制常量，扩展为**intmax_t**类型的

带符号整型常量，具有相同值。**UINTMAX_C**宏扩展**uintmax_t**类型的无符号整型常量。

473 **参考章节** **limits.h** 表5-2

21.6 ptrdiff_t、size_t、wchar_t、wint_t与sig_atomic_t的范围

语法概要

```
#include <stdint.h>
#define PTRDIFF_MIN     ...              // All C99
#define PTRDIFF_MAX     ...
#define SIZE_MAX        ...
#define WCHAR_MIN       ...
#define WCHAR_MAX       ...
#define WINT_MIN        ...
#define WINT_MAX        ...
#define SIG_ATOMIC_MIN  ...
#define SIG_ATOMIC_MAX  ...
```

本节介绍的宏扩展成**stddef.h**与**wchar.h**中定义的不同类型数字范围的预处理器常量表达式。所有实现都要定义。

PTRDIFF_MIN与**PTRDIFF_MAX**指定**ptrdiff_t**类型范围，应为至少16位的带符号类型。

SIZE_MAX是**size_t**类型可以表示的最大值。

WCHAR_MIN与**WCHAR_MAX**指定**wchar_t**类型范围，应为至少8位的带符号或无符号类型。

WINT_MIN与**WINT_MAX**指定**wint_t**类型范围，应为至少16位的带符号或无符号类型。

SIG_ATOMIC_MIN与**SIG_ATOMIC_MAX**指定**sig_atomic_t**类型范围，应为至少8位的带符号或无符号类型。

参考章节 **ptrdiff_t** 11.1; **sig_atomic_t** 19.6; **size_t** 11.1; **wchar_t** 24.1; **wint_t** 24.1

21.7 imaxabs、imaxdiv、imaxdiv_t

语法概要

```
#include <inttypes.h>
typedef ... imaxdiv_t;                          // All C99
intmax_t imaxabs( intmax_t x );
imaxdiv_t imaxdiv( intmax_t n, intmax_t d );
```

本节介绍的函数支持最大长度整型类型的基本算术，类似于**stdlib.h**中定义的**abs**与**div**

474 函数。**imaxabs**函数计算参数的绝对值，如果绝对值无法表示，则结果是未定义的。

imaxdiv函数在一次运算中同时计算**n/d**与**n%d**，结果分别存放在**quot**与**rem**成员中，这是个**imaxdiv_t**类型结构。**imaxdiv_t**中成员的顺序没有指定。

参考章节 **abs** 16.9; **div** 16.9

21.8 strtoimax、strtouimax

语法概要

```
#include <inttypes.h>
intmax_t strtoimax(
  const char * restrict str,
  char ** restrict ptr,
  int base);
uintmax_t strtoumax(
  const char * restrict str,
  char ** restrict ptr,
  int base);
```

这些函数将字符串转换成最大长度整数，与**stdlib.h**中的**strtol**与**strtoul**函数相似。如果结果造成溢出，则返回**INTMAX_MAX**、**INTMAX_MIN**或**UINTMAX_MAX**之一，并将**errno**设置为**ERANGE**。

参考章节 **errno**与**ERANGE** 11.2；**strtol**与**strtoul** 16.4

21.9 wcstoimax、wcstoumax

语法概要

```
#include <stddef.h>     // wchar_t
#include <inttypes.h>
intmax_t wcstoimax(
  const wchar_t * restrict str,
  wchar_t ** restrict ptr,
  int base);
uintmax_t wcstoumax(
  const wchar_t * restrict str,
  wchar_t ** restrict ptr,
  int base);
```

这些函数将宽字符串转换成最大长度整数，与**wchar.h**中的**wcstol**与**wcstoul**函数相似。如果结果造成溢出，则返回**INTMAX_MAX**、**INTMAX_MIN**或**UINTMAX_MAX**之一，并将**errno**设置为**ERANGE**。 475

参考章节 **errno**与**ERANGE** 11.2；**wcstol**与**wcstoul** 第24章 476

第22章 浮点数环境

本章介绍的函数是C99增加的，补充了**float.h**中的信息，可以对需要高度控制浮点数运算精度或性能的应用程序提供对浮点数环境的访问。这些函数在头文件**fenv.h**中提供。

参考章节 **float.h** 表5-3

22.1 概述

编写高精度浮点数运算的编程人员需要控制浮点数环境的各个方面：结果如何舍入，浮点数表达式如何简化与变换，下溢之类的浮点数事件是忽略还是产生程序错误。控制的方法是设置浮点数控制方式，可以影响浮点数运算的运行方法。运算通过浮点数异常向编程人员提供反馈，可以在C语言程序中中断控制流，并记录状态标志，以便编程人员阅读。C99编程人员还可以用第17章列出的专门浮点数数学函数控制浮点数行为。

浮点数运算可以在两个时间进行。C语言程序编译时，进行常量（编译时）浮点数运算，而C语言程序运行时，可以进行动态（执行时）浮点数运算。C99标准只对运行时的运算提供显式控制。实现可以提供自己的函数，控制编译时的运算。

C99引用的国际浮点数标准是IEC 60559:989，即“微处理器系统的二进制浮点数算术”（第2版）。这个标准原先的指定是IEC 559:1989与ANSI/IEEE 754-1985，即“IEEE二进制浮点数
477 算术标准”（IEEE 754后来进行了一般化，取消了ANSI/IEEE 854-1987，即“IEEE进制独立浮点数算术标准”中对进制与字长的依赖）。C99标准附录F详细介绍了C语言与IEC 60559的映射，这是可选的，除非C语言实现定义**__STDC_IEC_559__**宏。

编程规则

控制浮点数行为的函数是动态的，即一旦程序执行过程中被本章介绍的函数修改之后，这个改变一直保持到另一次显式地修改。特定函数如何进行浮点数运算取决于最近调用的是**fenv.h**中的哪个函数，因此在C语言程序编译时无法确定。如果底层硬件用全局控制寄存器控制浮点数算术，则这样不成问题；但如果编译器发出实际操作码来控制行为，则会更难实现。

C99标准建议编程人员总是假设函数的任何调用都得到缺省浮点数行为，除非已经证明不是这样。同样，被调函数不能改变环境，除非已经证明它们可以。函数不能依赖于任何状态标志，也不能改变调用时使用的标志。如果需要，可以预计实行的是缺省控制方式，不能改变调用者的方式。任何函数都可能产生浮点数异常。

22.2 浮点数环境

语法概要

```
#include <fenv.h>
#pragma STDC FENV_ACCESS   on-off-switch
typedef ... fenv_t;
```

```
#define FE_DEFL_ENV ...
int fegetenv(fenv_t *envp);
int fesetenv(fenv_t *envp);
int feholdexcept(fenv_t *envp);
int feupdateenv(const fenv_t *envp);
```

标准杂注**FENV_ACCESS**表示C语言程序是否设置浮点数控制方式，测试状态标志或在非缺省控制方式中运行。**FENV_ACCESS**设置为关闭时，这些操作的行为是未定义的。这个杂注适用于这种状态对C语言程序的编译与优化有重大影响时。这个杂注的缺省设置是由实现定义的，因此需要移植性的编程人员应假设它是关闭的。**FENV_ACCESS**杂注符合标准杂注的正常放置规则。

fenv_t类型是由实现定义的，保存整个浮点数状态，包括控制方式和异常状态位。

FE_DEFL_ENV宏扩展成用**fenv_t***类型值指定缺省浮点数环境。C语言实现可能定义其他 478
FE_开头加上大写字母的宏。编程人员应把这些宏看成指定只读对象。

fegetenv函数获得当前浮点数环境并将其存放在**envp**指定的对象中。如果成功，则返回0，否则返回一个非0值。

fesetenv函数将当前浮点数环境换成**envp**所指的环境。这个环境要先用**fegetenv**或**feholdexcept**设置，或应为**FE_DEFL_ENV**之类的预定义环境。如果成功，则返回0，否则返回一个非0值。

feholdexcept函数通常用于在一定时间内关掉浮点数异常。函数把当前浮点数环境存放在**envp**指定的对象中，然后安装一个忽略所有浮点数异常的环境。如果这种“不停止”环境顺利安装，则函数返回0，否则返回一个非0值。一些实现可能无法忽略所有浮点数异常。

feupdateenv函数将当前发出的浮点数异常保存在某个本地存储体中，将**envp**所指的环境保存为新环境，最后产生保存的异常。如果成功，则返回0，否则返回一个非0值。

