

C Primer Plus

Fourth Edition

SAMS

Stephen Prata

C

Primer Plus

Fourth Edition

Stephen Prata

SAMS

800 East 96th St., Indianapolis, Indiana, 46240 USA

C Primer Plus, Fourth Edition

Copyright © 2002 by Sams Publishing

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

International Standard Book Number: 0-672-32222-6

Library of Congress Catalog Card Number: 2001089225

Printed in the United States of America

First Printing: September 2001

06 05 04 03 6 5 4

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Apple is a registered trademark of Apple Computer, Inc.

Borland C++ is a registered trademark of Borland International, Inc.

CodeWarrior is a registered trademark of Metrowerks, Inc.

Cray is a registered trademark of Cray Computer, Inc.

IBM and PC are registered trademarks and PC DOS is a trademark of the International Business Machines Company.

Macintosh is a registered trademark of Macintosh Laboratory, Inc., licensed by Apple Computer, Inc.

Microsoft and MS-DOS are registered trademarks of Microsoft Corporation.

Primer Plus is a registered trademark of The Waite Group, Inc.

Think C is a registered trademark of Symantec Corporation.

Turbo C is a registered trademark of Borland International, Inc.

Unix is a trademark of American Telephone and Telegraph Corporation.

VAX is a registered trademark and VMS is a trademark of Digital Equipment Corporation.

Windows is a registered trademark of Microsoft Corporation.

WordPerfect is a registered trademark of WordPerfect Corporation.

WordStar is a registered trademark of MicroPro International Corporation.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or from the use of the Web site or programs accompanying it.

ASSOCIATE PUBLISHER

Linda Engelman

ACQUISITIONS EDITORS

Linda Scharp

Karen Wachs

DEVELOPMENT EDITOR

Karen Wachs

MANAGING EDITOR

Charlotte Clapp

PROJECT EDITOR

Sheila Schroeder

COPY EDITOR

Pat Kinyon

INDEXER

Sandra Henselmeier

PROOFREADER

Plan-It Publishing

TECHNICAL EDITOR

Jeff Perkins

Chris Maunder

TEAM COORDINATORS

Chris Feather

Lynne Williams

MEDIA DEVELOPER

Dan Scherf

INTERIOR DESIGNER

Gary Adair

PAGE LAYOUT

Tim Osborn

CONTENTS AT A GLANCE

PREFACE

xx

CHAPTER 1	Getting Ready	1
CHAPTER 2	Introducing C	23
CHAPTER 3	Data and C	49
CHAPTER 4	Character Strings and Formatted Input/Output	89
CHAPTER 5	Operators, Expressions and Statements	129
CHAPTER 6	C Control Statements: Looping	169
CHAPTER 7	C Control Statements: Branching and Jumps	219
CHAPTER 8	Character Input/Output and Redirection	267
CHAPTER 9	Functions	301
CHAPTER 10	Arrays and Pointers	345
CHAPTER 11	Character Strings and String Functions	397
CHAPTER 12	Storage Classes, Linkage, and Memory Management	449
CHAPTER 13	File Input/Output	493
CHAPTER 14	Structures and Other Data Forms	527
CHAPTER 15	Bit Fiddling	587
CHAPTER 16	The C Preprocessor and the C Library	615
CHAPTER 17	Advanced Data Representation	665
APPENDIX A	Answers to the Review Questions	741
APPENDIX B	Reference Section	781

INDEX

871

TABLE OF CONTENTS

CHAPTER 1: Getting Ready	1
Whence C?	1
Why C?	2
Design Features	2
Efficiency	3
Portability	3
Power and Flexibility	3
Programmer Oriented	3
Shortcomings	4
Whither C?	4
What Computers Do	5
High-Level Computer Languages and Compilers	6
Using C: Seven Steps	7
Step 1: Define the Program Objectives	8
Step 2: Design the Program	8
Step 3: Write the Code	9
Step 4: Compile	9
Step 5: Run the Program	10
Step 6: Test and Debug the Program	10
Step 7: Maintain and Modify the Program	10
Commentary	10
Programming Mechanics	11
Object Code Files, Executable Files, and Libraries	12
UNIX System	13
Linux System	15
Integrated Development Environments (Windows)	15
DOS Compilers for the IBM PC	17
C on the Macintosh	17
Language Standards	18
The First ANSI/ISO C Standard	18
The C99 Standard	19
Book Organization	19
Some Conventions	20
Typeface	20
Screen Output	20
Summary	21
Review Questions	22
Programming Exercise	22

CHAPTER 2: Introducing C	23
A Simple Sample of C	23
The Explanation	24
Pass 1 Quick Synopsis	24
Pass 2 Details	26
The Structure of a Simple Program	34
Tips on Making Your Programs Readable	35
Taking Another Step	36
Documentation	37
Multiple Declarations	37
Multiplication	37
Printing Multiple Values	37
While You're at It... Multiple Functions	38
Debugging	39
Syntax Errors	40
Semantic Errors	41
Program State	42
Keywords and Reserved Identifiers	43
Key Concepts	43
Summary	44
Review Questions	45
Programming Exercises	46
CHAPTER 3: Data and C	49
A Sample Program	49
What's New in This Program?	51
Data Variables and Constants	52
Data: Data-Type Keywords	52
Integer Versus Floating-Point Types	54
The Integer	54
The Floating-Point Number	54
C Data Types	55
The int Type	56
Other Integer Types	59
Using Characters: Type char	64
The _Bool Type	70
Portable Types: inttypes.h	70
Types float, double, and long double	72
Complex and Imaginary Types	76
Other Types	77
Type Sizes	79

Using Data Types	80
Arguments and Pitfalls	81
One More Example	82
What Happens	83
Flushing the Output	84
Key Concepts	84
Summary	85
Review Questions	86
Programming Exercises	88
CHAPTER 4: Character Strings and Formatted Input/Output	89
Introductory Program	89
Character Strings: An Introduction	91
Type char Arrays and the Null Character	91
Using Strings	92
The strlen() Function	93
Constants and the C Preprocessor	95
The const Modifier	98
Manifest Constants on the Job	98
Exploring and Exploiting printf() and scanf()	101
The printf() Function	101
Using printf()	102
Conversion Specification Modifiers for printf()	104
The Meaning of Conversion	110
Using scanf()	116
The * Modifier with printf() and scanf()	121
Usage Tips	123
Key Concepts	124
Summary	124
Review Questions	125
Programming Exercises	127
CHAPTER 5: Operators, Expressions, and Statements	129
Introducing Loops	129
Fundamental Operators	132
Assignment Operator: =	132
Addition Operator: +	134
Subtraction Operator: −	134
Sign Operators: − and +	134
Multiplication Operator: *	135
Division Operator: /	137
Operator Precedence	138
Precedence and the Order of Evaluation	140

Some Additional Operators	141
The sizeof Operator and the size_t Type	142
Modulus Operator: %	142
Increment and Decrement Operators: ++ and --	144
Decrementing: --	147
Precedence	148
Don't Be Too Clever	149
Expressions and Statements	150
Expressions	150
Statements	151
Compound Statements (Blocks)	154
Type Conversions	156
The Cast Operator	158
Function with Arguments	159
A Sample Program	161
Key Concepts	163
Summary	163
Review Questions	164
Programming Exercises	167
CHAPTER 6: C Control Statements: Looping	169
An Initial Example	170
Program Comments	171
C-Style Reading Loop	172
The while Statement	173
Terminating a while Loop	173
When a Loop Terminates	174
while: An Entry-Condition Loop	174
Syntax Points	175
Which Is Bigger: Using Relational Operators and Expressions	176
What Is Truth?	178
What Else Is True?	179
Troubles with Truth	180
The New _Bool Type	182
Precedence of Relational Operators	183
Indefinite Loops and Counting Loops	186
The for Loop	187
Using for for Flexibility	188
More Assignment Operators: +=, -=, *=, /=, %=	192
The Comma Operator	193
Zeno Meets the for Loop	196

An Exit-Condition Loop: do while	198
Which Loop?	200
Nested Loops	201
Program Discussion	202
A Nested Variation	202
Arrays	203
Using a for Loop with an Array	204
A Loop Example Using a Function	
Return Value	206
Program Discussion	208
Using Functions with Return Values	209
Key Concepts	210
Summary	211
Review Questions	212
Programming Exercises	216
CHAPTER 7: C Control Statements: Branching and Jumps	219
The if Statement	220
Adding else to the if Statement	222
Another Example: Introducing getchar() and putchar()	223
The ctype.h Family of Character Functions	226
Multiple Choice else if	228
Pairing else with if	231
More Nested ifs	232
Let's Get Logical	236
Alternate Spellings: the iso646.h header file	237
Precedence	238
Order of Evaluation	238
Ranges	240
A Word-Count Program	240
The Conditional Operator: ?	244
Loop Aids: continue and break	246
The continue Statement	246
The break Statement	249
Multiple Choice: switch and break	250
Using the switch Statement	252
Reading Only the First Character of a Line	254
Multiple Labels	254
switch and if else	256
The goto Statement	257
Avoiding goto	257

Key Concepts	260
Summary	261
Review Questions	262
Programming Exercises	265
CHAPTER 8: Character Input/Output and Input Validation	267
Single-Character I/O: <code>getchar()</code> and <code>putchar()</code>	268
Buffers	269
Terminating Keyboard Input	270
Files, Streams, and Keyboard Input	270
The End of File	271
Redirection and Files	274
Unix, Linux, and DOS Redirection	275
Creating a Friendlier User Interface	279
Working with Buffered Input	279
Mixing Numeric and Character Input	281
Input Validation	284
Analyzing the Program	288
The Input Stream and Numbers	289
Menu Browsing	290
Tasks	290
Toward a Smoother Execution	290
Mixing Character and Numeric Input	292
Key Concepts	295
Summary	296
Review Questions	296
Programming Exercises	297
CHAPTER 9: Functions	301
Reviewing Functions	301
Creating and Using a Simple Function	303
Analyzing the Program	303
Function Arguments	306
Defining a Function with an Argument:	
Formal Parameters	308
Prototyping a Function with Arguments	308
Calling a Function with an Argument: Actual Arguments	309
The Black Box Viewpoint	310
Returning a Value from a Function with <code>return</code>	310
Function Types	313
ANSI C Function Prototyping	314
The Problem	315
The ANSI Solution	316

