

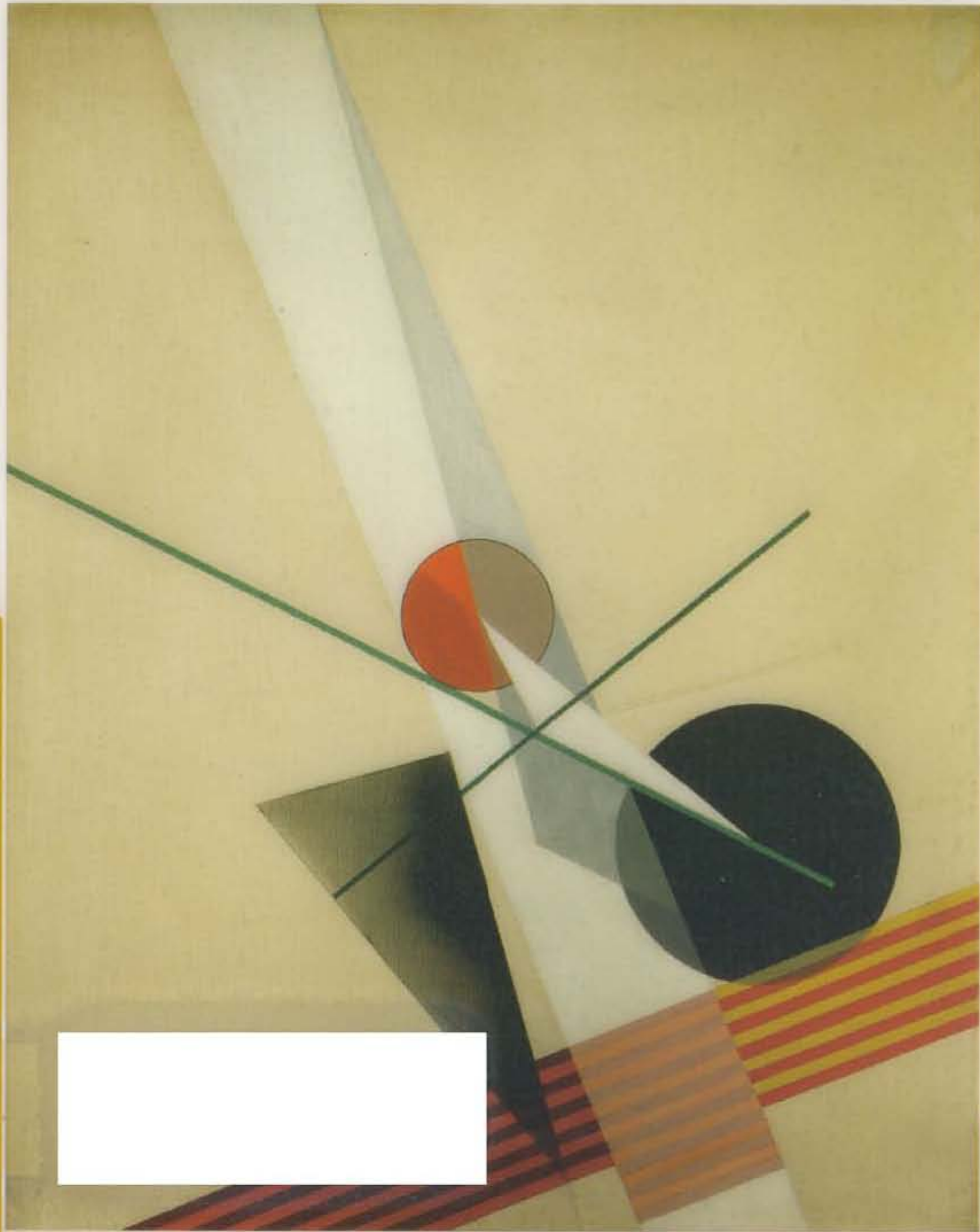
Covers both C89
and C99

K.N.KING

C PROGRAMMING

A Modern Approach

SECOND EDITION



K. N. KING

C PROGRAMMING

A Modern Approach SECOND EDITION

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

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

FEATURES OF THE SECOND EDITION

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

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

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

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

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


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

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

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



Cover design by Joan Greenfield
Cover art: Laszlo Moholy-Nagy, AXXI, Westfaelisches
Landesmuseum, Muenster, Westphalia, Germany
Photo: Erich Lessing / Art Resource, NY
Copyright: © ARS, NY

 W. W. NORTON
NEW YORK • LONDON

www.wwnorton.com

PREFACE

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

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


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

What's New in the Second Edition

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

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

C99

- **Includes a quick reference to all C89 and C99 library functions.** Appendix D in the first edition described all C89 standard library functions. In this edition, the appendix covers all C89 and C99 library functions.
- **Expanded coverage of GCC.** In the years since the first edition, use of GCC (originally the GNU C Compiler, now the GNU Compiler Collection) has spread. GCC has some significant advantages, including high quality, low (i.e., no) cost, and portability across a variety of hardware and software platforms. In recognition of its growing importance, I've included more information about GCC in this edition, including discussions of how to use it as well as common GCC error messages and warnings.
- **New coverage of abstract data types.** In the first edition, a significant portion of Chapter 19 was devoted to C++. This material seems less relevant today, since students may already have learned C++, Java, or C# before reading this book. In this edition, coverage of C++ has been replaced by a discussion of how to set up abstract data types in C.
- **Expanded coverage of international features.** Chapter 25, which is devoted to C's international features, is now much longer and more detailed. Information about the Unicode/UCS character set and its encodings is a highlight of the expanded coverage.
- **Updated to reflect today's CPUs and operating systems.** When I wrote the first edition, 16-bit architectures and the DOS operating system were still relevant to many readers, but such is not the case today. I've updated the discussion to focus more on 32-bit and 64-bit architectures. The rise of Linux and other versions of UNIX has dictated a stronger focus on that family of operating systems, although aspects of Windows and the Mac OS operating system that affect C programmers are mentioned as well.
- **More exercises and programming projects.** The first edition of this book contained 311 exercises. This edition has nearly 500 (498, to be exact), divided into two groups: exercises and programming projects.
- **Solutions to selected exercises and programming projects.** The most frequent request I received from readers of the first edition was to provide answers to the exercises. In response to this request, I've put the answers to roughly one-third of the exercises and programming projects on the web at knking.com/books/c2. 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 re-examined the old conventions of C, keeping the ones that are still helpful.
- **Emphasize software engineering.** I treat C as a mature software engineering tool, emphasizing how to use it to cope with issues that arise during programming-in-the-large. I stress making programs readable, maintainable, reliable, and portable, and I put special emphasis on information hiding.
- **Postpone C's low-level features.** These features, although handy for the kind of systems programming originally done in C, are not as relevant now that C is used for a great variety of applications. Instead of introducing them in the early chapters, as many C books do, I postpone them until Chapter 20.
- **De-emphasize "manual optimization."** Many books teach the reader to write tricky code in order to gain small savings in program efficiency. With today's abundance of optimizing C compilers, these techniques are often no longer necessary; in fact, they can result in programs that are less efficient.

Q&A Sections

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

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

Q&A

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.

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

The Standard C Library

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

Advanced Features of C

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

Reference

A	C Operators	735
B	C99 versus C89	737
C	C89 versus K&R C	743
D	Standard Library Functions	747
E	ASCII Character Set	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 <code>else</code> Clause	78
	Cascaded <code>if</code> Statements	80
	Program: Calculating a Broker's Commission	81
	The "Dangling <code>else</code> " Problem	82
	Conditional Expressions	83
	Boolean Values in C89	84
	Boolean Values in C99	85
5.3	The <code>switch</code> Statement	86
	The Role of the <code>break</code> Statement	88
	Program: Printing a Date in Legal Form	89
6	LOOPS	99
6.1	The <code>while</code> Statement	99
	Infinite Loops	101
	Program: Printing a Table of Squares	102
	Program: Summing a Series of Numbers	102
6.2	The <code>do</code> Statement	103
	Program: Calculating the Number of Digits in an Integer	104
6.3	The <code>for</code> Statement	105
	<code>for</code> Statement Idioms	106
	Omitting Expressions in a <code>for</code> Statement	107
	<code>for</code> Statements in C99	108
	The Comma Operator	109
	Program: Printing a Table of Squares (Revisited)	110
6.4	Exiting from a Loop	111
	The <code>break</code> Statement	111
	The <code>continue</code> Statement	112
	The <code>goto</code> 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 <code>scanf</code> and <code>printf</code>	139
	Reading and Writing Characters using <code>getchar</code> and <code>putchar</code>	140
	Program: Determining the Length of a Message	141
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 <code>sizeof</code> 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 <code>sizeof</code> 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 <code>static</code> in Array Parameter Declarations	200
	Compound Literals	200
9.4	The <code>return</code> Statement	201
9.5	Program Termination	202
	The <code>exit</code> 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 <code>const</code> 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 <code>*</code> and <code>++</code> 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 <code>printf</code> and <code>puts</code>	284
	Reading Strings Using <code>scanf</code> and <code>gets</code>	285
	Reading Strings Character by Character	286
13.4	Accessing the Characters in a String	287
13.5	Using the C String Library	289
	The <code>strcpy</code> (String Copy) Function	290
	The <code>strlen</code> (String Length) Function	291
	The <code>strcat</code> (String Concatenation) Function	291
	The <code>strcmp</code> (String Comparison) Function	292
	Program: Printing a One-Month Reminder List	293
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 <code>__func__</code> Identifier	333
14.4	Conditional Compilation	333
	The <code>#if</code> and <code>#endif</code> Directives	334
	The <code>defined</code> Operator	335
	The <code>#ifdef</code> and <code>#ifndef</code> Directives	335
	The <code>#elif</code> and <code>#else</code> Directives	336
	Uses of Conditional Compilation	337
14.5	Miscellaneous Directives	338
	The <code>#error</code> Directive	338
	The <code>#line</code> Directive	339
	The <code>#pragma</code> Directive	340
	The <code>_Pragma</code> Operator	341
15	WRITING LARGE PROGRAMS	349
15.1	Source Files	349
15.2	Header Files	350
	The <code>#include</code> Directive	351
	Sharing Macro Definitions and Type Definitions	353
	Sharing Function Prototypes	354
	Sharing Variable Declarations	355
	Nested Includes	357
	Protecting Header Files	357
	<code>#error</code> Directives in Header Files	358
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 <code>malloc</code> to Allocate Memory for a String	416
	Using Dynamic Storage Allocation in String Functions	417
	Arrays of Dynamically Allocated Strings	418
	Program: Printing a One-Month Reminder List (Revisited)	418
17.3	Dynamically Allocated Arrays	420
	Using <code>malloc</code> to Allocate Storage for an Array	420
	The <code>calloc</code> Function	421
	The <code>realloc</code> Function	421
17.4	Deallocating Storage	422
	The <code>free</code> Function	423
	The “Dangling Pointer” Problem	424
17.5	Linked Lists	424
	Declaring a Node Type	425
	Creating a Node	425
	The <code>-></code> Operator	426
	Inserting a Node at the Beginning of a Linked List	427
	Searching a Linked List	429
	Deleting a Node from a Linked List	431
	Ordered Lists	433
	Program: Maintaining a Parts Database (Revisited)	433
17.6	Pointers to Pointers	438

17.7	Pointers to Functions	439
	Function Pointers as Arguments	439
	The <code>qsort</code> Function	440
	Other Uses of Function Pointers	442
	Program: Tabulating the Trigonometric Functions	443
17.8	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 <code>auto</code> Storage Class	460
	The <code>static</code> Storage Class	461
	The <code>extern</code> Storage Class	462
	The <code>register</code> Storage Class	463
	The Storage Class of a Function	464
	Summary	465
18.3	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, <i>And</i> , <i>Exclusive Or</i> , and <i>Inclusive Or</i>	511
	Using the Bitwise Operators to Access Bits	512
	Using the Bitwise Operators to Access Bit-Fields	513
	Program: XOR Encryption	514
20.2	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 <code>volatile</code> 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 <code><stddef.h></code> Header: Common Definitions	535
21.5	The <code><stdbool.h></code> 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 <code><ctype.h></code> 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 <code><string.h></code> 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 <code><assert.h></code> Header: Diagnostics	628
24.2	The <code><errno.h></code> Header: Errors	629
	The <code>perror</code> and <code>strerror</code> Functions	630
24.3	The <code><signal.h></code> Header: Signal Handling	631
	Signal Macros	631
	The <code>signal</code> Function	632
	Predefined Signal Handlers	633
	The <code>raise</code> Function	634
	Program: Testing Signals	634
24.4	The <code><setjmp.h></code> Header: Nonlocal Jumps	635
	Program: Testing <code>setjmp/longjmp</code>	636
25	INTERNATIONAL FEATURES	641
25.1	The <code><locale.h></code> Header: Localization	642
	Categories	642
	The <code>setlocale</code> Function	643
	The <code>localeconv</code> 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 <code><iso646.h></code> Header: Alternative Spellings	656
25.4	Universal Character Names (C99)	656
25.5	The <code><wchar.h></code> 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 <code><wctype.h></code> 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 <code><stdarg.h></code> Header: Variable Arguments	677
	Calling a Function with a Variable Argument List	679
	The <code>v...printf</code> Functions	680
	The <code>v...scanf</code> Functions	681
26.2	The <code><stdlib.h></code> 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 <code><time.h></code> 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 <code><stdint.h></code> Header (C99): Integer Types	705
	<code><stdint.h></code> Types	706
	Limits of Specified-Width Integer Types	707
	Limits of Other Integer Types	708
	Macros for Integer Constants	708
27.2	The <code><inttypes.h></code> 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 <code><complex.h></code> Header (C99): Complex Arithmetic	717
	<code><complex.h></code> Macros	717
	The <code>CX_LIMITED_RANGE</code> Pragma	718
	<code><complex.h></code> 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 <code><tgmath.h></code> Header (C99): Type-Generic Math	723
	Type-Generic Macros	724
	Invoking a Type-Generic Macro	725
27.6	The <code><fenv.h></code> Header (C99): Floating-Point Environment	726
	Floating-Point Status Flags and Control Modes	727
	<code><fenv.h></code> Macros	727
	The <code>FENV_ACCESS</code> 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)&l: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 \triangle symbol. For a more extensive list of pitfalls, see Andrew Koenig's *C Traps and Pitfalls* (Reading, Mass.: Addison-Wesley, 1989). Modern compilers will detect common pitfalls and issue warnings, but no compiler spots them all.
- **Use software tools to make programs more reliable.** C programmers are prolific tool builders (and users). One of the most famous C tools is named `lint`. `lint`, which is traditionally provided with UNIX, can subject a program to a more extensive error analysis than most C compilers. If `lint` (or a similar program) is available, it's a good idea to use it. Another useful tool is a debugger. Because of the nature of C, many bugs can't be detected by a C compiler; these show up instead in the form of run-time errors or incorrect output. Consequently, using a good debugger is practically mandatory for C programmers.
- **Take advantage of existing code libraries.** One of the benefits of using C is that so many other people also use it; it's a good bet that they've written code you can employ in your own programs. C code is often bundled into libraries (collections of functions); obtaining a suitable library is a good way to reduce errors—and save considerable programming effort. Libraries for common

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 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 0.099999999999999987, thanks to rounding error.

Declarations

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

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

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

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

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

Notice that each complete declaration ends with a semicolon.

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

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

blocks ▶ 10.3

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

C99

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

Assignment

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

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

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

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

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

but not

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


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

```
profit = 2150.48;
```

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

```
profit = 2150.48f;
```

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

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

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

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

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

Printing the Value of a Variable

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

```
Height: h
```

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

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

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

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

```
Profit: $2150.48
```

we'd call `printf` as follows:

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

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

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

PROGRAM Computing the Dimensional Weight of a Box

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

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

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

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

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

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

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

int main(void)
{
```



```

int height, length, width, volume, weight;

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

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

return 0;
}

```

The output of the program is

```

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

```

Initialization

variable initialization ► 18.5

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



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

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

```
int height = 8;
```

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

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

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

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

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

Printing Expressions

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

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

could be replaced by

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

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

2.5 Reading Input

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

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

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

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

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

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

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

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

PROGRAM Computing the Dimensional Weight of a Box (Revisited)

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

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

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

#include <stdio.h>

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

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

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

    return 0;
}
```

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

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

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

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

2.6 Defining Names for Constants

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

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

```
#define INCHES_PER_POUND 166
```

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

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

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

will become

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

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

The value of a macro can be an expression:

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

parentheses in macros ► 14.3

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

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

PROGRAM Converting from Fahrenheit to Celsius

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

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

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

```
celsius.c /* Converts a Fahrenheit temperature to Celsius */
```

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

```
#define FREEZING_PT 32.0f
```

```
#define SCALE_FACTOR (5.0f / 9.0f)
```

```
int main(void)
```

```
{
```

```
    float fahrenheit, celsius;
```

```
    printf("Enter Fahrenheit temperature: ");
```



```

scanf("%f", &fahrenheit);
celsius = (fahrenheit - FREEZING_PT) * SCALE_FACTOR;
printf("Celsius equivalent: %.1f\n", celsius);
return 0;
}

```

The statement

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

converts the Fahrenheit temperature to Celsius. Since `FREEZING_PT` stands for `32.0f` and `SCALE_FACTOR` stands for `(5.0f / 9.0f)`, the compiler sees this statement as

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

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

The call of `printf` writes the Celsius temperature:

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

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

2.7 Identifiers

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

C99

universal character names ▶25.4

Here are some examples of legal identifiers:

```
times10  get_next_char  _done
```

The following are *not* legal identifiers:

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

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

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

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

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**C99**

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.

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.



restrictions on identifiers ▶21.1

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.

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`.
- *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.

Q&A

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

```
float fahrenheit, celsius;    /*** WRONG ***/
```

or

```
float 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:

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

C99 minate. In C89, the value returned to the operating system is undefined. In C99, if `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 color if it's accidentally included in a comment. A program such as `lint` can also help.

`lint` > 1.2

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 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.

C99

disabling code ► 14.4 In any event, there's a better way to disable portions of a program, as we'll see later.

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.

external linkage ► 18.2 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, `ABCDEFGH` and `abcdefgh` might be treated as the same name. (In C99, the first 31 characters are significant, and the case of letters is taken into account.)

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 $(\text{volume} + 165) / 166$ 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`

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

Section 2.8

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.
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, expr1, expr2, ...);
```

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)	<code>\a</code>
Backspace	<code>\b</code>
New line	<code>\n</code>
Horizontal tab	<code>\t</code>

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□
s s r s r r r s s s r r s s s s r r r r r r
```

`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.

Q&A

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:

```
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`.

detecting errors in `scanf` ▶ 22.3

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.)

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

- What output do the following calls of `printf` produce?
 - `printf("%6d,%4d", 86, 1040);`
 - `printf("%12.5e", 30.253);`
 - `printf("%.4f", 83.162);`
 - `printf("%-6.2g", .0000009979);`
- Write calls of `printf` that display a `float` variable `x` in the following formats.
 - Exponential notation; left-justified in a field of size 8; one digit after the decimal point.
 - Exponential notation; right-justified in a field of size 10; six digits after the decimal point.
 - Fixed decimal notation; left-justified in a field of size 8; three digits after the decimal point.
 - Fixed decimal notation; right-justified in a field of size 6; no digits after the decimal point.

Section 3.2

- 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.
 - `"%d"` versus `" %d"`
 - `"%d-%d-%d"` versus `"%d -%d -%d"`
 - `"%f"` versus `"%f "`
 - `"%f,%f"` versus `"%f, %f"`
- 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.)
- 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.)
- 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

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 Price	Purchase 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.

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 *GSI 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
GSI 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

<i>Unary</i>		<i>Binary</i>			
		<i>Additive</i>	<i>Multiplicative</i>		
+	unary plus	+	addition	*	multiplication
-	unary minus	-	subtraction	/	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.
- 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).

undefined behavior ▶ 4.4

Q&A

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:

<code>i + j * k</code>	is equivalent to	<code>i + (j * k)</code>
<code>-i * -j</code>	is equivalent to	<code>(-i) * (-j)</code>
<code>+i + j / k</code>	is equivalent to	<code>(+i) + (j / k)</code>

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

$$\begin{aligned} i - j - k & \text{ is equivalent to } (i - j) - k \\ i * j / k & \text{ is equivalent to } (i * j) / k \end{aligned}$$

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

$$- + i \quad \text{is equivalent to} \quad -(+i)$$

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

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

Q&A

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;
```

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

Q&A

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

<i>Precedence</i>	<i>Name</i>	<i>Symbol(s)</i>	<i>Associativity</i>
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]

A: The `%` operator requires integer operands. Try the `fmod` function instead. fmod function ▶ 23.3

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 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.

C99

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

A: Yes, indeed. An *lvalue* is an expression that can appear on the *left* side of an assignment; an *rvalue* 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.)
- $8 / 5$
 - $-8 / 5$
 - $8 / -5$
 - $-8 / -5$
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.)
- $8 \% 5$
 - $-8 \% 5$
 - $8 \% -5$
 - $-8 \% -5$
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.
- ```
i = 7; j = 8;
i *= j + 1;
printf("%d %d", i, j);
```
  - ```
i = j = k = 1;
i += j += k;
printf("%d %d %d", i, j, k);
```
 - ```
i = 1; j = 2; k = 3;
i -= j -= k;
printf("%d %d %d", i, j, k);
```
  - ```
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);
```

- Ⓜ 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 Güllüoğlu 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
<code><</code>	less than
<code>></code>	greater than
<code><=</code>	less than or equal to
<code>>=</code>	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 <i>and</i>
	logical <i>or</i>

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 non-zero.

- $expr1 \ || \ expr2$ 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 $(i \neq 0)$. 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 $0 \leq i < n$, 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:

<i>Transaction size</i>	<i>Commission rate</i>
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

Q&A

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

```
_Bool flag;
```

unsigned integer types ▶ 7.1

`_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:

```
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 */
    ...
```

`<stdbool.h>` header ▶ 21.5

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

```
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.

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. 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`:

labels ▶ 6.4

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

The compiler may not detect the error, since it assumes that `default` 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);`

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);`
- *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?
- \*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

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:

<i>Area code</i>	<i>Major city</i>
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.

- ④ 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.

- ④ 4. Here's a simplified version of the Beaufort scale, which is used to estimate wind force:

<i>Speed (knots)</i>	<i>Description</i>
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:

<i>Income</i>	<i>Amount of tax</i>
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.

- ④ 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:

<i>Departure time</i>	<i>Arrival time</i>
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 $13 \times 60 + 15 = 795$ 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
```

- Ⓜ 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.

6.1 The while Statement

Of all the ways to repeat a loop, the while loop is the simplest and most flexible. The while loop is summarized in the figure.

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.

Q&A

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:

```

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. 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).

linked lists ► 17.5

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`.

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 places an additional restriction on the `goto` statement: it can't be used to bypass the declaration of a variable-length array.)

C99

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);
```

- Ⓜ 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

- Ⓜ 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.

- Ⓜ 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.

- Ⓜ 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.

- Ⓜ 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:

$$e = 1 + 1/1! + 1/2! + 1/3! + \dots$$

Write a program that approximates e by computing the value of

$$1 + 1/1! + 1/2! + 1/3! + \dots + 1/n!$$

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.

Programming Projects

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

```
0111111111111111
```

which has the value 32,767 ($2^{15} - 1$). The largest 32-bit integer is

```
01111111111111111111111111111111
```

which has the value 2,147,483,647 ($2^{31} - 1$). An integer with no sign bit (the leftmost bit is considered part of the number's magnitude) is said to be **unsigned**. The largest 16-bit unsigned integer is 65,535 ($2^{16} - 1$), and the largest 32-bit unsigned integer is 4,294,967,295 ($2^{32} - 1$).

By default, integer variables are signed in C—the leftmost bit is reserved for the sign. To tell the compiler that a variable has no sign bit, we declare it to be **unsigned**. Unsigned numbers are primarily useful for systems programming and low-level, machine-dependent applications. We'll discuss typical applications for unsigned numbers in Chapter 20; until then, we'll generally avoid them.

C's integer types come in different sizes. The `int` type is usually 32 bits, but may be 16 bits on older CPUs. Since some programs require numbers that are too large to store in `int` form, C also provides **long** integers. At times, we may need to conserve memory by instructing the compiler to store a number in less space than normal; such a number is called a **short** integer.

To construct an integer type that exactly meets our needs, we can specify that a variable is long or short, signed or unsigned. We can even combine specifiers (e.g., long unsigned int). However, only the following six combinations actually produce different types:

```
short int
unsigned short int
int
unsigned int

long int
unsigned long int
```

Other combinations are synonyms for one of these six types. (For example, long signed int is the same as long int, since integers are always signed unless otherwise specified.) Incidentally, the order of the specifiers doesn't matter; unsigned short int is the same as short unsigned int.

C allows us to abbreviate the names of integer types by dropping the word `int`. For example, unsigned short int may be abbreviated to unsigned short, and long int may be abbreviated to just long. Omitting `int` is a widespread practice among C programmers, and some newer C-based languages (including Java) actually require the programmer to write short or long rather than short int or long int. For these reasons, I'll often omit the word `int` when it's not strictly necessary.

The range of values represented by each of the six integer types varies from one machine to another. However, there are a couple of rules that all compilers must obey. First, the C standard requires that `short int`, `int`, and `long int` each cover a certain minimum range of values (see Section 23.2 for details). Second, the standard requires that `int` not be shorter than `short int`, and `long int` not be shorter than `int`. However, it's possible that `short int` represents the same range of values as `int`; also, `int` may have the same range as `long int`.

Table 7.1 shows the usual range of values for the integer types on a 16-bit machine; note that `short int` and `int` have identical ranges.

Table 7.1
Integer Types on a
16-bit Machine

Type	Smallest Value	Largest Value
<code>short int</code>	-32,768	32,767
<code>unsigned short int</code>	0	65,535
<code>int</code>	-32,768	32,767
<code>unsigned int</code>	0	65,535
<code>long int</code>	-2,147,483,648	2,147,483,647
<code>unsigned long int</code>	0	4,294,967,295

Table 7.2 shows the usual ranges on a 32-bit machine; here `int` and `long int` have identical ranges.

Table 7.2
Integer Types on a
32-bit Machine

Type	Smallest Value	Largest Value
<code>short int</code>	-32,768	32,767
<code>unsigned short int</code>	0	65,535
<code>int</code>	-2,147,483,648	2,147,483,647
<code>unsigned int</code>	0	4,294,967,295
<code>long int</code>	-2,147,483,648	2,147,483,647
<code>unsigned long int</code>	0	4,294,967,295

In recent years, 64-bit CPUs have become more common. Table 7.3 shows typical ranges for the integer types on a 64-bit machine (especially under UNIX).

Table 7.3
Integer Types on a
64-bit Machine

Type	Smallest Value	Largest Value
<code>short int</code>	-32,768	32,767
<code>unsigned short int</code>	0	65,535
<code>int</code>	-2,147,483,648	2,147,483,647
<code>unsigned int</code>	0	4,294,967,295
<code>long int</code>	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
<code>unsigned long int</code>	0	18,446,744,073,709,551,615

Once more, let me emphasize that the ranges shown in Tables 7.1, 7.2, and 7.3 aren't mandated by the C standard and may vary from one compiler to another. One way to determine the ranges of the integer types for a particular implementation is to check the `<limits.h>` header, which is part of the standard library. This header defines macros that represent the smallest and largest values of each integer type.

C99 Integer Types in C99

C99 provides two additional standard integer types, `long long int` and `unsigned long long int`. These types were added because of the growing need for very large integers and the ability of newer processors to support 64-bit arithmetic. Both `long long` types are required to be at least 64 bits wide, so the range of `long long int` values is typically -2^{63} ($-9,223,372,036,854,775,808$) to $2^{63} - 1$ ($9,223,372,036,854,775,807$), and range of `unsigned long long int` values is usually 0 to $2^{64} - 1$ ($18,446,744,073,709,551,615$).

The `short int`, `int`, `long int`, and `long long int` types (along with the `signed char` type) are called *standard signed integer types* in C99. The `unsigned short int`, `unsigned int`, `unsigned long int`, and `unsigned long long int` types (along with the `unsigned char` type and the `_Bool` type) are called *standard unsigned integer types*.

In addition to the standard integer types, the C99 standard allows implementation-defined *extended integer types*, both signed and unsigned. For example, a compiler might provide signed and unsigned 128-bit integer types.

Integer Constants

Let's turn our attention to *constants*—numbers that appear in the text of a program, not numbers that are read, written, or computed. C allows integer constants to be written in decimal (base 10), octal (base 8), or hexadecimal (base 16).

Octal and Hexadecimal Numbers

An octal number is written using only the digits 0 through 7. Each position in an octal number represents a power of 8 (just as each position in a decimal number represents a power of 10). Thus, the octal number 237 represents the decimal number $2 \times 8^2 + 3 \times 8^1 + 7 \times 8^0 = 128 + 24 + 7 = 159$.

A hexadecimal (or hex) number is written using the digits 0 through 9 plus the letters A through F, which stand for 10 through 15, respectively. Each position in a hex number represents a power of 16; the hex number 1AF has the decimal value $1 \times 16^2 + 10 \times 16^1 + 15 \times 16^0 = 256 + 160 + 15 = 431$.

- **Decimal** constants contain digits between 0 and 9, but must not begin with a zero:

15 255 32767

- **Octal** constants contain only digits between 0 and 7, and *must* begin with a zero:

017 0377 077777

- **Hexadecimal** constants contain digits between 0 and 9 and letters between a and f, and always begin with 0x:

```
0xf 0xff 0x7fff
```

The letters in a hexadecimal constant may be either upper or lower case:

```
0xff 0xfF 0xFf 0xFF 0Xff 0XFf 0XFF 0XFF
```

Keep in mind that octal and hexadecimal are nothing more than an alternative way of writing numbers; they have no effect on how the numbers are actually stored. (Integers are always stored in binary, regardless of what notation we've used to express them.) We can switch from one notation to another at any time, and even mix them: `10 + 015 + 0x20` has the value 55 (decimal). Octal and hex are most convenient for writing low-level programs; we won't use these notations much until Chapter 20.

The type of a *decimal* integer constant is normally `int`. However, if the value of the constant is too large to store as an `int`, the constant has type `long int` instead. In the unlikely case that the constant is too large to store as a `long int`, the compiler will try `unsigned long int` as a last resort. The rules for determining the type of an *octal* or *hexadecimal* constant are slightly different: the compiler will go through the types `int`, `unsigned int`, `long int`, and `unsigned long int` until it finds one capable of representing the constant.

To force the compiler to treat a constant as a long integer, just follow it with the letter L (or l):

```
15L 0377L 0x7ffffL
```

To indicate that a constant is unsigned, put the letter U (or u) after it:

```
15U 0377U 0x7ffffU
```

L and U may be used in combination to show that a constant is both long *and* unsigned: `0xffffffffUL`. (The order of the L and U doesn't matter, nor does their case.)

C99 Integer Constants in C99

In C99, integer constants that end with either LL or ll (the case of the two letters must match) have type `long long int`. Adding the letter U (or u) before or after the LL or ll denotes a constant of type `unsigned long long int`.

C99's general rules for determining the type of an integer constant are a bit different from those in C89. The type of a decimal constant with no suffix (U, u, L, l, LL, or ll) is the "smallest" of the types `int`, `long int`, or `long long int` that can represent the value of that constant. For an octal or hexadecimal constant, however, the list of possible types is `int`, `unsigned int`, `long int`, `unsigned long int`, `long long int`, and `unsigned long long int`, in that order. Any suffix at the end of a constant changes the list of possible types. For

example, a constant that ends with U (or u) must have one of the types `unsigned int`, `unsigned long int`, or `unsigned long long int`. A decimal constant that ends with L (or l) must have one of the types `long int` or `long long int`. There's also a provision for a constant to have an extended integer type if it's too large to represent using one of the standard integer types.

Integer Overflow

When arithmetic operations are performed on integers, it's possible that the result will be too large to represent. For example, when an arithmetic operation is performed on two `int` values, the result must be able to be represented as an `int`. If the result can't be represented as an `int` (because it requires too many bits), we say that *overflow* has occurred.

The behavior when integer overflow occurs depends on whether the operands were signed or unsigned. When overflow occurs during an operation on *signed* integers, the program's behavior is undefined. Recall from Section 4.4 that the consequences of undefined behavior may vary. Most likely the result of the operation will simply be wrong, but the program could crash or exhibit other undesirable behavior.

When overflow occurs during an operation on *unsigned* integers, though, the result *is* defined: we get the correct answer modulo 2^n , where n is the number of bits used to store the result. For example, if we add 1 to the unsigned 16-bit number 65,535, the result is guaranteed to be 0.

Reading and Writing Integers

Suppose that a program isn't working because one of its `int` variables is overflowing. Our first thought is to change the type of the variable from `int` to `long int`. But we're not done yet; we need to see how the change will affect the rest of the program. In particular, we must check whether the variable is used in a call of `printf` or `scanf`. If so, the format string in the call will need to be changed, since the `%d` conversion works only for the `int` type.

Reading and writing unsigned, short, and long integers requires several new conversion specifiers:

Q&A

- When reading or writing an *unsigned* integer, use the letter `u`, `o`, or `x` instead of `d` in the conversion specification. If the `u` specifier is present, the number is read (or written) in decimal notation; `o` indicates octal notation, and `x` indicates hexadecimal notation.

```
unsigned int u;

scanf("%u", &u);    /* reads u in base 10 */
printf("%u", u);    /* writes u in base 10 */
scanf("%o", &u);    /* reads u in base 8 */
printf("%o", u);    /* writes u in base 8 */
```



```
scanf("%x", &u); /* reads u in base 16 */
printf("%x", u); /* writes u in base 16 */
```

- When reading or writing a *short* integer, put the letter *h* in front of *d*, *o*, *u*, or *x*:

```
short s;

scanf("%hd", &s);
printf("%hd", s);
```

- When reading or writing a *long* integer, put the letter *l* (“ell,” not “one”) in front of *d*, *o*, *u*, or *x*:

```
long l;

scanf("%ld", &l);
printf("%ld", l);
```

- **C99** When reading or writing a *long long* integer (C99 only), put the letters *ll* in front of *d*, *o*, *u*, or *x*:

```
long long ll;

scanf("%lld", &ll);
printf("%lld", ll);
```

PROGRAM Summing a Series of Numbers (Revisited)

In Section 6.1, we wrote a program that sums a series of integers entered by the user. One problem with this program is that the sum (or one of the input numbers) might exceed the largest value allowed for an `int` variable. Here’s what might happen if the program is run on a machine whose integers are 16 bits long:

```
This program sums a series of integers.
Enter integers (0 to terminate): 10000 20000 30000 0
The sum is: -5536
```

The sum was 60,000, which wouldn’t fit in an `int` variable, so overflow occurred. When overflow occurs with signed numbers, the outcome is undefined. In this case, we got an apparently meaningless number. To improve the program, let’s switch to `long` variables.

```
sum2.c /* Sums a series of numbers (using long variables) */

#include <stdio.h>

int main(void)
{
    long n, sum = 0;

    printf("This program sums a series of integers.\n");
```

```

printf("Enter integers (0 to terminate): ");

scanf("%ld", &n);
while (n != 0) {
    sum += n;
    scanf("%ld", &n);
}
printf("The sum is: %ld\n", sum);

return 0;
}

```

The change was fairly simple: we declared `n` and `sum` to be `long` variables instead of `int` variables, then we changed the conversion specifications in `scanf` and `printf` to `%ld` instead of `%d`.

7.2 Floating Types

The integer types aren't suitable for all applications. Sometimes we'll need variables that can store numbers with digits after the decimal point, or numbers that are exceedingly large or small. Numbers like these are stored in floating-point format (so called because the decimal point "floats"). C provides three *floating types*, corresponding to different floating-point formats:

<code>float</code>	Single-precision floating-point
<code>double</code>	Double-precision floating-point
<code>long double</code>	Extended-precision floating-point

`float` is suitable when the amount of precision isn't critical (calculating temperatures to one decimal point, for example). `double` provides greater precision—enough for most programs. `long double`, which supplies the ultimate in precision, is rarely used.

The C standard doesn't state how much precision the `float`, `double`, and `long double` types provide, since different computers may store floating-point numbers in different ways. Most modern computers follow the specifications in IEEE Standard 754 (also known as IEC 60559), so we'll use it as an example.

The IEEE Floating-Point Standard

IEEE Standard 754, developed by the Institute of Electrical and Electronics Engineers, provides two primary formats for floating-point numbers: single precision (32 bits) and double precision (64 bits). Numbers are stored in a form of scientific notation, with each number having three parts: a *sign*, an *exponent*, and a *fraction*. The number of bits reserved for the exponent determines how large (or small) numbers can be, while the number of bits in the fraction determines the precision. In single-precision format, the exponent is 8 bits long, while the fraction occupies 23

bits. As a result, a single-precision number has a maximum value of approximately 3.40×10^{38} , with a precision of about 6 decimal digits.

The IEEE standard also describes two other formats, single extended precision and double extended precision. The standard doesn't specify the number of bits in these formats, although it requires that the single extended type occupy at least 43 bits and the double extended type at least 79 bits. For more information about the IEEE standard and floating-point arithmetic in general, see "What every computer scientist should know about floating-point arithmetic" by David Goldberg (*ACM Computing Surveys*, vol. 23, no. 1 (March 1991): 5–48).

Table 7.4 shows the characteristics of the floating types when implemented according to the IEEE standard. (The table shows the smallest positive *normalized* values. Subnormal numbers can be smaller.) The long double type isn't shown in the table, since its length varies from one machine to another, with 80 bits and 128 bits being the most common sizes.

Table 7.4
Floating Type
Characteristics
(IEEE Standard)

Type	Smallest Positive Value	Largest Value	Precision
float	1.17549×10^{-38}	3.40282×10^{38}	6 digits
double	2.22507×10^{-308}	1.79769×10^{308}	15 digits

On computers that don't follow the IEEE standard, Table 7.4 won't be valid. In fact, on some machines, `float` may have the same set of values as `double`, or `double` may have the same values as `long double`. Macros that define the characteristics of the floating types can be found in the `<float.h>` header.

In C99, the floating types are divided into two categories. The `float`, `double`, and `long double` types fall into one category, called the *real floating types*. Floating types also include the *complex types* (`float _Complex`, `double _Complex`, and `long double _Complex`), which are new in C99.

Floating Constants

Floating constants can be written in a variety of ways. The following constants, for example, are all valid ways of writing the number 57.0:

```
57.0  57.  57.0e0  57E0  5.7e1  5.7e+1  .57e2  570.e-1
```

A floating constant must contain a decimal point and/or an exponent; the exponent indicates the power of 10 by which the number is to be scaled. If an exponent is present, it must be preceded by the letter E (or e). An optional + or - sign may appear after the E (or e).

By default, floating constants are stored as double-precision numbers. In other words, when a C compiler finds the constant 57.0 in a program, it arranges for the number to be stored in memory in the same format as a `double` variable. This rule generally causes no problems, since `double` values are converted automatically to `float` when necessary.

Q&A

subnormal numbers ▶ 23.4

`<float.h>` header ▶ 23.1

C99

complex types ▶ 27.3

On occasion, it may be necessary to force the compiler to store a floating constant in `float` or `long double` format. To indicate that only single precision is desired, put the letter `F` (or `f`) at the end of the constant (for example, `57.0F`). To indicate that a constant should be stored in `long double` format, put the letter `L` (or `l`) at the end (`57.0L`).

C99

C99 has a provision for writing floating constants in hexadecimal. Such a constant begins with `0x` or `0X` (like a hexadecimal integer constant). This feature is rarely used.

Q&A

Reading and Writing Floating-Point Numbers

As we've discussed, the conversion specifications `%e`, `%f`, and `%g` are used for reading and writing single-precision floating-point numbers. Values of types `double` and `long double` require slightly different conversions:

- When *reading* a value of type `double`, put the letter `l` in front of `e`, `f`, or `g`:

```
double d;
```

```
scanf("%lf", &d);
```

Q&A

Note: Use `l` only in a `scanf` format string, not a `printf` string. In a `printf` format string, the `e`, `f`, and `g` conversions can be used to write either `float` or `double` values. (C99 legalizes the use of `%le`, `%lf`, and `%lg` in calls of `printf`, although the `l` has no effect.)

C99

- When reading or writing a value of type `long double`, put the letter `L` in front of `e`, `f`, or `g`:

```
long double ld;
```

```
scanf("%Lf", &ld);
```

```
printf("%Lf", ld);
```

7.3 Character Types

Q&A

The only remaining basic type is `char`, the character type. The values of type `char` can vary from one computer to another, because different machines may have different underlying character sets.

Character Sets

ASCII character set ► Appendix E

Today's most popular character set is **ASCII** (American Standard Code for Information Interchange), a 7-bit code capable of representing 128 characters. In ASCII, the digits 0 to 9 are represented by the codes 0110000–0111001, and the uppercase letters A to Z are represented by 1000001–1011010. ASCII is often extended

to a 256-character code known as *Latin-1* that provides the characters necessary for Western European and many African languages.

A variable of type `char` can be assigned any single character:

```
char ch;

ch = 'a';    /* lower-case a */
ch = 'A';    /* upper-case A */
ch = '0';    /* zero          */
ch = ' ';    /* space          */
```

Notice that character constants are enclosed in single quotes, not double quotes.

Operations on Characters

Working with characters in C is simple, because of one fact: *C treats characters as small integers*. After all, characters are encoded in binary, and it doesn't take much imagination to view these binary codes as integers. In ASCII, for example, character codes range from 0000000 to 1111111, which we can think of as the integers from 0 to 127. The character 'a' has the value 97, 'A' has the value 65, '0' has the value 48, and ' ' has the value 32. The connection between characters and integers in C is so strong that character constants actually have `int` type rather than `char` type (an interesting fact, but not one that will often matter to us).

When a character appears in a computation, C simply uses its integer value. Consider the following examples, which assume the ASCII character set:

```
char ch;
int i;

i = 'a';    /* i is now 97    */
ch = 65;    /* ch is now 'A'   */
ch = ch + 1; /* ch is now 'B'   */
ch++;      /* ch is now 'C'   */
```

Characters can be compared, just as numbers can. The following `if` statement checks whether `ch` contains a lower-case letter; if so, it converts `ch` to upper case.