参考章节 杂注与放置规则 3.7；产生浮点数异常 22.3

22.3 浮点数异常

语法概要

```
#include <fenv.h>
macro FE_DIVBYZERO ...
macro FE_INEXACT ...
macro FE_INVALID ...
macro FE_OVERFLOW ...
macro FE_UNDERFLOW ...
...
macro FE_ALL_EXCEPT ...
typedef ... fexcept_t;
int fegetexceptflag(fexcept_t *flagp, int excepts);
int fesetexceptflag(const fexcept_t *flagp, int excepts);
int fetestexcept(int excepts);
int feraiseexcept(int excepts);
int feclearexcept(int excepts);
```

浮点数异常是某些浮点数运算的副作用。所有异常都设置一个状态标志，表示发生了异常。

[479] 异常是否中断程序的控制流程取决于设置的浮点数控制方式。

fexcept_t类型是由实现定义的，保存实现支持的所有浮点数状态标志，通常是整数类型，其位表示不同异常，但也可以更复杂。例如，**fexcept_t**可以保存发出状态标志的位置信息。

C语言实现可能支持不同的浮点数异常。对每个支持的异常，实现要定义**FE_DIVBYZERO**，**FE_INEXACT**、**FE_INVALID**、**FE_OVERFLOW**与**FE_UNDERFLOW**之类的宏。不支持的异常要保持未定义。每个定义的宏扩展为整型常量表达式，这些值应能够通过按位或运算合并起来，表示任何异常子集。通常，每个宏扩展成2的不同指数。**FE_ALL_EXCEPT**宏是所有被支持的异常的按位或，从本节的函数语法可以看出，异常的个数不能超过int类型的位数，其至少包含16位。

fegetexceptflag函数将当前浮点数状态标志设置存储在**flagp**所指的对象中。并不是所有状态标志都存储在**flagp***中，只有**excepts**参数中列出的异常才进行设置，其他异常则在**flagp***中保持不变。**excepts**参数是“有意义”的异常的掩码。如果成功，则返回0，否则返回一个非0值。

fesetexceptflag函数将当前浮点数状态标志设置为**flagp**所指对象中存放的值。并不是所有状态标志都设置，只有**excepts**参数中列出的异常才进行设置，其他异常则在**flagp***中保持不变。**excepts**参数是“有意义”的异常的掩码。如果所有指定的标志都设置为相应状态，则函数返回0，否则返回一个非0值。

fetestexcept函数返回相应于异常标志的异常宏的按位或的运算结果，其在当前环境中设置，在**excepts**参数中存在。这样，**fetestexcept**返回**excepts**中当前设置的异常子集。

feraiseexcept函数产生**excepts**参数中表示的异常。产生异常的顺序没有指定，有些异常的副作用可能是产生另一些异常。例如，**FE_INEXACT**通常和其他异常组合。

feclearexcept函数清除对应于**excepts**中所表示异常的当前异常状态标志。如果
[480] **excepts**中的所有异常都清除了，则函数返回0，否则返回一个非0值。

22.4 浮点数舍入方式

语法概要

```
#include <fenv.h>

macro FE_DOWNWARD ...
macro FE_UPWARD ...
macro FE_TONEAREST ...
macro FE_TOWARDZERO ...

int fegetround(void);
int fetestround(int rounds);
```

C99实现要定义**FE_DOWNWARD**、**FE_UPWARD**、**FE_TONEAREST**与**FE_TOWARDZERO**之类的宏以适应本节的函数可以设置或读取的每个舍入方向。宏扩展成不同的非负整型常量表达式，对应于**int**类型值。不支持的舍入方向不定义相应宏。

fegetround函数返回当前舍入方向，表示为一个舍入方向宏值。同样，**fesetround**函
[481] 数设置舍入方向，并在成功时返回0。如果舍入方向设置或读取失败，则返回负值。

第23章 复数算术

本章介绍的函数支持复数算术，在C99头文件**complex.h**中定义。

23.1 复数库规则

所有角度的单位为弧度。复数也可以写成$x+yi$，其中x和y是实数。同样，$w=u+vi$，$c=a+bi$。

由于复数函数的分支不连续，因此应采用下列实现定义规则。如果实现具有带符号0，则0的符号区别分支的两端，否则库实现要处理断点，使逆时针沿有限端到达断点时，函数连续。

参考章节 复数类型 5.2.1

23.2 complex、_Complex_I、imaginary、_Imaginary_I、I

语法概要

```
#include <complex.h>                          // All C99
#define complex _Complex
#define imaginary _Imaginary
#define _Complex_I    ...
#define _Imaginary_I  ...
#define I ...
```

如果支持复数类型，则定义**complex**宏为关键字**_Complex** 的同义词；如果支持虚数类型，则定义**imaginary**宏为关键字**_Imaginary**的同义词。如果支持相应类型，则**_Complex_I**与**_Imaginary_I**宏定义为**const float_Complex**与**const float_Imaginary**类型的常量表达式，取值为虚数单位i，或$\sqrt{(-1)}$。

如果支持复数类型，则宏**I**扩展为**_Complex_I**；如果支持虚数类型，则**I**也可以扩展成**_Imaginary_I**。

由于标识符**complex**、**imaginary**和**I**可能在用C99之前的标准编写的程序中使用，因此可以使用**#undefine**并重新定义这些宏。

参考章节 复数类型 5.2.1

23.3 CX_LIMITED_RANGE

语法概要

```
#include <complex.h>                          // All C99
#pragma STDC CX_LIMITED_RANGE on-or-off-switch
```

标准杂注**CX_LIMITED_RANGE**打开时，告诉实现可以使用复数乘法、除法与绝对值的“明显”实现版本。这个杂注的缺省状态为关闭。杂注**CX_LIMITED_RANGE**符合标准杂注的放置规则。“明显”实现版本如下：

乘法：$z*w=(x+iy)(u+iv)=(xu-yv)+i(yu+xv)$

除法：$z/w=(x+iy)(u+iv)=((xu+yv)+i(yu-xv))/(u^2+v^2)$

绝对值：$|z|=|x+iy|=\sqrt{(x^2+y^2)}$

这些实现版本是不完善的，因为可能出现不必要的上溢与下溢，也不能很好地处理无限大。但是，如果编程人员知道它们在当前程序中是安全的，则使用它们可能速度较快。

484 **参考章节** 标准杂注 3.7

23.4 cacos、casin、catan、ccos、csin、ctan

语法概要

```
#include <complex.h>                          // All C99

double complex cacos (double complex z);
float complex cacosf(float complex z);
long double complex cacosl(long double complex z);

double complex casin (double complex z);
float complex casinf(float complex z);
long double complex casinl(long double complex z);

double complex catan (double complex z);
float complex catanf(float complex z);
long double complex catanl(long double complex z);

double complex ccos (double      complex z);
float complex ccosf(float complex z);
long double complex caosl(long double complex z);

double complex csin (double      complex z);
float complex csinf(float complex z);
long double complex csinl(long double complex z);

double complex ctan (double      complex z);
float complex ctanf(float complex z);
long double complex ctanl(long double complex z);
```

表23-1列出了函数的定义域与值域，这里$(a+bi)=f(x+yi)$。

表23-1 复数三角函数的定义域与值域

C语言函数名	函　数	分支断点	值　域
cacos	复数反余弦	$y=0, x>+1$与$y=0, x<-1$	$0\leqslant a\leqslant\pi$
casin	复数反正弦	$y=0, x>+1$与$y=0, x<-1$	$-\pi/2\leqslant a\leqslant+\pi/2$
catan	复数反正切	$x=0, y>+1$与$x=0, y<-1$	$-\pi/2\leqslant a\leqslant+\pi/2$
ccos	复数余弦		
csin	复数正弦		
ctan	复数正切		