No Arguments and Unspecified Arguments	317
Hooray for Prototypes	318
Recursion	318
Recursion Revealed	318
Recursion Fundamentals	320
Tail Recursion	321
Recursion and Reversal	323
Recursion Pros and Cons	324
All C Functions Are Created Equal	325
Compiling Programs with Two or More Source Code Files	326
UNIX	326
Linux	326
DOS Command-Line Compilers	326
Windows and Macintosh Compilers	327
Using Header Files	327
Finding Addresses: The & Operator	330
Altering Variables in the Calling Function	332
Pointers: A First Look	334
The Indirection Operator: *	334
Declaring Pointers	335
Using Pointers to Communicate Between Functions	336
Key Concepts	340
Summary	341
Review Questions	341
Programming Exercises	342
CHAPTER 10: Arrays and Pointers	345
Arrays	345
Initialization	346
Designated Initializers (C99)	350
Assigning Array Values	351
Array Bounds	352
Specifying an Array Size	353
Multidimensional Arrays	354
Initializing a Two-Dimensional Array	357
More Dimensions	358
Pointers and Arrays	358
Functions, Arrays, and Pointers	361
Using Pointer Arguments	364
Comment: Pointers and Arrays	367
Pointer Operations	367

Protecting Array Contents	370
Using const with Formal Parameters	371
More About const	373
Pointers and Multidimensional Arrays	375
Pointers to Multi-Dimensional Arrays	377
Pointer Compatibility	379
Functions and Multidimensional Arrays	380
Variable-Length Arrays (VLAs)	383
Compound Literals	387
Key Concepts	389
Summary	390
Review Questions	391
Programming Exercises	393
CHAPTER 11: Character Strings and String Functions	397
Defining Strings Within a Program	399
Character String Constants (String Literals)	399
Character String Arrays and Initialization	400
Array Versus Pointer	401
Arrays of Character Strings	404
Pointers and Strings	405
String Input	406
Creating Space	407
The gets() Function	407
The fgets() Function	409
The scanf() Function	410
String Output	412
The puts() Function	412
The fputs() Function	413
The printf() Function	414
The Do-It-Yourself Option	414
String Functions	417
The strlen() Function	417
The strcat() and strncat() Functions	419
The strcmp() and strncmp() Functions	420
The strcpy() and strncpy() Functions	425
The sprintf() Function	429
Other String Functions	430
A String Example: Sorting Strings	432
Sorting	435
The ctype.h Character Functions and Strings	435

Command-Line Arguments	437
Command-Line Arguments in Integrated Environments	439
Command-Line Arguments with the Macintosh	440
String to Number Conversions	440
Key Concepts	443
Summary	443
Review Questions	444
Programming Exercises	447
CHAPTER 12: Storage Classes, Linkage, and Memory Management	449
Storage Classes	449
Scope	450
Linkage	452
Storage Duration	452
Automatic Variables	453
Register Variables	457
Static Variables with Block Scope	457
Static Variables with External Linkage	459
Static Variables with Internal Linkage	463
Multiple Files	464
Storage-Class Specifiers	464
Storage Classes and Functions	467
Which Storage Class?	467
A Random Number Function and a Static Variable	468
Roll 'Em	471
Allocated Memory: malloc() and free()	475
The Importance of free()	478
The calloc() Function	479
Dynamic Memory Allocation and Variable-Length Arrays	480
Storage Classes and Dynamic Memory Allocation	481
ANSI C Type Qualifiers	481
The const Type Qualifier	482
The volatile Type Qualifier	484
The restrict Type Qualifier	485
New Places for Old Keywords	486
Key Concepts	487
Summary	487
Review Questions	488
Programming Exercises	490

CHAPTER 13: File Input/Output	493
Communicating with Files	493
What Is a File?	494
Levels of I/O	495
Standard Files	495
Standard I/O	496
Checking for Command-Line Arguments	497
The <code>fopen()</code> Function	498
The <code>getc()</code> and <code>putc()</code> Functions	499
End of File	499
The <code>fclose()</code> Function	500
Standard Files	501
A Simple-Minded File-Condensing Program	501
File I/O: <code>fprintf()</code> , <code>fscanf()</code> , <code>fgets()</code> , and <code>fputs()</code>	503
The <code>fprintf()</code> and <code>fscanf()</code> Functions	503
The <code>fgets()</code> and <code>fputs()</code> Functions	504
Adventures in Random Access: <code>fseek()</code> and <code>ftell()</code>	506
How <code>fseek()</code> and <code>ftell()</code> Work	508
Binary Versus Text Mode	509
Portability	510
The <code>fgetpos()</code> and <code>fsetpos()</code> Functions	510
Behind the Scenes with Standard I/O	511
Other Standard I/O Functions	511
The <code>int ungetc(int c, FILE *fp)</code> Function	512
The <code>int fflush(FILE *fp)</code> Function	512
The <code>int setvbuf(FILE *fp, char *buf, int mode, size_t size)</code> Function	512
Binary I/O: <code>fread()</code> and <code>fwrite()</code>	513
The <code>size_t fwrite(void *ptr, size_t size, size_t nmemb, FILE *fp)</code> Function	514
The <code>size_t fread(void *ptr, size_t size, size_t nmemb, FILE *fp)</code> Function	515
The <code>int feof(FILE *fp)</code> and <code>int ferror(FILE *fp)</code> Functions	515
An Example	515
Random Access with Binary I/O	518
Key Concepts	520
Summary	520
Review Questions	521
Programming Exercises	523
CHAPTER 14: Structures and Other Data Forms	527
Sample Problem: Creating an Inventory of Books	527
Setting Up the Structure Declaration	529

Defining a Structure Variable	530
Initializing a Structure	531
Designated Initializers for Structures	532
Gaining Access to Structure Members	532
Arrays of Structures	533
Declaring an Array of Structures	535
Identifying Members of an Array of Structures	535
Program Discussion	536
Nested Structures	537
Pointers to Structures	539
Declaring and Initializing a Structure Pointer	540
Member Access by Pointer	541
Telling Functions About Structures	541
Passing Structure Members	542
Using the Structure Address	543
Passing a Structure as an Argument	544
More on Structure Features	545
Structures or Pointer to Structures?	548
Character Arrays or Character Pointers in a Structure	549
Structure, Pointers, and malloc()	550
Compound Literals and Structures (C99)	552
Flexible Array Members (C99)	554
Functions Using an Array of Structures	556
Saving the Structure Contents in a File	557
Program Points	560
Structures: What Next?	561
Unions: A Quick Look	562
Enumerated Types	565
enum Constants	566
Default Values	566
Assigned Values	566
Usage	567
Shared Namespaces	568
typedef: A Quick Look	569
Fancy Declarations	571
Functions and Pointers	573
Key Concepts	579
Summary	580
Review Questions	581
Programming Exercises	583

CHAPTER 15: Bit Fiddling	587
Binary Numbers, Bits, and Bytes	587
Binary Integers	588
Signed Integers	589
Binary Floating Point	589
Other Bases	590
Octal	590
Hexadecimal	591
C's Bitwise Operators	592
Bitwise Logical Operators	592
Usage: Masks	594
Usage: Turning Bits On	595
Usage: Turning Bits Off	595
Usage: Toggling Bits	595
Usage: Checking the Value of a Bit	596
Bitwise Shift Operators	596
Programming Example	598
Another Example	599
Bit Fields	601
Bit-Field Example	603
Bit Fields and Bitwise Operators	606
Key Concepts	611
Summary	611
Review Questions	612
Programming Exercises	614
CHAPTER 16: The C Preprocessor and the C Library	615
First Steps	616
Manifest Constants: <code>#define</code>	616
Tokens	620
Redefining Constants	620
Using Arguments with <code>#define</code>	621
Creating Strings from Macro Arguments:	
The <code>#</code> Operator	624
Preprocessor Glue: the <code>##</code> Operator	625
Variadic Macros: <code>...</code> and <code>__VA_ARGS__</code>	626
Macro or Function?	627
File Inclusion: <code>#include</code>	628
Header Files: An Example	629
Uses for Header Files	631

Other Directives	632
The #undef Directive	632
Being Defined—the C Preprocessor Perspective	633
Conditional Compilation	633
Predefined Macros	638
#line and #error	639
#pragma	639
Inline Functions	640
The C Library	643
Gaining Access to the C Library	643
Using the Library Descriptions	644
The Math Library	645
The General Utilities Library	648
The exit() and atexit() Functions	648
The qsort() Function	650
The Assert Library	654
memcpy() and memmove() from the string.h Library	656
Variable Arguments: stdarg.h	658
Key Concepts	660
Summary	660
Review Questions	661
Programming Exercises	662
CHAPTER 17: Advanced Data Representation	665
Exploring Data Representation	666
Beyond the Array to the Linked List	668
Using a Linked List	672
Afterthoughts	675
Abstract Data Types (ADTs)	676
Getting Abstract	677
Building an Interface	678
Using the Interface	681
Implementing the Interface	683
Getting Queued with an ADT	689
Implementing the Interface Data Representation	691
Testing the Queue	700
Simulating with a Queue	702
The Linked List Versus the Array	708
Binary Search Trees	711
A Binary Tree ADT	713
The Binary Search Tree Interface	713
The Binary Tree Implementation	716

Trying the Tree	731
Tree Thoughts	735
Other Directions	736
Key Concepts	737
Summary	737
Review Questions	737
Programming Exercises	738
APPENDIX A: Answers to the Review Questions	741
Chapter 1	741
Chapter 2	741
Chapter 3	743
Chapter 4	746
Chapter 5	748
Chapter 6	751
Chapter 7	754
Chapter 8	758
Chapter 9	759
Chapter 10	761
Chapter 11	763
Chapter 12	766
Chapter 13	767
Chapter 14	770
Chapter 15	773
Chapter 16	774
Chapter 17	776
APPENDIX B: Reference Section	781
Section I—Additional Reading	781
Magazine	781
Online Resources	781
C Language Books	782
Programming	783
Reference	783
C++ Books	784
Section II—C Operators	784
Arithmetic Operators	785
Relational Operators	785
Assignment Operators	786
Logical Operators	787
The Conditional Operator	787
Pointer-Related Operators	788

Sign Operators	788
Structure and Union Operators	788
Bitwise Operators	789
Miscellaneous Operators	790
Section III—Basic Types and Storage Classes	790
Summary: The Basic Data Types	790
Summary: How to Declare a Simple Variable	792
Summary: Qualifiers	793
Section IV—Expressions, Statements, and Program Flow	794
Summary: Expressions and Statements	794
Summary: The while Statement	795
Summary: The for Statement	796
Summary: The do while Statement	797
Summary: Using if Statements for Making Choices	797
Summary: Multiple Choice with switch	798
Summary: Program Jumps	799
Section V—The Standard ANSI C Library with C99 Additions	800
Diagnostics: assert.h	801
Complex Numbers: complex.h (C99)	801
Character Handling: ctype.h	803
Error Reporting: errno.h	804
Floating-Point Environment: fenv.h (C99)	805
Format Conversion of Integer Types: inttypes.h (C99)	807
Localization: locale.h	808
Math Library: math.h	811
Non-Local Jumps: setjmp.h	817
Signal Handling: signal.h	817
Variable Arguments: stdarg.h	818
Boolean Support: stdbool.h (C99)	819
Common Definitions: stddef.h	820
Integer Types: stdint.h	820
Standard I/O Library: stdio.h	824
General Utilities: stdlib.h	827
String Handling: string.h	834
Type-Generic Math: tgmath.h (C99)	837
Date and Time: time.h	838
Extended Multibyte and Wide Character Utilities: wchar.h (C99)	842
Wide Character Classification and Mapping Utilities: wctype.h (C99)	849
Section VI—Extended Integer Types	852
Exact Width Types	853
Minimum Width Types	853