```
if ('a' <= ch && ch <= 'z')
    ch = ch - 'a' + 'A';
```

Comparisons such as `'a' <= ch` are done using the integer values of the characters involved. These values depend on the character set in use, so programs that use `<`, `<=`, `>`, and `>=` to compare characters may not be portable.

The fact that characters have the same properties as numbers has some advantages. For example, we can easily write a `for` statement whose control variable steps through all the upper-case letters:

```
for (ch = 'A'; ch <= 'Z'; ch++) ...
```

On the other hand, treating characters as numbers can lead to various programming errors that won't be caught by the compiler, and lets us write meaningless expressions such as `'a' * 'b' / 'c'`. It can also hamper portability, since our programs may be based on assumptions about the underlying character set. (Our `for` loop, for example, assumes that the letters from A to Z have consecutive codes.)

Signed and Unsigned Characters

Since C allows characters to be used as integers, it shouldn't be surprising that the `char` type—like the integer types—exists in both signed and unsigned versions. Signed characters normally have values between -128 and 127 , while unsigned characters have values between 0 and 255 .

The C standard doesn't specify whether ordinary `char` is a signed or an unsigned type; some compilers treat it as a signed type, while others treat it as an unsigned type. (Some even allow the programmer to select, via a compiler option, whether `char` should be signed or unsigned.)

Most of the time, we don't really care whether `char` is signed or unsigned. Once in a while, though, we do, especially if we're using a character variable to store a small integer. For this reason, C allows the use of the words `signed` and `unsigned` to modify `char`:

Q&A

```
signed char sch;
unsigned char uch;
```

portability tip

Don't assume that `char` is either signed or unsigned by default. If it matters, use `signed char` or `unsigned char` instead of `char`.

enumerated types ▶ 16.5

In light of the close relationship between characters and integers, C89 uses the term *integral types* to refer to both the integer types and the character types. Enumerated types are also integral types.

C99

`_Bool` type ▶ 5.2

C99 doesn't use the term “integral types.” Instead, it expands the meaning of “integer types” to include the character types and the enumerated types. C99's `_Bool` type is considered to be an unsigned integer type.

Arithmetic Types

The integer types and floating types are collectively known as *arithmetic types*. Here's a summary of the arithmetic types in C89, divided into categories and sub-categories:

- Integral types
 - `char`
 - Signed integer types (`signed char`, `short int`, `int`, `long int`)
 - Unsigned integer types (`unsigned char`, `unsigned short int`, `unsigned int`, `unsigned long int`)

- Enumerated types
- Floating types (`float`, `double`, `long double`)

C99 C99 has a more complicated hierarchy for its arithmetic types:

- Integer types
 - `char`
 - Signed integer types, both standard (`signed char`, `short int`, `int`, `long int`, `long long int`) and extended
 - Unsigned integer types, both standard (`unsigned char`, `unsigned short int`, `unsigned int`, `unsigned long int`, `unsigned long long int`, `_Bool`) and extended
 - Enumerated types
- Floating types
 - Real floating types (`float`, `double`, `long double`)
 - Complex types (`float _Complex`, `double _Complex`, `long double _Complex`)

Escape Sequences

A character constant is usually one character enclosed in single quotes, as we've seen in previous examples. However, certain special characters—including the new-line character—can't be written in this way, because they're invisible (non-printing) or because they can't be entered from the keyboard. So that programs can deal with every character in the underlying character set, C provides a special notation, the *escape sequence*.

There are two kinds of escape sequences: *character escapes* and *numeric escapes*. We saw a partial list of character escapes in Section 3.1; Table 7.5 gives the complete set.

Table 7.5
Character Escapes

<i>Name</i>	<i>Escape Sequence</i>
Alert (bell)	<code>\a</code>
Backspace	<code>\b</code>
Form feed	<code>\f</code>
New line	<code>\n</code>
Carriage return	<code>\r</code>
Horizontal tab	<code>\t</code>
Vertical tab	<code>\v</code>
Backslash	<code>\\</code>
Question mark	<code>\?</code>
Single quote	<code>\'</code>
Double quote	<code>\"</code>

Q&A

The `\a`, `\b`, `\f`, `\r`, `\t`, and `\v` escapes represent common ASCII control characters. The `\n` escape represents the ASCII line-feed character. The `\\` escape allows a character constant or string to contain the `\` character. The `\'` escape

Q&A

allows a character constant to contain the ' character, while the \" escape allows a string to contain the " character. The \? escape is rarely used.

Character escapes are handy, but they have a problem: the list of character escapes doesn't include all the nonprinting ASCII characters, just the most common. Character escapes are also useless for representing characters beyond the basic 128 ASCII characters. Numeric escapes, which can represent *any* character, are the solution to this problem.

To write a numeric escape for a particular character, first look up the character's octal or hexadecimal value in a table like the one in Appendix E. For example, the ASCII escape character (decimal value: 27) has the value 33 in octal and 1B in hex. Either of these codes can be used to write an escape sequence:

- An *octal escape sequence* consists of the \ character followed by an octal number with at most three digits. (This number must be representable as an unsigned character, so its maximum value is normally 377 octal.) For example, the escape character could be written \33 or \033. Octal numbers in escape sequences—unlike octal constants—don't have to begin with 0.
- A *hexadecimal escape sequence* consists of \x followed by a hexadecimal number. Although C places no limit on the number of digits in the hexadecimal number, it must be representable as an unsigned character (hence it can't exceed FF if characters are eight bits long). Using this notation, the escape character would be written \x1b or \x1B. The x must be in lower case, but the hex digits (such as b) can be upper or lower case.

When used as a character constant, an escape sequence must be enclosed in single quotes. For example, a constant representing the escape character would be written '\33' (or '\x1b'). Escape sequences tend to get a bit cryptic, so it's often a good idea to give them names using #define:

```
#define ESC '\33' /* ASCII escape character */
```

Escape sequences can be embedded in strings as well, as we saw in Section 3.1.

Escape sequences aren't the only special notations for representing characters. Trigraph sequences provide a way to represent the characters #, [, \,], ^, {, |, }, and ~, which may not be available on keyboards in some countries. C99 adds universal character names, which resemble escape sequences. Unlike escape sequences, however, universal character names are allowed in identifiers.

Character-Handling Functions

Earlier in this section, we saw how to write an if statement that converts a lower-case letter to upper-case:

```
if ('a' <= ch && ch <= 'z')
    ch = ch - 'a' + 'A';
```

This isn't the best method, though. A faster—and more portable—way to convert case is to call C's toupper library function:

trigraph sequences ▶ 25.3

C99

universal character names ▶ 25.4


```
ch = toupper(ch); /* converts ch to upper case */
```

When it's called, `toupper` checks whether its argument (`ch` in this case) is a lower-case letter. If so, it returns the corresponding upper-case letter. Otherwise, `toupper` returns the value of the argument. In our example, we've used the assignment operator to store the return value of `toupper` back into the `ch` variable, although we could just as easily have done something else with it—stored it in another variable, say, or tested it in an `if` statement:

```
if (toupper(ch) == 'A') ...
```

Programs that call `toupper` need to have the following `#include` directive at the top:

```
#include <ctype.h>
```

`toupper` isn't the only useful character-handling function in the C library. Section 23.5 describes them all and gives examples of their use.

Reading and Writing Characters using `scanf` and `printf`

The `%c` conversion specification allows `scanf` and `printf` to read and write single characters:

```
char ch;
```

```
scanf("%c", &ch); /* reads a single character */
printf("%c", ch); /* writes a single character */
```

`scanf` doesn't skip white-space characters before reading a character. If the next unread character is a space, then the variable `ch` in the previous example will contain a space after `scanf` returns. To force `scanf` to skip white space before reading a character, put a space in its format string just before `%c`:

```
scanf(" %c", &ch); /* skips white space, then reads ch */
```

Recall from Section 3.2 that a blank in a `scanf` format string means “skip zero or more white-space characters.”

Since `scanf` doesn't normally skip white space, it's easy to detect the end of an input line: check to see if the character just read is the new-line character. For example, the following loop will read and ignore all remaining characters in the current input line:

```
do {
    scanf("%c", &ch);
} while (ch != '\n');
```

When `scanf` is called the next time, it will read the first character on the next input line.

Reading and Writing Characters using `getchar` and `putchar`

Q&A

C provides other ways to read and write single characters. In particular, we can use the `getchar` and `putchar` functions instead of calling `scanf` and `printf`. `putchar` writes a single character:

```
putchar(ch);
```

Each time `getchar` is called, it reads one character, which it returns. In order to save this character, we must use assignment to store it in a variable:

```
ch = getchar(); /* reads a character and stores it in ch */
```

`getchar` actually returns an `int` value rather than a `char` value (the reason will be discussed in later chapters). As a result, it's not unusual for a variable to have type `int` rather than `char` if it will be used to store a character read by `getchar`. Like `scanf`, `getchar` doesn't skip white-space characters as it reads.

Using `getchar` and `putchar` (rather than `scanf` and `printf`) saves time when the program is executed. `getchar` and `putchar` are fast for two reasons. First, they're much simpler than `scanf` and `printf`, which are designed to read and write many kinds of data in a variety of formats. Second, `getchar` and `putchar` are usually implemented as macros for additional speed.

macros ▶ 14.3

`getchar` has another advantage over `scanf`: because it returns the character that it reads, `getchar` lends itself to various C idioms, including loops that search for a character or skip over all occurrences of a character. Consider the `scanf` loop that we used to skip the rest of an input line:

```
do {
    scanf("%c", &ch);
} while (ch != '\n');
```

Rewriting this loop using `getchar` gives us the following:

```
do {
    ch = getchar();
} while (ch != '\n');
```

Moving the call of `getchar` into the controlling expression allows us to condense the loop:

```
while ((ch = getchar()) != '\n')
    ;
```

This loop reads a character, stores it into the variable `ch`, then tests if `ch` is not equal to the new-line character. If the test succeeds, the loop body (which is empty) is executed, then the loop test is performed once more, causing a new character to be read. Actually, we don't even need the `ch` variable; we can just compare the return value of `getchar` with the new-line character:

idiom `while (getchar() != '\n') /* skips rest of line */
;`

The resulting loop is a well-known C idiom that's cryptic but worth learning.

`getchar` is useful in loops that skip characters as well as loops that search for characters. Consider the following statement, which uses `getchar` to skip an indefinite number of blank characters:

idiom `while ((ch = getchar()) == ' ') /* skips blanks */
;`

When the loop terminates, `ch` will contain the first nonblank character that `getchar` encountered.



Be careful if you mix `getchar` and `scanf` in the same program. `scanf` has a tendency to leave behind characters that it has “peeked” at but not read, including the new-line character. Consider what happens if we try to read a number first, then a character:

```
printf("Enter an integer: ");
scanf("%d", &i);
printf("Enter a command: ");
command = getchar();
```

The call of `scanf` will leave behind any characters that weren't consumed during the reading of `i`, including (but not limited to) the new-line character. `getchar` will fetch the first leftover character, which wasn't what we had in mind.

PROGRAM Determining the Length of a Message

To illustrate how characters are read, let's write a program that calculates the length of a message. After the user enters the message, the program displays the length:

```
Enter a message: Brevity is the soul of wit.
Your message was 27 character(s) long.
```

The length includes spaces and punctuation, but not the new-line character at the end of the message.

We'll need a loop whose body reads a character and increments a counter. The loop will terminate as soon as a new-line character turns up. We could use either `scanf` or `getchar` to read characters; most C programmers would choose `getchar`. Using a straightforward `while` loop, we might end up with the following program.

```

length.c /* Determines the length of a message */

#include <stdio.h>

int main(void)
{
    char ch;
    int len = 0;

    printf("Enter a message: ");
    ch = getchar();
    while (ch != '\n') {
        len++;
        ch = getchar();
    }
    printf("Your message was %d character(s) long.\n", len);

    return 0;
}

```

Recalling our discussion of idioms involving while loops and `getchar`, we realize that the program can be shortened:

```

length2.c /* Determines the length of a message */

#include <stdio.h>

int main(void)
{
    int len = 0;

    printf("Enter a message: ");
    while (getchar() != '\n')
        len++;
    printf("Your message was %d character(s) long.\n", len);

    return 0;
}

```

7.4 Type Conversion

Computers tend to be more restrictive than C when it comes to arithmetic. For a computer to perform an arithmetic operation, the operands must usually be of the same size (the same number of bits) and be stored in the same way. A computer may be able to add two 16-bit integers directly, but not a 16-bit integer and a 32-bit integer or a 32-bit integer and a 32-bit floating-point number.

C, on the other hand, allows the basic types to be mixed in expressions. We can combine integers, floating-point numbers, and even characters in a single expression. The C compiler may then have to generate instructions that convert

some operands to different types so that the hardware will be able to evaluate the expression. If we add a 16-bit `short` and a 32-bit `int`, for example, the compiler will arrange for the `short` value to be converted to 32 bits. If we add an `int` and a `float`, the compiler will arrange for the `int` to be converted to `float` format. This conversion is a little more complicated, since `int` and `float` values are stored in different ways.

Because the compiler handles these conversions automatically, without the programmer's involvement, they're known as *implicit conversions*. C also allows the programmer to perform *explicit conversions*, using the cast operator. I'll discuss implicit conversions first, postponing explicit conversions until later in the section. Unfortunately, the rules for performing implicit conversions are somewhat complex, primarily because C has so many different arithmetic types.

Implicit conversions are performed in the following situations:

- When the operands in an arithmetic or logical expression don't have the same type. (C performs what are known as the *usual arithmetic conversions*.)
- When the type of the expression on the right side of an assignment doesn't match the type of the variable on the left side.
- When the type of an argument in a function call doesn't match the type of the corresponding parameter.
- When the type of the expression in a `return` statement doesn't match the function's return type.

We'll discuss the first two cases now and save the others for Chapter 9.

The Usual Arithmetic Conversions

The usual arithmetic conversions are applied to the operands of most binary operators, including the arithmetic, relational, and equality operators. For example, let's say that `f` has type `float` and `i` has type `int`. The usual arithmetic conversions will be applied to the operands in the expression `f + i`, because their types aren't the same. Clearly it's safer to convert `i` to type `float` (matching `f`'s type) rather than convert `f` to type `int` (matching `i`'s type). An integer can always be converted to `float`; the worst that can happen is a minor loss of precision. Converting a floating-point number to `int`, on the other hand, would cost us the fractional part of the number. Worse still, we'd get a completely meaningless result if the original number were larger than the largest possible integer or smaller than the smallest integer.

The strategy behind the usual arithmetic conversions is to convert operands to the "narrowest" type that will safely accommodate both values. (Roughly speaking, one type is narrower than another if it requires fewer bytes to store.) The types of the operands can often be made to match by converting the operand of the narrower type to the type of the other operand (this act is known as *promotion*). Among the most common promotions are the *integral promotions*, which convert a character or short integer to type `int` (or to `unsigned int` in some cases).

We can divide the rules for performing the usual arithmetic conversions into two cases:

- **The type of either operand is a floating type.** Use the following diagram to promote the operand whose type is narrower:

```

long double
  ↑
double
  ↑
float

```

That is, if one operand has type `long double`, then convert the other operand to type `long double`. Otherwise, if one operand has type `double`, convert the other operand to type `double`. Otherwise, if one operand has type `float`, convert the other operand to type `float`. Note that these rules cover mixtures of integer and floating types: if one operand has type `long int`, for example, and the other has type `double`, the `long int` operand is converted to `double`.

- **Neither operand type is a floating type.** First perform integral promotion on both operands (guaranteeing that neither operand will be a character or short integer). Then use the following diagram to promote the operand whose type is narrower:

```

unsigned long int
  ↑
long int
  ↑
unsigned int
  ↑
int

```

There's one special case, but it occurs only when `long int` and `unsigned int` have the same length (32 bits, say). Under these circumstances, if one operand has type `long int` and the other has type `unsigned int`, both are converted to `unsigned long int`.



When a signed operand is combined with an unsigned operand, the signed operand is converted to an unsigned value. The conversion involves adding or subtracting a multiple of $n + 1$, where n is the largest representable value of the unsigned type. This rule can cause obscure programming errors.

Suppose that the `int` variable `i` has the value `-10` and the `unsigned int` variable `u` has the value `10`. If we compare `i` and `u` using the `<` operator, we might expect to get the result `1` (true). Before the comparison, however, `i` is converted to `unsigned int`. Since a negative number can't be represented as an unsigned integer, the converted value won't be `-10`. Instead, the value `4,294,967,296` is added (assuming that `4,294,967,295` is the largest `unsigned int` value), giving

a converted value of 4,294,967,286. The comparison `i < u` will therefore produce 0. Some compilers produce a warning message such as “*comparison between signed and unsigned*” when a program attempts to compare a signed number with an unsigned number.

Because of traps like this one, it’s best to use unsigned integers as little as possible and, especially, never mix them with signed integers.

The following example shows the usual arithmetic conversions in action:

```
char c;
short int s;
int i;
unsigned int u;
long int l;
unsigned long int ul;
float f;
double d;
long double ld;

i = i + c;      /* c is converted to int          */
i = i + s;      /* s is converted to int          */
u = u + i;      /* i is converted to unsigned int */
l = l + u;      /* u is converted to long int     */
ul = ul + l;   /* l is converted to unsigned long int */
f = f + ul;    /* ul is converted to float       */
d = d + f;     /* f is converted to double       */
ld = ld + d;   /* d is converted to long double  */
```

Conversion During Assignment

The usual arithmetic conversions don’t apply to assignment. Instead, C follows the simple rule that the expression on the right side of the assignment is converted to the type of the variable on the left side. If the variable’s type is at least as “wide” as the expression’s, this will work without a snag. For example:

```
char c;
int i;
float f;
double d;

i = c;   /* c is converted to int   */
f = i;   /* i is converted to float  */
d = f;   /* f is converted to double */
```

Other cases are problematic. Assigning a floating-point number to an integer variable drops the fractional part of the number:

```
int i;

i = 842.97;   /* i is now 842 */
i = -842.97; /* i is now -842 */
```

Q&A

Moreover, assigning a value to a variable of a narrower type will give a meaningless result (or worse) if the value is outside the range of the variable's type:

```
c = 10000;    /** WRONG **/
i = 1.0e20;  /** WRONG **/
f = 1.0e100; /** WRONG **/
```

A “narrowing” assignment may elicit a warning from the compiler or from tools such as `lint`.

It's a good idea to append the `f` suffix to a floating-point constant if it will be assigned to a `float` variable, as we've been doing since Chapter 2:

```
f = 3.14159f;
```

Without the suffix, the constant `3.14159` would have type `double`, possibly causing a warning message.

C99 **Implicit Conversions in C99**

`_Bool` type ▶ 5.2

The rules for implicit conversions in C99 are somewhat different from the rules in C89, primarily because C99 has additional types (`_Bool`, long long types, extended integer types, and complex types).

For the purpose of defining conversion rules, C99 gives each integer type an “integer conversion rank.” Here are the ranks from highest to lowest:

1. long long int, unsigned long long int
2. long int, unsigned long int
3. int, unsigned int
4. short int, unsigned short int
5. char, signed char, unsigned char
6. `_Bool`

For simplicity, I'm ignoring extended integer types and enumerated types.

In place of C89's integral promotions, C99 has “integer promotions,” which involve converting any type whose rank is less than `int` and `unsigned int` to `int` (provided that all values of the type can be represented using `int`) or else to `unsigned int`.

As in C89, the C99 rules for performing the usual arithmetic conversions can be divided into two cases:

- **The type of either operand is a floating type.** As long as neither operand has a complex type, the rules are the same as before. (The conversion rules for complex types will be discussed in Section 27.3.)
- **Neither operand type is a floating type.** First perform integer promotion on both operands. If the types of the two operands are now the same, the process ends. Otherwise, use the following rules, stopping at the first one that applies:
 - If both operands have signed types or both have unsigned types, convert the

operand whose type has lesser integer conversion rank to the type of the operand with greater rank.

- If the unsigned operand has rank greater or equal to the rank of the type of the signed operand, convert the signed operand to the type of the unsigned operand.
- If the type of the signed operand can represent all of the values of the type of the unsigned operand, convert the unsigned operand to the type of the signed operand.
- Otherwise, convert both operands to the unsigned type corresponding to the type of the signed operand.

Incidentally, all arithmetic types can be converted to `_Bool` type. The result of the conversion is 0 if the original value is 0; otherwise, the result is 1.

Casting

Although C's implicit conversions are convenient, we sometimes need a greater degree of control over type conversion. For this reason, C provides *casts*. A cast expression has the form

cast expression

```
( type-name ) expression
```

type-name specifies the type to which the expression should be converted.

The following example shows how to use a cast expression to compute the fractional part of a `float` value:

```
float f, frac_part;
frac_part = f - (int) f;
```

The cast expression `(int) f` represents the result of converting the value of `f` to type `int`. C's usual arithmetic conversions then require that `(int) f` be converted back to type `float` before the subtraction can be performed. The difference between `f` and `(int) f` is the fractional part of `f`, which was dropped during the cast.

Cast expressions enable us to document type conversions that would take place anyway:

```
i = (int) f; /* f is converted to int */
```

They also enable us to overrule the compiler and force it to do conversions that we want. Consider the following example:

```
float quotient;
int dividend, divisor;

quotient = dividend / divisor;
```

As it's now written, the result of the division—an integer—will be converted to float form before being stored in `quotient`. We probably want `dividend` and `divisor` converted to float *before* the division, though, so that we get a more exact answer. A cast expression will do the trick:

```
quotient = (float) dividend / divisor;
```

`divisor` doesn't need a cast, since casting `dividend` to float forces the compiler to convert `divisor` to float also.

Incidentally, C regards `(type-name)` as a unary operator. Unary operators have higher precedence than binary operators, so the compiler interprets

```
(float) dividend / divisor
```

as

```
((float) dividend) / divisor
```

If you find this confusing, note that there are other ways to accomplish the same effect:

```
quotient = dividend / (float) divisor;
```

or

```
quotient = (float) dividend / (float) divisor;
```

Casts are sometimes necessary to avoid overflow. Consider the following example:

```
long i;
int j = 1000;

i = j * j;    /* overflow may occur */
```

At first glance, this statement looks fine. The value of `j * j` is 1,000,000, and `i` is a long, so it can easily store values of this size, right? The problem is that when two `int` values are multiplied, the result will have `int` type. But `j * j` is too large to represent as an `int` on some machines, causing an overflow. Fortunately, using a cast avoids the problem:

```
i = (long) j * j;
```

Since the cast operator takes precedence over `*`, the first `j` is converted to `long` type, forcing the second `j` to be converted as well. Note that the statement

```
i = (long) (j * j);    /*** WRONG ***/
```

wouldn't work, since the overflow would already have occurred by the time of the cast.

7.5 Type Definitions

In Section 5.2, we used the `#define` directive to create a macro that could be used as a Boolean type:

```
#define BOOL int
```

Q&A

There's a better way to set up a Boolean type, though, using a feature known as a *type definition*:

```
typedef int Bool;
```

Notice that the name of the type being defined comes *last*. Note also that I've capitalized the word `Bool`. Capitalizing the first letter of a type name isn't required; it's just a convention that some C programmers employ.

Using `typedef` to define `Bool` causes the compiler to add `Bool` to the list of type names that it recognizes. `Bool` can now be used in the same way as the built-in type names—in variable declarations, cast expressions, and elsewhere. For example, we might use `Bool` to declare variables:

```
Bool flag; /* same as int flag; */
```

The compiler treats `Bool` as a synonym for `int`; thus, `flag` is really nothing more than an ordinary `int` variable.

Advantages of Type Definitions

Type definitions can make a program more understandable (assuming that the programmer has been careful to choose meaningful type names). For example, suppose that the variables `cash_in` and `cash_out` will be used to store dollar amounts. Declaring `Dollars` as

```
typedef float Dollars;
```

and then writing

```
Dollars cash_in, cash_out;
```

is more informative than just writing

```
float cash_in, cash_out;
```

Type definitions can also make a program easier to modify. If we later decide that `Dollars` should really be defined as `double`, all we need do is change the type definition:

```
typedef double Dollars;
```

The declarations of `Dollars` variables need not be changed. Without the type definition, we would need to locate all `float` variables that store dollar amounts (not necessarily an easy task) and change their declarations.

Type Definitions and Portability

Type definitions are an important tool for writing portable programs. One of the problems with moving a program from one computer to another is that types may have different ranges on different machines. If `i` is an `int` variable, an assignment like

```
i = 100000;
```

is fine on a machine with 32-bit integers, but will fail on a machine with 16-bit integers.

portability tip

For greater portability, consider using `typedef` to define new names for integer types.

Suppose that we're writing a program that needs variables capable of storing product quantities in the range 0–50,000. We could use `long` variables for this purpose (since they're guaranteed to be able to hold numbers up to at least 2,147,483,647), but we'd rather use `int` variables, since arithmetic on `int` values may be faster than operations on `long` values; also, `int` variables may take up less space.

Instead of using the `int` type to declare quantity variables, we can define our own "quantity" type:

```
typedef int Quantity;
```

and use this type to declare variables:

```
Quantity q;
```

When we transport the program to a machine with shorter integers, we'll change the definition of `Quantity`:

```
typedef long Quantity;
```

This technique doesn't solve all our problems, unfortunately, since changing the definition of `Quantity` may affect the way `Quantity` variables are used. At the very least, calls of `printf` and `scanf` that use `Quantity` variables will need to be changed, with `%d` conversion specifications replaced by `%ld`.

The C library itself uses `typedef` to create names for types that can vary from one C implementation to another; these types often have names that end with `_t`, such as `ptrdiff_t`, `size_t`, and `wchar_t`. The exact definitions of these types will vary, but here are some typical examples:


```
typedef long int ptrdiff_t;
typedef unsigned long int size_t;
typedef int wchar_t;
```

C99

<stdint.h> header ▶27.1

In C99, the <stdint.h> header uses typedef to define names for integer types with a particular number of bits. For example, `int32_t` is a signed integer type with exactly 32 bits. Using these types is an effective way to make programs more portable.

7.6 The sizeof Operator

The `sizeof` operator allows a program to determine how much memory is required to store values of a particular type. The value of the expression

`sizeof expression``sizeof (type-name)`

is an unsigned integer representing the number of bytes required to store a value belonging to *type-name*. `sizeof(char)` is always 1, but the sizes of the other types may vary. On a 32-bit machine, `sizeof(int)` is normally 4. Note that `sizeof` is a rather unusual operator, since the compiler itself can usually determine the value of a `sizeof` expression.

Q&A

The `sizeof` operator can also be applied to constants, variables, and expressions in general. If `i` and `j` are `int` variables, then `sizeof(i)` is 4 on a 32-bit machine, as is `sizeof(i + j)`. When applied to an expression—as opposed to a type—`sizeof` doesn't require parentheses; we could write `sizeof i` instead of `sizeof(i)`. However, parentheses may be needed anyway because of operator precedence. The compiler would interpret `sizeof i + j` as `(sizeof i) + j`, because `sizeof`—a unary operator—takes precedence over the binary `+` operator. To avoid problems, I always use parentheses in `sizeof` expressions.

Printing a `sizeof` value requires care, because the type of a `sizeof` expression is an implementation-defined type named `size_t`. In C89, it's best to convert the value of the expression to a known type before printing it. `size_t` is guaranteed to be an unsigned integer type, so it's safest to cast a `sizeof` expression to unsigned long (the largest of C89's unsigned types) and then print it using the `%lu` conversion:

```
printf("Size of int: %lu\n", (unsigned long) sizeof(int));
```

C99

In C99, the `size_t` type can be larger than unsigned long. However, the `printf` function in C99 is capable of displaying `size_t` values directly, without needing a cast. The trick is to use the letter `z` in the conversion specification, followed by one of the usual integer codes (typically `u`):

```
printf("Size of int: %zu\n", sizeof(int)); /* C99 only */
```


Q & A

Q: Section 7.1 says that `%o` and `%x` are used to write unsigned integers in octal and hex notation. How do I write ordinary (signed) integers in octal or hex? [p. 130]

A: You can use `%o` and `%x` to print a signed integer as long as its value isn't negative. These conversions cause `printf` to treat a signed integer as though it were unsigned; in other words, `printf` will assume that the sign bit is part of the number's magnitude. As long as the sign bit is 0, there's no problem. If the sign bit is 1, `printf` will print an unexpectedly large number.

Q: But what if the number is negative? How can I write it in octal or hex?

A: There's no direct way to print a negative number in octal or hex. Fortunately, the need to do so is pretty rare. You can, of course, test whether the number is negative and print a minus sign yourself:

```
if (i < 0)
    printf("-%x", -i);
else
    printf("%x", i);
```

Q: Why are floating constants stored in `double` form rather than `float` form? [p. 133]

A: For historical reasons, C gives preference to the `double` type; `float` is treated as a second-class citizen. Consider, for instance, the discussion of `float` in Kernighan and Ritchie's *The C Programming Language*: "The main reason for using `float` is to save storage in large arrays, or, less often, to save time on machines where double-precision arithmetic is particularly expensive." C originally mandated that all floating-point arithmetic be done in double precision. (C89 and C99 have no such requirement.)

***Q:** What do hexadecimal floating constants look like, and what are they good for? [p. 134]

A: A hexadecimal floating constant begins with `0x` or `0X` and must contain an exponent, which is preceded by the letter `P` (or `p`). The exponent may have a sign, and the constant may end with `f`, `F`, `l`, or `L`. The exponent is expressed in decimal, but represents a power of 2, not a power of 10. For example, `0x1.Bp3` represents the number $1.6875 \times 2^3 = 13.5$. The hex digit `B` corresponds to the bit pattern `1011`. The `B` occurs to the right of the period, so each 1 bit represents a negative power of 2. Summing these powers of 2 ($2^{-1} + 2^{-3} + 2^{-4}$) yields `.6875`.

Hexadecimal floating constants are primarily useful for specifying constants that require great precision (including mathematical constants such as e and π). Hex numbers have a precise binary representation, whereas a constant written in decimal may be subject to a tiny rounding error when converted to binary. Hexa-

decimal numbers are also useful for defining constants with extreme values, such as the values of the macros in the `<float.h>` header. These constants are easy to write in hex but difficult to write in decimal.

***Q: Why do we use `%lf` to read a `double` value but `%f` to print it? [p. 134]**

A: This is a tough question to answer. First, notice that `scanf` and `printf` are unusual functions in that they aren't restricted to a fixed number of arguments. We say that `scanf` and `printf` have variable-length argument lists. When functions with variable-length argument lists are called, the compiler arranges for `float` arguments to be converted automatically to type `double`. As a result, `printf` can't distinguish between `float` and `double` arguments. This explains why `%f` works for both `float` and `double` arguments in calls of `printf`.

variable-length argument lists
▶26.1

`scanf`, on the other hand, is passed a *pointer* to a variable. `%f` tells `scanf` to store a `float` value at the address passed to it, while `%lf` tells `scanf` to store a `double` value at that address. The distinction between `float` and `double` is crucial here. If given the wrong conversion specification, `scanf` will likely store the wrong number of bytes (not to mention the fact that the bit pattern for a `float` isn't the same as that for a `double`).

Q: What's the proper way to pronounce `char`? [p. 134]

A: There's no universally accepted pronunciation. Some people pronounce `char` in the same way as the first syllable of "character." Others say "char," as in

```
char broiled;
```

Q: When does it matter whether a character variable is signed or unsigned? [p. 136]

A: If we store only 7-bit characters in the variable, it doesn't matter, since the sign bit will be zero. If we plan to store 8-bit characters, however, we'll probably want the variable to have unsigned `char` type. Consider the following example:

```
ch = '\xdb';
```

If `ch` has been declared to have type `char`, the compiler may choose to treat it as a signed character (many compilers do). As long as `ch` is used only as a character, there won't be any problem. But if `ch` is ever used in a context that requires the compiler to convert its value to an integer, we're likely to have trouble: the resulting integer will be negative, since `ch`'s sign bit is 1.

Here's another situation: In some kinds of programs, it's customary to use `char` variables to store one-byte integers. If we're writing such a program, we'll have to decide whether each variable should be signed `char` or unsigned `char`, just as we must decide whether ordinary integer variables should have type `int` or unsigned `int`.

Q: I don't understand how the new-line character can be the ASCII line-feed character. When a user enters input and presses the Enter key, doesn't the program read this as a carriage-return character or a carriage return plus a line feed? [p. 137]

A: Nope. As part of C's UNIX heritage, it always regards the end of a line as being marked by a single line-feed character. (In UNIX text files, a single line-feed character—but no carriage return—appears at the end of each line.) The C library takes care of translating the user's keypress into a line-feed character. When a program reads from a file, the I/O library translates the file's end-of-line marker (whatever it may be) into a single line-feed character. The same transformations occur—in reverse—when output is written to the screen or to a file. (See Section 22.1 for details.)

Although these translations may seem confusing, they serve an important purpose: insulating programs from details that may vary from one operating system to another.

***Q: What's the purpose of the `\?` escape sequence? [p. 138]**

A: The `\?` escape is related to trigraph sequences, which begin with `??`. If you should put `??` in a string, there's a possibility that the compiler will mistake it for the beginning of a trigraph. Replacing the second `?` by `\?` fixes the problem.

trigraph sequences ▶ 25.3

Q: If `getchar` is faster, why would we ever want to use `scanf` to read individual characters? [p. 140]

A: Although it's not as fast as `getchar`, the `scanf` function is more flexible. As we saw previously, the `"%c"` format string causes `scanf` to read the next input character; `" %c"` causes it to read the next non-white-space character. Also, `scanf` is good at reading characters that are mixed in with other kinds of data. Let's say that our input data consists of an integer, then a single nonnumeric character, then another integer. By using the format string `"%d%c%d"`, we can get `scanf` to read all three items.

***Q: Under what circumstances do the integral promotions convert a character or short integer to unsigned int? [p. 143]**

A: The integral promotions yield an unsigned `int` if the `int` type isn't large enough to include all possible values of the original type. Since characters are usually eight bits long, they are almost always converted to `int`, which is guaranteed to be at least 16 bits long. Signed short integers can always be converted to `int` as well. Unsigned short integers are problematic. If short integers have the same length as ordinary integers (as they do on a 16-bit machine), then unsigned short integers will have to be converted to unsigned `int`, since the largest unsigned short integer (65,535 on a 16-bit machine) is larger than the largest `int` (32,767).

Q: Exactly what happens if I assign a value to a variable that's not large enough to hold it? [p. 146]

A: Roughly speaking, if the value is of an integral type and the variable is of an unsigned type, the extra bits are thrown away; if the variable has a signed type, the result is implementation-defined. Assigning a floating-point number to a variable—integer or floating—that's too small to hold it produces undefined behavior: anything can happen, including program termination.

***Q:** Why does C bother to provide type definitions? Isn't defining a `BOOL` macro just as good as defining a `Bool` type using `typedef`? [p. 149]

A: There are two important differences between type definitions and macro definitions. First, type definitions are more powerful than macro definitions. In particular, array and pointer types can't be defined as macros. Suppose that we try to use a macro to define a "pointer to integer" type:

```
#define PTR_TO_INT int *
```

The declaration

```
PTR_TO_INT p, q, r;
```

will become

```
int * p, q, r;
```

after preprocessing. Unfortunately, only `p` is a pointer; `q` and `r` are ordinary integer variables. Type definitions don't have this problem.

Second, `typedef` names are subject to the same scope rules as variables; a `typedef` name defined inside a function body wouldn't be recognized outside the function. Macro names, on the other hand, are replaced by the preprocessor wherever they appear.

***Q:** You said that compilers "can usually determine the value of a `sizeof` expression." Can't a compiler *always* determine the value of a `sizeof` expression? [p. 151]

A: In C89, yes. In C99, however, there's one exception. The compiler can't determine the size of a variable-length array, because the number of elements in the array may change during the execution of the program.

variable-length arrays ▶ 8.3

Exercises

Section 7.1

1. Give the decimal value of each of the following integer constants.
 - (a) 077
 - (b) 0x77
 - (c) 0XABC

Section 7.2

2. Which of the following are not legal constants in C? Classify each legal constant as either integer or floating-point.
 - (a) 010E2
 - (b) 32.1E+5
 - (c) 0790
 - (d) 100_000
 - (e) 3.978e-2

- Ⓜ 3. Which of the following are not legal types in C?
- (a) short unsigned int
 - (b) short float
 - (c) long double
 - (d) unsigned long

Section 7.3

- Ⓜ 4. If `c` is a variable of type `char`, which one of the following statements is illegal?
- (a) `i += c; /* i has type int */`
 - (b) `c = 2 * c - 1;`
 - (c) `putchar(c);`
 - (d) `printf(c);`
5. Which one of the following is not a legal way to write the number 65? (Assume that the character set is ASCII.)
- (a) `'A'`
 - (b) `0b1000001`
 - (c) `0101`
 - (d) `0x41`
6. For each of the following items of data, specify which one of the types `char`, `short`, `int`, or `long` is the smallest one guaranteed to be large enough to store the item.
- (a) Days in a month
 - (b) Days in a year
 - (c) Minutes in a day
 - (d) Seconds in a day
7. For each of the following character escapes, give the equivalent octal escape. (Assume that the character set is ASCII.) You may wish to consult Appendix E, which lists the numerical codes for ASCII characters.
- (a) `\b`
 - (b) `\n`
 - (c) `\r`
 - (d) `\t`
8. Repeat Exercise 7, but give the equivalent hexadecimal escape.

Section 7.4

9. Suppose that `i` and `j` are variables of type `int`. What is the type of the expression `i / j + 'a'`?
- Ⓜ 10. Suppose that `i` is a variable of type `int`, `j` is a variable of type `long`, and `k` is a variable of type `unsigned int`. What is the type of the expression `i + (int) j * k`?
11. Suppose that `i` is a variable of type `int`, `f` is a variable of type `float`, and `d` is a variable of type `double`. What is the type of the expression `i * f / d`?
- Ⓜ 12. Suppose that `i` is a variable of type `int`, `f` is a variable of type `float`, and `d` is a variable of type `double`. Explain what conversions take place during the execution of the following statement:
- ```
d = i + f;
```



13. Assume that a program contains the following declarations:

```
char c = '\1';
short s = 2;
int i = -3;
long m = 5;
float f = 6.5f;
double d = 7.5;
```

Give the value and the type of each expression listed below.

- (a) `c * i`      (c) `f / c`      (e) `f - d`  
 (b) `s + m`      (d) `d / s`      (f) `(int) f`

- Ⓜ 14. Does the following statement always compute the fractional part of `f` correctly (assuming that `f` and `frac_part` are `float` variables)?

```
frac_part = f - (int) f;
```

If not, what's the problem?

## Section 7.5

15. Use `typedef` to create types named `Int8`, `Int16`, and `Int32`. Define the types so that they represent 8-bit, 16-bit, and 32-bit integers on your machine.

## Programming Projects

- Ⓜ 1. The `square2.c` program of Section 6.3 will fail (usually by printing strange answers) if `i * i` exceeds the maximum `int` value. Run the program and determine the smallest value of `n` that causes failure. Try changing the type of `i` to `short` and running the program again. (Don't forget to update the conversion specifications in the call of `printf`!) Then try `long`. From these experiments, what can you conclude about the number of bits used to store integer types on your machine?
- Ⓜ 2. Modify the `square2.c` program of Section 6.3 so that it pauses after every 24 squares and displays the following message:
- ```
Press Enter to continue...
```
- After displaying the message, the program should use `getchar` to read a character. `getchar` won't allow the program to continue until the user presses the Enter key.
3. Modify the `sum2.c` program of Section 7.1 to sum a series of `double` values.
4. Write a program that translates an alphabetic phone number into numeric form:
- ```
Enter phone number: CALLATT
2255288
```
- (In case you don't have a telephone nearby, here are the letters on the keys: 2=ABC, 3=DEF, 4=GHI, 5=JKL, 6=MNO, 7=PRS, 8=TUV, 9=WXYZ.) If the original phone number contains nonalphabetic characters (digits or punctuation, for example), leave them unchanged:
- ```
Enter phone number: 1-800-COL-LECT
1-800-265-5328
```
- You may assume that any letters entered by the user are upper case.

5. In the SCRABBLE Crossword Game, players form words using small tiles, each containing a letter and a face value. The face value varies from one letter to another, based on the letter's rarity. (Here are the face values: 1: AEILNORSTU, 2: DG, 3: BCMP, 4: FHVWY, 5: K, 8: JX, 10: QZ.) Write a program that computes the value of a word by summing the values of its letters:

```
Enter a word: pitfall
Scrabble value: 12
```

Your program should allow any mixture of lower-case and upper-case letters in the word. *Hint:* Use the `toupper` library function.

6. Write a program that prints the values of `sizeof(int)`, `sizeof(short)`, `sizeof(long)`, `sizeof(float)`, `sizeof(double)` and `sizeof(long double)`.
7. Modify Programming Project 6 from Chapter 3 so that the user may add, subtract, multiply, or divide two fractions (by entering either +, -, *, or / between the fractions).
8. Modify Programming Project 8 from Chapter 5 so that the user enters a time using the 12-hour clock. The input will have the form *hours:minutes* followed by either A, P, AM, or PM (either lower-case or upper-case). White space is allowed (but not required) between the numerical time and the AM/PM indicator. Examples of valid input:

```
1:15P
1:15PM
1:15p
1:15pm
1:15 P
1:15 PM
1:15 p
1:15 pm
```

You may assume that the input has one of these forms; there is no need to test for errors.

9. Write a program that asks the user for a 12-hour time, then displays the time in 24-hour form:

```
Enter a 12-hour time: 9:11 PM
Equivalent 24-hour time: 21:11
```

See Programming Project 8 for a description of the input format.

10. Write a program that counts the number of vowels (*a, e, i, o, and u*) in a sentence:

```
Enter a sentence: And that's the way it is.
Your sentence contains 6 vowels.
```

11. Write a program that takes a first name and last name entered by the user and displays the last name, a comma, and the first initial, followed by a period:

```
Enter a first and last name: Lloyd Fosdick
Fosdick, L.
```

The user's input may contain extra spaces before the first name, between the first and last names, and after the last name.

12. Write a program that evaluates an expression:

```
Enter an expression: 1+2.5*3
Value of expression: 10.5
```


The operands in the expression are floating-point numbers; the operators are +, -, *, and /. The expression is evaluated from left to right (no operator takes precedence over any other operator).

13. Write a program that calculates the average word length for a sentence:

```
Enter a sentence: It was deja vu all over again.
Average word length: 3.4
```

For simplicity, your program should consider a punctuation mark to be part of the word to which it is attached. Display the average word length to one decimal place.

14. Write a program that uses Newton's method to compute the square root of a positive floating-point number:

```
Enter a positive number: 3
Square root: 1.73205
```

Let x be the number entered by the user. Newton's method requires an initial guess y for the square root of x (we'll use $y = 1$). Successive guesses are found by computing the average of y and x/y . The following table shows how the square root of 3 would be found:

x	y	x/y	Average of y and x/y
3	1	3	2
3	2	1.5	1.75
3	1.75	1.71429	1.73214
3	1.73214	1.73196	1.73205
3	1.73205	1.73205	1.73205

Note that the values of y get progressively closer to the true square root of x . For greater accuracy, your program should use variables of type `double` rather than `float`. Have the program terminate when the absolute value of the difference between the old value of y and the new value of y is less than the product of `.00001` and y . *Hint:* Call the `fabs` function to find the absolute value of a `double`. (You'll need to include the `<math.h>` header at the beginning of your program in order to use `fabs`.)

15. Write a program that computes the factorial of a positive integer:

```
Enter a positive integer: 6
Factorial of 6: 720
```

- Use a `short` variable to store the value of the factorial. What is the largest value of n for which the program correctly prints the factorial of n ?
- Repeat part (a), using an `int` variable instead.
- Repeat part (a), using a `long` variable instead.
- Repeat part (a), using a `long long` variable instead (if your compiler supports the `long long` type).
- Repeat part (a), using a `float` variable instead.
- Repeat part (a), using a `double` variable instead.
- Repeat part (a), using a `long double` variable instead.

In cases (e)–(g), the program will display a close approximation of the factorial, not necessarily the exact value.

8 Arrays

If a program manipulates a large amount of data, it does so in a small number of ways.

So far, the only variables we've seen are *scalar*: capable of holding a single data item. C also supports *aggregate* variables, which can store collections of values. There are two kinds of aggregates in C: arrays and structures. This chapter shows how to declare and use arrays, both one-dimensional (Section 8.1) and multidimensional (Section 8.2). Section 8.3 covers C99's variable-length arrays. The focus of the chapter is on one-dimensional arrays, which play a much bigger role in C than do multidimensional arrays. Later chapters (Chapter 12 in particular) provide additional information about arrays; Chapter 16 covers structures.

8.1 One-Dimensional Arrays

An *array* is a data structure containing a number of data values, all of which have the same type. These values, known as *elements*, can be individually selected by their position within the array.

The simplest kind of array has just one dimension. The elements of a one-dimensional array are conceptually arranged one after another in a single row (or column, if you prefer). Here's how we might visualize a one-dimensional array named *a*:



To declare an array, we must specify the *type* of the array's elements and the *number* of elements. For example, to declare that the array *a* has 10 elements of type `int`, we would write

```
int a[10];
```

constant expressions ▶5.3

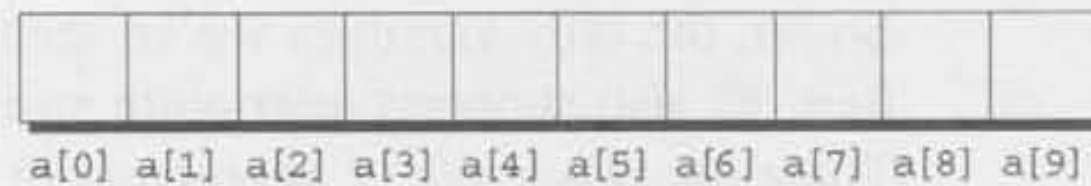
The elements of an array may be of any type; the length of the array can be specified by any (integer) constant expression. Since array lengths may need to be adjusted when the program is later changed, using a macro to define the length of an array is an excellent practice:

```
#define N 10
...
int a[N];
```

Array Subscripting

Q&A

To access a particular element of an array, we write the array name followed by an integer value in square brackets (this is referred to as *subscripting* or *indexing* the array). Array elements are always numbered starting from 0, so the elements of an array of length n are indexed from 0 to $n - 1$. For example, if a is an array with 10 elements, they're designated by $a[0]$, $a[1]$, ..., $a[9]$, as the following figure shows:



lvalues ▶4.2

Expressions of the form $a[i]$ are lvalues, so they can be used in the same way as ordinary variables:

```
a[0] = 1;
printf("%d\n", a[5]);
++a[i];
```

In general, if an array contains elements of type T , then each element of the array is treated as if it were a variable of type T . In this example, the elements $a[0]$, $a[5]$, and $a[i]$ behave like `int` variables.

Arrays and `for` loops go hand-in-hand. Many programs contain `for` loops whose job is to perform some operation on every element in an array. Here are a few examples of typical operations on an array a of length N :

```
idiom for (i = 0; i < N; i++)
        a[i] = 0;           /* clears a */

idiom for (i = 0; i < N; i++)
        scanf("%d", &a[i]); /* reads data into a */

idiom for (i = 0; i < N; i++)
        sum += a[i];       /* sums the elements of a */
```

Notice that we must use the `&` symbol when calling `scanf` to read an array element, just as we would with an ordinary variable.



C doesn't require that subscript bounds be checked; if a subscript goes out of range, the program's behavior is undefined. One cause of a subscript going out of bounds: forgetting that an array with n elements is indexed from 0 to $n - 1$, not 1 to n . (As one of my professors liked to say, "In this business, you're always off by one." He was right, of course.) The following example illustrates a bizarre effect that can be caused by this common blunder:

```
int a[10], i;

for (i = 1; i <= 10; i++)
    a[i] = 0;
```

With some compilers, this innocent-looking `for` statement causes an infinite loop! When `i` reaches 10, the program stores 0 into `a[10]`. But `a[10]` doesn't exist, so 0 goes into memory immediately after `a[9]`. If the variable `i` happens to follow `a[9]` in memory—as might be the case—then `i` will be reset to 0, causing the loop to start over.

An array subscript may be any integer expression:

```
a[i+j*10] = 0;
```

The expression can even have side effects:

```
i = 0;
while (i < N)
    a[i++] = 0;
```

Let's trace this code. After `i` is set to 0, the `while` statement checks whether `i` is less than `N`. If it is, 0 is assigned to `a[0]`, `i` is incremented, and the loop repeats. Note that `a[++i]` wouldn't be right, because 0 would be assigned to `a[1]` during the first loop iteration.



Be careful when an array subscript has a side effect. For example, the following loop—which is supposed to copy the elements of the array `b` into the array `a`—may not work properly:

```
i = 0;
while (i < N)
    a[i] = b[i++];
```

The expression `a[i] = b[i++]` accesses the value of `i` and also modifies `i` elsewhere in the expression, which—as we saw in Section 4.4—causes undefined behavior. Of course, we can easily avoid the problem by removing the increment from the subscript:

```
for (i = 0; i < N; i++)
    a[i] = b[i];
```

PROGRAM **Reversing a Series of Numbers**

Our first array program prompts the user to enter a series of numbers, then writes the numbers in reverse order:

```
Enter 10 numbers: 34 82 49 102 7 94 23 11 50 31
In reverse order: 31 50 11 23 94 7 102 49 82 34
```

Our strategy will be to store the numbers in an array as they're read, then go through the array backwards, printing the elements one by one. In other words, we won't actually reverse the elements in the array, but we'll make the user think we did.