485

23.5 cacosh、casinh、catanh、ccosh、csinh、 ctanh

语法概要

```
#include <complex.h>                          // All C99
```

```
double complex cacosh (double complex z);
float complex cacoshf(float complex z);
long double complex cacoshl(long double complex z);

double complex casinh (double complex z);
float complex casinhf(float complex z);
long double complex casinhl(long double complex z);

double complex catanh (double complex z);
float complex catanhf(float complex z);
long double complex catanhl(long double complex z);

double complex ccosh (double complex z);
float complex ccoshf(float complex z);
long double complex ccoshl(long double complex z);

double complex csinh (double complex z);
float complex csinhf(float complex z);
long double complex csinhl(long double complex z);

double complex ctanh (double complex z);
float complex ctanhf(float complex z);
long double complex ctanhl(long double complex z);
```

表23-2列出了函数的定义域与值域，这里$(a+bi)=f(x+yi)$。

表23-2 复数双曲函数的定义域与值域

C语言函数名	函 数	分支断点	值 域
cacosh	复数双曲反余弦	$y=0, x<+1$	$0\leqslant a, -\pi\leqslant b\leqslant+\pi$
casinh	复数双曲反正弦	$x=0, y>+1$与$x=0, y<-1$	$-\pi/2\leqslant b\leqslant+\pi/2$
catanh	复数双曲反正切	$y=0, x>+1$与$y=0, x<-1$	$-\pi/2\leqslant b\leqslant+\pi/2$
ccosh	复数双曲余弦		
csinh	复数双曲正弦		
ctanh	复数双曲正切		

486

23.6 cexp、clog、cabs、cpow、csqrt

语法概要

```
#include <complex.h>                    // All C99

double      complex cexp (double complex z);
float       complex cexpf(float       complex z);
long double complex cexpl(long double complex z);

double      complex clog (double complex z);
float       complex clogf(float       complex z);
long double complex clogl(long double complex z);

double              cabs (double complex z);
float               cabsf(float       complex z);
long double cabsl(long double complex z);

double      complex cpow (
    double complex z,
    double complex u);
```

```
float       complex cpowf(
     float complex z,
     float complex u);
long double complex cpowl(
     long double complex z,
     long double complex u);
double      complex csqrt (double complex z);
float       complex csqrtf(float complex z);
long double complex csqrtl(long double complex z);
```

表23-3列出了函数的定义域与值域，这里

$(a+bi)=f(z)=f(x+yi)$或

$(a+bi)=f(z,w)=f(x+yi,u+vi)$。

表23-3 复数指数函数的定义域与值域

C语言函数名	函　数	分支断点	值　域
cexp	e^z		
clog	lnz	$y=0, x<0$	$-\pi \leqslant b \leqslant +\pi$
cabs	绝对值 $a=sqrt(x^2+y^2)$		
cpow	z^w	$y=0, x<0$	
csqrt	平方根	$y=0, x<0$	

487

23.7 carg、cimag、creal、conj、cproj

语法概要

```
#include <complex.h>                          // All C99
double      carg (double complex z);
float       cargf(float       complex z);
long double cargl(long double complex z);
double      cimag (double complex z);
float       cimagf(float       complex z);
long double cimagl(long double complex z);
double      creal (double complex z);
float       crealf(float       complex z);
long double creall(long double complex z);
double complex conj (double      complex z);
float complex conjf(float complex z);
long double complex conjl(long double complex z);
double complex cproj (double complex z);
float complex cprojf(float complex z);
long double complex cprojl(long double complex z);
```

表23-4列出了函数的定义域与值域，这里

$(a+bi)=f(x+yi)$或

$(a+bi)=f(x+yi,u+vi)$

表23-4 其他复数函数的定义域与值域

C语言函数名	函 数	分支断点	值 域
carg	参数（也称为象限角）	$y=0, x<0$	$[-\pi,+\pi]$
cimag	z:y的虚数部分		
creal	z:x的实数部分		
conj	$a=x$,$b=-y$		
cproj	Riemann球上的投影		

carg(z)的值是复平面上从正实轴到从原点向**z**所画直线的夹角。

cproj(z)值在**z**不是无限大时为**z**。如果**z**是无限大，则**cproj(z)**为无限大的正实数，表示为复数。如果实现支持带符号0，而**z**是无限大，则**cproj(z)**的虚数部分与**z**的虚数部分具有相同符号。要让**z**为无限大，只要其中一个成员为无限大，即使另一个成员为NaN也没关系。 488

第24章 宽字符与多字节函数

本章介绍的函数支持多字节字符与字符串、宽字符与字符串。字符分类和映射函数放在头文件**wctype.h**中，其余字符与字符串函数放在头文件**wchar.h**中。通常，这些函数与**ctype.h**、**string.h**与**stdio.h**中的传统字符与字符串函数相似，只是以明显方式改变参数和返回类型。

24.1 基本类型和宏

语法概要

```
#include <wchar.h>
typedef ... wchar_t;
typedef ... wint_t;
typedef ... mbstate_t;
typedef ... size_t;

#define WEOF ...
#define WCHAR_MIN ...
#define WCHAR_MAX ...
```

类型**wchar_t**（宽字符类型）是整型，可以表示所支持区域设置中任何执行时扩展字符集的所有不同值。它可能是带符号或无符号类型，也在**stddef.h**中定义。**WCHAR_MIN**与**WCHAR_MAX**宏指定**wchar_t**类型的数字边界，它们的值不能对应某个扩展字符。

489

wint_t类型也是整型，保存**wchar_t**类型的所有值和至少一个不在扩展字符集中的值。这个常量值由**WEOF**指定，表示输入结束和其他异常条件。**wint_t**类型不能在普通参数升级时改变。

mbstate_t类型是个非数组对象类型，表示多字节字符序列与宽字符串之间的转换状态。

size_t与**stddef.h**中定义的类型相同。

参考章节 **size_t** 11.1；**wchar_t** 11.1；宽字符 2.7.3

24.2 多字节字符与宽字符的转换

语法概要

```
#include <wchar.h>
size_t mbrlen(const char *s, size_t n, mbstate_t *ps);
wint_t btowc(int c);
size_t mbrtowc(wchar_t *pwc, const char *s, size_t n,
    mbstate_t *ps);
int wctob(wint_t c);
size_t wcrtomb(char *s, wchar_t wc, mbstate_t *ps);
int mbsinit(const mbstate_t *ps);
```

本节介绍的转换函数都是**stdlib.h**头文件中定义的基本函数**mblen**、**mbtowc**与**wctomb**

（见16.10节）的扩展版本。这些函数是C89增补1增加的，更加灵活、其行为说明更加完整。

mbrlen函数检查**s**指定的字符串中最多**n**个字节，看这些字符是否表示与**ps**中转换状态相应的有效多字节字符。如果**ps**为null，则函数使用自己的内部状态对象，其在程序启动时初始化为初始状态。如果**s**为null指针，则调用时就像**s**为""、**n**为1一样。如果**s**有效，对应于null宽字符，则返回0（不管多字节字符由多少字节构成）。如果**s**是任何其他有效多字节字符，则返回构成字符的字节数（即返回的值在1到**n**之间）。如果**s**是不完整多字节字符，则返回－2。如果**s**是无效多字节字符，则返回－1，并将**errno**设置为**EILSEQ**。返回值为非负时，转换状态更新，返回值为－1时，转换状态未定义，而返回值为－2时，转换状态不变。

btowc函数返回对应于字节**c**的宽字符，这个宽字符作为初始转换状态的单字节多字节字符。如果**c**（转换成**unsigned char**）不对应于有效多字节字符或**c**是**EOF**，则**btowc**返回**WEOF**。