Fastest Minimum Width Types	854
Maximum Width Types	855
Integers That Can Hold Pointer Values	855
Extended Integer Constants	855
Section VII—Expanded Character Support	856
Trigraph Sequences	856
Digraphs	857
Alternative Spellings: iso646.h	857
Multibyte Characters	858
Universal Character Names (UCNs)	858
Wide Characters	859
Wide Characters and Multibyte Characters	860
Section VIII—C99 Numeric Computational Enhancements	860
The IEC Floating-Point Standard	861
The fenv.h Header File	861
The STDC FP_CONTRACT Pragma	862
Additions to the math.h Library	862
Support for Complex Numbers	863
Section IX—Differences Between C and C++	864
Function Prototypes	864
char Constants	865
The const Modifier	866
Structures and Unions	867
Enumerations	867
Pointer to void	868
Boolean Types	868
Alternative Spellings	868
Wide Character Support	868
Complex Types	868
Inline Functions	869
C++ Doesn't Have It	869
INDEX	871

PREFACE

C was a relatively little-known language when the first edition of *C Primer Plus* was written in 1984. Since then, the language has boomed, and many people have learned C with the help of this book. In fact, over 500,000 people have purchased *C Primer Plus* throughout its various editions.

With the emergence of a new standard for C, it's time for a 4th edition. As with all the editions, my aim has been to create an introduction to C that is instructive, clear, and helpful.

Approach and Goals

My goal is for this book to serve as a friendly, easy-to-use, self-study guide. To accomplish that objective, *C Primer Plus* employs the following strategies:

- Programming concepts are explained, along with details of the C language; the book does *not* assume that you are a professional programmer.
- Many short, easily-typed examples illustrate just one or two concepts at a time, because learning by doing is one of the most effective ways to master new information.
- Figures and illustrations clarify concepts that are difficult to grasp in words alone.
- Highlight boxes summarize the main features of C for easy reference and review.
- Review questions and programming exercises at the end of each chapter allow you to test and improve your understanding of C.

To gain the greatest benefit, you should take as active a role as possible in studying the topics in this book. Don't just read the examples, enter them into your system and try them. C is a very portable language, but you may find differences between how a program works on your system and how it works on ours. Experiment—change part of a program to see what the effect is. Modify a program to do something slightly different. Ignore the occasional warnings and see what happens when you do the wrong thing. Try the questions and exercises. The more you do yourself, the more you will learn and remember.

I hope that you'll find this newest edition an enjoyable and effective introduction to the C language.

Changes in the 4th Edition

There is a new standard for the C language. It's called the ISO/IEC 9899:1999 International Standard, but among friends it often goes by the simpler name of C99. It was adopted by the International Organization for Standardization (ISO) and the International Electrotechnical

Committee (IEC) in 1999 and approved as the American standard by the American National Standards Institute (ANSI) in 2000. This new edition of *C Primer Plus* incorporates the new standard. Here are some of the new features covered:

- Extended integer types
- Expanded character support
- Boolean support
- Variable-length arrays
- Compound literals
- Designated initializers
- Expanded computational support
- Inline functions

This edition also reorganizes the presentation of some topics. For example, the discussion of pointers in Chapter 10, “Arrays and Pointers,” has been consolidated and expanded, and Chapter 12, “Storage Classes, Linkage, and Memory Management,” incorporates dynamic memory allocation into the discussion of C storage classes and memory management. Numerous other changes and additions have been incorporated in response to reader requests to make this edition an even more effective learning tool.

ABOUT THE AUTHOR

Stephen Prata is a professor of physics and astronomy at the College of Marin in Kentfield, California, where he teaches astronomy, physics, and programming. He received his B.S. from the California Institute of Technology and his Ph.D. from the University of California, Berkeley. His association with computers began with the computer modeling of star clusters. Stephen has authored or coauthored over a dozen books, including *C++ Primer Plus* and *Unix Primer Plus*.

DEDICATION

With love to Vicky and Bill Prata, who, for more than 65 years, have been showing how rewarding a marriage can be. —SP

ACKNOWLEDGMENTS

I wish to thank Linda Scharp of Sams Publishing for getting this project underway and Karen Wachs of Sams Publishing for seeing it through. Also, thank you Ron Liechty of Metrowerks and Greg Comeau of Comeau Computing for your help with new C99 features and your noteworthy commitment to customer service.

TELL US WHAT YOU THINK!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

As an Associate Publisher for Sams Publishing, I welcome your comments. You can email or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author as well as your name and phone or fax number. I will carefully review your comments and share them with the author and editors who worked on the book.

Email: feedback@sampublishing.com

Mail: Michael Stephens
 Sams Publishing
 800 East 96th Street
 Indianapolis, IN 46240 USA

This page intentionally left blank

CHAPTER 3

DATA AND C

You will learn about the following in this chapter:

- Keywords:
`int, short, long, unsigned, char, float, double, _Bool, _Complex, _Imaginary`
- Operator:
`sizeof`
- Function:
`scanf()`
- The basic data types that C uses
- The distinctions between integer types and floating-point types
- Writing constants and declaring variables of those types.
- How to use the `printf()` and `scanf()` functions to read and write values of different types.

Programs work with data. You feed numbers, letters, and words to the computer, and you expect it to do something with the data. For example, you might want the computer to calculate an interest payment or display a sorted list of vintners. In this chapter, you do more than just read about data; you practice manipulating data, which is much more fun.

This chapter explores the two great families of data types: integer and floating point. C offers several varieties of these types. This chapter tells you what the types are, how to declare them, and how and when to use them. Also, you discover the differences between constants and variables and, as a bonus, your first interactive program is coming up shortly.

A Sample Program

Once again, you begin with a sample program. As before, you'll find some unfamiliar wrinkles that we'll soon iron out for you. The program's general intent should be clear, so try compiling and running the source code shown in Listing 3.1. To save time, you can omit typing the comments.

LISTING 3.1 The rhodium.c Program

```

/* rhodium.c -- your weight in rhodium */
#include <stdio.h>
int main(void)
{
    float weight;      /* user weight          */
    float value;       /* rhodium equivalent */

    printf("Are you worth your weight in rhodium?\n");
    printf("Let's check it out.\n");
    printf("Please enter your weight in pounds: ");

    /* get input from the user */
    scanf("%f", &weight);
    /* assume platinum is $2000 per ounce */
    /* 14.5833 converts pounds avd. to ounces troy */
    value = 2000.0 * weight * 14.5833;
    printf("Your weight in rhodium is worth $%.2f.\n", value);
    printf("You are easily worth that! If rhodium prices drop,\n");
    printf("eat more to maintain your value.\n");
    return 0;
}

```

**Errors and Warnings**

If you type this program incorrectly and, say, omit a semicolon, the compiler gives you a syntax error message. Even if you type it correctly, however, the compiler may give you a warning similar to “Warning—conversion from ‘double’ to ‘float,’ possible loss of data.” An error message means you did something wrong and prevents the program from being compiled. A *warning*, however, means you’ve done something that is valid code but possibly is not what you meant to do. A warning does not stop compilation. This particular warning pertains to how C handles values like 2000.0. It’s not a problem for this example, and the chapter explains the warning later.

When you type this program, you might want to change the **2000.0** to the current price of the precious metal rhodium. Don’t, however, fiddle with the **14.5833**, which represents the number of ounces in a pound. (That’s ounces troy, used for precious metals, and pounds avoirdupois, used for people—precious and otherwise.)

Note that “entering” your weight means to type your weight and then press the Enter or Return key. (Don’t just type your weight and wait.) Pressing Enter informs the computer that you have finished typing your response. The program expects you to enter a number, such as **160**, not words, such as **too much**. Entering letters rather than digits causes problems that require an **if** statement (Chapter 7, “C Control Statements: Branching and Jumps”) to defeat, so please be polite and enter a number. Here is a sample output:

```

Are you worth your weight in rhodium?
Let's check it out.
Please enter your weight in pounds: 160

```

Your weight in rhodium is worth \$4666656.00.
 You are easily worth that! If rhodium prices drop,
 eat more to maintain your value.

What's New in This Program?

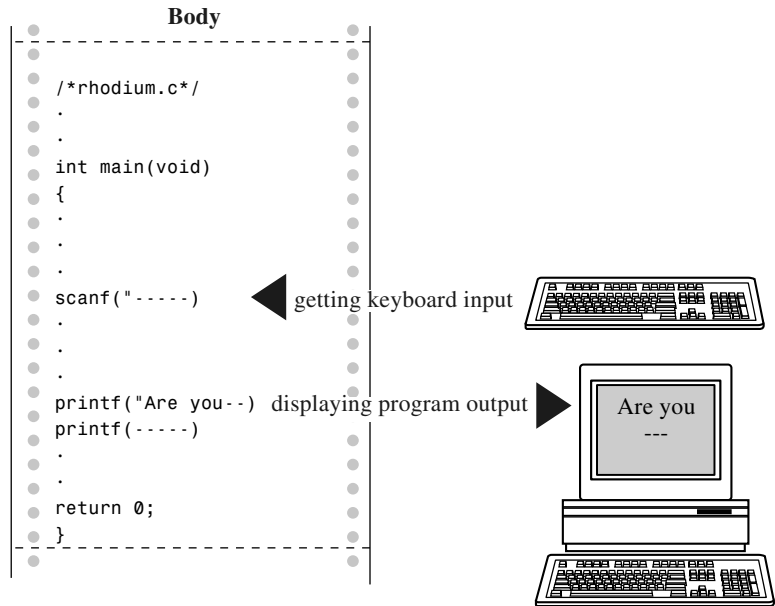
There are several new elements of C in this program:

- Notice that the code uses a new kind of variable declaration. The previous examples just used an integer variable type (`int`), but this one adds a floating-point variable type (`float`) so that you can handle a wider variety of data. The `float` type can hold numbers with decimal points.
- The program demonstrates some new ways of writing constants. You now have numbers with decimal points.
- To print this new kind of variable, use the `%f` specifier in the `printf()` code to handle a floating-point value. Use the `.2` modifier to the `%f` specifier to fine-tune the appearance of the output so that it displays two places to the right of the decimal.
- To provide keyboard input to the program, use the `scanf()` function. The `%f` instructs `scanf()` to read a floating-point number from the keyboard, and the `&weight` tells `scanf()` to assign the input value to the variable named `weight`. The `scanf()` function uses the `&` notation to indicate where it can find the `weight` variable. The next chapter discusses `&` further; meanwhile, trust us that you need it here.
- Perhaps the most outstanding new feature is that this program is interactive. The computer asks you for information and then uses the number you enter. An interactive program is more interesting to use than the noninteractive types. More important, the interactive approach makes programs more flexible. For example, the sample program can be used for any reasonable weight, not just for 160 pounds. You don't have to rewrite the program every time you want to try it on a new person. The `scanf()` and `printf()` functions make this interactivity possible. The `scanf()` function reads data from the keyboard and delivers that data to the program, and `printf()` reads data from a program and delivers that data to your screen. Together, these two functions enable you to establish a two-way communication with your computer (see Figure 3.1), and that makes using a computer much more fun.

