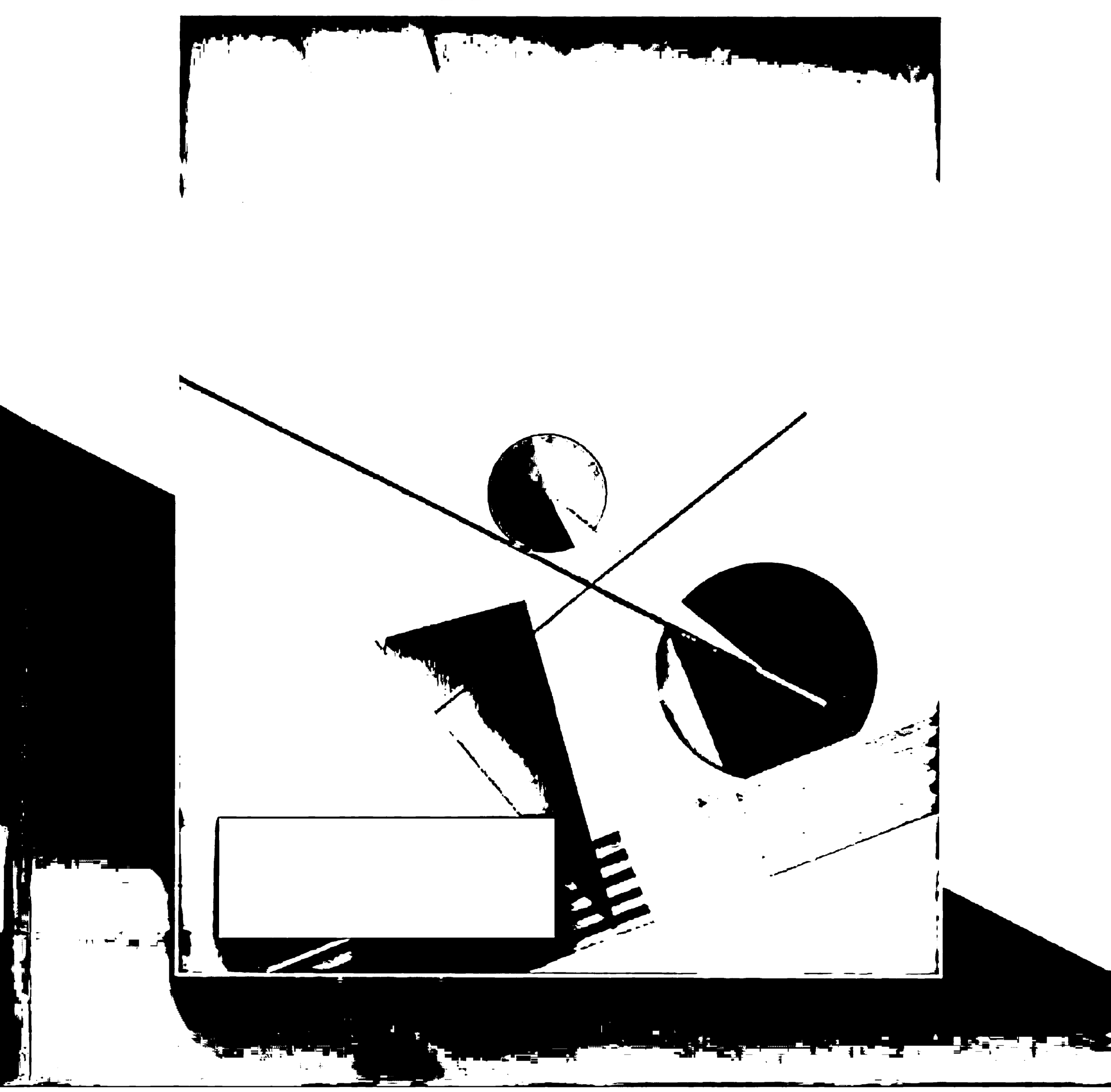
Covers both C89 and C99

K. N. KING

C PROGRAMMING

A Modern Approach SECOND EDITION



K. N. KING

C PROGRAMMING

A Modern Approach

SECOND EDITION

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 (*kking.com*)

Password-protected instructor site (also at *kking.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

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.

- ***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*. 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*) containing solutions to the remaining exercises and projects, plus PowerPoint presentations for most chapters. Faculty may contact me at *cbook@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 reexamined 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).

- **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 ➤ *Preface*

- **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.

idiom

- **Idioms**—code patterns frequently seen in C programs—are marked for quick reference.

portability tip

- **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*. 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*. 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 Javsicas. 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

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

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

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	801
	Bibliography	803
	Index	807

CONTENTS

Preface xxi

1 INTRODUCING C 1
1.1 History of C 1
Origins 1
Standardization 2
C-Based Languages 3
1.2 Strengths and Weaknesses of C 4
Strengths 4
Weaknesses 5
Effective Use of C 6

2 C FUNDAMENTALS 9
2.1 Writing a Simple Program 9
Program: Printing a Pun 9
Compiling and Linking 10
Integrated Development Environments 11
2.2 The General Form of a Simple Program 12
Directives 12
Functions 13
Statements 14
Printing Strings 14
2.3 Comments 15
2.4 Variables and Assignment 17
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
2.5 Reading Input **22**
Program: Computing the Dimensional Weight of a Box (Revisited) 22
2.6 Defining Names for Constants **23**
Program: Converting from Fahrenheit to Celsius 24
2.7 Identifiers **25**
Keywords 26
2.8 Layout of a C Program **27**

3 FORMATTED INPUT/OUTPUT **37**
3.1 The `printf` Function **37**
Conversion Specifications 38
Program: Using `printf` to Format Numbers 40
Escape Sequences 41
3.2 The `scanf` Function **42**
How `scanf` Works 43
Ordinary Characters in Format Strings 45
Confusing `printf` with `scanf` 45
Program: Adding Fractions 46

4 EXPRESSIONS **53**
4.1 Arithmetic Operators **54**
Operator Precedence and Associativity 55
Program: Computing a UPC Check Digit 56
4.2 Assignment Operators **58**
Simple Assignment 58
Lvalues 59
Compound Assignment 60
4.3 Increment and Decrement Operators **61**
4.4 Expression Evaluation **62**
Order of Subexpression Evaluation 64
4.5 Expression Statements **65**

5 SELECTION STATEMENTS **73**
5.1 Logical Expressions **74**
Relational Operators 74
Equality Operators 75
Logical Operators 75
5.2 The `if` Statement **76**
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

5.3 The `switch` Statement 86
The Role of the `break` Statement 88
Program: Printing a Date in Legal Form 89

6 LOOPS 99

6.1 The `while` Statement 99
Infinite Loops 101
Program: Printing a Table of Squares 102
Program: Summing a Series of Numbers 102

6.2 The `do` Statement 103
Program: Calculating the Number of Digits in an Integer 104

6.3 The `for` Statement 105
`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

6.4 Exiting from a Loop 111
The `break` Statement 111
The `continue` Statement 112
The `goto` Statement 113
Program: Balancing a Checkbook 114

6.5 The Null Statement 116

7 BASIC TYPES 125

7.1 Integer Types 125
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

7.2 Floating Types 132
Floating Constants 133
Reading and Writing Floating-Point Numbers 134

7.3 Character Types 134
Operations on Characters 135
Signed and Unsigned Characters 136

Arithmetic Types 136
Escape Sequences 137
Character-Handling Functions 138
Reading and Writing Characters using `scanf` and `printf` 139
Reading and Writing Characters using `getchar` and `putchar` 140
Program: Determining the Length of a Message 141

7.4 Type Conversion 142
The Usual Arithmetic Conversions 143
Conversion During Assignment 145
Implicit Conversions in C99 146
Casting 147

7.5 Type Definitions 149
Advantages of Type Definitions 149
Type Definitions and Portability 150

7.6 The `sizeof` Operator 151

8 ARRAYS 161

8.1 One-Dimensional Arrays 161
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 `sizeof` Operator with Arrays 167
Program: Computing Interest 168

8.2 Multidimensional Arrays 169
Initializing a Multidimensional Array 171
Constant Arrays 172
Program: Dealing a Hand of Cards 172

8.3 Variable-Length Arrays (C99) 174

9 FUNCTIONS 183

9.1 Defining and Calling Functions 183
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

9.2 Function Declarations 191

9.3 Arguments 193
Argument Conversions 194
Array Arguments 195
Variable-Length Array Parameters 198

Using `static` in Array Parameter Declarations 200
Compound Literals 200
9.4 The `return` Statement 201
9.5 Program Termination 202
The `exit` Function 203
9.6 Recursion 204
The Quicksort Algorithm 205
Program: Quicksort 207

10 PROGRAM ORGANIZATION 219
10.1 Local Variables 219
Static Local Variables 220
Parameters 221
10.2 External Variables 221
Example: Using External Variables to Implement a Stack 221
Pros and Cons of External Variables 222
Program: Guessing a Number 224
10.3 Blocks 227
10.4 Scope 228
10.5 Organizing a C Program 229
Program: Classifying a Poker Hand 230

11 POINTERS 241
11.1 Pointer Variables 241
Declaring Pointer Variables 242
11.2 The Address and Indirection Operators 243
The Address Operator 243
The Indirection Operator 244
11.3 Pointer Assignment 245
11.4 Pointers as Arguments 247
Program: Finding the Largest and Smallest Elements in an Array 249
Using `const` to Protect Arguments 250
11.5 Pointers as Return Values 251

12 POINTERS AND ARRAYS 257
12.1 Pointer Arithmetic 257
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
12.2 Using Pointers for Array Processing 260
Combining the * and ++ Operators 262

12.3 Using an Array Name as a Pointer **263**
Program: Reversing a Series of Numbers (Revisited) 264
Array Arguments (Revisited) 265
Using a Pointer as an Array Name 266
12.4 Pointers and Multidimensional Arrays **267**
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
12.5 Pointers and Variable-Length Arrays (C99) **270**

13 STRINGS **277**
13.1 String Literals **277**
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
13.2 String Variables **281**
Initializing a String Variable 281
Character Arrays versus Character Pointers 283
13.3 Reading and Writing Strings **284**
Writing Strings Using `printf` and `puts` 284
Reading Strings Using `scanf` and `gets` 285
Reading Strings Character by Character 286
13.4 Accessing the Characters in a String **287**
13.5 Using the C String Library **289**
The `strcpy` (String Copy) Function 290
The `strlen` (String Length) Function 291
The `strcat` (String Concatenation) Function 291
The `strcmp` (String Comparison) Function 292
Program: Printing a One-Month Reminder List 293
13.6 String Idioms **296**
Searching for the End of a String 296
Copying a String 298
13.7 Arrays of Strings **300**
Command-Line Arguments 302
Program: Checking Planet Names 303

14 THE PREPROCESSOR **315**
14.1 How the Preprocessor Works **315**
14.2 Preprocessing Directives **318**
14.3 Macro Definitions **319**
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 `__func__` Identifier 333

14.4 Conditional Compilation 333
The `#if` and `#endif` Directives 334
The `defined` Operator 335
The `#ifdef` and `#ifndef` Directives 335
The `#elif` and `#else` Directives 336
Uses of Conditional Compilation 337

14.5 Miscellaneous Directives 338
The `#error` Directive 338
The `#line` Directive 339
The `#pragma` Directive 340
The `_Pragma` Operator 341

15 WRITING LARGE PROGRAMS 349

15.1 Source Files 349

15.2 Header Files 350
The `#include` Directive 351
Sharing Macro Definitions and Type Definitions 353
Sharing Function Prototypes 354
Sharing Variable Declarations 355
Nested Includes 357
Protecting Header Files 357
`#error` Directives in Header Files 358

15.3 Dividing a Program into Files 359
Program: Text Formatting 359

15.4 Building a Multiple-File Program 366
Makefiles 366
Errors During Linking 368
Rebuilding a Program 369
Defining Macros Outside a Program 371

16 STRUCTURES, UNIONS, AND ENUMERATIONS 377

16.1 Structure Variables 377
Declaring Structure Variables 378
Initializing Structure Variables 379
Designated Initializers 380
Operations on Structures 381

16.2 Structure Types **382**
Declaring a Structure Tag 383
Defining a Structure Type 384
Structures as Arguments and Return Values 384
Compound Literals 386
16.3 Nested Arrays and Structures **386**
Nested Structures 387
Arrays of Structures 387
Initializing an Array of Structures 388
Program: Maintaining a Parts Database 389
16.4 Unions **396**
Using Unions to Save Space 398
Using Unions to Build Mixed Data Structures 399
Adding a "Tag Field" to a Union 400
16.5 Enumerations **401**
Enumeration Tags and Type Names 402
Enumerations as Integers 403
Using Enumerations to Declare "Tag Fields" 404

17 ADVANCED USES OF POINTERS **413**
17.1 Dynamic Storage Allocation **414**
Memory Allocation Functions 414
Null Pointers 414
17.2 Dynamically Allocated Strings **416**
Using `malloc` 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
17.3 Dynamically Allocated Arrays **420**
Using `malloc` to Allocate Storage for an Array 420
The `calloc` Function 421
The `realloc` Function 421
17.4 Deallocating Storage **422**
The `free` Function 423
The "Dangling Pointer" Problem 424
17.5 Linked Lists **424**
Declaring a Node Type 425
Creating a Node 425
The `->` 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
17.6 Pointers to Pointers **438**

17.7 Pointers to Functions 439
Function Pointers as Arguments 439
The `qsort` Function 440
Other Uses of Function Pointers 442
Program: Tabulating the Trigonometric Functions 443
17.8 Restricted Pointers (C99) 445
17.9 Flexible Array Members (C99) 447

18 DECLARATIONS 457
18.1 Declaration Syntax 457
18.2 Storage Classes 459
Properties of Variables 459
The `auto` Storage Class 460
The `static` Storage Class 461
The `extern` Storage Class 462
The `register` Storage Class 463
The Storage Class of a Function 464
Summary 465
18.3 Type Qualifiers 466
18.4 Declarators 467
Deciphering Complex Declarations 468
Using Type Definitions to Simplify Declarations 470
18.5 Initializers 470
Uninitialized Variables 472
18.6 Inline Functions (C99) 472
Inline Definitions 473
Restrictions on Inline Functions 474
Using Inline Functions with GCC 475

19 PROGRAM DESIGN 483
19.1 Modules 484
Cohesion and Coupling 486
Types of Modules 486
19.2 Information Hiding 487
A Stack Module 487
19.3 Abstract Data Types 491
Encapsulation 492
Incomplete Types 492
19.4 A Stack Abstract Data Type 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 Design Issues for Abstract Data Types 502
Naming Conventions 502
Error Handling 502
Generic ADTs 503
ADTs in Newer Languages 503

20 LOW-LEVEL PROGRAMMING 509
20.1 Bitwise Operators 509
Bitwise Shift Operators 510
Bitwise Complement, *And*, Exclusive *Or*, and Inclusive *Or* 511
Using the Bitwise Operators to Access Bits 512
Using the Bitwise Operators to Access Bit-Fields 513
Program: XOR Encryption 514
20.2 Bit-Fields in Structures 516
How Bit-Fields Are Stored 517
20.3 Other Low-Level Techniques 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 `volatile` Type Qualifier 523

21 THE STANDARD LIBRARY 529
21.1 Using the Library 529
Restrictions on Names Used in the Library 530
Functions Hidden by Macros 531
21.2 C89 Library Overview 531
21.3 C99 Library Changes 534
21.4 The `<stddef.h>` Header: Common Definitions 535
21.5 The `<stdbool.h>` Header (C99): Boolean Type and Values 536

22 INPUT/OUTPUT 539
22.1 Streams 540
File Pointers 540
Standard Streams and Redirection 540
Text Files versus Binary Files 541
22.2 File Operations 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
22.3 Formatted I/O 551
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
22.4 Character I/O 566
Output Functions 566
Input Functions 567
Program: Copying a File 568
22.5 Line I/O 569
Output Functions 569
Input Functions 570
22.6 Block I/O 571
22.7 File Positioning 572
Program: Modifying a File of Part Records 574
22.8 String I/O 575
Output Functions 576
Input Functions 576

23 LIBRARY SUPPORT FOR NUMBERS AND CHARACTER DATA 589
23.1 The `<float.h>` Header: Characteristics of Floating Types 589
23.2 The `<limits.h>` Header: Sizes of Integer Types 591
23.3 The `<math.h>` Header (C89): Mathematics 593
Errors 593
Trigonometric Functions 594
Hyperbolic Functions 595
Exponential and Logarithmic Functions 595
Power Functions 596
Nearest Integer, Absolute Value, and Remainder Functions 596
23.4 The `<math.h>` Header (C99): Mathematics 597
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

23.5 The `<ctype.h>` Header: Character Handling 612
Character-Classification Functions 612
Program: Testing the Character-Classification Functions 613
Character Case-Mapping Functions 614
Program: Testing the Case-Mapping Functions 614

23.6 The `<string.h>` Header: String Handling 615
Copying Functions 616
Concatenation Functions 617
Comparison Functions 617
Search Functions 619
Miscellaneous Functions 622

24 ERROR HANDLING 627

24.1 The `<assert.h>` Header: Diagnostics 628

24.2 The `<errno.h>` Header: Errors 629
The `perror` and `strerror` Functions 630

24.3 The `<signal.h>` Header: Signal Handling 631
Signal Macros 631
The `signal` Function 632
Predefined Signal Handlers 633
The `raise` Function 634
Program: Testing Signals 634

24.4 The `<setjmp.h>` Header: Nonlocal Jumps 635
Program: Testing `setjmp`/`longjmp` 636

25 INTERNATIONAL FEATURES 641

25.1 The `<locale.h>` Header: Localization 642
Categories 642
The `setlocale` Function 643
The `localeconv` Function 644

25.2 Multibyte Characters and Wide Characters 647

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
25.3 Digraphs and Trigraphs 654
Trigraphs 654
Digraphs 655
The `<iso646.h>` Header: Alternative Spellings 656
25.4 Universal Character Names (C99) 656
25.5 The `<wchar.h>` Header (C99): Extended Multibyte and Wide-Character Utilities 657
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
25.6 The `<wctype.h>` Header (C99): Wide-Character Classification and Mapping Utilities 671
Wide-Character Classification Functions 671
Extensible Wide-Character Classification Functions 672
Wide-Character Case-Mapping Functions 673
Extensible Wide-Character Case-Mapping Functions 673

26 MISCELLANEOUS LIBRARY FUNCTIONS 677
26.1 The `<stdarg.h>` Header: Variable Arguments 677
Calling a Function with a Variable Argument List 679
The `v...printf` Functions 680
The `v...scanf` Functions 681
26.2 The `<stdlib.h>` Header: General Utilities 682
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
26.3 The `<time.h>` Header: Date and Time 692
Time Manipulation Functions 693
Time Conversion Functions 695
Program: Displaying the Date and Time 698

27 ADDITIONAL C99 SUPPORT FOR MATHEMATICS 705

27.1 The <stdint.h> Header (C99): Integer Types 705
<stdint.h> Types 706
Limits of Specified-Width Integer Types 707
Limits of Other Integer Types 708
Macros for Integer Constants 708

27.2 The <inttypes.h> Header (C99): Format Conversion of Integer Types 709
Macros for Format Specifiers 710
Functions for Greatest-Width Integer Types 711

27.3 Complex Numbers (C99) 712
Definition of Complex Numbers 713
Complex Arithmetic 714
Complex Types in C99 714
Operations on Complex Numbers 715
Conversion Rules for Complex Types 715

27.4 The <complex.h> Header (C99): Complex Arithmetic 717
<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

27.5 The <tgmath.h> Header (C99): Type-Generic Math 723
Type-Generic Macros 724
Invoking a Type-Generic Macro 725

27.6 The <fenv.h> Header (C99): Floating-Point Environment 726
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

Appendix A C Operators 735

Appendix B C99 versus C89 737

Appendix C C89 versus K&R C 743

Appendix D Standard Library Functions 747

Appendix E ASCII Character Set 801

Bibliography 803

Index 807

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.

C99

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.

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 Osovlanski 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*, 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 ⚠ 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.

Q&A

- ***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

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*.

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.

GCC ➤2.1

***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.

headers ➤ 15.2

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:

$$f(x) = x + 1$$
$$g(y, z) = y^2 - z^2$$

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;
```

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.

return value of `main` ➤ 9.5

Q&A

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).

Q&A

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");
```

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.

compound statement ➤ 5.2

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.

range of int values ➤7.1

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`.

