

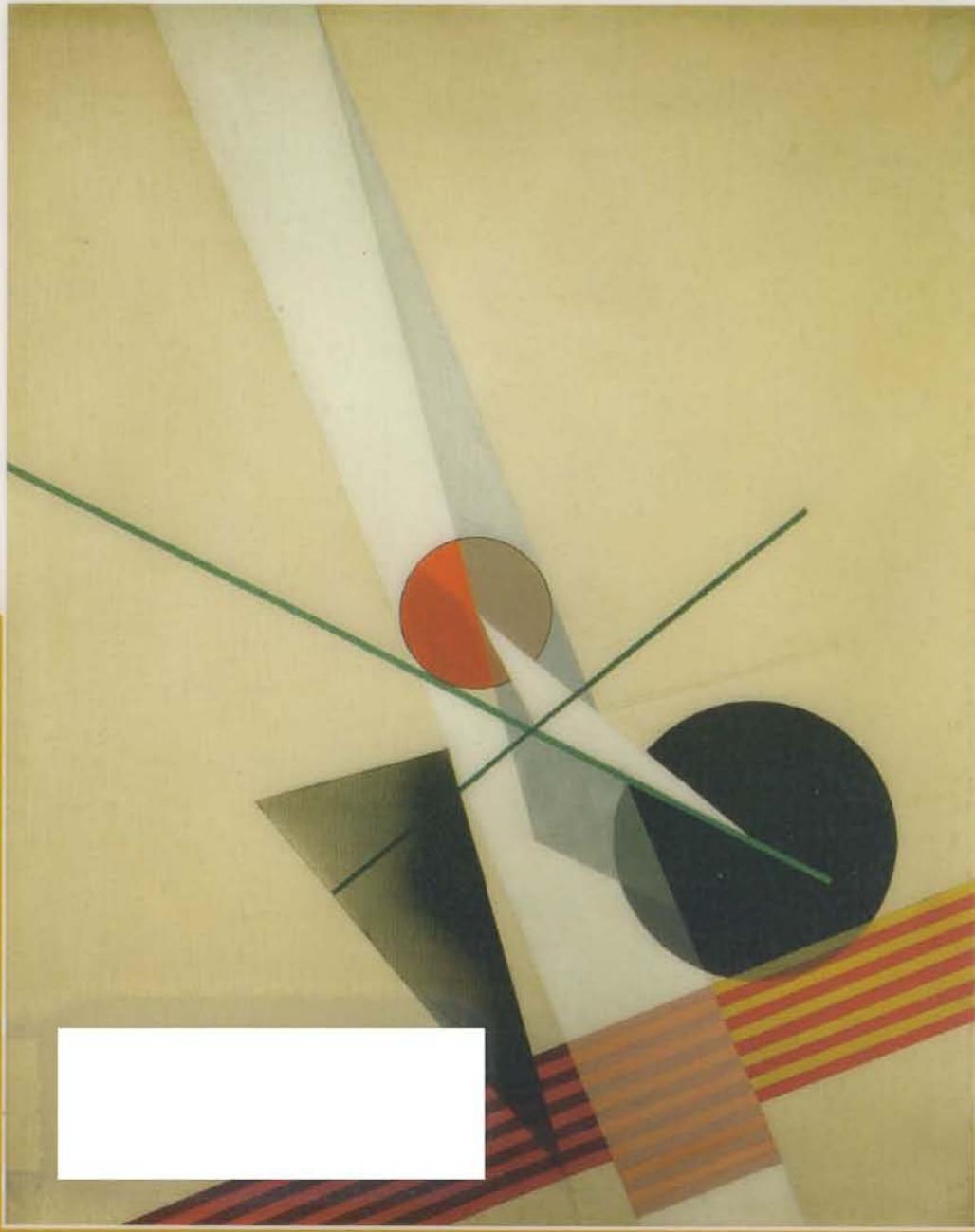
*Covers both C89  
and C99*

K.N.KING

# C PROGRAMMING

*A Modern Approach*

SECOND EDITION



# K.N.KING

# C PROGRAMMING

*A Modern Approach*

SECOND EDITION

*A clear, complete, and engaging presentation of the C programming language—now with coverage of both C89 and C99*

The first edition of *C Programming: A Modern Approach* was a hit with students and faculty alike because of its clarity and comprehensiveness as well as its trademark Q&A sections. King's spiral approach made the first edition accessible to a broad range of readers, from beginners to more advanced students. The first edition was used at over 225 colleges, making it one of the leading C textbooks of the last ten years.

## FEATURES OF THE SECOND EDITION

- Complete coverage of both the C89 standard and the C99 standard, with all C99 changes clearly marked
- Includes a quick reference to all C89 and C99 library functions
- Expanded coverage of GCC
- New coverage of abstract data types
- Updated to reflect today's CPUs and operating systems
- Nearly 500 exercises and programming projects—60% more than in the first edition
- Source code and solutions to selected exercises and programming projects for students, available at the author's website (*knking.com*)
- Password-protected instructor site (also at *knking.com*) containing solutions to the remaining exercises and projects, plus PowerPoint presentations for most chapters

"I thoroughly enjoyed reading the second edition of *C Programming* and I look forward to using it in future courses."

— Karen Reid, Senior Lecturer, Department of Computer Science, University of Toronto

"The second edition of King's *C Programming* improves on an already impressive base, and is the book I recommend to anyone who wants to learn C."

— Peter Seebach, moderator, *comp.lang.c.moderated*

"I assign *C Programming* to first-year engineering students. It is concise, clear, accessible to the beginner, and yet also covers all aspects of the language."

— Professor Markus Bussmann, Department of Mechanical and Industrial Engineering, University of Toronto

K. N. KING (Ph.D., University of California, Berkeley) is an associate professor of computer science at Georgia State University. He is also the author of *Modula-2: A Complete Guide* and *Java Programming: From the Beginning*.

ISBN 978-0-393-97950-3



9 0000 >

N/A

9 780393 979503

Cover design by Joan Greenfield

Cover art: László Moholy-Nagy, *AXXI*, Westfälisches Landesmuseum, Münster, Westphalia, Germany  
Photo: Erich Lessing / Art Resource, NY  
Copyright: © ABS, NY



W. W. NORTON

NEW YORK • LONDON

WWW.WWNORTON.COM

# PREFACE

*In computing, turning the obvious into the useful  
is a living definition of the word “frustration.”*

In the years since the first edition of *C Programming: A Modern Approach* was published, a host of new C-based languages have sprung up—Java and C# foremost among them—and related languages such as C++ and Perl have achieved greater prominence. Still, C remains as popular as ever, plugging away in the background, quietly powering much of the world’s software. It remains the *lingua franca* of the computer universe, as it was in 1996.

But even C must change with the times. The need for a new edition of *C Programming: A Modern Approach* became apparent when the C99 standard was published. Moreover, the first edition, with its references to DOS and 16-bit processors, was becoming dated. The second edition is fully up-to-date and has been improved in many other ways as well.

## What’s New in the Second Edition

Here’s a list of new features and improvements in the second edition:

- **Complete coverage of both the C89 standard and the C99 standard.** The biggest difference between the first and second editions is coverage of the C99 standard. My goal was to cover every significant difference between C89 and C99, including all the language features and library functions added in C99. Each C99 change is clearly marked, either with “C99” in the heading of a section or—in the case of shorter discussions—with a special icon in the left margin. I did this partly to draw attention to the changes and partly so that readers who aren’t interested in C99 or don’t have access to a C99 compiler will know what to skip. Many of the C99 additions are of interest only to a specialized audience, but some of the new features will be of use to nearly all C programmers.

C99

- ***Includes a quick reference to all C89 and C99 library functions.*** Appendix D in the first edition described all C89 standard library functions. In this edition, the appendix covers all C89 and C99 library functions.
- ***Expanded coverage of GCC.*** In the years since the first edition, use of GCC (originally the GNU C Compiler, now the GNU Compiler Collection) has spread. GCC has some significant advantages, including high quality, low (i.e., no) cost, and portability across a variety of hardware and software platforms. In recognition of its growing importance, I've included more information about GCC in this edition, including discussions of how to use it as well as common GCC error messages and warnings.
- ***New coverage of abstract data types.*** In the first edition, a significant portion of Chapter 19 was devoted to C++. This material seems less relevant today, since students may already have learned C++, Java, or C# before reading this book. In this edition, coverage of C++ has been replaced by a discussion of how to set up abstract data types in C.
- ***Expanded coverage of international features.*** Chapter 25, which is devoted to C's international features, is now much longer and more detailed. Information about the Unicode/UCS character set and its encodings is a highlight of the expanded coverage.
- ***Updated to reflect today's CPUs and operating systems.*** When I wrote the first edition, 16-bit architectures and the DOS operating system were still relevant to many readers, but such is not the case today. I've updated the discussion to focus more on 32-bit and 64-bit architectures. The rise of Linux and other versions of UNIX has dictated a stronger focus on that family of operating systems, although aspects of Windows and the Mac OS operating system that affect C programmers are mentioned as well.
- ***More exercises and programming projects.*** The first edition of this book contained 311 exercises. This edition has nearly 500 (498, to be exact), divided into two groups: exercises and programming projects.
- ***Solutions to selected exercises and programming projects.*** The most frequent request I received from readers of the first edition was to provide answers to the exercises. In response to this request, I've put the answers to roughly one-third of the exercises and programming projects on the web at [knking.com/books/c2](http://knking.com/books/c2). This feature is particularly useful for readers who aren't enrolled in a college course and need a way to check their work. Exercises and projects for which answers are provided are marked with a  icon (the "W" stands for "answer available on the Web").
- ***Password-protected instructor website.*** For this edition, I've built a new instructor resource site (accessible through [knking.com/books/c2](http://knking.com/books/c2)) containing solutions to the remaining exercises and projects, plus PowerPoint presentations for most chapters. Faculty may contact me at [chook@knking.com](mailto:chook@knking.com) for a password. Please use your campus email address and include a link to your department's website so that I can verify your identity.

I've also taken the opportunity to improve wording and explanations throughout the book. The changes are extensive and painstaking: every sentence has been checked and—if necessary—rewritten.

Although much has changed in this edition, I've tried to retain the original chapter and section numbering as much as possible. Only one chapter (the last one) is entirely new, but many chapters have additional sections. In a few cases, existing sections have been renumbered. One appendix (C syntax) has been dropped, but a new appendix that compares C99 with C89 has been added.

## Goals

The goals of this edition remain the same as those of the first edition:

- ***Be clear, readable, and possibly even entertaining.*** Many C books are too concise for the average reader. Others are badly written or just plain dull. I've tried to give clear, thorough explanations, leavened with enough humor to hold the reader's interest.
- ***Be accessible to a broad range of readers.*** I assume that the reader has at least a little previous programming experience, but I don't assume knowledge of a particular language. I've tried to keep jargon to a minimum and to define the terms that I use. I've also attempted to separate advanced material from more elementary topics, so that the beginner won't get discouraged.
- ***Be authoritative without being pedantic.*** To avoid arbitrarily deciding what to include and what not to include, I've tried to cover all the features of the C language and library. At the same time, I've tried to avoid burdening the reader with unnecessary detail.
- ***Be organized for easy learning.*** My experience in teaching C underscores the importance of presenting the features of C gradually. I use a spiral approach, in which difficult topics are introduced briefly, then revisited one or more times later in the book with details added each time. Pacing is deliberate, with each chapter building gradually on what has come before. For most students, this is probably the best approach: it avoids the extremes of boredom on the one hand, or “information overload” on the other.
- ***Motivate language features.*** Instead of just describing each feature of the language and giving a few simple examples of how the feature is used, I've tried to motivate each feature and discuss how it's used in practical situations.
- ***Emphasize style.*** It's important for every C programmer to develop a consistent style. Rather than dictating what this style should be, though, I usually describe a few possibilities and let the reader choose the one that's most appealing. Knowing alternative styles is a big help when reading other people's programs (which programmers often spend a great deal of time doing).
- ***Avoid dependence on a particular machine, compiler, or operating system.*** Since C is available on such a wide variety of platforms, I've tried to avoid

dependence on any particular machine, compiler, or operating system. All programs are designed to be portable to a wide variety of platforms.