This chapter explains the first two items in this list of new features: variables and constants of various data types. Chapter 4, “Character Strings and Formatted Input/Output,” covers the last three items, but this chapter will continue to make limited use of `scanf()` and `printf()`.

FIGURE 3.1

The `scanf()` and `printf()` functions at work.



Data Variables and Constants

A computer, under the guidance of a program, can do many things. It can add numbers, sort names, command the obedience of a speaker or video screen, calculate cometary orbits, prepare a mailing list, dial phone numbers, draw stick figures, draw conclusions, or anything else your imagination can create. To do these tasks, the program needs to work with *data*, the numbers and characters that bear the information you use. Some data are preset before a program is used and keep their values unchanged throughout the life of the program. These are *constants*. Other data may change or be assigned values as the program runs; these are *variables*. In the sample program, `weight` is a variable and `14.5833` is a constant. What about the `2000.0`? True, the price of rhodium isn't a constant in real life, but this program treats it as a constant. The difference between a variable and a constant is that a variable can have its value assigned or changed while the program is running, and a constant can't.

Data: Data-Type Keywords

Beyond the distinction between variable and constant is the distinction between different *types* of data. Some data are numbers. Some are letters or, more generally, characters. The computer needs a way to identify and use these different kinds. C does this by recognizing several fundamental *data types*. If a datum is a constant, the compiler can usually tell its type just by the way it looks: `42` is an integer, and `42.100` is floating point. A variable, however, needs to have its type announced in a declaration statement. You'll learn the details of declaring variables as you

move along. First, though, take a look at the fundamental types recognized by C. K&R C recognized seven keywords relating to types. The C90 standard added two to the list. The C99 standard adds yet another three (see Table 3.1).

TABLE 3.1 C Data Keywords

Original K&R Keywords	C90 Keywords	C99 Keywords
int	signed	_Bool
long	void	_Complex
short		_Imaginary
unsigned		
char		
float		
double		

The `int` keyword provides the basic class of integers used in C. The next three keywords (`long`, `short`, and `unsigned`) and the ANSI addition `signed` are used to provide variations of the basic type. Next, the `char` keyword designates the type used for letters of the alphabet and for other characters, such as `#`, `$`, `%`, and `*`. The `char` type also can be used to represent small integers. Next, `float`, `double`, and the combination `long double` are used to represent numbers with decimal points. The `_Bool` type is for Boolean values (`true` and `false`), and `_Complex` and `_Imaginary` represent complex and imaginary numbers, respectively.

The types created with these keywords can be divided into two families on the basis of how they are stored in the computer: *integer* types and *floating-point* types.

Bits, Bytes, and Words

The terms bit, byte, and word can be used to describe units of computer data or to describe units of computer memory. We'll concentrate on the second usage here.

The smallest unit of memory is called a *bit*. It can hold one of two values: `0` or `1`. (Or you can say that the bit is set to "off" or "on.") You can't store much information in one bit, but a computer has a tremendous stock of them. The bit is the basic building block of computer memory.

The *byte* is the usual unit of computer memory. For nearly all machines, a byte is 8 bits, and that is the standard definition, at least when used to measure storage. (The C language, however, has a different definition, as discussed in the "Using Characters: Type `char`" section later in this chapter). Because each bit can be either 0 or 1, there are 256 (that's 2 times itself 8 times) possible bit patterns of 0s and 1s that can fit in an 8-bit byte. These patterns can be used, for example, to represent

the integers from 0 to 255 or to represent a set of characters. Representation can be accomplished with binary code, which uses (conveniently enough) just 0s and 1s to represent numbers. (Chapter 15, “Bit Fiddling,” discusses binary code, but you can read through the introductory material of that chapter now if you like.)

A *word* is the natural unit of memory for a given computer design. For 8-bit microcomputers, such as the original Apples, a word is just 8 bits. Early IBM compatibles using the 80286 processor are 16-bit machines. This means that they have a word size of 16 bits. Machines like the Pentium-based PCs and the Macintosh PowerPCs have 32-bit words. More powerful computers can have 64-bit words or even larger.

Integer Versus Floating-Point Types

Integer types? Floating-point types? If you find these terms disturbingly unfamiliar, relax. We are about to give you a brief rundown of their meanings. If you are unfamiliar with bits, bytes, and words, you might want to read the nearby note about them first. Do you have to learn all the details? Not really, not any more than you have to learn the principles of internal combustion engines to drive a car, but knowing a little about what goes on inside a computer or engine can help you occasionally.

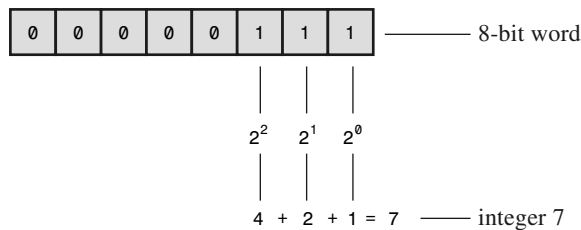
For a human, the difference between integers and floating-point numbers is reflected in the way they can be written. For a computer, the difference is reflected in the way they are stored. Let's look at each of the two classes in turn.

The Integer

An *integer* is a number with no fractional part. In C, an integer is never written with a decimal point. Examples are 2, -23, and 2456. Numbers like 3.14, 0.22, and 2.000 are not integers. Integers are stored as binary numbers. The integer 7, for example, is written 111 in binary. Therefore, to store this number in an 8-bit byte, just set the first 5 bits to 0 and the last 3 bits to 1 (see Figure 3.2).

FIGURE 3.2

Storing the integer 7 using a binary code.



The Floating-Point Number

A *floating-point* number more or less corresponds to what mathematicians call a *real number*. Real numbers include the numbers between the integers. Some floating-point numbers are 2.75, 3.16E7, 7.00, and 2e-8. Notice that adding a decimal point makes a value a floating-point value. So 7 is an integer type but 7.00 is a floating-point type. Obviously, there is more

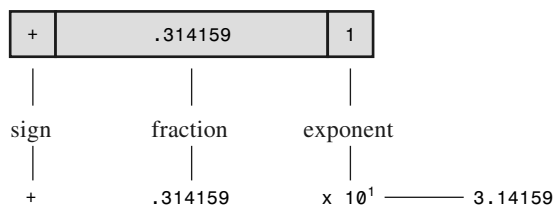
than one way to write a floating-point number. We will discuss the e-notation more fully later, but, in brief, the notation 3.16E7 means to multiply 3.16 by 10 to the 7th power; that is, by 1 followed by 7 zeros. The 7 would be termed the *exponent* of 10.

The key point here is that the scheme used to store a floating-point number is different from the one used to store an integer. Floating-point representation involves breaking up a number into a fractional part and an exponent part and storing the parts separately. Therefore, the 7.00 in this list would not be stored in the same manner as the integer 7, even though both have the same value. The decimal analogy would be to write 7.0 as 0.7E1. Here, 0.7 is the fractional part, and the 1 is the exponent part. Figure 3.3 shows another example of floating-point storage. A computer, of course, would use binary numbers and powers of two instead of powers of ten for internal storage. You'll find more on this topic in Chapter 15. Now, let's concentrate on the practical differences, which are

- An integer has no fractional part; a floating-point number can have a fractional part.
- Floating-point numbers can represent a much larger range of values than integers can. See Table 3.3 near the end of this chapter.
- For some arithmetic operations, such as subtracting one large number from another, floating-point numbers are subject to greater loss of precision.
- Because there are an infinite number of real numbers in any range—for example, in the range between 1.0 and 2.0—computer floating-point numbers can't represent all the values in the range. Instead, floating-point values are often approximations of a true value. For example, 7.0 might be stored as a 6.99999 `float` value—more about precision later.
- Floating-point operations are normally slower than integer operations. However, microprocessors developed specifically to handle floating-point operations are now available, and they have closed the gap.

FIGURE 3.3

Storing the number pi in floating-point format (decimal version).



C Data Types

Now let's look at the specifics of the basic data types used by C. For each type, we describe how to declare a variable, how to represent a constant, and what a typical use would be. Some older C compilers do not support all these types, so check your documentation to see which ones you have available.

The `int` Type

C offers many integer types, and you might wonder why one type isn't enough. The answer is that C gives the programmer the option of matching a type to a particular use. In particular, the C integer types vary in the range of values offered and in whether negative numbers can be used. The `int` type is the basic choice, but should you need other choices to meet the requirements of a particular task or machine, they are available.

The `int` type is a signed integer. That means it must be an integer and it can be positive, negative, or zero. The range in possible values depends on the computer system. Typically, an `int` uses one machine word for storage. Therefore, older IBM PC compatibles, which have a 16-bit word, use 16 bits to store an `int`. This allows a range in values from `-32768` to `32767`. Current personal computers typically have 32-bit integers and fit an `int` to that size. See Table 3.3 near the end of this chapter for examples. ISO/ANSI C specifies that the minimum range for type `int` should be from `-32767` to `32767`. Typically, systems represent signed integers by using the value of a particular bit to indicate the sign. Chapter 15 discusses common methods.

Declaring an `int` Variable

As you saw in Chapter 2, "Introducing C," the keyword `int` is used to declare the basic integer variable. First comes `int`, and then the chosen name of the variable, and then a semicolon. To declare more than one variable, you can declare each variable separately, or you can follow the `int` with a list of names in which each name is separated from the next by a comma. The following are valid declarations:

```
int erns;
int hogs, cows, goats;
```

You could have used a separate declaration for each variable, or you could have declared all four variables in the same statement. The effect is the same: Associate names and arrange storage space for four `int`-sized variables.

These declarations create variables but don't supply values for them. How do variables get values? You've seen two ways that they can pick up values in the program. First, there is assignment:

```
cows = 112;
```

Second, a variable can pick up a value from a function, from `scanf()`, for example. Now let's look at a third way.