& operator ➤ 11.2

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
```

These eight identifiers could all be used simultaneously, each for a completely different purpose. (Talk about obfuscation!) Sensible programmers try to make identifiers look different unless they're somehow related.

Since case matters in C, many programmers follow the convention of using only lower-case letters in identifiers (other than macros), with underscores inserted when necessary for legibility:

```
symbol_table  current_page  name_and_address
```

Other programmers avoid underscores, instead using an upper-case letter to begin each word within an identifier:

```
symbolTable  currentPage  nameAndAddress
```

(The first letter is sometimes capitalized as well.) Although the former style is common in traditional C, the latter style is becoming more popular thanks to its widespread use in Java and C# (and, to a lesser extent, C++). Other reasonable conventions exist; just be sure to capitalize an identifier the same way each time it appears in a program.

Q&A C places no limit on the maximum length of an identifier, so don't be afraid to use long, descriptive names. A name such as `current_page` is a lot easier to understand than a name like `cp`.

Keywords

The *keywords* in Table 2.1 have special significance to C compilers and therefore can't be used as identifiers. Note that five keywords were added in C99.

C99

Table 2.1 Keywords

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

†C99 only

Because of C's case-sensitivity, keywords must appear in programs exactly as shown in Table 2.1, with all letters in lower case. Names of functions in the standard library (such as `printf`) contain only lower-case letters also. Avoid the plight of the unfortunate programmer who enters an entire program in upper case, only to find that the compiler can't recognize keywords and calls of library functions.

Watch out for other restrictions on identifiers. Some compilers treat certain identifiers (`asm`, for example) as additional keywords. Identifiers that belong to the standard library are restricted as well. Accidentally using one of these names can cause an error during compilation or linking. Identifiers that begin with an underscore are also restricted.

restrictions on identifiers ➤21.1

2.8 Layout of a C Program

We can think of a C program as a series of *tokens:* groups of characters that can't be split up without changing their meaning. Identifiers and keywords are tokens. So are operators like + and -, punctuation marks such as the comma and semicolon, and string literals. For example, the statement

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

consists of seven tokens:

```
printf   (   "Height: %d\n"   ,   height   )   ;
  ①      ②          ③        ④      ⑤      ⑥   ⑦
```

Tokens ① and ⑤ are identifiers, token ③ is a string literal, and tokens ②, ④, ⑥, and ⑦ are punctuation.

The amount of space between tokens in a program isn't critical in most cases. At one extreme, tokens can be crammed together with no space between them at all, except where this would cause two tokens to merge into a third token. For example, we could delete most of the space in the `celsius.c` program of Section 2.6, provided that we leave space between tokens such as `int` and `main` and between `float` and `fahrenheit`:

```
/* 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;}
```

In fact, if the page were wider, we could put the entire `main` function on a single line. We can't put the whole *program* on one line, though, because each preprocessing directive requires a separate line.

Compressing programs in this fashion isn't a good idea. In fact, adding spaces and blank lines to a program can make it easier to read and understand. Fortunately,

C allows us to insert any amount of space—blanks, tabs, and new-line characters—between tokens. This rule has several important consequences for program layout:

- *Statements can be divided* over any number of lines. The following statement, for example, is so long that it would be hard to squeeze it onto a single line:

```
printf("Dimensional weight (pounds): %d\n",
  (volume + INCHES_PER_POUND - 1) / INCHES_PER_POUND);
```

- *Space between tokens* makes it easier for the eye to separate them. For this reason, I usually put a space before and after each operator:

```
volume = height * length * width;
```

  I also put a space after each comma. Some programmers go even further, putting spaces around parentheses and other punctuation.

- *Indentation* can make nesting easier to spot. For example, we should indent declarations and statements to make it clear that they're nested inside `main`.

Q&A

- *Blank lines* can divide a program into logical units, making it easier for the reader to discern the program's structure. A program with no blank lines is as hard to read as a book with no chapters.

The `celsius.c` program of Section 2.6 illustrates several of these guidelines. Let's take a closer look at the `main` function in that program:

```
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;
}
```

First, observe how the space around =, -, and * makes these operators stand out. Second, notice how the indentation of declarations and statements makes it obvious that they all belong to `main`. Finally, note how blank lines divide `main` into five parts: (1) declaring the `fahrenheit` and `celsius` variables; (2) obtaining the Fahrenheit temperature; (3) calculating the value of `celsius`; (4) printing the Celsius temperature; and (5) returning to the operating system.

While we're on the subject of program layout, notice how I've placed the { token underneath `main()` and put the matching } on a separate line, aligned with {. Putting } on a separate line lets us insert or delete statements at the end of the function; aligning it with { makes it easy to spot the end of `main`.

A final note: Although extra spaces can be added *between* tokens, it's not pos-

sible to add space *within* a token without changing the meaning of the program or causing an error. Writing

```
fl oat fahrenheit, celsius;    /*** WRONG ***/
```

or

```
fl
oat fahrenheit, celsius;       /*** WRONG ***/
```

produces an error when the program is compiled. Putting a space inside a string literal is allowed, although it changes the meaning of the string. However, putting a new-line character in a string (in other words, splitting the string over two lines) is illegal:

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

Continuing a string from one line to the next requires a special technique that we'll learn in a later chapter.

continuing a string ➤ 13.1

Q & A

Q: What does GCC stand for? [p. 11]

A: GCC originally stood for "GNU C compiler." It now stands for "GNU Compiler Collection," because the current version of GCC compiles programs written in a variety of languages, including Ada, C, C++, Fortran, Java, and Objective-C.

Q: OK, so what does GNU stand for?

A: GNU stands for "GNU's Not UNIX!" (and is pronounced *guh-NEW*, by the way). GNU is a project of the Free Software Foundation, an organization set up by Richard M. Stallman as a protest against the restrictions of licensed UNIX software. According to its web site, the Free Software Foundation believes that users should be free to "run, copy, distribute, study, change and improve" software. The GNU Project has rewritten much traditional UNIX software from scratch and made it publicly available at no charge.

GCC and other GNU software are crucial to Linux. Linux itself is only the "kernel" of an operating system (the part that handles program scheduling and basic I/O services); the GNU software is necessary to have a fully functional operating system.

For more information on the GNU Project, visit *www.gnu.org*.

Q: What's the big deal about GCC, anyway?

A: GCC is significant for many reasons, not least the fact that it's free and capable of compiling a number of languages. It runs under many operating systems and generates code for many different CPUs, including all the widely used ones. GCC is

the primary compiler for many UNIX-based operating systems, including Linux, BSD, and Mac OS X, and it's used extensively for commercial software development. For more information about GCC, visit *gcc.gnu.org*.

Q: How good is GCC at finding errors in programs?

A: GCC has various command-line options that control how thoroughly it checks programs. When these options are used, GCC is quite good at finding potential trouble spots in a program. Here are some of the more popular options:

`-Wall`	Causes the compiler to produce warning messages when it detects possible errors. (`-W` can be followed by codes for specific warnings; `-Wall` means "all `-W` options.") Should be used in conjunction with `-O` for maximum effect.
`-W`	Issues additional warning messages beyond those produced by `-Wall`.
`-pedantic`	Issues all warnings required by the C standard. Causes programs that use nonstandard features to be rejected.
`-ansi`	Turns off features of GCC that aren't standard C and enables a few standard features that are normally disabled.
`-std=c89` `-std=c99`	Specifies which version of C the compiler should use to check the program.

These options are often used in combination:

```
% gcc -O -Wall -W -pedantic -ansi -std=c99 -o pun pun.c
```

Q: Why is C so terse? It seems as though programs would be more readable if C used `begin` and `end` instead of `{` and `}`, `integer` instead of `int`, and so forth. [p. 12]

A: Legend has it that the brevity of C programs is due to the environment that existed in Bell Labs at the time the language was developed. The first C compiler ran on a DEC PDP-11 (an early minicomputer); programmers used a teletype—essentially a typewriter connected to a computer—to enter programs and print listings. Because teletypes were very slow (they could print only 10 characters per second), minimizing the number of characters in a program was clearly advantageous.

Q: In some C books, the `main` function ends with `exit(0)` instead of `return 0`. Are these the same? [p. 14]

A: When they appear inside `main`, these statements are indeed equivalent: both terminate the program, returning the value 0 to the operating system. Which one to use is mostly a matter of taste.

Q: What happens if a program reaches the end of the `main` function without executing a `return` statement? [p. 14]

A: The `return` statement isn't mandatory; if it's missing, the program will still ter-

minate. In C89, the value returned to the operating system is undefined. In C99, if
C99 `main` is declared to return an `int` (as in our examples), the program returns 0 to the operating system; otherwise, the program returns an unspecified value.

Q: Does the compiler remove a comment entirely or replace it with blank space?

A: Some old C compilers deleted all the characters in each comment, making it possible to write

```
a/**/b = 0;
```

and have the compiler interpret it as

```
ab = 0;
```

According to the C standard, however, the compiler must replace each comment by a single space character, so this trick doesn't work. Instead, we'd end up with the following (illegal) statement:

```
a b = 0;
```

Q: How can I tell if my program has an unterminated comment?

A: If you're lucky, the program won't compile because the comment has rendered the program illegal. If the program does compile, there are several techniques that you can use. Stepping through the program line by line with a debugger will reveal if any lines are being skipped. Some IDEs display comments in a distinctive color to distinguish them from surrounding code. If you're using such an environment, you can easily spot unterminated comments, since program text will have a different
`lint` ➤1.2 color if it's accidentally included in a comment. A program such as `lint` can also help.

Q: Is it legal to nest one comment inside another?

A: Old-style comments (`/* ... */`) can't be nested. For instance, the following code is illegal:

```
/*
   /*** WRONG ***/
*/
```

The `*/` symbol on the second line matches the `/*` symbol on the first line, so the compiler will flag the `*/` symbol on the third line as an error.

C's prohibition against nested comments can sometimes be a problem. Suppose we've written a long program containing many short comments. To disable a portion of the program temporarily (during testing, say), our first impulse is to "comment out" the offending lines with `/*` and `*/`. Unfortunately, this method
C99 won't work if the lines contain old-style comments. C99 comments (those beginning with `//`) can be nested inside old-style comments, however—another advantage to using this kind of comment.

In any event, there's a better way to disable portions of a program, as we'll see later.

disabling code ➤ 14.4

Q: Where does the `float` type get its name? [p. 17]

A: `float` is short for "floating-point," a technique for storing numbers in which the decimal point "floats." A `float` value is usually stored in two parts: the fraction (or mantissa) and the exponent. The number 12.0 might be stored as 1.5×2^3, for example, where 1.5 is the fraction and 3 is the exponent. Some programming languages call this type `real` instead of `float`.

Q: Why do floating-point constants need to end with the letter `f`? [p. 19]

A: For the full explanation, see Chapter 7. Here's the short answer: a constant that contains a decimal point but doesn't end with `f` has type `double` (short for "double precision"). `double` values are stored more accurately than `float` values. Moreover, `double` values can be larger than `float` values, which is why we need to add the letter `f` when assigning to a `float` variable. Without the `f`, a warning may be generated about the possibility of a number being stored into a `float` variable that exceeds the capacity of the variable.

***Q: Is it really true that there's no limit on the length of an identifier? [p. 26]**

A: Yes and no. The C89 standard says that identifiers may be arbitrarily long. However, compilers are only required to remember the first 31 characters (63 characters in C99). Thus, if two names begin with the same 31 characters, a compiler might be unable to distinguish between them.

C99

To make matters even more complicated, there are special rules for identifiers with external linkage; most function names fall into this category. Since these names must be made available to the linker, and since some older linkers can handle only short names, only the first six characters are significant in C89. Moreover, the case of letters may not matter. As a result, `ABCDEFG` and `abcdefh` might be treated as the same name. (In C99, the first 31 characters are significant, and the case of letters is taken into account.)

external linkage ➤ 18.2

C99

Most compilers and linkers are more generous than the standard, so these rules aren't a problem in practice. Don't worry about making identifiers too long—worry about making them too short.

Q: How many spaces should I use for indentation? [p. 28]

A: That's a tough question. Leave too little space, and the eye has trouble detecting indentation. Leave too much, and lines run off the screen (or page). Many C programmers indent nested statements eight spaces (one tab stop), which is probably too much. Studies have shown that the optimum amount of indentation is three spaces, but many programmers feel uncomfortable with numbers that aren't a power of two. Although I normally prefer to indent three or four spaces, I'll use two spaces in this book so that my programs will fit within the margins.

Exercises

Section 2.1

1. Create and run Kernighan and Ritchie's famous "hello, world" program:

```
#include <stdio.h>

int main(void)
{
  printf("hello, world\n");
}
```

Do you get a warning message from the compiler? If so, what's needed to make it go away?

Section 2.2

Ⓦ 2. Consider the following program:

```
#include <stdio.h>

int main(void)
{
  printf("Parkinson's Law:\nWork expands so as to ");
  printf("fill the time\n");
  printf("available for its completion.\n");
  return 0;
}
```

(a) Identify the directives and statements in this program.

(b) What output does the program produce?

Section 2.4

Ⓦ 3. Condense the `dweight.c` program by (1) replacing the assignments to `height`, `length`, and `width` with initializers and (2) removing the `weight` variable, instead calculating `(volume + 165) / 166` within the last `printf`.

Ⓦ 4. Write a program that declares several `int` and `float` variables—without initializing them—and then prints their values. Is there any pattern to the values? (Usually there isn't.)

Section 2.7

Ⓦ 5. Which of the following are not legal C identifiers?

(a) `100_bottles`
(b) `_100_bottles`
(c) `one__hundred__bottles`
(d) `bottles_by_the_hundred_`

6. Why is it not a good idea for an identifier to contain more than one adjacent underscore (as in `current___balance`, for example)?

7. Which of the following are keywords in C?

(a) `for`
(b) `If`
(c) `main`
(d) `printf`
(e) `while`

Ⓦ Answer available on the Web at *knking.com/books/c2*.

Section 2.8

W 8. How many tokens are there in the following statement?

```
answer=(3*q-p*p)/3;
```

9. Insert spaces between the tokens in Exercise 8 to make the statement easier to read.

10. In the `dweight.c` program (Section 2.4), which spaces are essential?

Programming Projects

1. Write a program that uses `printf` to display the following picture on the screen:

```
    *
   *
  *
*   *
 * *
  *
```

2. Write a program that computes the volume of a sphere with a 10-meter radius, using the formula $v = 4/3\pi r^3$. Write the fraction 4/3 as `4.0f/3.0f`. (Try writing it as `4/3`. What happens?) *Hint:* C doesn't have an exponentiation operator, so you'll need to multiply r by itself twice to compute r^3.

3. Modify the program of Programming Project 2 so that it prompts the user to enter the radius of the sphere.

W 4. Write a program that asks the user to enter a dollars-and-cents amount, then displays the amount with 5% tax added:

```
Enter an amount: 100.00
With tax added: $105.00
```

5. Write a program that asks the user to enter a value for x and then displays the value of the following polynomial:

$3x^5 + 2x^4 - 5x^3 - x^2 + 7x - 6$

Hint: C doesn't have an exponentiation operator, so you'll need to multiply x by itself repeatedly in order to compute the powers of x. (For example, `x * x * x` is `x` cubed.)

6. Modify the program of Programming Project 5 so that the polynomial is evaluated using the following formula:

$(((3x + 2)x - 5)x - 1)x + 7)x - 6$

Note that the modified program performs fewer multiplications. This technique for evaluating polynomials is known as ***Horner's Rule***.

7. Write a program that asks the user to enter a U.S. dollar amount and then shows how to pay that amount using the smallest number of $20, $10, $5, and $1 bills:

```
Enter a dollar amount: 93

$20 bills: 4
$10 bills: 1
 $5 bills: 0
 $1 bills: 3
```

Hint: Divide the amount by 20 to determine the number of $20 bills needed, and then reduce the amount by the total value of the $20 bills. Repeat for the other bill sizes. Be sure to use integer values throughout, not floating-point numbers.

8. Write a program that calculates the remaining balance on a loan after the first, second, and third monthly payments:

```
Enter amount of loan: 20000.00
Enter interest rate: 6.0
Enter monthly payment: 386.66

Balance remaining after first payment: $19713.34
Balance remaining after second payment: $19425.25
Balance remaining after third payment: $19135.71
```

Display each balance with two digits after the decimal point. *Hint:* Each month, the balance is decreased by the amount of the payment, but increased by the balance times the monthly interest rate. To find the monthly interest rate, convert the interest rate entered by the user to a percentage and divide it by 12.

3 Formatted Input/Output

In seeking the unattainable, simplicity only gets in the way.

`scanf` and `printf`, which support formatted reading and writing, are two of the most frequently used functions in C. As this chapter shows, both are powerful but tricky to use properly. Section 3.1 describes `printf`, and Section 3.2 covers `scanf`. Neither section gives complete details, which will have to wait until Chapter 22.

3.1 The `printf` Function

The `printf` function is designed to display the contents of a string, known as the *format string*, with values possibly inserted at specified points in the string. When it's called, `printf` must be supplied with the format string, followed by any values that are to be inserted into the string during printing:

`printf(`*string*`,` *expr*$_1$`,` *expr*$_2$`, ...);`

The values displayed can be constants, variables, or more complicated expressions. There's no limit on the number of values that can be printed by a single call of `printf`.

The format string may contain both ordinary characters and *conversion specifications*, which begin with the `%` character. A conversion specification is a placeholder representing a value to be filled in during printing. The information that follows the `%` character *specifies* how the value is *converted* from its internal form (binary) to printed form (characters)—that's where the term "conversion specification" comes from. For example, the conversion specification `%d` specifies that `printf` is to convert an `int` value from binary to a string of decimal digits, while `%f` does the same for a `float` value.

Ordinary characters in a format string are printed exactly as they appear in the string; conversion specifications are replaced by the values to be printed. Consider the following example:

```
int i, j;
float x, y;

i = 10;
j = 20;
x = 43.2892f;
y = 5527.0f;

printf("i = %d, j = %d, x = %f, y = %f\n", i, j, x, y);
```

This call of `printf` produces the following output:

```
i = 10, j = 20, x = 43.289200, y = 5527.000000
```

The ordinary characters in the format string are simply copied to the output line. The four conversion specifications are replaced by the values of the variables `i`, `j`, `x`, and `y`, in that order.

C compilers aren't required to check that the number of conversion specifications in a format string matches the number of output items. The following call of `printf` has more conversion specifications than values to be printed:

```
printf("%d %d\n", i);    /*** WRONG ***/
```

`printf` will print the value of `i` correctly, then print a second (meaningless) integer value. A call with too few conversion specifications has similar problems:

```
printf("%d\n", i, j);    /*** WRONG ***/
```

In this case, `printf` prints the value of `i` but doesn't show the value of `j`.

Furthermore, compilers aren't required to check that a conversion specification is appropriate for the type of item being printed. If the programmer uses an incorrect specification, the program will simply produce meaningless output. Consider the following call of `printf`, in which the `int` variable `i` and the `float` variable `x` are in the wrong order:

```
printf("%f %d\n", i, x);    /*** WRONG ***/
```

Since `printf` must obey the format string, it will dutifully display a `float` value, followed by an `int` value. Unfortunately, both will be meaningless.

Conversion Specifications

Conversion specifications give the programmer a great deal of control over the appearance of output. On the other hand, they can be complicated and hard to read. In fact, describing conversion specifications in complete detail is too arduous a

task to tackle this early in the book. Instead, we'll just take a brief look at some of their more important capabilities.

In Chapter 2, we saw that a conversion specification can include formatting information. In particular, we used `%.1f` to display a `float` value with one digit after the decimal point. More generally, a conversion specification can have the form *%m.pX* or *%-m.pX*, where *m* and *p* are integer constants and *X* is a letter. Both *m* and *p* are optional; if *p* is omitted, the period that separates *m* and *p* is also dropped. In the conversion specification `%10.2f`, *m* is `10`, *p* is `2`, and *X* is `f`. In the specification `%10f`, *m* is `10` and *p* (along with the period) is missing, but in the specification `%.2f`, *p* is `2` and *m* is missing.

The ***minimum field width***, *m*, specifies the minimum number of characters to print. If the value to be printed requires fewer than *m* characters, the value is right-justified within the field. (In other words, extra spaces precede the value.) For example, the specification `%4d` would display the number 123 as `•123`. (In this chapter, I'll use • to represent the space character.) If the value to be printed requires more than *m* characters, the field width automatically expands to the necessary size. Thus, the specification `%4d` would display the number 12345 as `12345`—no digits are lost. Putting a minus sign in front of *m* causes left justification; the specification `%-4d` would display 123 as `123•`.

The meaning of the ***precision***, *p*, isn't as easily described, since it depends on the choice of *X*, the ***conversion specifier***. *X* indicates which conversion should be applied to the value before it's printed. The most common conversion specifiers for numbers are:

Q&A

- `d` — Displays an integer in decimal (base 10) form. *p* indicates the minimum number of digits to display (extra zeros are added to the beginning of the number if necessary); if *p* is omitted, it is assumed to have the value 1. (In other words, `%d` is the same as `%.1d`.)
- `e` — Displays a floating-point number in exponential format (scientific notation). *p* indicates how many digits should appear after the decimal point (the default is 6). If *p* is `0`, the decimal point is not displayed.
- `f` — Displays a floating-point number in "fixed decimal" format, without an exponent. *p* has the same meaning as for the `e` specifier.
- `g` — Displays a floating-point number in either exponential format or fixed decimal format, depending on the number's size. *p* indicates the maximum number of significant digits (*not* digits after the decimal point) to be displayed. Unlike the `f` conversion, the `g` conversion won't show trailing zeros. Furthermore, if the value to be printed has no digits after the decimal point, `g` doesn't display the decimal point.

The `g` specifier is especially useful for displaying numbers whose size can't be predicted when the program is written or that tend to vary widely in size. When used to print a moderately large or moderately small number, the `g` specifier uses fixed decimal format. But when used to print a very large or very small number, the `g` specifier switches to exponential format so that the number will require fewer characters.

specifiers for integers ➤7.1
specifiers for floats ➤7.2
specifiers for characters ➤7.3
specifiers for strings ➤13.3

There are many other specifiers besides `%d`, `%e`, `%f`, and `%g`. I'll gradually introduce many of them in subsequent chapters. For the full list, and for a complete explanation of the other capabilities of conversion specifications, consult Section 22.3.

PROGRAM Using `printf` to Format Numbers

The following program illustrates the use of `printf` to print integers and floating-point numbers in various formats.

tprintf.c

```
/* Prints int and float values in various formats */

#include <stdio.h>

int main(void)
{
  int i;
  float x;

  i = 40;
  x = 839.21f;

  printf("|%d|%5d|%-5d|%5.3d|\n", i, i, i, i);
  printf("|%10.3f|%10.3e|%-10g|\n", x, x, x);

  return 0;
}
```

The | characters in the `printf` format strings are there merely to help show how much space each number occupies when printed; unlike `%` or `\`, the | character has no special significance to `printf`. The output of this program is:

```
|40|   40|40   |  040|
|   839.210| 8.392e+02|839.21     |
```

Let's take a closer look at the conversion specifications used in this program:

- `%d` — Displays `i` in decimal form, using a minimum amount of space.
- `%5d` — Displays `i` in decimal form, using a minimum of five characters. Since `i` requires only two characters, three spaces were added.
- `%-5d` — Displays `i` in decimal form, using a minimum of five characters; since the value of `i` doesn't require five characters, the spaces are added afterward (that is, `i` is left-justified in a field of length five).
- `%5.3d` — Displays `i` in decimal form, using a minimum of five characters overall and a minimum of three digits. Since `i` is only two digits long, an extra zero was added to guarantee three digits. The resulting number is only three characters long, so two spaces were added, for a total of five characters (`i` is right-justified).
- `%10.3f` — Displays `x` in fixed decimal form, using 10 characters overall,

with three digits after the decimal point. Since x requires only seven characters (three before the decimal point, three after the decimal point, and one for the decimal point itself), three spaces precede x.

- %10.3e — Displays x in exponential form, using 10 characters overall, with three digits after the decimal point. x requires nine characters altogether (including the exponent), so one space precedes x.
- %-10g — Displays x in either fixed decimal form or exponential form, using 10 characters overall. In this case, printf chose to display x in fixed decimal form. The presence of the minus sign forces left justification, so x is followed by four spaces.

Escape Sequences

The \n code that we often use in format strings is called an *escape sequence*. Escape sequences enable strings to contain characters that would otherwise cause problems for the compiler, including nonprinting (control) characters and characters that have a special meaning to the compiler (such as "). We'll provide a complete list of escape sequences later; for now, here's a sample:

escape sequences ➤7.3

Alert (bell)	\a
Backspace	\b
New line	\n
Horizontal tab	\t

When they appear in printf format strings, these escape sequences represent actions to perform upon printing. Printing \a causes an audible beep on most machines. Printing \b moves the cursor back one position. Printing \n advances the cursor to the beginning of the next line. Printing \t moves the cursor to the next tab stop.

Q&A

A string may contain any number of escape sequences. Consider the following printf example, in which the format string contains six escape sequences:

```
printf("Item\tUnit\tPurchase\n\tPrice\tDate\n");
```

Executing this statement prints a two-line heading:

```
Item    Unit    Purchase
        Price   Date
```

Another common escape sequence is \", which represents the " character. Since the " character marks the beginning and end of a string, it can't appear within a string without the use of this escape sequence. Here's an example:

```
printf("\"Hello!\"");
```

This statement produces the following output:

```
"Hello!"
```

Incidentally, you can't just put a single \ character in a string; the compiler will assume that it's the beginning of an escape sequence. To print a single \ character, put two \ characters in the string:

```
printf("\\");    /* prints one \ character */
```

3.2 The `scanf` Function

Just as `printf` prints output in a specified format, `scanf` reads input according to a particular format. A `scanf` format string, like a `printf` format string, may contain both ordinary characters and conversion specifications. The conversions allowed with `scanf` are essentially the same as those used with `printf`.

In many cases, a `scanf` format string will contain only conversion specifications, as in the following example:

```
int i, j;
float x, y;

scanf("%d%d%f%f", &i, &j, &x, &y);
```

Suppose that the user enters the following input line:

```
1 -20 .3 -4.0e3
```

`scanf` will read the line, converting its characters to the numbers they represent, and then assign 1, –20, 0.3, and –4000.0 to `i`, `j`, `x`, and `y`, respectively. "Tightly packed" format strings like `"%d%d%f%f"` are common in `scanf` calls. `printf` format strings are less likely to have adjacent conversion specifications.

`scanf`, like `printf`, contains several traps for the unwary. When using `scanf`, the programmer must check that the number of conversion specifications matches the number of input variables and that each conversion is appropriate for the corresponding variable—as with `printf`, the compiler isn't required to check for a possible mismatch. Another trap involves the & symbol, which normally precedes each variable in a `scanf` call. The & is usually (but not always) required, and it's the programmer's responsibility to remember to use it.

⚠ Forgetting to put the & symbol in front of a variable in a call of `scanf` will have unpredictable—and possibly disastrous—results. A program crash is a common outcome. At the very least, the value that is read from the input won't be stored in the variable; instead, the variable will retain its old value (which may be meaningless if the variable wasn't given an initial value). Omitting the & is an extremely common error—be careful! Some compilers can spot this error and produce a warning message such as *"format argument is not a pointer."* (The term *pointer* is defined in Chapter 11; the & symbol is used to create a pointer to a variable.) If you get a warning, check for a missing &.

Calling scanf is a powerful but unforgiving way to read data. Many professional C programmers avoid scanf, instead reading all data in character form and converting it to numeric form later. We'll use scanf quite a bit, especially in the early chapters of this book, because it provides a simple way to read numbers. Be aware, however, that many of our programs won't behave properly if the user enters unexpected input. As we'll see later, it's possible to have a program test whether scanf successfully read the requested data (and attempt to recover if it didn't). Such tests are impractical for the programs in this book—they would add too many statements and obscure the point of the examples.

detecting errors in scanf ➤22.3

How scanf Works

scanf can actually do much more than I've indicated so far. It is essentially a "pattern-matching" function that tries to match up groups of input characters with conversion specifications.

Like the printf function, scanf is controlled by the format string. When it is called, scanf begins processing the information in the string, starting at the left. For each conversion specification in the format string, scanf tries to locate an item of the appropriate type in the input data, skipping blank space if necessary. scanf then reads the item, stopping when it encounters a character that can't possibly belong to the item. If the item was read successfully, scanf continues processing the rest of the format string. If any item is not read successfully, scanf returns immediately without looking at the rest of the format string (or the remaining input data).

As it searches for the beginning of a number, scanf ignores *white-space characters* (the space, horizontal and vertical tab, form-feed, and new-line characters). As a result, numbers can be put on a single line or spread out over several lines. Consider the following call of scanf:

```
scanf("%d%d%f%f", &i, &j, &x, &y);
```

Suppose that the user enters three lines of input:

```
  1
-20    .3
   -4.0e3
```

scanf sees one continuous stream of characters:

```
••1¤-20•••.3¤•••-4.0e3¤
```

(I'm using • to represent the space character and ¤ to represent the new-line character.) Since it skips over white-space characters as it looks for the beginning of each number, scanf will be able to read the numbers successfully. In the following diagram, an *s* under a character indicates that it was skipped, and an *r* indicates it was read as part of an input item:

```
••1¤-20•••.3¤•••-4.0e3¤
ssrsrrrsssrrssssrrrrrr
```

scanf "peeks" at the final new-line character without actually reading it. This new-line will be the first character read by the next call of scanf.

What rules does scanf follow to recognize an integer or a floating-point number? When asked to read an integer, scanf first searches for a digit, a plus sign, or a minus sign; it then reads digits until it reaches a nondigit. When asked to read a floating-point number, scanf looks for

a plus or minus sign (optional), followed by

a series of digits (possibly containing a decimal point), followed by

an exponent (optional). An exponent consists of the letter e (or E), an optional sign, and one or more digits.

The %e, %f, and %g conversions are interchangeable when used with scanf; all three follow the same rules for recognizing a floating-point number.

When scanf encounters a character that can't be part of the current item, the character is "put back" to be read again during the scanning of the next input item or during the next call of scanf. Consider the following (admittedly pathological) arrangement of our four numbers:

Q&A

```
1-20.3-4.0e3¤
```

Let's use the same call of scanf as before:

```
scanf("%d%d%f%f", &i, &j, &x, &y);
```

Here's how scanf would process the new input:

- Conversion specification: %d. The first nonblank input character is 1; since integers can begin with 1, scanf then reads the next character, -. Recognizing that - can't appear inside an integer, scanf stores 1 into i and puts the - character back.
- Conversion specification: %d. scanf then reads the characters -, 2, 0, and . (period). Since an integer can't contain a decimal point, scanf stores –20 into j and puts the . character back.
- Conversion specification: %f. scanf reads the characters ., 3, and -. Since a floating-point number can't contain a minus sign after a digit, scanf stores 0.3 into x and puts the - character back.
- Conversion specification: %f. Lastly, scanf reads the characters -, 4, ., 0, e, 3, and ¤ (new-line). Since a floating-point number can't contain a new-line character, scanf stores -4.0×10^3 into y and puts the new-line character back.

In this example, scanf was able to match every conversion specification in the format string with an input item. Since the new-line character wasn't read, it will be left for the next call of scanf.

Ordinary Characters in Format Strings

The concept of pattern-matching can be taken one step further by writing format strings that contain ordinary characters in addition to conversion specifications. The action that `scanf` takes when it processes an ordinary character in a format string depends on whether or not it's a white-space character.

- *White-space characters.* When it encounters one or more consecutive white-space characters in a format string, `scanf` repeatedly reads white-space characters from the input until it reaches a non-white-space character (which is "put back"). The number of white-space characters in the format string is irrelevant; one white-space character in the format string will match any number of white-space characters in the input. (Incidentally, putting a white-space character in a format string doesn't force the input to contain white-space characters. A white-space character in a format string matches *any* number of white-space characters in the input, including none.)
- *Other characters.* When it encounters a non-white-space character in a format string, `scanf` compares it with the next input character. If the two characters match, `scanf` discards the input character and continues processing the format string. If the characters don't match, `scanf` puts the offending character back into the input, then aborts without further processing the format string or reading characters from the input.

  For example, suppose that the format string is `"%d/%d"`. If the input is

```
•5/•96
```

`scanf` skips the first space while looking for an integer, matches `%d` with `5`, matches `/` with `/`, skips a space while looking for another integer, and matches `%d` with `96`. On the other hand, if the input is

```
•5•/•96
```

`scanf` skips one space, matches `%d` with `5`, then attempts to match the `/` in the format string with a space in the input. There's no match, so `scanf` puts the space back; the `•/•96` characters remain to be read by the next call of `scanf`. To allow spaces after the first number, we should use the format string `"%d /%d"` instead.

Confusing `printf` with `scanf`

Although calls of `scanf` and `printf` may appear similar, there are significant differences between the two functions; ignoring these differences can be hazardous to the health of your program.

One common mistake is to put `&` in front of variables in a call of `printf`:

```
printf("%d %d\n", &i, &j);    /*** WRONG ***/
```

Fortunately, this mistake is fairly easy to spot: `printf` will display a couple of odd-looking numbers instead of the values of `i` and `j`.

Since `scanf` normally skips white-space characters when looking for data items, there's often no need for a format string to include characters other than conversion specifications. Incorrectly assuming that `scanf` format strings should resemble `printf` format strings—another common error—may cause `scanf` to behave in unexpected ways. Let's see what happens when the following call of `scanf` is executed:

```
scanf("%d, %d", &i, &j);
```

`scanf` will first look for an integer in the input, which it stores in the variable `i`. `scanf` will then try to match a comma with the next input character. If the next input character is a space, not a comma, `scanf` will terminate without reading a value for `j`.

Although `printf` format strings often end with `\n`, putting a new-line character at the end of a `scanf` format string is usually a bad idea. To `scanf`, a new-line character in a format string is equivalent to a space; both cause `scanf` to advance to the next non-white-space character. For example, if the format string is `"%d\n"`, `scanf` will skip white space, read an integer, then skip to the next non-white-space character. A format string like this can cause an interactive program to "hang" until the user enters a nonblank character.

PROGRAM

Adding Fractions

To illustrate `scanf`'s ability to match patterns, consider the problem of reading a fraction entered by the user. Fractions are customarily written in the form *numerator/denominator*. Instead of having the user enter the numerator and denominator of a fraction as separate integers, `scanf` makes it possible to read the entire fraction. The following program, which adds two fractions, illustrates this technique.

addfrac.c

```
/* Adds two fractions */

#include <stdio.h>

int main(void)
{
  int num1, denom1, num2, denom2, result_num, result_denom;

  printf("Enter first fraction: ");
  scanf("%d/%d", &num1, &denom1);

  printf("Enter second fraction: ");
  scanf("%d/%d", &num2, &denom2);

  result_num = num1 * denom2 + num2 * denom1;
```

```
  result_denom = denom1 * denom2;
  printf("The sum is %d/%d\n", result_num, result_denom);

  return 0;
}
```

A session with this program might have the following appearance:

```
Enter first fraction: 5/6
Enter second fraction: 3/4
The sum is 38/24
```

Note that the resulting fraction isn't reduced to lowest terms.

Q & A

***Q: I've seen the `%i` conversion used to read and write integers. What's the difference between `%i` and `%d`? [p. 39]**

A: In a `printf` format string, there's no difference between the two. In a `scanf` format string, however, `%d` can only match an integer written in decimal (base 10) form, while `%i` can match an integer expressed in octal (base 8), decimal, or hexadecimal (base 16). If an input number has a `0` prefix (as in `056`), `%i` treats it as an octal number; if it has a `0x` or `0X` prefix (as in `0x56`), `%i` treats it as a hex number. Using `%i` instead of `%d` to read a number can have surprising results if the user should accidentally put `0` at the beginning of the number. Because of this trap, I recommend sticking with `%d`.

octal numbers ➤7.1

hexadecimal numbers ➤7.1

Q: If `printf` treats `%` as the beginning of a conversion specification, how can I print the `%` character?

A: If `printf` encounters two consecutive `%` characters in a format string, it prints a single `%` character. For example, the statement

```
printf("Net profit: %d%%\n", profit);
```

might print

```
Net profit: 10%
```

Q: The `\t` escape is supposed to cause `printf` to advance to the next tab stop. How do I know how far apart tab stops are? [p. 41]

A: You don't. The effect of printing `\t` isn't defined in C; it depends on what your operating system does when asked to print a tab character. Tab stops are typically eight characters apart, but C makes no guarantee.

Q: What does `scanf` do if it's asked to read a number but the user enters nonnumeric input?

A: Let's look at the following example:

```
printf("Enter a number: ");
scanf("%d", &i);
```

Suppose that the user enters a valid number, followed by nonnumeric characters:

```
Enter a number: 23foo
```

In this case, `scanf` reads the `2` and the `3`, storing 23 in `i`. The remaining characters (`foo`) are left to be read by the next call of `scanf` (or some other input function). On the other hand, suppose that the input is invalid from the beginning:

```
Enter a number: foo
```

In this case, the value of `i` is undefined and `foo` is left for the next `scanf`.

What can we do about this sad state of affairs? Later, we'll see how to test whether a call of `scanf` has succeeded. If the call fails, we can have the program either terminate or try to recover, perhaps by discarding the offending input and asking the user to try again. (Ways to discard bad input are discussed in the Q&A section at the end of Chapter 22.)

detecting errors in `scanf` ➤*22.3*

Q: I don't understand how `scanf` can "put back" characters and read them again later. [p. 44]

A: As it turns out, programs don't read user input as it is typed. Instead, input is stored in a hidden buffer, to which `scanf` has access. It's easy for `scanf` to put characters back into the buffer for subsequent reading. Chapter 22 discusses input buffering in more detail.

Q: What does `scanf` do if the user puts punctuation marks (commas, for example) between numbers?

A: Let's look at a simple example. Suppose that we try to read a pair of integers using `scanf`:

```
printf("Enter two numbers: ");
scanf("%d%d", &i, &j);
```

If the user enters

```
4,28
```

`scanf` will read the 4 and store it in `i`. As it searches for the beginning of the second number, `scanf` encounters the comma. Since numbers can't begin with a comma, `scanf` returns immediately. The comma and the second number are left for the next call of `scanf`.

Of course, we can easily solve the problem by adding a comma to the format string if we're sure that the numbers will *always* be separated by a comma:

```
printf("Enter two numbers, separated by a comma: ");
scanf("%d,%d", &i, &j);
```

Exercises

Section 3.1

1. What output do the following calls of `printf` produce?

   (a) `printf("%6d,%4d", 86, 1040);`

   (b) `printf("%12.5e", 30.253);`

   (c) `printf("%.4f", 83.162);`

   (d) `printf("%-6.2g", .0000009979);`

W 2. Write calls of `printf` that display a `float` variable `x` in the following formats.

   (a) Exponential notation; left-justified in a field of size 8; one digit after the decimal point.

   (b) Exponential notation; right-justified in a field of size 10; six digits after the decimal point.

   (c) Fixed decimal notation; left-justified in a field of size 8; three digits after the decimal point.

   (d) Fixed decimal notation; right-justified in a field of size 6; no digits after the decimal point.

Section 3.2

3. For each of the following pairs of `scanf` format strings, indicate whether or not the two strings are equivalent. If they're not, show how they can be distinguished.

   (a) `"%d"` versus `" %d"`

   (b) `"%d-%d-%d"` versus `"%d -%d -%d"`

   (c) `"%f"` versus `"%f "`

   (d) `"%f,%f"` versus `"%f, %f"`

*4. Suppose that we call `scanf` as follows:

```
scanf("%d%f%d", &i, &x, &j);
```

If the user enters

```
10.3 5 6
```

what will be the values of `i`, `x`, and `j` after the call? (Assume that `i` and `j` are `int` variables and `x` is a `float` variable.)

W *5. Suppose that we call `scanf` as follows:

```
scanf("%f%d%f", &x, &i, &y);
```

If the user enters

```
12.3 45.6 789
```

what will be the values of `x`, `i`, and `y` after the call? (Assume that `x` and `y` are `float` variables and `i` is an `int` variable.)

6. Show how to modify the `addfrac.c` program of Section 3.2 so that the user is allowed to enter fractions that contain spaces before and after each / character.

---

*Starred exercises are tricky—the correct answer is usually not the obvious one. Read the question thoroughly, review the relevant section if necessary, and be careful!

Programming Projects

W 1. Write a program that accepts a date from the user in the form *mm/dd/yyyy* and then displays it in the form *yyyymmdd*:

```
Enter a date (mm/dd/yyyy): 2/17/2011
You entered the date 20110217
```

2. Write a program that formats product information entered by the user. A session with the program should look like this:

```
Enter item number: 583
Enter unit price: 13.5
Enter purchase date (mm/dd/yyyy): 10/24/2010

Item            Unit            Purchase
                Price           Date
583             $  13.50        10/24/2010
```

The item number and date should be left justified; the unit price should be right justified. Allow dollar amounts up to $9999.99. *Hint:* Use tabs to line up the columns.

W 3. Books are identified by an International Standard Book Number (ISBN). ISBNs assigned after January 1, 2007 contain 13 digits, arranged in five groups, such as 978-0-393-97950-3. (Older ISBNs use 10 digits.) The first group (the *GS1 prefix*) is currently either 978 or 979. The *group identifier* specifies the language or country of origin (for example, 0 and 1 are used in English-speaking countries). The *publisher code* identifies the publisher (393 is the code for W. W. Norton). The *item number* is assigned by the publisher to identify a specific book (97950 is the code for this book). An ISBN ends with a *check digit* that's used to verify the accuracy of the preceding digits. Write a program that breaks down an ISBN entered by the user:

```
Enter ISBN: 978-0-393-97950-3
GS1 prefix: 978
Group identifier: 0
Publisher code: 393
Item number: 97950
Check digit: 3
```

Note: The number of digits in each group may vary: you can't assume that groups have the lengths shown in this example. Test your program with actual ISBN values (usually found on the back cover of a book and on the copyright page).

4. Write a program that prompts the user to enter a telephone number in the form (xxx) xxx-xxxx and then displays the number in the form xxx.xxx.xxx:

```
Enter phone number [(xxx) xxx-xxxx]: (404) 817-6900
You entered 404.817.6900
```

5. Write a program that asks the user to enter the numbers from 1 to 16 (in any order) and then displays the numbers in a 4 by 4 arrangement, followed by the sums of the rows, columns, and diagonals:

```
Enter the numbers from 1 to 16 in any order:
16 3 2 13 5 10 11 8 9 6 7 12 4 15 14 1
```

```
16  3  2 13
 5 10 11  8
 9  6  7 12
 4 15 14  1

Row sums: 34 34 34 34
Column sums: 34 34 34 34
Diagonal sums: 34 34
```

If the row, column, and diagonal sums are all the same (as they are in this example), the numbers are said to form a ***magic square.*** The magic square shown here appears in a 1514 engraving by artist and mathematician Albrecht Dürer. (Note that the middle numbers in the last row give the date of the engraving.)

6. Modify the `addfrac.c` program of Section 3.2 so that the user enters both fractions at the same time, separated by a plus sign:

```
Enter two fractions separated by a plus sign: 5/6+3/4
The sum is 38/24
```

4 Expressions

One does not learn computing by using a hand calculator, but one can forget arithmetic.

One of C's distinguishing characteristics is its emphasis on expressions—formulas that show how to compute a value—rather than statements. The simplest expressions are variables and constants. A variable represents a value to be computed as the program runs; a constant represents a value that doesn't change. More complicated expressions apply operators to operands (which are themselves expressions). In the expression `a + (b * c)`, the `+` operator is applied to the operands `a` and `(b * c)`, both of which are expressions in their own right.

Operators are the basic tools for building expressions, and C has an unusually rich collection of them. To start off, C provides the rudimentary operators that are found in most programming languages:

- Arithmetic operators, including addition, subtraction, multiplication, and division.
- Relational operators to perform comparisons such as "`i` is *greater than* 0."
- Logical operators to build conditions such as "`i` is greater than 0 *and* `i` is less than 10."

But C doesn't stop here; it goes on to provide dozens of other operators. There are so many operators, in fact, that we'll need to introduce them gradually over the first twenty chapters of this book. Mastering so many operators can be a chore, but it's essential to becoming proficient at C.

In this chapter, we'll cover some of C's most fundamental operators: the arithmetic operators (Section 4.1), the assignment operators (Section 4.2), and the increment and decrement operators (Section 4.3). Section 4.1 also explains operator precedence and associativity, which are important for expressions that contain more than one operator. Section 4.4 describes how C expressions are evaluated. Finally, Section 4.5 introduces the expression statement, an unusual feature that allows any expression to serve as a statement.

4.1 Arithmetic Operators

The *arithmetic operators*—operators that perform addition, subtraction, multiplication, and division—are the workhorses of many programming languages, including C. Table 4.1 shows C's arithmetic operators.

Table 4.1
Arithmetic Operators

Unary	*Binary*	
	Additive	*Multiplicative*
+ unary plus - unary minus	+ addition - subtraction	* multiplication / division % remainder

The additive and multiplicative operators are said to be ***binary*** because they require *two* operands. The ***unary*** operators require *one* operand:

```
i = +1;    /* + used as a unary operator */
j = -i;    /* - used as a unary operator */
```

The unary + operator does nothing; in fact, it didn't even exist in K&R C. It's used primarily to emphasize that a numeric constant is positive.

The binary operators probably look familiar. The only one that might not is %, the remainder operator. The value of `i % j` is the remainder when `i` is divided by `j`. For example, the value of `10 % 3` is 1, and the value of `12 % 4` is 0.

Q&A The binary operators in Table 4.1—with the exception of %—allow either integer or floating-point operands, with mixing allowed. When `int` and `float` operands are mixed, the result has type `float`. Thus, `9 + 2.5f` has the value 11.5, and `6.7f / 2` has the value 3.35.

The / and % operators require special care:

- The / operator can produce surprising results. When both of its operands are integers, the / operator "truncates" the result by dropping the fractional part. Thus, the value of `1 / 2` is 0, not 0.5.
- The % operator requires integer operands; if either operand is not an integer, the program won't compile.
- Using zero as the right operand of either / or % causes undefined behavior.

undefined behavior ➤4.4

Q&A

- Describing the result when / and % are used with negative operands is tricky. The C89 standard states that if either operand is negative, the result of a division can be rounded either up or down. (For example, the value of `-9 / 7` could be either –1 or –2). If `i` or `j` is negative, the sign of `i % j` in C89 depends on the implementation. (For example, the value of `-9 % 7` could be either –2 or 5). In C99, on the other hand, the result of a division is always truncated toward zero (so `-9 / 7` has the value –1) and the value of `i % j` has the same sign as `i` (hence the value of `-9 % 7` is –2).

C99

Implementation-Defined Behavior

The term ***implementation-defined*** will arise often enough that it's worth taking a moment to discuss it. The C standard deliberately leaves parts of the language unspecified, with the understanding that an "implementation"—the software needed to compile, link, and execute programs on a particular platform—will fill in the details. As a result, the behavior of the program may vary somewhat from one implementation to another. The behavior of the / and % operators for negative operands in C89 is an example of implementation-defined behavior.

Leaving parts of the language unspecified may seem odd or even dangerous, but it reflects C's philosophy. One of the language's goals is efficiency, which often means matching the way that hardware behaves. Some CPUs yield –1 when –9 is divided by 7, while others produce –2; the C89 standard simply reflects this fact of life.

It's best to avoid writing programs that depend on implementation-defined behavior. If that's not possible, at least check the manual carefully—the C standard requires that implementation-defined behavior be documented.

Operator Precedence and Associativity

When an expression contains more than one operator, its interpretation may not be immediately clear. For example, does `i + j * k` mean "add `i` and `j`, then multiply the result by `k`," or does it mean "multiply `j` and `k`, then add `i`"? One solution to this problem is to add parentheses, writing either `(i + j) * k` or `i + (j * k)`. As a general rule, C allows the use of parentheses for grouping in all expressions.

What if we don't use parentheses, though? Will the compiler interpret `i + j * k` as `(i + j) * k` or `i + (j * k)`? Like many other languages, C uses ***operator precedence*** rules to resolve this potential ambiguity. The arithmetic operators have the following relative precedence:

Highest:	+	-	(unary)
	*	/	%
Lowest:	+	-	(binary)

Operators listed on the same line (such as + and -) have equal precedence.

When two or more operators appear in the same expression, we can determine how the compiler will interpret the expression by repeatedly putting parentheses around subexpressions, starting with high-precedence operators and working down to low-precedence operators. The following examples illustrate the result:

`i + j * k`	is equivalent to	`i + (j * k)`
`-i * -j`	is equivalent to	`(-i) * (-j)`
`+i + j / k`	is equivalent to	`(+i) + (j / k)`

Operator precedence rules alone aren't enough when an expression contains two or more operators at the same level of precedence. In this situation, the ***associativity***

of the operators comes into play. An operator is said to be ***left associative*** if it groups from left to right. The binary arithmetic operators (`*`, `/`, `%`, `+`, and `-`) are all left associative, so