```
reverse.c /* Reverses a series of numbers */

#include <stdio.h>

#define N 10

int main(void)
{
    int a[N], i;

    printf("Enter %d numbers: ", N);
    for (i = 0; i < N; i++)
        scanf("%d", &a[i]);

    printf("In reverse order:");
    for (i = N - 1; i >= 0; i--)
        printf(" %d", a[i]);
    printf("\n");

    return 0;
}
```

This program shows just how useful macros can be in conjunction with arrays. The macro `N` is used four times in the program: in the declaration of `a`, in the `printf` that displays a prompt, and in both `for` loops. Should we later decide to change the size of the array, we need only edit the definition of `N` and recompile the program. Nothing else will need to be altered; even the prompt will still be correct.

Array Initialization

An array, like any other variable, can be given an initial value at the time it's declared. The rules are somewhat tricky, though, so we'll cover some of them now and save others until later.

initializers ► 18.5

The most common form of *array initializer* is a list of constant expressions enclosed in braces and separated by commas:

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


If the initializer is *shorter* than the array, the remaining elements of the array are given the value 0:

```
int a[10] = {1, 2, 3, 4, 5, 6};
/* initial value of a is {1, 2, 3, 4, 5, 6, 0, 0, 0, 0} */
```

Using this feature, we can easily initialize an array to all zeros:

```
int a[10] = {0};
/* initial value of a is {0, 0, 0, 0, 0, 0, 0, 0, 0, 0} */
```

It's illegal for an initializer to be completely empty, so we've put a single 0 inside the braces. It's also illegal for an initializer to be *longer* than the array it initializes.

If an initializer is present, the length of the array may be omitted:

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

The compiler uses the length of the initializer to determine how long the array is. The array still has a fixed number of elements (10, in this example), just as if we had specified the length explicitly.

C99 Designated Initializers

It's often the case that relatively few elements of an array need to be initialized explicitly; the other elements can be given default values. Consider the following example:

```
int a[15] = {0, 0, 29, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 48};
```

We want element 2 of the array to be 29, element 9 to be 7, and element 14 to be 48, but the other values are just zero. For a large array, writing an initializer in this fashion is tedious and error-prone (what if there were 200 zeros between two of the nonzero values?).

C99's *designated initializers* can be used to solve this problem. Here's how we could redo the previous example using a designated initializer:

```
int a[15] = {[2] = 29, [9] = 7, [14] = 48};
```

Each number in brackets is said to be a *designator*.

Besides being shorter and easier to read (at least for some arrays), designated initializers have another advantage: the order in which the elements are listed no longer matters. Thus, our previous example could also be written in the following way:

```
int a[15] = {[14] = 48, [9] = 7, [2] = 29};
```

Designators must be integer constant expressions. If the array being initialized has length n , each designator must be between 0 and $n - 1$. However, if the length of the array is omitted, a designator can be any nonnegative integer. In the latter case, the compiler will deduce the length of the array from the largest designator.

In the following example, the fact that 23 appears as a designator will force the array to have length 24:

```
int b[] = {[5] = 10, [23] = 13, [11] = 36, [15] = 29};
```

An initializer may use both the older (element-by-element) technique and the newer (designated) technique:

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

Q&A

This initializer specifies that the array's first three elements will be 5, 1, and 9. Element 4 will have the value 3. The two elements after element 4 will be 7 and 2. Finally, element 8 will have the value 6. All elements for which no value is specified will default to zero.

PROGRAM Checking a Number for Repeated Digits

Our next program checks whether any of the digits in a number appear more than once. After the user enters a number, the program prints either *Repeated digit* or *No repeated digit*:

```
Enter a number: 28212
Repeated digit
```

The number 28212 has a repeated digit (2); a number like 9357 doesn't.

The program uses an array of Boolean values to keep track of which digits appear in a number. The array, named `digit_seen`, is indexed from 0 to 9 to correspond to the 10 possible digits. Initially, every element of the array is false. (The initializer for `digit_seen` is `{false}`, which only initializes the first element of the array. However, the compiler will automatically make the remaining elements zero, which is equivalent to false.)

When given a number `n`, the program examines `n`'s digits one at a time, storing each into the `digit` variable and then using it as an index into `digit_seen`. If `digit_seen[digit]` is true, then `digit` appears at least twice in `n`. On the other hand, if `digit_seen[digit]` is false, then `digit` has not been seen before, so the program sets `digit_seen[digit]` to true and keeps going.

```
repdigit.c /* Checks numbers for repeated digits */
```

```
#include <stdbool.h> /* C99 only */
#include <stdio.h>
```

```
int main(void)
{
    bool digit_seen[10] = {false};
    int digit;
    long n;

    printf("Enter a number: ");
    scanf("%ld", &n);
```



```

while (n > 0) {
    digit = n % 10;
    if (digit_seen[digit])
        break;
    digit_seen[digit] = true;
    n /= 10;
}

if (n > 0)
    printf("Repeated digit\n");
else
    printf("No repeated digit\n");

return 0;
}

```

C99

<stdbool.h> header ▶21.5

This program uses the names `bool`, `true`, and `false`, which are defined in C99's `<stdbool.h>` header. If your compiler doesn't support this header, you'll need to define these names yourself. One way to do so is to put the following lines above the main function:

```

#define true 1
#define false 0
typedef int bool;

```

Notice that `n` has type `long`, allowing the user to enter numbers up to 2,147,483,647 (or more, on some machines).

Using the `sizeof` Operator with Arrays

The `sizeof` operator can determine the size of an array (in bytes). If `a` is an array of 10 integers, then `sizeof(a)` is typically 40 (assuming that each integer requires four bytes).

We can also use `sizeof` to measure the size of an array element, such as `a[0]`. Dividing the array size by the element size gives the length of the array:

```
sizeof(a) / sizeof(a[0])
```

Some programmers use this expression when the length of the array is needed. To clear the array `a`, for example, we could write

```
for (i = 0; i < sizeof(a) / sizeof(a[0]); i++)
    a[i] = 0;
```

With this technique, the loop doesn't have to be modified if the array length should change at a later date. Using a macro to represent the array length has the same advantage, of course, but the `sizeof` technique is slightly better, since there's no macro name to remember (and possibly get wrong).

One minor annoyance is that some compilers produce a warning message for the expression `i < sizeof(a) / sizeof(a[0])`. The variable `i` probably has

type `int` (a signed type), whereas `sizeof` produces a value of type `size_t` (an unsigned type). We know from Section 7.4 that comparing a signed integer with an unsigned integer is a dangerous practice, although in this case it's safe because both `i` and `sizeof(a) / sizeof(a[0])` have nonnegative values. To avoid a warning, we can add a cast that converts `sizeof(a) / sizeof(a[0])` to a signed integer:

```
for (i = 0; i < (int) (sizeof(a) / sizeof(a[0])); i++)
    a[i] = 0;
```

Writing `(int) (sizeof(a) / sizeof(a[0]))` is a bit unwieldy; defining a macro that represents it is often helpful:

```
#define SIZE ((int) (sizeof(a) / sizeof(a[0])))

for (i = 0; i < SIZE; i++)
    a[i] = 0;
```

If we're back to using a macro, though, what's the advantage of `sizeof`? We'll answer that question in a later chapter (the trick is to add a parameter to the macro).

parameterized macros ► 14.3

PROGRAM Computing Interest

Our next program prints a table showing the value of \$100 invested at different rates of interest over a period of years. The user will enter an interest rate and the number of years the money will be invested. The table will show the value of the money at one-year intervals—at that interest rate and the next four higher rates—assuming that interest is compounded once a year. Here's what a session with the program will look like:

```
Enter interest rate: 6
Enter number of years: 5
```

Years	6%	7%	8%	9%	10%
1	106.00	107.00	108.00	109.00	110.00
2	112.36	114.49	116.64	118.81	121.00
3	119.10	122.50	125.97	129.50	133.10
4	126.25	131.08	136.05	141.16	146.41
5	133.82	140.26	146.93	153.86	161.05

Clearly, we can use a `for` statement to print the first row. The second row is a little trickier, since its values depend on the numbers in the first row. Our solution is to store the first row in an array as it's computed, then use the values in the array to compute the second row. Of course, this process can be repeated for the third and later rows. We'll end up with two `for` statements, one nested inside the other. The outer loop will count from 1 to the number of years requested by the user. The inner loop will increment the interest rate from its lowest value to its highest value.


```

interest.c /* Prints a table of compound interest */

#include <stdio.h>

#define NUM_RATES ((int) (sizeof(value) / sizeof(value[0])))
#define INITIAL_BALANCE 100.00

int main(void)
{
    int i, low_rate, num_years, year;
    double value[5];

    printf("Enter interest rate: ");
    scanf("%d", &low_rate);
    printf("Enter number of years: ");
    scanf("%d", &num_years);

    printf("\nYears");
    for (i = 0; i < NUM_RATES; i++) {
        printf("%6d%%", low_rate + i);
        value[i] = INITIAL_BALANCE;
    }
    printf("\n");

    for (year = 1; year <= num_years; year++) {
        printf("%3d    ", year);
        for (i = 0; i < NUM_RATES; i++) {
            value[i] += (low_rate + i) / 100.0 * value[i];
            printf("%7.2f", value[i]);
        }
        printf("\n");
    }

    return 0;
}

```

Note the use of `NUM_RATES` to control two of the `for` loops. If we later change the size of the `value` array, the loops will adjust automatically.

8.2 Multidimensional Arrays

An array may have any number of dimensions. For example, the following declaration creates a two-dimensional array (a *matrix*, in mathematical terminology):

```
int m[5][9];
```

The array `m` has 5 rows and 9 columns. Both rows and columns are indexed from 0, as the following figure shows:

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

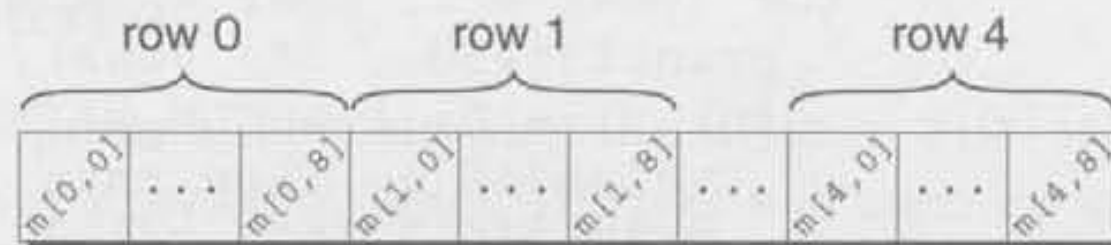
To access the element of m in row i , column j , we must write $m[i][j]$. The expression $m[i]$ designates row i of m , and $m[i][j]$ then selects element j in this row.



comma operator ▶ 6.3

Resist the temptation to write $m[i, j]$ instead of $m[i][j]$. C treats the comma as an operator in this context, so $m[i, j]$ is the same as $m[j]$.

Although we visualize two-dimensional arrays as tables, that's not the way they're actually stored in computer memory. C stores arrays in *row-major order*, with row 0 first, then row 1, and so forth. For example, here's how the m array is stored:



We'll usually ignore this detail, but sometimes it will affect our code.

Just as `for` loops go hand-in-hand with one-dimensional arrays, nested `for` loops are ideal for processing multidimensional arrays. Consider, for example, the problem of initializing an array for use as an identity matrix. (In mathematics, an *identity matrix* has 1's on the main diagonal, where the row and column index are the same, and 0's everywhere else.) We'll need to visit each element in the array in some systematic fashion. A pair of nested `for` loops—one that steps through every row index and one that steps through each column index—is perfect for the job:

```
#define N 10

double ident[N][N];
int row, col;

for (row = 0; row < N; row++)
    for (col = 0; col < N; col++)
        if (row == col)
            ident[row][col] = 1.0;
        else
            ident[row][col] = 0.0;
```

Multidimensional arrays play a lesser role in C than in many other programming languages, primarily because C provides a more flexible way to store multidimensional data: arrays of pointers.

arrays of pointers ▶ 13.7

Initializing a Multidimensional Array

We can create an initializer for a two-dimensional array by nesting one-dimensional initializers:

```
int m[5][9] = { {1, 1, 1, 1, 1, 0, 1, 1, 1},
                {0, 1, 0, 1, 0, 1, 0, 1, 0},
                {0, 1, 0, 1, 1, 0, 0, 1, 0},
                {1, 1, 0, 1, 0, 0, 0, 1, 0},
                {1, 1, 0, 1, 0, 0, 1, 1, 1} };
```

Each inner initializer provides values for one row of the matrix. Initializers for higher-dimensional arrays are constructed in a similar fashion.

C provides a variety of ways to abbreviate initializers for multidimensional arrays:

- If an initializer isn't large enough to fill a multidimensional array, the remaining elements are given the value 0. For example, the following initializer fills only the first three rows of `m`; the last two rows will contain zeros:

```
int m[5][9] = { {1, 1, 1, 1, 1, 0, 1, 1, 1},
                {0, 1, 0, 1, 0, 1, 0, 1, 0},
                {0, 1, 0, 1, 1, 0, 0, 1, 0} };
```

- If an inner list isn't long enough to fill a row, the remaining elements in the row are initialized to 0:

```
int m[5][9] = { {1, 1, 1, 1, 1, 0, 1, 1, 1},
                {0, 1, 0, 1, 0, 1, 0, 1},
                {0, 1, 0, 1, 1, 0, 0, 1},
                {1, 1, 0, 1, 0, 0, 0, 1},
                {1, 1, 0, 1, 0, 0, 1, 1, 1} };
```

- We can even omit the inner braces:

```
int m[5][9] = {1, 1, 1, 1, 1, 0, 1, 1, 1,
               0, 1, 0, 1, 0, 1, 0, 1, 0,
               0, 1, 0, 1, 1, 0, 0, 1, 0,
               1, 1, 0, 1, 0, 0, 0, 1, 0,
               1, 1, 0, 1, 0, 0, 1, 1, 1};
```

Once the compiler has seen enough values to fill one row, it begins filling the next.



Omitting the inner braces in a multidimensional array initializer can be risky, since an extra element (or even worse, a missing element) will affect the rest of the initializer. Leaving out the braces causes some compilers to produce a warning message such as “*missing braces around initializer.*”



C99's designated initializers work with multidimensional arrays. For example, we could create a 2×2 identity matrix as follows:

```
double ident[2][2] = {[0][0] = 1.0, [1][1] = 1.0};
```

As usual, all elements for which no value is specified will default to zero.

Constant Arrays

Any array, whether one-dimensional or multidimensional, can be made “constant” by starting its declaration with the word `const`:

```
const char hex_chars[] =
    {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
     'A', 'B', 'C', 'D', 'E', 'F'};
```

An array that’s been declared `const` should not be modified by the program; the compiler will detect direct attempts to modify an element.

Declaring an array to be `const` has a couple of primary advantages. It documents that the program won’t change the array, which can be valuable information for someone reading the code later. It also helps the compiler catch errors, by informing it that we don’t intend to modify the array.

`const` type qualifier ▶ 18.3

`const` isn’t limited to arrays; it works with any variable, as we’ll see later. However, `const` is particularly useful in array declarations, because arrays may contain reference information that won’t change during program execution.

PROGRAM Dealing a Hand of Cards

Our next program illustrates both two-dimensional arrays and constant arrays. The program deals a random hand from a standard deck of playing cards. (In case you haven’t had time to play games recently, each card in a standard deck has a *suit*—clubs, diamonds, hearts, or spades—and a *rank*—two, three, four, five, six, seven, eight, nine, ten, jack, queen, king, or ace.) We’ll have the user specify how many cards should be in the hand:

```
Enter number of cards in hand: 5
Your hand: 7c 2s 5d as 2h
```

It’s not immediately obvious how we’d write such a program. How do we pick cards randomly from the deck? And how do we avoid picking the same card twice? Let’s tackle these problems separately.

`time` function ▶ 26.3

`srand` function ▶ 26.2

`rand` function ▶ 26.2

To pick cards randomly, we’ll use several C library functions. The `time` function (from `<time.h>`) returns the current time, encoded in a single number. The `srand` function (from `<stdlib.h>`) initializes C’s random number generator. Passing the return value of `time` to `srand` prevents the program from dealing the same cards every time we run it. The `rand` function (also from `<stdlib.h>`) produces an apparently random number each time it’s called. By using the `%` operator, we can scale the return value from `rand` so that it falls between 0 and 3 (for suits) or between 0 and 12 (for ranks).

To avoid picking the same card twice, we’ll need to keep track of which cards have already been chosen. For that purpose, we’ll use an array named `in_hand`

that has four rows (one for each suit) and 13 columns (one for each rank). In other words, each element in the array corresponds to one of the 52 cards in the deck. All elements of the array will be false to start with. Each time we pick a card at random, we'll check whether the element of `in_hand` corresponding to that card is true or false. If it's true, we'll have to pick another card. If it's false, we'll store true in that card's array element to remind us later that this card has already been picked.

Once we've verified that a card is "new"—not already selected—we'll need to translate its numerical rank and suit into characters and then display the card. To translate the rank and suit to character form, we'll set up two arrays of characters—one for the rank and one for the suit—and then use the numbers to subscript the arrays. These arrays won't change during program execution, so we may as well declare them to be `const`.

```
deal.c /* Deals a random hand of cards */

#include <stdbool.h> /* C99 only */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define NUM_SUITS 4
#define NUM_RANKS 13

int main(void)
{
    bool in_hand[NUM_SUITS][NUM_RANKS] = {false};
    int num_cards, rank, suit;
    const char rank_code[] = {'2','3','4','5','6','7','8',
                              '9','t','j','q','k','a'};
    const char suit_code[] = {'c','d','h','s'};

    srand((unsigned) time(NULL));

    printf("Enter number of cards in hand: ");
    scanf("%d", &num_cards);

    printf("Your hand:");
    while (num_cards > 0) {
        suit = rand() % NUM_SUITS; /* picks a random suit */
        rank = rand() % NUM_RANKS; /* picks a random rank */
        if (!in_hand[suit][rank]) {
            in_hand[suit][rank] = true;
            num_cards--;
            printf(" %c%c", rank_code[rank], suit_code[suit]);
        }
    }
    printf("\n");

    return 0;
}
```

Notice the initializer for the `in_hand` array:

```
bool in_hand[NUM_SUITS][NUM_RANKS] = {false};
```

Even though `in_hand` is a two-dimensional array, we can use a single pair of braces (at the risk of possibly incurring a warning from the compiler). Also, we've supplied only one value in the initializer, knowing that the compiler will fill in 0 (false) for the other elements.

8.3 Variable-Length Arrays (C99)

Section 8.1 stated that the length of an array variable must be specified by a constant expression. In C99, however, it's sometimes possible to use an expression that's *not* constant. The following modification of the `reverse.c` program (Section 8.1) illustrates this ability:

```
reverse2.c /* Reverses a series of numbers using a variable-length
             array - C99 only */

#include <stdio.h>

int main(void)
{
    int i, n;

    printf("How many numbers do you want to reverse? ");
    scanf("%d", &n);

    int a[n]; /* C99 only - length of array depends on n */

    printf("Enter %d numbers: ", n);
    for (i = 0; i < n; i++)
        scanf("%d", &a[i]);

    printf("In reverse order:");
    for (i = n - 1; i >= 0; i--)
        printf(" %d", a[i]);
    printf("\n");

    return 0;
}
```

The array `a` in this program is an example of a *variable-length array* (or *VLA* for short). The length of a VLA is computed when the program is executed, not when the program is compiled. The chief advantage of a VLA is that the programmer doesn't have to pick an arbitrary length when declaring an array; instead, the program itself can calculate exactly how many elements are needed. If the programmer makes the choice, it's likely that the array will be too long (wasting memory) or too short (causing the program to fail). In the `reverse2.c` program, the num-

ber entered by the user determines the length of `a`; the programmer doesn't have to choose a fixed length, unlike in the original version of the program.

The length of a VLA doesn't have to be specified by a single variable. Arbitrary expressions, possibly containing operators, are also legal. For example:

```
int a[3*i+5];
int b[j+k];
```

Like other arrays, VLAs can be multidimensional:

```
int c[m][n];
```

static storage duration ► 18.2

The primary restriction on VLAs is that they can't have static storage duration. (We haven't yet seen any arrays with this property.) Another restriction is that a VLA may not have an initializer.

Variable-length arrays are most often seen in functions other than `main`. One big advantage of a VLA that belongs to a function `f` is that it can have a different length each time `f` is called. We'll explore this feature in Section 9.3.

Q & A

Q: Why do array subscripts start at 0 instead of 1? [p. 162]

A: Having subscripts begin at 0 simplifies the compiler a bit. Also, it can make array subscripting marginally faster.

Q: What if I want an array with subscripts that go from 1 to 10 instead of 0 to 9?

A: Here's a common trick: declare the array to have 11 elements instead of 10. The subscripts will go from 0 to 10, but you can just ignore element 0.

Q: Is it possible to use a character as an array subscript?

A: Yes, because C treats characters as integers. You'll probably need to "scale" the character before you use it as a subscript, though. Let's say that we want the `letter_count` array to keep track of a count for each letter in the alphabet. The array will need 26 elements, so we'd declare it in the following way:

```
int letter_count[26];
```

However, we can't use letters to subscript `letter_count` directly, because their integer values don't fall between 0 and 25. To scale a lower-case letter to the proper range, we can simply subtract `'a'`; to scale an upper-case letter, we'll subtract `'A'`. For example, if `ch` contains a lower-case letter, we'd write

```
letter_count[ch-'a'] = 0;
```

to clear the count that corresponds to `ch`. A minor caveat: this technique isn't completely portable, because it assumes that letters have consecutive codes. However, it works with most character sets, including ASCII.

Q: It seems like a designated initializer could end up initializing an array element more than once. Consider the following array declaration:

```
int a[] = {4, 9, 1, 8, [0] = 5, 7};
```

Is this declaration legal, and if so, what is the length of the array? [p. 166]

A: Yes, the declaration is legal. Here's how it works: as it processes an initializer list, the compiler keeps track of which array element is to be initialized next. Normally, the next element is the one following the element that was last initialized. However, when a designator appears in the list, it forces the next element be the one represented by the designator, *even if that element has already been initialized*.

Here's a step-by-step look at how the compiler will process the initializer for the array `a`:

- The 4 initializes element 0; the next element to be initialized is element 1.
- The 9 initializes element 1; the next element to be initialized is element 2.
- The 1 initializes element 2; the next element to be initialized is element 3.
- The 8 initializes element 3; the next element to be initialized is element 4.
- The `[0]` designator causes the next element to become 0, so the 5 initializes element 0 (replacing the 4 previously stored there). The next element to be initialized is element 1.
- The 7 initializes element 1 (replacing the 9 previously stored there). The next element to be initialized is element 2 (which is irrelevant since we're at the end of the list).

The net effect is the same as if we had written

```
int a[] = {5, 7, 1, 8};
```

Thus, the length of this array is four.

Q: The compiler gives me an error message if I try to copy one array into another by using the assignment operator. What's wrong?

A: Although it looks quite plausible, the assignment

```
a = b; /* a and b are arrays */
```

is indeed illegal. The reason for its illegality isn't obvious; it has to do with the peculiar relationship between arrays and pointers in C, a topic we'll explore in Chapter 12.

The simplest way to copy one array into another is to use a loop that copies the elements, one by one:

```
for (i = 0; i < N; i++)
    a[i] = b[i];
```

`memcpy` function ▶ 23.6

Another possibility is to use the `memcpy` ("memory copy") function from the `<string.h>` header. `memcpy` is a low-level function that simply copies bytes from one place to another. To copy the array `b` into the array `a`, use `memcpy` as follows:


```
memcpy(a, b, sizeof(a));
```

Many programmers prefer `memcpy`, especially for large arrays, because it's potentially faster than an ordinary loop.

***Q:** Section 6.4 mentioned that C99 doesn't allow a `goto` statement to bypass the declaration of a variable-length array. What's the reason for this restriction?

A: The memory used to store a variable-length array is usually allocated when the declaration of the array is reached during program execution. Bypassing the declaration using a `goto` statement could result in a program accessing the elements of an array that was never allocated.

Exercises

Section 8.1

1. We discussed using the expression `sizeof(a) / sizeof(a[0])` to calculate the number of elements in an array. The expression `sizeof(a) / sizeof(t)`, where `t` is the type of `a`'s elements, would also work, but it's considered an inferior technique. Why?
2. The Q&A section shows how to use a *letter* as an array subscript. Describe how to use a *digit* (in character form) as a subscript.
3. Write a declaration of an array named `weekend` containing seven `bool` values. Include an initializer that makes the first and last values `true`; all other values should be `false`.
4. (C99) Repeat Exercise 3, but this time use a designated initializer. Make the initializer as short as possible.
5. The Fibonacci numbers are 0, 1, 1, 2, 3, 5, 8, 13, ..., where each number is the sum of the two preceding numbers. Write a program fragment that declares an array named `fib_numbers` of length 40 and fills the array with the first 40 Fibonacci numbers. *Hint:* Fill in the first two numbers individually, then use a loop to compute the remaining numbers.

Section 8.2

6. Calculators, watches, and other electronic devices often rely on seven-segment displays for numerical output. To form a digit, such devices "turn on" some of the seven segments while leaving others "off":

Suppose that we want to set up an array that remembers which segments should be "on" for each digit. Let's number the segments as follows:

$$\begin{array}{c} 0 \\ 5 | \underline{6} | 1 \\ 4 | \underline{3} | 2 \end{array}$$

Here's what the array might look like, with each row representing one digit:

```
const int segments[10][7] = {{1, 1, 1, 1, 1, 1, 0}, ...};
```

I've given you the first row of the initializer; fill in the rest.

7. Using the shortcuts described in Section 8.2, shrink the initializer for the `segments` array (Exercise 6) as much as you can.
8. Write a declaration for a two-dimensional array named `temperature_readings` that stores one month of hourly temperature readings. (For simplicity, assume that a month has 30 days.) The rows of the array should represent days of the month; the columns should represent hours of the day.
9. Using the array of Exercise 8, write a program fragment that computes the average temperature for a month (averaged over all days of the month and all hours of the day).
10. Write a declaration for an 8×8 `char` array named `chess_board`. Include an initializer that puts the following data into the array (one character per array element):

```
r n b q k b n r
p p p p p p p p
. . . . .
. . . . .
. . . . .
P P P P P P P P
R N B Q K B N R
```

11. Write a program fragment that declares an 8×8 `char` array named `checker_board` and then uses a loop to store the following data into the array (one character per array element):

```
B R B R B R B R
R B R B R B R B
B R B R B R B R
R B R B R B R B
B R B R B R B R
R B R B R B R B
B R B R B R B R
R B R B R B R B
```

Hint: The element in row i , column j , should be the letter B if $i + j$ is an even number.

Programming Projects

- Modify the `repdigit.c` program of Section 8.1 so that it shows which digits (if any) were repeated:


```
Enter a number: 939577
Repeated digit(s): 7 9
```
- Modify the `repdigit.c` program of Section 8.1 so that it prints a table showing how many times each digit appears in the number:


```
Enter a number: 41271092
Digit:          0  1  2  3  4  5  6  7  8  9
Occurrences:    1  2  2  0  1  0  0  1  0  1
```
- Modify the `repdigit.c` program of Section 8.1 so that the user can enter more than one number to be tested for repeated digits. The program should terminate when the user enters a number that's less than or equal to 0.

4. Modify the `reverse.c` program of Section 8.1 to use the expression `(int) (sizeof(a) / sizeof(a[0]))` (or a macro with this value) for the array length.
- Ⓜ 5. Modify the `interest.c` program of Section 8.1 so that it compounds interest *monthly* instead of *annually*. The form of the output shouldn't change; the balance should still be shown at annual intervals.

6. The prototypical Internet newbie is a fellow named BIFF, who has a unique way of writing messages. Here's a typical BIFF communiqué:

H3Y DUD3, C 15 R1LLY C00L!!!!!!!!!!!!!!

Write a "BIFF filter" that reads a message entered by the user and translates it into BIFF-speak:

Enter message: Hey dude, C is rilly cool
 In BIFF-speak: H3Y DUD3, C 15 R1LLY C00L!!!!!!!!!!!!!!

Your program should convert the message to upper-case letters, substitute digits for certain letters (A→4, B→8, E→3, I→1, O→0, S→5), and then append 10 or so exclamation marks. *Hint:* Store the original message in an array of characters, then go back through the array, translating and printing characters one by one.

7. Write a program that reads a 5 × 5 array of integers and then prints the row sums and the column sums:

Enter row 1: 8 3 9 0 10
 Enter row 2: 3 5 17 1 1
 Enter row 3: 2 8 6 23 1
 Enter row 4: 15 7 3 2 9
 Enter row 5: 6 14 2 6 0

Row totals: 30 27 40 36 28
 Column totals: 34 37 37 32 21

- Ⓜ 8. Modify Programming Project 7 so that it prompts for five quiz grades for each of five students, then computes the total score and average score for each *student*, and the average score, high score, and low score for each *quiz*.

9. Write a program that generates a "random walk" across a 10 × 10 array. The array will contain characters (all '.' initially). The program must randomly "walk" from element to element, always going up, down, left, or right by one element. The elements visited by the program will be labeled with the letters A through Z, in the order visited. Here's an example of the desired output:

```
A . . . . .
B C D . . . . .
. F E . . . . .
H G . . . . .
I . . . . .
J . . . . . Z .
K . . R S T U V Y .
L M P Q . . . W X .
. N O . . . . .
. . . . .
```

Hint: Use the `srand` and `rand` functions (see `deal.c`) to generate random numbers. After generating a number, look at its remainder when divided by 4. There are four possible values for the remainder—0, 1, 2, and 3—indicating the direction of the next move. Before performing a move, check that (a) it won't go outside the array, and (b) it doesn't take us to

an element that already has a letter assigned. If either condition is violated, try moving in another direction. If all four directions are blocked, the program must terminate. Here's an example of premature termination:

```
A B G H I . . . . .
. C F . J K . . . . .
. D E . M L . . . . .
. . . . N O . . . . .
. . W X Y P Q . . . . .
. . V U T S R . . . . .
. . . . .
. . . . .
. . . . .
. . . . .
```

Y is blocked on all four sides, so there's no place to put Z.

10. Modify Programming Project 8 from Chapter 5 so that the departure times are stored in an array and the arrival times are stored in a second array. (The times are integers, representing the number of minutes since midnight.) The program will use a loop to search the array of departure times for the one closest to the time entered by the user.
11. Modify Programming Project 4 from Chapter 7 so that the program labels its output:

```
Enter phone number: 1-800-COL-LECT
In numeric form: 1-800-265-5328
```

The program will need to store the phone number (either in its original form or in its numeric form) in an array of characters until it can be printed. You may assume that the phone number is no more than 15 characters long.
12. Modify Programming Project 5 from Chapter 7 so that the SCRABBLE values of the letters are stored in an array. The array will have 26 elements, corresponding to the 26 letters of the alphabet. For example, element 0 of the array will store 1 (because the SCRABBLE value of the letter A is 1), element 1 of the array will store 3 (because the SCRABBLE value of the letter B is 3), and so forth. As each character of the input word is read, the program will use the array to determine the SCRABBLE value of that character. Use an array initializer to set up the array.
13. Modify Programming Project 11 from Chapter 7 so that the program labels its output:

```
Enter a first and last name: Lloyd Fosdick
You entered the name: Fosdick, L.
```

The program will need to store the last name (but not the first name) in an array of characters until it can be printed. You may assume that the last name is no more than 20 characters long.
14. Write a program that reverses the words in a sentence:

```
Enter a sentence: you can cage a swallow can't you?
Reversal of sentence: you can't swallow a cage can you?
```

Hint: Use a loop to read the characters one by one and store them in a one-dimensional char array. Have the loop stop at a period, question mark, or exclamation point (the "terminating character"), which is saved in a separate char variable. Then use a second loop to search backward through the array for the beginning of the last word. Print the last word, then search backward for the next-to-last word. Repeat until the beginning of the array is reached. Finally, print the terminating character.
15. One of the oldest known encryption techniques is the Caesar cipher, attributed to Julius Caesar. It involves replacing each letter in a message with another letter that is a fixed number of

positions later in the alphabet. (If the replacement would go past the letter Z, the cipher “wraps around” to the beginning of the alphabet. For example, if each letter is replaced by the letter two positions after it, then Y would be replaced by A, and Z would be replaced by B.) Write a program that encrypts a message using a Caesar cipher. The user will enter the message to be encrypted and the shift amount (the number of positions by which letters should be shifted):

```
Enter message to be encrypted: Go ahead, make my day.
Enter shift amount (1-25): 3
Encrypted message: Jr dkhdg, pdnh pb gdb.
```

Notice that the program can decrypt a message if the user enters 26 minus the original key:

```
Enter message to be encrypted: Jr dkhdg, pdnh pb gdb.
Enter shift amount (1-25): 23
Encrypted message: Go ahead, make my day.
```

You may assume that the message does not exceed 80 characters. Characters other than letters should be left unchanged. Lower-case letters remain lower-case when encrypted, and upper-case letters remain upper-case. *Hint:* To handle the wrap-around problem, use the expression $((ch - 'A') + n) \% 26 + 'A'$ to calculate the encrypted version of an upper-case letter, where *ch* stores the letter and *n* stores the shift amount. (You'll need a similar expression for lower-case letters.)

16. Write a program that tests whether two words are anagrams (permutations of the same letters):

```
Enter first word: smartest
Enter second word: mattress
The words are anagrams.
```

```
Enter first word: dumbest
Enter second word: stumble
The words are not anagrams.
```

Write a loop that reads the first word, character by character, using an array of 26 integers to keep track of how many times each letter has been seen. (For example, after the word *smartest* has been read, the array should contain the values 1 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 1 2 2 0 0 0 0 0, reflecting the fact that *smartest* contains one *a*, one *e*, one *m*, one *r*, two *s*'s and two *t*'s.) Use another loop to read the second word, except this time decrementing the corresponding array element as each letter is read. Both loops should ignore any characters that aren't letters, and both should treat upper-case letters in the same way as lower-case letters. After the second word has been read, use a third loop to check whether all the elements in the array are zero. If so, the words are anagrams. *Hint:* You may wish to use functions from `<ctype.h>`, such as `isalpha` and `tolower`.

17. Write a program that prints an $n \times n$ magic square (a square arrangement of the numbers 1, 2, ..., n^2 in which the sums of the rows, columns, and diagonals are all the same). The user will specify the value of *n*:

```
This program creates a magic square of a specified size.
The size must be an odd number between 1 and 99.
Enter size of magic square: 5
```

```
17  24  1   8  15
23  5   7  14  16
 4   6  13  20  22
10  12  19  21  3
11  18  25  2   9
```

Store the magic square in a two-dimensional array. Start by placing the number 1 in the middle of row 0. Place each of the remaining numbers 2, 3, ..., n^2 by moving up one row and over one column. Any attempt to go outside the bounds of the array should “wrap around” to the opposite side of the array. For example, instead of storing the next number in row -1 , we would store it in row $n - 1$ (the last row). Instead of storing the next number in column n , we would store it in column 0. If a particular array element is already occupied, put the number directly below the previously stored number. If your compiler supports variable-length arrays, declare the array to have n rows and n columns. If not, declare the array to have 99 rows and 99 columns.

9 Functions

If you have a procedure with ten parameters, you probably missed some.

We saw in Chapter 2 that a function is simply a series of statements that have been grouped together and given a name. Although the term “function” comes from mathematics, C functions don’t always resemble math functions. In C, a function doesn’t necessarily have arguments, nor does it necessarily compute a value. (In some programming languages, a “function” returns a value, whereas a “procedure” doesn’t. C lacks this distinction.)

Functions are the building blocks of C programs. Each function is essentially a small program, with its own declarations and statements. Using functions, we can divide a program into small pieces that are easier for us—and others—to understand and modify. Functions can take some of the tedium out of programming by allowing us to avoid duplicating code that’s used more than once. Moreover, functions are reusable: we can take a function that was originally part of one program and use it in others.

Our programs so far have consisted of just the `main` function. In this chapter, we’ll see how to write functions other than `main`, and we’ll learn more about `main` itself. Section 9.1 shows how to define and call functions. Section 9.2 then discusses function declarations and how they differ from function definitions. Next, Section 9.3 examines how arguments are passed to functions. The remainder of the chapter covers the `return` statement (Section 9.4), the related issue of program termination (Section 9.5), and recursion (Section 9.6).

9.1 Defining and Calling Functions

Before we go over the formal rules for defining a function, let’s look at three simple programs that define functions.

PROGRAM **Computing Averages**

Suppose we often need to compute the average of two `double` values. The C library doesn't have an "average" function, but we can easily define our own. Here's what it would look like:

```
double average(double a, double b)
{
    return (a + b) / 2;
}
```

Q&A

The word `double` at the beginning is `average`'s *return type*: the type of data that the function returns each time it's called. The identifiers `a` and `b` (the function's *parameters*) represent the two numbers that will be supplied when `average` is called. Each parameter must have a type (just like every variable has a type); in this example, both `a` and `b` have type `double`. (It may look odd, but the word `double` must appear twice, once for `a` and once for `b`.) A function parameter is essentially a variable whose initial value will be supplied later, when the function is called.

Every function has an executable part, called the *body*, which is enclosed in braces. The body of `average` consists of a single `return` statement. Executing this statement causes the function to "return" to the place from which it was called; the value of $(a + b) / 2$ will be the value returned by the function.

To call a function, we write the function name, followed by a list of *arguments*. For example, `average(x, y)` is a call of the `average` function. Arguments are used to supply information to a function; in this case, `average` needs to know which two numbers to average. The effect of the call `average(x, y)` is to copy the values of `x` and `y` into the parameters `a` and `b`, and then execute the body of `average`. An argument doesn't have to be a variable; any expression of a compatible type will do, allowing us to write `average(5.1, 8.9)` or `average(x/2, y/3)`.

We'll put the call of `average` in the place where we need to use the return value. For example, we could write

```
printf("Average: %g\n", average(x, y));
```

to compute the average of `x` and `y` and then print it. This statement has the following effect:

1. The `average` function is called with `x` and `y` as arguments.
2. `x` and `y` are copied into `a` and `b`.
3. `average` executes its `return` statement, returning the average of `a` and `b`.
4. `printf` prints the value that `average` returns. (The return value of `average` becomes one of `printf`'s arguments.)

Note that the return value of `average` isn't saved anywhere; the program prints it and then discards it. If we had needed the return value later in the program, we could have captured it in a variable:


```
avg = average(x, y);
```

This statement calls `average`, then saves its return value in the variable `avg`.

Now, let's use the `average` function in a complete program. The following program reads three numbers and computes their averages, one pair at a time:

```
Enter three numbers: 3.5 9.6 10.2
Average of 3.5 and 9.6: 6.55
Average of 9.6 and 10.2: 9.9
Average of 3.5 and 10.2: 6.85
```

Among other things, this program shows that a function can be called as often as we need.

```
average.c /* Computes pairwise averages of three numbers */

#include <stdio.h>

double average(double a, double b)
{
    return (a + b) / 2;
}

int main(void)
{
    double x, y, z;

    printf("Enter three numbers: ");
    scanf("%lf%lf%lf", &x, &y, &z);
    printf("Average of %g and %g: %g\n", x, y, average(x, y));
    printf("Average of %g and %g: %g\n", y, z, average(y, z));
    printf("Average of %g and %g: %g\n", x, z, average(x, z));

    return 0;
}
```

Notice that I've put the definition of `average` before `main`. We'll see in Section 9.2 that putting `average` after `main` causes problems.

PROGRAM Printing a Countdown

Not every function returns a value. For example, a function whose job is to produce output may not need to return anything. To indicate that a function has no return value, we specify that its return type is `void`. (`void` is a type with no values.) Consider the following function, which prints the message `T minus n` and `counting`, where `n` is supplied when the function is called:

```
void print_count(int n)
{
    printf("T minus %d and counting\n", n);
}
```

`print_count` has one parameter, `n`, of type `int`. It returns nothing, so I've specified `void` as the return type and omitted the `return` statement. Since `print_count` doesn't return a value, we can't call it in the same way we call `average`. Instead, a call of `print_count` must appear in a statement by itself:

```
print_count(i);
```

Here's a program that calls `print_count` 10 times inside a loop:

```
countdown.c /* Prints a countdown */

#include <stdio.h>

void print_count(int n)
{
    printf("T minus %d and counting\n", n);
}

int main(void)
{
    int i;

    for (i = 10; i > 0; --i)
        print_count(i);

    return 0;
}
```

Initially, `i` has the value 10. When `print_count` is called for the first time, `i` is copied into `n`, so that `n` takes on the value 10 as well. As a result, the first call of `print_count` will print

```
T minus 10 and counting
```

`print_count` then returns to the point at which it was called, which happens to be the body of a `for` statement. The `for` statement resumes where it left off, decrementing `i` to 9 and testing whether it's greater than 0. It is, so `print_count` is called again, this time printing

```
T minus 9 and counting
```

Each time `print_count` is called, `i` is different, so `print_count` will print 10 different messages.

PROGRAM Printing a Pun (Revisited)

Some functions have no parameters at all. Consider `print_pun`, which prints a bad pun each time it's called:

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


The word `void` in parentheses indicates that `print_pun` has no arguments. (This time, we're using `void` as a placeholder that means "nothing goes here.")

To call a function with no arguments, we write the function's name, followed by parentheses:

```
print_pun();
```

The parentheses *must* be present, even though there are no arguments.

Here's a tiny program that tests the `print_pun` function:

```
pun2.c /* Prints a bad pun */
#include <stdio.h>

void print_pun(void)
{
    printf("To C, or not to C: that is the question.\n");
}

int main(void)
{
    print_pun();
    return 0;
}
```

The execution of this program begins with the first statement in `main`, which happens to be a call of `print_pun`. When `print_pun` begins to execute, it in turn calls `printf` to display a string. When `printf` returns, `print_pun` returns to `main`.

Function Definitions

Now that we've seen several examples, let's look at the general form of a *function definition*:

function definition

```
return-type function-name ( parameters )
{
    declarations
    statements
}
```

The return type of a function is the type of value that the function returns. The following rules govern the return type:

- Functions may not return arrays, but there are no other restrictions on the return type.
- Specifying that the return type is `void` indicates that the function doesn't return a value.

C99

- If the return type is omitted in C89, the function is presumed to return a value of type `int`. In C99, it's illegal to omit the return type of a function.

As a matter of style, some programmers put the return type *above* the function name:

```
double
average(double a, double b)
{
    return (a + b) / 2;
}
```

Putting the return type on a separate line is especially useful if the return type is lengthy, like `unsigned long int`.

Q&A

After the function name comes a list of parameters. Each parameter is preceded by a specification of its type; parameters are separated by commas. If the function has no parameters, the word `void` should appear between the parentheses. *Note:* A separate type must be specified for each parameter, even when several parameters have the same type:

```
double average(double a, b)    /*** WRONG ***/
{
    return (a + b) / 2;
}
```

The body of a function may include both declarations and statements. For example, the `average` function could be written

```
double average(double a, double b)
{
    double sum;           /* declaration */

    sum = a + b;         /* statement */
    return sum / 2;      /* statement */
}
```

C99

Variables declared in the body of a function belong exclusively to that function; they can't be examined or modified by other functions. In C89, variable declarations must come first, before all statements in the body of a function. In C99, variable declarations and statements can be mixed, as long as each variable is declared prior to the first statement that uses the variable. (Some pre-C99 compilers also allow mixing of declarations and statements.)

The body of a function whose return type is `void` (which I'll call a "void function") can be empty:

```
void print_pun(void)
{
}
```

Leaving the body empty may make sense during program development; we can leave room for the function without taking the time to complete it, then come back later and write the body.

Function Calls

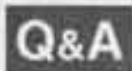
A function call consists of a function name followed by a list of arguments, enclosed in parentheses:

```
average(x, y)
print_count(i)
print_pun()
```



If the parentheses are missing, the function won't get called:

```
print_pun;    /* ** WRONG ** */
```



The result is a legal (albeit meaningless) expression statement that looks correct, but has no effect. Some compilers issue a warning such as “*statement with no effect.*”

A call of a `void` function is always followed by a semicolon to turn it into a statement:

```
print_count(i);
print_pun();
```

A call of a non-`void` function, on the other hand, produces a value that can be stored in a variable, tested, printed, or used in some other way:

```
avg = average(x, y);
if (average(x, y) > 0)
    printf("Average is positive\n");
printf("The average is %g\n", average(x, y));
```

The value returned by a non-`void` function can always be discarded if it's not needed:

```
average(x, y);    /* discards return value */
```

expression statements ▶ 4.5

This call of `average` is an example of an expression statement: a statement that evaluates an expression but then discards the result.

Ignoring the return value of `average` is an odd thing to do, but for some functions it makes sense. The `printf` function, for example, returns the number of characters that it prints. After the following call, `num_chars` will have the value 9:

```
num_chars = printf("Hi, Mom!\n");
```

Since we're probably not interested in the number of characters printed, we'll normally discard `printf`'s return value:

```
printf("Hi, Mom!\n");    /* discards return value */
```

To make it clear that we're deliberately discarding the return value of a function, C allows us to put `(void)` before the call:

```
(void) printf("Hi, Mom!\n");
```

casting ▶ 7.4

What we're doing is casting (converting) the return value of `printf` to type `void`. (In C, "casting to `void`" is a polite way of saying "throwing away.") Using `(void)` makes it clear to others that you deliberately discarded the return value, not just forgot that there was one. Unfortunately, there are a great many functions in the C library whose values are routinely ignored; using `(void)` when calling them all can get tiresome, so I haven't done so in this book.

PROGRAM Testing Whether a Number Is Prime

To see how functions can make programs easier to understand, let's write a program that tests whether a number is prime. The program will prompt the user to enter a number, then respond with a message indicating whether or not the number is prime:

```
Enter a number: 34
Not prime
```

Instead of putting the prime-testing details in `main`, we'll define a separate function that returns `true` if its parameter is a prime number and `false` if it isn't. When given a number `n`, the `is_prime` function will divide `n` by each of the numbers between 2 and the square root of `n`; if the remainder is ever 0, we know that `n` isn't prime.