Initializing a Variable

To *initialize* a variable means to assign it an initial, or starting, value. In C, this can be done as part of the declaration. Just follow the variable name with the assignment operator (=) and the value you want the variable to have. Here are some examples:

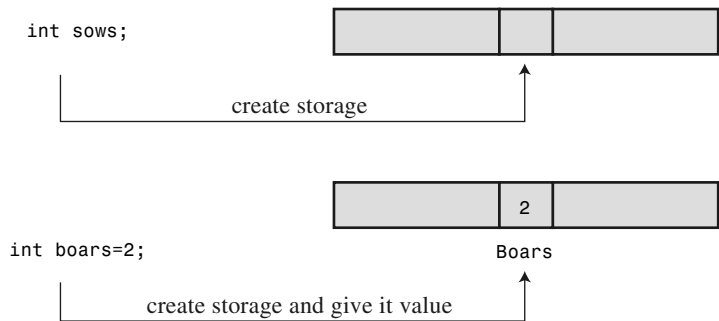
```
int hogs = 21;
int cows = 32, goats = 14;
int dogs, cats = 94;          /* valid, but poor, form */
```

In the last line, only `cats` is initialized. A quick reading might lead you to think that `dogs` is also initialized to `94`, so it is best to avoid putting initialized and noninitialized variables in the same declaration statement.

In short, these declarations create and label the storage for the variables and assign starting values to each (see Figure 3.4).

FIGURE 3.4

Defining and initializing a variable.



Type `int` Constants

The various integers (`21`, `32`, `14`, and `94`) in the last example are integer constants. When you write a number without a decimal point and without an exponent, C recognizes it as an integer. Therefore, `22` and `-44` are integer constants, but `22.0` and `2.2E1` are not. C treats most integer constants as type `int`. Very large integers can be treated differently; see the later discussion of the `long int` type in section “Type `long` and `long long` Constants.”

Printing `int` Values

You can use the `printf()` function to print `int` types. As you saw in Chapter 2, the `%d` notation is used to indicate just where in a line the integer is to be printed. The `%d` is called a *format specifier* because it indicates the form that `printf()` uses to display a value. Each `%d` in the format string must be matched by a corresponding `int` value in the list of items to be printed. That value can be an `int` variable, an `int` constant, or any other expression having an `int` value. It's your job to make sure the number of format specifiers matches the number of values; the compiler won't catch mistakes of that kind. Listing 3.2 presents a simple program that initializes a variable and prints the value of the variable, the value of a constant, and the value of a simple expression. It also shows what can happen if you are not careful.

LISTING 3.2 The `print1.c` Program

```
/* print1.c-displays some properties of printf() */
#include <stdio.h>
int main(void)
{
    int ten = 10;
```

LISTING 3.2 Continued

```

    int two = 2;
    printf("Doing it right: ");
    printf("%d minus %d is %d\n", ten, 2, ten - two );
    printf("Doing it wrong: ");
    printf("%d minus %d is %d\n", ten ); // forgot 2 arguments
}

```

Compiling and running the program produced this output on one system:

```

Doing it right: 10 minus 2 is 8
Doing it wrong: 10 minus 10 is 2

```

Therefore, the first `%d` represents the `int` variable `ten`, the second `%d` represents the `int` constant `2`, and the third `%d` represents the value of the `int` expression `ten - two`. The second time, however, the program used `ten` to provide a value for the first `%d` and used whatever values happened to be lying around in memory for the next two! (The numbers you get could very well be different from those shown here.)

You might be annoyed that the compiler doesn't catch such an obvious error. Blame the unusual design of `printf()`. Most functions take a specific number of arguments, and the compiler can check to see whether you've used the correct number. However, `printf()` can have one, two, three, or more arguments, and that keeps the compiler from using its usual methods for error checking. Remember, check to see that the number of format specifiers you give to `printf()` matches the number of values to be displayed.

Octal and Hexadecimal

Normally, C assumes that integer constants are decimal, or base 10, numbers. However, octal (base 8) and hexadecimal (base 16) numbers are popular with many programmers. Because 8 and 16 are powers of 2, and 10 is not, these number systems occasionally offer a more convenient way for expressing computer-related values. For example, the number 65536, which often pops up in 16-bit machines, is just 10000 in hexadecimal. Also, each digit in a hexadecimal number corresponds to exactly four bits. But how can the computer tell whether 10000 is meant to be a decimal, hexadecimal, or octal value? In C, special prefixes indicate which number base you are using. A prefix of `0x` or `0X` (zero-exe) means that you are specifying a hexadecimal value, so 16 is written as `0x10`, or `0X10`, in hexadecimal. Similarly, a `0` (zero) prefix means that you are writing in octal. For example, the decimal value 16 is written as `020` in octal. Chapter 15 discusses these alternative number bases more fully.

Be aware that this option of using different number systems is provided as a service for your convenience. It doesn't affect how the number is stored. That is, you can write `16` or `020` or `0x10`, and the number is stored exactly the same way in each case—in the binary code used internally by computers.

Displaying Octal and Hexadecimal

Just as C enables you write a number in any one of three number systems, it also enables you to display a number in any of these three systems. To display an integer in octal notation

instead of decimal, use `%o` instead of `%d`. To display an integer in hexadecimal, use `%x`. If you want to display the C prefixes, you can use specifiers `%#o`, `%#x`, and `%#X` to generate the `0`, `0x`, and `0X` prefixes, respectively. Listing 3.3 shows a short example. (Recall that you may have to insert a `getchar();` statement in the code for some IDEs to keep the program execution window from closing immediately.)

LISTING 3.3 The `bases.c` Program

```
/* bases.c--prints 100 in decimal, octal, and hex */
#include <stdio.h>
int main(void)
{
    int x = 100;
    printf("dec = %d; octal = %o; hex = %x\n", x, x, x);
    printf("dec = %d; octal = %#o; hex = %#x\n", x, x, x);
    return 0;
}
```

Compiling and running this program produces this output:

```
dec = 100; octal = 144; hex = 0x64
dec = 100; octal = 0144; hex = 0x64
```

You see the same value displayed in three different number systems. The `printf()` function makes the conversions. Note that the `0` and the `0x` prefixes are not displayed in the output unless you include the `#` as part of the specifier.

Other Integer Types

When you are just learning the language, the `int` type will probably meet most of your integer needs. To be complete, however, we'll cover the other forms now. If you like, you can skim this section and jump to the discussion of the `char` type in the "Using Characters: Type `char`" section, returning here when you have a need.

C offers three adjective keywords to modify the basic integer type: `short`, `long`, and `unsigned`.

- The type `short int` or, more briefly, `short` may use less storage than `int`, thus saving space when only small numbers are needed. Like `int`, `short` is a signed type.
- The type `long int`, or `long`, may use more storage than `int`, thus enabling you to express larger integer values. Like `int`, `long` is a signed type.
- The type `long long int`, or `long long` (both introduced in the C99 standard), may use more storage than `long`, thus enabling you to express even larger integer values. Like `int`, `long long` is a signed type.
- The type `unsigned int`, or `unsigned`, is used for variables that have only nonnegative values. This type shifts the range of numbers that can be stored. For example, a 16-bit `unsigned int` allows a range from `0` to `65535` in value instead of from `-32768` to `32767`. The bit used to indicate the sign of signed numbers now becomes another binary digit, allowing the larger number.

- The types `unsigned long int`, or `unsigned long`, and `unsigned short int`, or `unsigned short`, are recognized as valid by the C90 standard. To this list, C99 adds `unsigned long long int`, or `unsigned long long`.
- The keyword `signed` can be used with any of the signed types to make your intent explicit. For example, `short`, `short int`, `signed short`, and `signed short int` are all names for the same type.

Declaring Other Integer Types

Other integer types are declared in the same manner as the `int` type. The following list shows several examples. Not all older C compilers recognize the last three, and the final example is new with the C99 standard.

```
long int estine;
long johns;
short int erns;
short ribs;
unsigned int s_count;
unsigned players;
unsigned long headcount;
unsigned short yesvotes;
long long ago;
```

Why Multiple Integer Types?

Why do we say that `long` and `short` types “may” use more or less storage than `int`? Because C guarantees only that `short` is no longer than `int` and that `long` is no shorter than `int`. The idea is to fit the types to the machine. On an IBM PC running Windows 3.1, for example, an `int` and a `short` are both 16 bits, and a `long` is 32 bits. On a Windows XL machine or a Macintosh PowerPC, however, a `short` is 16 bits, and both `int` and `long` are 32 bits. The natural word size on a Pentium chip or a PowerPC chip is 32 bits. Because this allows integers in excess of 2 billion (see Table 3.3), the implementers of C on these processor/operating system combinations did not see a necessity for anything larger; therefore, `long` is the same as `int`. For many uses, integers of that size are not needed, so a space-saving `short` was created. The original IBM PC, on the other hand, has only a 16-bit word, which means that a larger `long` was needed.

Now that 64-bit processors are beginning to become more common, there’s a need for 64-bit integers, and that’s the motivation for the `long long` type.

The most common practice today is to set up `long long` as 64 bits, `long` as 32 bits, `short` as 16 bits, and `int` to either 16 bits or 32 bits, depending on the machine’s natural word size. In principle, however, these four types could represent four distinct sizes.

The C standard provides guidelines specifying the minimum allowable size for each basic data type. The minimum range for both `short` and `int` is $-32,767$ to $32,767$, corresponding to a 16-bit unit, and the minimum range for `long` is $-2,147,483,647$ to $2,147,483,647$, corresponding to a 32-bit unit. (Note: For legibility, we’ve used commas, but C code doesn’t allow that option.) For `unsigned short` and `unsigned int`, the minimum range is 0 to 65,535, and

for `unsigned long`, the minimum range is 0 to 4,294,967,295. The `long long` type is intended to support 64-bit needs. Its minimum range is a substantial $-9,223,372,036,854,775,807$ to $9,223,372,036,854,775,807$, and the minimum range for `unsigned long long` is 0 to 18,446,744,073,709,551,615. (For those of you writing checks, that's eighteen quintillion, four hundred and forty-six quadrillion, seven hundred forty-four trillion, seventy-three billion, seven hundred nine million, five hundred fifty-one thousand, six hundred fifteen in U.S. notation, but who's counting?)

When do you use the various `int` types? First, consider `unsigned` types. It is natural to use them for counting because you don't need negative numbers, and the unsigned types enable you to reach higher positive numbers than the signed types.

Use the `long` type if you need to use numbers that `long` can handle and that `int` cannot. However, on systems for which `long` is bigger than `int`, using `long` can slow down calculations, so don't use `long` if it is not essential. One further point, if you are writing code on a machine for which `int` and `long` are the same size, and if you do need 32-bit integers, you should use `long` instead of `int` so that the program will function correctly if transferred to a 16-bit machine.

Similarly, use `long long` if you need 64-bit integer values. Some computers already use 64-bit processors, and 64-bit processing in servers, workstations, and even desktops may soon become common.