- **Use illustrations to clarify key concepts.** I've tried to put in as many figures as I could, since I think these are crucial for understanding many aspects of C. In particular, I've tried to "animate" algorithms whenever possible by showing snapshots of data at different points in the computation.

## What's So Modern about *A Modern Approach*?

One of my most important goals has been to take a "modern approach" to C. Here are some of the ways I've tried to achieve this goal:

- **Put C in perspective.** Instead of treating C as the only programming language worth knowing, I treat it as one of many useful languages. I discuss what kind of applications C is best suited for; I also show how to capitalize on C's strengths while minimizing its weaknesses.
- **Emphasize standard versions of C.** I pay minimal attention to versions of the language prior to the C89 standard. There are just a few scattered references to K&R C (the 1978 version of the language described in the first edition of Brian Kernighan and Dennis Ritchie's book, *The C Programming Language*). Appendix C lists the major differences between C89 and K&R C.
- **Debunk myths.** Today's compilers are often at odds with commonly held assumptions about C. I don't hesitate to debunk some of the myths about C or challenge beliefs that have long been part of the C folklore (for example, the belief that pointer arithmetic is always faster than array subscripting). I've re-examined the old conventions of C, keeping the ones that are still helpful.
- **Emphasize software engineering.** I treat C as a mature software engineering tool, emphasizing how to use it to cope with issues that arise during programming-in-the-large. I stress making programs readable, maintainable, reliable, and portable, and I put special emphasis on information hiding.
- **Postpone C's low-level features.** These features, although handy for the kind of systems programming originally done in C, are not as relevant now that C is used for a great variety of applications. Instead of introducing them in the early chapters, as many C books do, I postpone them until Chapter 20.
- **De-emphasize "manual optimization."** Many books teach the reader to write tricky code in order to gain small savings in program efficiency. With today's abundance of optimizing C compilers, these techniques are often no longer necessary; in fact, they can result in programs that are less efficient.

## Q&A Sections

Each chapter ends with a "Q&A section"—a series of questions and answers related to material covered in the chapter. Topics addressed in these sections include:

- **Frequently asked questions.** I've tried to answer questions that come up frequently in my own courses, in other books, and on newsgroups related to C.
- **Additional discussion and clarification of tricky issues.** Although readers with experience in a variety of languages may be satisfied with a brief explanation and a couple of examples, readers with less experience need more.
- **Side issues that don't belong in the main flow.** Some questions raise technical issues that won't be of interest to all readers.
- **Material too advanced or too esoteric to interest the average reader.** Questions of this nature are marked with an asterisk (\*). Curious readers with a fair bit of programming experience may wish to delve into these questions immediately; others should definitely skip them on a first reading. *Warning:* These questions often refer to topics covered in later chapters.
- **Common differences among C compilers.** I discuss some frequently used (but nonstandard) features provided by particular compilers.

Some questions in Q&A sections relate directly to specific places in the chapter; these places are marked by a special icon to signal the reader that additional information is available.

### **Q&A**

## Other Features

In addition to Q&A sections, I've included a number of useful features, many of which are marked with simple but distinctive icons (shown at left).



cross-references ► Preface

**idiom**

**portability tip**

- **Warnings** alert readers to common pitfalls. C is famous for its traps; documenting them all is a hopeless—if not impossible—task. I've tried to pick out the pitfalls that are most common and/or most important.
- **Cross-references** provide a hypertext-like ability to locate information. Although many of these are pointers to topics covered later in the book, some point to previous topics that the reader may wish to review.
- **Idioms**—code patterns frequently seen in C programs—are marked for quick reference.
- **Portability tips** give hints for writing programs that are independent of a particular machine, compiler, or operating system.
- **Sidebars** cover topics that aren't strictly part of C but that every knowledgeable C programmer should be aware of. (See “Source Code” on the next page for an example of a sidebar.)
- **Appendices** provide valuable reference information.

## Programs

Choosing illustrative programs isn't an easy job. If programs are too brief and artificial, readers won't get any sense of how the features are used in the real world. On the other hand, if a program is *too* realistic, its point can easily be lost in a forest of

details. I've chosen a middle course, using small, simple examples to make concepts clear when they're first introduced, then gradually building up to complete programs. I haven't included programs of great length; it's been my experience that instructors don't have the time to cover them and students don't have the patience to read them. I don't ignore the issues that arise in the creation of large programs, though—Chapter 15 (Writing Large Programs) and Chapter 19 (Program Design) cover them in detail.

I've resisted the urge to rewrite programs to take advantage of the features of C99, since not every reader may have access to a C99 compiler or wish to use C99. I have, however, used C99's `<stdbool.h>` header in a few programs, because it conveniently defines macros named `bool`, `true`, and `false`. If your compiler doesn't support the `<stdbool.h>` header, you'll need to provide your own definitions for these names.

The programs in this edition have undergone one very minor change. The main function now has the form `int main(void) { ... }` in most cases. This change reflects recommended practice and is compatible with C99, which requires an explicit return type for each function.

---

### *Source Code*

Source code for all programs is available at [knking.com/books/c2](http://knking.com/books/c2). Updates, corrections, and news about the book can also be found at this site.

---

### **Audience**

This book is designed as a primary text for a C course at the undergraduate level. Previous programming experience in a high-level language or assembler is helpful but not necessary for a computer-literate reader (an “adept beginner,” as one of my former editors put it).

Since the book is self-contained and usable for reference as well as learning, it makes an excellent companion text for a course in data structures, compiler design, operating systems, computer graphics, embedded systems, or other courses that use C for project work. Thanks to its Q&A sections and emphasis on practical problems, the book will also appeal to readers who are enrolled in a training class or who are learning C by self-study.

### **Organization**

The book is divided into four parts:

- **Basic Features of C.** Chapters 1–10 cover enough of C to allow the reader to write single-file programs using arrays and functions.
- **Advanced Features of C.** Chapters 11–20 build on the material in the earlier chapters. The topics become a little harder in these chapters, which provide in-

depth coverage of pointers, strings, the preprocessor, structures, unions, enumerations, and low-level features of C. In addition, two chapters (15 and 19) offer guidance on program design.

- **The Standard C Library.** Chapters 21–27 focus on the C library, a large collection of functions that come with every compiler. These chapters are most likely to be used as reference material, although portions are suitable for lectures.
- **Reference.** Appendix A gives a complete list of C operators. Appendix B describes the major differences between C99 and C89, and Appendix C covers the differences between C89 and K&R C. Appendix D is an alphabetical listing of all functions in the C89 and C99 standard libraries, with a thorough description of each. Appendix E lists the ASCII character set. An annotated bibliography points the reader toward other sources of information.

A full-blown course on C should cover Chapters 1–20 in sequence, with topics from Chapters 21–27 added as needed. (Chapter 22, which includes coverage of file input/output, is the most important chapter of this group.) A shorter course can omit the following topics without losing continuity: Section 8.3 (variable-length arrays), Section 9.6 (recursion), Section 12.4 (pointers and multidimensional arrays), Section 12.5 (pointers and variable-length arrays), Section 14.5 (miscellaneous directives), Section 17.7 (pointers to functions), Section 17.8 (restricted pointers), Section 17.9 (flexible array members), Section 18.6 (inline functions), Chapter 19 (program design), Section 20.2 (bit-fields in structures), and Section 20.3 (other low-level techniques).

## Exercises and Programming Projects

Having a variety of good problems is obviously essential for a textbook. This edition of the book contains both exercises (shorter problems that don't require writing a full program) and programming projects (problems that require writing or modifying an entire program).

A few exercises have nonobvious answers (some individuals uncharitably call these "trick questions"—the nerve!). Since C programs often contain abundant examples of such code, I feel it's necessary to provide some practice. However, I'll play fair by marking these exercises with an asterisk (\*). Be careful with a starred exercise: either pay close attention and think hard or skip it entirely.

## Errors, Lack of (?)

I've taken great pains to ensure the accuracy of this book. Inevitably, however, any book of this size contains a few errors. If you spot one, please contact me at [cbook@knking.com](mailto:cbook@knking.com). I'd also appreciate hearing about which features you found especially helpful, which ones you could do without, and what you'd like to see added.

## Acknowledgments

First, I'd like to thank my editors at Norton, Fred McFarland and Aaron Javicas. Fred got the second edition underway and Aaron stepped in with brisk efficiency to bring it to completion. I'd also like to thank associate managing editor Kim Yi, copy editor Mary Kelly, production manager Roy Tedoff, and editorial assistant Carly Fraser.

I owe a huge debt to the following colleagues, who reviewed some or all of the manuscript for the second edition:

Markus Bussmann, University of Toronto  
Jim Clarke, University of Toronto  
Karen Reid, University of Toronto  
Peter Seebach, moderator of *comp.lang.c.moderated*

Jim and Peter deserve special mention for their detailed reviews, which saved me from a number of embarrassing slips. The reviewers for the first edition, in alphabetical order, were: Susan Anderson-Freed, Manuel E. Bermudez, Lisa J. Brown, Steven C. Cater, Patrick Harrison, Brian Harvey, Henry H. Leitner, Darrell Long, Arthur B. Maccabe, Carolyn Rosner, and Patrick Terry.

I received many useful comments from readers of the first edition; I thank everyone who took the time to write. Students and colleagues at Georgia State University also provided valuable feedback. Ed Bullwinkel and his wife Nancy were kind enough to read much of the manuscript. I'm particularly grateful to my department chair, Yi Pan, who was very supportive of the project.

My wife, Susan Cole, was a pillar of strength as always. Our cats, Dennis, Pounce, and Tex, were also instrumental in the completion of the book. Pounce and Tex were happy to contribute the occasional catfight to help keep me awake while I was working late at night.

Finally, I'd like to acknowledge the late Alan J. Perlis, whose epigrams appear at the beginning of each chapter. I had the privilege of studying briefly under Alan at Yale in the mid-70s. I think he'd be amused at finding his epigrams in a C book.