mbrtowc函数将多字节字符**s**转换成对应于转换状态**ps**的宽字符（如果**ps**为null，则使用内部状态对象，其在程序启动时初始化为初始状态）。如果**pwc**不是null指针，则结果存放在**pwc**指定的对象中。如果**s**为null指针，则调用 **mbrtowc**等价于 **mbrtowc(NULL,"",1,ps)**，即 490
把**s**当作空字符串，忽略**pwc**与**n**值。如果**s**有效，对应于null宽字符，则返回0（不管多字节字符由多少字节构成）。如果**s**是任何其他有效多字节字符，则返回构成字符的字节数。如果**s**是不完整多字节字符，则返回－2。如果**s**是无效多字节字符，则返回－1。**ps**指定的转换状态（或**ps**为null指针时的内部转换状态）在发生有效转换时更新。如果**s**不完整，则转换状态不变，如果**s**无效，则转换状态未定义。

wctob函数（C89增补1）返回对应于初始转换状态宽字符**c**的单字节多字节字符。如果没有这种单字节，则返回**EOF**。

wcrtomb函数将宽字符**wc**转换成对应于转换状态**ps**的多字节字符（如果**ps**为null，则使用内部状态对象，其在程序启动时初始化为初始状态）。多字节字符存放在数组中，其第一个元素由**s**指定，至少应有**MB_CUR_MAX**个字符。转换状态更新。如果**wc**是null宽字符，则存储null字节，前面加上恢复初始转换状态所要的任何shift序列。函数返回**s**中存放的字符数。如果**s**是null指针，则忽略**wc**，调用**wcrtomb**的效果就是恢复初始转换状态和返回1（就像把**L'\0'**转换到隐藏缓冲区中）。如果**wc**不是有效宽字符，则**errno**存放**EILSEQ**，并返回－1。

如果**ps**为null或指向表示初始转换状态的对象，则**mbsinit**函数返回非0值，否则返回0。

参考章节 **EILSEQ** 11.2；**errno** 11.2；多字节字符 2.1.5；**mbstate_t** 11.1；**size_t** 11.1；**wchar_t** 11.1；**wint_t** 11.1

24.3 宽字符串与多字节字符串的转换

语法概要

```
#include <wchar.h>
size_t mbsrtowcs(wchar_t *pwcs, const char **src, size_t n,
    mbstate_t *ps);
size_t wcsrtombs(char *s, const wchar_t **src, size_t n,
    mbstate_t *ps);
```

本节的函数是**mbstowcs**与**wcstombs**的“可重启动”版本，在**stdlib.h**中定义（见16.11节）。这些函数是C89增补1增加的。

mbsrtowcs函数将以null终止的字符串**s**中的一系列多字节字符转换成相应宽字符序列，将结果存放在**pwcs**指定的数组中。**ps**指定初始转换状态，**src**间接指定输入多字节字符序列。在
491 正常操作中，到null终止字符为止的每个多字节字符像调用**mbrtowc**函数一样一一转换，输出宽字符放在**pwcs**指定的字符数组中。转换之后，**src**所指的指针设置为null指针，表示已经转换整个输入字符串，并返回**pwcs**中存储的宽字符数（不包括null终止字符）。转换状态更新为初始shift状态，这是转换输入多字节字符串末尾的null字符得到的。输出指针**pwcs**可能是个null指针，这时**mbsrtowcs**只是计算转换所需的输出宽字符串长度。

如果发生转换错误，则转换输入多字节字符串也会提前停止。这时**src**所指定的指针更新为指向发生转换错误的多字节字符。函数返回 -1，**errno**中存放**EILSEQ**，转换状态不确定。

wcsrtombs函数将以**pwcs**指定值开始的宽字符序列转换成多字节字符序列，将结果存放在**s**指定的字符数组中。**ps**指定初始转换状态，**src**间接指定输入宽字符序列。在正常操作中，到null终止宽字符为止的每个宽字符像调用**wcrtomb**函数一样一一转换，输出宽字符放在**pwcs**指定的字符数组中。转换之后，**src**所指的指针设置为null指针，表示已经转换整个输入字符串，并返回**pwcs**中存储的宽字符数（不包括null终止字符）。转换状态更新为初始shift状态，这是转换输入宽字符串末尾的null字符得到的。输出指针**pwcs**可能是个null指针，这时**wcsrtombs**只是计算转换所需的输出宽字符串长度。

如果已经在**s**中写入**n**个输出字节（而**s**不是null指针），则输入宽字符串的转换会在转换null终止宽字符之前停止。这时**src**指定的指针设置成指向最后一个转换的宽字符之后，转换状态更新（通常更新为初始状态，但也不一定）并返回n。

如果发生转换错误，则转换输入宽字符串也会提前停止。这时**src**所指定的指针更新为指向发生转换错误的宽字节字符。函数返回 -1，**errno**中存放**EILSEQ**，转换状态不确定。

492 **参考章节**　转换状态 2.1.5；多字节字符 2.1.5；宽字符 2.1.5

24.4 转换成算术类型

语法概要

```
#include <wchar.h>

double wcstod(
  const wchar_t * restrict str,
  wchar_t ** restrict ptr );
float wcstof(
  const wchar_t * restrict str,
  wchar_t ** restrict ptr );
long double wcstold(
  const wchar_t * restrict str,
  wchar_t ** restrict ptr );

long wcstol(
  const wchar_t * restrict str,
  wchar_t ** restrict ptr, int base );
long long wcstoll(
  const wchar_t * restrict str,
  wchar_t ** restrict ptr, int base );
```

```
unsigned long wcstoul(
  const char * restrict str,
  wchar_t ** restrict ptr, int base );
unsigned long strtoull(
  const char * restrict str,
  wchar_t ** restrict ptr, int base );
```

本节的**wcsto**...函数与16.4节的相应**strto**...函数相似，只是参数类型不同，用**iswspace**函数探测空白符和用小数点宽字符代替点号。这些宽字符串转换函数可以接受**strto**...接受的字符串，还可以接受由实现定义的输入字符串。

函数**wcstod**、**wcstol**与**wcstoul**是C89增补1增加的，其余是C99增加的。

24.5 输入与输出函数

表24-1列出了宽字符串输入与输出函数及相应字节输入与输出函数，同时还列出了本书介绍字节与宽字符函数的章节。

24.6 字符串函数

表24-2列出了宽字符串函数及相应字节函数，同时还列出了本书介绍字节字符串与宽字符串函数的章节。

493

表24-1 宽字符输入与输出函数

宽字符函数	参考章节	字节字符函数
fgetwc	15.6	fgetc
fgetws	15.7	fgets
fputwc	15.9	fputc
fputws	15.10	fputs
fwide	15.2	
fwprintf	15.11	fprintf
fwscanf	15.8	fscanf
getwc	15.6	getc
getwchar	15.6	getchar
putwc	15.9	putc
putwchar	15.9	putchar
swprintf	15.11	sprintf
swscanf	15.8	sscanf
ungetwc	15.6	ungetc
vfwprintf	15.12	vfprintf
vfwscanf	15.12	vfscanf
vswprintf	15.12	vsprintf
vswscanf	15.12	vsscanf
vwprintf	15.12	vprintf
vwscanf	15.8	vscanf
wprintf	15.11	printf
wscanf	15.8	scanf

24.7 日期与时间转换

wcsftime宽字符函数对应于**strftime**字节函数。

参考章节 **strftime** 18.6

24.8 宽字符分类函数与映射函数

表24-3列出宽字符分类与映射函数及相应字符函数，同时还列出了本书介绍这些函数的章节。

宽字符函数**towctrans**没有相应的字符函数，其语法如下：

```
#include <wctype.h>
wint_t towctrans( wint_t wc, wctrans_t desc );
```

towctrans函数将宽字符**wc**映射为一个新值，并返回新值。映射由**wctrans_t**类型的值指定，可以用**wctrans**函数取得（12.11节）。调用**towctrans**时的**LC_CTYPE**区域设置类别应和调用**wctrans_t**时相同，产生**desc**值。