Use `short` to save storage space if, say, you need a 16-bit value on a system where `int` is 32-bit. Usually, saving storage space is important only if your program uses arrays of integers that are large in relation to a system's available memory. Another reason to use `short` is that it may correspond in size to hardware registers used by particular components in a computer.



Integer Overflow

What happens if an integer tries to get too big for its type? Let's set an integer to its largest possible value, add to it, and see what happens. Try both signed and unsigned types. (The `printf()` function uses the `%u` specifier to display unsigned `int` values.)

```
/* toobig.c--exceeds maximum int size on our system */
#include <stdio.h>
int main(void)
{
    int i = 2147483647;
    unsigned int j = 4294967295;

    printf("%d %d %d\n", i, i+1, i+2);
    printf("%u %u %u\n", j, j+1, j+2);
    return 0;
}
```

Here is the result for our system:

```
2147483647 -2147483648 -2147483647
4294967295 0 1
```

The unsigned integer `j` is acting like a car's odometer. When it reaches its maximum value, it starts over at the beginning. The integer `i` acts similarly. The main difference is that the `unsigned int` variable `j`, like an odometer, begins at 0, but the `int` variable `i` begins at `-2147483648`. Notice that you are not informed that `i` has exceeded (overflowed) its maximum value. You would have to include your own programming to keep tabs on that.

The behavior described here is mandated by the rules of C for unsigned types. The standard doesn't define how signed types should behave, but the behavior shown here is typical.

long Constants and long long Constants

Normally, when you use a number like 2345 in your program code, it is stored as an `int` type. What if you use a number like 1000000 on a system in which `int` will not hold such a large number? Then the compiler treats it as a `long int`, assuming that type is large enough. If the number is larger than the `long` maximum, C treats it as `unsigned long`. If that is still insufficient, C treats the value as `long long` or `unsigned long long`, if those types are available.

Octal and hexadecimal constants are treated as type `int` unless the value is too large. Then the compiler tries `unsigned int`. If that doesn't work, it tries, in order, `long`, `unsigned long`, `long long`, and `unsigned long long`.

Sometimes you might want the compiler to store a small number as a `long` integer. Programming that involves explicit use of memory addresses on an IBM PC, for instance, can create such a need. Also, some standard C functions require type `long` values. To cause a small constant to be treated as type `long`, you can append an `l` (lowercase `L`) or `L` as a suffix. The second form is better because it looks less like the digit 1. Therefore, a system with a 16-bit `int` and a 32-bit `long` treats the integer 7 as 16 bits and the integer 7L as 32 bits. The `l` and `L` suffixes can also be used with octal and hex integers, as in `020L` and `0x10L`.

Similarly, on those systems supporting the `long long` type, you can use an `ll` or `LL` suffix to indicate a `long long` value, as in `3LL`. Add a `u` or `U` to the suffix for `unsigned long long`, as in `5ull` or `10LLU` or `6LLU` or `9ULL`.

Printing short, long, long long, and unsigned Types

To print an `unsigned int` number, use the `%u` notation. To print a `long` value, use the `%ld` format specifier. If `int` and `long` are the same size on your system, just `%d` will suffice, but your program will not work properly when transferred to a system on which the two types are different, so use the `%ld` specifier for `long`. You can use the `l` prefix for `x` and `o`, too. Therefore, you would use `%lx` to print a long integer in hexadecimal format and `%lo` to print in octal format. Note that although C allows both uppercase and lowercase letters for constant suffixes, these format specifiers use just lowercase.

C has several additional `printf()` formats. First, you can use an `h` prefix for `short` types. Therefore, `%hd` displays a `short` integer in decimal form, and `%ho` displays a `short` integer in octal form. Both the `h` and `l` prefixes can be used with `u` for unsigned types. For instance, you would use the `%lu` notation for printing `unsigned long` types. Listing 3.4 provides an example. Systems supporting the `long long` types use `%lld` and `%llu` for the signed and unsigned versions. Chapter 4 provides a fuller discussion of format specifiers.

LISTING 3.4 The `print2.c` Program

```

/* print2.c-more printf() properties */
#include <stdio.h>
int main(void)
{
    unsigned int un = 3000000000; /* system with 32-bit int */
    short end = 200;             /* and 16-bit short      */
    long big = 65537;
    long long verybig = 12345678908642;
    printf("un = %u and not %d\n", un, un);
    printf("end = %hd and %d\n", end, end);
    printf("big = %ld and not %hd\n", big, big);
    printf("verybig= %lld and not %ld\n", verybig, verybig);
    return 0;
}

```

Here is the output on one system:

```

un = 3000000000 and not -1294967296
end = 200 and 200
big = 65537 and not 1
verybig= 12345678908642 and not 1942899938

```

This example points out that using the wrong specification can produce unexpected results. First, note that using the `%d` specifier for the unsigned variable `un` produces a negative number! The reason for this is that the unsigned value 3000000000 and the signed value -129496296 have exactly the same internal representation in memory on our system. (Chapter 15 explains this property in more detail.) So if you tell `printf()` that the number is unsigned, it prints one value, and if you tell it that the same number is signed, it prints the other value. This behavior shows up with values larger than the maximum signed value. Smaller positive values, such as 96, are stored and displayed the same for both signed and unsigned types.

Next, note that the `short` variable `sn` is displayed the same whether you tell `printf()` that `end` is a `short` (the `%hd` specifier) or an `int` (the `%d` specifier). That's because C automatically expands a type `short` value to a type `int` value when it's passed as an argument to a function. This may raise two questions in your mind: Why does this conversion take place, and what's the use of the `h` modifier? The answer to the first question is that the `int` type is intended to be the integer size that the computer handles most efficiently. So, on a computer for which `short` and `int` are different sizes, it may be faster to pass the value as an `int`. The answer to the second question is that you can use the `h` modifier to show how a longer integer would look if truncated to the size of `short`. The third line of output illustrates this point. When the value 65537 is written in binary format as a 32-bit number, it looks like 00000000000000010000000000000001. Using the `%hd` specifier persuaded `printf()` to look at just the last 16 bits; so it displayed the value as 1. Similarly, the final output line shows the full value of `verybig` and then the value stored in the last 32 bits, as viewed through the `%ld` specifier.

Earlier you saw that it is your responsibility to make sure the number of specifiers matches the number of values to be displayed. Here you see that it is also your responsibility to use the correct specifier for the type of value to be displayed.



Match the Type *printf()* Specifiers

Remember to check to see that you have one format specifier for each value being displayed in a `printf()` statement. And also check that the type of each format specifier matches the type of the corresponding display value.

Using Characters: Type `char`

The `char` type is used for storing characters such as letters and punctuation marks, but technically it is an integer type. Why? Because the `char` type actually stores integers, not characters. To handle characters, the computer uses a numerical code in which certain integers represent certain characters. The most commonly used code in the US is the ASCII code given in Reference Section X, “ASCII Table.” It is the code this book assumes. In it, for example, the integer value **65** represents an uppercase A. So to store the letter A, you actually need to store the integer **65**. (Many IBM mainframes use a different code, called EBCDIC, but the principle is the same. Computer systems outside the U.S. may use entirely different codes.)

The standard ASCII code runs numerically from 0 to 127. This range is small enough that 7 bits can hold it. The `char` type is typically defined as an 8-bit unit of memory, so it is more than large enough to encompass the standard ASCII code. Many systems, such as the IBM PC and the Apple Macintosh, offer extended ASCII codes (different for the two systems) that still stay within an 8-bit limit. More generally, C guarantees that the `char` type is large enough to store the basic character set for the system on which C is implemented.

Many character sets have many more than 127 or even 255 values. For example, there is the Japanese kanji character set. The commercial Unicode initiative has created a system to represent a variety of characters sets worldwide and currently has over 40,000 characters. The International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) has developed a standard called ISO/IEC 10646 for character sets. Fortunately, the Unicode standard has been kept compatible with the more extensive ISO/IEC 10646 standard.

A platform that used one of these sets as its basic character set could use a 16-bit or even a 32-bit `char` representation. The C language defines a byte to be the number of bits used by type `char`, so as far as C documentation goes, a byte would be 16 or 32 bits, rather than 8 bits on such systems.

Declaring Type `char` Variables

As you might expect, `char` variables are declared in the same manner as other variables. Here are some examples:

```
char response;
char itable, latan;
```

This code would create three `char` variables: `response`, `itable`, and `latan`.

Character Constants and Initialization

Suppose you want to initialize a character constant to the letter A. Computer languages are supposed to make things easy, so you shouldn't have to memorize the ASCII code, and you don't. You can assign the character `A` to `grade` with the following initialization:

```
char grade = 'A';
```

A single letter contained between single quotes is a C character constant. When the compiler sees `'A'`, it converts the `'A'` to the proper code value. The single quotes are essential.

```
char broiled;           /* declare a char variable      */
broiled = 'T';          /* OK                                           */
broiled = T;            /* NO! Thinks T is a variable                 */
broiled = "T";          /* NO! Thinks "T" is a string                 */
```

If you leave off the quotes, the compiler thinks that `T` is the name of a variable. If you use double quotes, it thinks you are using a string. We'll discuss strings in Chapter 4.

Because characters are really stored as numeric values, you can also use the numerical code to assign values:

```
char grade = 65; /* ok for ASCII, but poor style */
```

In this example, `65` is type `int`, but, because the value is smaller than the maximum `char` size, it can be assigned to `grade` without any problems. Because `65` is the ASCII code for the letter A, this example assigns the value `A` to `grade`. Note, however, that this example assumes that the system is using ASCII code. Using `'A'` instead of `65` produces code that works on any system. Therefore, it's much better to use character constants than numeric code values.

Somewhat oddly, C treats character constants as type `int` rather than type `char`. For example, on an ASCII system with a 32-bit `int` and an 8-bit `char`, the code

```
char grade = 'B';
```

represents `'B'` as the numerical value `66` stored in a 32-bit unit, but `grade` winds up with `66` stored in an 8-bit unit. This characteristic of character constants makes it possible to define a character constant like `'FATE'`, with four separate 8-bit ASCII codes stored in a 32-bit unit. However, attempting to assign such a character constant to a `char` variable results in only the last 8 bits being used, so the variable gets the value `'E'`.

Nonprinting Characters

The single-quote technique is fine for characters, digits, and punctuation marks, but if you look through Reference Section X, you see that some of the ASCII characters are nonprinting. For example, some represent actions such as backspacing or going to the next line or making the terminal bell ring (or speaker beep). How can these be represented? C offers three ways.

The first way we have already mentioned—just use the ASCII code. For example, the ASCII value for the beep character is 7, so you can do this:

```
char beep = 7;
```

The second way to represent certain awkward characters in C is to use special symbol sequences. These are called *escape sequences*. Table 3.2 shows the escape sequences and their meanings.