# BRIEF CONTENTS

## Basic Features of C

1	Introducing C	1
2	C Fundamentals	9
3	Formatted Input/Output	37
4	Expressions	53
5	Selection Statements	73
6	Loops	99
7	Basic Types	125
8	Arrays	161
9	Functions	183
10	Program Organization	219

## The Standard C Library

21	The Standard Library	529
22	Input/Output	539
23	Library Support for Numbers and Character Data	589
24	Error Handling	627
25	International Features	641
26	Miscellaneous Library Functions	677
27	Additional C99 Support for Mathematics	705

## Advanced Features of C

11	Pointers	241
12	Pointers and Arrays	257
13	Strings	277
14	The Preprocessor	315
15	Writing Large Programs	349
16	Structures, Unions, and Enumerations	377
17	Advanced Uses of Pointers	413
18	Declarations	457
19	Program Design	483
20	Low-Level Programming	509

## Reference

A	C Operators	735
B	C99 versus C89	737
C	C89 versus K&R C	743
D	Standard Library Functions	747
E	ASCII Character Set Bibliography Index	801 803 807

# CONTENTS

<b>Preface</b>	<b>xxi</b>
<b>1 INTRODUCING C</b>	<b>1</b>
<b>1.1 History of C</b>	<b>1</b>
Origins	1
Standardization	2
C-Based Languages	3
<b>1.2 Strengths and Weaknesses of C</b>	<b>4</b>
Strengths	4
Weaknesses	5
Effective Use of C	6
<b>2 C FUNDAMENTALS</b>	<b>9</b>
<b>2.1 Writing a Simple Program</b>	<b>9</b>
Program: Printing a Pun	9
Compiling and Linking	10
Integrated Development Environments	11
<b>2.2 The General Form of a Simple Program</b>	<b>12</b>
Directives	12
Functions	13
Statements	14
Printing Strings	14
<b>2.3 Comments</b>	<b>15</b>
<b>2.4 Variables and Assignment</b>	<b>17</b>
Types	17
Declarations	17
Assignment	18

Printing the Value of a Variable	19
Program: Computing the Dimensional Weight of a Box	20
Initialization	21
Printing Expressions	22
<b>2.5 Reading Input</b>	<b>22</b>
Program: Computing the Dimensional Weight of a Box (Revisited)	22
<b>2.6 Defining Names for Constants</b>	<b>23</b>
Program: Converting from Fahrenheit to Celsius	24
<b>2.7 Identifiers</b>	<b>25</b>
Keywords	26
<b>2.8 Layout of a C Program</b>	<b>27</b>
<b>3 FORMATTED INPUT/OUTPUT</b>	<b>37</b>
<b>3.1 The printf Function</b>	<b>37</b>
Conversion Specifications	38
Program: Using printf to Format Numbers	40
Escape Sequences	41
<b>3.2 The scanf Function</b>	<b>42</b>
How scanf Works	43
Ordinary Characters in Format Strings	45
Confusing printf with scanf	45
Program: Adding Fractions	46
<b>4 EXPRESSIONS</b>	<b>53</b>
<b>4.1 Arithmetic Operators</b>	<b>54</b>
Operator Precedence and Associativity	55
Program: Computing a UPC Check Digit	56
<b>4.2 Assignment Operators</b>	<b>58</b>
Simple Assignment	58
Lvalues	59
Compound Assignment	60
<b>4.3 Increment and Decrement Operators</b>	<b>61</b>
<b>4.4 Expression Evaluation</b>	<b>62</b>
Order of Subexpression Evaluation	64
<b>4.5 Expression Statements</b>	<b>65</b>
<b>5 SELECTION STATEMENTS</b>	<b>73</b>
<b>5.1 Logical Expressions</b>	<b>74</b>
Relational Operators	74
Equality Operators	75
Logical Operators	75
<b>5.2 The if Statement</b>	<b>76</b>
Compound Statements	77

The else Clause	78
Cascaded if Statements	80
Program: Calculating a Broker's Commission	81
The "Dangling else" Problem	82
Conditional Expressions	83
Boolean Values in C89	84
Boolean Values in C99	85
<b>5.3 The switch Statement</b>	<b>86</b>
The Role of the break Statement	88
Program: Printing a Date in Legal Form	89
<b>6 LOOPS</b>	<b>99</b>
<b>6.1 The while Statement</b>	<b>99</b>
Infinite Loops	101
Program: Printing a Table of Squares	102
Program: Summing a Series of Numbers	102
<b>6.2 The do Statement</b>	<b>103</b>
Program: Calculating the Number of Digits in an Integer	104
<b>6.3 The for Statement</b>	<b>105</b>
for Statement Idioms	106
Omitting Expressions in a for Statement	107
for Statements in C99	108
The Comma Operator	109
Program: Printing a Table of Squares (Revisited)	110
<b>6.4 Exiting from a Loop</b>	<b>111</b>
The break Statement	111
The continue Statement	112
The goto Statement	113
Program: Balancing a Checkbook	114
<b>6.5 The Null Statement</b>	<b>116</b>
<b>7 BASIC TYPES</b>	<b>125</b>
<b>7.1 Integer Types</b>	<b>125</b>
Integer Types in C99	128
Integer Constants	128
Integer Constants in C99	129
Integer Overflow	130
Reading and Writing Integers	130
Program: Summing a Series of Numbers (Revisited)	131
<b>7.2 Floating Types</b>	<b>132</b>
Floating Constants	133
Reading and Writing Floating-Point Numbers	134
<b>7.3 Character Types</b>	<b>134</b>
Operations on Characters	135
Signed and Unsigned Characters	136

Arithmetic Types	136
Escape Sequences	137
Character-Handling Functions	138
Reading and Writing Characters using <code>scanf</code> and <code>printf</code>	139
Reading and Writing Characters using <code>getchar</code> and <code>putchar</code>	140
Program: Determining the Length of a Message	141
<b>7.4 Type Conversion</b>	<b>142</b>
The Usual Arithmetic Conversions	143
Conversion During Assignment	145
Implicit Conversions in C99	146
Casting	147
<b>7.5 Type Definitions</b>	<b>149</b>
Advantages of Type Definitions	149
Type Definitions and Portability	150
<b>7.6 The <code>sizeof</code> Operator</b>	<b>151</b>
<b>8 ARRAYS</b>	<b>161</b>
<b>8.1 One-Dimensional Arrays</b>	<b>161</b>
Array Subscripting	162
Program: Reversing a Series of Numbers	164
Array Initialization	164
Designated Initializers	165
Program: Checking a Number for Repeated Digits	166
Using the <code>sizeof</code> Operator with Arrays	167
Program: Computing Interest	168
<b>8.2 Multidimensional Arrays</b>	<b>169</b>
Initializing a Multidimensional Array	171
Constant Arrays	172
Program: Dealing a Hand of Cards	172
<b>8.3 Variable-Length Arrays (C99)</b>	<b>174</b>
<b>9 FUNCTIONS</b>	<b>183</b>
<b>9.1 Defining and Calling Functions</b>	<b>183</b>
Program: Computing Averages	184
Program: Printing a Countdown	185
Program: Printing a Pun (Revisited)	186
Function Definitions	187
Function Calls	189
Program: Testing Whether a Number Is Prime	190
<b>9.2 Function Declarations</b>	<b>191</b>
<b>9.3 Arguments</b>	<b>193</b>
Argument Conversions	194
Array Arguments	195
Variable-Length Array Parameters	198

Using <code>static</code> in Array Parameter Declarations	200
Compound Literals	200
<b>9.4 The <code>return</code> Statement</b>	<b>201</b>
<b>9.5 Program Termination</b>	<b>202</b>
The <code>exit</code> Function	203
<b>9.6 Recursion</b>	<b>204</b>
The Quicksort Algorithm	205
Program: Quicksort	207
<b>10 PROGRAM ORGANIZATION</b>	<b>219</b>
<b>10.1 Local Variables</b>	<b>219</b>
Static Local Variables	220
Parameters	221
<b>10.2 External Variables</b>	<b>221</b>
Example: Using External Variables to Implement a Stack	221
Pros and Cons of External Variables	222
Program: Guessing a Number	224
<b>10.3 Blocks</b>	<b>227</b>
<b>10.4 Scope</b>	<b>228</b>
<b>10.5 Organizing a C Program</b>	<b>229</b>
Program: Classifying a Poker Hand	230
<b>11 POINTERS</b>	<b>241</b>
<b>11.1 Pointer Variables</b>	<b>241</b>
Declaring Pointer Variables	242
<b>11.2 The Address and Indirection Operators</b>	<b>243</b>
The Address Operator	243
The Indirection Operator	244
<b>11.3 Pointer Assignment</b>	<b>245</b>
<b>11.4 Pointers as Arguments</b>	<b>247</b>
Program: Finding the Largest and Smallest Elements in an Array	249
Using <code>const</code> to Protect Arguments	250
<b>11.5 Pointers as Return Values</b>	<b>251</b>
<b>12 POINTERS AND ARRAYS</b>	<b>257</b>
<b>12.1 Pointer Arithmetic</b>	<b>257</b>
Adding an Integer to a Pointer	258
Subtracting an Integer from a Pointer	259
Subtracting One Pointer from Another	259
Comparing Pointers	260
Pointers to Compound Literals	260
<b>12.2 Using Pointers for Array Processing</b>	<b>260</b>
Combining the <code>*</code> and <code>++</code> Operators	262

<b>12.3</b>	<b>Using an Array Name as a Pointer</b>	<b>263</b>
	Program: Reversing a Series of Numbers (Revisited)	264
	Array Arguments (Revisited)	265
	Using a Pointer as an Array Name	266
<b>12.4</b>	<b>Pointers and Multidimensional Arrays</b>	<b>267</b>
	Processing the Elements of a Multidimensional Array	267
	Processing the Rows of a Multidimensional Array	268
	Processing the Columns of a Multidimensional Array	269
	Using the Name of a Multidimensional Array as a Pointer	269
<b>12.5</b>	<b>Pointers and Variable-Length Arrays (C99)</b>	<b>270</b>
<b>13</b>	<b>STRINGS</b>	<b>277</b>
<b>13.1</b>	<b>String Literals</b>	<b>277</b>
	Escape Sequences in String Literals	278
	Continuing a String Literal	278
	How String Literals Are Stored	279
	Operations on String Literals	279
	String Literals versus Character Constants	280
<b>13.2</b>	<b>String Variables</b>	<b>281</b>
	Initializing a String Variable	281
	Character Arrays versus Character Pointers	283
<b>13.3</b>	<b>Reading and Writing Strings</b>	<b>284</b>
	Writing Strings Using <code>printf</code> and <code>puts</code>	284
	Reading Strings Using <code>scanf</code> and <code>gets</code>	285
	Reading Strings Character by Character	286
<b>13.4</b>	<b>Accessing the Characters in a String</b>	<b>287</b>
<b>13.5</b>	<b>Using the C String Library</b>	<b>289</b>
	The <code>strcpy</code> (String Copy) Function	290
	The <code>strlen</code> (String Length) Function	291
	The <code>strcat</code> (String Concatenation) Function	291
	The <code>strcmp</code> (String Comparison) Function	292
	Program: Printing a One-Month Reminder List	293
<b>13.6</b>	<b>String Idioms</b>	<b>296</b>
	Searching for the End of a String	296
	Copying a String	298
<b>13.7</b>	<b>Arrays of Strings</b>	<b>300</b>
	Command-Line Arguments	302
	Program: Checking Planet Names	303
<b>14</b>	<b>THE PREPROCESSOR</b>	<b>315</b>
<b>14.1</b>	<b>How the Preprocessor Works</b>	<b>315</b>
<b>14.2</b>	<b>Preprocessing Directives</b>	<b>318</b>
<b>14.3</b>	<b>Macro Definitions</b>	<b>319</b>
	Simple Macros	319
	Parameterized Macros	321

The # Operator	324
The ## Operator	324
General Properties of Macros	325
Parentheses in Macro Definitions	326
Creating Longer Macros	328
Predefined Macros	329
Additional Predefined Macros in C99	330
Empty Macro Arguments	331
Macros with a Variable Number of Arguments	332
The <code>_func_</code> Identifier	333
<b>14.4 Conditional Compilation</b>	<b>333</b>
The <code>#if</code> and <code>#endif</code> Directives	334
The <code>defined</code> Operator	335
The <code>#ifdef</code> and <code>#ifndef</code> Directives	335
The <code>#elif</code> and <code>#else</code> Directives	336
Uses of Conditional Compilation	337
<b>14.5 Miscellaneous Directives</b>	<b>338</b>
The <code>#error</code> Directive	338
The <code>#line</code> Directive	339
The <code>#pragma</code> Directive	340
The <code>_Pragma</code> Operator	341
<b>15 WRITING LARGE PROGRAMS</b>	<b>349</b>
<b>15.1 Source Files</b>	<b>349</b>
<b>15.2 Header Files</b>	<b>350</b>
The <code>#include</code> Directive	351
Sharing Macro Definitions and Type Definitions	353
Sharing Function Prototypes	354
Sharing Variable Declarations	355
Nested Includes	357
Protecting Header Files	357
<code>#error</code> Directives in Header Files	358
<b>15.3 Dividing a Program into Files</b>	<b>359</b>
Program: Text Formatting	359
<b>15.4 Building a Multiple-File Program</b>	<b>366</b>
Makefiles	366
Errors During Linking	368
Rebuilding a Program	369
Defining Macros Outside a Program	371
<b>16 STRUCTURES, UNIONS, AND ENUMERATIONS</b>	<b>377</b>
<b>16.1 Structure Variables</b>	<b>377</b>
Declaring Structure Variables	378
Initializing Structure Variables	379
Designated Initializers	380
Operations on Structures	381