```
i - j - k     is equivalent to   (i - j) - k
i * j / k     is equivalent to   (i * j) / k
```

An operator is *right associative* if it groups from right to left. The unary arithmetic operators (`+` and `-`) are both right associative, so

```
- + i              is equivalent to   -(+i)
```

Precedence and associativity rules are important in many languages, but especially so in C. However, C has so many operators (almost fifty!) that few programmers bother to memorize the precedence and associativity rules. Instead, they consult a table of operators when in doubt or just use plenty of parentheses.

table of operators ➤*Appendix A*

PROGRAM

Computing a UPC Check Digit

For a number of years, manufacturers of goods sold in U.S. and Canadian stores have put a bar code on each product. This code, known as a Universal Product Code (UPC), identifies both the manufacturer and the product. Each bar code represents a twelve-digit number, which is usually printed underneath the bars. For example, the following bar code comes from a package of Stouffer's French Bread Pepperoni Pizza:

The digits

```
0  13800 15173  5
```

appear underneath the bar code. The first digit identifies the type of item (`0` or `7` for most items, `2` for items that must be weighed, `3` for drugs and health-related merchandise, and `5` for coupons). The first group of five digits identifies the manufacturer (13800 is the code for Nestlé USA's Frozen Food Division). The second group of five digits identifies the product (including package size). The final digit is a "check digit," whose only purpose is to help identify an error in the preceding digits. If the UPC is scanned incorrectly, the first 11 digits probably won't be consistent with the last digit, and the store's scanner will reject the entire code.

Here's one method of computing the check digit:

Add the first, third, fifth, seventh, ninth, and eleventh digits.
Add the second, fourth, sixth, eighth, and tenth digits.

Multiply the first sum by 3 and add it to the second sum.
Subtract 1 from the total.
Compute the remainder when the adjusted total is divided by 10.
Subtract the remainder from 9.

Using the Stouffer's example, we get 0 + 3 + 0 + 1 + 1 + 3 = 8 for the first sum and 1 + 8 + 0 + 5 + 7 = 21 for the second sum. Multiplying the first sum by 3 and adding the second yields 45. Subtracting 1 gives 44. The remainder upon dividing by 10 is 4. When the remainder is subtracted from 9, the result is 5. Here are a couple of other UPCs, in case you want to try your hand at computing the check digit (raiding the kitchen cabinet for the answer is *not* allowed):

Jif Creamy Peanut Butter (18 oz.): 0 51500 24128 ?
Ocean Spray Jellied Cranberry Sauce (8 oz.): 0 31200 01005 ?

The answers appear at the bottom of the page.

Let's write a program that calculates the check digit for an arbitrary UPC. We'll ask the user to enter the first 11 digits of the UPC, then we'll display the corresponding check digit. To avoid confusion, we'll ask the user to enter the number in three parts: the single digit at the left, the first group of five digits, and the second group of five digits. Here's what a session with the program will look like:

```
Enter the first (single) digit: 0
Enter first group of five digits: 13800
Enter second group of five digits: 15173
Check digit: 5
```

Instead of reading each digit group as a *five*-digit number, we'll read it as five *one*-digit numbers. Reading the numbers as single digits is more convenient; also, we won't have to worry that one of the five-digit numbers is too large to store in an `int` variable. (Some older compilers limit the maximum value of an `int` variable to 32,767.) To read single digits, we'll use `scanf` with the `%1d` conversion specification, which matches a one-digit integer.

upc.c

```
/* Computes a Universal Product Code check digit */

#include <stdio.h>

int main(void)
{
  int d, i1, i2, i3, i4, i5, j1, j2, j3, j4, j5,
      first_sum, second_sum, total;

  printf("Enter the first (single) digit: ");
  scanf("%1d", &d);
  printf("Enter first group of five digits: ");
  scanf("%1d%1d%1d%1d%1d", &i1, &i2, &i3, &i4, &i5);
  printf("Enter second group of five digits: ");
  scanf("%1d%1d%1d%1d%1d", &j1, &j2, &j3, &j4, &j5);
```

The missing check digits are 8 (Jif) and 6 (Ocean Spray).

```
    first_sum = d + i2 + i4 + j1 + j3 + j5;
    second_sum = i1 + i3 + i5 + j2 + j4;
    total = 3 * first_sum + second_sum;

    printf("Check digit: %d\n", 9 - ((total - 1) % 10));

    return 0;
}
```

Note that the expression `9 - ((total - 1) % 10)` could have been written as `9 - (total - 1) % 10`, but the extra set of parentheses makes it easier to understand.

4.2 Assignment Operators

Once the value of an expression has been computed, we'll often need to store it in a variable for later use. C's = (*simple assignment*) operator is used for that purpose. For updating a value already stored in a variable, C provides an assortment of compound assignment operators.

Simple Assignment

The effect of the assignment $v = e$ is to evaluate the expression e and copy its value into v. As the following examples show, e can be a constant, a variable, or a more complicated expression:

```
i = 5;              /* i is now 5   */
j = i;              /* j is now 5   */
k = 10 * i + j;     /* k is now 55  */
```

If v and e don't have the same type, then the value of e is converted to the type of v as the assignment takes place:

```
int i;
float f;

i = 72.99f;     /* i is now 72 */
f = 136;        /* f is now 136.0 */
```

conversion during assignment ➤ 7.4

We'll return to the topic of type conversion later.

In many programming languages, assignment is a *statement*; in C, however, assignment is an *operator*, just like +. In other words, the act of assignment produces a result, just as adding two numbers produces a result. The value of an assignment $v = e$ is the value of v *after* the assignment. Thus, the value of `i = 72.99f` is 72 (not 72.99).

Side Effects

We don't normally expect operators to modify their operands, since operators in mathematics don't. Writing `i + j` doesn't modify either `i` or `j`; it simply computes the result of adding `i` and `j`.

Most C operators don't modify their operands, but some do. We say that these operators have ***side effects***, since they do more than just compute a value. The simple assignment operator is the first operator we've seen that has side effects; it modifies its left operand. Evaluating the expression `i = 0` produces the result 0 and—as a side effect—assigns 0 to `i`.

Since assignment is an operator, several assignments can be chained together:

```
i = j = k = 0;
```

The = operator is right associative, so this assignment is equivalent to

```
i = (j = (k = 0));
```

The effect is to assign 0 first to `k`, then to `j`, and finally to `i`.

Watch out for unexpected results in chained assignments as a result of type conversion:

```
int i;
float f;

f = i = 33.3f;
```

`i` is assigned the value 33, then `f` is assigned 33.0 (not 33.3, as you might think).

In general, an assignment of the form *v* = *e* is allowed wherever a value of type *v* would be permitted. In the following example, the expression `j = i` copies `i` to `j`; the new value of `j` is then added to `1`, producing the new value of `k`:

```
i = 1;
k = 1 + (j = i);
printf("%d %d %d\n", i, j, k);   /* prints "1 1 2" */
```

Using the assignment operator in this fashion usually isn't a good idea. For one thing, "embedded assignments" can make programs hard to read. They can also be a source of subtle bugs, as we'll see in Section 4.4.

Lvalues

Q&A

Most C operators allow their operands to be variables, constants, or expressions containing other operators. The assignment operator, however, requires an *lvalue*

as its left operand. An lvalue (pronounced "L-value") represents an object stored in computer memory, not a constant or the result of a computation. Variables are lvalues; expressions such as `10` or `2 * i` are not. At this point, variables are the only lvalues that we know about; other kinds of lvalues will appear in later chapters.

Since the assignment operator requires an lvalue as its left operand, it's illegal to put any other kind of expression on the left side of an assignment expression:

```
12 = i;       /*** WRONG ***/
i + j = 0;    /*** WRONG ***/
-i = j;       /*** WRONG ***/
```

The compiler will detect errors of this nature, and you'll get an error message such as *"invalid lvalue in assignment."*

Compound Assignment

Assignments that use the old value of a variable to compute its new value are common in C programs. The following statement, for example, adds 2 to the value stored in `i`:

```
i = i + 2;
```

C's *compound assignment* operators allow us to shorten this statement and others like it. Using the += operator, we simply write:

```
i += 2;    /* same as i = i + 2; */
```

The += operator adds the value of the right operand to the variable on the left.

There are nine other compound assignment operators, including the following:

```
-=   *=   /=   %=
```

other assignment operators ➤20.1

(We'll cover the remaining compound assignment operators in a later chapter.) All compound assignment operators work in much the same way:

v += *e* adds *v* to *e*, storing the result in *v*
v -= *e* subtracts *e* from *v*, storing the result in *v*
v *= *e* multiplies *v* by *e*, storing the result in *v*
v /= *e* divides *v* by *e*, storing the result in *v*
v %= *e* computes the remainder when *v* is divided by *e*, storing the result in *v*

Note that I've been careful not to say that *v* += *e* is "equivalent" to *v* = *v* + *e*. One problem is operator precedence: `i *= j + k` isn't the same as `i = i * j + k`. There are also rare cases in which *v* += *e* differs from *v* = *v* + *e* because *v* itself has a side effect. Similar remarks apply to the other compound assignment operators.

Q&A

When using the compound assignment operators, be careful not to switch the two characters that make up the operator. Switching the characters may yield an expression that is acceptable to the compiler but that doesn't have the intended meaning. For example, if you meant to write `i += j` but typed `i =+ j` instead, the

program will still compile. Unfortunately, the latter expression is equivalent to `i = (+j)`, which merely copies the value of `j` into `i`.

The compound assignment operators have the same properties as the = operator. In particular, they're right associative, so the statement

```
i += j += k;
```

means

```
i += (j += k);
```

4.3 Increment and Decrement Operators

Two of the most common operations on a variable are "incrementing" (adding 1) and "decrementing" (subtracting 1). We can, of course, accomplish these tasks by writing

```
i = i + 1;
j = j - 1;
```

The compound assignment operators allow us to condense these statements a bit:

```
i += 1;
j -= 1;
```

Q&A

But C allows increments and decrements to be shortened even further, using the `++` (*increment*) and `--` (*decrement*) operators.

At first glance, the increment and decrement operators are simplicity itself: `++` adds 1 to its operand, whereas `--` subtracts 1. Unfortunately, this simplicity is misleading—the increment and decrement operators can be tricky to use. One complication is that `++` and `--` can be used as *prefix* operators (`++i` and `--i`, for example) or *postfix* operators (`i++` and `i--`). The correctness of a program may hinge on picking the proper version.

Another complication is that, like the assignment operators, `++` and `--` have side effects: they modify the values of their operands. Evaluating the expression `++i` (a "pre-increment") yields `i + 1` and—as a side effect—increments `i`:

```
i = 1;
printf("i is %d\n", ++i);    /* prints "i is 2" */
printf("i is %d\n", i);      /* prints "i is 2" */
```

Evaluating the expression `i++` (a "post-increment") produces the result `i`, but causes `i` to be incremented afterwards:

```
i = 1;
printf("i is %d\n", i++);    /* prints "i is 1" */
printf("i is %d\n", i);      /* prints "i is 2" */
```

The first `printf` shows the original value of `i`, before it is incremented. The second `printf` shows the new value. As these examples illustrate, `++i` means "increment `i` immediately," while `i++` means "use the old value of `i` for now, but increment `i` later." How much later? The C standard doesn't specify a precise time, but it's safe to assume that `i` will be incremented before the next statement is executed.

Q&A

The `--` operator has similar properties:

```
i = 1;
printf("i is %d\n", --i);    /* prints "i is 0" */
printf("i is %d\n", i);      /* prints "i is 0" */

i = 1;
printf("i is %d\n", i--);    /* prints "i is 1" */
printf("i is %d\n", i);      /* prints "i is 0" */
```

When `++` or `--` is used more than once in the same expression, the result can often be hard to understand. Consider the following statements:

```
i = 1;
j = 2;
k = ++i + j++;
```

What are the values of `i`, `j`, and `k` after these statements are executed? Since `i` is incremented *before* its value is used, but `j` is incremented *after* it is used, the last statement is equivalent to

```
i = i + 1;
k = i + j;
j = j + 1;
```

so the final values of `i`, `j`, and `k` are 2, 3, and 4, respectively. In contrast, executing the statements

```
i = 1;
j = 2;
k = i++ + j++;
```

will give `i`, `j`, and `k` the values 2, 3, and 3, respectively.

For the record, the postfix versions of `++` and `--` have higher precedence than unary plus and minus and are left associative. The prefix versions have the same precedence as unary plus and minus and are right associative.

4.4 Expression Evaluation

Table 4.2 summarizes the operators we've seen so far. (Appendix A has a similar table that shows *all* operators.) The first column shows the precedence of each

Table 4.2
A Partial List of C Operators

Precedence	*Name*	*Symbol(s)*	*Associativity*
1	increment (postfix) decrement (postfix)	++ --	left
2	increment (prefix) decrement (prefix) unary plus unary minus	++ -- + -	right
3	multiplicative	* / %	left
4	additive	+ -	left
5	assignment	= *= /= %= += -=	right

operator relative to the other operators in the table (the highest precedence is 1; the lowest is 5). The last column shows the associativity of each operator.

Table 4.2 (or its larger cousin in Appendix A) has a variety of uses. Let's look at one of these. Suppose that we run across a complicated expression such as

```
a = b += c++ - d + --e / -f
```

as we're reading someone's program. This expression would be easier to understand if there were parentheses to show how the expression is constructed from subexpressions. With the help of Table 4.2, adding parentheses to an expression is easy: after examining the expression to find the operator with highest precedence, we put parentheses around the operator and its operands, indicating that it should be treated as a single operand from that point onwards. We then repeat the process until the expression is fully parenthesized.

In our example, the operator with highest precedence is ++, used here as a postfix operator, so we put parentheses around ++ and its operand:

```
a = b += (c++) - d + --e / -f
```

We now spot a prefix -- operator and a unary minus operator (both precedence 2) in the expression:

```
a = b += (c++) - d + (--e) / (-f)
```

Note that the other minus sign has an operand to its immediate left, so it must be a subtraction operator, not a unary minus operator.

Next, we notice the / operator (precedence 3):

```
a = b += (c++) - d + ((--e) / (-f))
```

The expression contains two operators with precedence 4, subtraction and addition. Whenever two operators with the same precedence are adjacent to an operand, we've got to be careful about associativity. In our example, - and + are both adjacent to d, so associativity rules apply. The - and + operators group from left to right, so parentheses go around the subtraction first, then the addition:

```
a = b += (((c++) - d) + ((--e) / (-f)))
```

The only remaining operators are = and +=. Both operators are adjacent to b, so we must take associativity into account. Assignment operators group from right to left, so parentheses go around the += expression first, then the = expression:

```
(a = (b += (((c++) - d) + ((--e) / (-f)))))
```

The expression is now fully parenthesized.

Order of Subexpression Evaluation

The rules of operator precedence and associativity allow us to break any C expression into subexpressions—to determine uniquely where the parentheses would go if the expression were fully parenthesized. Paradoxically, these rules don't always allow us to determine the value of the expression, which may depend on the order in which its subexpressions are evaluated.

logical *and* and *or* operators ➤5.1

conditional operator ➤5.2

comma operator ➤6.3

C doesn't define the order in which subexpressions are evaluated (with the exception of subexpressions involving the logical *and*, logical *or*, conditional, and comma operators). Thus, in the expression `(a + b) * (c - d)` we don't know whether `(a + b)` will be evaluated before `(c - d)`.

Most expressions have the same value regardless of the order in which their subexpressions are evaluated. However, this may not be true when a subexpression modifies one of its operands. Consider the following example:

```
a = 5;
c = (b = a + 2) - (a = 1);
```

The effect of executing the second statement is undefined; the C standard doesn't say what will happen. With most compilers, the value of `c` will be either 6 or 2. If the subexpression `(b = a + 2)` is evaluated first, `b` is assigned the value 7 and `c` is assigned 6. But if `(a = 1)` is evaluated first, `b` is assigned 3 and `c` is assigned 2.

⚠ Avoid writing expressions that access the value of a variable and also modify the variable elsewhere in the expression. The expression `(b = a + 2) - (a = 1)` accesses the value of `a` (in order to compute `a + 2`) and also modifies the value of `a` (by assigning it `1`). Some compilers may produce a warning message such as "*operation on 'a' may be undefined*" when they encounter such an expression.

To prevent problems, it's a good idea to avoid using the assignment operators in subexpressions; instead, use a series of separate assignments. For example, the statements above could be rewritten as

```
a = 5;
b = a + 2;
a = 1;
c = b - a;
```

The value of `c` will always be 6 after these statements are executed.

Besides the assignment operators, the only operators that modify their operands are increment and decrement. When using these operators, be careful that your expressions don't depend on a particular order of evaluation. In the following example, `j` may be assigned one of two values:

```
i = 2;
j = i * i++;
```

It's natural to assume that `j` is assigned the value 4. However, the effect of executing the statement is undefined, and `j` could just as well be assigned 6 instead. Here's the scenario: (1) The second operand (the original value of `i`) is fetched, then `i` is incremented. (2) The first operand (the new value of `i`) is fetched. (3) The new and old values of `i` are multiplied, yielding 6. "Fetching" a variable means to retrieve the value of the variable from memory. A later change to the variable won't affect the fetched value, which is typically stored in a special location (known as a ***register***) inside the CPU.

registers ➤ 18.2

Undefined Behavior

According to the C standard, statements such as `c = (b = a + 2) - (a = 1);` and `j = i * i++;` cause ***undefined behavior,*** which is different from implementation-defined behavior (see Section 4.1). When a program ventures into the realm of undefined behavior, all bets are off. The program may behave differently when compiled with different compilers. But that's not the only thing that can happen. The program may not compile in the first place, if it compiles it may not run, and if it does run, it may crash, behave erratically, or produce meaningless results. In other words, undefined behavior should be avoided like the plague.

4.5 Expression Statements

C has the unusual rule that *any* expression can be used as a statement. That is, any expression—regardless of its type or what it computes—can be turned into a statement by appending a semicolon. For example, we could turn the expression `++i` into a statement:

```
++i;
```

When this statement is executed, `i` is first incremented, then the new value of `i` is fetched (as though it were to be used in an enclosing expression). However, since `++i` isn't part of a larger expression, its value is discarded and the next statement executed. (The change to `i` is permanent, of course.)

Q&A

Since its value is discarded, there's little point in using an expression as a statement unless the expression has a side effect. Let's look at three examples. In

the first example, 1 is stored into `i`, then the new value of `i` is fetched but not used:

```
i = 1;
```

In the second example, the value of `i` is fetched but not used; however, `i` is decremented afterwards:

```
i--;
```

In the third example, the value of the expression `i * j - 1` is computed and then discarded:

```
i * j - 1;
```

Since `i` and `j` aren't changed, this statement has no effect and therefore serves no purpose.

A slip of the finger can easily create a "do-nothing" expression statement. For example, instead of entering

```
i = j;
```

we might accidentally type

```
i + j;
```

(This kind of error is more common than you might expect, since the = and + characters usually occupy the same key.) Some compilers can detect meaningless expression statements; you'll get a warning such as *"statement with no effect."*

Q & A

Q: I notice that C has no exponentiation operator. How can I raise a number to a power?

A: Raising an integer to a small positive integer power is best done by repeated multiplication (`i * i * i` is `i` cubed). To raise a number to a noninteger power, call the pow function.

pow function ➤23.3

Q: I want to apply the % operator to a floating-point operand, but my program won't compile. What can I do? [p. 54]

fmod function ➤23.3

A: The % operator requires integer operands. Try the `fmod` function instead.

Q: Why are the rules for using the / and % operators with negative operands so complicated? [p. 54]

A: The rules aren't as complicated as they may first appear. In both C89 and C99, the goal is to ensure that the value of `(a / b) * b + a % b` will always be equal to `a`

(and indeed, both standards guarantee that this is the case, provided that the value of `a / b` is "representable"). The problem is that there are two ways for `a / b` and `a % b` to satisfy this equality if either `a` or `b` is negative, as seen in C89, where either `-9 / 7` is –1 and `-9 % 7` is –2, or `-9 / 7` is –2 and `-9 % 7` is 5. In the first case, `(-9 / 7) * 7 + -9 % 7` has the value $-1 \times 7 + -2 = -9$, and in the second
C99 case, `(-9 / 7) * 7 + -9 % 7` has the value $-2 \times 7 + 5 = -9$. By the time C99 rolled around, most CPUs were designed to truncate the result of division toward zero, so this was written into the standard as the only allowable outcome.

Q: If C has lvalues, does it also have rvalues? [p. 59]

A: Yes, indeed. An *l*value is an expression that can appear on the *left* side of an assignment; an *r*value is an expression that can appear on the *right* side. Thus, an rvalue could be a variable, constant, or more complex expression. In this book, as in the C standard, we'll use the term "expression" instead of "rvalue."

***Q: You said that *v* += *e* isn't equivalent to *v* = *v* + *e* if *v* has a side effect. Can you explain? [p. 60]**

A: Evaluating *v* += *e* causes *v* to be evaluated only once; evaluating *v* = *v* + *e* causes *v* to be evaluated twice. Any side effect caused by evaluating *v* will occur twice in the latter case. In the following example, `i` is incremented once:

```
a[i++] += 2;
```

If we use = instead of +=, here's what the statement will look like:

```
a[i++] = a[i++] + 2;
```

The value of `i` is modified as well as used elsewhere in the statement, so the effect of executing the statement is undefined. It's likely that `i` will be incremented twice, but we can't say with certainty what will happen.

Q: Why does C provide the `++` and `--` operators? Are they faster than other ways of incrementing and decrementing, or they are just more convenient? [p. 61]

A: C inherited `++` and `--` from Ken Thompson's earlier B language. Thompson apparently created these operators because his B compiler could generate a more compact translation for `++i` than for `i = i + 1`. These operators have become a deeply ingrained part of C (in fact, many of C's most famous idioms rely on them). With modern compilers, using `++` and `--` won't make a compiled program any smaller or faster; the continued popularity of these operators stems mostly from their brevity and convenience.

Q: Do `++` and `--` work with `float` variables?

A: Yes; the increment and decrement operations can be applied to floating-point numbers as well as integers. In practice, however, it's fairly rare to increment or decrement a `float` variable.

***Q:** **When I use the postfix version of ++ or --, just when is the increment or decrement performed? [p. 62]**

A: That's an excellent question. Unfortunately, it's also a difficult one to answer. The C standard introduces the concept of "sequence point" and says that "updating the stored value of the operand shall occur between the previous and the next sequence point." There are various kinds of sequence points in C: the end of an expression statement is one example. By the end of an expression statement, all increments and decrements within the statement must have been performed; the next statement can't begin to execute until this condition has been met.

Certain operators that we'll encounter in later chapters (logical *and*, logical *or*, conditional, and comma) also impose sequence points. So do function calls: the arguments in a function call must be fully evaluated before the call can be performed. If an argument happens to be an expression containing a ++ or -- operator, the increment or decrement must occur before the call can take place.

Q: What do you mean when you say that the value of an expression statement is discarded? [p. 65]

A: By definition, an expression represents a value. If `i` has the value 5, for example, then evaluating `i + 1` produces the value 6. Let's turn `i + 1` into a statement by putting a semicolon after it:

```
i + 1;
```

When this statement is executed, the value of `i + 1` is computed. Since we have failed to save this value—or at least use it in some way—it is lost.

Q: But what about statements like `i = 1;`? I don't see what is being discarded.

A: Don't forget that = is an operator in C and produces a value just like any other operator. The assignment

```
i = 1;
```

assigns 1 to `i`. The value of the entire expression is 1, which is discarded. Discarding the expression's value is no great loss, since the reason for writing the statement in the first place was to modify `i`.

Exercises

Section 4.1

1. Show the output produced by each of the following program fragments. Assume that `i`, `j`, and `k` are `int` variables.

(a)
```
i = 5; j = 3;
printf("%d %d", i / j, i % j);
```

(b)
```
i = 2; j = 3;
printf("%d", (i + 10) % j);
```

(c)
```
i = 7; j = 8; k = 9;
printf("%d", (i + 10) % k / j);
```

(d)
```
i = 1; j = 2; k = 3;
printf("%d", (i + 5) % (j + 2) / k);
```

W *2. If `i` and `j` are positive integers, does `(-i)/j` always have the same value as `-(i/j)`? Justify your answer.

3. What is the value of each of the following expressions in C89? (Give all possible values if an expression may have more than one value.)

(a) `8 / 5`
(b) `-8 / 5`
(c) `8 / -5`
(d) `-8 / -5`

4. Repeat Exercise 3 for C99.

5. What is the value of each of the following expressions in C89? (Give all possible values if an expression may have more than one value.)

(a) `8 % 5`
(b) `-8 % 5`
(c) `8 % -5`
(d) `-8 % -5`

6. Repeat Exercise 5 for C99.

7. The algorithm for computing the UPC check digit ends with the following steps:

Subtract 1 from the total.
Compute the remainder when the adjusted total is divided by 10.
Subtract the remainder from 9.

It's tempting to try to simplify the algorithm by using these steps instead:

Compute the remainder when the total is divided by 10.
Subtract the remainder from 10.

Why doesn't this technique work?

8. Would the `upc.c` program still work if the expression `9 - ((total - 1) % 10)` were replaced by `(10 - (total % 10)) % 10`?

Section 4.2

W 9. Show the output produced by each of the following program fragments. Assume that `i`, `j`, and `k` are `int` variables.

(a)
```
i = 7; j = 8;
i *= j + 1;
printf("%d %d", i, j);
```
(b)
```
i = j = k = 1;
i += j += k;
printf("%d %d %d", i, j, k);
```
(c)
```
i = 1; j = 2; k = 3;
i -= j -= k;
printf("%d %d %d", i, j, k);
```
(d)
```
i = 2; j = 1; k = 0;
i *= j *= k;
printf("%d %d %d", i, j, k);
```

10. Show the output produced by each of the following program fragments. Assume that `i` and `j` are `int` variables.

(a)
```
i = 6;
j = i += i;
printf("%d %d", i, j);
```

(b)
```
i = 5;
j = (i -= 2) + 1;
printf("%d %d", i, j);
```

(c)
```
i = 7;
j = 6 + (i = 2.5);
printf("%d %d", i, j);
```

(d)
```
i = 2; j = 8;
j = (i = 6) + (j = 3);
printf("%d %d", i, j);
```

Section 4.3

*11. Show the output produced by each of the following program fragments. Assume that `i`, `j`, and `k` are `int` variables.

(a)
```
i = 1;
printf("%d ", i++ - 1);
printf("%d", i);
```

(b)
```
i = 10; j = 5;
printf("%d ", i++ - ++j);
printf("%d %d", i, j);
```

(c)
```
i = 7; j = 8;
printf("%d ", i++ - --j);
printf("%d %d", i, j);
```

(d)
```
i = 3; j = 4; k = 5;
printf("%d ", i++ - j++ + --k);
printf("%d %d %d", i, j, k);
```

12. Show the output produced by each of the following program fragments. Assume that `i` and `j` are `int` variables.

(a)
```
i = 5;
j = ++i * 3 - 2;
printf("%d %d", i, j);
```

(b)
```
i = 5;
j = 3 - 2 * i++;
printf("%d %d", i, j);
```

(c)
```
i = 7;
j = 3 * i-- + 2;
printf("%d %d", i, j);
```

(d)
```
i = 7;
j = 3 + --i * 2;
printf("%d %d", i, j);
```

W 13. Only one of the expressions `++i` and `i++` is exactly the same as `(i += 1)`: which is it? Justify your answer.

Section 4.4

14. Supply parentheses to show how a C compiler would interpret each of the following expressions.

(a) `a * b - c * d + e`
(b) `a / b % c / d`
(c) `- a - b + c - + d`
(d) `a * - b / c - d`

Section 4.5

15. Give the values of `i` and `j` after each of the following expression statements has been executed. (Assume that `i` has the value 1 initially and `j` has the value 2.)

(a) `i += j;`
(b) `i--;`
(c) `i * j / i;`
(d) `i % ++j;`

Programming Projects

1. Write a program that asks the user to enter a two-digit number, then prints the number with its digits reversed. A session with the program should have the following appearance:

```
Enter a two-digit number: 28
The reversal is: 82
```

Read the number using `%d`, then break it into two digits. *Hint:* If `n` is an integer, then `n % 10` is the last digit in `n` and `n / 10` is `n` with the last digit removed.

W 2. Extend the program in Programming Project 1 to handle *three*-digit numbers.

3. Rewrite the program in Programming Project 2 so that it prints the reversal of a three-digit number without using arithmetic to split the number into digits. *Hint:* See the `upc.c` program of Section 4.1.

4. Write a program that reads an integer entered by the user and displays it in octal (base 8):

```
Enter a number between 0 and 32767: 1953
In octal, your number is: 03641
```

The output should be displayed using five digits, even if fewer digits are sufficient. *Hint:* To convert the number to octal, first divide it by 8; the remainder is the last digit of the octal number (1, in this case). Then divide the original number by 8 and repeat the process to arrive at the next-to-last digit. (`printf` is capable of displaying numbers in base 8, as we'll see in Chapter 7, so there's actually an easier way to write this program.)

5. Rewrite the `upc.c` program of Section 4.1 so that the user enters 11 digits at one time, instead of entering one digit, then five digits, and then another five digits.

```
Enter the first 11 digits of a UPC: 01380015173
Check digit: 5
```

6. European countries use a 13-digit code, known as a European Article Number (EAN) instead of the 12-digit Universal Product Code (UPC) found in North America. Each EAN ends with a check digit, just as a UPC does. The technique for calculating the check digit is also similar:

> Add the second, fourth, sixth, eighth, tenth, and twelfth digits.
> Add the first, third, fifth, seventh, ninth, and eleventh digits.
> Multiply the first sum by 3 and add it to the second sum.

Subtract 1 from the total.
Compute the remainder when the adjusted total is divided by 10.
Subtract the remainder from 9.

For example, consider Güllüoglu Turkish Delight Pistachio & Coconut, which has an EAN of 8691484260008. The first sum is 6 + 1 + 8 + 2 + 0 + 0 = 17, and the second sum is 8 + 9 + 4 + 4 + 6 + 0 = 31. Multiplying the first sum by 3 and adding the second yields 82. Subtracting 1 gives 81. The remainder upon dividing by 10 is 1. When the remainder is subtracted from 9, the result is 8, which matches the last digit of the original code. Your job is to modify the `upc.c` program of Section 4.1 so that it calculates the check digit for an EAN. The user will enter the first 12 digits of the EAN as a single number:

```
Enter the first 12 digits of an EAN: 869148426000
Check digit: 8
```

5 Selection Statements

Programmers are not to be measured by their ingenuity and their logic but by the completeness of their case analysis.

return statement ➤2.2
expression statement ➤4.5

Although C has many operators, it has relatively few statements. We've encountered just two so far: the `return` statement and the expression statement. Most of C's remaining statements fall into three categories, depending on how they affect the order in which statements are executed:

- *Selection statements.* The `if` and `switch` statements allow a program to select a particular execution path from a set of alternatives.
- *Iteration statements.* The `while`, `do`, and `for` statements support iteration (looping).
- *Jump statements.* The `break`, `continue`, and `goto` statements cause an unconditional jump to some other place in the program. (The `return` statement belongs in this category, as well.)

The only other statements in C are the compound statement, which groups several statements into a single statement, and the null statement, which performs no action.

This chapter discusses the selection statements and the compound statement. (Chapter 6 covers the iteration statements, the jump statements, and the null statement.) Before we can write `if` statements, we'll need logical expressions: conditions that `if` statements can test. Section 5.1 explains how logical expressions are built from the relational operators (<, <=, >, and >=), the equality operators (== and !=), and the logical operators (&&, ||, and !). Section 5.2 covers the `if` statement and compound statement, as well as introducing the conditional operator (?:), which can test a condition within an expression. Section 5.3 describes the `switch` statement.

5.1 Logical Expressions

Several of C's statements, including the `if` statement, must test the value of an expression to see if it is "true" or "false." For example, an `if` statement might need to test the expression `i < j`; a true value would indicate that `i` is less than `j`. In many programming languages, an expression such as `i < j` would have a special "Boolean" or "logical" type. Such a type would have only two values, *false* and *true*. In C, however, a comparison such as `i < j` yields an integer: either 0 (false) or 1 (true). With this in mind, let's look at the operators that are used to build logical expressions.

Relational Operators

C's *relational operators* (Table 5.1) correspond to the <, >, ≤, and ≥ operators of mathematics, except that they produce 0 (false) or 1 (true) when used in expressions. For example, the value of `10 < 11` is 1; the value of `11 < 10` is 0.

Table 5.1
Relational Operators

Symbol	*Meaning*
`<`	less than
`>`	greater than
`<=`	less than or equal to
`>=`	greater than or equal to

The relational operators can be used to compare integers and floating-point numbers, with operands of mixed types allowed. Thus, `1 < 2.5` has the value 1, while `5.6 < 4` has the value 0.

The precedence of the relational operators is lower than that of the arithmetic operators; for example, `i + j < k - 1` means `(i + j) < (k - 1)`. The relational operators are left associative.

The expression

```
i < j < k
```

is legal in C, but doesn't have the meaning that you might expect. Since the < operator is left associative, this expression is equivalent to

```
(i < j) < k
```

In other words, the expression first tests whether `i` is less than `j`; the 1 or 0 produced by this comparison is then compared to `k`. The expression does *not* test whether `j` lies between `i` and `k`. (We'll see later in this section that the correct expression would be `i < j && j < k`.)

Equality Operators

Although the relational operators are denoted by the same symbols as in many other programming languages, the ***equality operators*** have a unique appearance (Table 5.2). The "equal to" operator is two adjacent = characters, not one, since a single = character represents the assignment operator. The "not equal to" operator is also two characters: ! and =.

Table 5.2
Equality Operators

Symbol	*Meaning*
==	equal to
!=	not equal to

Like the relational operators, the equality operators are left associative and produce either 0 (false) or 1 (true) as their result. However, the equality operators have *lower* precedence than the relational operators. For example, the expression

```
i < j == j < k
```

is equivalent to

```
(i < j) == (j < k)
```

which is true if `i < j` and `j < k` are both true or both false.

Clever programmers sometimes exploit the fact that the relational and equality operators return integer values. For example, the value of the expression `(i >= j) + (i == j)` is either 0, 1, or 2, depending on whether `i` is less than, greater than, or equal to `j`, respectively. Tricky coding like this generally isn't a good idea, however; it makes programs hard to understand.

Logical Operators

More complicated logical expressions can be built from simpler ones by using the ***logical operators:*** *and*, *or*, and *not* (Table 5.3). The ! operator is unary, while && and || are binary.

Table 5.3
Logical Operators

Symbol	*Meaning*
!	logical negation
&&	logical *and*
||	logical *or*

The logical operators produce either 0 or 1 as their result. Often, the operands will have values of 0 or 1, but this isn't a requirement; the logical operators treat any nonzero operand as a true value and any zero operand as a false value.

The logical operators behave as follows:

- !*expr* has the value 1 if *expr* has the value 0.
- *expr1* && *expr2* has the value 1 if the values of *expr1* and *expr2* are both nonzero.

- *expr1* | | *expr2* has the value 1 if either *expr1* or *expr2* (or both) has a nonzero value.

In all other cases, these operators produce the value 0.

Both && and | | perform "short-circuit" evaluation of their operands. That is, these operators first evaluate the left operand, then the right operand. If the value of the expression can be deduced from the value of the left operand alone, then the right operand isn't evaluated. Consider the following expression:

```
(i != 0) && (j / i > 0)
```

To find the value of this expression, we must first evaluate `(i != 0)`. If `i` isn't equal to 0, then we'll need to evaluate `(j / i > 0)` to determine whether the entire expression is true or false. However, if `i` is equal to 0, then the entire expression must be false, so there's no need to evaluate `(j / i > 0)`. The advantage of short-circuit evaluation is apparent—without it, evaluating the expression would have caused a division by zero.

Be wary of side effects in logical expressions. Thanks to the short-circuit nature of the && and | | operators, side effects in operands may not always occur. Consider the following expression:

```
i > 0 && ++j > 0
```

Although `j` is apparently incremented as a side effect of evaluating the expression, that isn't always the case. If `i > 0` is false, then `++j > 0` is not evaluated, so `j` isn't incremented. The problem can be fixed by changing the condition to `++j > 0 && i > 0` or, even better, by incrementing `j` separately.

The ! operator has the same precedence as the unary plus and minus operators. The precedence of && and | | is lower than that of the relational and equality operators; for example, `i < j && k == m` means `(i < j) && (k == m)`. The ! operator is right associative; && and | | are left associative.

5.2 The `if` Statement

The `if` statement allows a program to choose between two alternatives by testing the value of an expression. In its simplest form, the `if` statement has the form

`if` statement

`if` (*expression*) *statement*

Notice that the parentheses around the expression are mandatory; they're part of the `if` statement, not part of the expression. Also note that the word `then` doesn't come after the parentheses, as it would in some programming languages.

When an `if` statement is executed, the expression in the parentheses is evaluated; if the value of the expression is nonzero—which C interprets as true—the statement after the parentheses is executed. Here's an example:

```
if (line_num == MAX_LINES)
  line_num = 0;
```

The statement `line_num = 0;` is executed if the condition `line_num == MAX_LINES` is true (has a nonzero value).

Don't confuse == (equality) with = (assignment). The statement

```
if (i == 0) ...
```

tests whether `i` is equal to 0. However, the statement

```
if (i = 0) ...
```

assigns 0 to `i`, then tests whether the *result* is nonzero. In this case, the test always fails.

Confusing == with = is perhaps the most common C programming error, probably because = means "is equal to" in mathematics (and in certain programming languages). Some compilers issue a warning if they notice = where == would normally appear.

Q&A

Often the expression in an `if` statement will test whether a variable falls within a range of values. To test whether $0 \le i < n$, for example, we'd write

idiom

```
if (0 <= i && i < n) ...
```

To test the *opposite* condition (`i` is outside the range), we'd write

idiom

```
if (i < 0 || i >= n) ...
```

Note the use of the || operator instead of the && operator.

Compound Statements

In our `if` statement template, notice that *statement* is singular, not plural:

`if` (*expression*) *statement*

What if we want an `if` statement to control *two* or more statements? That's where the ***compound statement*** comes in. A compound statement has the form

compound statement

{ *statements* }

By putting braces around a group of statements, we can force the compiler to treat it as a single statement.

Here's an example of a compound statement:

```
{ line_num = 0; page_num++; }
```

For clarity, I'll usually put a compound statement on several lines, with one statement per line:

```
{
  line_num = 0;
  page_num++;
}
```

Notice that each inner statement still ends with a semicolon, but the compound statement itself does not.

Here's what a compound statement would look like when used inside an `if` statement:

```
if (line_num == MAX_LINES) {
  line_num = 0;
  page_num++;
}
```

Compound statements are also common in loops and other places where the syntax of C requires a single statement, but we want more than one.

The `else` Clause

An `if` statement may have an `else` clause:

`if` statement with `else` clause

`if` (*expression*) *statement* `else` *statement*

The statement that follows the word `else` is executed if the expression in parentheses has the value 0.

Here's an example of an `if` statement with an `else` clause:

```
if (i > j)
  max = i;
else
  max = j;
```

Notice that both "inner" statements end with a semicolon.

When an `if` statement contains an `else` clause, a layout issue arises: where should the `else` be placed? Many C programmers align it with the `if` at the beginning of the statement, as in the previous example. The inner statements are usually indented, but if they're short they can be put on the same line as the `if` and `else`:

```
if (i > j) max = i;
else max = j;
```

There are no restrictions on what kind of statements can appear inside an `if` statement. In fact, it's not unusual for `if` statements to be nested inside other `if` statements. Consider the following `if` statement, which finds the largest of the numbers stored in `i`, `j`, and `k` and stores that value in `max`:

```
if (i > j)
  if (i > k)
    max = i;
  else
    max = k;
else
  if (j > k)
    max = j;
  else
    max = k;
```

`if` statements can be nested to any depth. Notice how aligning each `else` with the matching `if` makes the nesting easier to see. If you still find the nesting confusing, don't hesitate to add braces:

```
if (i > j) {
  if (i > k)
    max = i;
  else
    max = k;
} else {
  if (j > k)
    max = j;
  else
    max = k;
}
```

Adding braces to statements—even when they're not necessary—is like using parentheses in expressions: both techniques help make a program more readable while at the same time avoiding the possibility that the compiler won't understand the program the way we thought it did.

Some programmers use as many braces as possible inside `if` statements (and iteration statements as well). A programmer who adopts this convention would include a pair of braces for every `if` clause and every `else` clause:

```
if (i > j) {
  if (i > k) {
    max = i;
  } else {
    max = k;
  }
} else {
  if (j > k) {
    max = j;
  } else {
    max = k;
  }
}
```

Using braces even when they're not required has two advantages. First, the program becomes easier to modify, because more statements can easily be added to any `if` or `else` clause. Second, it helps avoid errors that can result from forgetting to use braces when adding statements to an `if` or `else` clause.

Cascaded `if` Statements

We'll often need to test a series of conditions, stopping as soon as one of them is true. A "cascaded" `if` statement is often the best way to write such a series of tests. For example, the following cascaded `if` statement tests whether `n` is less than 0, equal to 0, or greater than 0:

```
if (n < 0)
  printf("n is less than 0\n");
else
  if (n == 0)
    printf("n is equal to 0\n");
  else
    printf("n is greater than 0\n");
```

Although the second `if` statement is nested inside the first, C programmers don't usually indent it. Instead, they align each `else` with the original `if`:

```
if (n < 0)
  printf("n is less than 0\n");
else if (n == 0)
  printf("n is equal to 0\n");
else
  printf("n is greater than 0\n");
```

This arrangement gives the cascaded `if` a distinctive appearance:

```
if ( expression )
  statement
else if ( expression )
  statement
...
else if ( expression )
  statement
else
  statement
```

The last two lines (`else` *statement*) aren't always present, of course. This way of indenting the cascaded `if` statement avoids the problem of excessive indentation when the number of tests is large. Moreover, it assures the reader that the statement is nothing more than a series of tests.

Keep in mind that a cascaded `if` statement isn't some new kind of statement; it's just an ordinary `if` statement that happens to have another `if` statement as its `else` clause (and *that* `if` statement has another `if` statement as its `else` clause, *ad infinitum*).

PROGRAM

Calculating a Broker's Commission

When stocks are sold or purchased through a broker, the broker's commission is often computed using a sliding scale that depends upon the value of the stocks traded. Let's say that a broker charges the amounts shown in the following table:

Transaction size	*Commission rate*
Under $2,500	$30 + 1.7%
$2,500–$6,250	$56 + 0.66%
$6,250–$20,000	$76 + 0.34%
$20,000–$50,000	$100 + 0.22%
$50,000–$500,000	$155 + 0.11%
Over $500,000	$255 + 0.09%

The minimum charge is $39. Our next program asks the user to enter the amount of the trade, then displays the amount of the commission:

```
Enter value of trade: 30000
Commission: $166.00
```

The heart of the program is a cascaded `if` statement that determines which range the trade falls into.

broker.c

```
/* Calculates a broker's commission */

#include <stdio.h>

int main(void)
{
  float commission, value;

  printf("Enter value of trade: ");
  scanf("%f", &value);

  if (value < 2500.00f)
    commission = 30.00f + .017f * value;
  else if (value < 6250.00f)
    commission = 56.00f + .0066f * value;
  else if (value < 20000.00f)
    commission = 76.00f + .0034f * value;
  else if (value < 50000.00f)
    commission = 100.00f + .0022f * value;
  else if (value < 500000.00f)
    commission = 155.00f + .0011f * value;
  else
    commission = 255.00f + .0009f * value;

  if (commission < 39.00f)
    commission = 39.00f;

  printf("Commission: $%.2f\n", commission);

  return 0;
}
```

The cascaded `if` statement could have been written this way instead (the changes are indicated in **bold**):