TABLE 3.2 Escape Sequences

Sequence	Meaning
\a	Alert (ANSI C)
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\\	Backslash (\)
\'	Single quote (')
\"	Double quote (")
\?	Question mark (?)
\0oo	Octal value (o represents an octal digit)
\xhh	Hexadecimal value (h represents a hexadecimal digit)

Escape sequences must be enclosed in single quotes when assigned to a character variable. For example, you could make the statement

```
nerf = '\n';
```

and then print the variable `nerf` to advance the printer or screen one line.

Now take a closer look at what each escape sequence does. The alert character (`\a`), added by C90, produces an audible or visible alert. The nature of the alert depends on the hardware, with the beep being the most common. (With some systems, the alert character has no effect.) The ANSI standard states that the alert character shall not change the active position. By *active position*, the standard means the location on the display device (screen, teletype, printer, and so on) at which the next character would otherwise appear. In short, the active position is a gen-

eralization of the screen cursor with which you are probably accustomed. Using the alert character in a program displayed on a screen should produce a beep without moving the screen cursor.

Next, the `\b`, `\f`, `\n`, `\r`, `\t`, and `\v` escape sequences are common output device control characters. They are best described in terms of how they affect the active position. A backspace (`\b`) moves the active position back one space on the current line. A form feed character (`\f`) advances the active position to the start of the next page. A newline character (`\n`) sets the active position to the beginning of the next line. A carriage return (`\r`) moves the active position to the beginning of the current line. A horizontal tab character (`\t`) moves the active position to the next horizontal tab stop (typically, they are found at character positions 1, 9, 17, 25, and so on). A vertical tab (`\v`) moves the active position to the next vertical tab position.

These escape characters do not necessarily work with all display devices. For example, the form feed and vertical tab characters produce odd symbols on a PC screen instead of any cursor movement, but they work as described if sent to a printer instead of to the screen.

The next three escape sequences (`\\`, `\'`, and `\"`) enable you to use `\`, `'`, and `"` as character constants. (Because these symbols are used to define character constants as part of a `printf()` command, the situation could get confusing if you use them literally.) Suppose you want to print the following line:

Gramps sez, "a \ is a backslash."

The use this code:

```
printf("Gramps sez, \"a \\ is a backslash.\\\"\\n");
```

The final two forms (`\000` and `\xhh`) are special representations of the ASCII code. To represent a character by its octal ASCII code, precede it with a backslash (`\`) and enclose the whole thing in single quotes. For example, if your compiler doesn't recognize the alert character (`\a`), you could use the ASCII code instead:

```
beep = '\007';
```

You can omit the leading zeros, so `'\07'` or even `'\7'` will do. This notation causes numbers to be interpreted as octal, even if there is no initial `0`.

Beginning with C90, C provides a third option—using a hexadecimal form for character constants. In this case, the backslash is followed by an `x` or `X` and one to three hexadecimal digits. For example, the Control+P character has an ASCII hex code of 10 (16, in decimal), so it can be expressed as `'\x10'` or `'\X010'`. Figure 3.5 shows some representative integer types.

When you use ASCII code, note the difference between numbers and number characters. For example, the character 4 is represented by ASCII code value 52. The notation `'4'` represents the symbol 4, not the numerical value 4.

At this point, you may have three questions.

- *Why aren't the escape sequences enclosed in single quotes in the last example (`printf("Gramps sez, \"a \\ is a backslash\\\"n");`)?*—When a character, be it an escape sequence or not, is part of a string of characters enclosed in double quotes, don't enclose it in single quotes. Notice that none of the other characters in this example (`G`, `r`, `a`, `m`, `p`, `s`, and so on) are marked off by single quotes. A string of characters enclosed in double quotes is called a *character string*. (Chapter 4 explores strings.) Similarly, `printf("Hello!\007\n");` will print `Hello!` and beep, but `printf("Hello!7\n");` will print `Hello!7`. Digits that are not part of an escape sequence are treated as ordinary characters to be printed.
- *When should you use the ASCII code, and when should you use the escape sequences?*—If you have a choice between using one of the special escape sequences, say `'\f'`, or an equivalent ASCII code, say `'\014'`, use the `'\f'`. First, the representation is more mnemonic. Second, it is more portable. If you have a system that doesn't use ASCII code, the `'\f'` will still work.
- *If you need to use numeric code, why use, say, `'\032'` instead of `032`?*—First, using `'\032'` instead of `032` makes it clear to someone reading the code that you intend to represent a character code. Second, an escape sequence like `\032` can be embedded in part of a C string, the way `\007` was in point #1.

FIGURE 3.5
Writing constants with
the *int* family.

Examples of Integer Constants			
type	hexadecimal	octal	decimal
char	<code>\0x41</code>	<code>\0101</code>	N.A.
int	<code>0x41</code>	<code>0101</code>	<code>65</code>
unsigned int	<code>0x41u</code>	<code>0101u</code>	<code>65u</code>
long	<code>0x41L</code>	<code>0101L</code>	<code>65L</code>
unsigned long	<code>0x41UL</code>	<code>0101UL</code>	<code>65UL</code>
long long	<code>0x41LL</code>	<code>0101LL</code>	<code>65LL</code>
unsigned long long	<code>0x41ULL</code>	<code>0101ULL</code>	<code>65ULL</code>

Printing Characters

The `printf()` function uses `%c` to indicate that a character should be printed. Recall that a character variable is stored as a 1-byte integer value. Therefore, if you print the value of a `char` variable with the usual `%d` specifier, you get an integer. The `%c` format specifier tells `printf()` to display the character that has that integer as its code value. Listing 3.5 shows a `char` variable both ways.

LISTING 3.5 The `charcode.c` Program

```

/* charcode.c--displays code number for a character */
#include <stdio.h>
int main(void)
{
    char ch;
    printf("Please enter a character.\n");
    scanf("%c", &ch); /* user inputs character */
    printf("The code for %c is %d.\n", ch, ch);
    return 0;
}

```

Here is a sample run:

```

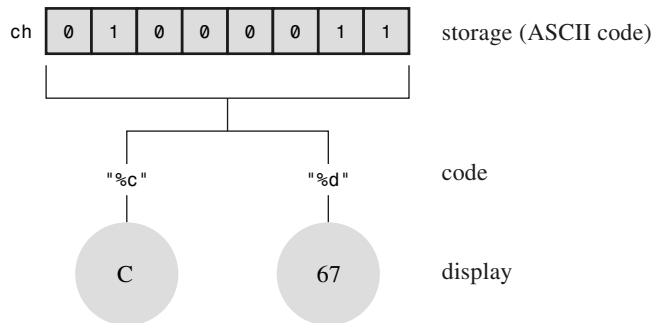
Please enter a character.
C
The code for C is 67.

```

When you use the program, remember to press the Enter or Return key after typing the character. The `scanf()` function then fetches the character you typed, and the ampersand (&) causes the character to be assigned to the variable `ch`. The `printf()` function then prints the value of `ch` twice, first as a character (prompted by the `%c` code) and then as a decimal integer (prompted by the `%d` code). Note that the `printf()` specifiers determine how data is displayed, not how it is stored (see Figure 3.6).

FIGURE 3.6

Data display versus data storage.



Signed or Unsigned?

Some C implementations make `char` a signed type. This means a `char` can hold values typically in the range -128 through 127 . Other implementations make `char` an unsigned type, which provides a range of 0 through 255 . Your compiler manual should tell you which type your `char` is, or you can check the `limits.h` header file, discussed in the next chapter.

With C90, C enabled you to use the keywords `signed` and `unsigned` with `char`. Then, regardless of what your default `char` is, `signed char` would be signed, and `unsigned char` would be unsigned. These versions of `char` are useful if you're using the type to handle small integers. For character use, just use the standard `char` type without modifiers.

The `_Bool` Type

The `_Bool` type is a C99 addition that's used to represent Boolean values, that is, the logical values `true` and `false`. Because C uses the value 1 for `true` and 0 for `false`, the `_Bool` type really is just an integer type, but one that, in principle, only requires one bit of memory, because that is enough to cover the full range from 0 to 1.

Programs use Boolean values to choose which code to execute next. Code execution is covered more fully in Chapter 6, “C Control Statements: Looping” and Chapter 7, “C Control Statements: Branching and Jumps”, so let's defer further discussion until then.

Portable Types: `inttypes.h`

Are there even more integer types? No, but there are more names that you can use for the existing types. You might think you've seen more than an adequate number of names, but the primary names do have a problem. Knowing that a variable is an `int` doesn't tell you how many bits it is unless you check the documentation for your system. To get around this problem, C99 provides an alternative set of names that describe exactly what you get. For example, the name `int16_t` indicates a 16-bit signed integer type and the name `uint32_t` indicates a 32-bit unsigned integer type.

To make these names available to a program, include the `inttypes.h` header file. That file uses the `typedef` facility (first described briefly in Chapter 5, “Operators, Expressions, and Statements”) to create new type names. For example, it will make `uint32_t` a synonym or alias for a standard type with the desired characteristics—perhaps `unsigned int` on one system and `unsigned long` on another. Your compiler will provide a header file consistent with the computer system you are using. These new designations are called *exact width types*. Note that, unlike `int`, `uint32_t` is not a keyword, so the compiler won't recognize it unless you include the `inttypes.h` header file.

One possible problem with attempting to provide exact width types is that a particular system might not support some of the choices, so there is no guarantee that there will be, say, an `int8_t` type (8-bit signed). To get around that problem, the C99 standard defines a second set of names that promise the type is at least big enough to meet the specification and that no other type that can do the job is smaller. These types are called *minimum width types*. For example, `int_least8_t` will be an alias for the smallest available type that can hold an 8-bit signed integer value. If the smallest type on a particular system were 8 bits, the `int8_t` type would not be defined. But the `int_least8_t` type would be available, perhaps implemented as a 16-bit integer.

Of course, some programmers are more concerned with speed than with space. For them, C99 defines a set of types that will allow the fastest computations. These are called the *fastest minimum width types*. For example, the `int_fast8_t` will be defined as an alternative name for the integer type on your system that allows the fastest calculations for 8-bit signed values.

Finally, for some programmers, only the biggest possible integer type on a system will do; `int_max_t` stands for that type, a type that can hold any valid signed integer value. Similarly, `uint_max_t`

`max_t` stands for the largest available unsigned type. Incidentally, if these types could be bigger than `long long` and `unsigned long` because C implementations can define types beyond the required ones.

C99 not only provides these new, portable type names, it also has to assist with input and output. For example, `printf()` requires specific specifiers for particular types. So what do you do to display an `int32_t` value when it might require a `%d` specifier for one definition and a `%ld` for another? The C99 standard provides some string macros (introduced in Chapter 4) to be used to display the portable types. For example, the `inttypes.h` header file will define `PRId16` as a string representing the appropriate specifier (`hd` or `d`, for instance) for a 16-bit signed value. Listing 3.6 shows a brief example illustrating how to use a portable type and its associated specifier.