<b>16.2</b>	<b>Structure Types</b>	<b>382</b>
	Declaring a Structure Tag	383
	Defining a Structure Type	384
	Structures as Arguments and Return Values	384
	Compound Literals	386
<b>16.3</b>	<b>Nested Arrays and Structures</b>	<b>386</b>
	Nested Structures	387
	Arrays of Structures	387
	Initializing an Array of Structures	388
	Program: Maintaining a Parts Database	389
<b>16.4</b>	<b>Unions</b>	<b>396</b>
	Using Unions to Save Space	398
	Using Unions to Build Mixed Data Structures	399
	Adding a "Tag Field" to a Union	400
<b>16.5</b>	<b>Enumerations</b>	<b>401</b>
	Enumeration Tags and Type Names	402
	Enumerations as Integers	403
	Using Enumerations to Declare "Tag Fields"	404
<b>17</b>	<b>ADVANCED USES OF POINTERS</b>	<b>413</b>
<b>17.1</b>	<b>Dynamic Storage Allocation</b>	<b>414</b>
	Memory Allocation Functions	414
	Null Pointers	414
<b>17.2</b>	<b>Dynamically Allocated Strings</b>	<b>416</b>
	Using <code>malloc</code> to Allocate Memory for a String	416
	Using Dynamic Storage Allocation in String Functions	417
	Arrays of Dynamically Allocated Strings	418
	Program: Printing a One-Month Reminder List (Revisited)	418
<b>17.3</b>	<b>Dynamically Allocated Arrays</b>	<b>420</b>
	Using <code>malloc</code> to Allocate Storage for an Array	420
	The <code>calloc</code> Function	421
	The <code>realloc</code> Function	421
<b>17.4</b>	<b>Deallocating Storage</b>	<b>422</b>
	The <code>free</code> Function	423
	The "Dangling Pointer" Problem	424
<b>17.5</b>	<b>Linked Lists</b>	<b>424</b>
	Declaring a Node Type	425
	Creating a Node	425
	The <code>-&gt;</code> Operator	426
	Inserting a Node at the Beginning of a Linked List	427
	Searching a Linked List	429
	Deleting a Node from a Linked List	431
	Ordered Lists	433
	Program: Maintaining a Parts Database (Revisited)	433
<b>17.6</b>	<b>Pointers to Pointers</b>	<b>438</b>

17.7	<b>Pointers to Functions</b>	439
	Function Pointers as Arguments	439
	The <code>qsort</code> Function	440
	Other Uses of Function Pointers	442
	Program: Tabulating the Trigonometric Functions	443
17.8	<b>Restricted Pointers (C99)</b>	445
17.9	<b>Flexible Array Members (C99)</b>	447
<b>18</b>	<b>DECLARATIONS</b>	<b>457</b>
18.1	<b>Declaration Syntax</b>	457
18.2	<b>Storage Classes</b>	459
	Properties of Variables	459
	The <code>auto</code> Storage Class	460
	The <code>static</code> Storage Class	461
	The <code>extern</code> Storage Class	462
	The <code>register</code> Storage Class	463
	The Storage Class of a Function	464
	Summary	465
18.3	<b>Type Qualifiers</b>	466
18.4	<b>Declarators</b>	467
	Deciphering Complex Declarations	468
	Using Type Definitions to Simplify Declarations	470
18.5	<b>Initializers</b>	470
	Uninitialized Variables	472
18.6	<b>Inline Functions (C99)</b>	472
	Inline Definitions	473
	Restrictions on Inline Functions	474
	Using Inline Functions with GCC	475
<b>19</b>	<b>PROGRAM DESIGN</b>	<b>483</b>
19.1	<b>Modules</b>	484
	Cohesion and Coupling	486
	Types of Modules	486
19.2	<b>Information Hiding</b>	487
	A Stack Module	487
19.3	<b>Abstract Data Types</b>	491
	Encapsulation	492
	Incomplete Types	492
19.4	<b>A Stack Abstract Data Type</b>	493
	Defining the Interface for the Stack ADT	493
	Implementing the Stack ADT Using a Fixed-Length Array	495
	Changing the Item Type in the Stack ADT	496
	Implementing the Stack ADT Using a Dynamic Array	497
	Implementing the Stack ADT Using a Linked List	499

19.5	<b>Design Issues for Abstract Data Types</b>	502
	Naming Conventions	502
	Error Handling	502
	Generic ADTs	503
	ADTs in Newer Languages	503
<b>20</b>	<b>LOW-LEVEL PROGRAMMING</b>	<b>509</b>
20.1	<b>Bitwise Operators</b>	509
	Bitwise Shift Operators	510
	Bitwise Complement, <i>And</i> , Exclusive <i>Or</i> , and Inclusive <i>Or</i>	511
	Using the Bitwise Operators to Access Bits	512
	Using the Bitwise Operators to Access Bit-Fields	513
	Program: XOR Encryption	514
20.2	<b>Bit-Fields in Structures</b>	516
	How Bit-Fields Are Stored	517
20.3	<b>Other Low-Level Techniques</b>	518
	Defining Machine-Dependent Types	518
	Using Unions to Provide Multiple Views of Data	519
	Using Pointers as Addresses	520
	Program: Viewing Memory Locations	521
	The <i>volatile</i> Type Qualifier	523
<b>21</b>	<b>THE STANDARD LIBRARY</b>	<b>529</b>
21.1	<b>Using the Library</b>	529
	Restrictions on Names Used in the Library	530
	Functions Hidden by Macros	531
21.2	<b>C89 Library Overview</b>	531
21.3	<b>C99 Library Changes</b>	534
21.4	<b>The &lt;stddef.h&gt; Header: Common Definitions</b>	535
21.5	<b>The &lt;stdbool.h&gt; Header (C99): Boolean Type and Values</b>	536
<b>22</b>	<b>INPUT/OUTPUT</b>	<b>539</b>
22.1	<b>Streams</b>	540
	File Pointers	540
	Standard Streams and Redirection	540
	Text Files versus Binary Files	541
22.2	<b>File Operations</b>	543
	Opening a File	543
	Modes	544
	Closing a File	545
	Attaching a File to an Open Stream	546
	Obtaining File Names from the Command Line	546
	Program: Checking Whether a File Can Be Opened	547

Temporary Files	548
File Buffering	549
Miscellaneous File Operations	551
<b>22.3 Formatted I/O</b>	<b>551</b>
The ...printf Functions	552
...printf Conversion Specifications	552
C99 Changes to ...printf Conversion Specifications	555
Examples of ...printf Conversion Specifications	556
The ...scanf Functions	558
...scanf Format Strings	559
...scanf Conversion Specifications	560
C99 Changes to ...scanf Conversion Specifications	562
scanf Examples	563
Detecting End-of-File and Error Conditions	564
<b>22.4 Character I/O</b>	<b>566</b>
Output Functions	566
Input Functions	567
Program: Copying a File	568
<b>22.5 Line I/O</b>	<b>569</b>
Output Functions	569
Input Functions	570
<b>22.6 Block I/O</b>	<b>571</b>
<b>22.7 File Positioning</b>	<b>572</b>
Program: Modifying a File of Part Records	574
<b>22.8 String I/O</b>	<b>575</b>
Output Functions	576
Input Functions	576
<b>23 LIBRARY SUPPORT FOR NUMBERS AND CHARACTER DATA</b>	<b>589</b>
<b>23.1 The &lt;float.h&gt; Header: Characteristics of Floating Types</b>	<b>589</b>
<b>23.2 The &lt;limits.h&gt; Header: Sizes of Integer Types</b>	<b>591</b>
<b>23.3 The &lt;math.h&gt; Header (C89): Mathematics</b>	<b>593</b>
Errors	593
Trigonometric Functions	594
Hyperbolic Functions	595
Exponential and Logarithmic Functions	595
Power Functions	596
Nearest Integer, Absolute Value, and Remainder Functions	596
<b>23.4 The &lt;math.h&gt; Header (C99): Mathematics</b>	<b>597</b>
IEEE Floating-Point Standard	598
Types	599
Macros	600

Errors	600
Functions	601
Classification Macros	602
Trigonometric Functions	603
Hyperbolic Functions	603
Exponential and Logarithmic Functions	604
Power and Absolute Value Functions	605
Error and Gamma Functions	606
Nearest Integer Functions	606
Remainder Functions	608
Manipulation Functions	608
Maximum, Minimum, and Positive Difference Functions	609
Floating Multiply-Add	610
Comparison Macros	611
<b>23.5 The &lt;ctype.h&gt; Header: Character Handling</b>	<b>612</b>
Character-Classification Functions	612
Program: Testing the Character-Classification Functions	613
Character Case-Mapping Functions	614
Program: Testing the Case-Mapping Functions	614
<b>23.6 The &lt;string.h&gt; Header: String Handling</b>	<b>615</b>
Copying Functions	616
Concatenation Functions	617
Comparison Functions	617
Search Functions	619
Miscellaneous Functions	622
<b>24 ERROR HANDLING</b>	<b>627</b>
<b>24.1 The &lt;assert.h&gt; Header: Diagnostics</b>	<b>628</b>
<b>24.2 The &lt;errno.h&gt; Header: Errors</b>	<b>629</b>
The perror and strerror Functions	630
<b>24.3 The &lt;signal.h&gt; Header: Signal Handling</b>	<b>631</b>
Signal Macros	631
The signal Function	632
Predefined Signal Handlers	633
The raise Function	634
Program: Testing Signals	634
<b>24.4 The &lt;setjmp.h&gt; Header: Nonlocal Jumps</b>	<b>635</b>
Program: Testing setjmp/longjmp	636
<b>25 INTERNATIONAL FEATURES</b>	<b>641</b>
<b>25.1 The &lt;locale.h&gt; Header: Localization</b>	<b>642</b>
Categories	642
The setlocale Function	643
The localeconv Function	644
<b>25.2 Multibyte Characters and Wide Characters</b>	<b>647</b>