```
prime.c /* Tests whether a number is prime */

#include <stdbool.h> /* C99 only */
#include <stdio.h>

bool is_prime(int n)
{
    int divisor;

    if (n <= 1)
        return false;
    for (divisor = 2; divisor * divisor <= n; divisor++)
        if (n % divisor == 0)
            return false;
    return true;
}

int main(void)
{
    int n;

    printf("Enter a number: ");
    scanf("%d", &n);
    if (is_prime(n))
        printf("Prime\n");
    else
        printf("Not prime\n");
}
```



```

    return 0;
}

```

Notice that `main` contains a variable named `n` even though `is_prime`'s parameter is also named `n`. In general, a function may declare a variable with the same name as a variable in another function. The two variables represent different locations in memory, so assigning a new value to one variable doesn't change the other. (This property extends to parameters as well.) Section 10.1 discusses this point in more detail.

As `is_prime` demonstrates, a function may have more than one `return` statement. However, we can execute just one of these statements during a given call of the function, because reaching a `return` statement causes the function to return to where it was called. We'll learn more about the `return` statement in Section 9.4.

9.2 Function Declarations

In the programs in Section 9.1, the definition of each function was always placed *above* the point at which it was called. In fact, C doesn't require that the definition of a function precede its calls. Suppose that we rearrange the `average.c` program by putting the definition of `average` *after* the definition of `main`:

```

#include <stdio.h>

int main(void)
{
    double x, y, z;

    printf("Enter three numbers: ");
    scanf("%lf%lf%lf", &x, &y, &z);
    printf("Average of %g and %g: %g\n", x, y, average(x, y));
    printf("Average of %g and %g: %g\n", y, z, average(y, z));
    printf("Average of %g and %g: %g\n", x, z, average(x, z));

    return 0;
}

double average(double a, double b)
{
    return (a + b) / 2;
}

```

When the compiler encounters the first call of `average` in `main`, it has no information about `average`: it doesn't know how many parameters `average` has, what the types of these parameters are, or what kind of value `average` returns. Instead of producing an error message, though, the compiler assumes that `average` returns an `int` value (recall from Section 9.1 that the return type of a

default argument promotions ▶ 9.3

function is `int` by default). We say that the compiler has created an *implicit declaration* of the function. The compiler is unable to check that we're passing average the right number of arguments and that the arguments have the proper type. Instead, it performs the default argument promotions and hopes for the best. When it encounters the definition of `average` later in the program, the compiler notices that the function's return type is actually `double`, not `int`, and so we get an error message.

One way to avoid the problem of call-before-definition is to arrange the program so that the definition of each function precedes all its calls. Unfortunately, such an arrangement doesn't always exist, and even when it does, it may make the program harder to understand by putting its function definitions in an unnatural order.

Fortunately, C offers a better solution: declare each function before calling it. A *function declaration* provides the compiler with a brief glimpse at a function whose full definition will appear later. A function declaration resembles the first line of a function definition with a semicolon added at the end:

function declaration

```
return-type function-name ( parameters ) ;
```

Needless to say, the declaration of a function must be consistent with the function's definition.

Q&A

Here's how our program would look with a declaration of `average` added:

```
#include <stdio.h>

double average(double a, double b);    /* DECLARATION */

int main(void)
{
    double x, y, z;

    printf("Enter three numbers: ");
    scanf("%lf%lf%lf", &x, &y, &z);
    printf("Average of %g and %g: %g\n", x, y, average(x, y));
    printf("Average of %g and %g: %g\n", y, z, average(y, z));
    printf("Average of %g and %g: %g\n", x, z, average(x, z));

    return 0;
}

double average(double a, double b)    /* DEFINITION */
{
    return (a + b) / 2;
}
```

Function declarations of the kind we've been discussing are known as *function prototypes* to distinguish them from an older style of function declaration in which the parentheses are left empty. A prototype provides a complete description

Q&A

of how to call a function: how many arguments to supply, what their types should be, and what type of result will be returned.

Incidentally, a function prototype doesn't have to specify the *names* of the function's parameters, as long as their *types* are present:

```
double average(double, double);
```

It's usually best not to omit parameter names, since they help document the purpose of each parameter and remind the programmer of the order in which arguments must appear when the function is called. However, there are legitimate reasons for omitting parameter names, and some programmers prefer to do so.

Q&A

C99

C99 has adopted the rule that either a declaration or a definition of a function must be present prior to any call of the function. Calling a function for which the compiler has not yet seen a declaration or definition is an error.

9.3 Arguments

Let's review the difference between a parameter and an argument. *Parameters* appear in function *definitions*; they're dummy names that represent values to be supplied when the function is called. *Arguments* are expressions that appear in function *calls*. When the distinction between *argument* and *parameter* isn't important, I'll sometimes use *argument* to mean either.

In C, arguments are *passed by value*: when a function is called, each argument is evaluated and its value assigned to the corresponding parameter. Since the parameter contains a copy of the argument's value, any changes made to the parameter during the execution of the function don't affect the argument. In effect, each parameter behaves like a variable that's been initialized to the value of the matching argument.

The fact that arguments are passed by value has both advantages and disadvantages. Since a parameter can be modified without affecting the corresponding argument, we can use parameters as variables within the function, thereby reducing the number of genuine variables needed. Consider the following function, which raises a number *x* to a power *n*:

```
int power(int x, int n)
{
    int i, result = 1;

    for (i = 1; i <= n; i++)
        result = result * x;

    return result;
}
```

Since *n* is a *copy* of the original exponent, we can modify it inside the function, thus removing the need for *i*:

```

int power(int x, int n)
{
    int result = 1;

    while (n-- > 0)
        result = result * x;

    return result;
}

```

Unfortunately, C's requirement that arguments be passed by value makes it difficult to write certain kinds of functions. For example, suppose that we need a function that will decompose a double value into an integer part and a fractional part. Since a function can't *return* two numbers, we might try passing a pair of variables to the function and having it modify them:

```

void decompose(double x, long int_part, double frac_part)
{
    int_part = (long) x; /* drops the fractional part of x */
    frac_part = x - int_part;
}

```

Suppose that we call the function in the following way:

```
decompose(3.14159, i, d);
```

At the beginning of the call, 3.14159 is copied into `x`, `i`'s value is copied into `int_part`, and `d`'s value is copied into `frac_part`. The statements inside `decompose` then assign 3 to `int_part` and .14159 to `frac_part`, and the function returns. Unfortunately, `i` and `d` weren't affected by the assignments to `int_part` and `frac_part`, so they have the same values after the call as they did before the call. With a little extra effort, `decompose` can be made to work, as we'll see in Section 11.4. However, we'll need to cover more of C's features first.

Argument Conversions

C allows function calls in which the types of the arguments don't match the types of the parameters. The rules governing how the arguments are converted depend on whether or not the compiler has seen a prototype for the function (or the function's full definition) prior to the call:

- **The compiler has encountered a prototype prior to the call.** The value of each argument is implicitly converted to the type of the corresponding parameter as if by assignment. For example, if an `int` argument is passed to a function that was expecting a `double`, the argument is converted to `double` automatically.
- **The compiler has not encountered a prototype prior to the call.** The compiler performs the *default argument promotions*: (1) `float` arguments are converted to `double`. (2) The integral promotions are performed, causing `char`

C99

and short arguments to be converted to `int`. (In C99, the integer promotions are performed.)



Relying on the default argument promotions is dangerous. Consider the following program:

```
#include <stdio.h>

int main(void)
{
    double x = 3.0;
    printf("Square: %d\n", square(x));

    return 0;
}

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

At the time `square` is called, the compiler hasn't seen a prototype yet, so it doesn't know that `square` expects an argument of type `int`. Instead, the compiler performs the default argument promotions on `x`, with no effect. Since it's expecting an argument of type `int` but has been given a `double` value instead, the effect of calling `square` is undefined. The problem can be fixed by casting `square`'s argument to the proper type:

```
printf("Square: %d\n", square((int) x));
```

C99

Of course, a much better solution is to provide a prototype for `square` before calling it. In C99, calling `square` without first providing a declaration or definition of the function is an error.

Array Arguments

Q&A

Arrays are often used as arguments. When a function parameter is a one-dimensional array, the length of the array can be (and is normally) left unspecified:

```
int f(int a[]) /* no length specified */
{
    ...
}
```

The argument can be any one-dimensional array whose elements are of the proper type. There's just one problem: how will `f` know how long the array is? Unfortunately, C doesn't provide any easy way for a function to determine the length of an array passed to it. Instead, we'll have to supply the length—if the function needs it—as an additional argument.



Although we can use the `sizeof` operator to help determine the length of an array *variable*, it doesn't give the correct answer for an array *parameter*:

```
int f(int a[])
{
    int len = sizeof(a) / sizeof(a[0]);
    /*** WRONG: not the number of elements in a ***/
    ...
}
```

Section 12.3 explains why.

The following function illustrates the use of one-dimensional array arguments. When given an array `a` of `int` values, `sum_array` returns the sum of the elements in `a`. Since `sum_array` needs to know the length of `a`, we must supply it as a second argument.

```
int sum_array(int a[], int n)
{
    int i, sum = 0;
    for (i = 0; i < n; i++)
        sum += a[i];
    return sum;
}
```

The prototype for `sum_array` has the following appearance:

```
int sum_array(int a[], int n);
```

As usual, we can omit the parameter names if we wish:

```
int sum_array(int [], int);
```

When `sum_array` is called, the first argument will be the name of an array, and the second will be its length. For example:

```
#define LEN 100

int main(void)
{
    int b[LEN], total;
    ...
    total = sum_array(b, LEN);
    ...
}
```

Notice that we don't put brackets after an array name when passing it to a function:

```
total = sum_array(b[], LEN);    /*** WRONG ***/
```


An important point about array arguments: A function has no way to check that we've passed it the correct array length. We can exploit this fact by telling the function that the array is smaller than it really is. Suppose that we've only stored 50 numbers in the `b` array, even though it can hold 100. We can sum just the first 50 elements by writing

```
total = sum_array(b, 50); /* sums first 50 elements */
```

`sum_array` will ignore the other 50 elements. (Indeed, it won't know that they even exist!)



Be careful not to tell a function that an array argument is *larger* than it really is:

```
total = sum_array(b, 150); /*** WRONG ***/
```

In this example, `sum_array` will go past the end of the array, causing undefined behavior.

Another important thing to know is that a function is allowed to change the elements of an array parameter, and the change is reflected in the corresponding argument. For example, the following function modifies an array by storing zero into each of its elements:

```
void store_zeros(int a[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        a[i] = 0;
}
```

The call

```
store_zeros(b, 100);
```

will store zero into the first 100 elements of the array `b`. This ability to modify the elements of an array argument may seem to contradict the fact that C passes arguments by value. In fact, there's no contradiction, but I won't be able to explain why until Section 12.3.

Q&A

If a parameter is a multidimensional array, only the length of the first dimension may be omitted when the parameter is declared. For example, if we revise the `sum_array` function so that `a` is a two-dimensional array, we must specify the number of columns in `a`, although we don't have to indicate the number of rows:

```
#define LEN 10
int sum_two_dimensional_array(int a[][LEN], int n)
{
    int i, j, sum = 0;
```

```

    for (i = 0; i < n; i++)
        for (j = 0; j < LEN; j++)
            sum += a[i][j];

    return sum;
}

```

arrays of pointers ► 13.7

Not being able to pass multidimensional arrays with an arbitrary number of columns can be a nuisance. Fortunately, we can often work around this difficulty by using arrays of pointers. C99's variable-length array parameters provide an even better solution to the problem.

C99 Variable-Length Array Parameters

variable-length arrays ► 8.3

C99 adds several new twists to array arguments. The first has to do with variable-length arrays (VLAs), a feature of C99 that allows the length of an array to be specified using a non-constant expression. Variable-length arrays can also be parameters, as it turns out.

Consider the `sum_array` function discussed earlier in this section. Here's the definition of `sum_array`, with the body omitted:

```

int sum_array(int a[], int n)
{
    ...
}

```

As it stands now, there's no direct link between `n` and the length of the array `a`. Although the function body treats `n` as `a`'s length, the actual length of the array could in fact be larger than `n` (or smaller, in which case the function won't work correctly).

Using a variable-length array parameter, we can explicitly state that `a`'s length is `n`:

```

int sum_array(int n, int a[n])
{
    ...
}

```

The value of the first parameter (`n`) specifies the length of the second parameter (`a`). Note that the order of the parameters has been switched; order is important when variable-length array parameters are used.



The following version of `sum_array` is illegal:

```

int sum_array(int a[n], int n)    /** WRONG ***/
{
    ...
}

```

The compiler will issue an error message at `int a[n]`, because it hasn't yet seen `n`.

There are several ways to write the prototype for our new version of `sum_array`. One possibility is to make it look exactly like the function definition:

```
int sum_array(int n, int a[n]); /* Version 1 */
```

Another possibility is to replace the array length by an asterisk (*):

```
int sum_array(int n, int a[*]); /* Version 2a */
```

The reason for using the `*` notation is that parameter names are optional in function declarations. If the name of the first parameter is omitted, it wouldn't be possible to specify that the length of the array is `n`, but the `*` provides a clue that the length of the array is related to parameters that come earlier in the list:

```
int sum_array(int, int [*]); /* Version 2b */
```

It's also legal to leave the brackets empty, as we normally do when declaring an array parameter:

```
int sum_array(int n, int a[]); /* Version 3a */
int sum_array(int, int []); /* Version 3b */
```

Leaving the brackets empty isn't a good choice, because it doesn't expose the relationship between `n` and `a`.

In general, the length of a variable-length array parameter can be any expression. For example, suppose that we were to write a function that concatenates two arrays `a` and `b` by copying the elements of `a`, followed by the elements of `b`, into a third array named `c`:

```
int concatenate(int m, int n, int a[m], int b[n], int c[m+n])
{
    ...
}
```

The length of `c` is the sum of the lengths of `a` and `b`. The expression used to specify the length of `c` involves two other parameters, but in general it could refer to variables outside the function or even call other functions.

Variable-length array parameters with a single dimension—as in all our examples so far—have limited usefulness. They make a function declaration or definition more descriptive by stating the desired length of an array argument. However, no additional error-checking is performed; it's still possible for an array argument to be too long or too short.

It turns out that variable-length array parameters are most useful for multidimensional arrays. Earlier in this section, we tried to write a function that sums the elements in a two-dimensional array. Our original function was limited to arrays with a fixed number of columns. If we use a variable-length array parameter, we can generalize the function to any number of columns:

```

int sum_two_dimensional_array(int n, int m, int a[n][m])
{
    int i, j, sum = 0;

    for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
            sum += a[i][j];

    return sum;
}

```

Prototypes for this function include the following:

```

int sum_two_dimensional_array(int n, int m, int a[n][m]);
int sum_two_dimensional_array(int n, int m, int a[*][*]);
int sum_two_dimensional_array(int n, int m, int a[][m]);
int sum_two_dimensional_array(int n, int m, int a[][*]);

```

C99 Using `static` in Array Parameter Declarations

C99 allows the use of the keyword `static` in the declaration of array parameters. (The keyword itself existed before C99. Section 18.2 discusses its traditional uses.)

In the following example, putting `static` in front of the number 3 indicates that the length of `a` is guaranteed to be at least 3:

```

int sum_array(int a[static 3], int n)
{
    ...
}

```

Using `static` in this way has no effect on the behavior of the program. The presence of `static` is merely a “hint” that may allow a C compiler to generate faster instructions for accessing the array. (If the compiler knows that an array will always have a certain minimum length, it can arrange to “prefetch” these elements from memory when the function is called, before the elements are actually needed by statements within the function.)

One last note about `static`: If an array parameter has more than one dimension, `static` can be used only in the first dimension (for example, when specifying the number of rows in a two-dimensional array).

C99 Compound Literals

Let’s return to the original `sum_array` function one last time. When `sum_array` is called, the first argument is usually the name of an array (the one whose elements are to be summed). For example, we might call `sum_array` in the following way:

```

int b[] = {3, 0, 3, 4, 1};
total = sum_array(b, 5);

```


The only problem with this arrangement is that `b` must be declared as a variable and then initialized prior to the call. If `b` isn't needed for any other purpose, it can be mildly annoying to create it solely for the purpose of calling `sum_array`.

In C99, we can avoid this annoyance by using a *compound literal*: an unnamed array that's created "on the fly" by simply specifying which elements it contains. The following call of `sum_array` has a compound literal (shown in **bold**) as its first argument:

```
total = sum_array((int []){3, 0, 3, 4, 1}, 5);
```

In this example, the compound literal creates an array containing the five integers 3, 0, 3, 4, and 1. We didn't specify the length of the array, so it's determined by the number of elements in the literal. We also have the option of specifying a length explicitly: `(int [4]) {1, 9, 2, 1}` is equivalent to `(int []) {1, 9, 2, 1}`.

In general, a compound literal consists of a type name within parentheses, followed by a set of values enclosed by braces. A compound literal resembles a cast applied to an initializer. In fact, compound literals and initializers obey the same rules. A compound literal may contain designators, just like a designated initializer, and it may fail to provide full initialization (in which case any uninitialized elements default to zero). For example, the literal `(int [10]) {8, 6}` has 10 elements; the first two have the values 8 and 6, and the remaining elements have the value 0.

Compound literals created inside a function may contain arbitrary expressions, not just constants. For example, we could write

```
total = sum_array((int []){2 * i, i + j, j * k}, 3);
```

where `i`, `j`, and `k` are variables. This aspect of compound literals greatly enhances their usefulness.

A compound literal is an lvalue, so the values of its elements can be changed. If desired, a compound literal can be made "read-only" by adding the word `const` to its type, as in `(const int []) {5, 4}`.

designated initializers ▶ 8.1

lvalues ▶ 4.2

9.4 The return Statement

A non-void function must use the `return` statement to specify what value it will return. The `return` statement has the form

return statement

```
return expression ;
```

The expression is often just a constant or variable:

```
return 0;
return status;
```

conditional operator ▶5.2

More complex expressions are possible. For example, it's not unusual to see the conditional operator used in a return expression:

```
return n >= 0 ? n : 0;
```

When this statement is executed, the expression `n >= 0 ? n : 0` is evaluated first. The statement returns the value of `n` if it's not negative; otherwise, it returns 0.

If the type of the expression in a return statement doesn't match the function's return type, the expression will be implicitly converted to the return type. For example, if a function is declared to return an `int`, but the return statement contains a `double` expression, the value of the expression is converted to `int`.

return statements may appear in functions whose return type is `void`, provided that no expression is given:

```
return; /* return in a void function */
```

Q&A

Putting an expression in such a return statement will get you a compile-time error. In the following example, the return statement causes the function to return immediately when given a negative argument:

```
void print_int(int i)
{
    if (i < 0)
        return;
    printf("%d", i);
}
```

If `i` is less than 0, `print_int` will return without calling `printf`.

A return statement may appear at the end of a void function:

```
void print_pun(void)
{
    printf("To C, or not to C: that is the question.\n");
    return; /* OK, but not needed */
}
```

Using `return` is unnecessary, though, since the function will return automatically after its last statement has been executed.

If a non-void function reaches the end of its body—that is, it fails to execute a return statement—the behavior of the program is undefined if it attempts to use the value returned by the function. Some compilers will issue a warning such as “*control reaches end of non-void function*” if they detect the possibility of a non-void function “falling off” the end of its body.

9.5 Program Termination

Since `main` is a function, it must have a return type. Normally, the return type of `main` is `int`, which is why the programs we've seen so far have defined `main` in the following way:


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

Older C programs often omit `main`'s return type, taking advantage of the fact that it traditionally defaults to `int`:

```
main()
{
    ...
}
```

C99 Omitting the return type of a function isn't legal in C99, so it's best to avoid this practice. Omitting the word `void` in `main`'s parameter list remains legal, but—as a matter of style—it's best to be explicit about the fact that `main` has no parameters. (We'll see later that `main` sometimes *does* have two parameters, usually named `argc` and `argv`.)

`argc` and `argv` ▶ 13.7

Q&A The value returned by `main` is a status code that—in some operating systems—can be tested when the program terminates. `main` should return 0 if the program terminates normally; to indicate abnormal termination, `main` should return a value other than 0. (Actually, there's no rule to prevent us from using the return value for other purposes.) It's good practice to make sure that every C program returns a status code, even if there are no plans to use it, since someone running the program later may decide to test it.

The `exit` Function

Executing a `return` statement in `main` is one way to terminate a program. Another is calling the `exit` function, which belongs to `<stdlib.h>`. The argument passed to `exit` has the same meaning as `main`'s return value: both indicate the program's status at termination. To indicate normal termination, we'd pass 0:

`<stdlib.h>` header ▶ 26.2

```
exit(0); /* normal termination */
```

Since 0 is a bit cryptic, C allows us to pass `EXIT_SUCCESS` instead (the effect is the same):

```
exit(EXIT_SUCCESS); /* normal termination */
```

Passing `EXIT_FAILURE` indicates abnormal termination:

```
exit(EXIT_FAILURE); /* abnormal termination */
```

`EXIT_SUCCESS` and `EXIT_FAILURE` are macros defined in `<stdlib.h>`. The values of `EXIT_SUCCESS` and `EXIT_FAILURE` are implementation-defined; typical values are 0 and 1, respectively.

As methods of terminating a program, `return` and `exit` are closely related. In fact, the statement

```
return expression;
```

in `main` is equivalent to

```
exit (expression) ;
```

The difference between `return` and `exit` is that `exit` causes program termination regardless of which function calls it. The `return` statement causes program termination only when it appears in the `main` function. Some programmers use `exit` exclusively to make it easier to locate all exit points in a program.

9.6 Recursion

A function is *recursive* if it calls itself. For example, the following function computes $n!$ recursively, using the formula $n! = n \times (n - 1)!$:

```
int fact(int n)
{
    if (n <= 1)
        return 1;
    else
        return n * fact(n - 1);
}
```

Some programming languages rely heavily on recursion, while others don't even allow it. C falls somewhere in the middle: it allows recursion, but most C programmers don't use it that often.

To see how recursion works, let's trace the execution of the statement

```
i = fact(3);
```

Here's what happens:

```
fact(3) finds that 3 is not less than or equal to 1, so it calls
    fact(2), which finds that 2 is not less than or equal to 1, so it calls
        fact(1), which finds that 1 is less than or equal to 1, so it returns 1, causing
        fact(2) to return  $2 \times 1 = 2$ , causing
    fact(3) to return  $3 \times 2 = 6$ .
```

Notice how the unfinished calls of `fact` “pile up” until `fact` is finally passed 1. At that point, the old calls of `fact` begin to “unwind” one by one, until the original call—`fact(3)`—finally returns with the answer, 6.

Here's another example of recursion: a function that computes x^n , using the formula $x^n = x \times x^{n-1}$.

```
int power(int x, int n)
{
    if (n == 0)
        return 1;
    else
        return x * power(x, n - 1);
}
```


The call `power(5, 3)` would be executed as follows:

```
power(5, 3) finds that 3 is not equal to 0, so it calls
  power(5, 2), which finds that 2 is not equal to 0, so it calls
    power(5, 1), which finds that 1 is not equal to 0, so it calls
      power(5, 0), which finds that 0 is equal to 0, so it returns 1, causing
        power(5, 1) to return  $5 \times 1 = 5$ , causing
          power(5, 2) to return  $5 \times 5 = 25$ , causing
            power(5, 3) to return  $5 \times 25 = 125$ .
```

Incidentally, we can condense the `power` function a bit by putting a conditional expression in the return statement:

```
int power(int x, int n)
{
  return n == 0 ? 1 : x * power(x, n - 1);
}
```

Both `fact` and `power` are careful to test a “termination condition” as soon as they’re called. When `fact` is called, it immediately checks whether its parameter is less than or equal to 1. When `power` is called, it first checks whether its second parameter is equal to 0. All recursive functions need some kind of termination condition in order to prevent infinite recursion.

The Quicksort Algorithm

At this point, you may wonder why we’re bothering with recursion; after all, neither `fact` nor `power` really needs it. Well, you’ve got a point. Neither function makes much of a case for recursion, because each calls itself just once. Recursion is much more helpful for sophisticated algorithms that require a function to call itself two or more times.

In practice, recursion often arises naturally as a result of an algorithm design technique known as *divide-and-conquer*, in which a large problem is divided into smaller pieces that are then tackled by the same algorithm. A classic example of the divide-and-conquer strategy can be found in the popular sorting algorithm known as *Quicksort*. The Quicksort algorithm goes as follows (for simplicity, we’ll assume that the array being sorted is indexed from 1 to n):

1. Choose an array element e (the “partitioning element”), then rearrange the array so that elements $1, \dots, i-1$ are less than or equal to e , element i contains e , and elements $i+1, \dots, n$ are greater than or equal to e .
2. Sort elements $1, \dots, i-1$ by using Quicksort recursively.
3. Sort elements $i+1, \dots, n$ by using Quicksort recursively.

After step 1, the element e is in its proper location. Since the elements to the left of e are all less than or equal to it, they’ll be in their proper places once they’ve been sorted in step 2; similar reasoning applies to the elements to the right of e .

Step 1 of the Quicksort algorithm is obviously critical. There are various methods to partition an array, some much better than others. We’ll use a technique

that's easy to understand but not particularly efficient. I'll first describe the partitioning algorithm informally; later, we'll translate it into C code.

The algorithm relies on two "markers" named *low* and *high*, which keep track of positions within the array. Initially, *low* points to the first element of the array and *high* points to the last element. We start by copying the first element (the partitioning element) into a temporary location elsewhere, leaving a "hole" in the array. Next, we move *high* across the array from right to left until it points to an element that's smaller than the partitioning element. We then copy the element into the hole that *low* points to, which creates a new hole (pointed to by *high*). We now move *low* from left to right, looking for an element that's larger than the partitioning element. When we find one, we copy it into the hole that *high* points to. The process repeats, with *low* and *high* taking turns, until they meet somewhere in the middle of the array. At that time, both will point to a hole; all we need do is copy the partitioning element into the hole. The following diagrams illustrate how Quicksort would sort an array of integers:

Let's start with an array containing seven elements. *low* points to the first element; *high* points to the last one.



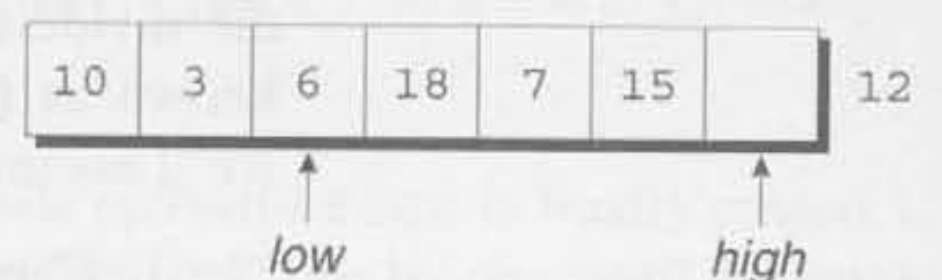
The first element, 12, is the partitioning element. Copying it somewhere else leaves a hole at the beginning of the array.



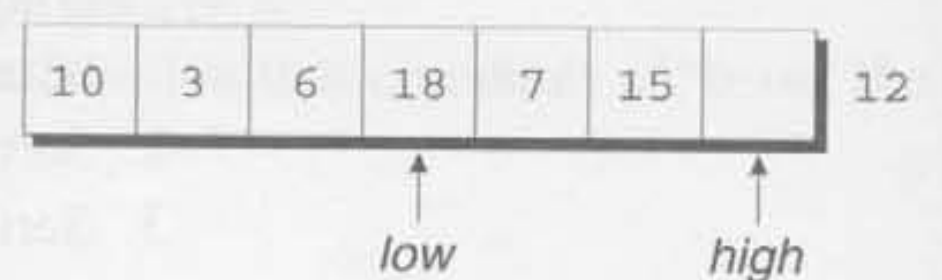
We now compare the element pointed to by *high* with 12. Since 10 is smaller than 12, it's on the wrong side of the array, so we move it to the hole and shift *low* to the right.



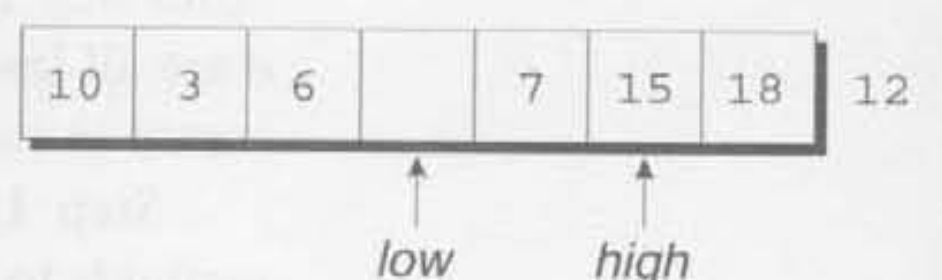
low points to the number 3, which is less than 12 and therefore doesn't need to be moved. We shift *low* to the right instead.



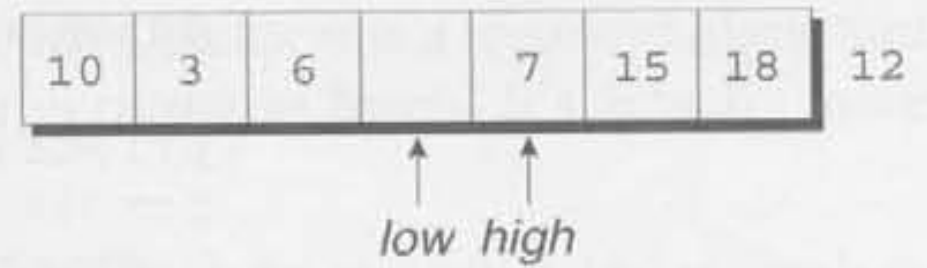
Since 6 is also less than 12, we shift *low* again.



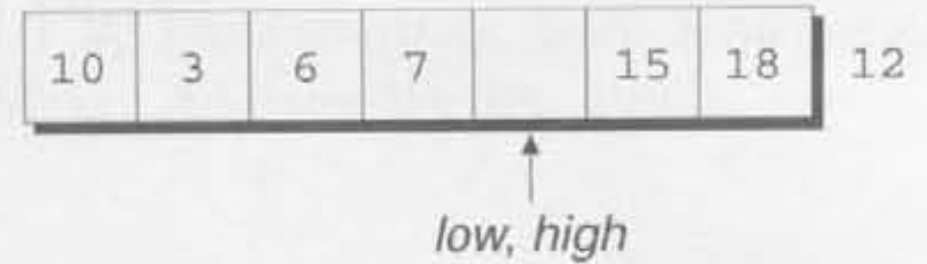
low now points to 18, which is larger than 12 and therefore out of position. After moving 18 to the hole, we shift *high* to the left.



high points to 15, which is greater than 12 and thus doesn't need to be moved. We shift *high* to the left and continue.



high points to 7, which is out of position. After moving 7 to the hole, we shift *low* to the right.



low and *high* are now equal, so we move the partitioning element to the hole.



At this point, we've accomplished our objective: all elements to the left of the partitioning element are less than or equal to 12, and all elements to the right are greater than or equal to 12. Now that the array has been partitioned, we can use Quicksort recursively to sort the first four elements of the array (10, 3, 6, and 7) and the last two (15 and 18).

PROGRAM Quicksort

Let's develop a recursive function named `quicksort` that uses the Quicksort algorithm to sort an array of integers. To test the function, we'll have `main` read 10 numbers into an array, call `quicksort` to sort the array, then print the elements in the array:

```
Enter 10 numbers to be sorted: 9 16 47 82 4 66 12 3 25 51
In sorted order: 3 4 9 12 16 25 47 51 66 82
```

Since the code for partitioning the array is a bit lengthy, I'll put it in a separate function named `split`.

```
qsort.c /* Sorts an array of integers using Quicksort algorithm */
```

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

```
#define N 10
```

```
void quicksort(int a[], int low, int high);
int split(int a[], int low, int high);
```

```
int main(void)
```

```
{
```

```
    int a[N], i;
```

```
    printf("Enter %d numbers to be sorted: ", N);
```

```
    for (i = 0; i < N; i++)
```

```
        scanf("%d", &a[i]);
```

```

        quicksort(a, 0, N - 1);

        printf("In sorted order: ");
        for (i = 0; i < N; i++)
            printf("%d ", a[i]);
        printf("\n");

        return 0;
    }

void quicksort(int a[], int low, int high)
{
    int middle;

    if (low >= high) return;
    middle = split(a, low, high);
    quicksort(a, low, middle - 1);
    quicksort(a, middle + 1, high);
}

int split(int a[], int low, int high)
{
    int part_element = a[low];

    for (;;) {
        while (low < high && part_element <= a[high])
            high--;
        if (low >= high) break;
        a[low++] = a[high];

        while (low < high && a[low] <= part_element)
            low++;
        if (low >= high) break;
        a[high--] = a[low];
    }

    a[high] = part_element;
    return high;
}

```

Although this version of Quicksort works, it's not the best. There are numerous ways to improve the program's performance, including:

- **Improving the partitioning algorithm.** Our method isn't the most efficient. Instead of choosing the first element in the array as the partitioning element, it's better to take the median of the first element, the middle element, and the last element. The partitioning process itself can also be sped up. In particular, it's possible to avoid the `low < high` tests in the two `while` loops.
- **Using a different method to sort small arrays.** Instead of using Quicksort recursively all the way down to arrays with one element, it's better to use a simpler method for small arrays (those with fewer than, say, 25 elements).

- **Making Quicksort nonrecursive.** Although Quicksort is a recursive algorithm by nature—and is easiest to understand in recursive form—it’s actually more efficient if the recursion is removed.

For details about improving Quicksort, consult a book on algorithm design, such as Robert Sedgewick’s *Algorithms in C, Parts 1–4: Fundamentals, Data Structures, Sorting, Searching*, Third Edition (Boston, Mass.: Addison-Wesley, 1998).

Q & A

Q: Some C books appear to use terms other than *parameter* and *argument*. Is there any standard terminology? [p. 184]

A: As with many other aspects of C, there’s no general agreement on terminology, although the C89 and C99 standards use *parameter* and *argument*. The following table should help you translate:

<i>This book:</i>	<i>Other books:</i>
parameter	formal argument, formal parameter
argument	actual argument, actual parameter

Keep in mind that—when no confusion would result—I sometimes deliberately blur the distinction between the two terms, using *argument* to mean either.

Q: I’ve seen programs in which parameter types are specified in separate declarations after the parameter list, as in the following example:

```
double average(a, b)
double a, b;
{
    return (a + b) / 2;
}
```

Is this practice legal? [p. 188]

A: This method of defining functions comes from K&R C, so you may encounter it in older books and programs. C89 and C99 support this style so that older programs will still compile. I’d avoid using it in new programs, however, for a couple of reasons.

First, functions that are defined in the older way aren’t subject to the same degree of error-checking. When a function is defined in the older way—and no prototype is present—the compiler won’t check that the function is called with the right number of arguments, nor will it check that the arguments have the proper types. Instead, it will perform the default argument promotions.

Second, the C standard says that the older style is “obsolescent,” meaning that its use is discouraged and that it may be dropped from C eventually.

Q: Some programming languages allow procedures and functions to be nested within each other. Does C allow function definitions to be nested?

A: No. C does not permit the definition of one function to appear in the body of another. Among other things, this restriction simplifies the compiler.

***Q:** Why does the compiler allow the use of function names that aren't followed by parentheses? [p. 189]

A: We'll see in a later chapter that the compiler treats a function name not followed by parentheses as a *pointer* to the function. Pointers to functions have legitimate uses, so the compiler can't automatically assume that a function name without parentheses is an error. The statement

```
print_pun;
```

is legal because the compiler treats `print_pun` as a pointer and therefore an expression, making this a valid (although pointless) expression statement.

***Q:** In the function call `f(a, b)`, how does the compiler know whether the comma is punctuation or whether it's an operator?

A: It turns out that the arguments in a function call can't be arbitrary expressions. Instead, they must be "assignment expressions," which can't contain commas used as operators unless they're enclosed in parentheses. In other words, in the call `f(a, b)` the comma is punctuation; in the call `f((a, b))` it's an operator.

Q: Do the names of parameters in a function prototype have to match the names given later in the function's definition? [p. 192]

A: No. Some programmers take advantage of this fact by giving long names to parameters in the prototype, then using shorter names in the actual definition. Or a French-speaking programmer might use English names in prototypes, then switch to more familiar French names in function definitions.

Q: I still don't understand why we bother with function prototypes. If we just put definitions of all the functions before `main`, we're covered, right?

A: Wrong. First, you're assuming that only `main` calls the other functions, which is unrealistic. In practice, some of the functions will call each other. If we put all function definitions above `main`, we'll have to watch their order carefully. Calling a function that hasn't been defined yet can lead to big problems.

But that's not all. Suppose that two functions call each other (which isn't as far-fetched as it may sound). No matter which function we define first, it will end up calling a function that hasn't been defined yet.

But there's still more! Once programs reach a certain size, it won't be feasible to put all the functions in one file anymore. When we reach that point, we'll need prototypes to tell the compiler about functions in other files.

Q: I've seen function declarations that omit all information about parameters:


```
double average();
```

Is this practice legal? [p. 192]

- A: Yes. This declaration informs the compiler that `average` returns a `double` value but provides no information about the number and types of its parameters. (Leaving the parentheses empty doesn't necessarily mean that `average` has no parameters.)

In K&R C, this form of function declaration is the only one allowed; the form that we've been using—the function prototype, in which parameter information *is* included—was introduced in C89. The older kind of function declaration is now obsolescent, although still allowed.

Q: Why would a programmer deliberately omit parameter names in a function prototype? Isn't it easier to just leave the names? [p. 193]

- A: Omitting parameter names in prototypes is typically done for defensive purposes. If a macro happens to have the same name as a parameter, the parameter name will be replaced during preprocessing, thereby damaging the prototype in which it appears. This isn't likely to be a problem in a small program written by one person but can occur in large applications written by many people.

Q: Is it legal to put a function declaration inside the body of another function?

- A: Yes. Here's an example:

```
int main(void)
{
    double average(double a, double b);
    ...
}
```

This declaration of `average` is valid only for the body of `main`; if other functions need to call `average`, they'll each have to declare it.

The advantage of this practice is that it's clearer to the reader which functions call which other functions. (In this example, we see that `main` will be calling `average`.) On the other hand, it can be a nuisance if several functions need to call the same function. Even worse, trying to add and remove declarations during program maintenance can be a real pain. For these reasons, I'll always put function declarations outside function bodies.

Q: If several functions have the same return type, can their declarations be combined? For example, since both `print_pun` and `print_count` have `void` as their return type, is the following declaration legal?

```
void print_pun(void), print_count(int n);
```

- A: Yes. In fact, C even allows us to combine function declarations with variable declarations:

```
double x, y, average(double a, double b);
```


Combining declarations in this way usually isn't a good idea, though; it can easily cause confusion.

Q: What happens if I specify a length for a one-dimensional array parameter? [p. 195]

A: The compiler ignores it. Consider the following example:

```
double inner_product(double v[3], double w[3]);
```

Other than documenting that `inner_product`'s arguments are supposed to be arrays of length 3, specifying a length doesn't buy us much. The compiler won't check that the arguments actually have length 3, so there's no added security. In fact, the practice is misleading in that it suggests that `inner_product` can only be passed arrays of length 3, when in fact we can pass arrays of arbitrary length.

***Q: Why can the first dimension in an array parameter be left unspecified, but not the other dimensions? [p. 197]**

A: First, we need to discuss how arrays are passed in C. As Section 12.3 explains, when an array is passed to a function, the function is given a *pointer* to the first element in the array.

Next, we need to know how the subscripting operator works. Suppose that `a` is a one-dimensional array passed to a function. When we write

```
a[i] = 0;
```

the compiler generates instructions that compute the address of `a[i]` by multiplying `i` by the size of an array element and adding the result to the address that `a` represents (the pointer passed to the function). This calculation doesn't depend on the length of `a`, which explains why we can omit it when defining the function.

What about multidimensional arrays? Recall that C stores arrays in row-major order, with the elements in row 0 stored first, then the elements in row 1, and so forth. Suppose that `a` is a two-dimensional array parameter and we write

```
a[i][j] = 0;
```

The compiler generates instructions to do the following: (1) multiply `i` by the size of a single row of `a`; (2) add this result to the address that `a` represents; (3) multiply `j` by the size of an array element; and (4) add this result to the address computed in step 2. To generate these instructions, the compiler must know the size of a row in the array, which is determined by the number of columns. The bottom line: the programmer must declare the number of columns in `a`.

Q: Why do some programmers put parentheses around the expression in a return statement?

A: The examples in the first edition of Kernighan and Ritchie's *The C Programming Language* always have parentheses in return statements, even though they aren't required. Programmers (and authors of subsequent books) picked up the habit from K&R. I don't use these parentheses, since they're unnecessary and

contribute nothing to readability. (Kernighan and Ritchie apparently agree: the return statements in the second edition of *The C Programming Language* lack parentheses.)

Q: What happens if a non-void function attempts to execute a return statement that has no expression? [p. 202]

A: That depends on the version of C. In C89, executing a return statement without an expression in a non-void function causes undefined behavior (but only if the program attempts to use the value returned by the function). In C99, such a statement is illegal and should be detected as an error by the compiler.

C99

Q: How can I test main's return value to see if a program has terminated normally? [p. 203]

A: That depends on your operating system. Many operating systems allow this value to be tested within a "batch file" or "shell script" that contains commands to run several programs. For example, the line

```
if errorlevel 1 command
```

in a Windows batch file will execute *command* if the last program terminated with a status code greater than or equal to 1.

In UNIX, each shell has its own method for testing the status code. In the Bourne shell, the variable `$_` contains the status of the last program run. The C shell has a similar variable, but its name is `$status`.

Q: Why does my compiler produce a "control reaches end of non-void function" warning when it compiles main?

A: The compiler has noticed that `main`, despite having `int` as its return type, doesn't have a return statement. Putting the statement

```
return 0;
```

at the end of `main` will keep the compiler happy. Incidentally, this is good practice even if your compiler doesn't object to the lack of a return statement.

C99

When a program is compiled using a C99 compiler, this warning shouldn't occur. In C99, it's OK to "fall off" the end of `main` without returning a value; the standard states that `main` automatically returns 0 in this situation.

Q: With regard to the previous question: Why not just define main's return type to be void?

A: Although this practice is fairly common, it's illegal according to the C89 standard. Even if it weren't illegal, it wouldn't be a good idea, since it presumes that no one will ever test the program's status upon termination.

C99

C99 opens the door to legalizing this practice, by allowing `main` to be declared "in some other implementation-defined manner" (with a return type other than `int` or parameters other than those specified by the standard). However, any such usage isn't portable, so it's best to declare `main`'s return type to be `int`.

Q: Is it legal for a function `f1` to call a function `f2`, which then calls `f1`?