表24-2 宽字符串函数

宽字符串函数	参考章节	字节字符串函数
wcscat	13.1	strcat
wcschr	13.5	strchr
wcscmp	13.2	strcmp
wcscoll	13.10	strcoll
wcscpy	13.3	strcpy
wcscspn	13.6	strcspn
wcslen	13.4	strlen
wcsncat	13.1	strncat
wcsncmp	13.2	strncmp
wcsncpy	13.3	strncpy
wcspbrk	13.6	strpbrk
wcsrchr	13.5	strrchr
wcsspn	13.6	strspn
wcsstr	13.7	strstr
wcstok	13.7	strtok
wcsxfrm	13.10	strxfrm
wmemchr	14.1	memchr
wmemcmp	14.1	memcmp
wmemcpy	14.3	memcpy
wmemmove	14.3	memmove
wmemset	14.4	memset

表24-3 宽字符函数

宽字符函数	参考章节	字节字符函数
iswalnum	12.1	isalnum
iswalpha	12.1	isalpha
iswblank	12.	isblank
iswcntrl	12.1	iscntrl

（续）

宽字符函数	参考章节	字节字符函数
iswctype	12.	isctype
iswdigit	12.3	isdigit
iswgraph	12.4	isgraph
iswlower	12.5	islower
iswprint	12.4	isprint
iswpunct	12.4	ispunct
iswspace	12.6	isspace
iswupper	12.5	isupper
iswxdigit	12.3	isxdigit
towlower	12.9	tolower
towupper	12.9	toupper
wctrans	12.11	ctrans

494
~
495

第三部分 附 录

附录A ASCII字符集

十六进制	八进制	0 0 十进制	字符	名称	0x20 040 十进制	字符	0x40 0100 十进制	字符	0x60 0140 十进制	字符
0	0	0	^@	NUL	32	SP	64	@	96	`
1	1	1	^A	SOH	33	!	65	A	97	a
2	2	2	^B	STX	34	"	66	B	98	b
3	3	3	^C	ETX	35	#	67	C	99	c
4	4	4	^D	EOT	36	$	68	D	100	d
5	5	5	^E	ENQ	37	%	69	E	101	e
6	6	6	^F	ACK	38	&	70	F	102	f
7	7	7	^G	BEL, \a	39	'	71	G	103	g
8	010	8	^H	BS, \b	40	(	72	H	104	h
9	011	9	^I	TAB, \t	41	)	73	I	105	i
0xA	012	10	^J	LF, \n	42	*	74	J	106	j
0xB	013	11	^K	VT, \v	43	+	75	K	107	k
0xC	014	12	^L	FF, \f	44	,	76	L	108	l
0xD	015	13	^M	CR, \r	45	-	77	M	109	m
0xE	016	14	^N	SO	46	.	78	N	110	n
0xF	017	15	^O	SI	47	/	79	O	111	o
0x10	020	16	^P	DLE	48	0	80	P	112	p
0x11	021	17	^Q	DC1	49	1	81	Q	113	q
0x12	022	18	^R	DC2	50	2	82	R	114	r
0x13	023	19	^S	DC3	51	3	83	S	115	s
0x14	024	20	^T	DC4	52	4	84	T	116	t
0x15	025	21	^U	NAK	53	5	85	U	117	u
0x16	026	22	^V	SYN	54	6	86	V	118	v
0x17	027	23	^W	ETB	55	7	87	W	119	w
0x18	030	24	^X	CAN	56	8	88	X	120	x
0x19	031	25	^Y	EM	57	9	89	Y	121	y
0x1A	032	26	^Z	SUB	58	:	90	Z	122	z
0x1B	033	27	^[	ESC	59	;	91	[	123	{
0x1C	034	28	^\	FS	60	<	92	\	124	|
0x1D	035	29	^]	GS	61	=	93	]	125	}
0x1E	036	30	^^	RS	62	>	94	^	126	~
0x1F	037	31	^_	US	63	?	95	_	127	DEL

附录B C语言语法

abstract-declarator :
pointer
pointer$_{opt}$ *direct-abstract-declarator*

additive-expression :
multiplicative-expression
additive-expression add-op multiplicative-expression

add-op : one of
`+ -`

address-expression :
`&` *cast-expression*

array-declarator :
direct-declarator `[` *constant-expression*$_{opt}$ `]` *(until C99)*
direct-declarator `[` *array-qualifier-list*$_{opt}$ *array-size-expression*$_{opt}$ `]` *(C99)*
direct-declarator `[` *array-qualifier-list*$_{opt}$ `*` `]` *(C99)*

array-qualifier :
`static`
`restrict`
`const`
`volatile`

array-qualifier-list :
array-qualifier
array-qualifier-list array-qualifier

array-size-expression :
assignment-expression
`*`

assignment-expression :
conditional-expression
unary-expression assignment-op assignment-expression

assignment-op : one of
`= += -= *= /= %= <<= >>= &= ^= |=`

binary-exponent :
`p` *sign-partopt digit-sequence*
`P` *sign-partopt digit-sequence*

bit-field :
declarator$_{opt}$ `:` *width*

bitwise-and-expression :
equality-expression
bitwise-and-expression `&` *equality-expression*

bitwise-negation-expression :
~ *cast-expression*

bitwise-or-expression :
bitwise-xor-expression
bitwise-or-expression | *bitwise-xor-expression*

bitwise-xor-expression :
bitwise-and-expression
bitwise-xor-expression ^ *bitwise-and-expression*

break-statement :
break;

case-label :
case *constant-expression*

cast-expression :
unary-expression
(*type-name*) *cast-expression*

c-char :
any source character except the apostrophe ('), backslash (\), or newline
escape-character
universal-character-name *(C99)*

c-char-sequence :
c-char
c-char-sequence c-char

character-constant :
' *c-char-sequence* '
L' *c-char-sequence* ' *(C89)*

character-escape-code : one of
n t b r f
v \ ' "
a ? *(C89)*

character-type-specifier :
char
signed char
unsigned char

comma-expression :
assignment-expression
comma-expression , *assignment-expression*

complex-type-specifier : *(C99)*
float _Complex
double _Complex
long double _Complex

component-declaration :
type-specifier component-declarator-list ;

component-declarator :
simple-component

500

bit-field

component-declarator-list :
component-declarator
component-declarator-list , *component-declarator*

component-selection-expression :
direct-component-selection
indirect-component-selection

compound-literal :
(*type-name*) { *initializer-list* $_{,\ opt}$ } *(C99)*

compound-statement :
{ *declaration-or-statement-list*$_{opt}$ }

conditional-expression :
logical-or-expression
logical-or-expression ? *expression* : *conditional-expression*

conditional-statement :
if-statement
if-else-statement

constant :
integer-constant
floating-constant
character-constant
string-constant

constant-expression :
conditional-expression

continue-statement :
501 **continue**;

decimal-constant :
nonzero-digit
decimal-constant digit

decimal-floating-constant :
digit-sequence exponent floating-suffix$_{opt}$
dotted-digits exponent$_{opt}$ *floating-suffix*$_{opt}$

declaration :
declaration-specifiers initialized-declarator-list ;

declaration-list :
declaration
declaration-list declaration

declaration-or-statement :
declaration
statement

declaration-or-statement-list :
declaration-or-statement
declaration-or-statement-list declaration-or-statement

declaration-specifiers :
storage-class-specifier declaration-specifiers$_{opt}$
type-specifier declaration-specifiers$_{opt}$
type-qualifier declaration-specifiers$_{opt}$
function-specifier declaration-specifiers$_{opt}$ *(C99)*

declarator :
pointer-declarator
direct-declarator

default-label :
default

designation :
designator-list =

designator :
[*constant-expression*]
. *identifier*

designator-list :
designator
designator-list designator

digit : one of
0 1 2 3 4 5 6 7 8 9

digit-sequence :
digit
digit-sequence digit

direct-abstract-declarator :
(*abstract-declarator*)
direct-abstract-declarator$_{opt}$ [*constant-expressionopt*]
direct-abstract-declarator$_{opt}$ [*expression*] *(C99)*
direct-abstract-declarator$_{opt}$ [*] *(C99)*
direct-abstract-declaratoropt (*parameter-type-listopt*)