Multibyte Characters	648
Wide Characters	649
Unicode and the Universal Character Set	649
Encodings of Unicode	650
Multibyte/Wide-Character Conversion Functions	651
Multibyte/Wide-String Conversion Functions	653
<b>25.3 Digraphs and Trigraphs</b>	<b>654</b>
Trigraphs	654
Digraphs	655
The <iso646.h> Header: Alternative Spellings	656
<b>25.4 Universal Character Names (C99)</b>	<b>656</b>
<b>25.5 The &lt;wchar.h&gt; Header (C99): Extended Multibyte and Wide-Character Utilities</b>	<b>657</b>
Stream Orientation	658
Formatted Wide-Character Input/Output Functions	659
Wide-Character Input/Output Functions	661
General Wide-String Utilities	662
Wide-Character Time-Conversion Functions	667
Extended Multibyte/Wide-Character Conversion Utilities	667
<b>25.6 The &lt;wctype.h&gt; Header (C99): Wide-Character Classification and Mapping Utilities</b>	<b>671</b>
Wide-Character Classification Functions	671
Extensible Wide-Character Classification Functions	672
Wide-Character Case-Mapping Functions	673
Extensible Wide-Character Case-Mapping Functions	673
<b>26 MISCELLANEOUS LIBRARY FUNCTIONS</b>	<b>677</b>
<b>26.1 The &lt;stdarg.h&gt; Header: Variable Arguments</b>	<b>677</b>
Calling a Function with a Variable Argument List	679
The v...printf Functions	680
The v...scanf Functions	681
<b>26.2 The &lt;stdlib.h&gt; Header: General Utilities</b>	<b>682</b>
Numeric Conversion Functions	682
Program: Testing the Numeric Conversion Functions	684
Pseudo-Random Sequence Generation Functions	686
Program: Testing the Pseudo-Random Sequence Generation Functions	687
Communication with the Environment	687
Searching and Sorting Utilities	689
Program: Determining Air Mileage	690
Integer Arithmetic Functions	691
<b>26.3 The &lt;time.h&gt; Header: Date and Time</b>	<b>692</b>
Time Manipulation Functions	693
Time Conversion Functions	695
Program: Displaying the Date and Time	698

<b>27 ADDITIONAL C99 SUPPORT FOR MATHEMATICS</b>	<b>705</b>
<b>27.1 The &lt;stdint.h&gt; Header (C99): Integer Types</b>	<b>705</b>
<stdint.h> Types	706
Limits of Specified-Width Integer Types	707
Limits of Other Integer Types	708
Macros for Integer Constants	708
<b>27.2 The &lt;inttypes.h&gt; Header (C99): Format Conversion of Integer Types</b>	<b>709</b>
Macros for Format Specifiers	710
Functions for Greatest-Width Integer Types	711
<b>27.3 Complex Numbers (C99)</b>	<b>712</b>
Definition of Complex Numbers	713
Complex Arithmetic	714
Complex Types in C99	714
Operations on Complex Numbers	715
Conversion Rules for Complex Types	715
<b>27.4 The &lt;complex.h&gt; Header (C99): Complex Arithmetic</b>	<b>717</b>
<complex.h> Macros	717
The CX_LIMITED_RANGE Pragma	718
<complex.h> Functions	718
Trigonometric Functions	719
Hyperbolic Functions	720
Exponential and Logarithmic Functions	721
Power and Absolute-Value Functions	721
Manipulation Functions	722
Program: Finding the Roots of a Quadratic Equation	722
<b>27.5 The &lt;tgmath.h&gt; Header (C99): Type-Generic Math</b>	<b>723</b>
Type-Generic Macros	724
Invoking a Type-Generic Macro	725
<b>27.6 The &lt;fenv.h&gt; Header (C99): Floating-Point Environment</b>	<b>726</b>
Floating-Point Status Flags and Control Modes	727
<fenv.h> Macros	727
The FENV_ACCESS Pragma	728
Floating-Point Exception Functions	729
Rounding Functions	730
Environment Functions	730
<b>Appendix A C Operators</b>	<b>735</b>
<b>Appendix B C99 versus C89</b>	<b>737</b>
<b>Appendix C C89 versus K&amp;R C</b>	<b>743</b>
<b>Appendix D Standard Library Functions</b>	<b>747</b>
<b>Appendix E ASCII Character Set</b>	<b>801</b>
<b>Bibliography</b>	<b>803</b>
<b>Index</b>	<b>807</b>

# 1

# Introducing C

*When someone says "I want a programming language in which I need only say what I wish done," give him a lollipop.\**

What is C? The simple answer—a widely used programming language developed in the early 1970s at Bell Laboratories—conveys little of C’s special flavor. Before we become immersed in the details of the language, let’s take a look at where C came from, what it was designed for, and how it has changed over the years (Section 1.1). We’ll also discuss C’s strengths and weaknesses and see how to get the most out of the language (Section 1.2).

## 1.1 History of C

Let’s take a quick look at C’s history, from its origins, to its coming of age as a standardized language, to its influence on recent languages.

### Origins

C is a by-product of the UNIX operating system, which was developed at Bell Laboratories by Ken Thompson, Dennis Ritchie, and others. Thompson single-handedly wrote the original version of UNIX, which ran on the DEC PDP-7 computer, an early minicomputer with only 8K words of main memory (this was 1969, after all!).

Like other operating systems of the time, UNIX was written in assembly language. Programs written in assembly language are usually painful to debug and hard to enhance; UNIX was no exception. Thompson decided that a higher-level

\*The epigrams at the beginning of each chapter are from “Epigrams on Programming” by Alan J. Perlis (*ACM SIGPLAN Notices* (September, 1982): 7–13).

language was needed for the further development of UNIX, so he designed a small language named B. Thompson based B on BCPL, a systems programming language developed in the mid-1960s. BCPL, in turn, traces its ancestry to Algol 60, one of the earliest (and most influential) programming languages.

Ritchie soon joined the UNIX project and began programming in B. In 1970, Bell Labs acquired a PDP-11 for the UNIX project. Once B was up and running on the PDP-11, Thompson rewrote a portion of UNIX in B. By 1971, it became apparent that B was not well-suited to the PDP-11, so Ritchie began to develop an extended version of B. He called his language NB ("New B") at first, and then, as it began to diverge more from B, he changed the name to C. The language was stable enough by 1973 that UNIX could be rewritten in C. The switch to C provided an important benefit: portability. By writing C compilers for other computers at Bell Labs, the team could get UNIX running on those machines as well.

## Standardization

C continued to evolve during the 1970s, especially between 1977 and 1979. It was during this period that the first book on C appeared. *The C Programming Language*, written by Brian Kernighan and Dennis Ritchie and published in 1978, quickly became the bible of C programmers. In the absence of an official standard for C, this book—known as K&R or the “White Book” to aficionados—served as a de facto standard.

During the 1970s, there were relatively few C programmers, and most of them were UNIX users. By the 1980s, however, C had expanded beyond the narrow confines of the UNIX world. C compilers became available on a variety of machines running under different operating systems. In particular, C began to establish itself on the fast-growing IBM PC platform.

With C's increasing popularity came problems. Programmers who wrote new C compilers relied on K&R as a reference. Unfortunately, K&R was fuzzy about some language features, so compilers often treated these features differently. Also, K&R failed to make a clear distinction between which features belonged to C and which were part of UNIX. To make matters worse, C continued to change after K&R was published, with new features being added and a few older features removed. The need for a thorough, precise, and up-to-date description of the language soon became apparent. Without such a standard, numerous dialects would have arisen, threatening the portability of C programs, one of the language's major strengths.

The development of a U.S. standard for C began in 1983 under the auspices of the American National Standards Institute (ANSI). After many revisions, the standard was completed in 1988 and formally approved in December 1989 as ANSI standard X3.159-1989. In 1990, it was approved by the International Organization for Standardization (ISO) as international standard ISO/IEC 9899:1990. This version of the language is usually referred to as C89 or C90, to distinguish it from the

original version of C, often called K&R C. Appendix C summarizes the major differences between C89 and K&R C.

The language underwent a few changes in 1995 (described in a document known as Amendment 1). More significant changes occurred with the publication of a new standard, ISO/IEC 9899:1999, in 1999. The language described in this standard is commonly known as C99. The terms “ANSI C,” “ANSI/ISO C,” and “ISO C”—once used to describe C89—are now ambiguous, thanks to the existence of two standards.

Because C99 isn’t yet universal, and because of the need to maintain millions (if not billions) of lines of code written in older versions of C, I’ll use a special icon (shown in the left margin) to mark discussions of features that were added in C99. A compiler that doesn’t recognize these features isn’t “C99-compliant.” If history is any guide, it will be some years before all C compilers are C99-compliant, if they ever are. Appendix B lists the major differences between C99 and C89.

C99

## C-Based Languages

C has had a huge influence on modern-day programming languages, many of which borrow heavily from it. Of the many C-based languages, several are especially prominent:

- **C++** includes all the features of C, but adds classes and other features to support object-oriented programming.
- **Java** is based on C++ and therefore inherits many C features.
- **C#** is a more recent language derived from C++ and Java.
- **Perl** was originally a fairly simple scripting language; over time it has grown and adopted many of the features of C.

Considering the popularity of these newer languages, it’s logical to ask whether it’s worth the trouble to learn C. I think it is, for several reasons. First, learning C can give you greater insight into the features of C++, Java, C#, Perl, and the other C-based languages. Programmers who learn one of these languages first often fail to master basic features that were inherited from C. Second, there are a lot of older C programs around; you may find yourself needing to read and maintain this code. Third, C is still widely used for developing new software, especially in situations where memory or processing power is limited or where the simplicity of C is desired.

If you haven’t already used one of the newer C-based languages, you’ll find that this book is excellent preparation for learning these languages. It emphasizes data abstraction, information hiding, and other principles that play a large role in object-oriented programming. C++ includes all the features of C, so you’ll be able to use everything you learn from this book if you later tackle C++. Many of the features of C can be found in the other C-based languages as well.

## 1.2 Strengths and Weaknesses of C

Like any other programming language, C has strengths and weaknesses. Both stem from the language’s original use (writing operating systems and other systems software) and its underlying philosophy:

- **C is a low-level language.** To serve as a suitable language for systems programming, C provides access to machine-level concepts (bytes and addresses, for example) that other programming languages try to hide. C also provides operations that correspond closely to a computer’s built-in instructions, so that programs can be fast. Since application programs rely on it for input/output, storage management, and numerous other services, an operating system can’t afford to be slow.
- **C is a small language.** C provides a more limited set of features than many languages. (The reference manual in the second edition of K&R covers the entire language in 49 pages.) To keep the number of features small, C relies heavily on a “library” of standard functions. (A “function” is similar to what other programming languages might call a “procedure,” “subroutine,” or “method.”)
- **C is a permissive language.** C assumes that you know what you’re doing, so it allows you a wider degree of latitude than many languages. Moreover, C doesn’t mandate the detailed error-checking found in other languages.

### Strengths

C’s strengths help explain why the language has become so popular:

- **Efficiency.** Efficiency has been one of C’s advantages from the beginning. Because C was intended for applications where assembly language had traditionally been used, it was crucial that C programs could run quickly and in limited amounts of memory.
- **Portability.** Although program portability wasn’t a primary goal of C, it has turned out to be one of the language’s strengths. When a program must run on computers ranging from PCs to supercomputers, it is often written in C. One reason for the portability of C programs is that—thanks to C’s early association with UNIX and the later ANSI/ISO standards—the language hasn’t splintered into incompatible dialects. Another is that C compilers are small and easily written, which has helped make them widely available. Finally, C itself has features that support portability (although there’s nothing to prevent programmers from writing nonportable programs).
- **Power.** C’s large collection of data types and operators help make it a powerful language. In C, it’s often possible to accomplish quite a bit with just a few lines of code.

- **Flexibility.** Although C was originally designed for systems programming, it has no inherent restrictions that limit it to this arena. C is now used for applications of all kinds, from embedded systems to commercial data processing. Moreover, C imposes very few restrictions on the use of its features; operations that would be illegal in other languages are often permitted in C. For example, C allows a character to be added to an integer value (or, for that matter, a floating-point number). This flexibility can make programming easier, although it may allow some bugs to slip through.
- **Standard library.** One of C's great strengths is its standard library, which contains hundreds of functions for input/output, string handling, storage allocation, and other useful operations.
- **Integration with UNIX.** C is particularly powerful in combination with UNIX (including the popular variant known as Linux). In fact, some UNIX tools assume that the user knows C.