A: Yes. This is just an indirect form of recursion in which one call of `f1` leads to another. (But make sure that either `f1` or `f2` eventually terminates!)

Exercises

Section 9.1

1. The following function, which computes the area of a triangle, contains two errors. Locate the errors and show how to fix them. (*Hint:* There are no errors in the formula.)

```
double triangle_area(double base, height)
double product;
{
    product = base * height;
    return product / 2;
}
```

- Ⓜ 2. Write a function `check(x, y, n)` that returns 1 if both `x` and `y` fall between 0 and `n - 1`, inclusive. The function should return 0 otherwise. Assume that `x`, `y`, and `n` are all of type `int`.
3. Write a function `gcd(m, n)` that calculates the greatest common divisor of the integers `m` and `n`. (Programming Project 2 in Chapter 6 describes Euclid's algorithm for computing the GCD.)
- Ⓜ 4. Write a function `day_of_year(month, day, year)` that returns the day of the year (an integer between 1 and 366) specified by the three arguments.
5. Write a function `num_digits(n)` that returns the number of digits in `n` (a positive integer). *Hint:* To determine the number of digits in a number `n`, divide it by 10 repeatedly. When `n` reaches 0, the number of divisions indicates how many digits `n` originally had.
- Ⓜ 6. Write a function `digit(n, k)` that returns the k^{th} digit (from the right) in `n` (a positive integer). For example, `digit(829, 1)` returns 9, `digit(829, 2)` returns 2, and `digit(829, 3)` returns 8. If `k` is greater than the number of digits in `n`, have the function return 0.
7. Suppose that the function `f` has the following definition:


```
int f(int a, int b) { ... }
```

 Which of the following statements are legal? (Assume that `i` has type `int` and `x` has type `double`.)
 - (a) `i = f(83, 12);`
 - (b) `x = f(83, 12);`
 - (c) `i = f(3.15, 9.28);`
 - (d) `x = f(3.15, 9.28);`
 - (e) `f(83, 12);`

Section 9.2

- Ⓜ 8. Which of the following would be valid prototypes for a function that returns nothing and has one `double` parameter?

 - (a) `void f(double x);`

- (b) `void f(double);`
- (c) `void f(x);`
- (d) `f(double x);`

Section 9.3

- *9. What will be the output of the following program?

```
#include <stdio.h>

void swap(int a, int b);

int main(void)
{
    int i = 1, j = 2;

    swap(i, j);
    printf("i = %d, j = %d\n", i, j);
    return 0;
}

void swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

- Ⓜ 10. Write functions that return the following values. (Assume that `a` and `n` are parameters, where `a` is an array of `int` values and `n` is the length of the array.)
- (a) The largest element in `a`.
 - (b) The average of all elements in `a`.
 - (c) The number of positive elements in `a`.

11. Write the following function:

```
float compute_GPA(char grades[], int n);
```

The `grades` array will contain letter grades (A, B, C, D, or F, either upper-case or lower-case); `n` is the length of the array. The function should return the average of the grades (assume that A = 4, B = 3, C = 2, D = 1, and F = 0).

12. Write the following function:

```
double inner_product(double a[], double b[], int n);
```

The function should return $a[0] * b[0] + a[1] * b[1] + \dots + a[n-1] * b[n-1]$.

13. Write the following function, which evaluates a chess position:

```
int evaluate_position(char board[8][8]);
```

`board` represents a configuration of pieces on a chessboard, where the letters K, Q, R, B, N, P represent White pieces, and the letters k, q, r, b, n, and p represent Black pieces. `evaluate_position` should sum the values of the White pieces (Q = 9, R = 5, B = 3, N = 3, P = 1). It should also sum the values of the Black pieces (done in a similar way). The function will return the difference between the two numbers. This value will be positive if White has an advantage in material and negative if Black has an advantage.

Section 9.4

14. The following function is supposed to return `true` if any element of the array `a` has the value 0 and `false` if all elements are nonzero. Sadly, it contains an error. Find the error and show how to fix it:

```
bool has_zero(int a[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        if (a[i] == 0)
            return true;
    else
        return false;
}
```

- Ⓜ 15. The following (rather confusing) function finds the median of three numbers. Rewrite the function so that it has just one return statement.

```
double median(double x, double y, double z)
{
    if (x <= y)
        if (y <= z) return y;
        else if (x <= z) return z;
        else return x;
    if (z <= y) return y;
    if (x <= z) return x;
    return z;
}
```

Section 9.6

16. Condense the `fact` function in the same way we condensed `power`.
- Ⓜ 17. Rewrite the `fact` function so that it's no longer recursive.
18. Write a recursive version of the `gcd` function (see Exercise 3). Here's the strategy to use for computing `gcd(m, n)`: If `n` is 0, return `m`. Otherwise, call `gcd` recursively, passing `n` as the first argument and `m % n` as the second.
- Ⓜ*19. Consider the following "mystery" function:

```
void pb(int n)
{
    if (n != 0) {
        pb(n / 2);
        putchar('0' + n % 2);
    }
}
```

Trace the execution of the function by hand. Then write a program that calls the function, passing it a number entered by the user. What does the function do?

Programming Projects

- Write a program that asks the user to enter a series of integers (which it stores in an array), then sorts the integers by calling the function `selection_sort`. When given an array with n elements, `selection_sort` must do the following:
 - Search the array to find the largest element, then move it to the last position in the array.
 - Call itself recursively to sort the first $n - 1$ elements of the array.

2. Modify Programming Project 5 from Chapter 5 so that it uses a function to compute the amount of income tax. When passed an amount of taxable income, the function will return the tax due.
3. Modify Programming Project 9 from Chapter 8 so that it includes the following functions:


```
void generate_random_walk(char walk[10][10]);
void print_array(char walk[10][10]);
```

main first calls `generate_random_walk`, which initializes the array to contain '.' characters and then replaces some of these characters by the letters A through Z, as described in the original project. main then calls `print_array` to display the array on the screen.
4. Modify Programming Project 16 from Chapter 8 so that it includes the following functions:


```
void read_word(int counts[26]);
bool equal_array(int counts1[26], int counts2[26]);
```

main will call `read_word` twice, once for each of the two words entered by the user. As it reads a word, `read_word` will use the letters in the word to update the `counts` array, as described in the original project. (main will declare two arrays, one for each word. These arrays are used to track how many times each letter occurs in the words.) main will then call `equal_array`, passing it the two arrays. `equal_array` will return `true` if the elements in the two arrays are identical (indicating that the words are anagrams) and `false` otherwise.
5. Modify Programming Project 17 from Chapter 8 so that it includes the following functions:


```
void create_magic_square(int n, char magic_square[n][n]);
void print_magic_square(int n, char magic_square[n][n]);
```

After obtaining the number n from the user, main will call `create_magic_square`, passing it an $n \times n$ array that is declared inside main. `create_magic_square` will fill the array with the numbers 1, 2, ..., n^2 as described in the original project. main will then call `print_magic_square`, which will display the array in the format described in the original project. *Note:* If your compiler doesn't support variable-length arrays, declare the array in main to be 99×99 instead of $n \times n$ and use the following prototypes instead:


```
void create_magic_square(int n, char magic_square[99][99]);
void print_magic_square(int n, char magic_square[99][99]);
```
6. Write a function that computes the value of the following polynomial:

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

Write a program that asks the user to enter a value for x , calls the function to compute the value of the polynomial, and then displays the value returned by the function.
7. The `power` function of Section 9.6 can be made faster by having it calculate x^n in a different way. We first notice that if n is a power of 2, then x^n can be computed by squaring. For example, x^4 is the square of x^2 , so x^4 can be computed using only two multiplications instead of three. As it happens, this technique can be used even when n is not a power of 2. If n is even, we use the formula $x^n = (x^{n/2})^2$. If n is odd, then $x^n = x \times x^{n-1}$. Write a recursive function that computes x^n . (The recursion ends when $n = 0$, in which case the function returns 1.) To test your function, write a program that asks the user to enter values for x and n , calls `power` to compute x^n , and then displays the value returned by the function.
8. Write a program that simulates the game of craps, which is played with two dice. On the first roll, the player wins if the sum of the dice is 7 or 11. The player loses if the sum is 2, 3,

or 12. Any other roll is called the “point” and the game continues. On each subsequent roll, the player wins if he or she rolls the point again. The player loses by rolling 7. Any other roll is ignored and the game continues. At the end of each game, the program will ask the user whether or not to play again. When the user enters a response other than `y` or `Y`, the program will display the number of wins and losses and then terminate.

```
You rolled: 8
Your point is 8
You rolled: 3
You rolled: 10
You rolled: 8
You win!
```

```
Play again? y
```

```
You rolled: 6
Your point is 6
You rolled: 5
You rolled: 12
You rolled: 3
You rolled: 7
You lose!
```

```
Play again? y
```

```
You rolled: 11
You win!
```

```
Play again? n
```

```
Wins: 2 Losses: 1
```

Write your program as three functions: `main`, `roll_dice`, and `play_game`. Here are the prototypes for the latter two functions:

```
int roll_dice(void);
bool play_game(void);
```

`roll_dice` should generate two random numbers, each between 1 and 6, and return their sum. `play_game` should play one craps game (calling `roll_dice` to determine the outcome of each dice roll); it will return `true` if the player wins and `false` if the player loses. `play_game` is also responsible for displaying messages showing the results of the player's dice rolls. `main` will call `play_game` repeatedly, keeping track of the number of wins and losses and displaying the “you win” and “you lose” messages. *Hint:* Use the `rand` function to generate random numbers. See the `deal.c` program in Section 8.2 for an example of how to call `rand` and the related `srand` function.

10 Program Organization

As Will Rogers would have said, "There is no such thing as a free variable."

Having covered functions in Chapter 9, we're ready to confront several issues that arise when a program contains more than one function. The chapter begins with a discussion of the differences between local variables (Section 10.1) and external variables (Section 10.2). Section 10.3 then considers blocks (compound statements containing declarations). Section 10.4 tackles the scope rules that apply to local names, external names, and names declared in blocks. Finally, Section 10.5 suggests a way to organize function prototypes, function definitions, variable declarations, and the other parts of a C program.

10.1 Local Variables

A variable declared in the body of a function is said to be *local* to the function. In the following function, `sum` is a local variable:

```
int sum_digits(int n)
{
    int sum = 0;    /* local variable */

    while (n > 0) {
        sum += n % 10;
        n /= 10;
    }

    return sum;
}
```

By default, local variables have the following properties:

- **Automatic storage duration.** The *storage duration* (or *extent*) of a variable is the portion of program execution during which storage for the variable exists. Storage for a local variable is “automatically” allocated when the enclosing function is called and deallocated when the function returns, so the variable is said to have *automatic storage duration*. A local variable doesn’t retain its value when its enclosing function returns. When the function is called again, there’s no guarantee that the variable will still have its old value.
- **Block scope.** The *scope* of a variable is the portion of the program text in which the variable can be referenced. A local variable has *block scope*: it is visible from its point of declaration to the end of the enclosing function body. Since the scope of a local variable doesn’t extend beyond the function to which it belongs, other functions can use the same name for other purposes.

Section 18.2 covers these and other related concepts in more detail.

C99

Since C99 doesn’t require variable declarations to come at the beginning of a function, it’s possible for a local variable to have a very small scope. In the following example, the scope of `i` doesn’t begin until the line on which it’s declared, which could be near the end of the function body:

```
void f(void)
{
    ...
    int i;
    ...
}
```

} scope of i

Static Local Variables

Putting the word `static` in the declaration of a local variable causes it to have *static storage duration* instead of automatic storage duration. A variable with static storage duration has a permanent storage location, so it retains its value throughout the execution of the program. Consider the following function:

```
void f(void)
{
    static int i;    /* static local variable */
    ...
}
```

Since the local variable `i` has been declared `static`, it occupies the same memory location throughout the execution of the program. When `f` returns, `i` won’t lose its value.

Q&A

A static local variable still has block scope, so it’s not visible to other functions. In a nutshell, a static variable is a place to hide data from other functions but retain it for future calls of the same function.

Parameters

Parameters have the same properties—automatic storage duration and block scope—as local variables. In fact, the only real difference between parameters and local variables is that each parameter is initialized automatically when a function is called (by being assigned the value of the corresponding argument).

10.2 External Variables

Passing arguments is one way to transmit information to a function. Functions can also communicate through *external variables*—variables that are declared outside the body of any function.

The properties of external variables (or *global variables*, as they're sometimes called) are different from those of local variables:

- **Static storage duration.** External variables have static storage duration, just like local variables that have been declared `static`. A value stored in an external variable will stay there indefinitely.
- **File scope.** An external variable has *file scope*: it is visible from its point of declaration to the end of the enclosing file. As a result, an external variable can be accessed (and potentially modified) by all functions that follow its declaration.

Example: Using External Variables to Implement a Stack

To illustrate how external variables might be used, let's look at a data structure known as a *stack*. (Stacks are an abstract concept, not a C feature; they can be implemented in most programming languages.) A stack, like an array, can store multiple data items of the same type. However, the operations on a stack are limited: we can either *push* an item onto the stack (add it to one end—the “stack top”) or *pop* it from the stack (remove it from the same end). Examining or modifying an item that's not at the top of the stack is forbidden.

One way to implement a stack in C is to store its items in an array, which we'll call `contents`. A separate integer variable named `top` marks the position of the stack top. When the stack is empty, `top` has the value 0. To push an item on the stack, we simply store the item in `contents` at the position indicated by `top`, then increment `top`. Popping an item requires decrementing `top`, then using it as an index into `contents` to fetch the item that's being popped.

Based on this outline, here's a program fragment (not a complete program) that declares the `contents` and `top` variables for a stack and provides a set of functions that represent operations on the stack. All five functions need access to the `top` variable, and two functions need access to `contents`, so we'll make `contents` and `top` external.

```

#include <stdbool.h>    /* C99 only */

#define STACK_SIZE 100

/* external variables */
int contents[STACK_SIZE];
int top = 0;

void make_empty(void)
{
    top = 0;
}

bool is_empty(void)
{
    return top == 0;
}

bool is_full(void)
{
    return top == STACK_SIZE;
}

void push(int i)
{
    if (is_full())
        stack_overflow();
    else
        contents[top++] = i;
}

int pop(void)
{
    if (is_empty())
        stack_underflow();
    else
        return contents[--top];
}

```

Pros and Cons of External Variables

External variables are convenient when many functions must share a variable or when a few functions share a large number of variables. In most cases, however, it's better for functions to communicate through parameters rather than by sharing variables. Here's why:

- If we change an external variable during program maintenance (by altering its type, say), we'll need to check every function in the same file to see how the change affects it.

- If an external variable is assigned an incorrect value, it may be difficult to identify the guilty function. It's like trying to solve a murder committed at a crowded party—there's no easy way to narrow the list of suspects.
- Functions that rely on external variables are hard to reuse in other programs. A function that depends on external variables isn't self-contained; to reuse the function, we'll have to drag along any external variables that it needs.

Many C programmers rely far too much on external variables. One common abuse: using the same external variable for different purposes in different functions. Suppose that several functions need a variable named `i` to control a `for` statement. Instead of declaring `i` in each function that uses it, some programmers declare it at the top of the program, thereby making the variable visible to all functions. This practice is poor not only for the reasons listed earlier, but also because it's misleading; someone reading the program later may think that the uses of the variable are related, when in fact they're not.

When you use external variables, make sure they have meaningful names. (Local variables don't always need meaningful names: it's often hard to think of a better name than `i` for the control variable in a `for` loop.) If you find yourself using names like `i` and `temp` for external variables, that's a clue that perhaps they should really be local variables.



Making variables external when they should be local can lead to some rather frustrating bugs. Consider the following example, which is supposed to display a 10 × 10 arrangement of asterisks:

```
int i;

void print_one_row(void)
{
    for (i = 1; i <= 10; i++)
        printf("*");
}

void print_all_rows(void)
{
    for (i = 1; i <= 10; i++) {
        print_one_row();
        printf("\n");
    }
}
```

Instead of printing 10 rows, `print_all_rows` prints only one row. When `print_one_row` returns after being called the first time, `i` will have the value 11. The `for` statement in `print_all_rows` then increments `i` and tests whether it's less than or equal to 10. It's not, so the loop terminates and the function returns.

PROGRAM Guessing a Number

To get more experience with external variables, we'll write a simple game-playing program. The program generates a random number between 1 and 100, which the user attempts to guess in as few tries as possible. Here's what the user will see when the program is run:

```
Guess the secret number between 1 and 100.
```

```
A new number has been chosen.
```

```
Enter guess: 55
```

```
Too low; try again.
```

```
Enter guess: 65
```

```
Too high; try again.
```

```
Enter guess: 60
```

```
Too high; try again.
```

```
Enter guess: 58
```

```
You won in 4 guesses!
```

```
Play again? (Y/N) y
```

```
A new number has been chosen.
```

```
Enter guess: 78
```

```
Too high; try again.
```

```
Enter guess: 34
```

```
You won in 2 guesses!
```

```
Play again? (Y/N) n
```

This program will need to carry out several different tasks: initializing the random number generator, choosing a secret number, and interacting with the user until the correct number is picked. If we write a separate function to handle each task, we might end up with the following program.

```
guess.c /* Asks user to guess a hidden number */

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

#define MAX_NUMBER 100

/* external variable */
int secret_number;

/* prototypes */
void initialize_number_generator(void);
void choose_new_secret_number(void);
void read_guesses(void);

int main(void)
{
    char command;
```



```

printf("Guess the secret number between 1 and %d.\n\n",
      MAX_NUMBER);
initialize_number_generator();
do {
    choose_new_secret_number();
    printf("A new number has been chosen.\n");
    read_guesses();
    printf("Play again? (Y/N) ");
    scanf(" %c", &command);
    printf("\n");
} while (command == 'y' || command == 'Y');

return 0;
}

/*****
 * initialize_number_generator: Initializes the random
 *                             number generator using
 *                             the time of day.
 *****/
void initialize_number_generator(void)
{
    srand((unsigned) time(NULL));
}

/*****
 * choose_new_secret_number: Randomly selects a number
 *                           between 1 and MAX_NUMBER and
 *                           stores it in secret_number.
 *****/
void choose_new_secret_number(void)
{
    secret_number = rand() % MAX_NUMBER + 1;
}

/*****
 * read_guesses: Repeatedly reads user guesses and tells
 *               the user whether each guess is too low,
 *               too high, or correct. When the guess is
 *               correct, prints the total number of
 *               guesses and returns.
 *****/
void read_guesses(void)
{
    int guess, num_guesses = 0;

    for (;;) {
        num_guesses++;
        printf("Enter guess: ");
        scanf("%d", &guess);
        if (guess == secret_number) {
            printf("You won in %d guesses!\n\n", num_guesses);
            return;
        } else if (guess < secret_number)

```

```

        printf("Too low; try again.\n");
    else
        printf("Too high; try again.\n");
    }
}

```

time function ▶26.3
 srand function ▶26.2
 rand function ▶26.2

For random number generation, the `guess.c` program relies on the `time`, `srand`, and `rand` functions, which we first used in `deal.c` (Section 8.2). This time, we're scaling the return value of `rand` so that it falls between 1 and `MAX_NUMBER`.

Although `guess.c` works fine, it relies on an external variable. We made `secret_number` external so that both `choose_new_secret_number` and `read_guesses` could access it. If we alter `choose_new_secret_number` and `read_guesses` just a little, we should be able to move `secret_number` into the main function. We'll modify `choose_new_secret_number` so that it returns the new number, and we'll rewrite `read_guesses` so that `secret_number` can be passed to it as an argument.

Here's our new program, with changes in **bold**:

```

guess2.c /* Asks user to guess a hidden number */

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

#define MAX_NUMBER 100

/* prototypes */
void initialize_number_generator(void);
int new_secret_number(void);
void read_guesses(int secret_number);

int main(void)
{
    char command;
    int secret_number;

    printf("Guess the secret number between 1 and %d.\n\n",
           MAX_NUMBER);
    initialize_number_generator();
    do {
        secret_number = new_secret_number();
        printf("A new number has been chosen.\n");
        read_guesses(secret_number);
        printf("Play again? (Y/N) ");
        scanf(" %c", &command);
        printf("\n");
    } while (command == 'y' || command == 'Y');

    return 0;
}

```



```

/*****
 * initialize_number_generator: Initializes the random
 *                             number generator using
 *                             the time of day.
 *****/
void initialize_number_generator(void)
{
    srand((unsigned) time(NULL));
}

/*****
 * new_secret_number: Returns a randomly chosen number
 *                   between 1 and MAX_NUMBER.
 *****/
int new_secret_number(void)
{
    return rand() % MAX_NUMBER + 1;
}

/*****
 * read_guesses: Repeatedly reads user guesses and tells
 *              the user whether each guess is too low,
 *              too high, or correct. When the guess is
 *              correct, prints the total number of
 *              guesses and returns.
 *****/
void read_guesses(int secret_number)
{
    int guess, num_guesses = 0;

    for (;;) {
        num_guesses++;
        printf("Enter guess: ");
        scanf("%d", &guess);
        if (guess == secret_number) {
            printf("You won in %d guesses!\n\n", num_guesses);
            return;
        } else if (guess < secret_number)
            printf("Too low; try again.\n");
        else
            printf("Too high; try again.\n");
    }
}

```

10.3 Blocks

In Section 5.2, we encountered compound statements of the form

```
{ statements }
```



```

int i;           /* Declaration 1 */

void f(int i)   /* Declaration 2 */
{
    i = 1;
}

void g(void)
{
    int i = 2;   /* Declaration 3 */
    if (i > 0) {
        int i;   /* Declaration 4 */
        i = 3;
    }
    i = 4;
}

void h(void)
{
    i = 5;
}

```

- In Declaration 2, *i* is a parameter with block scope.
- In Declaration 3, *i* is an automatic variable with block scope.
- In Declaration 4, *i* is also automatic and has block scope.

i is used five times. C's scope rules allow us to determine the meaning of *i* in each case:

- The *i* = 1 assignment refers to the parameter in Declaration 2, not the variable in Declaration 1, since Declaration 2 hides Declaration 1.
- The *i* > 0 test refers to the variable in Declaration 3, since Declaration 3 hides Declaration 1 and Declaration 2 is out of scope.
- The *i* = 3 assignment refers to the variable in Declaration 4, which hides Declaration 3.
- The *i* = 4 assignment refers to the variable in Declaration 3. It can't refer to Declaration 4, which is out of scope.
- The *i* = 5 assignment refers to the variable in Declaration 1.

10.5 Organizing a C Program

Now that we've seen the major elements that make up a C program, it's time to develop a strategy for their arrangement. For now, we'll assume that a program

always fits into a single file. Chapter 15 shows how to organize a program that's split over several files.

So far, we've seen that a program may contain the following:

- Preprocessing directives such as `#include` and `#define`
- Type definitions
- Declarations of external variables
- Function prototypes
- Function definitions

C imposes only a few rules on the order of these items: A preprocessing directive doesn't take effect until the line on which it appears. A type name can't be used until it's been defined. A variable can't be used until it's declared. Although C isn't as picky about functions, I strongly recommend that every function be defined or declared prior to its first call. (C99 makes this a requirement anyway.)

C99

There are several ways to organize a program so that these rules are obeyed. Here's one possible ordering:

- `#include` directives
- `#define` directives
- Type definitions
- Declarations of external variables
- Prototypes for functions other than `main`
- Definition of `main`
- Definitions of other functions

It makes sense to put `#include` directives first, since they bring in information that will likely be needed in several places within the program. `#define` directives create macros, which are generally used throughout the program. Putting type definitions above the declarations of external variables is logical, since the declarations of these variables may refer to the type names just defined. Declaring external variables next makes them available to all the functions that follow. Declaring all functions except for `main` avoids the problems that arise when a function is called before the compiler has seen its prototype. This practice also makes it possible to arrange the function definitions in any order whatsoever: alphabetically by function name or with related functions grouped together, for example. Defining `main` before the other functions makes it easier for a reader to locate the program's starting point.

A final suggestion: Precede each function definition by a boxed comment that gives the name of the function, explains its purpose, discusses the meaning of each parameter, describes its return value (if any), and lists any side effects it has (such as modifying external variables).

PROGRAM **Classifying a Poker Hand**

To show how a C program might be organized, let's attempt a program that's a little more complex than our previous examples. The program will read and classify

a poker hand. Each card in the hand will have both a *suit* (clubs, diamonds, hearts, or spades) and a *rank* (two, three, four, five, six, seven, eight, nine, ten, jack, queen, king, or ace). We won't allow the use of jokers, and we'll assume that aces are high. The program will read a hand of five cards, then classify the hand into one of the following categories (listed in order from best to worst):

- straight flush (both a straight and a flush)
- four-of-a-kind (four cards of the same rank)
- full house (a three-of-a-kind and a pair)
- flush (five cards of the same suit)
- straight (five cards with consecutive ranks)
- three-of-a-kind (three cards of the same rank)
- two pairs
- pair (two cards of the same rank)
- high card (any other hand)

If a hand falls into two or more categories, the program will choose the best one.

For input purposes, we'll abbreviate ranks and suits as follows (letters may be either upper- or lower-case):

Ranks: 2 3 4 5 6 7 8 9 t j q k a
Suits: c d h s

If the user enters an illegal card or tries to enter the same card twice, the program will ignore the card, issue an error message, and then request another card. Entering the number 0 instead of a card will cause the program to terminate.

A session with the program will have the following appearance:

```
Enter a card: 2s
Enter a card: 5s
Enter a card: 4s
Enter a card: 3s
Enter a card: 6s
Straight flush
```

```
Enter a card: 8c
Enter a card: as
Enter a card: 8c
Duplicate card; ignored.
Enter a card: 7c
Enter a card: ad
Enter a card: 3h
Pair
```

```
Enter a card: 6s
Enter a card: d2
Bad card; ignored.
Enter a card: 2d
Enter a card: 9c
Enter a card: 4h
Enter a card: ts
```

High card

Enter a card: 0

From this description of the program, we see that it has three tasks:

Read a hand of five cards.

Analyze the hand for pairs, straights, and so forth.

Print the classification of the hand.

We'll divide the program into three functions—`read_cards`, `analyze_hand`, and `print_result`—that perform these three tasks. `main` does nothing but call these functions inside an endless loop. The functions will need to share a fairly large amount of information, so we'll have them communicate through external variables. `read_cards` will store information about the hand into several external variables. `analyze_hand` will then examine these variables, storing its findings into other external variables for the benefit of `print_result`.

Based on this preliminary design, we can begin to sketch an outline of the program:

```

/* #include directives go here */

/* #define directives go here */

/* declarations of external variables go here */

/* prototypes */
void read_cards(void);
void analyze_hand(void);
void print_result(void);

/*****
 * main: Calls read_cards, analyze_hand, and print_result *
 *       repeatedly. *
 *****/
int main(void)
{
    for (;;) {
        read_cards();
        analyze_hand();
        print_result();
    }
}

/*****
 * read_cards: Reads the cards into external variables; *
 *             checks for bad cards and duplicate cards. *
 *****/
void read_cards(void)
{
    ...
}

```



```

/*****
 * analyze_hand: Determines whether the hand contains a
 *               straight, a flush, four-of-a-kind,
 *               and/or three-of-a-kind; determines the
 *               number of pairs; stores the results into
 *               external variables.
 *****/
void analyze_hand(void)
{
    ...
}

/*****
 * print_result: Notifies the user of the result, using
 *               the external variables set by
 *               analyze_hand.
 *****/
void print_result(void)
{
    ...
}

```

The most pressing question that remains is how to represent the hand of cards. Let's see what operations `read_cards` and `analyze_hand` will perform on the hand. During the analysis of the hand, `analyze_hand` will need to know how many cards are in each rank and each suit. This suggests that we use two arrays, `num_in_rank` and `num_in_suit`. The value of `num_in_rank[r]` will be the number of cards with rank `r`, and the value of `num_in_suit[s]` will be the number of cards with suit `s`. (We'll encode ranks as numbers between 0 and 12, and suits as numbers between 0 and 3.) We'll also need a third array, `card_exists`, so that `read_cards` can detect duplicate cards. Each time `read_cards` reads a card with rank `r` and suit `s`, it checks whether the value of `card_exists[r][s]` is true. If so, the card was previously entered; if not, `read_cards` assigns true to `card_exists[r][s]`.

Both the `read_cards` function and the `analyze_hand` function will need access to the `num_in_rank` and `num_in_suit` arrays, so I'll make them external variables. The `card_exists` array is used only by `read_cards`, so it can be local to that function. As a rule, variables should be made external only if necessary.

Having decided on the major data structures, we can now finish the program:

```

poker.c /* Classifies a poker hand */

#include <stdbool.h> /* C99 only */
#include <stdio.h>
#include <stdlib.h>

#define NUM_RANKS 13
#define NUM_SUITS 4
#define NUM_CARDS 5

```

```

/* external variables */
int num_in_rank[NUM_RANKS];
int num_in_suit[NUM_SUITS];
bool straight, flush, four, three;
int pairs; /* can be 0, 1, or 2 */

/* prototypes */
void read_cards(void);
void analyze_hand(void);
void print_result(void);

/*****
 * main: Calls read_cards, analyze_hand, and print_result *
 * repeatedly. *
 *****/
int main(void)
{
    for (;;) {
        read_cards();
        analyze_hand();
        print_result();
    }
}

/*****
 * read_cards: Reads the cards into the external *
 * variables num_in_rank and num_in_suit; *
 * checks for bad cards and duplicate cards. *
 *****/
void read_cards(void)
{
    bool card_exists[NUM_RANKS][NUM_SUITS];
    char ch, rank_ch, suit_ch;
    int rank, suit;
    bool bad_card;
    int cards_read = 0;

    for (rank = 0; rank < NUM_RANKS; rank++) {
        num_in_rank[rank] = 0;
        for (suit = 0; suit < NUM_SUITS; suit++)
            card_exists[rank][suit] = false;
    }

    for (suit = 0; suit < NUM_SUITS; suit++)
        num_in_suit[suit] = 0;

    while (cards_read < NUM_CARDS) {
        bad_card = false;

        printf("Enter a card: ");

        rank_ch = getchar();
        switch (rank_ch) {

```



```

        case '0':          exit(EXIT_SUCCESS);
        case '2':          rank = 0; break;
        case '3':          rank = 1; break;
        case '4':          rank = 2; break;
        case '5':          rank = 3; break;
        case '6':          rank = 4; break;
        case '7':          rank = 5; break;
        case '8':          rank = 6; break;
        case '9':          rank = 7; break;
        case 't': case 'T': rank = 8; break;
        case 'j': case 'J': rank = 9; break;
        case 'q': case 'Q': rank = 10; break;
        case 'k': case 'K': rank = 11; break;
        case 'a': case 'A': rank = 12; break;
        default:          bad_card = true;
    }

    suit_ch = getchar();
    switch (suit_ch) {
        case 'c': case 'C': suit = 0; break;
        case 'd': case 'D': suit = 1; break;
        case 'h': case 'H': suit = 2; break;
        case 's': case 'S': suit = 3; break;
        default:          bad_card = true;
    }

    while ((ch = getchar()) != '\n')
        if (ch != ' ') bad_card = true;

    if (bad_card)
        printf("Bad card; ignored.\n");
    else if (card_exists[rank][suit])
        printf("Duplicate card; ignored.\n");
    else {
        num_in_rank[rank]++;
        num_in_suit[suit]++;
        card_exists[rank][suit] = true;
        cards_read++;
    }
}

/*****
 * analyze_hand: Determines whether the hand contains a
 *               straight, a flush, four-of-a-kind,
 *               and/or three-of-a-kind; determines the
 *               number of pairs; stores the results into
 *               the external variables straight, flush,
 *               four, three, and pairs.
 *****/
void analyze_hand(void)
{
    int num_consec = 0;
    int rank, suit;

```

```

straight = false;
flush = false;
four = false;
three = false;
pairs = 0;

/* check for flush */
for (suit = 0; suit < NUM_SUITS; suit++)
    if (num_in_suit[suit] == NUM_CARDS)
        flush = true;

/* check for straight */
rank = 0;
while (num_in_rank[rank] == 0) rank++;
for (; rank < NUM_RANKS && num_in_rank[rank] > 0; rank++)
    num_consec++;
if (num_consec == NUM_CARDS) {
    straight = true;
    return;
}

/* check for 4-of-a-kind, 3-of-a-kind, and pairs */
for (rank = 0; rank < NUM_RANKS; rank++) {
    if (num_in_rank[rank] == 4) four = true;
    if (num_in_rank[rank] == 3) three = true;
    if (num_in_rank[rank] == 2) pairs++;
}
}

/*****
 * print_result: Prints the classification of the hand,
 *               based on the values of the external
 *               variables straight, flush, four, three,
 *               and pairs.
 *****/
void print_result(void)
{
    if (straight && flush) printf("Straight flush");
    else if (four)        printf("Four of a kind");
    else if (three &&
             pairs == 1) printf("Full house");
    else if (flush)      printf("Flush");
    else if (straight)   printf("Straight");
    else if (three)      printf("Three of a kind");
    else if (pairs == 2) printf("Two pairs");
    else if (pairs == 1) printf("Pair");
    else                 printf("High card");

    printf("\n\n");
}

```

Notice the use of the `exit` function in `read_cards` (in case '0' of the first switch statement). `exit` is convenient for this program because of its ability to terminate execution from anywhere in the program.

Q & A

Q: What impact do local variables with static storage duration have on recursive functions? [p. 220]

A: When a function is called recursively, fresh copies are made of its automatic variables for each call. This doesn't occur for static variables, though. Instead, all calls of the function share the *same* static variables.

Q: In the following example, *j* is initialized to the same value as *i*, but there are two variables named *i*:

```
int i = 1;

void f(void)
{
    int j = i;
    int i = 2;
    ...
}
```

Is this code legal? If so, what is *j*'s initial value, 1 or 2?

A: The code is indeed legal. The scope of a local variable doesn't begin until its declaration. Therefore, the declaration of *j* refers to the external variable named *i*. The initial value of *j* will be 1.

Exercises

Section 10.4

1. The following program outline shows only function definitions and variable declarations.

```
int a;

void f(int b)
{
    int c;
}

void g(void)
{
    int d;
    {
        int e;
    }
}

int main(void)
{
    int f;
}
```

For each of the following scopes, list all variable and parameter names visible in that scope:

- (a) The `f` function
- (b) The `g` function
- (c) The block in which `e` is declared
- (d) The `main` function

2. The following program outline shows only function definitions and variable declarations.

```
int b, c;
void f(void)
{
    int b, d;
}
void g(int a)
{
    int c;
    {
        int a, d;
    }
}
int main(void)
{
    int c, d;
}
```

For each of the following scopes, list all variable and parameter names visible in that scope. If there's more than one variable or parameter with the same name, indicate which one is visible.

- (a) The `f` function
- (b) The `g` function
- (c) The block in which `a` and `d` are declared
- (d) The `main` function

- *3. Suppose that a program has only one function (`main`). How many different variables named `i` could this program contain?

Programming Projects

1. Modify the stack example of Section 10.2 so that it stores characters instead of integers. Next, add a `main` function that asks the user to enter a series of parentheses and/or braces, then indicates whether or not they're properly nested:

```
Enter parentheses and/or braces: (((){}{()})
Parentheses/braces are nested properly
```

Hint: As the program reads characters, have it push each left parenthesis or left brace. When it reads a right parenthesis or brace, have it pop the stack and check that the item popped is a matching parenthesis or brace. (If not, the parentheses/braces aren't nested properly.) When the program reads the new-line character, have it check whether the stack is empty; if so, the parentheses/braces are matched. If the stack *isn't* empty (or if `stack_underflow` is ever

called), the parentheses/braces aren't matched. If `stack_overflow` is called, have the program print the message `Stack overflow` and terminate immediately.

2. Modify the `poker.c` program of Section 10.5 by moving the `num_in_rank` and `num_in_suit` arrays into `main`, which will pass them as arguments to `read_cards` and `analyze_hand`.
- Ⓜ 3. Remove the `num_in_rank`, `num_in_suit`, and `card_exists` arrays from the `poker.c` program of Section 10.5. Have the program store the cards in a 5×2 array instead. Each row of the array will represent a card. For example, if the array is named `hand`, then `hand[0][0]` will store the rank of the first card and `hand[0][1]` will store the suit of the first card.
4. Modify the `poker.c` program of Section 10.5 by having it recognize an additional category, "royal flush" (ace, king, queen, jack, ten of the same suit). A royal flush ranks higher than all other hands.
- Ⓜ 5. Modify the `poker.c` program of Section 10.5 by allowing "ace-low" straights (ace, two, three, four, five).
6. Some calculators (notably those from Hewlett-Packard) use a system of writing mathematical expressions known as Reverse Polish Notation (RPN). In this notation, operators are placed *after* their operands instead of *between* their operands. For example, $1 + 2$ would be written `1 2 +` in RPN, and $1 + 2 * 3$ would be written `1 2 3 * +`. RPN expressions can easily be evaluated using a stack. The algorithm involves reading the operators and operands in an expression from left to right, performing the following actions:

When an operand is encountered, push it onto the stack.

When an operator is encountered, pop its operands from the stack, perform the operation on those operands, and then push the result onto the stack.

Write a program that evaluates RPN expressions. The operands will be single-digit integers. The operators are `+`, `-`, `*`, `/`, and `=`. The `=` operator causes the top stack item to be displayed; afterwards, the stack is cleared and the user is prompted to enter another expression. The process continues until the user enters a character that is not an operator or operand:

```
Enter an RPN expression: 1 2 3 * + =
Value of expression: 7
Enter an RPN expression: 5 8 * 4 9 - / =
Value of expression: -8
Enter an RPN expression: q
```

If the stack overflows, the program will display the message `Expression is too complex` and terminate. If the stack underflows (because of an expression such as `1 2 ++`), the program will display the message `Not enough operands in expression` and terminate. *Hints:* Incorporate the stack code from Section 10.2 into your program. Use `scanf("%c", &ch)` to read the operators and operands.

7. Write a program that prompts the user for a number and then displays the number, using characters to simulate the effect of a seven-segment display:

```
Enter a number: 491-9014
```

```

|_| |_| |_| |_| |_|
|_| |_| |_| |_| |_|
|_| |_| |_| |_| |_|
|_| |_| |_| |_| |_|
|_| |_| |_| |_| |_|
|_| |_| |_| |_| |_|
|_| |_| |_| |_| |_|

```

Characters other than digits should be ignored. Write the program so that the maximum number of digits is controlled by a macro named `MAX_DIGITS`, which has the value 10. If

the number contains more than this number of digits, the extra digits are ignored. *Hints:* Use two external arrays. One is the `segments` array (see Exercise 6 in Chapter 8), which stores data representing the correspondence between digits and segments. The other array, `digits`, will be an array of characters with 4 rows (since each segmented digit is four characters high) and `MAX_DIGITS * 4` columns (digits are three characters wide, but a space is needed between digits for readability). Write your program as four functions: `main`, `clear_digits_array`, `process_digit`, and `print_digits_array`. Here are the prototypes for the latter three functions:

```
void clear_digits_array(void);  
void process_digit(int digit, int position);  
void print_digits_array(void);
```

`clear_digits_array` will store blank characters into all elements of the `digits` array, `process_digit` will store the seven-segment representation of `digit` into a specified position in the `digits` array (positions range from 0 to `MAX_DIGITS - 1`). `print_digits_array` will display the rows of the `digits` array, each on a single line, producing output such as that shown in the example.

APPENDIX A

C Operators

<i>Precedence</i>	<i>Name</i>	<i>Symbol(s)</i>	<i>Associativity</i>
1	Array subscripting	[]	Left
1	Function call	()	Left
1	Structure and union member	. ->	Left
1	Increment (postfix)	++	Left
1	Decrement (postfix)	--	Left
2	Increment (prefix)	++	Right
2	Decrement (prefix)	--	Right
2	Address	&	Right
2	Indirection	*	Right
2	Unary plus	+	Right
2	Unary minus	-	Right
2	Bitwise complement	~	Right
2	Logical negation	!	Right
2	Size	sizeof	Right
3	Cast	()	Right
4	Multiplicative	* / %	Left
5	Additive	+ -	Left
6	Bitwise shift	<< >>	Left
7	Relational	< > <= >=	Left
8	Equality	== !=	Left
9	Bitwise <i>and</i>	&	Left
10	Bitwise exclusive <i>or</i>	^	Left
11	Bitwise inclusive <i>or</i>		Left
12	Logical <i>and</i>	&&	Left
13	Logical <i>or</i>		Left
14	Conditional	? :	Right
15	Assignment	= *= /= %=	Right
		+= -= <<= >>=	
		&= ^= =	
16	Comma	,	Left

APPENDIX B

C99 versus C89

This appendix lists many of the most significant differences between C89 and C99. (The smaller differences are too numerous to mention here.) The headings indicate which chapter contains the primary discussion of each C99 feature. Some of the changes attributed to C99 actually occurred earlier, in Amendment 1 to the C89 standard; these changes are marked “Amendment 1.”

2 C Fundamentals

// comments C99 adds a second kind of comment, which begins with `//`.

identifiers C89 requires compilers to remember the first 31 characters of identifiers; in C99, the requirement is 63 characters. Only the first six characters of names with external linkage are significant in C89. Moreover, the case of letters may not matter. In C99, the first 31 characters are significant, and the case of letters is taken into account.

keywords Five keywords are new in C99: `inline`, `restrict`, `_Bool`, `_Complex`, and `_Imaginary`.

returning from main In C89, if a program reaches the end of the `main` function without executing a `return` statement, the value returned to the operating system is undefined. In C99, if `main` is declared to return an `int`, the program returns 0 to the operating system.

4 Expressions

/ and % operators The C89 standard states that if either operand is negative, the result of an integer division can be rounded either up or down. Moreover, if `i` or `j` is negative, the sign of `i % j` depends on the implementation. In C99, the result of a division is always truncated toward zero and the value of `i % j` has the same sign as `i`.

5 Selection Statements

_Bool type C99 provides a Boolean type named `_Bool`; C89 has no Boolean type.

6 Loops

for statements In C99, the first expression in a `for` statement can be replaced by a declaration, allowing the statement to declare its own control variable(s).

7 Basic Types

long long integer types C99 provides two additional standard integer types, `long long int` and `unsigned long long int`.

extended integer types In addition to the standard integer types, C99 allows implementation-defined extended signed and unsigned integer types.

long long integer constants C99 provides a way to indicate that an integer constant has type `long long int` or `unsigned long long int`.

types of integer constants C99's rules for determining the type of an integer constant are different from those in C89.

hexadecimal floating constants C99 provides a way to write floating constants in hexadecimal.

implicit conversions The rules for implicit conversions in C99 are somewhat different from the rules in C89, primarily because of C99's additional basic types.

8 Arrays

designated initializers C99 supports designated initializers, which can be used to initialize arrays, structures, and unions.

variable-length arrays In C99, the length of an array may be specified by an expression that's not constant, provided that the array doesn't have static storage duration and its declaration doesn't contain an initializer.

9 Functions

no default return type If the return type of a function is omitted in C89, the function is presumed to return a value of type `int`. In C99, it's illegal to omit the return type of a function.

mixed declarations and statements In C89, declarations must precede statements within a block (including the body of a function). In C99, declarations and statements can be mixed, as long as each variable is declared prior to the first statement that uses the variable.

<i>declaration or definition required prior to function call</i>	C99 requires that either a declaration or a definition of a function be present prior to any call of the function. C89 doesn't have this requirement; if a function is called without a prior declaration or definition, the compiler assumes that the function returns an <code>int</code> value.
<i>variable-length array parameters</i>	C99 allows variable-length array parameters. In a function declaration, the <code>*</code> symbol may appear inside brackets to indicate a variable-length array parameter.
<i>static array parameters</i>	C99 allows the use of the word <code>static</code> in the declaration of an array parameter, indicating a minimum length for the first dimension of the array.
<i>compound literals</i>	C99 supports the use of compound literals, which allow the creation of unnamed array and structure values.
<i>declaration of <code>main</code></i>	C99 allows <code>main</code> to be declared in an implementation-defined manner, with a return type other than <code>int</code> and/or parameters other than those specified by the standard.
<i>return statement without expression</i>	In C89, executing a <code>return</code> statement without an expression in a non-void function causes undefined behavior (but only if the program attempts to use the value returned by the function). In C99, such a statement is illegal.

14 The Preprocessor

<i>additional predefined macros</i>	C99 provides several new predefined macros.
<i>empty macro arguments</i>	C99 allows any or all of the arguments in a macro call to be empty, provided that the call contains the correct number of commas.
<i>macros with a variable number of arguments</i>	In C89, a macro must have a fixed number of arguments, if it has any at all. C99 allows macros that take an unlimited number of arguments.
<i><code>__func__</code> identifier</i>	In C99, the <code>__func__</code> identifier behaves like a string variable that stores the name of the currently executing function.
<i>standard pragmas</i>	In C89, there are no standard pragmas. C99 has three: <code>CX_LIMITED_RANGE</code> , <code>FENV_ACCESS</code> , and <code>FP_CONTRACT</code> .
<i><code>_Pragma</code> operator</i>	C99 provides the <code>_Pragma</code> operator, which is used in conjunction with the <code>#pragma</code> directive.

16 Structures, Unions, and Enumerations

<i>structure type compatibility</i>	In C89, structures defined in different files are compatible if their members have the same names and appear in the same order, with corresponding members having
-------------------------------------	---

compatible types. C99 also requires that either both structures have the same tag or neither has a tag.

trailing comma in enumerations In C99, the last constant in an enumeration may be followed by a comma.

17 Advanced Uses of Pointers

restricted pointers C99 has a new keyword, `restrict`, that can appear in the declaration of a pointer.

flexible array members C99 allows the last member of a structure to be an array of unspecified length.

18 Declarations

block scopes for selection and iteration statements In C99, selection statements (`if` and `switch`) and iteration statements (`while`, `do`, and `for`)—along with the “inner” statements that they control—are considered to be blocks.

array, structure, and union initializers In C89, a brace-enclosed initializer for an array, structure, or union must contain only constant expressions. In C99, this restriction applies only if the variable has static storage duration.

inline functions C99 allows functions to be declared `inline`.

21 The Standard Library

<stdbool.h> header The `<stdbool.h>` header, which defines the `bool`, `true`, and `false` macros, is new in C99.

22 Input/Output

...printf conversion specifications The conversion specifications for the `...printf` functions have undergone a number of changes in C99, with new length modifiers, new conversion specifiers, the ability to write infinity and NaN, and support for wide characters. Also, the `%le`, `%lE`, `%lf`, `%lg`, and `%lG` conversions are legal in C99; they caused undefined behavior in C89.

...scanf conversion specifications In C99, the conversion specifications for the `...scanf` functions have new length modifiers, new conversion specifiers, the ability to read infinity and NaN, and support for wide characters.

snprintf function C99 adds the `snprintf` function to the `<stdio.h>` header.

23 Library Support for Numbers and Character Data

additional macros in <float.h> header C99 adds the `DECIMAL_DIG` and `FLT_EVAL_METHOD` macros to the `<float.h>` header.

- additional macros in <limits.h> header* In C99, the `<limits.h>` header contains three new macros that describe the characteristics of the `long long int` types.
- math_errhandling macro* C99 gives implementations a choice of how to inform a program that an error has occurred in a mathematical function: via a value stored in `errno`, via a floating-point exception, or both. The value of the `math_errhandling` macro (defined in `<math.h>`) indicates how errors are signaled by a particular implementation.
- additional functions in <math.h> header* C99 adds two new versions of most `<math.h>` functions, one for `float` and one for `long double`. C99 also adds a number of completely new functions and function-like macros to `<math.h>`.

24 Error Handling

- EILSEQ macro* C99 adds the `EILSEQ` macro to the `<errno.h>` header.

25 International Features

- digraphs* Digraphs, which are two-character symbols that can be used as substitutes for the `[`, `]`, `{`, `}`, `#`, and `##` tokens, are new in C99. (Amendment 1)
- <iso646.h> header* The `<iso646.h>` header, which defines macros that represent operators containing the characters `&`, `|`, `~`, `!`, and `^`, is new in C99. (Amendment 1)
- universal character names* Universal character names, which provide a way to embed UCS characters in the source code of a program, are new in C99.
- <wchar.h> header* The `<wchar.h>` header, which provides functions for wide-character input/output and wide string manipulation, is new in C99. (Amendment 1)
- <wctype.h> header* The `<wctype.h>` header, the wide-character version of `<ctype.h>`, is new in C99. `<wctype.h>` provides functions for classifying and changing the case of wide characters. (Amendment 1)

26 Miscellaneous Library Functions

- va_copy macro* C99 adds a function-like macro named `va_copy` to the `<stdarg.h>` header.
- additional functions in <stdio.h> header* C99 adds the `vsnprintf`, `vfscanf`, `vscanf`, and `vsscanf` functions to the `<stdio.h>` header.
- additional functions in <stdlib.h> header* C99 adds five numeric conversion functions, the `_Exit` function, and `long long` versions of the `abs` and `div` functions to the `<stdlib.h>` header.
- additional strftime conversion specifiers* C99 adds a number of new `strftime` conversion specifiers. It also allows the use of an `E` or `O` character to modify the meaning of certain conversion specifiers.

27 Additional C99 Support for Mathematics

- `<stdint.h>` *header* The `<stdint.h>` header, which declares integer types with specified widths, is new in C99.
- `<inttypes.h>` *header* The `<inttypes.h>` header, which provides macros that are useful for input/output of the integer types in `<stdint.h>`, is new in C99.
- complex types* C99 provides three complex types: `float _Complex`, `double _Complex`, and `long double _Complex`.
- `<complex.h>` *header* The `<complex.h>` header, which provides functions that perform mathematical operations on complex numbers, is new in C99.
- `<tgmath.h>` *header* The `<tgmath.h>` header, which provides type-generic macros that make it easier to call library functions in `<math.h>` and `<complex.h>`, is new in C99.
- `<fenv.h>` *header* The `<fenv.h>` header, which gives programs access to floating-point status flags and control modes, is new in C99.

APPENDIX C

C89 versus K&R C

This appendix lists most of the significant differences between C89 and K&R C (the language described in the first edition of Kernighan and Ritchie's *The C Programming Language*). The headings indicate which chapter of this book discusses each C89 feature. This appendix doesn't address the C library, which has changed much over the years. For other (less important) differences between C89 and K&R C, consult Appendices A and C in the second edition of K&R.

Most of today's C compilers can handle all of C89, but this appendix is useful if you happen to encounter older programs that were originally written for pre-C89 compilers.

2 C Fundamentals

- identifiers* In K&R C, only the first eight characters of an identifier are significant.
- keywords* K&R C lacks the keywords `const`, `enum`, `signed`, `void`, and `volatile`. In K&R C, the word `entry` is a keyword.

4 Expressions

- unary +* K&R C doesn't support the unary `+` operator.

5 Selection Statements

- switch* In K&R C, the controlling expression (and case labels) in a `switch` statement must have type `int` after promotion. In C89, the expression and labels may be of any integral type, including `unsigned int` and `long int`.

7 Basic Types

<i>unsigned types</i>	K&R C provides only one unsigned type (<code>unsigned int</code>).
<i>signed</i>	K&R C doesn't support the <code>signed</code> type specifier.
<i>number suffixes</i>	K&R C doesn't support the <code>U</code> (or <code>u</code>) suffix to specify that an integer constant is unsigned, nor does it support the <code>F</code> (or <code>f</code>) suffix to indicate that a floating constant is to be stored as a <code>float</code> value instead of a <code>double</code> value. In K&R C, the <code>L</code> (or <code>l</code>) suffix can't be used with floating constants.
<i>long float</i>	K&R C allows the use of <code>long float</code> as a synonym for <code>double</code> ; this usage isn't legal in C89.
<i>long double</i>	K&R C doesn't support the <code>long double</code> type.
<i>escape sequences</i>	The escape sequences <code>\a</code> , <code>\v</code> , and <code>\?</code> don't exist in K&R C. Also, K&R C doesn't support hexadecimal escape sequences.
<i>size_t</i>	In K&R C, the <code>sizeof</code> operator returns a value of type <code>int</code> ; in C89, it returns a value of type <code>size_t</code> .
<i>usual arithmetic conversions</i>	K&R C requires that <code>float</code> operands be converted to <code>double</code> . Also, K&R C specifies that combining a shorter unsigned integer with a longer signed integer always produces an unsigned result.

9 Functions

function definitions In a C89 function definition, the types of the parameters are included in the parameter list:

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

K&R C requires that the types of parameters be specified in separate lists:

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

function declarations A C89 function declaration (prototype) specifies the types of the function's parameters (and the names as well, if desired):

```
double square(double x);
double square(double); /* alternate form */
int rand(void); /* no parameters */
```


A K&R C function declaration omits all information about parameters:

```
double square();
int rand();
```

function calls When a K&R C definition or declaration is used, the compiler doesn't check that the function is called with arguments of the proper number and type. Furthermore, the arguments aren't automatically converted to the types of the corresponding parameters. Instead, the integral promotions are performed, and `float` arguments are converted to `double`.

`void` K&R C doesn't support the `void` type.

12 Pointers and Arrays

pointer subtraction Subtracting two pointers produces an `int` value in K&R C but a `ptrdiff_t` value in C89.

13 Strings

string literals In K&R C, adjacent string literals aren't concatenated. Also, K&R C doesn't prohibit the modification of string literals.

string initialization In K&R C, an initializer for a character array of length n is limited to $n - 1$ characters (leaving room for a null character at the end). C89 allows the initializer to have length n .

14 The Preprocessor

`#elif`, `#error`,
`#pragma` K&R C doesn't support the `#elif`, `#error`, and `#pragma` directives.

`#`, `##`, `defined` K&R C doesn't support the `#`, `##`, and `defined` operators.

16 Structures, Unions, and Enumerations

structure and union members and tags In C89, each structure and union has its own name space for members; structure and union tags are kept in a separate name space. K&R C uses a single name space for members and tags, so members can't have the same name (with some exceptions), and members and tags can't overlap.

whole-structure operations K&R C doesn't allow structures to be assigned, passed as arguments, or returned by functions.

enumerations K&R C doesn't support enumerations.

17 Advanced Uses of Pointers

- void ** In C89, *void ** is used as a “generic” pointer type; for example, `malloc` returns a value of type *void **. In K&R C, *char ** is used for this purpose.
- pointer mixing* K&R C allows pointers of different types to be mixed in assignments and comparisons. In C89, pointers of type *void ** can be mixed with pointers of other types, but any other mixing isn’t allowed without casting. Similarly, K&R C allows the mixing of integers and pointers in assignments and comparisons; C89 requires casting.
- pointers to functions* If *pf* is a pointer to a function, C89 permits using either `(*pf) (...)` or `pf (...)` to call the function. K&R C allows only `(*pf) (...)`.

18 Declarations

- const and volatile* K&R C doesn’t support the `const` and `volatile` type qualifiers.
- initialization of arrays, structures, and unions* K&R C doesn’t allow the initialization of automatic arrays and structures, nor does it allow initialization of unions (regardless of storage duration).

25 International Features

- wide characters* K&R C doesn’t support wide character constants and wide string literals.
- trigraph sequences* K&R C doesn’t support trigraph sequences.

26 Miscellaneous Library Functions

- variable arguments* K&R C doesn’t provide a portable way to write functions with a variable number of arguments, and it lacks the `...` (ellipsis) notation.

APPENDIX D

Standard Library Functions

This appendix describes all library functions supported by C89 and C99.* When using this appendix, please keep the following points in mind:

- In the interest of brevity and clarity, I've omitted many details. Some functions (notably `printf` and `scanf` and their variants) are covered in depth elsewhere in the book, so their descriptions here are minimal. For more information about a function (including examples of how it's used), see the section(s) listed in *italic* at the lower right corner of the function's description.
- As in other parts of the book, italics are used to indicate C99 differences. The names and prototypes of functions that were added in C99 are shown in italics. Changes to C89 prototypes (the addition of the word `restrict` to the declaration of certain parameters) are also italicized.
- Function-like macros are included in this appendix (with the exception of the type-generic macros in `<tgmath.h>`). Each prototype for a macro is followed by the word *macro*.
- In C99, some `<math.h>` functions have three versions (one each for `float`, `double`, and `long double`). All three are grouped into a single entry, under the name of the `double` version. For example, there's only one entry (under `acos`) for the `acos`, `acosf`, and `acosl` functions. The name of each additional version (`acosf` and `acosl`, in this example) appears to the left of its prototype. The `<complex.h>` functions, which also come in three versions, are treated in a similar fashion.
- Most of the `<wchar.h>` functions are wide-character versions of functions found in other headers. Unless there's a significant difference in behavior, the

*This material is adapted from international standard ISO/IEC 9899:1999.

description of each wide-character function simply refers the reader to the corresponding function found elsewhere.

- If some aspect of a function's behavior is described as *implementation-defined*, that means that it depends on how the C library is implemented. The function will always behave consistently, but the results may vary from one system to another. (In other words, check the manual to see what happens.) *Undefined* behavior, on the other hand, is bad news: not only may the behavior vary between systems, but the program may act strangely or even crash.
- The descriptions of many `<math.h>` functions refer to the terms *domain error* and *range error*. The way in which these errors are indicated changed between C89 and C99. For the C89 treatment of these errors, see Section 23.3. For the C99 treatment, see Section 23.4.
- The behavior of the following functions is affected by the current locale:

<code><ctype.h></code>	All functions
<code><stdio.h></code>	Formatted input/output functions
<code><stdlib.h></code>	Multibyte/wide-character conversion functions, numeric conversion functions
<code><string.h></code>	<code>strcoll</code> , <code>strxfrm</code>
<code><time.h></code>	<code>strftime</code>
<code><wchar.h></code>	<code>wscoll</code> , <code>wcsftime</code> , <code>wcsxfrm</code> , formatted input/output functions, numeric conversion functions, extended multibyte/wide-character conversion functions
<code><wctype.h></code>	All functions

The `isalpha` function, for example, usually checks whether a character lies between `a` and `z` or `A` and `Z`. In some locales, other characters are considered alphabetic as well.

abort *Abort Program* `<stdlib.h>`

```
void abort(void);
```

Raises the SIGABRT signal. If the signal isn't caught (or if the signal handler returns), the program terminates abnormally and returns an implementation-defined code indicating unsuccessful termination. Whether output buffers are flushed, open streams are closed, or temporary files are removed is implementation-defined.

26.2

abs *Integer Absolute Value* `<stdlib.h>`

```
int abs(int j);
```

Returns Absolute value of `j`. The behavior is undefined if the absolute value of `j` can't be represented.

26.2

acos *Arc Cosine* `<math.h>`

```
double acos(double x);
```

```
acosf float acosf(float x);
```

```
acosl long double acosl(long double x);
```


Returns Arc cosine of x ; the return value is in the range 0 to π . A domain error occurs if x isn't between -1 and $+1$. 23.3

acosh *Arc Hyperbolic Cosine (C99)* <math.h>

`double acosh(double x);`

acoshf `float acoshf(float x);`

acoshl `long double acoshl(long double x);`

Returns Arc hyperbolic cosine of x ; the return value is in the range 0 to $+\infty$. A domain error occurs if x is less than 1. 23.4

asctime *Convert Broken-Down Time to String* <time.h>

`char *asctime(const struct tm *timeptr);`

Returns A pointer to a null-terminated string of the form

Sun Jun 3 17:48:34 2007\n

constructed from the broken-down time in the structure pointed to by `timeptr`.

26.3

asin *Arc Sine* <math.h>

`double asin(double x);`

asinf `float asinf(float x);`

asinl `long double asinl(long double x);`

Returns Arc sine of x ; the return value is in the range $-\pi/2$ to $+\pi/2$. A domain error occurs if x isn't between -1 and $+1$. 23.3

asinh *Arc Hyperbolic Sine (C99)* <math.h>

`double asinh(double x);`

asinhf `float asinhf(float x);`

asinh1 `long double asinh1(long double x);`

Returns Arc hyperbolic sine of x . 23.4

assert *Assert Truth of Expression* <assert.h>

`void assert(scalar expression);` *macro*

If the value of `expression` is nonzero, `assert` does nothing. If the value is zero, `assert` writes a message to `stderr` (specifying the text of `expression`, the name of the source file containing the `assert`, and the line number of the `assert`); it then terminates the program by calling `abort`. To disable `assert`, define the macro `NDEBUG` before including `<assert.h>`. *C99 changes:* The argument is allowed to have any scalar type; C89 specifies that the type is `int`. Also, C99 requires that the message written by `assert` include the name of the function in which the `assert` appears; C89 doesn't have this requirement. 24.1

atan *Arc Tangent* <math.h>

`double atan(double x);`

atanf `float atanf(float x);`

atan1	<code>long double atan1(long double x);</code>	
<i>Returns</i>	Arc tangent of x ; the return value is in the range $-\pi/2$ to $+\pi/2$.	23.3
atan2	<i>Arc Tangent of Quotient</i>	<math.h>
	<code>double atan2(double y, double x);</code>	
atan2f	<code>float atan2f(float y, float x);</code>	
atan2l	<code>long double atan2l(long double y, long double x);</code>	
<i>Returns</i>	Arc tangent of y/x ; the return value is in the range $-\pi$ to $+\pi$. A domain error may occur if x and y are both zero.	23.3
atanh	<i>Arc Hyperbolic Tangent (C99)</i>	<math.h>
	<code>double atanh(double x);</code>	
atanhf	<code>float atanhf(float x);</code>	
atanhl	<code>long double atanh1(long double x);</code>	
<i>Returns</i>	Arc hyperbolic tangent of x . A domain error occurs if x is not between -1 and $+1$. A range error may occur if x is equal to -1 or $+1$.	23.4
atexit	<i>Register Function to Be Called at Program Exit</i>	<stdlib.h>
	<code>int atexit(void (*func)(void));</code>	
	Registers the function pointed to by <code>func</code> as a termination function. The function will be called if the program terminates normally (via <code>return</code> or <code>exit</code> but not <code>abort</code>).	
<i>Returns</i>	Zero if successful, nonzero if unsuccessful (an implementation-dependent limit has been reached).	26.2
atof	<i>Convert String to Floating-Point Number</i>	<stdlib.h>
	<code>double atof(const char *nptr);</code>	
<i>Returns</i>	A double value corresponding to the longest initial part of the string pointed to by <code>nptr</code> that has the form of a floating-point number. Returns zero if no conversion could be performed. The function's behavior is undefined if the number can't be represented.	26.2
atoi	<i>Convert String to Integer</i>	<stdlib.h>
	<code>int atoi(const char *nptr);</code>	
<i>Returns</i>	An <code>int</code> value corresponding to the longest initial part of the string pointed to by <code>nptr</code> that has the form of an integer. Returns zero if no conversion could be performed. The function's behavior is undefined if the number can't be represented.	26.2
atol	<i>Convert String to Long Integer</i>	<stdlib.h>
	<code>long int atol(const char *nptr);</code>	
<i>Returns</i>	A long <code>int</code> value corresponding to the longest initial part of the string pointed to by <code>nptr</code> that has the form of an integer. Returns zero if no conversion	

could be performed. The function's behavior is undefined if the number can't be represented. 26.2

atoll *Convert String to Long Long Integer (C99)* <stdlib.h>

```
long long int atoll(const char *nptr);
```

Returns A long long int value corresponding to the longest initial part of the string pointed to by `nptr` that has the form of an integer. Returns zero if no conversion could be performed. The function's behavior is undefined if the number can't be represented. 26.2

bsearch *Binary Search* <stdlib.h>

```
void *bsearch(const void *key, const void *base,
             size_t memb, size_t size,
             int (*compar)(const void *,
                          const void *));
```

Searches for the value pointed to by `key` in the sorted array pointed to by `base`. The array has `nmemb` elements, each `size` bytes long. `compar` is a pointer to a comparison function. When passed pointers to the key and an array element, in that order, the comparison function must return a negative, zero, or positive integer, depending on whether the key is less than, equal to, or greater than the array element.

Returns A pointer to an array element that tests equal to the key. Returns a null pointer if the key isn't found. 26.2

btowc *Convert Byte to Wide Character (C99)* <wchar.h>

```
wint_t btowc(int c);
```

Returns Wide-character representation of `c`. Returns `WEOF` if `c` is equal to `EOF` or if `c` (when cast to `unsigned char`) isn't a valid single-byte character in the initial shift state. 25.5

cabs *Complex Absolute Value (C99)* <complex.h>

```
double cabs(double complex z);
```

```
cabsf float cabsf(float complex z);
```

```
cabsl long double cabsl(long double complex z);
```

Returns Complex absolute value of `z`. 27.4

cacos *Complex Arc Cosine (C99)* <complex.h>

```
double complex cacos(double complex z);
```

```
cacosf float complex cacosf(float complex z);
```

```
cacosl long double complex cacosl(long double complex z);
```

Returns Complex arc cosine of `z`, with branch cuts outside the interval $[-1, +1]$ along the real axis. The return value lies in a strip mathematically unbounded along the imaginary axis and in the interval $[0, \pi]$ along the real axis. 27.4

<i>cacosh</i>	<i>Complex Arc Hyperbolic Cosine (C99)</i>	<i><complex.h></i>
	<i>double complex cacosh(double complex z);</i>	
<i>cacoshf</i>	<i>float complex cacoshf(float complex z);</i>	
<i>cacoshl</i>	<i>long double complex cacoshl(long double complex z);</i>	
<i>Returns</i>	Complex arc hyperbolic cosine of <i>z</i> , with a branch cut at values less than 1 along the real axis. The return value lies in a half-strip of nonnegative values along the real axis and in the interval $[-i\pi, +i\pi]$ along the imaginary axis. 27.4	

<i>calloc</i>	<i>Allocate and Clear Memory Block</i>	<i><stdlib.h></i>
	<i>void *calloc(size_t nmemb, size_t size);</i>	
	Allocates a block of memory for an array with <i>nmemb</i> elements, each with <i>size</i> bytes. The block is cleared by setting all bits to zero.	
<i>Returns</i>	A pointer to the beginning of the block. Returns a null pointer if a block of the requested size can't be allocated. 17.3	

<i>carg</i>	<i>Complex Argument (C99)</i>	<i><complex.h></i>
	<i>double carg(double complex z);</i>	
<i>cargf</i>	<i>float cargf(float complex z);</i>	
<i>cargl</i>	<i>long double cargl(long double complex z);</i>	
<i>Returns</i>	Argument (phase angle) of <i>z</i> , with a branch cut along the negative real axis. The return value lies in the interval $[-\pi, +\pi]$. 27.4	

<i>casin</i>	<i>Complex Arc Sine (C99)</i>	<i><complex.h></i>
	<i>double complex casin(double complex z);</i>	
<i>casinf</i>	<i>float complex casinf(float complex z);</i>	
<i>casinl</i>	<i>long double complex casinl(long double complex z);</i>	
<i>Returns</i>	Complex arc sine of <i>z</i> , with branch cuts outside the interval $[-1, +1]$ along the real axis. The return value lies in a strip mathematically unbounded along the imaginary axis and in the interval $[-\pi/2, +\pi/2]$ along the real axis. 27.4	

<i>casinh</i>	<i>Complex Arc Hyperbolic Sine (C99)</i>	<i><complex.h></i>
	<i>double complex casinh(double complex z);</i>	
<i>casinhf</i>	<i>float complex casinhf(float complex z);</i>	
<i>casinhl</i>	<i>long double complex casinhl(long double complex z);</i>	
<i>Returns</i>	Complex arc hyperbolic sine of <i>z</i> , with branch cuts outside the interval $[-i, +i]$ along the imaginary axis. The return value lies in a strip mathematically unbounded along the real axis and in the interval $[-i\pi/2, +i\pi/2]$ along the imaginary axis. 27.4	

<i>catan</i>	<i>Complex Arc Tangent (C99)</i>	<i><complex.h></i>
	<i>double complex catan(double complex z);</i>	
<i>catanf</i>	<i>float complex catanf(float complex z);</i>	
<i>catanl</i>	<i>long double complex catanl(long double complex z);</i>	

Returns Complex arc tangent of z , with branch cuts outside the interval $[-i, +i]$ along the imaginary axis. The return value lies in a strip mathematically unbounded along the imaginary axis and in the interval $[-\pi/2, +\pi/2]$ along the real axis. 27.4

catanh *Complex Arc Hyperbolic Tangent (C99)* <complex.h>

`double complex catanh(double complex z);`

`catanhf float complex catanhf(float complex z);`

`catanhl long double complex catanhl(long double complex z);`

Returns Complex arc hyperbolic tangent of z , with branch cuts outside the interval $[-1, +1]$ along the real axis. The return value lies in a strip mathematically unbounded along the real axis and in the interval $[-i\pi/2, +i\pi/2]$ along the imaginary axis. 27.4

cbirt *Cube Root (C99)* <math.h>

`double cbirt(double x);`

`cbirtf float cbirtf(float x);`

`cbirtl long double cbirtl(long double x);`

Returns Real cube root of x . 23.4

ccos *Complex Cosine (C99)* <complex.h>

`double complex ccos(double complex z);`

`ccosf float complex ccosf(float complex z);`

`ccosl long double complex ccosl(long double complex z);`

Returns Complex cosine of z . 27.4

ccosh *Complex Hyperbolic Cosine (C99)* <complex.h>

`double complex ccosh(double complex z);`

`ccoshf float complex ccoshf(float complex z);`

`ccoshl long double complex ccoshl(long double complex z);`

Returns Complex hyperbolic cosine of z . 27.4

ceil *Ceiling* <math.h>

`double ceil(double x);`

`ceilf float ceilf(float x);`

`ceill long double ceill(long double x);`

Returns Smallest integer that is greater than or equal to x . 23.3

cexp *Complex Base- e Exponential (C99)* <complex.h>

`double complex cexp(double complex z);`

`cexpf float complex cexpf(float complex z);`

`cexpl long double complex cexpl(long double complex z);`

Returns Complex base- e exponential of z . 27.4

cimag *Imaginary Part of Complex Number (C99)* <complex.h>

`double cimag(double complex z);`

cimagf	<code>float cimagf(float complex z);</code>		
cimagl	<code>long double cimagl(long double complex z);</code>		
Returns	Imaginary part of <i>z</i> .		27.4
clearerr	<i>Clear Stream Error</i>	<code><stdio.h></code>	
	<code>void clearerr(FILE *stream);</code>		
	Clears the end-of-file and error indicators for the stream pointed to by <i>stream</i> .		22.3
clock	<i>Processor Clock</i>	<code><time.h></code>	
	<code>clock_t clock(void);</code>		
Returns	Elapsed processor time (measured in “clock ticks”) since the beginning of program execution. (To convert into seconds, divide by <code>CLOCKS_PER_SEC</code> .) Returns <code>(clock_t) (-1)</code> if the time is unavailable or can’t be represented.		26.3
clog	<i>Complex Natural Logarithm (C99)</i>	<code><complex.h></code>	
	<code>double complex clog(double complex z);</code>		
clogf	<code>float complex clogf(float complex z);</code>		
clogl	<code>long double complex clogl(long double complex z);</code>		
Returns	Complex natural (base- <i>e</i>) logarithm of <i>z</i> , with a branch cut along the negative real axis. The return value lies in a strip mathematically unbounded along the real axis and in the interval $[-i\pi, +i\pi]$ along the imaginary axis.		27.4
conj	<i>Complex Conjugate (C99)</i>	<code><complex.h></code>	
	<code>double complex conj(double complex z);</code>		
conjf	<code>float complex conjf(float complex z);</code>		
conjl	<code>long double complex conjl(long double complex z);</code>		
Returns	Complex conjugate of <i>z</i> .		27.4
copysign	<i>Copy Sign (C99)</i>	<code><math.h></code>	
	<code>double copysign(double x, double y);</code>		
copysignf	<code>float copysignf(float x, float y);</code>		
copysignl	<code>long double copysignl(long double x, long double y);</code>		
Returns	A value with the magnitude of <i>x</i> and the sign of <i>y</i> .		23.4
COS	<i>Cosine</i>	<code><math.h></code>	
	<code>double cos(double x);</code>		
cosf	<code>float cosf(float x);</code>		
cosl	<code>long double cosl(long double x);</code>		
Returns	Cosine of <i>x</i> (measured in radians).		23.3
cosh	<i>Hyperbolic Cosine</i>	<code><math.h></code>	
	<code>double cosh(double x);</code>		
coshf	<code>float coshf(float x);</code>		

coshl long double coshl(long double x);
 Returns Hyperbolic cosine of x. A range error occurs if the magnitude of x is too large.
 23.3

cpow Complex Power (C99) <complex.h>
 double complex cpow(double complex x,
 double complex y);
cpowf float complex cpowf(float complex x,
 float complex y);
cpowl long double complex cpowl(long double complex x,
 long double complex y);
 Returns x raised to the power y, with a branch cut for the first parameter along the negative real axis.
 27.4

cproj Complex Projection (C99) <complex.h>
 double complex cproj(double complex z);
cprojf float complex cprojf(float complex z);
cprojl long double complex cprojl(long double complex z);
 Returns Projection of z onto the Riemann sphere. z is returned unless one of its parts is infinite, in which case the return value is INFINITY + I * copysign(0.0, cimag(z)).
 27.4

creal Real Part of Complex Number (C99) <complex.h>
 double creal(double complex z);
crealf float crealf(float complex z);
creall long double creall(long double complex z);
 Returns Real part of z.
 27.4

csin Complex Sine (C99) <complex.h>
 double complex csin(double complex z);
csinf float complex csinf(float complex z);
csinl long double complex csinl(long double complex z);
 Returns Complex sine of z.
 27.4

csinh Complex Hyperbolic Sine (C99) <complex.h>
 double complex csinh(double complex z);
csinhf float complex csinhf(float complex z);
csinhl long double complex csinhl(long double complex z);
 Returns Complex hyperbolic sine of z.
 27.4

csqrt Complex Square Root (C99) <complex.h>
 double complex csqrt(double complex z);
csqrtf float complex csqrtf(float complex z);
csqrtl long double complex csqrtl(long double complex z);

Returns Complex square root of z , with a branch cut along the negative real axis. The return value lies in the right half-plane (including the imaginary axis). 27.4

ctan *Complex Tangent (C99)* <complex.h>

`double complex ctan(double complex z);`

ctanf `float complex ctanf(float complex z);`

ctanl `long double complex ctanl(long double complex z);`

Returns Complex tangent of z . 27.4

ctanh *Complex Hyperbolic Tangent (C99)* <complex.h>

`double complex ctanh(double complex z);`

ctanhf `float complex ctanhf(float complex z);`

ctanhl `long double complex ctanhl(long double complex z);`

Returns Complex hyperbolic tangent of z . 27.4

ctime *Convert Calendar Time to String* <time.h>

`char *ctime(const time_t *timer);`

Returns A pointer to a string describing a local time equivalent to the calendar time pointed to by `timer`. Equivalent to `asctime(localtime(timer))`. 26.3

difftime *Time Difference* <time.h>

`double difftime(time_t time1, time_t time0);`

Returns Difference between `time0` (the earlier time) and `time1`, measured in seconds. 26.3

div *Integer Division* <stdlib.h>

`div_t div(int numer, int denom);`

Returns A `div_t` structure containing members named `quot` (the quotient when `numer` is divided by `denom`) and `rem` (the remainder). The behavior is undefined if either part of the result can't be represented. 26.2

erf *Error Function (C99)* <math.h>

`double erf(double x);`

erff `float erff(float x);`

erfl `long double erfl(long double x);`

Returns $\text{erf}(x)$, where erf is the Gaussian error function. 23.4

erfc *Complementary Error Function (C99)* <math.h>

`double erfc(double x);`

erfcf `float erfcf(float x);`

erfc1 `long double erfc1(long double x);`

Returns $\text{erfc}(x) = 1 - \text{erf}(x)$, where erf is the Gaussian error function. A range error occurs if x is too large. 23.4

exit *Exit from Program* <stdlib.h>

void exit(int status);

Calls all functions registered with `atexit`, flushes all output buffers, closes all open streams, removes any files created by `tmpfile`, and terminates the program. The value of `status` indicates whether the program terminated normally. The only portable values for `status` are 0 and `EXIT_SUCCESS` (both indicate successful termination) plus `EXIT_FAILURE` (unsuccessful termination).

9.5, 26.2

_Exit *Exit from Program (C99)* <stdlib.h>

void _Exit(int status);

Causes normal program termination. Doesn't call functions registered with `atexit` or signal handlers registered with `signal`. The status returned is determined in the same way as for `exit`. Whether output buffers are flushed, open streams are closed, or temporary files are removed is implementation-defined.

26.2

exp *Base-e Exponential* <math.h>

double exp(double x);

expf float expf(float x);

expl long double expl(long double x);

Returns e raised to the power x . A range error occurs if the magnitude of x is too large.

23.3

exp2 *Base-2 Exponential (C99)* <math.h>

double exp2(double x);

exp2f float exp2f(float x);

exp2l long double exp2l(long double x);

Returns 2 raised to the power x . A range error occurs if the magnitude of x is too large.

23.4

expm1 *Base-e Exponential Minus 1 (C99)* <math.h>

double expm1(double x);

expm1f float expm1f(float x);

expm1l long double expm1l(long double x);

Returns e raised to the power x , minus 1. A range error occurs if x is too large.

23.4

fabs *Floating Absolute Value* <math.h>

double fabs(double x);

fabsf float fabsf(float x);

fabsl long double fabsl(long double x);

Returns Absolute value of x .

23.3

fclose	<i>Close File</i>	<code><stdio.h></code>
	<code>int fclose(FILE *stream);</code>	
	Closes the stream pointed to by <code>stream</code> . Flushes any unwritten output remaining in the stream's buffer. Deallocates the buffer if it was allocated automatically.	
<i>Returns</i>	Zero if successful, EOF if an error was detected.	22.2

fdim	<i>Positive Difference (C99)</i>	<code><math.h></code>
	<code>double fdim(double x, double y);</code>	
fdimf	<code>float fdimf(float x, float y);</code>	
fdiml	<code>long double fdiml(long double x, long double y);</code>	
<i>Returns</i>	Positive difference of <code>x</code> and <code>y</code> :	
	$\begin{cases} x - y & \text{if } x > y \\ +0 & \text{if } x \leq y \end{cases}$	
	A range error may occur.	23.4

feclearexcept	<i>Clear Floating-Point Exceptions (C99)</i>	<code><fenv.h></code>
	<code>int feclearexcept(int excepts);</code>	
	Attempts to clear the floating-point exceptions represented by <code>excepts</code> .	
<i>Returns</i>	Zero if <code>excepts</code> is zero or if all specified exceptions were successfully cleared; otherwise, returns a nonzero value.	27.6

fegetenv	<i>Get Floating-Point Environment (C99)</i>	<code><fenv.h></code>
	<code>int fegetenv(fenv_t *envp);</code>	
	Attempts to store the current floating-point environment in the object pointed to by <code>envp</code> .	
<i>Returns</i>	Zero if the environment was successfully stored; otherwise, returns a nonzero value.	27.6

fegetexceptflag	<i>Get Floating-Point Exception Flags (C99)</i>	<code><fenv.h></code>
	<code>int fegetexceptflag(fexcept_t *flagp, int excepts);</code>	
	Attempts to retrieve the states of the floating-point status flags represented by <code>excepts</code> and store them in the object pointed to by <code>flagp</code> .	
<i>Returns</i>	Zero if the states of the status flags were successfully stored; otherwise, returns a nonzero value.	27.6

fegetround	<i>Get Floating-Point Rounding Direction (C99)</i>	<code><fenv.h></code>
	<code>int fegetround(void);</code>	
<i>Returns</i>	Value of the rounding-direction macro that represents the current rounding direction. Returns a negative value if the current rounding direction can't be determined or doesn't match any rounding-direction macro.	27.6

<i>feholdexcept</i>	<i>Save Floating-Point Environment (C99)</i>	<fenv.h>
	<i>int feholdexcept(fenv_t *envp);</i>	
	Saves the current floating-point environment in the object pointed to by <i>envp</i> , clears the floating-point status flags, and attempts to install a non-stop mode for all floating-point exceptions.	
<i>Returns</i>	Zero if non-stop floating-point exception handling was successfully installed; otherwise, returns a nonzero value.	27.6
<i>feof</i>	<i>Test for End-of-File</i>	<stdio.h>
	<i>int feof(FILE *stream);</i>	
<i>Returns</i>	A nonzero value if the end-of-file indicator is set for the stream pointed to by <i>stream</i> ; otherwise, returns zero.	22.3
<i>feraiseexcept</i>	<i>Raise Floating-Point Exceptions (C99)</i>	<fenv.h>
	<i>int feraiseexcept(int excepts);</i>	
	Attempts to raise supported floating-point exceptions represented by <i>excepts</i> .	
<i>Returns</i>	Zero if <i>excepts</i> is zero or if all specified exceptions were successfully raised; otherwise, returns a nonzero value.	27.6
<i>ferror</i>	<i>Test for File Error</i>	<stdio.h>
	<i>int ferror(FILE *stream);</i>	
<i>Returns</i>	A nonzero value if the error indicator is set for the stream pointed to by <i>stream</i> ; otherwise, returns zero.	22.3
<i>fesetenv</i>	<i>Set Floating-Point Environment (C99)</i>	<fenv.h>
	<i>int fesetenv(const fenv_t *envp);</i>	
	Attempts to establish the floating-point environment represented by the object pointed to by <i>envp</i> .	
<i>Returns</i>	Zero if the environment was successfully established; otherwise, returns a nonzero value.	27.6
<i>fesetexceptflag</i>	<i>Set Floating-Point Exception Flags (C99)</i>	<fenv.h>
	<i>int fesetexceptflag(const fexcept_t *flagp, int excepts);</i>	
	Attempts to set the floating-point status flags represented by <i>excepts</i> to the states stored in the object pointed to by <i>flagp</i> .	
<i>Returns</i>	Zero if <i>excepts</i> is zero or if all specified exceptions were successfully set; otherwise, returns a nonzero value.	27.6
<i>fesetround</i>	<i>Set Floating-Point Rounding Direction (C99)</i>	<fenv.h>
	<i>int fesetround(int round);</i>	

	Attempts to establish the rounding direction represented by <code>round</code> .	
<i>Returns</i>	Zero if the requested rounding direction was established; otherwise, returns a non-zero value.	27.6
<i>fetestexcept</i>	<i>Test Floating-Point Exception Flags (C99)</i>	<fenv.h>
	<code>int fetestexcept(int excepts);</code>	
<i>Returns</i>	Bitwise <i>or</i> of the floating-point exception macros corresponding to the currently set flags for the exceptions represented by <code>excepts</code> .	27.6
<i>feupdateenv</i>	<i>Update Floating-Point Environment (C99)</i>	<fenv.h>
	<code>int feupdateenv(const fenv_t *envp);</code>	
	Attempts to save the currently raised floating-point exceptions, install the floating-point environment represented by the object pointed to by <code>envp</code> , and then raise the saved exceptions.	
<i>Returns</i>	Zero if all actions were successfully carried out; otherwise, returns a nonzero value.	27.6
<i>fflush</i>	<i>Flush File Buffer</i>	<stdio.h>
	<code>int fflush(FILE *stream);</code>	
	Writes any unwritten data in the buffer associated with <code>stream</code> , which points to a stream that was opened for output or updating. If <code>stream</code> is a null pointer, <code>fflush</code> flushes all streams that have unwritten data stored in a buffer.	
<i>Returns</i>	Zero if successful, EOF if a write error occurs.	22.2
<i>fgetc</i>	<i>Read Character from File</i>	<stdio.h>
	<code>int fgetc(FILE *stream);</code>	
	Reads a character from the stream pointed to by <code>stream</code> .	
<i>Returns</i>	Character read from the stream. If <code>fgetc</code> encounters the end of the stream, it sets the stream's end-of-file indicator and returns EOF. If a read error occurs, <code>fgetc</code> sets the stream's error indicator and returns EOF.	22.4
<i>fgetpos</i>	<i>Get File Position</i>	<stdio.h>
	<code>int fgetpos(FILE * restrict stream, fpos_t * restrict pos);</code>	
	Stores the current position of the stream pointed to by <code>stream</code> in the object pointed to by <code>pos</code> .	
<i>Returns</i>	Zero if successful. If the call fails, returns a nonzero value and stores an implementation-defined positive value in <code>errno</code> .	22.7
<i>fgets</i>	<i>Read String from File</i>	<stdio.h>
	<code>char *fgets(char * restrict s, int n, FILE * restrict stream);</code>	

Reads characters from the stream pointed to by `stream` and stores them in the array pointed to by `s`. Reading stops at the first new-line character (which is stored in the string), when `n - 1` characters have been read, or at end-of-file. `fgets` appends a null character to the string.

Returns `s` (a pointer to the array in which the input is stored). Returns a null pointer if a read error occurs or `fgets` encounters the end of the stream before it has stored any characters. 22.5

fgetcwc *Read Wide Character from File (C99)* <wchar.h>

`wint_t fgetcwc(FILE *stream);`

Wide-character version of `fgetc`. 25.5

fgetws *Read Wide String from File (C99)* <wchar.h>

`wchar_t *fgetws(wchar_t * restrict s, int n,
FILE * restrict stream);`

Wide-character version of `fgets`. 25.5

floor *Floor* <math.h>

`double floor(double x);`

floorf `float floorf(float x);`

floorl `long double floorl(long double x);`

Returns Largest integer that is less than or equal to `x`. 23.3

fma *Floating Multiply-Add (C99)* <math.h>

`double fma(double x, double y, double z);`

fmaf `float fmaf(float x, float y, float z);`

fmal `long double fmal(long double x, long double y,
long double z);`

Returns $(x \times y) + z$. The result is rounded only once, using the rounding mode corresponding to `FLT_ROUNDS`. A range error may occur. 23.4

fmax *Floating Maximum (C99)* <math.h>

`double fmax(double x, double y);`

fmaxf `float fmaxf(float x, float y);`

fmaxl `long double fmaxl(long double x, long double y);`

Returns Maximum of `x` and `y`. If one argument is a NaN and the other is numeric, the numeric value is returned. 23.4

fmin *Floating Minimum (C99)* <math.h>

`double fmin(double x, double y);`

fminf `float fminf(float x, float y);`

fminl `long double fminl(long double x, long double y);`

Returns Minimum of `x` and `y`. If one argument is a NaN and the other is numeric, the numeric value is returned. 23.4

fmod	<i>Floating Modulus</i>	<code><math.h></code>
	<code>double fmod(double x, double y);</code>	
	<code>float fmodf(float x, float y);</code>	
	<code>long double fmodl(long double x, long double y);</code>	
<i>Returns</i>	Remainder when <i>x</i> is divided by <i>y</i> . If <i>y</i> is zero, either a domain error occurs or zero is returned.	23.3

fopen	<i>Open File</i>	<code><stdio.h></code>
	<code>FILE *fopen(const char * restrict filename, const char * restrict mode);</code>	
	Opens the file whose name is pointed to by <i>filename</i> and associates it with a stream. <i>mode</i> specifies the mode in which the file is to be opened. Clears the error and end-of-file indicators for the stream.	
<i>Returns</i>	A file pointer to be used when performing subsequent operations on the file. Returns a null pointer if the file can't be opened.	22.2

fpclassify	<i>Floating-Point Classification (C99)</i>	<code><math.h></code>
	<code>int fpclassify(real-floating x);</code>	<i>macro</i>
<i>Returns</i>	Either <code>FP_INFINITE</code> , <code>FP_NAN</code> , <code>FP_NORMAL</code> , <code>FP_SUBNORMAL</code> , or <code>FP_ZERO</code> , depending on whether <i>x</i> is infinity, not a number, normal, subnormal, or zero, respectively.	23.4

fprintf	<i>Formatted File Write</i>	<code><stdio.h></code>
	<code>int fprintf(FILE * restrict stream, const char * restrict format, ...);</code>	
	Writes output to the stream pointed to by <i>stream</i> . The string pointed to by <i>format</i> specifies how subsequent arguments will be displayed.	
<i>Returns</i>	Number of characters written. Returns a negative value if an error occurs.	22.3

fputc	<i>Write Character to File</i>	<code><stdio.h></code>
	<code>int fputc(int c, FILE *stream);</code>	
	Writes the character <i>c</i> to the stream pointed to by <i>stream</i> .	
<i>Returns</i>	<i>c</i> (the character written). If a write error occurs, <code>fputc</code> sets the stream's error indicator and returns EOF.	22.4

fputs	<i>Write String to File</i>	<code><stdio.h></code>
	<code>int fputs(const char * restrict s, FILE * restrict stream);</code>	
	Writes the string pointed to by <i>s</i> to the stream pointed to by <i>stream</i> .	
<i>Returns</i>	A nonnegative value if successful. Returns EOF if a write error occurs.	22.5

fputc Write Wide Character to File (C99) <wchar.h>

```
wint_t fputc(wchar_t c, FILE *stream);
```

Wide-character version of `fputc`. 25.5

fputws Write Wide String to File (C99) <wchar.h>

```
int fputws(const wchar_t * restrict s,
           FILE * restrict stream);
```

Wide-character version of `fputs`. 25.5

fread Read Block from File <stdio.h>

```
size_t fread(void * restrict ptr, size_t size,
             size_t nmemb, FILE * restrict stream);
```

Attempts to read `nmemb` elements, each `size` bytes long, from the stream pointed to by `stream` and store them in the array pointed to by `ptr`.

Returns Number of elements actually read. This number will be less than `nmemb` if `fread` encounters end-of-file or a read error occurs. Returns zero if either `nmemb` or `size` is zero. 22.6

free Free Memory Block <stdlib.h>

```
void free(void *ptr);
```

Releases the memory block pointed to by `ptr`. (If `ptr` is a null pointer, the call has no effect.) The block must have been allocated by a call of `calloc`, `malloc`, or `realloc`. 17.4

freopen Reopen File <stdio.h>

```
FILE *freopen(const char * restrict filename,
              const char * restrict mode,
              FILE * restrict stream);
```

Closes the file associated with `stream`, then opens the file whose name is pointed to by `filename` and associates it with `stream`. The mode parameter has the same meaning as in a call of `fopen`. *C99 change*: If `filename` is a null pointer, `freopen` attempts to change the stream's mode to that specified by `mode`.

Returns Value of `stream` if the operation succeeds. Returns a null pointer if the file can't be opened. 22.2

frexp Split into Fraction and Exponent <math.h>