502

direct-component-selection :
postfix-expression . *identifier*

direct-declarator :
simple-declarator
(*declarator*)
function-declarator
array-declarator

do-statement :
do *statement* **while** (*expression*) ;

dotted-digits :
digit-sequence .
digit-sequence . *digit-sequence*
. *digit-sequence*

dotted-hex-digits :
hex-digit-sequence .
hex-digit-sequence . *hex-digit-sequence*

. *hex-digit-sequence*

enumeration-constant :
identifier

enumeration-constant-definition :
enumeration-constant
enumeration-constant = *expression*

enumeration-definition-list :
enumeration-constant-definition
enumeration-definition-list , *enumeration-constant-definition*

enumeration-tag :
identifier

enumeration-type-definition :
enum *enumeration-tag*$_{opt}$ { *enumeration-definition-list* }
enum *enumeration-tag*$_{opt}$ { *enumeration-definition-list* , } *(C99)*

enumeration-type-reference :
enum *enumeration-tag*

enumeration-type-specifier :
enumeration-type-definition
enumeration-type-reference

equality-expression :
relational-expression
equality-expression equality-op relational-expression

equality-op : one of
503 == !=

escape-character :
\ *escape-code*
universal-character-name *(C99)*

escape-code :
character-escape-code
octal-escape-code
hex-escape-code *(C89)*

exponent :
e *sign-partopt digit-sequence*
E *sign-partopt digit-sequence*

expression :
comma-expression

expression-list :
assignment-expression
expression-list , *assignment-expression*

expression-statement :
expression ;

field-list :
component-declaration
field-list component-declaration

floating-constant :
decimal-floating-constant
hexadecimal-floating-constant *(C99)*

floating-point-type-specifier :
float
double
long double *(C89)*
complex-type-specifier *(C99)*

floating-suffix : one of
f F l L

for-expressions :
(*initial-clauseopt* ; *expressionopt* ; *expressionopt*)

for-statement :
for *for-expressions statement*

function-call :
postfix-expression ($expression\text{-}list_{opt}$)

function-declarator :
direct-declarator (*parameter-type-list*) *(C89)*
direct-declarator ($identifier\text{-}list_{opt}$)

function-definition :
function-def-specifier compound-statement

function-def-specifier :
$declaration\text{-}specifiers_{opt}$ *declarator* $declaration\text{-}list_{opt}$ 504

function-specifier : *(C99)*
inline

goto-statement :
goto *named-label* ;

h-char-sequence :
any sequence of characters except > *and end-of-line*

hexadecimal-constant :
0x *hex-digit*
0X *hex-digit*
hexadecimal-constant hex-digit

hexadecimal-floating-constant: *(C99)*
hex-prefix dotted-hex-digits binary-exponent $floating\text{-}suffix_{opt}$
hex-prefix hex-digit-sequence binary-exponent $floating\text{-}suffix_{opt}$

hex-digit : one of
0 1 2 3 4 5 6 7 8 9
A B C D E F a b c d e f

hex-digit-sequence :
hex-digit
hex-digit-sequence hex-digit

hex-escape-code :
x *hex-digit*

hex-escape-code hex-digit *(C89)*

hex-prefix:
0x
0X

hex-quad:
hex-digit hex-digit hex-digit hex-digit

identifier :
identifier-nondigit
identifier identifier-nondigit
identifier digit

identifier-list :
identifier
parameter-list **,** *identifier*

identifier-nondigit :
nondigit
universal-character-name
other implementation-defined characters

if-else-statement :
if **(** *expression* **)** *statement* **else** *statement*

505 *if-statement :*
if **(** *expression* **)** *statement*

indirect-component-selection :
postfix-expression **->** *identifier*

indirection-expression :
***** *cast-expression*

initial-clause:
expression
declaration *(C99)*

initialized-declarator :
declarator
declarator **=** *initializer*

initialized-declarator-list :
initialized-declarator
initialized-declarator-list **,** *initialized-declarator*

initializer :
assignment-expression
{ *initializer-list* $_{,\,opt}$ **}**

initializer-list :
initializer
initializer-list **,** *initializer*
designation initializer *(C99)*
initializer-list **,** *designation initializer* *(C99)*

integer-constant :

decimal-constant integer-suffix$_{opt}$
octal-constant integer-suffix$_{opt}$
hexadecimal-constant integer-suffix$_{opt}$

integer-suffix :
long-suffix unsigned-suffix$_{opt}$
long-long-suffix unsigned-suffix$_{opt}$ *(C99)*
unsigned-suffix long-suffix$_{opt}$
unsigned-suffix long-long-suffix$_{opt}$ *(C99)*

integer-type-specifier :
signed-type-specifier
unsigned-type-specifier
character-type-specifier
bool-type-specifier *(C99)*

iterative-statement :
while-statement
do-statement
for-statement

label :
named-label
case-label
default-label

506

labeled-statement :
label **:** *statement*

logical-and-expression :
bitwise-or-expression
logical-and-expression **&&** *bitwise-or-expression*

logical-negation-expression :
! *cast-expression*

logical-or-expression :
logical-and-expression
logical-or-expression **||** *logical-and-expression*

long-long-suffix : one of *(C99)*
ll LL

long-suffix : one of
l L

multiplicative-expression :
cast-expression
multiplicative-expression mult-op cast-expression

mult-op **: one of**
*** / %**

named-label :
identifier

nondigit : one of

```
A  B  C  D  E  F  G  H  I  J  K  L  M
N  O  P  Q  R  S  T  U  V  W  X  Y  Z
a  b  c  d  e  f  g  h  i  j  k  l  m
n  o  p  q  r  s  t  u  v  w  x  y  z
_
```

nonzero-digit : one of

```
1  2  3  4  5  6  7  8  9
```

null-statement :

;

octal-constant :

0

octal-constant octal-digit

octal-digit : one of

```
0  1  2  3  4  5  6  7
```

octal-escape-code :

octal-digit

507 *octal-digit octal-digit*

octal-digit octal-digit octal-digit

on-off-switch:

ON

OFF

DEFAULT

parameter-declaration :

declaration-specifiers declarator

declaration-specifiers abstract-declarator$_{opt}$

parameter-list :

parameter-list

parameter-list , parameter-declaration

parameter-type-list :

parameter-list

parameter-list , ...

parenthesized-expression :

(*expression*)

pointer :

* *type-qualifier-list*$_{opt}$

* *type-qualifier-list*$_{opt}$ *pointer*

pointer-declarator :

pointer direct-declarator

postdecrement-expression :

postfix-expression --

postfix-expression :

primary-expression

subscript-expression

component-selection-expression

function-call
postincrement-expression
postdecrement-expression
compound-literal *(C99)*

postincrement-expression :
postfix-expression **++**

predecrement-expression :
-- *unary-expression*

preincrement-expression :
++ *unary-expression*

preprocessor-tokens :
any sequence of C tokens—or non-whitespace characters
that cannot be interpreted as tokens—that does not begin with **<** *or* **"**

primary-expression :
identifier
constant
parenthesized-expression

508

q-char-sequence :
any sequence of characters except **"** *and* end-*of-line*

relational-expression :
shift-expression
relational-expression relational-op shift-expression

relational-op : one of
< <= > >=

return-statement :
return *expressionopt* **;**

s-char :
any source character except the double quote **"**,
backslash ****, *or newline character*
escape-character
universal-character-name *(C99)*

s-char-sequence :
s-char
s-char-sequence s-char

shift-expression :
additive-expression
shift-expression shift-op additive-expression

shift-op : one of
<< >>

signed-type-specifier :
short *or* **short int** *or* **signed short** *or* **signed short int**
int *or* **signed int** *or* **signed**
long *or* **long int** *or* **signed long** *or* **signed long int**
long long *or* **long long int** *or* **signed long long** *or*
signed long long int

sign-part : one of
+ -

simple-component :
declarator

simple-declarator :
identifier

sizeof-expression :
sizeof (*type-name*)
sizeof *unary-expression*

statement :
expression-statement
labeled-statement
compound-statement
conditional-statement
iterative-statement
switch-statement
509 *break-statement*
continue-statement
return-statement
goto-statement
null-statement

storage-class-specifier : one of
auto extern register static typedef

string-constant :
" *s-char-sequenceopt* "
L" *s-char-sequenceopt* " *(C89)*

structure-tag :
identifier

structure-type-definition :
struct *structure-tagopt* { *field-list* }

structure-type-reference :
struct *structure-tag*

structure-type-specifier :
structure-type-definition
structure-type-reference

subscript-expression :
postfix-expression [*expression*]

switch-statement :
switch (*expression*) *statement*

top-level-declaration :
declaration
function-definition

translation-unit :
top-level-declaration
translation-unit top-level-declaration

typedef-name :
identifier

type-name :
declaration-specifiers abstract-declarator$_{opt}$

type-qualifier :
const
volatile
restrict *(C99)*

type-qualifier-list : *(C89)*
type-qualifier
type-qualifer-list type-qualifier

type-specifier :
enumeration-type-specifier
floating-point-type-specifier
integer-type-specifier
structure-type-specifier
typedef-name
union-type-specifier
void-type-specifier