## Weaknesses

C's weaknesses arise from the same source as many of its strengths: C's closeness to the machine. Here are a few of C's most notorious problems:

- **C programs can be error-prone.** C's flexibility makes it an error-prone language. Programming mistakes that would be caught in many other languages can't be detected by a C compiler. In this respect, C is a lot like assembly language, where most errors aren't detected until the program is run. To make matters worse, C contains a number of pitfalls for the unwary. In later chapters, we'll see how an extra semicolon can create an infinite loop or a missing & symbol can cause a program crash.
- **C programs can be difficult to understand.** Although C is a small language by most measures, it has a number of features that aren't found in all programming languages (and that consequently are often misunderstood). These features can be combined in a great variety of ways, many of which—although obvious to the original author of a program—can be hard for others to understand. Another problem is the terse nature of C programs. C was designed at a time when interactive communication with computers was tedious at best. As a result, C was purposefully kept terse to minimize the time required to enter and edit programs. C's flexibility can also be a negative factor; programmers who are too clever for their own good can make programs almost impossible to understand.
- **C programs can be difficult to modify.** Large programs written in C can be hard to change if they haven't been designed with maintenance in mind. Modern programming languages usually provide features such as classes and packages that support the division of a large program into more manageable pieces. C, unfortunately, lacks such features.

## Obfuscated C

Even C's most ardent admirers admit that C code can be hard to read. The annual International Obfuscated C Code Contest actually encourages contestants to write the most confusing C programs possible. The winners are truly baffling, as 1990's "Best Small Program" shows:

```
v,i,j,k,l,s,a[99];
main()
{
    for (scanf ("%d", &s); *a-s; v=a[j*=v]-a[i], k=i<s, j+=(v=j<s&&
        (!k&&!printf(2+"\\n\\n%c"-(!l<<!j), "#Q"[l^v?(l^j)&1:2])&&
        ++l||a[i]<s&&v&&v-i+j&&v+i-j))&&! (l%=s), v) | (i==j?a[i+=k]=0:
        ++a[i])>=s*k&&++a[--i])
    ;
}
```

This program, written by Doron Osovianski and Baruch Nissenbaum, prints all solutions to the Eight Queens problem (the problem of placing eight queens on a chessboard in such a way that no queen attacks any other queen). In fact, it works for any number of queens between four and 99. For more winning programs, visit [www.ioccc.org](http://www.ioccc.org), the contest's web site.

---

## Effective Use of C

Using C effectively requires taking advantage of C's strengths while avoiding its weaknesses. Here are a few suggestions:

- **Learn how to avoid C pitfalls.** Hints for avoiding pitfalls are scattered throughout this book—just look for the  $\Delta$  symbol. For a more extensive list of pitfalls, see Andrew Koenig's *C Traps and Pitfalls* (Reading, Mass.: Addison-Wesley, 1989). Modern compilers will detect common pitfalls and issue warnings, but no compiler spots them all.
- **Use software tools to make programs more reliable.** C programmers are prolific tool builders (and users). One of the most famous C tools is named `lint`. `lint`, which is traditionally provided with UNIX, can subject a program to a more extensive error analysis than most C compilers. If `lint` (or a similar program) is available, it's a good idea to use it. Another useful tool is a debugger. Because of the nature of C, many bugs can't be detected by a C compiler; these show up instead in the form of run-time errors or incorrect output. Consequently, using a good debugger is practically mandatory for C programmers.
- **Take advantage of existing code libraries.** One of the benefits of using C is that so many other people also use it; it's a good bet that they've written code you can employ in your own programs. C code is often bundled into libraries (collections of functions); obtaining a suitable library is a good way to reduce errors—and save considerable programming effort. Libraries for common

### Q&A

tasks, including user-interface development, graphics, communications, database management, and networking, are readily available. Some libraries are in the public domain, some are open source, and some are sold commercially.

- **Adopt a sensible set of coding conventions.** A coding convention is a style rule that a programmer has decided to adopt even though it's not enforced by the language. Well-chosen conventions help make programs more uniform, easier to read, and easier to modify. Conventions are important when using any programming language, but especially so with C. As noted above, C's highly flexible nature makes it possible for programmers to write code that is all but unreadable. The programming examples in this book follow one set of conventions, but there are other, equally valid, conventions in use. (We'll discuss some of the alternatives from time to time.) Which set you use is less important than adopting *some* conventions and sticking to them.
- **Avoid "tricks" and overly complex code.** C encourages programming tricks. There are usually several ways to accomplish a given task in C; programmers are often tempted to choose the method that's most concise. Don't get carried away; the shortest solution is often the hardest to comprehend. In this book, I'll illustrate a style that's reasonably concise but still understandable.
- **Stick to the standard.** Most C compilers provide language features and library functions that aren't part of the C89 or C99 standards. For portability, it's best to avoid using nonstandard features and libraries unless they're absolutely necessary.

## Q & A

### Q: What is this Q&A section anyway?

A: Glad you asked. The Q&A section, which appears at the end of each chapter, serves several purposes.

The primary purpose of Q&A is to tackle questions that are frequently asked by students learning C. Readers can participate in a dialogue (more or less) with the author, much the same as if they were attending one of my C classes.

Another purpose of Q&A is to provide additional information about topics covered in the chapter. Readers of this book will likely have widely varying backgrounds. Some will be experienced in other programming languages, whereas others will be learning to program for the first time. Readers with experience in a variety of languages may be satisfied with a brief explanation and a couple of examples, but readers with less experience may need more. The bottom line: If you find the coverage of a topic to be sketchy, check Q&A for more details.

On occasion, Q&A will discuss common differences among C compilers. For example, we'll cover some frequently used (but nonstandard) features that are provided by particular compilers.

**Q: What does `lint` do? [p. 6]**

A: `lint` checks a C program for a host of potential errors, including—but not limited to—suspicious combinations of types, unused variables, unreachable code, and nonportable code. It produces a list of diagnostic messages, which the programmer must then sift through. The advantage of using `lint` is that it can detect errors that are missed by the compiler. On the other hand, you've got to remember to use `lint`; it's all too easy to forget about it. Worse still, `lint` can produce messages by the hundreds, of which only a fraction refer to actual errors.

**Q: Where did `lint` get its name?**

A: Unlike the names of many other UNIX tools, `lint` isn't an acronym; it got its name from the way it picks up pieces of "fluff" from a program.

**Q: How do I get a copy of `lint`?**

A: `lint` is a standard UNIX utility; if you rely on another operating system, then you probably don't have `lint`. Fortunately, versions of `lint` are available from third parties. An enhanced version of `lint` known as `splint` (Secure Programming Lint) is included in many Linux distributions and can be downloaded for free from [www.splint.org](http://www.splint.org).

**Q: Is there some way to force a compiler to do a more thorough job of error-checking, without having to use `lint`?**

A: Yes. Most compilers will do a more thorough check of a program if asked to. In addition to checking for errors (undisputed violations of the rules of C), most compilers also produce warning messages, indicating potential trouble spots. Some compilers have more than one "warning level"; selecting a higher level causes the compiler to check for more problems than choosing a lower level. If your compiler supports warning levels, it's a good idea to select the highest level, causing the compiler to perform the most thorough job of checking that it's capable of. Error-checking options for the GCC compiler, which is distributed with Linux, are discussed in the Q&A section at the end of Chapter 2.

**\*Q: I'm interested in making my program as reliable as possible. Are there any other tools available besides `lint` and debuggers?**

A: Yes. Other common tools include "bounds-checkers" and "leak-finders." C doesn't require that array subscripts be checked; a bounds-checker adds this capability. A leak-finder helps locate "memory leaks": blocks of memory that are dynamically allocated but never deallocated.

---

\*Starred questions cover material too advanced or too esoteric to interest the average reader, and often refer to topics covered in later chapters. Curious readers with a fair bit of programming experience may wish to delve into these questions immediately; others should definitely skip them on a first reading.

# 2 C Fundamentals

*One man's constant is another man's variable.*

This chapter introduces several basic concepts, including preprocessing directives, functions, variables, and statements, that we'll need in order to write even the simplest programs. Later chapters will cover these topics in much greater detail.

To start off, Section 2.1 presents a small C program and describes how to compile and link it. Section 2.2 then discusses how to generalize the program, and Section 2.3 shows how to add explanatory remarks, known as comments. Section 2.4 introduces variables, which store data that may change during the execution of a program, and Section 2.5 shows how to use the `scanf` function to read data into variables. Constants—data that won't change during program execution—can be given names, as Section 2.6 shows. Finally, Section 2.7 explains C's rules for creating names (identifiers) and Section 2.8 gives the rules for laying out a program.

## 2.1 Writing a Simple Program

In contrast to programs written in some languages, C programs require little “boilerplate”—a complete program can be as short as a few lines.

### PROGRAM Printing a Pun

The first program in Kernighan and Ritchie's classic *The C Programming Language* is extremely short; it does nothing but write the message `hello, world`. Unlike other C authors, I won't use this program as my first example. I will, however, uphold another C tradition: the bad pun. Here's the pun:

To C, or not to C: that is the question.

The following program, which we'll name `pun.c`, displays this message each time it is run.

```
pun.c #include <stdio.h>

int main(void)
{
    printf("To C, or not to C: that is the question.\n");
    return 0;
}
```

Section 2.2 explains the form of this program in some detail. For now, I'll just make a few brief observations. The line

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

is necessary to “include” information about C’s standard I/O (input/output) library. The program’s executable code goes inside `main`, which represents the “main” program. The only line inside `main` is a command to display the desired message. `printf` is a function from the standard I/O library that can produce nicely formatted output. The `\n` code tells `printf` to advance to the next line after printing the message. The line

```
return 0;
```

indicates that the program “returns” the value 0 to the operating system when it terminates.

## Compiling and Linking

Despite its brevity, getting `pun.c` to run is more involved than you might expect. First, we need to create a file named `pun.c` containing the program (any text editor will do). The name of the file doesn’t matter, but the `.c` extension is often required by compilers.

Next, we’ve got to convert the program to a form that the machine can execute. For a C program, that usually involves three steps:

- **Preprocessing.** The program is first given to a *preprocessor*, which obeys commands that begin with # (known as *directives*). A preprocessor is a bit like an editor; it can add things to the program and make modifications.
- **Compiling.** The modified program now goes to a *compiler*, which translates it into machine instructions (*object code*). The program isn’t quite ready to run yet, however.
- **Linking.** In the final step, a *linker* combines the object code produced by the compiler with any additional code needed to yield a complete executable program. This additional code includes library functions (like `printf`) that are used in the program.

Fortunately, this process is often automated, so you won't find it too onerous. In fact, the preprocessor is usually integrated with the compiler, so you probably won't even notice it at work.

The commands necessary to compile and link vary, depending on the compiler and operating system. Under UNIX, the C compiler is usually named `cc`. To compile and link the `pun.c` program, enter the following command in a terminal or command-line window:

```
% cc pun.c
```

(The `%` character is the UNIX prompt, not something that you need to enter.) Linking is automatic when using `cc`; no separate link command is necessary.

After compiling and linking the program, `cc` leaves the executable program in a file named `a.out` by default. `cc` has many options; one of them (the `-o` option) allows us to choose the name of the file containing the executable program. For example, if we want the executable version of `pun.c` to be named `pun`, we would enter the following command:

```
% cc -o pun pun.c
```

---

### The GCC Compiler

One of the most popular C compilers is the GCC compiler, which is supplied with Linux but is available for many other platforms as well. Using this compiler is similar to using the traditional UNIX `cc` compiler. For example, to compile the `pun.c` program, we would use the following command:

```
% gcc -o pun pun.c
```

**Q&A** The Q&A section at the end of the chapter provides more information about GCC.

---

## Integrated Development Environments

So far, we've assumed the use of a "command-line" compiler that's invoked by entering a command in a special window provided by the operating system. The alternative is to use an **integrated development environment (IDE)**, a software package that allows us to edit, compile, link, execute, and even debug a program without leaving the environment. The components of an IDE are designed to work together. For example, when the compiler detects an error in a program, it can arrange for the editor to highlight the line that contains the error. There's a great deal of variation among IDEs, so I won't discuss them further in this book. However, I would recommend checking to see which IDEs are available for your platform.