```
double frexp(double value, int *exp);
```

```
float frexpf(float value, int *exp);
```

```
long double frexpl(long double value, int *exp);
```

Splits `value` into a fractional part f and an exponent n in such a way that $value = f \times 2^n$

f is normalized so that either $0.5 \leq f < 1$ or $f = 0$. Stores n in the object pointed to by `exp`.

Returns f , the fractional part of value. 23.3

fscanf *Formatted File Read* <stdio.h>

```
int fscanf(FILE * restrict stream,
           const char * restrict format, ...);
```

Reads input items from the stream pointed to by `stream`. The string pointed to by `format` specifies the format of the items to be read. The arguments that follow `format` point to objects in which the items are to be stored.

Returns Number of input items successfully read and stored. Returns EOF if an input failure occurs before any items can be read. 22.3

fseek *File Seek* <stdio.h>

```
int fseek(FILE *stream, long int offset, int whence);
```

Changes the file position indicator for the stream pointed to by `stream`. If `whence` is `SEEK_SET`, the new position is the beginning of the file plus `offset` bytes. If `whence` is `SEEK_CUR`, the new position is the current position plus `offset` bytes. If `whence` is `SEEK_END`, the new position is the end of the file plus `offset` bytes. The value of `offset` may be negative. For text streams, either `offset` must be zero or `whence` must be `SEEK_SET` and `offset` a value obtained by a previous call of `ftell`. For binary streams, `fseek` may not support calls in which `whence` is `SEEK_END`.

Returns Zero if the operation is successful, nonzero otherwise. 22.7

fsetpos *Set File Position* <stdio.h>

```
int fsetpos(FILE *stream, const fpos_t *pos);
```

Sets the file position indicator for the stream pointed to by `stream` according to the value pointed to by `pos` (obtained from a previous call of `fgetpos`).

Returns Zero if successful. If the call fails, returns a nonzero value and stores an implementation-defined positive value in `errno`. 22.7

ftell *Determine File Position* <stdio.h>

```
long int ftell(FILE *stream);
```

Returns Current file position indicator for the stream pointed to by `stream`. If the call fails, returns `-1L` and stores an implementation-defined positive value in `errno`.

22.7

fwide *Get and Set Stream Orientation (C99)* <wchar.h>

```
int fwide(FILE *stream, int mode);
```

Determines the current orientation of a stream and, if desired, attempts to set its orientation. If `mode` is greater than zero, `fwide` tries to make the stream wide-oriented if it has no orientation. If `mode` is less than zero, it tries to make the

stream byte-oriented if it has no orientation. If mode is zero, the orientation is not changed.

Returns A positive value if the stream has wide orientation after the call, a negative value if it has byte orientation, or zero if it has no orientation. 25.5

fwprintf *Wide-Character Formatted File Write (C99)* <wchar.h>

```
int fwprintf(FILE * restrict stream,
             const wchar_t * restrict format, ...);
```

Wide-character version of fprintf. 25.5

fwrite *Write Block to File* <stdio.h>

```
size_t fwrite(const void * restrict ptr, size_t size,
              size_t nmemb, FILE * restrict stream);
```

Writes nmemb elements, each size bytes long, from the array pointed to by ptr to the stream pointed to by stream.

Returns Number of elements actually written. This number will be less than nmemb if a write error occurs. In C99, returns zero if either nmemb or size is zero. 22.6

fwscanf *Wide-Character Formatted File Read (C99)* <wchar.h>

```
int fwscanf(FILE * restrict stream,
            const wchar_t * restrict format, ...);
```

Wide-character version of fscanf. 25.5

getc *Read Character from File* <stdio.h>

```
int getc(FILE *stream);
```

Reads a character from the stream pointed to by stream. *Note:* getc is normally implemented as a macro; it may evaluate stream more than once.

Returns Character read from the stream. If getc encounters the end of the stream, it sets the stream's end-of-file indicator and returns EOF. If a read error occurs, getc sets the stream's error indicator and returns EOF. 22.4

getchar *Read Character* <stdio.h>

```
int getchar(void);
```

Reads a character from the stdin stream. *Note:* getchar is normally implemented as a macro.

Returns Character read from the stream. If getchar encounters the end of the stream, it sets the stream's end-of-file indicator and returns EOF. If a read error occurs, getchar sets the stream's error indicator and returns EOF. 7.3, 22.4

getenv *Get Environment String* <stdlib.h>

```
char *getenv(const char *name);
```

Searches the operating system's environment list to see if any string matches the one pointed to by name.

Returns A pointer to the string associated with the matching name. Returns a null pointer if no match is found. 26.2

gets *Read String* <stdio.h>

`char *gets(char *s);`

Reads characters from the `stdin` stream and stores them in the array pointed to by `s`. Reading stops at the first new-line character (which is discarded) or at end-of-file. `gets` appends a null character to the string.

Returns `s` (a pointer to the array in which the input is stored). Returns a null pointer if a read error occurs or `gets` encounters the end of the stream before it has stored any characters. 13.3, 22.5

getwc *Read Wide Character from File (C99)* <wchar.h>

`wint_t getwc(FILE *stream);`

Wide-character version of `getc`.

25.5

getwchar *Read Wide Character (C99)* <wchar.h>

`wint_t getwchar(void);`

Wide-character version of `getchar`.

25.5

gmtime *Convert Calendar Time to Broken-Down UTC Time* <time.h>

`struct tm *gmtime(const time_t *timer);`

Returns A pointer to a structure containing a broken-down UTC time equivalent to the calendar time pointed to by `timer`. Returns a null pointer if the calendar time can't be converted to UTC. 26.3

hypot *Hypotenuse (C99)* <math.h>

`double hypot(double x, double y);`

hypotf `float hypotf(float x, float y);`

hypotl `long double hypotl(long double x, long double y);`

Returns $\sqrt{x^2 + y^2}$ (the hypotenuse of a right triangle with legs `x` and `y`). A range error may occur. 23.4

ilogb *Unbiased Exponent (C99)* <math.h>

`int ilogb(double x);`

ilogbf `int ilogbf(float x);`

ilogbl `int ilogbl(long double x);`

Returns Exponent of `x` as a signed integer; equivalent to calling the corresponding `logb` function and casting the returned value to type `int`. Returns `FP_ILOGB0` if `x` is zero, `INT_MAX` if `x` is infinite, and `FP_ILOGBNAN` if `x` is a NaN; a domain error or range error may occur in these cases. 23.4

imaxabs *Greatest-Width Integer Absolute Value (C99)* <inttypes.h>

`intmax_t imaxabs(intmax_t j);`

Returns Absolute value of *j*. The behavior is undefined if the absolute value of *j* can't be represented. 27.2

imaxdiv *Greatest-Width Integer Division (C99)* <inttypes.h>

imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);

Returns A structure of type *imaxdiv_t* containing members named *quot* (the quotient when *numer* is divided by *denom*) and *rem* (the remainder). The behavior is undefined if either part of the result can't be represented. 27.2

isalnum *Test for Alphanumeric* <ctype.h>

int isalnum(int c);

Returns A nonzero value if *c* is alphanumeric and zero otherwise. (*c* is alphanumeric if either *isalpha(c)* or *isdigit(c)* is true.) 23.5

isalpha *Test for Alphabetic* <ctype.h>

int isalpha(int c);

Returns A nonzero value if *c* is alphabetic and zero otherwise. In the "C" locale, *c* is alphabetic if either *islower(c)* or *isupper(c)* is true. 23.5

isblank *Test for Blank (C99)* <ctype.h>

int isblank(int c);

Returns A nonzero value if *c* is a blank character that is used to separate words within a line of text. In the "C" locale, the blank characters are space (' ') and horizontal tab ('\t'). 23.5

iscntrl *Test for Control Character* <ctype.h>

int iscntrl(int c);

Returns A nonzero value if *c* is a control character and zero otherwise. 23.5

isdigit *Test for Digit* <ctype.h>

int isdigit(int c);

Returns A nonzero value if *c* is a decimal digit and zero otherwise. 23.5

isfinite *Test for Finite Number (C99)* <math.h>

int isfinite(real-floating x); *macro*

Returns A nonzero value if *x* is finite (zero, subnormal, or normal, but not infinite or NaN) and zero otherwise. 23.4

isgraph *Test for Graphical Character* <ctype.h>

int isgraph(int c);

Returns A nonzero value if *c* is a printing character (except a space) and zero otherwise. 23.5

isgreater *Test for Greater Than (C99)* <math.h>

int isgreater(real-floating x, real-floating y); *macro*

	<i>Returns</i>	$(x) > (y)$. Unlike the $>$ operator, <code>isgreater</code> doesn't raise the <i>invalid</i> floating-point exception if one or both of the arguments is a NaN.	23.4
<code>isgreaterequal</code>		<i>Test for Greater Than or Equal (C99)</i>	<code><math.h></code>
		<code>int isgreaterequal(real-floating x, real-floating y);</code>	<i>macro</i>
	<i>Returns</i>	$(x) \geq (y)$. Unlike the \geq operator, <code>isgreaterequal</code> doesn't raise the <i>invalid</i> floating-point exception if one or both of the arguments is a NaN.	23.4
<code>isinf</code>		<i>Test for Infinity (C99)</i>	<code><math.h></code>
		<code>int isinf(real-floating x);</code>	<i>macro</i>
	<i>Returns</i>	A nonzero value if x is infinity (positive or negative) and zero otherwise.	23.4
<code>isless</code>		<i>Test for Less Than (C99)</i>	<code><math.h></code>
		<code>int isless(real-floating x, real-floating y);</code>	<i>macro</i>
	<i>Returns</i>	$(x) < (y)$. Unlike the $<$ operator, <code>isless</code> doesn't raise the <i>invalid</i> floating-point exception if one or both of the arguments is a NaN.	23.4
<code>islessequal</code>		<i>Test for Less Than or Equal (C99)</i>	<code><math.h></code>
		<code>int islessequal(real-floating x, real-floating y);</code>	<i>macro</i>
	<i>Returns</i>	$(x) \leq (y)$. Unlike the \leq operator, <code>islessequal</code> doesn't raise the <i>invalid</i> floating-point exception if one or both of the arguments is a NaN.	23.4
<code>islessgreater</code>		<i>Test for Less Than or Greater Than (C99)</i>	<code><math.h></code>
		<code>int islessgreater(real-floating x, real-floating y);</code>	<i>macro</i>
	<i>Returns</i>	$(x) < (y) \ \ (x) > (y)$. Unlike this expression, <code>islessgreater</code> doesn't raise the <i>invalid</i> floating-point exception if one or both of the arguments is a NaN; also, x and y are evaluated only once.	23.4
<code>islower</code>		<i>Test for Lower-Case Letter</i>	<code><ctype.h></code>
		<code>int islower(int c);</code>	
	<i>Returns</i>	A nonzero value if c is a lower-case letter and zero otherwise.	23.5
<code>isnan</code>		<i>Test for NaN (C99)</i>	<code><math.h></code>
		<code>int isnan(real-floating x);</code>	<i>macro</i>
	<i>Returns</i>	A nonzero value if x is a NaN value and zero otherwise.	23.4
<code>isnormal</code>		<i>Test for Normal Number (C99)</i>	<code><math.h></code>
		<code>int isnormal(real-floating x);</code>	<i>macro</i>
	<i>Returns</i>	A nonzero value if x has a normal value (not zero, subnormal, infinite, or NaN) and zero otherwise.	23.4
<code>isprint</code>		<i>Test for Printing Character</i>	<code><ctype.h></code>
		<code>int isprint(int c);</code>	

Returns A nonzero value if *c* is a printing character (including a space) and zero otherwise. 23.5

ispunct *Test for Punctuation Character* <ctype.h>

`int ispunct(int c);`

Returns A nonzero value if *c* is a punctuation character and zero otherwise. All printing characters except the space (' ') and the alphanumeric characters are considered punctuation. *C99 change:* In the "C" locale, all printing characters except those for which `isspace` or `isalnum` is true are considered punctuation. 23.5

isspace *Test for White-Space Character* <ctype.h>

`int isspace(int c);`

Returns A nonzero value if *c* is a white-space character and zero otherwise. In the "C" locale, the white-space characters are space (' '), form feed ('\f'), new-line ('\n'), carriage return ('\r'), horizontal tab ('\t'), and vertical tab ('\v'). 23.5

isunordered *Test for Unordered (C99)* <math.h>

`int isunordered(real-floating x, real-floating y);` *macro*

Returns 1 if *x* and *y* are unordered (at least one is a NaN) and 0 otherwise. 23.4

isupper *Test for Upper-Case Letter* <ctype.h>

`int isupper(int c);`

Returns A nonzero value if *c* is an upper-case letter and zero otherwise. 23.5

iswalnum *Test for Alphanumeric Wide Character (C99)* <wctype.h>

`int iswalnum(wint_t wc);`

Returns A nonzero value if *wc* is alphanumeric and zero otherwise. (*wc* is alphanumeric if either `iswalpha(wc)` or `iswdigit(wc)` is true.) 25.6

iswalpha *Test for Alphabetic Wide Character (C99)* <wctype.h>

`int iswalpha(wint_t wc);`

Returns A nonzero value if *wc* is alphabetic and zero otherwise. (*wc* is alphabetic if `iswupper(wc)` or `iswlower(wc)` is true, or if *wc* is one of a locale-specific set of alphabetic wide characters for which none of `iswcntrl`, `iswdigit`, `iswpunct`, or `iswspace` is true.) 25.6

iswblank *Test for Blank Wide Character (C99)* <wctype.h>

`int iswblank(wint_t wc);`

Returns A nonzero value if *wc* is a standard blank wide character or one of a locale-specific set of wide characters for which `iswspace` is true and that are used to separate words within a line of text. In the "C" locale, `iswblank` returns true only for the standard blank characters: space (L' ') and horizontal tab (L' '\t'). 25.6

<i>iswcntrl</i>	<i>Test for Control Wide Character (C99)</i>	<code><wctype.h></code>
	<code>int iswcntrl(wint_t wc);</code>	
<i>Returns</i>	A nonzero value if <i>wc</i> is a control wide character and zero otherwise.	25.6

<i>iswctype</i>	<i>Test Type of Wide Character (C99)</i>	<code><wctype.h></code>
	<code>int iswctype(wint_t wc, wctype_t desc);</code>	
<i>Returns</i>	A nonzero value if the wide character <i>wc</i> has the property described by <i>desc</i> . (<i>desc</i> must be a value returned by a call of <i>wctype</i> ; the current setting of the <code>LC_CTYPE</code> category must be the same during both calls.) Returns zero otherwise.	25.6

<i>iswdigit</i>	<i>Test for Digit Wide Character (C99)</i>	<code><wctype.h></code>
	<code>int iswdigit(wint_t wc);</code>	
<i>Returns</i>	A nonzero value if <i>wc</i> corresponds to a decimal digit and zero otherwise.	25.6

<i>iswgraph</i>	<i>Test for Graphical Wide Character (C99)</i>	<code><wctype.h></code>
	<code>int iswgraph(wint_t wc);</code>	
<i>Returns</i>	A nonzero value if <code>iswprint(wc)</code> is true and <code>iswspace(wc)</code> is false. Returns zero otherwise.	25.6

<i>iswlower</i>	<i>Test for Lower-Case Wide Character (C99)</i>	<code><wctype.h></code>
	<code>int iswlower(wint_t wc);</code>	
<i>Returns</i>	A nonzero value if <i>wc</i> corresponds to a lower-case letter or is one of a locale-specific set of wide characters for which none of <code>iswcntrl</code> , <code>iswdigit</code> , <code>iswpunct</code> , or <code>iswspace</code> is true. Returns zero otherwise.	25.6

<i>iswprint</i>	<i>Test for Printing Wide Character (C99)</i>	<code><wctype.h></code>
	<code>int iswprint(wint_t wc);</code>	
<i>Returns</i>	A nonzero value if <i>wc</i> is a printing wide character and zero otherwise.	25.6

<i>iswpunct</i>	<i>Test for Punctuation Wide Character (C99)</i>	<code><wctype.h></code>
	<code>int iswpunct(wint_t wc);</code>	
<i>Returns</i>	A nonzero value if <i>wc</i> is a printing wide character that is one of a locale-specific set of punctuation wide characters for which neither <code>iswspace</code> nor <code>iswalnum</code> is true. Returns zero otherwise.	25.6

<i>iswspace</i>	<i>Test for White-Space Wide Character (C99)</i>	<code><wctype.h></code>
	<code>int iswspace(wint_t wc);</code>	
<i>Returns</i>	A nonzero value if <i>wc</i> is one of a locale-specific set of white-space wide characters for which none of <code>iswalnum</code> , <code>iswgraph</code> , or <code>iswpunct</code> is true. Returns zero otherwise.	25.6

<i>iswupper</i>	<i>Test for Upper-Case Wide Character (C99)</i>	<i><wctype.h></i>
	<code>int iswupper(wint_t wc);</code>	
<i>Returns</i>	A nonzero value if <i>wc</i> corresponds to an upper-case letter or is one of a locale-specific set of wide characters for which none of <i>iswcntrl</i> , <i>iswdigit</i> , <i>iswpunct</i> , or <i>iswspace</i> is true. Returns zero otherwise. 25.6	

<i>iswxdigit</i>	<i>Test for Hexadecimal-Digit Wide Character (C99)</i>	<i><wctype.h></i>
	<code>int iswxdigit(wint_t wc);</code>	
<i>Returns</i>	A nonzero value if <i>wc</i> corresponds to a hexadecimal digit (0–9, a–f, A–F) and zero otherwise. 25.6	

<i>isxdigit</i>	<i>Test for Hexadecimal Digit</i>	<i><ctype.h></i>
	<code>int isxdigit(int c);</code>	
<i>Returns</i>	A nonzero value if <i>c</i> is a hexadecimal digit (0–9, a–f, A–F) and zero otherwise. 23.5	

<i>labs</i>	<i>Long Integer Absolute Value</i>	<i><stdlib.h></i>
	<code>long int labs(long int j);</code>	
<i>Returns</i>	Absolute value of <i>j</i> . The behavior is undefined if the absolute value of <i>j</i> can't be represented. 26.2	

<i>ldexp</i>	<i>Combine Fraction and Exponent</i>	<i><math.h></i>
	<code>double ldexp(double x, int exp);</code>	
<i>ldexpf</i>	<code>float ldexpf(float x, int exp);</code>	
<i>ldexpl</i>	<code>long double ldexpl(long double x, int exp);</code>	
<i>Returns</i>	$x \times 2^{\text{exp}}$. A range error may occur. 23.3	

<i>ldiv</i>	<i>Long Integer Division</i>	<i><stdlib.h></i>
	<code>ldiv_t ldiv(long int numer, long int denom);</code>	
<i>Returns</i>	An <i>ldiv_t</i> structure containing members named <i>quot</i> (the quotient when <i>numer</i> is divided by <i>denom</i>) and <i>rem</i> (the remainder). The behavior is undefined if either part of the result can't be represented. 26.2	

<i>lgamma</i>	<i>Logarithm of Gamma Function (C99)</i>	<i><math.h></i>
	<code>double lgamma(double x);</code>	
<i>lgammaf</i>	<code>float lgammaf(float x);</code>	
<i>lgammal</i>	<code>long double lgammal(long double x);</code>	
<i>Returns</i>	$\ln(\Gamma(x))$, where Γ is the gamma function. A range error occurs if <i>x</i> is too large and may occur if <i>x</i> is a negative integer or zero. 23.4	

<i>llabs</i>	<i>Long Long Integer Absolute Value (C99)</i>	<i><stdlib.h></i>
	<code>long long int llabs(long long int j);</code>	

	<i>Returns</i>	Absolute value of <i>j</i> . The behavior is undefined if the absolute value of <i>j</i> can't be represented.	26.2
lldiv	<i>Long Long Integer Division (C99)</i>	<stdlib.h>	
		<code>lldiv_t lldiv(long long int numer, long long int denom);</code>	
	<i>Returns</i>	An <code>lldiv_t</code> structure containing members named <code>quot</code> (the quotient when <code>numer</code> is divided by <code>denom</code>) and <code>rem</code> (the remainder). The behavior is undefined if either part of the result can't be represented.	26.2
llrint	<i>Round to Long Long Integer Using Current Direction (C99)</i>	<math.h>	
		<code>long long int llrint(double x);</code>	
llrintf		<code>long long int llrintf(float x);</code>	
llrintl		<code>long long int llrintl(long double x);</code>	
	<i>Returns</i>	<i>x</i> rounded to the nearest integer using the current rounding direction. If the rounded value is outside the range of the <code>long long int</code> type, the result is unspecified and a domain or range error may occur.	23.4
llround	<i>Round to Nearest Long Long Integer (C99)</i>	<math.h>	
		<code>long long int llround(double x);</code>	
llroundf		<code>long long int llroundf(float x);</code>	
llroundl		<code>long long int llroundl(long double x);</code>	
	<i>Returns</i>	<i>x</i> rounded to the nearest integer, with halfway cases rounded away from zero. If the rounded value is outside the range of the <code>long long int</code> type, the result is unspecified and a domain or range error may occur.	23.4
localeconv	<i>Get Locale Conventions</i>	<locale.h>	
		<code>struct lconv *localeconv(void);</code>	
	<i>Returns</i>	A pointer to a structure containing information about the current locale.	25.1
localtime	<i>Convert Calendar Time to Broken-Down Local Time</i>	<time.h>	
		<code>struct tm *localtime(const time_t *timer);</code>	
	<i>Returns</i>	A pointer to a structure containing a broken-down local time equivalent to the calendar time pointed to by <code>timer</code> . Returns a null pointer if the calendar time can't be converted to local time.	26.3
log	<i>Natural Logarithm</i>	<math.h>	
		<code>double log(double x);</code>	
logf		<code>float logf(float x);</code>	
logl		<code>long double logl(long double x);</code>	
	<i>Returns</i>	Logarithm of <i>x</i> to the base <i>e</i> . A domain error occurs if <i>x</i> is negative. A range error may occur if <i>x</i> is zero.	23.3

log10	<i>Common Logarithm</i>	<code><math.h></code>
	<code>double log10(double x);</code>	
	<code>float log10f(float x);</code>	
	<code>long double log10l(long double x);</code>	
<i>Returns</i>	Logarithm of x to the base 10. A domain error occurs if x is negative. A range error may occur if x is zero. 23.3	

log1p	<i>Natural Logarithm of 1 Plus Argument (C99)</i>	<code><math.h></code>
	<code>double log1p(double x);</code>	
	<code>float log1pf(float x);</code>	
	<code>long double log1pl(long double x);</code>	
<i>Returns</i>	Logarithm of $1 + x$ to the base e . A domain error occurs if x is less than -1 . A range error may occur if x is equal to -1 . 23.4	

log2	<i>Base-2 Logarithm (C99)</i>	<code><math.h></code>
	<code>double log2(double x);</code>	
	<code>float log2f(float x);</code>	
	<code>long double log2l(long double x);</code>	
<i>Returns</i>	Logarithm of x to the base 2. A domain error occurs if x is negative. A range error may occur if x is zero. 23.4	

logb	<i>Radix-Independent Exponent (C99)</i>	<code><math.h></code>
	<code>double logb(double x);</code>	
	<code>float logbf(float x);</code>	
	<code>long double logbl(long double x);</code>	
<i>Returns</i>	$\log_r(x)$, where r is the radix of floating-point arithmetic (defined by the macro <code>FLT_RADIX</code> , which typically has the value 2). A domain error or range error may occur if x is zero. 23.4	

longjmp	<i>Nonlocal Jump</i>	<code><setjmp.h></code>
	<code>void longjmp(jmp_buf env, int val);</code>	
	Restores the environment stored in <code>env</code> and returns from the call of <code>setjmp</code> that originally saved <code>env</code> . If <code>val</code> is nonzero, it will be <code>setjmp</code> 's return value; if <code>val</code> is 0, <code>setjmp</code> returns 1. 24.4	

lrint	<i>Round to Long Integer Using Current Direction (C99)</i>	<code><math.h></code>
	<code>long int lrint(double x);</code>	
	<code>long int lrintf(float x);</code>	
	<code>long int lrintl(long double x);</code>	
<i>Returns</i>	x rounded to the nearest integer using the current rounding direction. If the rounded value is outside the range of the <code>long int</code> type, the result is unspecified and a domain or range error may occur. 23.4	

lround	<i>Round to Nearest Long Integer (C99)</i>	<code><math.h></code>
	<code>long int lround(double x);</code>	
lroundf	<code>long int lroundf(float x);</code>	
lroundl	<code>long int lroundl(long double x);</code>	
<i>Returns</i>	<code>x</code> rounded to the nearest integer, with halfway cases rounded away from zero. If the rounded value is outside the range of the <code>long int</code> type, the result is unspecified and a domain or range error may occur. 23.4	

malloc	<i>Allocate Memory Block</i>	<code><stdlib.h></code>
	<code>void *malloc(size_t size);</code>	
	Allocates a block of memory with <code>size</code> bytes. The block is not cleared.	
<i>Returns</i>	A pointer to the beginning of the block. Returns a null pointer if a block of the requested size can't be allocated. 17.2	

mblen	<i>Length of Multibyte Character</i>	<code><stdlib.h></code>
	<code>int mblen(const char *s, size_t n);</code>	
<i>Returns</i>	If <code>s</code> is a null pointer, returns a nonzero or zero value, depending on whether or not multibyte characters have state-dependent encodings. If <code>s</code> points to a null character, returns zero. Otherwise, returns the number of bytes in the multibyte character pointed to by <code>s</code> ; returns <code>-1</code> if the next <code>n</code> or fewer bytes don't form a valid multibyte character. 25.2	

mbrlen	<i>Length of Multibyte Character – Restartable (C99)</i>	<code><wchar.h></code>
	<code>size_t mbrlen(const char * restrict s, size_t n, mbstate_t * restrict ps);</code>	
	Determines the number of bytes in the array pointed to by <code>s</code> that are required to complete a multibyte character. <code>ps</code> should point to an object of type <code>mbstate_t</code> that contains the current conversion state. A call of <code>mbrlen</code> is equivalent to	
	<code>mbrtowc(NULL, s, n, ps)</code>	
	except that if <code>ps</code> is a null pointer, the address of an internal object is used instead.	
<i>Returns</i>	See <code>mbrtowc</code> . 25.5	

mbrtowc	<i>Convert Multibyte Character to Wide Character – Restartable (C99)</i>	<code><wchar.h></code>
	<code>size_t mbrtowc(wchar_t * restrict pwc, const char * restrict s, size_t n, mbstate_t * restrict ps);</code>	
	If <code>s</code> is a null pointer, a call of <code>mbrtowc</code> is equivalent to	
	<code>mbrtowc(NULL, "", 1, ps)</code>	
	Otherwise, <code>mbrtowc</code> examines up to <code>n</code> bytes in the array pointed to by <code>s</code> to see if	

they complete a valid multibyte character. If so, the multibyte character is converted into a wide character. If `pwc` isn't a null pointer, the wide character is stored in the object pointed to by `pwc`. The value of `ps` should be a pointer to an object of type `mbstate_t` that contains the current conversion state. If `ps` is a null pointer, `mbrtowc` uses an internal object to store the conversion state. If the result of the conversion is the null wide character, the `mbstate_t` object used during the call is left in the initial conversion state.

Returns 0 if the conversion produces a null wide character. Returns a number between 1 and `n` if the conversion produces a wide character other than null, where the value returned is the number of bytes used to complete the multibyte character. Returns `(size_t) (-2)` if the `n` bytes pointed to by `s` weren't enough to complete a multibyte character. Returns `(size_t) (-1)` and stores `EILSEQ` in `errno` if an encoding error occurs. 25.5

mbstowcs *Convert Multibyte String to Wide String – Restorable (C99)* <wchar.h>

```
int mbstowcs(const mbstate_t *ps);
```

Returns A nonzero value if `ps` is a null pointer or it points to an `mbstate_t` object that describes an initial conversion state; otherwise, returns zero. 25.5

mbsrtowcs *Convert Multibyte String to Wide String – Restartable (C99)* <wchar.h>

```
size_t mbsrtowcs(wchar_t * restrict dst,
                 const char ** restrict src,
                 size_t len, mbstate_t * restrict ps);
```

Converts a sequence of multibyte characters from the array indirectly pointed to by `src` into a sequence of corresponding wide characters. `ps` should point to an object of type `mbstate_t` that contains the current conversion state. If the argument corresponding to `ps` is a null pointer, `mbsrtowcs` uses an internal object to store the conversion state. If `dst` isn't a null pointer, the converted characters are stored in the array that it points to. Conversion continues up to and including a terminating null character, which is also stored. Conversion stops earlier if a sequence of bytes is encountered that doesn't form a valid multibyte character or—if `dst` isn't a null pointer—when `len` wide characters have been stored in the array. If `dst` isn't a null pointer, the object pointed to by `src` is assigned either a null pointer (if a terminating null character was reached) or the address just past the last multibyte character converted (if any). If the conversion ends at a null character and if `dst` isn't a null pointer, the resulting state is the initial conversion state.

Returns Number of multibyte characters successfully converted, not including any terminating null character. Returns `(size_t) (-1)` and stores `EILSEQ` in `errno` if an invalid multibyte character is encountered. 25.5

mbstowcs *Convert Multibyte String to Wide String* <stdlib.h>

```
size_t mbstowcs(wchar_t * restrict pwcs,
                 const char * restrict s, size_t n);
```


Converts the sequence of multibyte characters pointed to by *s* into a sequence of wide characters, storing at most *n* wide characters in the array pointed to by *pwc*. Conversion ends if a null character is encountered; it is converted into a null wide character.

Returns Number of array elements modified, not including the null wide character, if any. Returns `(size_t) (-1)` if an invalid multibyte character is encountered. 25.2

mbtowc *Convert Multibyte Character to Wide Character* <stdlib.h>

```
int mbtowc(wchar_t * restrict pwc,
           const char * restrict s, size_t n);
```

If *s* isn't a null pointer, converts the multibyte character pointed to by *s* into a wide character; at most *n* bytes will be examined. If the multibyte character is valid and *pwc* isn't a null pointer, stores the value of the wide character in the object pointed to by *pwc*.

Returns If *s* is a null pointer, returns a nonzero or zero value, depending on whether or not multibyte characters have state-dependent encodings. If *s* points to a null character, returns zero. Otherwise, returns the number of bytes in the multibyte character pointed to by *s*; returns `-1` if the next *n* or fewer bytes don't form a valid multibyte character. 25.2

memchr *Search Memory Block for Character* <string.h>

```
void *memchr(const void *s, int c, size_t n);
```

Returns A pointer to the first occurrence of the character *c* among the first *n* characters of the object pointed to by *s*. Returns a null pointer if *c* isn't found. 23.6

memcmp *Compare Memory Blocks* <string.h>

```
int memcmp(const void *s1, const void *s2, size_t n);
```

Returns A negative, zero, or positive integer, depending on whether the first *n* characters of the object pointed to by *s1* are less than, equal to, or greater than the first *n* characters of the object pointed to by *s2*. 23.6

memcpy *Copy Memory Block* <string.h>

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

Copies *n* characters from the object pointed to by *s2* into the object pointed to by *s1*. The behavior is undefined if the objects overlap.

Returns *s1* (a pointer to the destination). 23.6

memmove *Copy Memory Block* <string.h>

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

Copies *n* characters from the object pointed to by *s2* into the object pointed to by *s1*. Will work properly if the objects overlap.

Returns *s1* (a pointer to the destination). 23.6

memset	<i>Initialize Memory Block</i>	<code><string.h></code>
	<code>void *memset(void *s, int c, size_t n);</code>	
	Stores <code>c</code> in each of the first <code>n</code> characters of the object pointed to by <code>s</code> .	
<i>Returns</i>	<code>s</code> (a pointer to the object).	23.6

mkttime	<i>Convert Broken-Down Local Time to Calendar Time</i>	<code><time.h></code>
	<code>time_t mktime(struct tm *timeptr);</code>	
	Converts a broken-down local time (stored in the structure pointed to by <code>timeptr</code>) into a calendar time. The members of the structure aren't required to be within their legal ranges; also, the values of <code>tm_wday</code> (day of the week) and <code>tm_yday</code> (day of the year) are ignored. <code>mktime</code> stores values in <code>tm_wday</code> and <code>tm_yday</code> after adjusting the other members to bring them into their proper ranges.	
<i>Returns</i>	A calendar time corresponding to the structure pointed to by <code>timeptr</code> . Returns (<code>time_t</code>) (-1) if the calendar time can't be represented.	26.3

modf	<i>Split into Integer and Fractional Parts</i>	<code><math.h></code>
	<code>double modf(double value, double *iptr);</code>	
<i>modff</i>	<code>float modff(float value, float *iptr);</code>	
<i>modfl</i>	<code>long double modfl(long double value, long double *iptr);</code>	
	Splits <code>value</code> into integer and fractional parts; stores the integer part in the object pointed to by <code>iptr</code> .	
<i>Returns</i>	Fractional part of <code>value</code> .	23.3

nan	<i>Create NaN (C99)</i>	<code><math.h></code>
	<code>double nan(const char *tagp);</code>	
<i>nanf</i>	<code>float nanf(const char *tagp);</code>	
<i>nanl</i>	<code>long double nanl(const char *tagp);</code>	
<i>Returns</i>	A "quiet" NaN whose binary pattern is determined by the string pointed to by <code>tagp</code> . Returns zero if quiet NaNs aren't supported.	23.4

nearbyint	<i>Round to Integral Value Using Current Direction (C99)</i>	<code><math.h></code>
	<code>double nearbyint(double x);</code>	
<i>nearbyintf</i>	<code>float nearbyintf(float x);</code>	
<i>nearbyintl</i>	<code>long double nearbyintl(long double x);</code>	
<i>Returns</i>	<code>x</code> rounded to an integer (in floating-point format) using the current rounding direction. Doesn't raise the <i>inexact</i> floating-point exception.	23.4

nextafter	<i>Next Number After (C99)</i>	<code><math.h></code>
	<code>double nextafter(double x, double y);</code>	
<i>nextafterf</i>	<code>float nextafterf(float x, float y);</code>	
<i>nextafterl</i>	<code>long double nextafterl(long double x, long double y);</code>	

Returns Next representable value after x in the direction of y . Returns the value just before x if $y < x$ or the value just after x if $x < y$. Returns y if x equals y . A range error may occur if the magnitude of x is the largest representable finite value and the result is infinite or not representable. 23.4

nexttoward *Next Number Toward (C99)* <math.h>

double nexttoward(double x, long double y);

nexttowardf *float nexttowardf(float x, long double y);*

nexttowardl *long double nexttowardl(long double x, long double y);*

Returns Next representable value after x in the direction of y (see *nextafter*). Returns y converted to the function's type if x equals y . 23.4

perror *Print Error Message* <stdio.h>

*void perror(const char *s);*

Writes the following message to the `stderr` stream:

string: error-message

string is the string pointed to by *s* and *error-message* is an implementation-defined message that matches the one returned by the call `strerror(errno)`. 24.2

pow *Power* <math.h>

double pow(double x, double y);

powf *float powf(float x, float y);*

powl *long double powl(long double x, long double y);*

Returns x raised to the power y . A domain or range error may occur in certain cases, which vary between C89 and C99. 23.3

printf *Formatted Write* <stdio.h>

*int printf(const char * restrict format, ...);*

Writes output to the `stdout` stream. The string pointed to by *format* specifies how subsequent arguments will be displayed.

Returns Number of characters written. Returns a negative value if an error occurs. 3.1, 22.3

putc *Write Character to File* <stdio.h>

*int putc(int c, FILE *stream);*

Writes the character c to the stream pointed to by *stream*. *Note:* `putc` is normally implemented as a macro; it may evaluate *stream* more than once.

Returns c (the character written). If a write error occurs, `putc` sets the stream's error indicator and returns EOF. 22.4

putchar *Write Character* <stdio.h>

int putchar(int c);

Writes the character c to the `stdout` stream. *Note:* `putchar` is normally implemented as a macro.

Returns *c* (the character written). If a write error occurs, `putchar` sets the stream's error indicator and returns EOF. 7.3, 22.4

puts *Write String* <stdio.h>

```
int puts(const char *s);
```

Writes the string pointed to by *s* to the `stdout` stream, then writes a new-line character.

Returns A nonnegative value if successful. Returns EOF if a write error occurs. 13.3, 22.5

putwc *Write Wide Character to File (C99)* <wchar.h>

```
wint_t putwc(wchar_t c, FILE *stream);
```

Wide-character version of `putc`.

25.5

putwchar *Write Wide Character (C99)* <wchar.h>

```
wint_t putwchar(wchar_t c);
```

Wide-character version of `putc`.

25.5

qsort *Sort Array* <stdlib.h>

```
void qsort(void *base, size_t nmem, size_t size,
           int (*compar)(const void *, const void *));
```

Sorts the array pointed to by *base*. The array has *nmem* elements, each *size* bytes long. *compar* is a pointer to a comparison function. When passed pointers to two array elements, the comparison function must return a negative, zero, or positive integer, depending on whether the first array element is less than, equal to, or greater than the second. 17.7, 26.2

raise *Raise Signal* <signal.h>

```
int raise(int sig);
```

Raises the signal whose number is *sig*.

Returns Zero if successful, nonzero otherwise. 24.3

rand *Generate Pseudo-Random Number* <stdlib.h>

```
int rand(void);
```

Returns A pseudo-random integer between 0 and `RAND_MAX` (inclusive). 26.2

realloc *Resize Memory Block* <stdlib.h>

```
void *realloc(void *ptr, size_t size);
```

ptr is assumed to point to a block of memory previously obtained from `calloc`, `malloc`, or `realloc`. `realloc` allocates a block of *size* bytes, copying the contents of the old block if necessary.

Returns A pointer to the beginning of the new memory block. Returns a null pointer if a block of the requested size can't be allocated. 17.3

remainder	<i>Remainder (C99)</i>	<code><math.h></code>
	<code>double remainder(double x, double y);</code>	
remainderf	<code>float remainderf(float x, float y);</code>	
remainderl	<code>long double remainderl(long double x, long double y);</code>	
<i>Returns</i>	$x - ny$, where n is the integer nearest the exact value of x/y . (If x/y is halfway between two integers, n is even.) If $x - ny = 0$, the return value has the same sign as x . If y is zero, either a domain error occurs or zero is returned. 23.4	

remove	<i>Remove File</i>	<code><stdio.h></code>
	<code>int remove(const char *filename);</code>	
	Deletes the file whose name is pointed to by <code>filename</code> .	
<i>Returns</i>	Zero if successful, nonzero otherwise. 22.2	

remquo	<i>Remainder and Quotient (C99)</i>	<code><math.h></code>
	<code>double remquo(double x, double y, int *quo);</code>	
remquof	<code>float remquof(float x, float y, int *quo);</code>	
remquol	<code>long double remquol(long double x, long double y, int *quo);</code>	
	Computes both the remainder and the quotient when x is divided by y . The object pointed to by <code>quo</code> is modified so that it contains n low-order bits of the integer quotient $ x/y $, where n is implementation-defined but must be at least three. The value stored in this object will be negative if $x/y < 0$.	
<i>Returns</i>	Same value as the corresponding remainder function. If y is zero, either a domain error occurs or zero is returned. 23.4	

rename	<i>Rename File</i>	<code><stdio.h></code>
	<code>int rename(const char *old, const char *new);</code>	
	Changes the name of a file. <code>old</code> and <code>new</code> point to strings containing the old name and new name, respectively.	
<i>Returns</i>	Zero if the renaming is successful. Returns a nonzero value if the operation fails (perhaps because the old file is currently open). 22.2	

rewind	<i>Rewind File</i>	<code><stdio.h></code>
	<code>void rewind(FILE *stream);</code>	
	Sets the file position indicator for the stream pointed to by <code>stream</code> to the beginning of the file. Clears the error and end-of-file indicators for the stream. 22.7	

rint	<i>Round to Integral Value Using Current Direction (C99)</i>	<code><math.h></code>
	<code>double rint(double x);</code>	
rintf	<code>float rintf(float x);</code>	
rintl	<code>long double rintl(long double x);</code>	
<i>Returns</i>	x rounded to an integer (in floating-point format) using the current rounding direc-	

tion. May raise the *inexact* floating-point exception if the result has a different value than *x*. 23.4

round *Round to Nearest Integral Value (C99)* <math.h>

`double round(double x);`

roundf `float roundf(float x);`

roundl `long double roundl(long double x);`

Returns *x* rounded to the nearest integer (in floating-point format). Halfway cases are rounded away from zero. 23.4

scalbln *Scale Floating-Point Number Using Long Integer (C99)* <math.h>

`double scalbln(double x, long int n);`

scalblnf `float scalblnf(float x, long int n);`

scalblnl `long double scalblnl(long double x, long int n);`

Returns $x \times \text{FLT_RADIX}^n$, computed in an efficient way. A range error may occur. 23.4

scalbn *Scale Floating-Point Number Using Integer (C99)* <math.h>

`double scalbn(double x, int n);`

scalbnf `float scalbnf(float x, int n);`

scalbnl `long double scalbnl(long double x, int n);`

Returns $x \times \text{FLT_RADIX}^n$, computed in an efficient way. A range error may occur. 23.4

scanf *Formatted Read* <stdio.h>

`int scanf(const char * restrict format, ...);`

Reads input items from the `stdin` stream. The string pointed to by `format` specifies the format of the items to be read. The arguments that follow `format` point to objects in which the items are to be stored.

Returns Number of input items successfully read and stored. Returns EOF if an input failure occurs before any items can be read. 3.2, 22.3

setbuf *Set Buffer* <stdio.h>

`void setbuf(FILE * restrict stream,
char * restrict buf);`

If `buf` isn't a null pointer, a call of `setbuf` is equivalent to:

`(void) setvbuf(stream, buf, _IOFBF, BUFSIZ);`

Otherwise, it's equivalent to:

`(void) setvbuf(stream, NULL, _IONBF, 0);` 22.2

setjmp *Prepare for Nonlocal Jump* <setjmp.h>

`int setjmp(jmp_buf env);` *macro*

Stores the current environment in `env` for use in a later call of `longjmp`.

Returns Zero when called directly. Returns a nonzero value when returning from a call of `longjmp`. 24.4

setlocale *Set Locale* <locale.h>

```
char *setlocale(int category, const char *locale);
```

Sets a portion of the program's locale. *category* indicates which portion is affected. *locale* points to a string representing the new locale.

Returns If *locale* is a null pointer, returns a pointer to the string associated with *category* for the current locale. Otherwise, returns a pointer to the string associated with *category* for the new locale. Returns a null pointer if the operation fails.

25.1

setvbuf *Set Buffer* <stdio.h>

```
int setvbuf(FILE * restrict stream,
            char * restrict buf,
            int mode, size_t size);
```

Changes the buffering of the stream pointed to by *stream*. The value of *mode* can be either `_IOFBF` (full buffering), `_IOLBF` (line buffering), or `_IONBF` (no buffering). If *buf* is a null pointer, a buffer is automatically allocated if needed. Otherwise, *buf* points to a memory block that can be used as the buffer; *size* is the number of bytes in the block. *Note*: `setvbuf` must be called after the stream is opened but before any other operations are performed on it.

Returns Zero if the operation is successful. Returns a nonzero value if *mode* is invalid or the request can't be honored.

22.2

signal *Install Signal Handler* <signal.h>

```
void (*signal(int sig, void (*func)(int)))(int);
```

Installs the function pointed to by *func* as the handler for the signal whose number is *sig*. Passing `SIG_DFL` as the second argument causes default handling for the signal; passing `SIG_IGN` causes the signal to be ignored.

Returns A pointer to the previous handler for this signal; returns `SIG_ERR` and stores a positive value in `errno` if the handler can't be installed.

24.3

signbit *Sign Bit (C99)* <math.h>

```
int signbit(real-floating x);
```

macro

Returns A nonzero value if the sign of *x* is negative and zero otherwise. The value of *x* may be any number, including infinity and NaN.

23.4

sin *Sine* <math.h>

```
double sin(double x);
```

```
float sinf(float x);
```

```
long double sinl(long double x);
```

Returns Sine of *x* (measured in radians).

23.3

sinh *Hyperbolic Sine* <math.h>