510

unary-expression :
postfix-expression
sizeof-expression
unary-minus-expression
unary-plus-expression
logical-negation-expression
bitwise-negation-expression
address-expression
indirection-expression
preincrement-expression
predecrement-expression

unary-minus-expression :
- *cast-expression*

unary-plus-expression : *(C89)*
+ *cast-expression*

union-tag :
identifier

union-type-definition :
union *union-tagopt* **{** *field-list* **}**

union-type-reference :
union *union-tag*

union-type-specifier :
union-type-definition
union-type-reference

universal-character-name:
\u *hex-quad*

```
            \U hex-quad hex-quad
```

unsigned-suffix : one of

```
            u  U
```

unsigned-type-specifier :

```
            unsigned short int_opt
            unsigned int_opt
            unsigned long int_opt
            unsigned long long int_opt                                  (C99)
```

void-type-specifier :

```
            void
```

while-statement :

511

```
            while ( expression ) statement
```

width :

```
            constant-expression
```

512

附录C 练习答案

附录C包含第2章~第9章的练习答案。

第2章答案

1. 保留字、十六进制常量、宽字符串常量和括号是词法记号，注释和空白符只用于分隔记号，三字符组在记号识别之前删除。

2. 每个源字符串中的记号数如下：

（a）3个记号

（b）2个记号；-是运算符，而不属于常量的一部分

（c）1个记号

（d）3个记号；第2个是"FOO"

（e）1个记号

（f）4个记号；**不是单个运算符

（g）无记号；等于"x\"，是非终止字符串常量

（h）无记号；标识符不能有$符号

（i）3个记号；*=是一个运算符

（j）无记号或3个记号；##不是词法记号，但恰巧是一个预处理器记号

3. 消除注释后的结果为***/；括号间标识注释，注释中的引号不需要平衡。

```
/**/*/***/*/*"//*//**/*/
(--) (---) (-----)(--)
```

4. 操作顺序如下：

（1）转换三字符组

（2）处理续行

（3）删除注释

（4）把字符组合成记号

5. 可能的反对意见：

（a）难以标识（读取）标识符中的多个单词使用单词开头字母大写或单词间加下划线

（b）标识符拼写接近保留字

（c）小写l与大写O很容易和数字1与0相混

（d）很像数字字面值（第一个是字母o）

（e）如果编译器接受这个标识符，则这个编译器经过了扩展

6.（a）例如：`x = a //*divide*/ b;`

（b）假设标准C语言实现只能区分标识符前31个字符，则标准C语言程序的同一标识符在第31个字符之后拼写不同时，会在C++中被标记为错误。

（c）例如声明：`int class = 0;`

第3章答案

1.（a）标准C语言和传统C语言不允许左括号前面的空格。ident不是一个参数的宏，而是

没有参数的宏，扩展为“(x) x”。

（b）等号与分号是不必要的而且可能是错误。在一些传统C语言编译器中，#号后面的空格可能造成问题。

（c）这个宏定义是正确的。

（d）这个宏定义正确，可以将保留字定义为宏。

2.

	标准C语言	传统C语言
（a）	b+a	b+a
（b）	x 4（两个记号）	x4（一个记号）
（c）	"a book"	# a book
（d）	p?free(p):NULL	p?p?p?…:NULL:NULL:NULL（无穷）

3. 预处理之后（忽略空白符）的结果是下列3行：

```
int blue = 0;
int blue = 0;
int red = 0;
```

4. 由于参数和宏体没有放在括号中，因此扩展宏的结果可能在大的表达式中造成误解。更安全的定义如下：

```
#define DBL(a)    ((a)+(a))
```

5. 宏的扩展步骤如下：

```
M(M)(A,B)
MM(A,B)
A = "B"
```

6. 这个方案要求存在defined与#error：

```
#if !defined(SIZE) || (SIZE<1) || (SIZE>10)
#error "SIZE not properly defined "
#endif
```

7. 在预处理器命令#include </a/file.h>中，/a/file.h序列是个记号（单个文件名），而不是编译器的记号。

8. 假设编程人员在x==0时要打印一个错误，但x==0是个运行测试，而#error是个编译命令。如果编译这个程序，则不管x值如何，总是出现错误消息并停止编译。

514

第4章答案

1. 每次调用时，函数返回参数值。只有P中的i声明使用static存储类指定符时，连续调用才会改变返回值。

2. 将f声明为函数、整型变量、类型名和枚举常量都会相互冲突，只能留下一个声明。用f同时作为结构标志与联合标志会相互冲突，删除联合标志，使f还可以声明为结构成员。用f作为标号不会与其他声明相互冲突，但一些较早的C语言实现中会相互冲突。

3.

代码	int i;	long i;	float i;
1 int i;	(声明)		
2 void f(i)		(声明)	
3 long i;		(声明)	
4 {			
5 long l = i;		(使用)	
6 {			
7 float i;			(声明)
8 i = 3.4;			(使用)
9 }			
10 l = i+2;		(使用)	
11 }			
12 int *p = &i;	(使用)		

4. (a) `extern void P(void);`
   (b) `register int i;`
   (c) `typedef char *LT;`
   (d) `extern void Q(int i, const char *cp);`
   (e) `extern int R( double *(*p) (long i) );`
   (f) `static char STR[11];` (注：对null字符保留空间)
   (g) `const char STR2[] = {INIT_STR2};` (`INIT_STR2`周围的花括号可选)
   也可：`const char *STR2 = INIT_STR2;` (无花括号)
   (h) `int *IP = &i;`
5. `int m[3][3] = {{1,2,3},{1,2,3}, {1,2,3}};`

第5章答案

1. 注意下列类型都不涉及`int`类型，因为`int`类型的长度不一定比`short`类型长。
   (a) `long`类型或`unsigned long`类型(`unsigned short`类型可能无法处理99999)。
   (b) 包含两个成员的结构类型：类型`short`（区号）和类型`long`（市话号码）(或这些类型的无符号类型)。
   (c) `char`（任何变体）。
   (d) 标准C语言中使用`signed char`类型，其他C语言实现中使用`short`类型（`char`类型可能无符号）。
   (e) 标准C语言中使用`signed char`类型，其他C语言实现中使用`short`类型（`char`类型可能无符号）。
   (f) `double`类型满足要求，但用类型`long`占用的空间更少并且将存款余额折算成美分存储。 515

2. `UP_ARROW_KEY`类型为`int`，取值为`0x86`（134）。如果计算机使用`signed char`类型，则扩展字符的值为负数，因此`is_up_arrow`的参数为`0x86`时，`return`语句中的测试变成`-122==134`，结果是false而不是ture。编写这个函数的正确方法是把字符码转换成`char`类型或把参数转换成`unsigned char`类型，可使用下列`return`语句之一：

```
return c == (char) UP_ARROW_KEY;
return (unsigned char) c == UP_ARROW_KEY;
```