```
if (value < 2500.00f)
  commission = 30.00f + .017f * value;
else if (value >= 2500.00f && value < 6250.00f)
  commission = 56.00f + .0066f * value;
else if (value >= 6250.00f && value < 20000.00f)
  commission = 76.00f + .0034f * value;
...
```

Although the program will still work, the added conditions aren't necessary. For example, the first `if` clause tests whether `value` is less than 2500 and, if so, computes the commission. When we reach the second `if` test (`value >= 2500.00f && value < 6250.00f`), we know that `value` can't be less than 2500 and therefore must be greater than or equal to 2500. The condition `value >= 2500.00f` will always be true, so there's no point in checking it.

The "Dangling `else`" Problem

When `if` statements are nested, we've got to watch out for the notorious "dangling `else`" problem. Consider the following example:

```
if (y != 0)
  if (x != 0)
    result = x / y;
else
  printf("Error: y is equal to 0\n");
```

To which `if` statement does the `else` clause belong? The indentation suggests that it belongs to the outer `if` statement. However, C follows the rule that an `else` clause belongs to the nearest `if` statement that hasn't already been paired with an `else`. In this example, the `else` clause actually belongs to the inner `if` statement, so a correctly indented version would look like this:

```
if (y != 0)
  if (x != 0)
    result = x / y;
  else
    printf("Error: y is equal to 0\n");
```

To make the `else` clause part of the outer `if` statement, we can enclose the inner `if` statement in braces:

```
if (y != 0) {
  if (x != 0)
    result = x / y;
} else
    printf("Error: y is equal to 0\n");
```

This example illustrates the value of braces; if we'd used them in the original `if` statement, we wouldn't have gotten into this situation in the first place.

Conditional Expressions

C's `if` statement allows a program to perform one of two actions depending on the value of a condition. C also provides an *operator* that allows an expression to produce one of two *values* depending on the value of a condition.

The ***conditional operator*** consists of two symbols (`?` and `:`), which must be used together in the following way:

conditional expression

expr1 ? *expr2* : *expr3*

expr1, *expr2*, and *expr3* can be expressions of any type. The resulting expression is said to be a ***conditional expression***. The conditional operator is unique among C operators in that it requires *three* operands instead of one or two. For this reason, it is often referred to as a ***ternary*** operator.

The conditional expression *expr1* ? *expr2* : *expr3* should be read "if *expr1* then *expr2* else *expr3*." The expression is evaluated in stages: *expr1* is evaluated first; if its value isn't zero, then *expr2* is evaluated, and its value is the value of the entire conditional expression. If the value of *expr1* is zero, then the value of *expr3* is the value of the conditional.

The following example illustrates the conditional operator:

```
int i, j, k;
i = 1;
j = 2;
k = i > j ? i : j;              /* k is now 2 */
k = (i >= 0 ? i : 0) + j;       /* k is now 3 */
```

The conditional expression `i > j ? i : j` in the first assignment to `k` returns the value of either `i` or `j`, depending on which one is larger. Since `i` has the value 1 and `j` has the value 2, the `i > j` comparison fails, and the value of the conditional is 2, which is assigned to `k`. In the second assignment to `k`, the `i >= 0` comparison succeeds; the conditional expression `(i >= 0 ? i : 0)` has the value 1, which is then added to `j` to produce 3. The parentheses are necessary, by the way; the precedence of the conditional operator is less than that of the other operators we've discussed so far, with the exception of the assignment operators.

Conditional expressions tend to make programs shorter but harder to understand, so it's probably best to avoid them. There are, however, a few places in which they're tempting; one is the `return` statement. Instead of writing

```
if (i > j)
  return i;
else
  return j;
```

many programmers would write

```
return i > j ? i : j;
```

Calls of `printf` can sometimes benefit from condition expressions. Instead of

```
if (i > j)
  printf("%d\n", i);
else
  printf("%d\n", j);
```

we could simply write

```
printf("%d\n", i > j ? i : j);
```

macro definitions ➤ 14.3 Conditional expressions are also common in certain kinds of macro definitions.

Boolean Values in C89

For many years, the C language lacked a proper Boolean type, and there is none defined in the C89 standard. This omission is a minor annoyance, since many programs need variables that can store either *false* or *true*. One way to work around this limitation of C89 is to declare an `int` variable and then assign it either 0 or 1:

```
int flag;

flag = 0;
...
flag = 1;
```

Although this scheme works, it doesn't contribute much to program readability. It's not obvious that `flag` is to be assigned only Boolean values and that 0 and 1 represent false and true.

To make programs more understandable, C89 programmers often define macros with names such as `TRUE` and `FALSE`:

```
#define TRUE 1
#define FALSE 0
```

Assignments to `flag` now have a more natural appearance:

```
flag = FALSE;
...
flag = TRUE;
```

To test whether `flag` is true, we can write

```
if (flag == TRUE) ...
```

or just

```
if (flag) ...
```

The latter form is better, not only because it's more concise, but also because it will still work correctly if `flag` has a value other than 0 or 1.

To test whether `flag` is false, we can write

```
if (flag == FALSE) ...
```

or

```
if (!flag) ...
```

Carrying this idea one step further, we might even define a macro that can be used as a type:

```
#define BOOL int
```

BOOL can take the place of `int` when declaring Boolean variables:

```
BOOL flag;
```

It's now clear that `flag` isn't an ordinary integer variable, but instead represents a Boolean condition. (The compiler still treats `flag` as an `int` variable, of course.) In later chapters, we'll discover better ways to set up a Boolean type in C89 by using type definitions and enumerations.

type definitions ➤7.5
enumerations ➤16.5

C99 Boolean Values in C99

The longstanding lack of a Boolean type has been remedied in C99, which provides the `_Bool` type. In this version of C, a Boolean variable can be declared by writing

Q&A

```
_Bool flag;
```

`_Bool` is an integer type (more precisely, an *unsigned* integer type), so a `_Bool` variable is really just an integer variable in disguise. Unlike an ordinary integer variable, however, a `_Bool` variable can only be assigned 0 or 1. In general, attempting to store a nonzero value into a `_Bool` variable will cause the variable to be assigned 1:

unsigned integer types ➤7.1

```
flag = 5;    /* flag is assigned 1 */
```

It's legal (although not advisable) to perform arithmetic on `_Bool` variables; it's also legal to print a `_Bool` variable (either 0 or 1 will be displayed). And, of course, a `_Bool` variable can be tested in an `if` statement:

```
if (flag)    /* tests whether flag is 1 */
  ...
```

In addition to defining the `_Bool` type, C99 also provides a new header, `<stdbool.h>`, that makes it easier to work with Boolean values. This header provides a macro, `bool`, that stands for `_Bool`. If `<stdbool.h>` is included, we can write

<stdbool.h> header ➤21.5

```
bool flag;    /* same as _Bool flag; */
```

The `<stdbool.h>` header also supplies macros named `true` and `false`, which stand for 1 and 0, respectively, making it possible to write

```
flag = false;
...
flag = true;
```

Because the `<stdbool.h>` header is so handy, I'll use it in subsequent programs whenever Boolean variables are needed.

5.3 The `switch` Statement

In everyday programming, we'll often need to compare an expression against a series of values to see which one it currently matches. We saw in Section 5.2 that a cascaded `if` statement can be used for this purpose. For example, the following cascaded `if` statement prints the English word that corresponds to a numerical grade:

```
if (grade == 4)
  printf("Excellent");
else if (grade == 3)
  printf("Good");
else if (grade == 2)
  printf("Average");
else if (grade == 1)
  printf("Poor");
else if (grade == 0)
  printf("Failing");
else
  printf("Illegal grade");
```

As an alternative to this kind of cascaded `if` statement, C provides the `switch` statement. The following `switch` is equivalent to our cascaded `if`:

```
switch (grade) {
  case 4:  printf("Excellent");
           break;
  case 3:  printf("Good");
           break;
  case 2:  printf("Average");
           break;
  case 1:  printf("Poor");
           break;
  case 0:  printf("Failing");
           break;
  default: printf("Illegal grade");
           break;
}
```

When this statement is executed, the value of the variable `grade` is tested against 4, 3, 2, 1, and 0. If it matches 4, for example, the message `Excellent` is printed, then the `break` statement transfers control to the statement following the `switch`. If the value of `grade` doesn't match any of the choices listed, the `default` case applies, and the message `Illegal grade` is printed.

`break` statement ➤6.4

A switch statement is often easier to read than a cascaded if statement. Moreover, switch statements are often faster than if statements, especially when there are more than a handful of cases.

Q&A

In its most common form, the switch statement has the form

switch statement

```
switch ( expression ) {
  case constant-expression : statements
  ...
  case constant-expression : statements
  default : statements
}
```

The switch statement is fairly complex; let's look at its components one by one:

characters ➤7.3

- *Controlling expression.* The word switch must be followed by an integer expression in parentheses. Characters are treated as integers in C and thus can be tested in switch statements. Floating-point numbers and strings don't qualify, however.
- *Case labels.* Each case begins with a label of the form

  case *constant-expression* :

  A *constant expression* is much like an ordinary expression except that it can't contain variables or function calls. Thus, 5 is a constant expression, and 5 + 10 is a constant expression, but n + 10 isn't a constant expression (unless n is a macro that represents a constant). The constant expression in a case label must evaluate to an integer (characters are also acceptable).
- *Statements.* After each case label comes any number of statements. No braces are required around the statements. (Enjoy it—this is one of the few places in C where braces aren't required.) The last statement in each group is normally break.

Duplicate case labels aren't allowed. The order of the cases doesn't matter; in particular, the default case doesn't need to come last.

Only one constant expression may follow the word case; however, several case labels may precede the same group of statements:

```
switch (grade) {
  case 4:
  case 3:
  case 2:
  case 1:  printf("Passing");
           break;
  case 0:  printf("Failing");
           break;
  default: printf("Illegal grade");
           break;
}
```

To save space, programmers sometimes put several case labels on the same line:

```
switch (grade) {
  case 4: case 3: case 2: case 1:
          printf("Passing");
          break;
  case 0:  printf("Failing");
          break;
  default: printf("Illegal grade");
          break;
}
```

Unfortunately, there's no way to write a case label that specifies a range of values, as there is in some programming languages.

A `switch` statement isn't required to have a `default` case. If `default` is missing and the value of the controlling expression doesn't match any of the case labels, control simply passes to the next statement after the `switch`.

The Role of the `break` Statement

Now, let's take a closer look at the mysterious `break` statement. As we've seen, executing a `break` statement causes the program to "break" out of the `switch` statement; execution continues at the next statement after the `switch`.

The reason that we need `break` has to do with the fact that the `switch` statement is really a form of "computed jump." When the controlling expression is evaluated, control jumps to the case label matching the value of the `switch` expression. A case label is nothing more than a marker indicating a position within the `switch`. When the last statement in the case has been executed, control "falls through" to the first statement in the following case; the case label for the next case is ignored. Without `break` (or some other jump statement), control will flow from one case into the next. Consider the following `switch` statement:

```
switch (grade) {
  case 4:  printf("Excellent");
  case 3:  printf("Good");
  case 2:  printf("Average");
  case 1:  printf("Poor");
  case 0:  printf("Failing");
  default: printf("Illegal grade");
}
```

If the value of `grade` is 3, the message printed is

```
GoodAveragePoorFailingIllegal grade
```

Forgetting to use `break` is a common error. Although omitting `break` is sometimes done intentionally to allow several cases to share code, it's usually just an oversight.

Since deliberately falling through from one case into the next is rare, it's a good idea to point out any deliberate omission of `break`:

```
switch (grade) {
  case 4: case 3: case 2: case 1:
          num_passing++;
          /* FALL THROUGH */
  case 0:  total_grades++;
          break;
}
```

Without the comment, someone might later fix the "error" by adding an unwanted `break` statement.

Although the last case in a `switch` statement never needs a `break` statement, it's common practice to put one there anyway to guard against a "missing `break`" problem if cases should later be added.

PROGRAM

Printing a Date in Legal Form

Contracts and other legal documents are often dated in the following way:

Dated this _____ *day of* _____, *20*__.

Let's write a program that displays dates in this form. We'll have the user enter the date in month/day/year form, then we'll display the date in "legal" form:

```
Enter date (mm/dd/yy): 7/19/14
Dated this 19th day of July, 2014.
```

We can get `printf` to do most of the formatting. However, we're left with two problems: how to add "th" (or "st" or "nd" or "rd") to the day, and how to print the month as a word instead of a number. Fortunately, the `switch` statement is ideal for both situations; we'll have one `switch` print the day suffix and another print the month name.

date.c

```
/* Prints a date in legal form */

#include <stdio.h>

int main(void)
{
int month, day, year;

printf("Enter date (mm/dd/yy): ");
scanf("%d /%d /%d", &month, &day, &year);

printf("Dated this %d", day);
switch (day) {
  case 1: case 21: case 31:
    printf("st"); break;
  case 2: case 22:
    printf("nd"); break;
```

```
    case 3: case 23:
      printf("rd"); break;
    default: printf("th"); break;
  }
  printf(" day of ");

  switch (month) {
    case 1:  printf("January");   break;
    case 2:  printf("February");  break;
    case 3:  printf("March");     break;
    case 4:  printf("April");     break;
    case 5:  printf("May");       break;
    case 6:  printf("June");      break;
    case 7:  printf("July");      break;
    case 8:  printf("August");    break;
    case 9:  printf("September"); break;
    case 10: printf("October");   break;
    case 11: printf("November");  break;
    case 12: printf("December");  break;
  }

  printf(", 20%.2d.\n", year);
  return 0;
}
```

Note the use of `%.2d` to display the last two digits of the year. If we had used `%d` instead, single-digit years would be displayed incorrectly (2005 would be printed as `205`).

Q & A

Q: **My compiler doesn't give a warning when I use = instead of ==. Is there some way to force the compiler to notice the problem? [p. 77]**

A: Here's a trick that some programmers use: instead of writing

```
if (i == 0) ...
```

they habitually write

```
if (0 == i) ...
```

Now suppose that the == operator is accidentally written as =:

```
if (0 = i) ...
```

The compiler will produce an error message, since it's not possible to assign a value to 0. I don't use this trick, because I think it makes programs look unnatural. Also, it can be used only when one of the operands in the test condition isn't an lvalue.

Fortunately, many compilers are capable of checking for suspect uses of the = operator in `if` conditions. The GCC compiler, for example, will perform this

check if the `-Wparentheses` option is used or if `-Wall` (all warnings) is selected. GCC allows the programmer to suppress the warning in a particular case by enclosing the `if` condition in a second set of parentheses:

```
if ((i = j)) ...
```

Q: C books seem to use several different styles of indentation and brace placement for compound statements. Which style is best?

A: According to *The New Hacker's Dictionary* (Cambridge, Mass.: MIT Press, 1996), there are four common styles of indentation and brace placement:

- The *K&R style*, used in Kernighan and Ritchie's *The C Programming Language*, is the one I've chosen for the programs in this book. In the K&R style, the left brace appears at the end of a line:

  ```
  if (line_num == MAX_LINES) {
    line_num = 0;
    page_num++;
  }
  ```

  The K&R style keeps programs compact by not putting the left brace on a line by itself. A disadvantage: the left brace can be hard to find. (I don't consider this a problem, since the indentation of the inner statements makes it clear where the left brace should be.) The K&R style is the one most often used in Java, by the way.

- The *Allman style*, named after Eric Allman (the author of `sendmail` and other UNIX utilities), puts the left brace on a separate line:

  ```
  if (line_num == MAX_LINES)
  {
    line_num = 0;
    page_num++;
  }
  ```

  This style makes it easy to check that braces come in matching pairs.

- The *Whitesmiths style*, popularized by the Whitesmiths C compiler, dictates that braces be indented:

  ```
  if (line_num == MAX_LINES)
    {
    line_num = 0;
    page_num++;
    }
  ```

- The *GNU style*, used in software developed by the GNU Project, indents the braces, then further indents the inner statements:

  ```
  if (line_num == MAX_LINES)
    {
      line_num = 0;
      page_num++;
    }
  ```

Which style you use is mainly a matter of taste; there's no proof that one style is clearly better than the others. In any event, choosing the right style is less important than applying it consistently.

Q: If `i` is an `int` variable and `f` is a `float` variable, what is the type of the conditional expression `(i > 0 ? i : f)`?

A: When `int` and `float` values are mixed in a conditional expression, as they are here, the expression has type `float`. If `i > 0` is true, the value of the expression will be the value of `i` after conversion to `float` type.

Q: Why doesn't C99 have a better name for its Boolean type? [p. 85]

A: `_Bool` isn't a very elegant name, is it? More common names, such as `bool` or `boolean`, weren't chosen because existing C programs might already define these names, causing older code not to compile.

C99

Q: OK, so why wouldn't the name `_Bool` break older programs as well?

A: The C89 standard specifies that names beginning with an underscore followed by an uppercase letter are reserved for future use and should not be used by programmers.

***Q: The template given for the `switch` statement described it as the "most common form." Are there other forms? [p. 87]**

A: The `switch` statement is a bit more general than described in this chapter, although the description given here is general enough for virtually all programs.

labels ➤6.4

For example, a `switch` statement can contain labels that aren't preceded by the word `case`, which leads to an amusing (?) trap. Suppose that we accidentally misspell the word `default`:

```
switch (...) {
  ...
  defualt: ...
}
```

The compiler may not detect the error, since it assumes that `defualt` is an ordinary label.

Q: I've seen several methods of indenting the `switch` statement. Which way is best?

A: There are at least two common methods. One is to put the statements in each case *after* the case label:

```
switch (coin) {
  case 1:  printf("Cent");
           break;
  case 5:  printf("Nickel");
           break;
  case 10: printf("Dime");
           break;
```

```
  case 25: printf("Quarter");
           break;
}
```

If each case consists of a single action (a call of `printf`, in this example), the `break` statement could even go on the same line as the action:

```
switch (coin) {
  case 1:  printf("Cent"); break;
  case 5:  printf("Nickel"); break;
  case 10: printf("Dime"); break;
  case 25: printf("Quarter"); break;
}
```

The other method is to put the statements *under* the case label, indenting the statements to make the case label stand out:

```
switch (coin) {
  case 1:
    printf("Cent");
    break;
  case 5:
    printf("Nickel");
    break;
  case 10:
    printf("Dime");
    break;
  case 25:
    printf("Quarter");
    break;
}
```

In one variation of this scheme, each case label is aligned under the word `switch`.

The first method is fine when the statements in each case are short and there are relatively few of them. The second method is better for large `switch` statements in which the statements in each case are complex and/or numerous.

Exercises

Section 5.1

1. The following program fragments illustrate the relational and equality operators. Show the output produced by each, assuming that `i`, `j`, and `k` are `int` variables.

(a)
```
i = 2; j = 3;
k = i * j == 6;
printf("%d", k);
```

(b)
```
i = 5; j = 10; k = 1;
printf("%d", k > i < j);
```

(c)
```
i = 3; j = 2; k = 1;
printf("%d", i < j == j < k);
```

(d)
```
i = 3; j = 4; k = 5;
printf("%d", i % j + i < k);
```

W 2. The following program fragments illustrate the logical operators. Show the output produced by each, assuming that `i`, `j`, and `k` are `int` variables.

(a)
```
i = 10; j = 5;
printf("%d", !i < j);
```
(b)
```
i = 2; j = 1;
printf("%d", !!i + !j);
```
(c)
```
i = 5; j = 0; k = -5;
printf("%d", i && j || k);
```
(d)
```
i = 1; j = 2; k = 3;
printf("%d", i < j || k);
```

*3. The following program fragments illustrate the short-circuit behavior of logical expressions. Show the output produced by each, assuming that `i`, `j`, and `k` are `int` variables.

(a)
```
i = 3; j = 4; k = 5;
printf("%d ", i < j || ++j < k);
printf("%d %d %d", i, j, k);
```
(b)
```
i = 7; j = 8; k = 9;
printf("%d ", i - 7 && j++ < k);
printf("%d %d %d", i, j, k);
```
(c)
```
i = 7; j = 8; k = 9;
printf("%d ", (i = j) || (j = k));
printf("%d %d %d", i, j, k);
```
(d)
```
i = 1; j = 1; k = 1;
printf("%d ", ++i || ++j && ++k);
printf("%d %d %d", i, j, k);
```

W *4. Write a single expression whose value is either –1, 0, or +1, depending on whether `i` is less than, equal to, or greater than `j`, respectively.

Section 5.2

*5. Is the following `if` statement legal?

```
if (n >= 1 <= 10)
  printf("n is between 1 and 10\n");
```

If so, what does it do when `n` is equal to 0?

W *6. Is the following `if` statement legal?

```
if (n == 1-10)
  printf("n is between 1 and 10\n");
```

If so, what does it do when `n` is equal to 5?

7. What does the following statement print if `i` has the value 17? What does it print if `i` has the value –17?

```
printf("%d\n", i >= 0 ? i : -i);
```

8. The following `if` statement is unnecessarily complicated. Simplify it as much as possible. (*Hint:* The entire statement can be replaced by a single assignment.)

```
if (age >= 13)
  if (age <= 19)
    teenager = true;
  else
    teenager = false;
else if (age < 13)
  teenager = false;
```

9. Are the following `if` statements equivalent? If not, why not?

```
if (score >= 90)                if (score < 60)
  printf("A");                    printf("F");
else if (score >= 80)           else if (score < 70)
  printf("B");                    printf("D");
else if (score >= 70)           else if (score < 80)
  printf("C");                    printf("C");
else if (score >= 60)           else if (score < 90)
  printf("D");                    printf("B");
else                            else
  printf("F");                    printf("A");
```

Section 5.3

W*10. What output does the following program fragment produce? (Assume that `i` is an integer variable.)