## 2.2 The General Form of a Simple Program

Let's take a closer look at `pun.c` and see how we can generalize it a bit. Simple C programs have the form

*directives*

```
int main(void)
{
    statements
}
```

In this template, and in similar templates elsewhere in this book, items printed in Courier would appear in a C program exactly as shown; items in *italics* represent text to be supplied by the programmer.

Notice how the braces show where `main` begins and ends. C uses `{` and `}` in much the same way that some other languages use words like `begin` and `end`. This illustrates a general point about C: it relies heavily on abbreviations and special symbols, one reason that C programs are concise (or—less charitably—cryptic).

### Q&A

Even the simplest C programs rely on three key language features: directives (editing commands that modify the program prior to compilation), functions (named blocks of executable code, of which `main` is an example), and statements (commands to be performed when the program is run). We'll take a closer look at these features now.

### Directives

Before a C program is compiled, it is first edited by a preprocessor. Commands intended for the preprocessor are called directives. Chapters 14 and 15 discuss directives in detail. For now, we're interested only in the `#include` directive.

The `pun.c` program begins with the line

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

This directive states that the information in `<stdio.h>` is to be “included” into the program before it is compiled. `<stdio.h>` contains information about C's standard I/O library. C has a number of **headers** like `<stdio.h>`; each contains information about some part of the standard library. The reason we're including `<stdio.h>` is that C, unlike some programming languages, has no built-in “read” and “write” commands. The ability to perform input and output is provided instead by functions in the standard library.

Directives always begin with a `#` character, which distinguishes them from other items in a C program. By default, directives are one line long; there's no semicolon or other special marker at the end of a directive.

## Functions

**Functions** are like “procedures” or “subroutines” in other programming languages—they’re the building blocks from which programs are constructed. In fact, a C program is little more than a collection of functions. Functions fall into two categories: those written by the programmer and those provided as part of the C implementation. I’ll refer to the latter as *library functions*, since they belong to a “library” of functions that are supplied with the compiler.

The term “function” comes from mathematics, where a function is a rule for computing a value when given one or more arguments:

$$\begin{aligned}f(x) &= x + 1 \\g(y, z) &= y^2 - z^2\end{aligned}$$

C uses the term “function” more loosely. In C, a function is simply a series of statements that have been grouped together and given a name. Some functions compute a value; some don’t. A function that computes a value uses the `return` statement to specify what value it “returns.” For example, a function that adds 1 to its argument might execute the statement

```
return x + 1;
```

while a function that computes the difference of the squares of its arguments might execute the statement

```
return y * y - z * z;
```

Although a C program may consist of many functions, only the `main` function is mandatory. `main` is special: it gets called automatically when the program is executed. Until Chapter 9, where we’ll learn how to write other functions, `main` will be the only function in our programs.




---

The name `main` is critical; it can’t be `begin` or `start` or even `MAIN`.

---

If `main` is a function, does it return a value? Yes: it returns a status code that is given to the operating system when the program terminates. Let’s take another look at the `pun.c` program:

```
#include <stdio.h>

int main(void)
{
    printf("To C, or not to C: that is the question.\n");
    return 0;
}
```

The word `int` just before `main` indicates that the `main` function returns an integer value. The word `void` in parentheses indicates that `main` has no arguments.

The statement

```
return 0;
```

return value of main ➤ 9.5

### Q&A

has two effects: it causes the `main` function to terminate (thus ending the program) and it indicates that the `main` function returns a value of 0. We'll have more to say about `main`'s return value in a later chapter. For now, we'll always have `main` return the value 0, which indicates normal program termination.

If there's no `return` statement at the end of the `main` function, the program will still terminate. However, many compilers will produce a warning message (because the function was supposed to return an integer but failed to).

## Statements

A **statement** is a command to be executed when the program runs. We'll explore statements later in the book, primarily in Chapters 5 and 6. The `pun.c` program uses only two kinds of statements. One is the `return` statement; the other is the **function call**. Asking a function to perform its assigned task is known as *calling* the function. The `pun.c` program, for example, calls the `printf` function to display a string on the screen:

```
printf("To C, or not to C: that is the question.\n");
```

compound statement ➤ 5.2

`C` requires that each statement end with a semicolon. (As with any good rule, there's one exception: the compound statement, which we'll encounter later.) The semicolon shows the compiler where the statement ends; since statements can continue over several lines, it's not always obvious where they end. Directives, on the other hand, are normally one line long, and they *don't* end with a semicolon.

## Printing Strings

`printf` is a powerful function that we'll examine in Chapter 3. So far, we've only used `printf` to display a **string literal**—a series of characters enclosed in double quotation marks. When `printf` displays a string literal, it doesn't show the quotation marks.

`printf` doesn't automatically advance to the next output line when it finishes printing. To instruct `printf` to advance one line, we must include `\n` (the **new-line character**) in the string to be printed. Writing a new-line character terminates the current output line; subsequent output goes onto the next line. To illustrate this point, consider the effect of replacing the statement

```
printf("To C, or not to C: that is the question.\n");
```

by two calls of `printf`:

```
printf("To C, or not to C: ");
printf("that is the question.\n");
```

The first call of `printf` writes To C, or not to C: . The second call writes that is the question. and advances to the next line. The net effect is the same as the original `printf`—the user can't tell the difference.

The new-line character can appear more than once in a string literal. To display the message

```
Brevity is the soul of wit.  
--Shakespeare
```

we could write

```
printf("Brevity is the soul of wit.\n --Shakespeare\n");
```

## 2.3 Comments

Our `pun.c` program still lacks something important: documentation. Every program should contain identifying information: the program name, the date written, the author, the purpose of the program, and so forth. In C, this information is placed in **comments**. The symbol `/*` marks the beginning of a comment and the symbol `*/` marks the end:

```
/* This is a comment */
```

Comments may appear almost anywhere in a program, either on separate lines or on the same lines as other program text. Here's what `pun.c` might look like with comments added at the beginning:

```
/* Name: pun.c */  
/* Purpose: Prints a bad pun. */  
/* Author: K. N. King */  
  
#include <stdio.h>  
  
int main(void)  
{  
    printf("To C, or not to C: that is the question.\n");  
    return 0;  
}
```

Comments may extend over more than one line; once it has seen the `/*` symbol, the compiler reads (and ignores) whatever follows until it encounters the `*/` symbol. If we like, we can combine a series of short comments into one long comment:

```
/* Name: pun.c  
Purpose: Prints a bad pun.  
Author: K. N. King */
```

A comment like this can be hard to read, though, because it's not easy to see where

the comment ends. Putting \*/ on a line by itself helps:

```
/* Name: pun.c
   Purpose: Prints a bad pun.
   Author: K. N. King
*/
```

Even better, we can form a “box” around the comment to make it stand out:

```
***** 
 * Name: pun.c
 * Purpose: Prints a bad pun.
 * Author: K. N. King
*****
```

Programmers often simplify boxed comments by omitting three of the sides:

```
/*
 * Name: pun.c
 * Purpose: Prints a bad pun.
 * Author: K. N. King
*/
```

A short comment can go on the same line with other program code:

```
int main(void) /* Beginning of main program */
```

A comment like this is sometimes called a “winged comment.”



Forgetting to terminate a comment may cause the compiler to ignore part of your program. Consider the following example:

```
printf("My ");
   /* forgot to close this comment...
printf("cat ");
printf("has ");
   /* so it ends here */
printf("fleas");
```

Because we’ve neglected to terminate the first comment, the compiler ignores the middle two statements, and the example prints My fleas.

**C99**

C99 provides a second kind of comment, which begins with // (two adjacent slashes):

```
// This is a comment
```

This style of comment ends automatically at the end of a line. To create a comment that’s more than one line long, we can either use the older comment style /\* ... \*/ or else put // at the beginning of each comment line:

```
// Name: pun.c
// Purpose: Prints a bad pun.
// Author: K. N. King
```

The newer comment style has a couple of important advantages. First, because a comment automatically ends at the end of a line, there's no chance that an unterminated comment will accidentally consume part of a program. Second, multiline comments stand out better, thanks to the `//` that's required at the beginning of each line.

## 2.4 Variables and Assignment

Few programs are as simple as the one in Section 2.1. Most programs need to perform a series of calculations before producing output, and thus need a way to store data temporarily during program execution. In C, as in most programming languages, these storage locations are called *variables*.

### Types

Every variable must have a *type*, which specifies what kind of data it will hold. C has a wide variety of types. For now, we'll limit ourselves to just two: `int` and `float`. Choosing the proper type is critical, since the type affects how the variable is stored and what operations can be performed on the variable. The type of a numeric variable determines the largest and smallest numbers that the variable can store; it also determines whether or not digits are allowed after the decimal point.

A variable of type `int` (short for *integer*) can store a whole number such as 0, 1, 392, or -2553. The range of possible values is limited, though. The largest `int` value is typically 2,147,483,647 but can be as small as 32,767.

#### Q&A

A variable of type `float` (short for *floating-point*) can store much larger numbers than an `int` variable. Furthermore, a `float` variable can store numbers with digits after the decimal point, like 379.125. `float` variables have drawbacks, however. Arithmetic on `float` numbers may be slower than arithmetic on `int` numbers. Most significantly, the value of a `float` variable is often just an approximation of the number that was stored in it. If we store 0.1 in a `float` variable, we may later find that the variable has a value such as 0.0999999999999987, thanks to rounding error.

### Declarations

Variables must be *declared*—described for the benefit of the compiler—before they can be used. To declare a variable, we first specify the *type* of the variable, then its *name*. (Variable names are chosen by the programmer, subject to the rules described in Section 2.7.) For example, we might declare variables `height` and `profit` as follows:

```
int height;
float profit;
```

The first declaration states that `height` is a variable of type `int`, meaning that `height` can store an integer value. The second declaration says that `profit` is a variable of type `float`.

If several variables have the same type, their declarations can be combined:

```
int height, length, width, volume;
float profit, loss;
```

Notice that each complete declaration ends with a semicolon.

Our first template for `main` didn't include declarations. When `main` contains declarations, these must precede statements:

```
int main(void)
{
    declarations
    statements
}
```

blocks ▶ 10.3 As we'll see in Chapter 9, this is true of functions in general, as well as blocks (statements that contain embedded declarations). As a matter of style, it's a good idea to leave a blank line between the declarations and the statements.

**C99**

In C99, declarations don't have to come before statements. For example, `main` might contain a declaration, then a statement, and then another declaration. For compatibility with older compilers, the programs in this book don't take advantage of this rule. However, it's common in C++ and Java programs not to declare variables until they're first needed, so this practice can be expected to become popular in C99 programs as well.

## Assignment

A variable can be given a value by means of **assignment**. For example, the statements

```
height = 8;
length = 12;
width = 10;
```

assign values to `height`, `length`, and `width`. The numbers 8, 12, and 10 are said to be **constants**.

Before a variable can be assigned a value—or used in any other way, for that matter—it must first be declared. Thus, we could write

```
int height;
height = 8;
```

but not

```
height = 8;      /*** WRONG ***/
int height;
```

A constant assigned to a `float` variable usually contains a decimal point. For example, if `profit` is a `float` variable, we might write

```
profit = 2150.48;
```

**Q&A**

It's best to append the letter `f` (for "float") to a constant that contains a decimal point if the number is assigned to a `float` variable:

```
profit = 2150.48f;
```

Failing to include the `f` may cause a warning from the compiler.