```
double sinh(double x);
```


sinhf *float sinhf(float x);*
sinhl *long double sinhl(long double x);*
Returns Hyperbolic sine of *x*. A range error occurs if the magnitude of *x* is too large. 23.3

snprintf *Bounded Formatted String Write (C99)* <stdio.h>
*int snprintf(char * restrict s, size_t n,*
*const char * restrict format, ...);*
 Equivalent to *fprintf*, but stores characters in the array pointed to by *s* instead of writing them to a stream. No more than *n* - 1 characters will be written to the array. The string pointed to by *format* specifies how subsequent arguments will be displayed. Stores a null character in the array at the end of output.
Returns Number of characters that would have been stored in the array (not including the null character) had there been no length restriction. Returns a negative value if an encoding error occurs. 22.8

sprintf *Formatted String Write* <stdio.h>
*int sprintf(char * restrict s,*
*const char * restrict format, ...);*
 Equivalent to *fprintf*, but stores characters in the array pointed to by *s* instead of writing them to a stream. The string pointed to by *format* specifies how subsequent arguments will be displayed. Stores a null character in the array at the end of output.
Returns Number of characters stored in the array, not including the null character. In C99, returns a negative value if an encoding error occurs. 22.8

sqrt *Square Root* <math.h>
double sqrt(double x);
sqrtf *float sqrtf(float x);*
sqrtl *long double sqrtl(long double x);*
Returns Nonnegative square root of *x*. A domain error occurs if *x* is negative. 23.3

srand *Seed Pseudo-Random Number Generator* <stdlib.h>
void srand(unsigned int seed);
 Uses *seed* to initialize the sequence of pseudo-random numbers produced by calling *rand*. 26.2

sscanf *Formatted String Read* <stdio.h>
*int sscanf(const char * restrict s,*
*const char * restrict format, ...);*
 Equivalent to *fscanf*, but reads characters from the string pointed to by *s* instead of reading them from a stream. The string pointed to by *format* specifies the format of the items to be read. The arguments that follow *format* point to objects in which the items are to be stored.

Returns Number of input items successfully read and stored. Returns EOF if an input failure occurs before any items could be read. 22.8

strcat *String Concatenation* <string.h>

```
char *strcat(char * restrict s1,
             const char * restrict s2);
```

Appends characters from the string pointed to by *s2* to the string pointed to by *s1*.

Returns *s1* (a pointer to the concatenated string). 13.5, 23.6

strchr *Search String for Character* <string.h>

```
char *strchr(const char *s, int c);
```

Returns A pointer to the first occurrence of the character *c* in the string pointed to by *s*. Returns a null pointer if *c* isn't found. 23.6

strcmp *String Comparison* <string.h>

```
int strcmp(const char *s1, const char *s2);
```

Returns A negative, zero, or positive integer, depending on whether the string pointed to by *s1* is less than, equal to, or greater than the string pointed to by *s2*. 13.5, 23.6

strcoll *String Comparison Using Locale-Specific Collating Sequence* <string.h>

```
int strcoll(const char *s1, const char *s2);
```

Returns A negative, zero, or positive integer, depending on whether the string pointed to by *s1* is less than, equal to, or greater than the string pointed to by *s2*. The comparison is performed according to the rules of the current locale's LC_COLLATE category. 23.6

strcpy *String Copy* <string.h>

```
char *strcpy(char * restrict s1,
             const char * restrict s2);
```

Copies the string pointed to by *s2* into the array pointed to by *s1*.

Returns *s1* (a pointer to the destination). 13.5, 23.6

strcspn *Search String for Initial Span of Characters Not in Set* <string.h>

```
size_t strcspn(const char *s1, const char *s2);
```

Returns Length of the longest initial segment of the string pointed to by *s1* that doesn't contain any character in the string pointed to by *s2*. 23.6

strerror *Convert Error Number to String* <string.h>

```
char *strerror(int errnum);
```

Returns A pointer to a string containing an error message corresponding to the value of *errnum*. 24.2

strftime	<i>Write Formatted Date and Time to String</i>	<code><time.h></code>
	<pre>size_t strftime(char * restrict s, size_t maxsize, const char * restrict format, const struct tm * restrict timeptr);</pre>	
	<p>Stores characters in the array pointed to by <i>s</i> under control of the string pointed to by <i>format</i>. The format string may contain ordinary characters, which are copied unchanged, and conversion specifiers, which are replaced by values from the structure pointed to by <i>timeptr</i>. The <i>maxsize</i> parameter limits the number of characters (including the null character) that can be stored.</p>	
<i>Returns</i>	<p>Number of characters stored (not including the terminating null character). Returns zero if the number of characters to be stored (including the null character) exceeds <i>maxsize</i>.</p> <p style="text-align: right;">26.3</p>	
<hr/>		
strlen	<i>String Length</i>	<code><string.h></code>
	<pre>size_t strlen(const char *s);</pre>	
<i>Returns</i>	<p>Length of the string pointed to by <i>s</i>, not including the null character.</p> <p style="text-align: right;">13.5, 23.6</p>	
<hr/>		
strncat	<i>Bounded String Concatenation</i>	<code><string.h></code>
	<pre>char *strncat(char * restrict s1, const char * restrict s2, size_t n);</pre>	
	<p>Appends characters from the array pointed to by <i>s2</i> to the string pointed to by <i>s1</i>. Copying stops when a null character is encountered or <i>n</i> characters have been copied.</p>	
<i>Returns</i>	<p><i>s1</i> (a pointer to the concatenated string).</p> <p style="text-align: right;">13.5, 23.6</p>	
<hr/>		
strncmp	<i>Bounded String Comparison</i>	<code><string.h></code>
	<pre>int strncmp(const char *s1, const char *s2, size_t n);</pre>	
<i>Returns</i>	<p>A negative, zero, or positive integer, depending on whether the first <i>n</i> characters of the array pointed to by <i>s1</i> are less than, equal to, or greater than the first <i>n</i> characters of the array pointed to by <i>s2</i>. Comparison stops if a null character is encountered in either array.</p> <p style="text-align: right;">23.6</p>	
<hr/>		
strncpy	<i>Bounded String Copy</i>	<code><string.h></code>
	<pre>char *strncpy(char * restrict s1, const char * restrict s2, size_t n);</pre>	
	<p>Copies the first <i>n</i> characters of the array pointed to by <i>s2</i> into the array pointed to by <i>s1</i>. If it encounters a null character in the array pointed to by <i>s2</i>, <i>strncpy</i> adds null characters to the array pointed to by <i>s1</i> until a total of <i>n</i> characters have been written.</p>	
<i>Returns</i>	<p><i>s1</i> (a pointer to the destination).</p> <p style="text-align: right;">13.5, 23.6</p>	

strpbrk *Search String for One of a Set of Characters* <string.h>

```
char *strpbrk(const char *s1, const char *s2);
```

Returns A pointer to the leftmost character in the string pointed to by *s1* that matches any character in the string pointed to by *s2*. Returns a null pointer if no match is found. 23.6

strrchr *Search String in Reverse for Character* <string.h>

```
char *strrchr(const char *s, int c);
```

Returns A pointer to the last occurrence of the character *c* in the string pointed to by *s*. Returns a null pointer if *c* isn't found. 23.6

strspn *Search String for Initial Span of Characters in Set* <string.h>

```
size_t strspn(const char *s1, const char *s2);
```

Returns Length of the longest initial segment in the string pointed to by *s1* that consists entirely of characters in the string pointed to by *s2*. 23.6

strstr *Search String for Substring* <string.h>

```
char *strstr(const char *s1, const char *s2);
```

Returns A pointer to the first occurrence in the string pointed to by *s1* of the sequence of characters in the string pointed to by *s2*. Returns a null pointer if no match is found. 23.6

strtod *Convert String to Double* <stdlib.h>

```
double strtod(const char * restrict nptr,
              char ** restrict endptr);
```

Skips white-space characters in the string pointed to by *nptr*, then converts subsequent characters into a double value. If *endptr* isn't a null pointer, *strtod* modifies the object pointed to by *endptr* so that it points to the first leftover character. If no double value is found, or if it has the wrong form, *strtod* stores *nptr* in the object pointed to by *endptr*. If the number is too large or small to represent, it stores *ERANGE* in *errno*. *C99 changes*: The string pointed to by *nptr* may contain a hexadecimal floating-point number, infinity, or NaN. Whether *ERANGE* is stored in *errno* when the number is too small to represent is implementation-defined.

Returns The converted number. Returns zero if no conversion could be performed. If the number is too large to represent, returns plus or minus *HUGE_VAL*, depending on the number's sign. Returns zero if the number is too small to represent. *C99 change*: If the number is too small to represent, *strtod* returns a value whose magnitude is no greater than the smallest normalized positive double. 26.2

strtof *Convert String to Float (C99)* <stdlib.h>

```
float strtof(const char * restrict nptr,
             char ** restrict endptr);
```


`strtof` is identical to `strtod`, except that it converts a string to a float value.

Returns The converted number. Returns zero if no conversion could be performed. If the number is too large to represent, returns plus or minus `HUGE_VALF`, depending on the number's sign. If the number is too small to represent, returns a value whose magnitude is no greater than the smallest normalized positive float. 26.2

`strtoimax` *Convert String to Greatest-Width Integer (C99)* <inttypes.h>

```
intmax_t strtoimax(const char * restrict nptr,
                  char ** restrict endptr, int base);
```

`strtoimax` is identical to `strtol`, except that it converts a string to a value of type `intmax_t` (the widest signed integer type).

Returns The converted number. Returns zero if no conversion could be performed. If the number can't be represented, returns `INTMAX_MAX` or `INTMAX_MIN`, depending on the number's sign. 27.2

`strtok` *Search String for Token* <string.h>

```
char *strtok(char * restrict s1,
             const char * restrict s2);
```

Searches the string pointed to by `s1` for a "token" consisting of characters not in the string pointed to by `s2`. If a token exists, the character following it is changed to a null character. If `s1` is a null pointer, a search begun by the most recent call of `strtok` is continued; the search begins immediately after the null character at the end of the previous token.

Returns A pointer to the first character of the token. Returns a null pointer if no token could be found. 23.6

`strtol` *Convert String to Long Integer* <stdlib.h>

```
long int strtol(const char * restrict nptr,
               char ** restrict endptr, int base);
```

Skips white-space characters in the string pointed to by `nptr`, then converts subsequent characters into a `long int` value. If `base` is between 2 and 36, it is used as the radix of the number. If `base` is zero, the number is assumed to be decimal unless it begins with 0 (octal) or with 0x or 0X (hexadecimal). If `endptr` isn't a null pointer, `strtol` modifies the object pointed to by `endptr` so that it points to the first leftover character. If no `long int` value is found, or if it has the wrong form, `strtol` stores `nptr` in the object pointed to by `endptr`. If the number can't be represented, it stores `ERANGE` in `errno`.

Returns The converted number. Returns zero if no conversion could be performed. If the number can't be represented, returns `LONG_MAX` or `LONG_MIN`, depending on the number's sign. 26.2

`strtold` *Convert String to Long Double (C99)* <stdlib.h>

```
long double strtold(const char * restrict nptr,
                   char ** restrict endptr);
```


`strtold` is identical to `strtod`, except that it converts a string to a long double value.

Returns The converted number. Returns zero if no conversion could be performed. If the number is too large to represent, returns plus or minus `HUGE_VALL`, depending on the number's sign. If the number is too small to represent, returns a value whose magnitude is no greater than the smallest normalized positive long double. 26.2

`strtoll` Convert String to Long Long Integer (C99) <stdlib.h>

```
long long int strtoll(const char * restrict nptr,
                    char ** restrict endptr,
                    int base);
```

`strtoll` is identical to `strtol`, except that it converts a string to a long long int value.

Returns The converted number. Returns zero if no conversion could be performed. If the number can't be represented, returns `LLONG_MAX` or `LLONG_MIN`, depending on the number's sign. 26.2

`strtoul` Convert String to Unsigned Long Integer <stdlib.h>

```
unsigned long int strtoul(const char * restrict nptr,
                        char ** restrict endptr,
                        int base);
```

`strtoul` is identical to `strtol`, except that it converts a string to an unsigned long int value.

Returns The converted number. Returns zero if no conversion could be performed. If the number can't be represented, returns `ULONG_MAX`. 26.2

`strtoull` Convert String to Unsigned Long Long Integer (C99) <stdlib.h>

```
unsigned long long int strtoull(
    const char * restrict nptr,
    char ** restrict endptr, int base);
```

`strtoull` is identical to `strtol`, except that it converts a string to an unsigned long long int value.

Returns The converted number. Returns zero if no conversion could be performed. If the number can't be represented, returns `ULLONG_MAX`. 26.2

`strtoumax` Convert String to Unsigned Greatest-Width Integer (C99) <inttypes.h>

```
uintmax_t strtoumax(const char * restrict nptr,
                   char ** restrict endptr,
                   int base);
```

`strtoumax` is identical to `strtol`, except that it converts a string to a value of type `uintmax_t` (the widest unsigned integer type).

Returns The converted number. Returns zero if no conversion could be performed. If the number can't be represented, returns `UINTMAX_MAX`. 27.2

strxfrm *Transform String* <string.h>

```
size_t strxfrm(char * restrict s1,
               const char * restrict s2, size_t n);
```

Transforms the string pointed to by *s2*, placing the first *n* characters of the result—including the null character—in the array pointed to by *s1*. Calling `strcmp` with two transformed strings should produce the same outcome (negative, zero, or positive) as calling `strcmp` with the original strings. If *n* is zero, *s1* is allowed to be a null pointer.

Returns Length of the transformed string. If this value is *n* or more, the contents of the array pointed to by *s1* are indeterminate. 23.6

swprintf *Wide-Character Formatted String Write (C99)* <wchar.h>

```
int swprintf(wchar_t * restrict s, size_t n,
             const wchar_t * restrict format, ...);
```

Equivalent to `fwprintf`, but stores wide characters in the array pointed to by *s* instead of writing them to a stream. The string pointed to by *format* specifies how subsequent arguments will be displayed. No more than *n* wide characters will be written to the array, including a terminating null wide character.

Returns Number of wide characters stored in the array, not including the null wide character. Returns a negative value if an encoding error occurs or the number of wide characters to be written is *n* or more. 25.5

swscanf *Wide-Character Formatted String Read (C99)* <wchar.h>

```
int swscanf(const wchar_t * restrict s,
            const wchar_t * restrict format, ...);
```

Wide-character version of `sscanf`. 25.5

system *Perform Operating-System Command* <stdlib.h>

```
int system(const char *string);
```

Passes the string pointed to by *string* to the operating system's command processor (shell) to be executed. Program termination may occur as a result of executing the command.

Returns If *string* is a null pointer, returns a nonzero value if a command processor is available. If *string* isn't a null pointer, `system` returns an implementation-defined value (if it returns at all). 26.2

tan *Tangent* <math.h>

```
double tan(double x);
```

```
float tanf(float x);
```

```
long double tanl(long double x);
```

Returns Tangent of *x* (measured in radians). 23.3

tanh	<i>Hyperbolic Tangent</i>	<math.h>
	double tanh(double x);	
tanhf	float tanhf(float x);	
tanhl	long double tanhl(long double x);	
<i>Returns</i>	Hyperbolic tangent of x.	23.3
tgamma	<i>Gamma Function (C99)</i>	<math.h>
	double tgamma(double x);	
tgammaf	float tgammaf(float x);	
tgammal	long double tgammal(long double x);	
<i>Returns</i>	$\Gamma(x)$, where Γ is the gamma function. A domain error or range error may occur if x is a negative integer or zero. A range error may occur if the magnitude of x is too large or too small.	23.4
time	<i>Current Time</i>	<time.h>
	time_t time(time_t *timer);	
<i>Returns</i>	Current calendar time. Returns (time_t) (-1) if the calendar time isn't available. If timer isn't a null pointer, also stores the return value in the object pointed to by timer.	26.3
tmpfile	<i>Create Temporary File</i>	<stdio.h>
	FILE *tmpfile(void);	
	Creates a temporary file that will automatically be removed when it's closed or the program ends. Opens the file in "wb+" mode.	
<i>Returns</i>	A file pointer to be used when performing subsequent operations on the file. Returns a null pointer if a temporary file can't be created.	22.2
tmpnam	<i>Generate Temporary File Name</i>	<stdio.h>
	char *tmpnam(char *s);	
	Generates a name for a temporary file. If s is a null pointer, tmpnam stores the file name in a static object. Otherwise, it copies the file name into the character array pointed to by s. (The array must be long enough to store L_tmpnam characters.)	
<i>Returns</i>	A pointer to the file name. Returns a null pointer if a file name can't be generated.	22.2
tolower	<i>Convert to Lower Case</i>	<ctype.h>
	int tolower(int c);	
<i>Returns</i>	If c is an upper-case letter, returns the corresponding lower-case letter. If c isn't an upper-case letter, returns c unchanged.	23.5
toupper	<i>Convert to Upper Case</i>	<ctype.h>
	int toupper(int c);	

Returns If *c* is a lower-case letter, returns the corresponding upper-case letter. If *c* isn't a lower-case letter, returns *c* unchanged. 23.5

towctrans *Transliterate Wide Character (C99)* <wctype.h>

wint_t towctrans(wint_t wc, wctrans_t desc);

Returns Mapped value of *wc* using the mapping described by *desc*. (*desc* must be a value returned by a call of *wctrans*; the current setting of the LC_CTYPE category must be the same during both calls.) 25.6

towlower *Convert Wide Character to Lower Case (C99)* <wctype.h>

wint_t tolower(wint_t wc);

Returns If *iswupper(wc)* is true, returns a corresponding wide character for which *iswlower* is true in the current locale, if such a character exists. Otherwise, returns *wc* unchanged. 25.6

towupper *Convert Wide Character to Upper Case (C99)* <wctype.h>

wint_t towupper(wint_t wc);

Returns If *iswlower(wc)* is true, returns a corresponding wide character for which *iswupper* is true in the current locale, if such a character exists. Otherwise, returns *wc* unchanged. 25.6

trunc *Truncate to Nearest Integral Value (C99)* <math.h>

double trunc(double x);

truncf *float truncf(float x);*

truncl *long double truncl(long double x);*

Returns *x* rounded to the integer (in floating-point format) nearest to it but no larger in magnitude. 23.4

ungetc *Unread Character* <stdio.h>

*int ungetc(int c, FILE *stream);*

Pushes the character *c* back onto the stream pointed to by *stream* and clears the stream's end-of-file indicator. The number of characters that can be pushed back by consecutive calls of *ungetc* varies; only the first call is guaranteed to succeed. Calling a file positioning function (*fseek*, *fsetpos*, or *rewind*) causes the pushed-back character(s) to be lost.

Returns *c* (the pushed-back character). Returns EOF if an attempt is made to push back EOF or to push back too many characters without a read or file positioning operation. 22.4

ungetwc *Unread Wide Character (C99)* <wchar.h>

*wint_t ungetwc(wint_t c, FILE *stream);*

Wide-character version of *ungetc*. 25.5

va_arg	<i>Fetch Argument from Variable Argument List</i>	<stdarg.h>
	<i>type</i> va_arg(va_list ap, <i>type</i>);	<i>macro</i>
	Fetches an argument in the variable argument list associated with ap, then modifies ap so that the next use of va_arg fetches the following argument. ap must have been initialized by va_start (or va_copy in C99) prior to the first use of va_arg.	
<i>Returns</i>	Value of the argument, assuming that its type (after the default argument promotions have been applied) is compatible with <i>type</i> . 26.1	
va_copy	<i>Copy Variable Argument List (C99)</i>	<stdarg.h>
	void va_copy(va_list dest, va_list src);	<i>macro</i>
	Copies src into dest. The value of dest will be the same as if va_start had been applied to dest followed by the same sequence of va_arg applications that was used to reach the present state of src. 26.1	
va_end	<i>End Processing of Variable Argument List</i>	<stdarg.h>
	void va_end(va_list ap);	<i>macro</i>
	Ends the processing of the variable argument list associated with ap. 26.1	
va_start	<i>Start Processing of Variable Argument List</i>	<stdarg.h>
	void va_start(va_list ap, <i>parmN</i>);	<i>macro</i>
	Must be invoked before accessing a variable argument list. Initializes ap for later use by va_arg and va_end. <i>parmN</i> is the name of the last ordinary parameter (the one followed by , ...). 26.1	
vfprintf	<i>Formatted File Write Using Variable Argument List</i>	<stdio.h>
	int vfprintf(FILE * restrict stream, const char * restrict format, va_list arg);	
	Equivalent to fprintf with the variable argument list replaced by arg.	
<i>Returns</i>	Number of characters written. Returns a negative value if an error occurs. 26.1	
vfscanf	<i>Formatted File Read Using Variable Argument List (C99)</i>	<stdio.h>
	int vfscanf(FILE * restrict stream, const char * restrict format, va_list arg);	
	Equivalent to fscanf with the variable argument list replaced by arg.	
<i>Returns</i>	Number of input items successfully read and stored. Returns EOF if an input failure occurs before any items can be read. 26.1	
vwprintf	<i>Wide-Character Formatted File Write Using Variable Argument List (C99)</i>	<wchar.h>


```
int vfwprintf(FILE * restrict stream,
              const wchar_t * restrict format,
              va_list arg);
```

Wide-character version of `fprintf`. 25.5

`vfwscanf` *Wide-Character Formatted File Read Using Variable Argument List (C99)* <wchar.h>

```
int vfwscanf(FILE * restrict stream,
             const wchar_t * restrict format,
             va_list arg);
```

Wide-character version of `vfscanf`. 25.5

`vprintf` *Formatted Write Using Variable Argument List* <stdio.h>

```
int vprintf(const char * restrict format, va_list arg);
```

Equivalent to `printf` with the variable argument list replaced by `arg`.

Returns Number of characters written. Returns a negative value if an error occurs. 26.1

`vscanf` *Formatted Read Using Variable Argument List (C99)* <stdio.h>

```
int vscanf(const char * restrict format, va_list arg);
```

Equivalent to `scanf` with the variable argument list replaced by `arg`.

Returns Number of input items successfully read and stored. Returns EOF if an input failure occurs before any items can be read. 26.1

`vsnprintf` *Bounded Formatted String Write Using Variable Argument List (C99)* <stdio.h>

```
int vsnprintf(char * restrict s, size_t n,
              const char * restrict format,
              va_list arg);
```

Equivalent to `snprintf` with the variable argument list replaced by `arg`.

Returns Number of characters that would have been stored in the array pointed to by `s` (not including the null character) had there been no length restriction. Returns a negative value if an encoding error occurs. 26.1

`vsprintf` *Formatted String Write Using Variable Argument List* <stdio.h>

```
int vsprintf(char * restrict s,
             const char * restrict format,
             va_list arg);
```

Equivalent to `sprintf` with the variable argument list replaced by `arg`.

Returns Number of characters stored in the array pointed to by `s`, not including the null character. In C99, returns a negative value if an encoding error occurs. 26.1

vsscanf	<i>Formatted String Read Using Variable Argument List (C99)</i>	<code><stdio.h></code>
	<pre>int vsscanf(const char * restrict s, const char * restrict format, va_list arg);</pre>	
	Equivalent to <code>scanf</code> with the variable argument list replaced by <code>arg</code> .	
<i>Returns</i>	Number of input items successfully read and stored. Returns EOF if an input failure occurs before any items can be read. 26.1	

vswprintf	<i>Wide-Character Formatted String Write Using Variable Argument List (C99)</i>	<code><wchar.h></code>
	<pre>int vswprintf(wchar_t * restrict s, size_t n, const wchar_t * restrict format, va_list arg);</pre>	
	Equivalent to <code>swprintf</code> with the variable argument list replaced by <code>arg</code> .	
<i>Returns</i>	Number of wide characters stored in the array pointed to by <code>s</code> , not including the null wide character. Returns a negative value if an encoding error occurs or the number of wide characters to be written is <code>n</code> or more. 25.5	

vswscanf	<i>Wide-Character Formatted String Read Using Variable Argument List (C99)</i>	<code><wchar.h></code>
	<pre>int vswscanf(const wchar_t * restrict s, const wchar_t * restrict format, va_list arg);</pre>	
	Wide-character version of <code>vsscanf</code> . 25.5	

vwprintf	<i>Wide-Character Formatted Write Using Variable Argument List (C99)</i>	<code><wchar.h></code>
	<pre>int vwprintf(const wchar_t * restrict format, va_list arg);</pre>	
	Wide-character version of <code>vprintf</code> . 25.5	

vwscanf	<i>Wide-Character Formatted Read Using Variable Argument List (C99)</i>	<code><wchar.h></code>
	<pre>int vwscanf(const wchar_t * restrict format, va_list arg);</pre>	
	Wide-character version of <code>vscanf</code> . 25.5	

wcrtomb	<i>Convert Wide Character to Multibyte Character – Restartable (C99)</i>	<code><wchar.h></code>
	<pre>size_t wcrtomb(char * restrict s, wchar_t wc, mbstate_t * restrict ps);</pre>	
	If <code>s</code> is a null pointer, a call of <code>wcrtomb</code> is equivalent to <code>wcrtomb(buf, L'\0', ps)</code>	

where `buf` is an internal buffer. Otherwise, `wcrtomb` converts `wc` from a wide character into a multibyte character (possibly including shift sequences), which it stores in the array pointed to by `s`. The value of `ps` should be a pointer to an object of type `mbstate_t` that contains the current conversion state. If `ps` is a null pointer, `wcrtomb` uses an internal object to store the conversion state. If `wc` is a null wide character, `wcrtomb` stores a null byte, preceded by a shift sequence if necessary to restore the initial shift state, and the `mbstate_t` object used during the call is left in the initial conversion state.

Returns Number of bytes stored in the array, including shift sequences. If `wc` isn't a valid wide character, returns `(size_t) (-1)` and stores `EILSEQ` in `errno`. 25.5

wcscat *Wide-String Concatenation (C99)* <wchar.h>

```
wchar_t *wcscat(wchar_t * restrict s1,
                const wchar_t * restrict s2);
```

Wide-character version of `strcat`. 25.5

wcschr *Search Wide String for Character (C99)* <wchar.h>

```
wchar_t *wcschr(const wchar_t *s, wchar_t c);
```

Wide-character version of `strchr`. 25.5

wcscmp *Wide-String Comparison (C99)* <wchar.h>

```
int wcscmp(const wchar_t *s1, const wchar_t *s2);
```

Wide-character version of `strcmp`. 25.5

wscoll *Wide-String Comparison Using Locale-Specific Collating Sequence (C99)* <wchar.h>

```
int wscoll(const wchar_t *s1, const wchar_t *s2);
```

Wide-character version of `strcoll`. 25.5

wcscpy *Wide-String Copy (C99)* <wchar.h>

```
wchar_t *wcscpy(wchar_t * restrict s1,
                const wchar_t * restrict s2);
```

Wide-character version of `strcpy`. 25.5

wcscspn *Search Wide String for Initial Span of Characters Not in Set (C99)* <wchar.h>

```
size_t wcscspn(const wchar_t *s1, const wchar_t *s2);
```

Wide-character version of `strcspn`. 25.5

wcsftime *Write Formatted Date and Time to Wide String (C99)* <wchar.h>

```
size_t wcsftime(wchar_t * restrict s, size_t maxsize,
                const wchar_t * restrict format,
                const struct tm * restrict timeptr);
```

Wide-character version of `strftime`. 25.5

wcslen	<i>Wide-String Length (C99)</i>	<wchar.h>
	<code>size_t wcslen(const wchar_t *s);</code>	
	Wide-character version of <code>strlen</code> .	25.5
wcsncat	<i>Bounded Wide-String Concatenation (C99)</i>	<wchar.h>
	<code>wchar_t *wcsncat(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);</code>	
	Wide-character version of <code>strncat</code> .	25.5
wcsncmp	<i>Bounded Wide-String Comparison (C99)</i>	<wchar.h>
	<code>int wcsncmp(const wchar_t *s1, const wchar_t *s2, size_t n);</code>	
	Wide-character version of <code>strncmp</code> .	25.5
wcsncpy	<i>Bounded Wide-String Copy (C99)</i>	<wchar.h>
	<code>wchar_t *wcsncpy(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);</code>	
	Wide-character version of <code>strncpy</code> .	25.5
wcspbrk	<i>Search Wide String for One of a Set of Characters (C99)</i>	<wchar.h>
	<code>wchar_t *wcspbrk(const wchar_t *s1, const wchar_t *s2);</code>	
	Wide-character version of <code>strpbrk</code> .	25.5
wcsrchr	<i>Search Wide String in Reverse for Character (C99)</i>	<wchar.h>
	<code>wchar_t *wcsrchr(const wchar_t *s, wchar_t c);</code>	
	Wide-character version of <code>strrchr</code> .	25.5
wcsrtombs	<i>Convert Wide String to Multibyte String – Restartable (C99)</i>	<wchar.h>
	<code>size_t wcsrtombs(char * restrict dst, const wchar_t ** restrict src, size_t len, mbstate_t * restrict ps);</code>	
	Converts a sequence of wide characters from the array indirectly pointed to by <code>src</code> into a sequence of corresponding multibyte characters that begins in the conversion state described by the object pointed to by <code>ps</code> . If <code>ps</code> is a null pointer, <code>wcsrtombs</code> uses an internal object to store the conversion state. If <code>dst</code> isn't a null pointer, the converted characters are then stored in the array pointed to by <code>dst</code> . Conversion continues up to and including a terminating null wide character, which is also stored. Conversion stops earlier if a wide character is reached that doesn't correspond to a valid multibyte character or—if <code>dst</code> isn't a null pointer—	

when the next multibyte character would exceed the limit of `len` total bytes to be stored in the array pointed to by `dst`. If `dst` isn't a null pointer, the object pointed to by `src` is assigned either a null pointer (if a terminating null wide character was reached) or the address just past the last wide character converted (if any). If the conversion ends at a null wide character, the resulting state is the initial conversion state.

Returns Number of bytes in the resulting multibyte character sequence, not including any terminating null character. Returns `(size_t)(-1)` and stores `EILSEQ` in `errno` if a wide character is encountered that doesn't correspond to a valid multibyte character. 25.5

wcsspn *Search Wide String for Initial Span of Characters in Set (C99)* <wchar.h>
`size_t wcsspn(const wchar_t *s1, const wchar_t *s2);`
 Wide-character version of `strspn`. 25.5

wcsstr *Search Wide String for Substring (C99)* <wchar.h>
`wchar_t *wcsstr(const wchar_t *s1, const wchar_t *s2);`
 Wide-character version of `strstr`. 25.5

wcstod *Convert Wide String to Double (C99)* <wchar.h>
`double wcstod(const wchar_t * restrict nptr,
 wchar_t ** restrict endptr);`
 Wide-character version of `strtod`. 25.5

wcstof *Convert Wide String to Float (C99)* <wchar.h>
`float wcstof(const wchar_t * restrict nptr,
 wchar_t ** restrict endptr);`
 Wide-character version of `strtof`. 25.5

wcstoimax *Convert Wide String to Greatest-Width Integer (C99)* <inttypes.h>
`intmax_t wcstoimax(const wchar_t * restrict nptr,
 wchar_t ** restrict endptr,
 int base);`
 Wide-character version of `strtoimax`. 27.2

wcstok *Search Wide String for Token (C99)* <wchar.h>
`wchar_t *wcstok(wchar_t * restrict s1,
 const wchar_t * restrict s2,
 wchar_t ** restrict ptr);`

Searches the wide string pointed to by `s1` for a "token" consisting of wide characters not in the wide string pointed to by `s2`. If a token exists, the character following it is changed to a null wide character. If `s1` is a null pointer, a search begun by a previous call of `wcstok` is continued; the search begins immediately after the null wide character at the end of the previous token. `ptr` points to an object of

type `wchar_t *` that `wcstok` modifies to keep track of its progress. If `s1` is a null pointer, this object must be the same one used in a previous call of `wcstok`; it determines which wide string is to be searched and where the search is to begin.

Returns A pointer to the first wide character of the token. Returns a null pointer if no token could be found. 25.5

wcstol *Convert Wide String to Long Integer (C99)* <wchar.h>
`long int wcstol(const wchar_t * restrict nptr,`
`wchar_t ** restrict endptr, int base);`
 Wide-character version of `strtol`. 25.5

wcstold *Convert Wide String to Long Double (C99)* <wchar.h>
`long double wcstold(const wchar_t * restrict nptr,`
`wchar_t ** restrict endptr);`
 Wide-character version of `strtold`. 25.5

wcstoll *Convert Wide String to Long Long Integer (C99)* <wchar.h>
`long long int wcstoll(const wchar_t * restrict nptr,`
`wchar_t ** restrict endptr,`
`int base);`
 Wide-character version of `strtoll`. 25.5

wcstombs *Convert Wide String to Multibyte String* <stdlib.h>
`size_t wcstombs(char * restrict s,`
`const wchar_t * restrict pwcs,`
`size_t n);`
 Converts a sequence of wide characters into corresponding multibyte characters. `pwcs` points to an array containing the wide characters. The multibyte characters are stored in the array pointed to by `s`. Conversion ends if a null character is stored or if storing a multibyte character would exceed the limit of `n` bytes.
Returns Number of bytes stored, not including the terminating null character, if any. Returns `(size_t) (-1)` if a wide character is encountered that doesn't correspond to a valid multibyte character. 25.2

wcstoul *Convert Wide String to Unsigned Long Integer (C99)* <wchar.h>
`unsigned long int wcstoul(`
`const wchar_t * restrict nptr,`
`wchar_t ** restrict endptr, int base);`
 Wide-character version of `strtoul`. 25.5

wcstoull *Convert Wide String to Unsigned Long Long Integer (C99)* <wchar.h>
`unsigned long long int wcstoull(`
`const wchar_t * restrict nptr,`
`wchar_t ** restrict endptr, int base);`
 Wide-character version of `strtoull`. 25.5

wcstoumax	<i>Convert Wide String to Unsigned Greatest-Width Integer (C99)</i>	<inttypes.h>
	<pre>uintmax_t wcstoumax(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);</pre>	
	Wide-character version of strtoumax.	27.2
wcsxfrm	<i>Transform Wide String (C99)</i>	<wchar.h>
	<pre>size_t wcsxfrm(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);</pre>	
	Wide-character version of strxfrm.	25.5
wctob	<i>Convert Wide Character to Byte (C99)</i>	<wchar.h>
	<pre>int wctob(wint_t c);</pre>	
<i>Returns</i>	Single-byte representation of <i>c</i> as an unsigned char converted to int. Returns EOF if <i>c</i> doesn't correspond to one multibyte character in the initial shift state.	25.5
wctomb	<i>Convert Wide Character to Multibyte Character</i>	<stdlib.h>
	<pre>int wctomb(char *s, wchar_t wc);</pre>	
	Converts the wide character stored in <i>wc</i> into a multibyte character. If <i>s</i> isn't a null pointer, stores the result in the array that <i>s</i> points to.	
<i>Returns</i>	If <i>s</i> is a null pointer, returns a nonzero or zero value, depending on whether or not multibyte characters have state-dependent encodings. Otherwise, returns the number of bytes in the multibyte character that corresponds to <i>wc</i> ; returns -1 if <i>wc</i> doesn't correspond to a valid multibyte character.	25.2
wctrans	<i>Define Wide-Character Mapping (C99)</i>	<wctype.h>
	<pre>wctrans_t wctrans(const char *property);</pre>	
<i>Returns</i>	If <i>property</i> identifies a valid mapping of wide characters according to the LC_CTYPE category of the current locale, returns a nonzero value that can be used as the second argument to the towctrans function; otherwise, returns zero.	25.6
wctype	<i>Define Wide-Character Class (C99)</i>	<wctype.h>
	<pre>wctype_t wctype(const char *property);</pre>	
<i>Returns</i>	If <i>property</i> identifies a valid class of wide characters according to the LC_CTYPE category of the current locale, returns a nonzero value that can be used as the second argument to the iswctype function; otherwise, returns zero.	25.6
wmemchr	<i>Search Wide-Character Memory Block for Character (C99)</i>	<wchar.h>
	<pre>wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n);</pre>	
	Wide-character version of memchr.	25.5

wmemcmp	<i>Compare Wide-Character Memory Blocks (C99)</i>	<wchar.h>
	<code>int wmemcmp(const wchar_t * s1, const wchar_t * s2, size_t n);</code>	
	Wide-character version of memcmp.	25.5
wmemcpy	<i>Copy Wide-Character Memory Block (C99)</i>	<wchar.h>
	<code>wchar_t *wmemcpy(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);</code>	
	Wide-character version of memcpy.	25.5
wmemmove	<i>Copy Wide-Character Memory Block (C99)</i>	<wchar.h>
	<code>wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);</code>	
	Wide-character version of memmove.	25.5
wmemset	<i>Initialize Wide-Character Memory Block (C99)</i>	<wchar.h>
	<code>wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);</code>	
	Wide-character version of memset.	25.5
wprintf	<i>Wide-Character Formatted Write (C99)</i>	<wchar.h>
	<code>int wprintf(const wchar_t * restrict format, ...);</code>	
	Wide-character version of printf.	25.5
wscanf	<i>Wide-Character Formatted Read (C99)</i>	<wchar.h>
	<code>int wscanf(const wchar_t * restrict format, ...);</code>	
	Wide-character version of scanf.	25.5

APPENDIX E

ASCII Character Set

Decimal	Escape Sequence			Character						
	Oct	Hex	Char							
0	\0	\x00		<i>nul</i>	32		64	@	96	`
1	\1	\x01		<i>soh</i> (^A)	33	!	65	A	97	a
2	\2	\x02		<i>stx</i> (^B)	34	"	66	B	98	b
3	\3	\x03		<i>etx</i> (^C)	35	#	67	C	99	c
4	\4	\x04		<i>eot</i> (^D)	36	\$	68	D	100	d
5	\5	\x05		<i>enq</i> (^E)	37	%	69	E	101	e
6	\6	\x06		<i>ack</i> (^F)	38	&	70	F	102	f
7	\7	\x07	\a	<i>bel</i> (^G)	39	'	71	G	103	g
8	\10	\x08	\b	<i>bs</i> (^H)	40	(72	H	104	h
9	\11	\x09	\t	<i>ht</i> (^I)	41)	73	I	105	i
10	\12	\x0a	\n	<i>lf</i> (^J)	42	*	74	J	106	j
11	\13	\x0b	\v	<i>vt</i> (^K)	43	+	75	K	107	k
12	\14	\x0c	\f	<i>ff</i> (^L)	44	,	76	L	108	l
13	\15	\x0d	\r	<i>cr</i> (^M)	45	-	77	M	109	m
14	\16	\x0e		<i>so</i> (^N)	46	.	78	N	110	n
15	\17	\x0f		<i>si</i> (^O)	47	/	79	O	111	o
16	\20	\x10		<i>dle</i> (^P)	48	0	80	P	112	p
17	\21	\x11		<i>dc1</i> (^Q)	49	1	81	Q	113	q
18	\22	\x12		<i>dc2</i> (^R)	50	2	82	R	114	r
19	\23	\x13		<i>dc3</i> (^S)	51	3	83	S	115	s
20	\24	\x14		<i>dc4</i> (^T)	52	4	84	T	116	t
21	\25	\x15		<i>nak</i> (^U)	53	5	85	U	117	u
22	\26	\x16		<i>syn</i> (^V)	54	6	86	V	118	v
23	\27	\x17		<i>etb</i> (^W)	55	7	87	W	119	w
24	\30	\x18		<i>can</i> (^X)	56	8	88	X	120	x
25	\31	\x19		<i>em</i> (^Y)	57	9	89	Y	121	y
26	\32	\x1a		<i>sub</i> (^Z)	58	:	90	Z	122	z
27	\33	\x1b		<i>esc</i>	59	;	91	[123	{
28	\34	\x1c		<i>fs</i>	60	<	92	\	124	
29	\35	\x1d		<i>gs</i>	61	=	93]	125	}
30	\36	\x1e		<i>rs</i>	62	>	94	^	126	~
31	\37	\x1f		<i>us</i>	63	?	95	_	127	del

BIBLIOGRAPHY

The best book on programming for the layman is "Alice in Wonderland"; but that's because it's the best book on anything for the layman.

C Programming

- Feuer, A. R., *The C Puzzle Book*, Revised Printing, Addison-Wesley, Reading, Mass., 1999. Contains numerous “puzzles”—small C programs whose output the reader is asked to predict. The book shows the correct output of each program and provides a detailed explanation of how it works. Good for testing your C knowledge and reviewing the fine points of the language.
- Harbison, S. P., III, and G. L. Steele, Jr., *C: A Reference Manual*, Fifth Edition, Prentice-Hall, Upper Saddle River, N.J., 2002. The ultimate C reference—essential reading for the would-be C expert. Covers both C89 and C99 in considerable detail, with frequent discussions of implementation differences found in C compilers. Not a tutorial—assumes that the reader is already well versed in C.
- Kernighan, B. W., and D. M. Ritchie, *The C Programming Language*, Second Edition, Prentice-Hall, Englewood Cliffs, N.J., 1988. The original C book, affectionately known as K&R or simply “the White Book.” Includes both a tutorial and a complete C reference manual. The second edition reflects the changes made in C89.
- Koenig, A., *C Traps and Pitfalls*, Addison-Wesley, Reading, Mass., 1989. An excellent compendium of common (and some not-so-common) C pitfalls. Forewarned is forearmed.
- Plauger, P. J., *The Standard C Library*, Prentice-Hall, Englewood Cliffs, N.J., 1992. Not only explains all aspects of the C89 standard library, but provides complete source code! There’s no better way to learn the library than to study this book. Even if your interest in the library is minimal, the book is worth getting just for the opportunity to study C code written by a master.

- Ritchie, D. M., The development of the C programming language, in *History of Programming Languages II*, edited by T. J. Bergin, Jr., and R. G. Gibson, Jr., Addison-Wesley, Reading, Mass., 1996, pages 671–687. A brief history of C written by the language’s designer for the Second ACM SIGPLAN History of Programming Languages Conference, which was held in 1993. The article is followed by transcripts of Ritchie’s presentation at the conference and the question-and-answer session with the audience.
- Ritchie, D. M., S. C. Johnson, M. E. Lesk, and B. W. Kernighan, UNIX time-sharing system: the C programming language, *Bell System Technical Journal* 57, 6 (July–August 1978), 1991–2019. A famous article that discusses the origins of C and describes the language as it looked in 1978.
- Rosler, L., The UNIX system: the evolution of C—past and future, *AT&T Bell Laboratories Technical Journal* 63, 8 (October 1984), 1685–1699. Traces the evolution of C from 1978 to 1984 and beyond.
- Summit, S., *C Programming FAQs: Frequently Asked Questions*, Addison-Wesley, Reading, Mass., 1996. An expanded version of the FAQ list that has appeared for years in the Usenet *comp.lang.c* newsgroup.
- van der Linden, P., *Expert C Programming*, Prentice-Hall, Englewood Cliffs, N.J., 1994. Written by one of the C wizards at Sun Microsystems, this book manages to entertain and inform in equal amounts. With its profusion of anecdotes and jokes, it makes learning the fine points of C seem almost fun.

UNIX Programming

- Rochkind, M. J., *Advanced UNIX Programming*, Second Edition, Addison-Wesley, Boston, Mass., 2004. Covers UNIX system calls in considerable detail. This book, along with the one by Stevens and Rago, is a must-have for C programmers who use the UNIX operating system or one of its variants.
- Stevens, W. R., and S. A. Rago, *Advanced Programming in the UNIX Environment*, Second Edition, Addison-Wesley, Upper Saddle River, N.J., 2005. An excellent follow-up to this book for programmers working under the UNIX operating system. Focuses on using UNIX system calls, including standard C library functions as well as functions that are specific to UNIX.

Programming in General

- Bentley, J., *Programming Pearls*, Second Edition, Addison-Wesley, Reading, Mass., 2000. This updated version of Bentley’s classic programming book emphasizes writing efficient programs, but touches on other topics that are crucial for the professional programmer. The author’s light touch makes the book as enjoyable to read as it is informative.

- Kernighan, B. W., and R. Pike, *The Practice of Programming*, Addison-Wesley, Reading, Mass., 1999. Read this book for advice on programming style, choosing the right algorithm, testing and debugging, and writing portable programs. Examples are drawn from C, C++, and Java.
- McConnell, S., *Code Complete*, Second Edition, Microsoft Press, Redmond, Wash., 2004. Tries to bridge the gap between programming theory and practice by providing down-to-earth coding advice based on proven research. Includes plenty of examples in a variety of programming languages. Highly recommended.
- Raymond, E. S., ed., *The New Hacker's Dictionary*, Third Edition, MIT Press, Cambridge, Mass., 1996. Explains much of the jargon that programmers use, and it's great fun to read as well.

Web Resources

- ANSI eStandards Store (webstore.ansi.org). The C99 standard (ISO/IEC 9899:1999) can be purchased at this site. Each set of corrections to the standard (known as a Technical Corrigendum) can be downloaded for free.
- comp.lang.c* Frequently Asked Questions (c-faq.com). Steve Summit's FAQ list for the *comp.lang.c* newsgroup is a must-read for any C programmer.
- Dinkumware (www.dinkumware.com). Dinkumware is owned by P. J. Plauger, the acknowledged master of the C and C++ standard libraries. The web site includes a handy C99 library reference, among other things.
- Google Groups (groups.google.com). One of the best ways to find answers to programming questions is to search the Usenet newsgroups using the Google Groups search engine. If you have a question, it's likely that someone else has already asked the question on a newsgroup and the answer has been posted. Groups of particular interest to C programmers include *alt.comp.lang.learn.c-c++* (for C and C++ beginners), *comp.lang.c* (the primary C language group), and *comp.std.c* (devoted to discussion of the C standard).
- International Obfuscated C Code Contest (www.ioccc.org). Home of an annual contest in which participants vie to see who can write the most obscure C programs.
- ISO/IEC JTC1/SC22/WG14 (www.open-std.org/jtc1/sc22/wg14/). The official web site of WG14, the international working group that created the C99 standard and is responsible for updating it. Of particular interest among the many documents available at the site is the rationale for C99, which explains the reasons for the changes made in the standard.
- Lysator (www.lysator.liu.se/c/). A collection of links to C-related web sites maintained by Lysator, an academic computer society located at Sweden's Linköping University.