第一个方案更好，因为它在定义`UP_ARROW_KEY`值时有更大的自由度。

3.（a）合法

（b）合法

（c）非法，无法将`void *`指针取消引用

（d）非法，无法将`void *`指针取消引用

4.（a）`*(iv + i)`

（b）`*(*(im+i)+j)`

5. 结果为13。标准C语言中不需要转换，但这样的转换表达式可以使意图更明确，而且在一些较早的编译器中可能需要这样明确的转换表示。

6. `x.i = 0;`

`x.F.s = 0;`　（只有0和1是合法值）

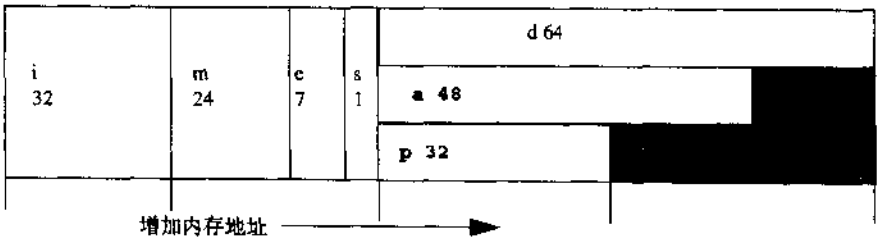
`x.F.e = 0; x.F.m = 0;`

`x.U.d = 0.0;`　（或`x.U.p = NULL;`，但不能用`x.U.a[0] = '\0';`，否则a中有些元素未定义）

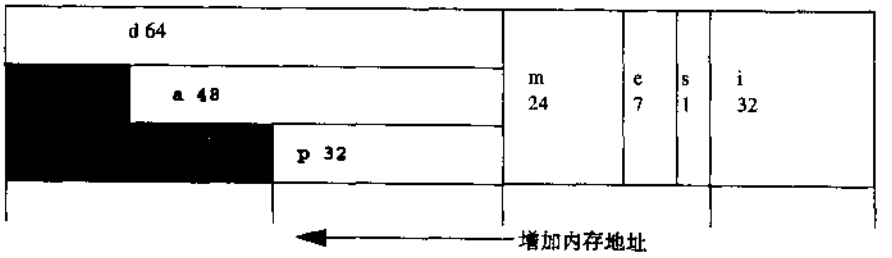
516

7. 示意图如下，显示每个字段占用的位数并用底部标志表示字边界。注意位字段的特定顺序。

高位存储法，从右到左按位存储



低位存储法，从左到右按位存储



8.
```
typedef int *fpi(); /* type definition */
fpi *x; /* variable declaration */
int *fpi() /* function; can't use typedef in header */
{
    return (fpi *)0;
}
```

第6章答案

1. 标准C语言和传统C语言中允许除(c)与(e)以外的所有转换，(c)与(e)在标准C语言中不允许。

2. 在这个方案中，我们假设传统C语言编译器允许混合指针赋值，此外其他方面都符合标准C语言规则。但对一些传统C语言编译器，答案与第1题相同。

(a) 标准C语言和传统C语言中允许。

(b) 标准C语言不允许，传统C语言允许。

(c) 标准C语言不允许，传统C语言允许。

(d) 标准C语言和传统C语言中都不允许。

(e) 标准C语言不允许，传统C语言允许。

(f) 标准C语言和传统C语言中允许。

3. (a) **unsigned**

517

(b) 传统C语言为**unsigned long**，标准C语言为**long**或**unsigned long**

(c) **double**

(d) 标准C语言为**long double**

(e) **int ***（对**int []**采用普通一元转换）

```
extern short f1(), f2(), (*pf)();
extern int i;
pf = (i>0 ? f1 : f2);  /* binary conv on f1 and f2 */
```

4. 实现可以用32位表示**char**类型（但较浪费）。不管表示方法如何，**sizeof(char)**的值总是1。**int**类型的范围不能小于**char**类型的范围，但可以相同或更大。

5. 它们之间不必有任何关系，可以是相同或不同。

6. 128值可以表示为32位十六进制数00000080_{16}。由于计算机A使用高位存储法，因此字节存放顺序为00_{16}，00_{16}，00_{16}，80_{16}。在低位存储法计算机上，字节从低位开始存储，得到80000000_{16}即 -2 147 483 648。如果A使用低位存储法而B使用高位存储法，结果也一样。

第7章答案

1. (a) **char ***

(b) **float**（传统C语言为**double**）

(c) **float**

(d) **int**

(e) **float**（传统C语言为**double**）

(f) **int**

(g) **int**

(h) **int**

(i) 非法表达式

(j) **float**

2. (a) **p1+=1; p2+=1; *p1=*p2;**

(b) ***p1=*p2; p1-=1; p2-=1;**

3. (a) **#define low_zeroes(n) (-1<<n)**（如果**n**的值不大于**int**类型的宽度）

（b）`#define low_ones(n)  (~low_zeroes(n))`

（c）
```
#define mid_zeroes(width, offset) \
        (low_zeroes(width+offset) | low_ones(offset))
```
（+运算符可以代替|）

（d）`#define mid_ones(width, offset (~mid_zeroes(width, offset))`

4. 表达式`j++==++j`是合法的，但结果在标准C语言中未定义，因为`j`在同一表达式中修改两次。根据`==`的操作数是否先求值，结果可能是0或1，但`j`的最后结果却是2。但是，`j++&&++j`是合法而定义好的，结果为0，到表达式结束时，`j`的值为1。

5.（a）允许，因为类型兼容

（b）不允许（左边引用的类型没有足够限定符）

（c）允许，因为只有一个类型指定长度

518

（d）允许，因为左边不是左值时，限定并不重要

（e）不允许，因为`float`与升级的类型（`double`）不兼容

（f）允许，因为引用的类型兼容

6. 无效。赋值非法，因为每个结构定义生成新类型。如果定义放在不同源文件中，则类型兼容，但这等于允许不同块中编译的程序具有定义良好的行为。

第8章答案

1.（a）
```
n = A;
L1:
if (n>=B) goto L2;
sum+=n;
n++;
goto L1;
L2:;
```
（b）
```
L:
if (a<b) {
   a++;
   goto L;
}
```
（c）
```
L:
sum += *p;
if (++p < q) goto L;
```

2. `j`的值为3。“`j`为未定义”是不对的，虽然程序跳入块中，但`i`的存储类为`static`，因此会在程序开始之前正确地初始化。

3. `sum`的值为3，`i`在每次循环时取值0、1、...。i为0、1和3时，`sum`递增，`continue`语句产生另一循环迭代。`i`为2时，`sum`不改变，但循环继续。但`i`为4时，`switch`中的`break`使控制到达循环中的`break`语句，从而终止循环。因此，`case 5:`并不执行。

第9章答案

1.（a）有效原型

（b）合法声明，但不是原型，括号中要有参数类型列表
（c）非法声明，省略号之前至少要有一个参数声明
（d）非法声明，每个参数名前面至少要有一个类型说明符、存储类说明符或类型限定符
（e）有效原型，参数名不是必需的
（f）合法定义，但不是原型，括号中要有参数类型

2.（a）不兼容，原型参数类型与普通参数转换不兼容，其在定义不是原型形式时需要
（b）不兼容，定义中出现原型没关系
（c）兼容，参数名不必相同
（d）不兼容，两个原型的省略号用法不一致 519
（e）兼容，都不是原型，因此传递升级的参数类型
（f）兼容

3.（a）不合法，赋值转换中不能将`short *`变为`int *`
（b）合法，`s`转换成`int`，而`ld`不变
（c）合法，`ld`转换成`short`类型
（d）合法，第一个参数不变，第二个参数转换成`int`，而第三个参数不变
（e）合法，参数转换成`int`之后再调用，在被调函数开头返回`short`类型
（f）合法，但可能有错，参数不变，但调用者将其解释为`int`类型

4. 调用由第一行的原型控制，后面的声明不隐藏前者，因为`p`具有外部连接。

5.（a）可以；数值转换为`short`类型之后再返回
（b）可以；数值转换为`short`类型之后再返回
（c）非法；表达式无法转换成返回值类型
（d）非法；表达式无法在赋值转换规则中转换 520

索　引

索引中的页码为英文原书页码，与书中页边标注的页码一致。

A

B

C

D

E

F

G

H

I

M

N

O

Q

R

S

T

U

V

W

X

Y