```
i = 1;
switch (i % 3) {
  case 0: printf("zero");
  case 1: printf("one");
  case 2: printf("two");
}
```

11. The following table shows telephone area codes in the state of Georgia along with the largest city in each area:

Area code	*Major city*
229	Albany
404	Atlanta
470	Atlanta
478	Macon
678	Atlanta
706	Columbus
762	Columbus
770	Atlanta
912	Savannah

Write a `switch` statement whose controlling expression is the variable `area_code`. If the value of `area_code` is in the table, the `switch` statement will print the corresponding city name. Otherwise, the `switch` statement will display the message `"Area code not recognized"`. Use the techniques discussed in Section 5.3 to make the `switch` statement as simple as possible.

Programming Projects

1. Write a program that calculates how many digits a number contains:

```
Enter a number: 374
The number 374 has 3 digits
```

You may assume that the number has no more than four digits. *Hint:* Use `if` statements to test the number. For example, if the number is between 0 and 9, it has one digit. If the number is between 10 and 99, it has two digits.

W 2. Write a program that asks the user for a 24-hour time, then displays the time in 12-hour form:

```
Enter a 24-hour time: 21:11
Equivalent 12-hour time: 9:11 PM
```

Be careful not to display 12:00 as 0:00.

3. Modify the `broker.c` program of Section 5.2 by making both of the following changes:

(a) Ask the user to enter the number of shares and the price per share, instead of the value of the trade.

(b) Add statements that compute the commission charged by a rival broker ($33 plus 3¢ per share for fewer than 2000 shares; $33 plus 2¢ per share for 2000 shares or more). Display the rival's commission as well as the commission charged by the original broker.

W 4. Here's a simplified version of the Beaufort scale, which is used to estimate wind force:

Speed (knots)	*Description*
Less than 1	Calm
1–3	Light air
4–27	Breeze
28–47	Gale
48–63	Storm
Above 63	Hurricane

Write a program that asks the user to enter a wind speed (in knots), then displays the corresponding description.

5. In one state, single residents are subject to the following income tax:

Income	*Amount of tax*	
Not over $750	1% of income	
$750–$2,250	$7.50	plus 2% of amount over $750
$2,250–$3,750	$37.50	plus 3% of amount over $2,250
$3,750–$5,250	$82.50	plus 4% of amount over $3,750
$5,250–$7,000	$142.50	plus 5% of amount over $5,250
Over $7,000	$230.00	plus 6% of amount over $7,000

Write a program that asks the user to enter the amount of taxable income, then displays the tax due.

W 6. Modify the `upc.c` program of Section 4.1 so that it checks whether a UPC is valid. After the user enters a UPC, the program will display either `VALID` or `NOT VALID`.

7. Write a program that finds the largest and smallest of four integers entered by the user:

```
Enter four integers: 21 43 10 35
Largest: 43
Smallest: 10
```

Use as few `if` statements as possible. *Hint:* Four `if` statements are sufficient.

8. The following table shows the daily flights from one city to another:

Departure time	*Arrival time*
8:00 a.m.	10:16 a.m.
9:43 a.m.	11:52 a.m.
11:19 a.m.	1:31 p.m.
12:47 p.m.	3:00 p.m.

2:00 p.m.	4:08 p.m.
3:45 p.m.	5:55 p.m.
7:00 p.m.	9:20 p.m.
9:45 p.m.	11:58 p.m.

Write a program that asks user to enter a time (expressed in hours and minutes, using the 24-hour clock). The program then displays the departure and arrival times for the flight whose departure time is closest to that entered by the user:

```
Enter a 24-hour time: 13:15
Closest departure time is 12:47 p.m., arriving at 3:00 p.m.
```

Hint: Convert the input into a time expressed in minutes since midnight, and compare it to the departure times, also expressed in minutes since midnight. For example, 13:15 is 13 × 60 + 15 = 795 minutes since midnight, which is closer to 12:47 p.m. (767 minutes since midnight) than to any of the other departure times.

9. Write a program that prompts the user to enter two dates and then indicates which date comes earlier on the calendar:

```
Enter first date (mm/dd/yy): 3/6/08
Enter second date (mm/dd/yy): 5/17/07
5/17/07 is earlier than 3/6/08
```

W 10. Using the `switch` statement, write a program that converts a numerical grade into a letter grade:

```
Enter numerical grade: 84
Letter grade: B
```

Use the following grading scale: A = 90–100, B = 80–89, C = 70–79, D = 60–69, F = 0–59. Print an error message if the grade is larger than 100 or less than 0. *Hint:* Break the grade into two digits, then use a `switch` statement to test the ten's digit.

11. Write a program that asks the user for a two-digit number, then prints the English word for the number:

```
Enter a two-digit number: 45
You entered the number forty-five.
```

Hint: Break the number into two digits. Use one `switch` statement to print the word for the first digit ("twenty," "thirty," and so forth). Use a second `switch` statement to print the word for the second digit. Don't forget that the numbers between 11 and 19 require special treatment.

6 Loops

A program without a loop and a structured variable isn't worth writing.

Chapter 5 covered C's selection statements, `if` and `switch`. This chapter introduces C's iteration statements, which allow us to set up loops.

A *loop* is a statement whose job is to repeatedly execute some other statement (the *loop body*). In C, every loop has a *controlling expression*. Each time the loop body is executed (an *iteration* of the loop), the controlling expression is evaluated; if the expression is true—has a value that's not zero—the loop continues to execute.

C provides three iteration statements: `while`, `do`, and `for`, which are covered in Sections 6.1, 6.2, and 6.3, respectively. The `while` statement is used for loops whose controlling expression is tested *before* the loop body is executed. The `do` statement is used if the expression is tested *after* the loop body is executed. The `for` statement is convenient for loops that increment or decrement a counting variable. Section 6.3 also introduces the comma operator, which is used primarily in `for` statements.

The last two sections of this chapter are devoted to C features that are used in conjunction with loops. Section 6.4 describes the `break`, `continue`, and `goto` statements. `break` jumps out of a loop and transfers control to the next statement after the loop, `continue` skips the rest of a loop iteration, and `goto` jumps to any statement within a function. Section 6.5 covers the null statement, which can be used to create loops with empty bodies.

6.1 The `while` Statement

Of all the ways to set up loops in C, the `while` statement is the simplest and most fundamental. The `while` statement has the form

`while` statement

`while ( `*expression*` ) `*statement*

The expression inside the parentheses is the controlling expression; the statement after the parentheses is the loop body. Here's an example:

```
while (i < n)    /* controlling expression */
  i = i * 2;     /* loop body */
```

Note that the parentheses are mandatory and that nothing goes between the right parenthesis and the loop body. (Some languages require the word `do`.)

When a `while` statement is executed, the controlling expression is evaluated first. If its value is nonzero (true), the loop body is executed and the expression is tested again. The process continues in this fashion—first testing the controlling expression, then executing the loop body—until the controlling expression eventually has the value zero.

The following example uses a `while` statement to compute the smallest power of 2 that is greater than or equal to a number `n`:

```
i = 1;
while (i < n)
  i = i * 2;
```

Suppose that `n` has the value 10. The following trace shows what happens when the `while` statement is executed:

`i = 1;`	`i` is now 1.
Is `i < n`?	Yes; continue.
`i = i * 2;`	`i` is now 2.
Is `i < n`?	Yes; continue.
`i = i * 2;`	`i` is now 4.
Is `i < n`?	Yes; continue.
`i = i * 2;`	`i` is now 8.
Is `i < n`?	Yes; continue.
`i = i * 2;`	`i` is now 16.
Is `i < n`?	No; exit from loop.

Notice how the loop keeps going as long as the controlling expression `(i < n)` is true. When the expression is false, the loop terminates, and `i` is greater than or equal to `n`, as desired.

Although the loop body must be a single statement, that's merely a technicality. If we want more than one statement, we can just use braces to create a single compound statement:

compound statements ➤5.2

```
while (i > 0) {
  printf("T minus %d and counting\n", i);
  i--;
}
```

Some programmers always use braces, even when they're not strictly necessary:

```
while (i < n) {    /* braces allowed, but not required */
  i = i * 2;
}
```

As a second example, let's trace the execution of the following statements, which display a series of "countdown" messages:

```
i = 10;
while (i > 0) {
  printf("T minus %d and counting\n", i);
  i--;
}
```

Before the `while` statement is executed, the variable `i` is assigned the value 10. Since 10 is greater than 0, the loop body is executed, causing the message `T minus 10 and counting` to be printed and `i` to be decremented. The condition `i > 0` is then tested again. Since 9 is greater than 0, the loop body is executed once more. This process continues until the message `T minus 1 and counting` is printed and `i` becomes 0. The test `i > 0` then fails, causing the loop to terminate.

The countdown example leads us to make several observations about the `while` statement:

- The controlling expression is false when a `while` loop terminates. Thus, when a loop controlled by the expression `i > 0` terminates, `i` must be less than or equal to 0. (Otherwise, we'd still be executing the loop!)
- The body of a `while` loop may not be executed at all. Since the controlling expression is tested *before* the loop body is executed, it's possible that the body isn't executed even once. If `i` has a negative or zero value when the countdown loop is first entered, the loop will do nothing.
- A `while` statement can often be written in a variety of ways. For example, we could make the countdown loop more concise by decrementing `i` inside the call of `printf`:

Q&A

```
while (i > 0)
  printf("T minus %d and counting\n", i--);
```

Infinite Loops

A `while` statement won't terminate if the controlling expression always has a nonzero value. In fact, C programmers sometimes deliberately create an *infinite loop* by using a nonzero constant as the controlling expression:

idiom

```
while (1) ...
```

A `while` statement of this form will execute forever unless its body contains a statement that transfers control out of the loop (`break`, `goto`, `return`) or calls a function that causes the program to terminate.

PROGRAM Printing a Table of Squares

Let's write a program that prints a table of squares. The program will first prompt the user to enter a number *n*. It will then print *n* lines of output, with each line containing a number between 1 and *n* together with its square:

```
This program prints a table of squares.
Enter number of entries in table: 5
         1         1
         2         4
         3         9
         4        16
         5        25
```

Let's have the program store the desired number of squares in a variable named n. We'll need a loop that repeatedly prints a number `i` and its square, starting with `i` equal to 1. The loop will repeat as long as `i` is less than or equal to n. We'll have to make sure to add 1 to `i` each time through the loop.

We'll write the loop as a `while` statement. (Frankly, we haven't got much choice, since the `while` statement is the only kind of loop we've covered so far.) Here's the finished program:

square.c

```
/* Prints a table of squares using a while statement */

#include <stdio.h>

int main(void)
{
  int i, n;

  printf("This program prints a table of squares.\n");
  printf("Enter number of entries in table: ");
  scanf("%d", &n);

  i = 1;
  while (i <= n) {
    printf("%10d%10d\n", i, i * i);
    i++;
  }

  return 0;
}
```

Note how `square.c` displays numbers in neatly aligned columns. The trick is to use a conversion specification like `%10d` instead of just `%d`, taking advantage of the fact that `printf` right-justifies numbers when a field width is specified.

PROGRAM Summing a Series of Numbers

As a second example of the `while` statement, let's write a program that sums a series of integers entered by the user. Here's what the user will see:

```
This program sums a series of integers.
Enter integers (0 to terminate): 8 23 71 5 0
The sum is: 107
```

Clearly we'll need a loop that uses `scanf` to read a number and then adds the number to a running total.

Letting `n` represent the number just read and `sum` the total of all numbers previously read, we end up with the following program:

sum.c

```
/* Sums a series of numbers */

#include <stdio.h>

int main(void)
{
  int n, sum = 0;

  printf("This program sums a series of integers.\n");
  printf("Enter integers (0 to terminate): ");

  scanf("%d", &n);
  while (n != 0) {
    sum += n;
    scanf("%d", &n);
  }
  printf("The sum is: %d\n", sum);

  return 0;
}
```

Notice that the condition `n != 0` is tested just after a number is read, allowing the loop to terminate as soon as possible. Also note that there are two identical calls of `scanf`, which is often hard to avoid when using `while` loops.

6.2 The `do` Statement

The `do` statement is closely related to the `while` statement; in fact, the `do` statement is essentially just a `while` statement whose controlling expression is tested *after* each execution of the loop body. The `do` statement has the form

do statement

`do` *statement* `while (` *expression* `) ;`

As with the `while` statement, the body of a `do` statement must be one statement (possibly compound, of course) and the controlling expression must be enclosed within parentheses.

When a `do` statement is executed, the loop body is executed first, then the controlling expression is evaluated. If the value of the expression is nonzero, the loop

body is executed again and then the expression is evaluated once more. Execution of the do statement terminates when the controlling expression has the value 0 *after* the loop body has been executed.

Let's rewrite the countdown example of Section 6.1, using a do statement this time:

```
i = 10;
do {
  printf("T minus %d and counting\n", i);
  --i;
} while (i > 0);
```

When the do statement is executed, the loop body is first executed, causing the message `T minus 10 and counting` to be printed and `i` to be decremented. The condition `i > 0` is now tested. Since 9 is greater than 0, the loop body is executed a second time. This process continues until the message `T minus 1 and counting` is printed and `i` becomes 0. The test `i > 0` now fails, causing the loop to terminate. As this example shows, the do statement is often indistinguishable from the `while` statement. The difference between the two is that the body of a do statement is always executed at least once; the body of a `while` statement is skipped entirely if the controlling expression is 0 initially.

Incidentally, it's a good idea to use braces in *all* do statements, whether or not they're needed, because a do statement without braces can easily be mistaken for a `while` statement:

```
do
  printf("T minus %d and counting\n", i--);
while (i > 0);
```

A careless reader might think that the word `while` was the beginning of a `while` statement.

PROGRAM Calculating the Number of Digits in an Integer

Although the `while` statement appears in C programs much more often than the do statement, the latter is handy for loops that must execute at least once. To illustrate this point, let's write a program that calculates the number of digits in an integer entered by the user:

```
Enter a nonnegative integer: 60
The number has 2 digit(s).
```

Our strategy will be to divide the user's input by 10 repeatedly until it becomes 0; the number of divisions performed is the number of digits. Clearly we'll need some kind of loop, since we don't know how many divisions it will take to reach 0. But should we use a `while` statement or a do statement? The do statement turns out to be more attractive, because every integer—even 0—has at least *one* digit. Here's the program:

numdigits.c

```
/* Calculates the number of digits in an integer */

#include <stdio.h>

int main(void)
{
  int digits = 0, n;

  printf("Enter a nonnegative integer: ");
  scanf("%d", &n);

  do {
    n /= 10;
    digits++;
  } while (n > 0);

  printf("The number has %d digit(s).\n", digits);

  return 0;
}
```

To see why the do statement is the right choice, let's see what would happen if we were to replace the do loop by a similar `while` loop:

```
while (n > 0) {
  n /= 10;
  digits++;
}
```

If n is 0 initially, this loop won't execute at all, and the program would print

```
The number has 0 digit(s).
```

6.3 The `for` Statement

We now come to the last of C's loops: the `for` statement. Don't be discouraged by the `for` statement's apparent complexity; it's actually the best way to write many loops. The `for` statement is ideal for loops that have a "counting" variable, but it's versatile enough to be used for other kinds of loops as well.

The `for` statement has the form

`for` statement

`for ( `*expr1*` ; `*expr2*` ; `*expr3*` )` *statement*

where *expr1*, *expr2*, and *expr3* are expressions. Here's an example:

```
for (i = 10; i > 0; i--)
  printf("T minus %d and counting\n", i);
```

When this `for` statement is executed, the variable `i` is initialized to 10, then `i` is tested to see if it's greater than 0. Since it is, the message `T minus 10 and`

counting is printed, then i is decremented. The condition i > 0 is then tested again. The loop body will be executed 10 times in all, with i varying from 10 down to 1.

The for statement is closely related to the while statement. In fact, except in a few rare cases, a for loop can always be replaced by an equivalent while loop:

Q&A

```
expr1 ;
while ( expr2 ) {
  statement
  expr3 ;
}
```

As this pattern shows, *expr1* is an initialization step that's performed only once, before the loop begins to execute, *expr2* controls loop termination (the loop continues executing as long as the value of *expr2* is nonzero), and *expr3* is an operation to be performed at the end of each loop iteration. Applying this pattern to our previous for loop example, we arrive at the following:

```
i = 10;
while (i > 0) {
  printf("T minus %d and counting\n", i);
  i--;
}
```

Studying the equivalent while statement can help us understand the fine points of a for statement. For example, suppose that we replace i-- by --i in our for loop example:

```
for (i = 10; i > 0; --i)
  printf("T minus %d and counting\n", i);
```

How does this change affect the loop? Looking at the equivalent while loop, we see that it has no effect:

```
i = 10;
while (i > 0) {
  printf("T minus %d and counting\n", i);
  --i;
}
```

Since the first and third expressions in a for statement are executed as statements, their values are irrelevant—they're useful only for their side effects. Consequently, these two expressions are usually assignments or increment/decrement expressions.

for Statement Idioms

The for statement is usually the best choice for loops that "count up" (increment a variable) or "count down" (decrement a variable). A for statement that counts up or down a total of n times will usually have one of the following forms:

- *Counting up from* **0** *to* **n–1***:*

idiom

```
for (i = 0; i < n; i++) …
```

- *Counting up from* **1** *to* **n***:*

idiom

```
for (i = 1; i <= n; i++) …
```

- *Counting down from* **n–1** *to* **0***:*

idiom

```
for (i = n - 1; i >= 0; i--) …
```

- *Counting down from* **n** *to* **1***:*

idiom

```
for (i = n; i > 0; i--) …
```

Imitating these patterns will help you avoid some of the following errors, which beginning C programmers often make:

- Using < instead of > (or vice versa) in the controlling expression. Notice that "counting up" loops use the < or <= operator, while "counting down" loops rely on > or >=.
- Using == in the controlling expression instead of <, <=, >, or >=. A controlling expression needs to be true at the beginning of the loop, then later become false so that the loop can terminate. A test such as `i == n` doesn't make much sense, because it won't be true initially.
- "Off-by-one" errors such as writing the controlling expression as `i <= n` instead of `i < n`.

Omitting Expressions in a `for` Statement

The `for` statement is even more flexible than we've seen so far. Some `for` loops may not need all three of the expressions that normally control the loop, so C allows us to omit any or all of the expressions.

If the *first* expression is omitted, no initialization is performed before the loop is executed:

```
i = 10;
for (; i > 0; --i)
  printf("T minus %d and counting\n", i);
```

In this example, `i` has been initialized by a separate assignment, so we've omitted the first expression in the `for` statement. (Notice that the semicolon between the first and second expressions remains. The two semicolons must always be present, even when we've omitted some of the expressions.)

If we omit the *third* expression in a `for` statement, the loop body is responsible for ensuring that the value of the second expression eventually becomes false. Our `for` statement example could be written like this:

```
for (i = 10; i > 0;)
  printf("T minus %d and counting\n", i--);
```

To compensate for omitting the third expression, we've arranged for `i` to be decremented inside the loop body.

When the *first* and *third* expressions are both omitted, the resulting loop is nothing more than a `while` statement in disguise. For example, the loop

```
for (; i > 0;)
  printf("T minus %d and counting\n", i--);
```

is the same as

```
while (i > 0)
  printf("T minus %d and counting\n", i--);
```

The `while` version is clearer and therefore preferable.

If the *second* expression is missing, it defaults to a true value, so the `for` statement doesn't terminate (unless stopped in some other fashion). For example, some programmers use the following `for` statement to establish an infinite loop:

Q&A

idiom

```
for (;;) …
```

C99 `for` Statements in C99

In C99, the first expression in a `for` statement can be replaced by a declaration. This feature allows the programmer to declare a variable for use by the loop:

```
for (int i = 0; i < n; i++)
  …
```

The variable `i` need not have been declared prior to this statement. (In fact, if a declaration of `i` already exists, this statement creates a *new* version of `i` that will be used solely within the loop.)

A variable declared by a `for` statement can't be accessed outside the body of the loop (we say that it's not ***visible*** outside the loop):

```
for (int i = 0; i < n; i++) {
  …
  printf("%d", i);    /* legal; i is visible inside loop */
  …
}
printf("%d", i);      /*** WRONG ***/
```

Having a `for` statement declare its own control variable is usually a good idea: it's convenient and it can make programs easier to understand. However, if the program needs to access the variable after loop termination, it's necessary to use the older form of the `for` statement.

Incidentally, a `for` statement may declare more than one variable, provided that all variables have the same type:

```
for (int i = 0, j = 0; i < n; i++)
  …
```

The Comma Operator

On occasion, we might like to write a `for` statement with two (or more) initialization expressions or one that increments several variables each time through the loop. We can do this by using a ***comma expression*** as the first or third expression in the `for` statement.

A comma expression has the form

comma expression

expr1 , *expr2*

where *expr1* and *expr2* are any two expressions. A comma expression is evaluated in two steps: First, *expr1* is evaluated and its value discarded. Second, *expr2* is evaluated; its value is the value of the entire expression. Evaluating *expr1* should always have a side effect; if it doesn't, then *expr1* serves no purpose.

For example, suppose that `i` and `j` have the values 1 and 5, respectively. When the comma expression `++i, i + j` is evaluated, `i` is first incremented, then `i + j` is evaluated, so the value of the expression is 7. (And, of course, `i` now has the value 2.) The precedence of the comma operator is less than that of all other operators, by the way, so there's no need to put parentheses around `++i` and `i + j`.

Occasionally, we'll need to chain together a series of comma expressions, just as we sometimes chain assignments together. The comma operator is left associative, so the compiler interprets

```
i = 1, j = 2, k = i + j
```

as

```
((i = 1), (j = 2)), (k = (i + j))
```

Since the left operand in a comma expression is evaluated before the right operand, the assignments `i = 1`, `j = 2`, and `k = i + j` will be performed from left to right.

The comma operator is provided for situations where C requires a single expression, but we'd like to have two or more expressions. In other words, the comma operator allows us to "glue" two expressions together to form a single expression. (Note the similarity to the compound statement, which allows us to treat a group of statements as a single statement.)

The need to glue expressions together doesn't arise that often. Certain macro definitions can benefit from the comma operator, as we'll see in a later chapter.

macro definitions ➤ 14.3

The `for` statement is the only other place where the comma operator is likely to be found. For example, suppose that we want to initialize two variables when entering a `for` statement. Instead of writing

```
sum = 0;
for (i = 1; i <= N; i++)
  sum += i;
```

we can write

```
for (sum = 0, i = 1; i <= N; i++)
  sum += i;
```

The expression `sum = 0, i = 1` first assigns 0 to `sum`, then assigns 1 to `i`. With additional commas, the `for` statement could initialize more than two variables.

PROGRAM Printing a Table of Squares (Revisited)

The `square.c` program (Section 6.1) can be improved by converting its `while` loop to a `for` loop:

square2.c

```
/* Prints a table of squares using a for statement */

#include <stdio.h>

int main(void)
{
  int i, n;

  printf("This program prints a table of squares.\n");
  printf("Enter number of entries in table: ");
  scanf("%d", &n);

  for (i = 1; i <= n; i++)
    printf("%10d%10d\n", i, i * i);

  return 0;
}
```

We can use this program to illustrate an important point about the `for` statement: C places no restrictions on the three expressions that control its behavior. Although these expressions usually initialize, test, and update the same variable, there's no requirement that they be related in any way. Consider the following version of the same program:

square3.c

```
/* Prints a table of squares using an odd method */

#include <stdio.h>

int main(void)
{
int i, n, odd, square;

printf("This program prints a table of squares.\n");
printf("Enter number of entries in table: ");
scanf("%d", &n);

i = 1;
odd = 3;
for (square = 1; i <= n; odd += 2) {
  printf("%10d%10d\n", i, square);
  ++i;
```