An `int` variable is normally assigned a value of type `int`, and a `float` variable is normally assigned a value of type `float`. Mixing types (such as assigning an `int` value to a `float` variable or assigning a `float` value to an `int` variable) is possible but not always safe, as we'll see in Section 4.2.

Once a variable has been assigned a value, it can be used to help compute the value of another variable:

```
height = 8;
length = 12;
width = 10;
volume = height * length * width; /* volume is now 960 */
```

In C, `*` represents the multiplication operator, so this statement multiplies the values stored in `height`, `length`, and `width`, then assigns the result to the variable `volume`. In general, the right side of an assignment can be a formula (or *expression*, in C terminology) involving constants, variables, and operators.

## Printing the Value of a Variable

We can use `printf` to display the current value of a variable. For example, to write the message

```
Height: h
```

where `h` is the current value of the `height` variable, we'd use the following call of `printf`:

```
printf("Height: %d\n", height);
```

`%d` is a placeholder indicating where the value of `height` is to be filled in during printing. Note the placement of `\n` just after `%d`, so that `printf` will advance to the next line after printing the value of `height`.

`%d` works only for `int` variables; to print a `float` variable, we'd use `%f` instead. By default, `%f` displays a number with six digits after the decimal point. To force `%f` to display `p` digits after the decimal point, we can put `.p` between `%` and `f`. For example, to print the line

```
Profit: $2150.48
```

we'd call `printf` as follows:

```
printf("Profit: %.2f\n", profit);
```

There's no limit to the number of variables that can be printed by a single call of `printf`. To display the values of both the `height` and `length` variables, we could use the following call of `printf`:

```
printf("Height: %d Length: %d\n", height, length);
```

## PROGRAM Computing the Dimensional Weight of a Box

Shipping companies don't especially like boxes that are large but very light, since they take up valuable space in a truck or airplane. In fact, companies often charge extra for such a box, basing the fee on its volume instead of its weight. In the United States, the usual method is to divide the volume by 166 (the allowable number of cubic inches per pound). If this number—the box's "dimensional" or "volumetric" weight—exceeds its actual weight, the shipping fee is based on the dimensional weight. (The 166 divisor is for international shipments; the dimensional weight of a domestic shipment is typically calculated using 194 instead.)

Let's say that you've been hired by a shipping company to write a program that computes the dimensional weight of a box. Since you're new to C, you decide to start off by writing a program that calculates the dimensional weight of a particular box that's 12" × 10" × 8". Division is represented by `/` in C, so the obvious way to compute the dimensional weight would be

```
weight = volume / 166;
```

where `weight` and `volume` are integer variables representing the box's weight and volume. Unfortunately, this formula isn't quite what we need. In C, when one integer is divided by another, the answer is "truncated": all digits after the decimal point are lost. The volume of a 12" × 10" × 8" box will be 960 cubic inches. Dividing by 166 gives the answer 5 instead of 5.783, so we have in effect rounded *down* to the next lowest pound; the shipping company expects us to round *up*. One solution is to add 165 to the volume before dividing by 166:

```
weight = (volume + 165) / 166;
```

A volume of 166 would give a weight of 331/166, or 1, while a volume of 167 would yield 332/166, or 2. Calculating the weight in this fashion gives us the following program.

```
dweight.c /* Computes the dimensional weight of a 12" x 10" x 8" box */
#include <stdio.h>
int main(void)
{
```

```

int height, length, width, volume, weight;

height = 8;
length = 12;
width = 10;
volume = height * length * width;
weight = (volume + 165) / 166;

printf("Dimensions: %dx%dx%d\n", length, width, height);
printf("Volume (cubic inches): %d\n", volume);
printf("Dimensional weight (pounds): %d\n", weight);

return 0;
}

```

The output of the program is

```

Dimensions: 12x10x8
Volume (cubic inches): 960
Dimensional weight (pounds): 6

```

## Initialization

variable initialization ➤ 18.5

Some variables are automatically set to zero when a program begins to execute, but most are not. A variable that doesn't have a default value and hasn't yet been assigned a value by the program is said to be *uninitialized*.



Attempting to access the value of an uninitialized variable (for example, by displaying the variable using `printf` or using it in an expression) may yield an unpredictable result such as 2568, -30891, or some equally strange number. With some compilers, worse behavior—even a program crash—may occur.

We can always give a variable an initial value by using assignment, of course. But there's an easier way: put the initial value of the variable in its declaration. For example, we can declare the `height` variable and initialize it in one step:

```
int height = 8;
```

In C jargon, the value 8 is said to be an *initializer*.

Any number of variables can be initialized in the same declaration:

```
int height = 8, length = 12, width = 10;
```

Notice that each variable requires its own initializer. In the following example, the initializer 10 is good only for the variable `width`, not for `height` or `length` (which remain uninitialized):

```
int height, length, width = 10;
```

## Printing Expressions

`printf` isn't limited to displaying numbers stored in variables; it can display the value of *any* numeric expression. Taking advantage of this property can simplify a program and reduce the number of variables. For instance, the statements

```
volume = height * length * width;
printf("%d\n", volume);
```

could be replaced by

```
printf("%d\n", height * length * width);
```

`printf`'s ability to print expressions illustrates one of C's general principles: *Wherever a value is needed, any expression of the same type will do.*

## 2.5 Reading Input

Because the `dweight.c` program calculates the dimensional weight of just one box, it isn't especially useful. To improve the program, we'll need to allow the user to enter the dimensions.

To obtain input, we'll use the `scanf` function, the C library's counterpart to `printf`. The `f` in `scanf`, like the `f` in `printf`, stands for "formatted"; both `scanf` and `printf` require the use of a *format string* to specify the appearance of the input or output data. `scanf` needs to know what form the input data will take, just as `printf` needs to know how to display output data.

To read an `int` value, we'd use `scanf` as follows:

```
scanf("%d", &i); /* reads an integer; stores into i */
```

The "`%d`" string tells `scanf` to read input that represents an integer; `i` is an `int` variable into which we want `scanf` to store the input. The `&` symbol is hard to explain at this point; for now, I'll just note that it is usually (but not always) required when using `scanf`.

Reading a `float` value requires a slightly different call of `scanf`:

```
scanf("%f", &x); /* reads a float value; stores into x */
```

`%f` works only with variables of type `float`, so I'm assuming that `x` is a `float` variable. The "`%f`" string tells `scanf` to look for an input value in `float` format (the number may contain a decimal point, but doesn't have to).

### PROGRAM Computing the Dimensional Weight of a Box (Revisited)

Here's an improved version of the dimensional weight program in which the user enters the dimensions. Note that each call of `scanf` is immediately preceded by a

call of `printf`. That way, the user will know when to enter input and what input to enter.

```
dweight2.c /* Computes the dimensional weight of a
   box from input provided by the user */

#include <stdio.h>

int main(void)
{
    int height, length, width, volume, weight;

    printf("Enter height of box: ");
    scanf("%d", &height);
    printf("Enter length of box: ");
    scanf("%d", &length);
    printf("Enter width of box: ");
    scanf("%d", &width);
    volume = height * length * width;
    weight = (volume + 165) / 166;

    printf("Volume (cubic inches): %d\n", volume);
    printf("Dimensional weight (pounds): %d\n", weight);

    return 0;
}
```

The output of the program has the following appearance (input entered by the user is underlined):

```
Enter height of box: 8
Enter length of box: 12
Enter width of box: 10
Volume (cubic inches): 960
Dimensional weight (pounds): 6
```

A message that asks the user to enter input (a *prompt*) normally shouldn't end with a new-line character, because we want the user to enter input on the same line as the prompt itself. When the user presses the Enter key, the cursor automatically moves to the next line—the program doesn't need to display a new-line character to terminate the current line.

The `dweight2.c` program suffers from one problem: it doesn't work correctly if the user enters nonnumeric input. Section 3.2 discusses this issue in more detail.

## 2.6 Defining Names for Constants

When a program contains constants, it's often a good idea to give them names. The `dweight.c` and `dweight2.c` programs rely on the constant 166, whose meaning may not be at all clear to someone reading the program later. Using a feature

known as **macro definition**, we can name this constant:

```
#define INCHES_PER_POUND 166
```

`#define` is a preprocessing directive, just as `#include` is, so there's no semicolon at the end of the line.

When a program is compiled, the preprocessor replaces each macro by the value that it represents. For example, the statement

```
weight = (volume + INCHES_PER_POUND - 1) / INCHES_PER_POUND;
```

will become

```
weight = (volume + 166 - 1) / 166;
```

giving the same effect as if we'd written the latter statement in the first place.

The value of a macro can be an expression:

```
#define RECIPROCAL_OF_PI (1.0f / 3.14159f)
```

(parentheses in macros ▶ 14.3)

If it contains operators, the expression should be enclosed in parentheses.

Notice that we've used only upper-case letters in macro names. This is a convention that most C programmers follow, not a requirement of the language. (Still, C programmers have been doing this for decades; you wouldn't want to be the first to deviate.)

## PROGRAM Converting from Fahrenheit to Celsius

The following program prompts the user to enter a Fahrenheit temperature; it then prints the equivalent Celsius temperature. The output of the program will have the following appearance (as usual, input entered by the user is underlined):

```
Enter Fahrenheit temperature: 212
Celsius equivalent: 100.0
```

The program will allow temperatures that aren't integers; that's why the Celsius temperature is displayed as 100.0 instead of 100. Let's look first at the entire program, then see how it's put together.

```
celsius.c /* Converts a Fahrenheit temperature to Celsius */
#include <stdio.h>

#define FREEZING_PT 32.0f
#define SCALE_FACTOR (5.0f / 9.0f)

int main(void)
{
    float fahrenheit, celsius;
    printf("Enter Fahrenheit temperature: ");
```

```

        scanf("%f", &fahrenheit);

        celsius = (fahrenheit - FREEZING_PT) * SCALE_FACTOR;

        printf("Celsius equivalent: %.1f\n", celsius);

        return 0;
    }

```

The statement

```
celsius = (fahrenheit - FREEZING_PT) * SCALE_FACTOR;
```

converts the Fahrenheit temperature to Celsius. Since FREEZING\_PT stands for 32.0f and SCALE\_FACTOR stands for (5.0f / 9.0f), the compiler sees this statement as

```
celsius = (fahrenheit - 32.0f) * (5.0f / 9.0f);
```

Defining SCALE\_FACTOR to be (5.0f / 9.0f) instead of (5 / 9) is important, because C truncates the result when two integers are divided. The value of (5 / 9) would be 0, which definitely isn't what we want.

The call of printf writes the Celsius temperature:

```
printf("Celsius equivalent: %.1f\n", celsius);
```

Notice the use of %.1f to display celsius with just one digit after the decimal point.

## 2.7 Identifiers

As we're writing a program, we'll have to choose names for variables, functions, macros, and other entities. These names are called *identifiers*. In C, an identifier may contain letters, digits, and underscores, but must begin with a letter or underscore. (In C99, identifiers may contain certain "universal character names" as well.)

**C99**

universal character names ► 25.4

Here are some examples of legal identifiers:

```
times10  get_next_char  _done
```

The following are *not* legal identifiers:

```
10times  get-next-char
```

The symbol 10times begins with a digit, not a letter or underscore. get-next-char contains minus signs, not underscores.

C is *case-sensitive*: it distinguishes between upper-case and lower-case letters in identifiers. For example, the following identifiers are all different:

```
job  joB  jOb  jOB  Job  JoB  JOb  JOB
```