```
    square += odd;
  }

  return 0;
}
```

The `for` statement in this program initializes one variable (`square`), tests another (`i`), and increments a third (`odd`). `i` is the number to be squared, `square` is the square of `i`, and `odd` is the odd number that must be added to the current square to get the next square (allowing the program to compute consecutive squares without performing any multiplications).

The tremendous flexibility of the `for` statement can sometimes be useful; we'll find it to be a great help when working with linked lists. The `for` statement can easily be misused, though, so don't go overboard. The `for` loop in `square3.c` would be a lot clearer if we rearranged its pieces so that the loop is clearly controlled by `i`.

linked lists ➤ 17.5

6.4 Exiting from a Loop

We've seen how to write loops that have an exit point before the loop body (using `while` and `for` statements) or after it (using `do` statements). Occasionally, however, we'll need a loop with an exit point in the middle. We may even want a loop to have more than one exit point. The `break` statement makes it possible to write either kind of loop.

After we've examined the `break` statement, we'll look at a couple of related statements: `continue` and `goto`. The `continue` statement makes it possible to skip part of a loop iteration without jumping out of the loop. The `goto` statement allows a program to jump from one statement to another. Thanks to the availability of statements such as `break` and `continue`, the `goto` statement is rarely used.

The `break` Statement

We've already discussed how a `break` statement can transfer control out of a `switch` statement. The `break` statement can also be used to jump out of a `while`, `do`, or `for` loop.

Suppose that we're writing a program that checks whether a number `n` is prime. Our plan is to write a `for` statement that divides `n` by the numbers between 2 and `n - 1`. We should break out of the loop as soon as any divisor is found; there's no need to try the remaining possibilities. After the loop has terminated, we can use an `if` statement to determine whether termination was premature (hence `n` isn't prime) or normal (`n` is prime):

```
for (d = 2; d < n; d++)
  if (n % d == 0)
    break;
```

```
if (d < n)
  printf("%d is divisible by %d\n", n, d);
else
  printf("%d is prime\n", n);
```

The `break` statement is particularly useful for writing loops in which the exit point is in the middle of the body rather than at the beginning or end. Loops that read user input, terminating when a particular value is entered, often fall into this category:

```
for (;;) {
  printf("Enter a number (enter 0 to stop): ");
  scanf("%d", &n);
  if (n == 0)
    break;
  printf("%d cubed is %d\n", n, n * n * n);
}
```

A `break` statement transfers control out of the *innermost* enclosing `while`, `do`, `for`, or `switch` statement. Thus, when these statements are nested, the `break` statement can escape only one level of nesting. Consider the case of a `switch` statement nested inside a `while` statement:

```
while (...) {
  switch (...) {
    ...
    break;
    ...
  }
}
```

The `break` statement transfers control out of the `switch` statement, but not out of the `while` loop. I'll return to this point later.

The `continue` Statement

The `continue` statement doesn't really belong here, because it doesn't exit from a loop. It's similar to `break`, though, so its inclusion in this section isn't completely arbitrary. `break` transfers control just *past* the end of a loop, while `continue` transfers control to a point just *before* the end of the loop body. With `break`, control leaves the loop; with `continue`, control remains inside the loop. There's another difference between `break` and `continue`: `break` can be used in `switch` statements and loops (`while`, `do`, and `for`), whereas `continue` is limited to loops.

The following example, which reads a series of numbers and computes their sum, illustrates a simple use of `continue`. The loop terminates when 10 nonzero numbers have been read. Whenever the number 0 is read, the `continue` statement is executed, skipping the rest of the loop body (the statements `sum += i;` and `n++;`) but remaining inside the loop.

```
n = 0;
sum = 0;
while (n < 10) {
  scanf("%d", &i);
  if (i == 0)
    continue;
  sum += i;
  n++;
  /* continue jumps to here */
}
```

If `continue` were not available, we could have written the example as follows:

```
n = 0;
sum = 0;
while (n < 10) {
  scanf("%d", &i);
  if (i != 0) {
    sum += i;
    n++;
  }
}
```

The `goto` Statement

`break` and `continue` are jump statements that transfer control from one point in the program to another. Both are restricted: the target of a `break` is a point just *beyond* the end of the enclosing loop, while the target of a `continue` is a point just *before* the end of the loop. The `goto` statement, on the other hand, is capable of jumping to *any* statement in a function, provided that the statement has a *label*.

C99

(C99 places an additional restriction on the `goto` statement: it can't be used to bypass the declaration of a variable-length array.)

variable-length arrays ➤8.3

A label is just an identifier placed at the beginning of a statement:

labeled statement

identifier : *statement*

A statement may have more than one label. The `goto` statement itself has the form

`goto` statement

`goto` *identifier* ;

Executing the statement `goto` *L*; transfers control to the statement that follows the label *L*, which must be in the same function as the `goto` statement itself.

If C didn't have a `break` statement, here's how we might use a `goto` statement to exit prematurely from a loop:

```
for (d = 2; d < n; d++)
  if (n % d == 0)
    goto done;
```

```
done:
if (d < n)
  printf("%d is divisible by %d\n", n, d);
else
  printf("%d is prime\n", n);
```

Q&A

exit function ➤9.5

The `goto` statement, a staple of older programming languages, is rarely needed in everyday C programming. The `break`, `continue`, and `return` statements—which are essentially restricted `goto` statements—and the `exit` function are sufficient to handle most situations that might require a `goto` in other languages.

Nonetheless, the `goto` statement can be helpful once in a while. Consider the problem of exiting a loop from within a `switch` statement. As we saw earlier, the `break` statement doesn't quite have the desired effect: it exits from the `switch`, but not from the loop. A `goto` statement solves the problem:

```
while (...) {
  switch (...) {
    ...
    goto loop_done;    /* break won't work here */
    ...
  }
}
loop_done: ...
```

The `goto` statement is also useful for exiting from nested loops.

PROGRAM

Balancing a Checkbook

Many simple interactive programs are menu-based: they present the user with a list of commands to choose from. Once the user has selected a command, the program performs the desired action, then prompts the user for another command. This process continues until the user selects an "exit" or "quit" command.

The heart of such a program will obviously be a loop. Inside the loop will be statements that prompt the user for a command, read the command, then decide what action to take:

```
for (;;) {
  prompt user to enter command;
  read command;
  execute command;
}
```

Executing the command will require a `switch` statement (or cascaded `if` statement):

```
for (;;) {
  prompt user to enter command;
  read command;
  switch (command) {
    case command1: perform operation1; break;
```

```
    case command2: perform operation2; break;
    .
    .
    .
    case commandn: perform operationn; break;
    default: print error message; break;
  }
}
```

To illustrate this arrangement, let's develop a program that maintains a checkbook balance. The program will offer the user a menu of choices: clear the account balance, credit money to the account, debit money from the account, display the current balance, and exit the program. The choices are represented by the integers 0, 1, 2, 3, and 4, respectively. Here's what a session with the program will look like:

```
*** ACME checkbook-balancing program ***
Commands: 0=clear, 1=credit, 2=debit, 3=balance, 4=exit

Enter command: 1
Enter amount of credit: 1042.56
Enter command: 2
Enter amount of debit: 133.79
Enter command: 1
Enter amount of credit: 1754.32
Enter command: 2
Enter amount of debit: 1400
Enter command: 2
Enter amount of debit: 68
Enter command: 2
Enter amount of debit: 50
Enter command: 3
Current balance: $1145.09
Enter command: 4
```

When the user enters the command 4 (exit), the program needs to exit from the `switch` statement *and* the surrounding loop. The `break` statement won't help, and we'd prefer not to use a `goto` statement. Instead, we'll have the program execute a `return` statement, which will cause the `main` function to return to the operating system.

checking.c

```
/* Balances a checkbook */

#include <stdio.h>

int main(void)
{
  int cmd;
  float balance = 0.0f, credit, debit;

  printf("*** ACME checkbook-balancing program ***\n");
  printf("Commands: 0=clear, 1=credit, 2=debit, ");
  printf("3=balance, 4=exit\n\n");
```

```
for (;;) {
  printf("Enter command: ");
  scanf("%d", &cmd);
  switch (cmd) {
    case 0:
      balance = 0.0f;
      break;
    case 1:
      printf("Enter amount of credit: ");
      scanf("%f", &credit);
      balance += credit;
      break;
    case 2:
      printf("Enter amount of debit: ");
      scanf("%f", &debit);
      balance -= debit;
      break;
    case 3:
      printf("Current balance: $%.2f\n", balance);
      break;
    case 4:
      return 0;
    default:
      printf("Commands: 0=clear, 1=credit, 2=debit, ");
      printf("3=balance, 4=exit\n\n");
      break;
  }
}
```

Note that the `return` statement is not followed by a `break` statement. A `break` immediately following a `return` can never be executed. and many compilers will issue a warning message.

6.5 The Null Statement

A statement can be *null*—devoid of symbols except for the semicolon at the end. Here's an example:

```
i = 0; ; j = 1;
```

This line contains three statements: an assignment to `i`. a null statement. and an assignment to `j`.

Q&A

The null statement is primarily good for one thing: writing loops whose bodies are empty. As an example, recall the prime-finding loop of Section 6.4:

```
for (d = 2; d < n; d++)
  if (n % d == 0)
    break;
```

If we move the `n % d == 0` condition into the loop's controlling expression, the body of the loop becomes empty:

```
for (d = 2; d < n && n % d != 0; d++)
  /* empty loop body */ ;
```

Each time through the loop, the condition `d < n` is tested first; if it's false, the loop terminates. Otherwise, the condition `n % d != 0` is tested, and if that's false, the loop terminates. (In the latter case, `n % d == 0` must be true; in other words, we've found a divisor of `n`.)

Note how we've put the null statement on a line by itself, instead of writing

```
for (d = 2; d < n && n % d != 0; d++);
```

Q&A C programmers customarily put the null statement on a line by itself. Otherwise, someone reading the program might get confused about whether the statement after the `for` was actually its body:

```
for (d = 2; d < n && n % d != 0; d++);
if (d < n)
  printf("%d is divisible by %d\n", n, d);
```

Converting an ordinary loop into one with an empty body doesn't buy much: the new loop is often more concise but usually no more efficient. In a few cases, though, a loop with an empty body is clearly superior to the alternatives. For example, we'll find these loops to be handy for reading character data.

reading characters ➤ 7.3

Accidentally putting a semicolon after the parentheses in an `if`, `while`, or `for` statement creates a null statement, thus ending the `if`, `while`, or `for` prematurely.

- In an `if` statement, putting a semicolon after the parentheses creates an `if` statement that apparently performs the same action regardless of the value of its controlling expression:

  ```
  if (d == 0);                                    /*** WRONG ***/
    printf("Error: Division by zero\n");
  ```

  The call of `printf` isn't inside the `if` statement, so it's performed regardless of whether `d` is equal to 0.

- In a `while` statement, putting a semicolon after the parentheses may create an infinite loop:

  ```
  i = 10;
  while (i > 0);                                  /*** WRONG ***/
  {
    printf("T minus %d and counting\n", i);
    --i;
  }
  ```

Another possibility is that the loop terminates, but the statement that should be the loop body is executed only once, after the loop has terminated:

```
i = 11;
while (--i > 0);                                    /*** WRONG ***/
  printf("T minus %d and counting\n", i);
```

This example prints the message

```
T minus 0 and counting
```

- In a `for` statement, putting a semicolon after the parentheses causes the statement that should be the loop body to be executed only once:

  ```
  for (i = 10; i > 0; i--);                         /*** WRONG ***/
    printf("T minus %d and counting\n", i);
  ```

  This example also prints the message

  ```
  T minus 0 and counting
  ```

Q & A

Q: **The following loop appears in Section 6.1:**

```
while (i > 0)
  printf("T minus %d and counting\n", i--);
```

Why not shorten the loop even more by removing the "`> 0`" test?

```
while (i)
  printf("T minus %d and counting\n", i--);
```

This version will stop when `i` reaches 0, so it should be just as good as the original. [p. 101]

A: The new version is certainly more concise, and many C programmers would write the loop in just this way. It does have drawbacks, though.

First, the new loop is not as easy to read as the original. It's clear that the loop will terminate when `i` reaches 0, but it's not obvious whether we're counting up or down. In the original loop, that information can be deduced from the controlling expression, `i > 0`.

Second, the new loop behaves differently than the original if `i` should happen to have a negative value when the loop begins to execute. The original loop terminates immediately, but the new loop doesn't.

Q: **Section 6.3 says that, except in rare cases, `for` loops can be converted to `while` loops using a standard pattern. Can you give an example of such a case? [p. 106]**

A: When the body of a `for` loop contains a `continue` statement, the `while` pattern shown in Section 6.3 is no longer valid. Consider the following example from Section 6.4:

```
n = 0;
sum = 0;
while (n < 10) {
  scanf("%d", &i);
  if (i == 0)
    continue;
  sum += i;
  n++;
}
```

At first glance, it looks as though we could convert the `while` loop into a `for` loop:

```
sum = 0;
for (n = 0; n < 10; n++) {
  scanf("%d", &i);
  if (i == 0)
    continue;
  sum += i;
}
```

Unfortunately, this loop isn't equivalent to the original. When `i` is equal to 0, the original loop doesn't increment `n`, but the new loop does.

Q: Which form of infinite loop is preferable, `while (1)` or `for (;;)`? [p. 108]

A: C programmers have traditionally preferred `for (;;)` for reasons of efficiency; older compilers would often force programs to test the `1` condition each time through the `while` loop. With modern compilers, however, there should be no difference in performance.

Q: I've heard that programmers should never use the `continue` statement. Is this true?

A: It's true that `continue` statements are rare. Still, `continue` is handy once in a while. Suppose we're writing a loop that reads some input data, checks that it's valid, and, if so, processes the input in some way. If there are a number of validity tests, or if they're complex, `continue` can be helpful. The loop would look something like this:

```
for (;;) {
  read data;
  if (data fails first test)
    continue;
  if (data fails second test)
    continue;
  .
  .
  .
```

```
    if (data fails last test)
      continue;
    process data;
}
```

Q: What's so bad about the `goto` statement? [p. 114]

A: The `goto` statement isn't inherently evil; it's just that we usually have better alternatives. Programs that use more than a few `goto` statements can quickly degenerate into "spaghetti code," with control blithely jumping from here to there. Spaghetti code is hard to understand and hard to modify.

`goto` statements make programs hard to read because they can jump either forward or backward. (In contrast, `break` and `continue` only jump forward.) A program that contains `goto` statements often requires the reader to jump back and forth in an attempt to follow the flow of control.

`goto` statements can make programs hard to modify, since they make it possible for a section of code to serve more than one purpose. For example, a statement that is preceded by a label might be reachable either by "falling through" from the previous statement or by executing one of several `goto` statements.

Q: Does the null statement have any uses besides indicating that the body of a loop is empty? [p. 116]

A: Very few. Since the null statement can appear wherever a statement is allowed, there are many *potential* uses for the null statement. In practice, however, there's only one other use of the null statement, and it's rare.

Suppose that we need to put a label at the end of a compound statement. A label can't stand alone: it must always be followed by a statement. Putting a null statement after the label solves the problem:

```
{
  ...
  goto end_of_stmt;
  ...
  end_of_stmt: ;
}
```

Q: Are there any other ways to make an empty loop body stand out besides putting the null statement on a line by itself? [p. 117]

A: Some programmers use a dummy `continue` statement:

```
for (d = 2; d < n && n % d != 0; d++)
  continue;
```

Others use an empty compound statement:

```
for (d = 2; d < n && n % d != 0; d++)
  {}
```

Exercises

Section 6.1

1. What output does the following program fragment produce?

```
i = 1;
while (i <= 128) {
  printf("%d ", i);
  i *= 2;
}
```

Section 6.2

2. What output does the following program fragment produce?

```
i = 9384;
do {
  printf("%d ", i);
  i /= 10;
} while (i > 0);
```

Section 6.3

*3. What output does the following `for` statement produce?

```
for (i = 5, j = i - 1; i > 0, j > 0; --i, j = i - 1)
  printf("%d ", i);
```

W 4. Which one of the following statements is not equivalent to the other two (assuming that the loop bodies are the same)?

(a) `for (i = 0; i < 10; i++) ...`
(b) `for (i = 0; i < 10; ++i) ...`
(c) `for (i = 0; i++ < 10; ) ...`

5. Which one of the following statements is not equivalent to the other two (assuming that the loop bodies are the same)?

(a) `while (i < 10) {...}`
(b) `for (; i < 10;) {...}`
(c) `do {...} while (i < 10);`

6. Translate the program fragment of Exercise 1 into a single `for` statement.

7. Translate the program fragment of Exercise 2 into a single `for` statement.

*8. What output does the following `for` statement produce?

```
for (i = 10; i >= 1; i /= 2)
  printf("%d ", i++);
```

9. Translate the `for` statement of Exercise 8 into an equivalent `while` statement. You will need one statement in addition to the `while` loop itself.

Section 6.4

W 10. Show how to replace a `continue` statement by an equivalent `goto` statement.

11. What output does the following program fragment produce?

```
sum = 0;
for (i = 0; i < 10; i++) {
  if (i % 2)
    continue;
  sum += i;
}
printf("%d\n", sum);
```

W 12. The following "prime-testing" loop appeared in Section 6.4 as an example:

```
for (d = 2; d < n; d++)
  if (n % d == 0)
    break;
```

This loop isn't very efficient. It's not necessary to divide n by all numbers between 2 and n – 1 to determine whether it's prime. In fact, we need only check divisors up to the square root of n. Modify the loop to take advantage of this fact. *Hint:* Don't try to compute the square root of n; instead, compare d * d with n.

Section 6.5

*13. Rewrite the following loop so that its body is empty:

```
for (n = 0; m > 0; n++)
  m /= 2;
```

W*14. Find the error in the following program fragment and fix it.

```
if (n % 2 == 0);
  printf("n is even\n");
```

Programming Projects

1. Write a program that finds the largest in a series of numbers entered by the user. The program must prompt the user to enter numbers one by one. When the user enters 0 or a negative number, the program must display the largest nonnegative number entered:

```
Enter a number: 60
Enter a number: 38.3
Enter a number: 4.89
Enter a number: 100.62
Enter a number: 75.2295
Enter a number: 0

The largest number entered was 100.62
```

Notice that the numbers aren't necessarily integers.

W 2. Write a program that asks the user to enter two integers, then calculates and displays their greatest common divisor (GCD):

```
Enter two integers: 12 28
Greatest common divisor: 4
```

Hint: The classic algorithm for computing the GCD, known as Euclid's algorithm, goes as follows: Let m and n be variables containing the two numbers. If n is 0, then stop: m contains the GCD. Otherwise, compute the remainder when m is divided by n. Copy n into m and copy the remainder into n. Then repeat the process, starting with testing whether n is 0.

3. Write a program that asks the user to enter a fraction, then reduces the fraction to lowest terms:

```
Enter a fraction: 6/12
In lowest terms: 1/2
```

Hint: To reduce a fraction to lowest terms, first compute the GCD of the numerator and denominator. Then divide both the numerator and denominator by the GCD.

W 4. Add a loop to the `broker.c` program of Section 5.2 so that the user can enter more than one trade and the program will calculate the commission on each. The program should terminate when the user enters 0 as the trade value:

```
Enter value of trade: 30000
Commission: $166.00

Enter value of trade: 20000
Commission: $144.00

Enter value of trade: 0
```

5. Programming Project 1 in Chapter 4 asked you to write a program that displays a two-digit number with its digits reversed. Generalize the program so that the number can have one, two, three, or more digits. *Hint:* Use a `do` loop that repeatedly divides the number by 10, stopping when it reaches 0.

W 6. Write a program that prompts the user to enter a number *n*, then prints all even squares between 1 and *n*. For example, if the user enters 100, the program should print the following:

```
4
16
36
64
100
```

7. Rearrange the `square3.c` program so that the `for` loop initializes `i`, tests `i`, and increments `i`. Don't rewrite the program; in particular, don't use any multiplications.

W 8. Write a program that prints a one-month calendar. The user specifies the number of days in the month and the day of the week on which the month begins:

```
Enter number of days in month: 31
Enter starting day of the week (1=Sun, 7=Sat): 3

       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31
```

Hint: This program isn't as hard as it looks. The most important part is a `for` statement that uses a variable `i` to count from 1 to `n`, where `n` is the number of days in the month, printing each value of `i`. Inside the loop, an `if` statement tests whether `i` is the last day in a week; if so, it prints a new-line character.

9. Programming Project 8 in Chapter 2 asked you to write a program that calculates the remaining balance on a loan after the first, second, and third monthly payments. Modify the program so that it also asks the user to enter the number of payments and then displays the balance remaining after each of these payments.

10. Programming Project 9 in Chapter 5 asked you to write a program that determines which of two dates comes earlier on the calendar. Generalize the program so that the user may enter any number of dates. The user will enter 0/0/0 to indicate that no more dates will be entered:

```
Enter a date (mm/dd/yy): 3/6/08
Enter a date (mm/dd/yy): 5/17/07
Enter a date (mm/dd/yy): 6/3/07
Enter a date (mm/dd/yy): 0/0/0
5/17/07 is the earliest date
```

11. The value of the mathematical constant *e* can be expressed as an infinite series:

    $e = 1 + 1/1! + 1/2! + 1/3! + \ldots$

    Write a program that approximates *e* by computing the value of

    $1 + 1/1! + 1/2! + 1/3! + \ldots + 1/n!$

    where n is an integer entered by the user.

12. Modify Programming Project 11 so that the program continues adding terms until the current term becomes less than ε, where ε is a small (floating-point) number entered by the user.

7 Basic Types

Make no mistake about it: Computers process numbers—not symbols. We measure our understanding (and control) by the extent to which we can arithmetize an activity.

So far, we've used only two of C's ***basic*** (built-in) ***types:*** `int` and `float`. (We've also seen `_Bool`, which is a basic type in C99.) This chapter describes the rest of the basic types and discusses important issues about types in general. Section 7.1 reveals the full range of integer types, which include long integers, short integers, and unsigned integers. Section 7.2 introduces the `double` and `long double` types, which provide a larger range of values and greater precision than `float`. Section 7.3 covers the `char` type, which we'll need in order to work with character data. Section 7.4 tackles the thorny topic of converting a value of one type to an equivalent value of another. Section 7.5 shows how to use `typedef` to define new type names. Finally, Section 7.6 describes the `sizeof` operator, which measures the amount of storage required for a type.

7.1 Integer Types

C supports two fundamentally different kinds of numeric types: integer types and floating types. Values of an ***integer type*** are whole numbers, while values of a floating type can have a fractional part as well. The integer types, in turn, are divided into two categories: signed and unsigned.

Signed and Unsigned Integers

The leftmost bit of a ***signed*** integer (known as the ***sign bit***) is 0 if the number is positive or zero, 1 if it's negative. Thus, the largest 16-bit integer has the binary representation