LISTING 3.6 The `altnames.c` Program

```
/* altnames.c -- portable names for integer types */
#include <stdio.h>
#include <inttypes.h>    // supports portable types
int main(void)
{
    int16_t me16;    // me16 a 16-bit signed variable

    me16 = 4593;
    printf("First, assume int16_t is short: ");
    printf("me16 = %hd\n", me16);
    printf("Next, let's not make any assumptions.\n");
    printf("Instead, use a \"macro\" from inttypes.h: ");
    printf("me16 = %" PRId16 "\n", me16);
    return 0;
}
```

In the final `printf()` argument, the `PRId16` is replaced by its `inttypes.h` definition of `"hd"`, making the line this:

```
printf("me16 = %" "hd" "\n", me16);
```

But C combines consecutive quoted strings into a single quoted string, making the line this:

```
printf("me16 = %hd\n", me16);
```

Here's the output; note that the example also uses the `\` escape sequence to display double quotation marks:

```
First, assume int16_t is short: me16 = 4593
Next, let's not make any assumptions.
Instead, use a "macro" from inttypes.h: me16 = 4593
```

Reference Section VI, "Expanded Integer Types," provides a complete rundown of the `inttypes.h` header file additions, and also lists all the specifier macros.

Types float, double, and long double

The various integer types serve well for most software development projects. However, financial and mathematically oriented programs often make use of *floating-point* numbers. In C, such numbers are called type **float**, **double**, or **long double**. They correspond to the **real** types of FORTRAN and Pascal. The floating-point approach, as already mentioned, enables you to represent a much greater range of numbers, including decimal fractions. Floating-point number representation is similar to *scientific notation*, a system used by scientists to express very large and very small numbers. Let's take a look.

In scientific notation, numbers are represented as decimal numbers times powers of 10. Here are some examples.

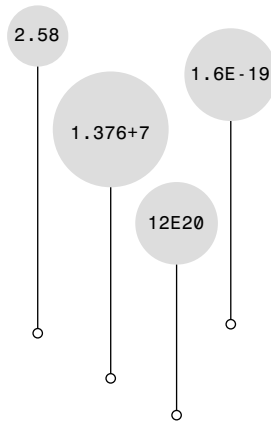
Number	Scientific Notation	Exponential Notation
1,000,000,000	= 1.0×10 ⁹	= 1.0e9
123,000	= 1.23×10 ⁵	= 1.23e5
322.56	= 3.2256×10 ²	= 3.2256e2
0.000056	= 5.6×10 ⁻⁵	= 5.6e-5

The first column shows the usual notation, the second column scientific notation, and the third column exponential notation, or e-notation, which is the way scientific notation is usually written for and by computers, with the *e* followed by the power of 10. Figure 3.7 shows more floating-point representations.

The C standard provides that a **float** has to be able to represent at least six significant figures and allow a range of at least 10⁻³⁷ to 10⁺³⁷. The first requirement means, for example, that a **float** has to represent accurately at least the first six digits in a number like 33.333333. The second requirement is handy if you like to use numbers such as the mass of the sun (2.0e30 kilograms), the charge of a proton (1.6e-19 coulombs), or the national debt. Often, systems use 32 bits to store a floating-point number. Eight bits are used to give the exponent its value and sign, and 24 bits are used to represent the nonexponent part, called the *mantissa* or *significand*, and its sign.

C also has a **double** (for double precision) floating-point type. The **double** type has the same minimum range requirements as **float**, but it extends the minimum number of significant figures that can be represented to 10. Typical **double** representations use 64 bits instead of 32. Some systems use all 32 additional bits for the nonexponent part. This increases the number of significant figures and reduces round-off errors. Other systems use some of the bits to accommodate a larger exponent; this increases the range of numbers that can be accommodated. Either approach leads to at least 13 significant figures, more than meeting the minimum standard.

FIGURE 3.7
Some floating-point
numbers.



C allows for a third floating-point type: **long double**. The intent is to provide for even more precision than **double**. However, C guarantees only that **long double** is at least as precise as **double**.

Declaring Floating-Point Variables

Floating-point variables are declared and initialized in the same manner as their integer cousins. Here are some examples:

```
float noah, jonah;
double trouble;
float planck = 6.63e-34;
long double gnp;
```

Floating-Point Constants

There are many choices open to you when you write a floating-point constant. The basic form of a floating-point constant is a signed series of digits, including a decimal point, followed by an **e** or **E**, followed by a signed exponent indicating the power of 10 used. Here are two valid floating-point constants:

```
-1.56E+12
2.87e-3
```

You can leave out positive signs. You can do without a decimal point (2E5) or an exponential part (19.28), but not both simultaneously. You can omit a fractional part (3.E16) or an integer part (.45E-6), but not both (that wouldn't leave much!). Here are some more valid floating-point constants:

```
3.14159
.2
4e16
.8E-5
100.
```

Don't use spaces in a floating-point constant.

WRONG 1.56 E+12

By default, the compiler assumes floating-point constants are **double** precision. Suppose, for example, that **some** is a **float** variable, and that you have the following statement:

```
some = 4.0 * 2.0;
```

Then the 4.0 and 2.0 are stored as **double**, using (typically) 64 bits for each. The product is calculated using double precision arithmetic, and only then is the answer trimmed to regular **float** size. This ensures greater precision for your calculations, but can slow down a program.

C enables you to override this default by using an **f** or **F** suffix to make the compiler treat a floating-point constant as type **float**; examples are 2.3f and 9.11E9F. An **l** or **L** suffix makes a number type **long double**; examples are 54.3l and 4.32e4L. Note that **L** is less likely to be mistaken for a **1** than is **l**. If the floating-point number has no suffix, it is type **double**.

C99 has added a new format for expressing floating-point constants. It uses a hexadecimal prefix (**0x** or **0X**) with hexadecimal digits, a **p** or **P** instead of **e** or **E**, and an exponent that is a power of 2 instead of a power of 10. Here's what such a number might look like:

```
0xa.1fp10
```

The **a** is 10, the **.1f** is $1/16^{\text{th}}$ plus $15/256^{\text{th}}$, and the **p10** is 2^{10} , or 1024, making the complete value 10364.0 in base ten notation.

Printing Floating-Point Values

The **printf()** function uses the **%f** format specifier to print type **float** and **double** numbers using decimal notation, and it uses **%e** to print them in exponential notation. If your system supports the C99 hexadecimal format for floating-point numbers, you can use **a** or **A** instead of **e** or **E**. The **long double** type requires the **%Lf**, **%Le**, and **%La** specifiers to print that type. Note that both **float** and **double** use the **%f**, **%e**, or **%a** specifiers for output. That's because C automatically expands type **float** values to type **double** when they are passed as arguments to any function, such as **printf()**, that doesn't explicitly prototype the argument type. Listing 3.7 illustrates these behaviors.

LISTING 3.7 The showfpt.c Program

```
/* showf_pt.c-displays float value in two ways */
#include <stdio.h>
int main(void)
{
    float aboat = 32000.0;
    double abet = 2.14e9;
    long double dip = 5.32e-5;
    printf("%f can be written %e\n", aboat, aboat);
```

LISTING 3.7 Continued

```

    printf("%f can be written %e\n", abet, abet);
    printf("%f can be written %e\n", dip, dip);
    return 0;
}

```

This is the output:

```

32000.000000 can be written 3.200000e+04
2140000000.000000 can be written 2.140000e+09
0.000053 can be written 5.320000e-05

```

This example illustrates the default output. The next chapter discusses how to control the appearance of this output by setting field widths and the number of places to the right of the decimal.

Floating-Point Overflow and Underflow

Suppose the biggest possible `float` value on your system is about 3.4E38 and you do this:

```

float toobig = 3.4E38 * 100.0f;
print("%e\n", toobig);

```

What happens? This is an example of *overflow*, when a calculation leads to a number too large to be expressed. The behavior for this case used to be undefined, but now C specifies that `toobig` gets assigned a special value that stands for infinity and that `printf()` displays either `inf` or `infinity` for the value.

What about dividing very small numbers? Here the situation is more involved. Recall that a `float` number is stored as an exponent and as a value part, or mantissa. There will be a number that has the smallest possible exponent and also the smallest value that still uses all the bits available to represent the mantissa. This will be the smallest number that still is represented to the full precision available to a `float` value. Now divide it by 2. Normally, this reduces the exponent, but the exponent already is as small as it can get. So, instead, the computer moves the bits in the mantissa over, vacating the first position and losing the last binary digit. An analogy would be taking a base-ten value with four significant digits like 0.1234E-10, dividing by 10, and getting 0.0123E-10. You get an answer, but you've lost a digit in the process. This situation is called *underflow*, and C refers to floating-point values that have lost the full precision of the type as *subnormal*. So dividing the smallest positive normal floating-point value by 2 results in a subnormal value. If you divide by a large enough value, you lose all the digits and are left with 0. The C library now provides functions that let you check whether your computations are producing subnormal values.

There's another special floating-point value that can show up: `NaN`, or not-a-number. For example, you give the `asin()` function a value, and it returns the angle that has that value as its sine. But the value of a sine can't be greater than 1, so the function is undefined for values in excess of 1. In such cases, the function returns the `NaN` value, which `printf()` displays as `nan`.



Floating-Point Round-off Errors

Take a number, add 1 to it, and subtract the original number. What do you get? You get 1. A floating-point calculation, such as the following, may give another answer:

```
/* floaterr.c--demonstrates round-off error */
#include <stdio.h>
int main(void)
{
    float a,b;

    b = 2.0e20 + 1.0;
    a = b - 2.0e20;
    printf("%f \n", a);
    return 0;
}
```

The output is this:

```
0.000000      ←gcc, Linux
-13584010575872.000000    ←Turbo C 1.5
4008175468544.000000     ←CodeWarrior 5.0, MSVC++ 6.0
```

The reason for these odd results is that the computer doesn't keep track of enough decimal places to do the operation correctly. The number 2.0e20 is 2 followed by 20 zeros and, by adding 1, you are trying to change the 21st digit. To do this correctly, the program would need to be able to store a 21-digit number. A `float` number is typically just 6 or 7 digits scaled to bigger or smaller numbers with an exponent. The attempt is doomed. On the other hand, if you used 2.0e4 instead of 2.0e20, you would get the correct answer because you are trying to change the 5th digit, and `float` numbers are precise enough for that.

Complex and Imaginary Types

Many computations in science and engineering use complex and imaginary numbers. C99 supports these numbers, with some reservations. A free-standing implementation, such as that used for embedded processors, doesn't need to have these types. (A VCR chip probably doesn't need complex numbers to do its job.) Also, more generally, the imaginary types are optional.

In brief, there are three complex types, called `float _Complex`, `double _Complex`, and `long double _Complex`. A `float _Complex` variable, for example, would contain two `float` values, one representing the real part of a complex number and one representing the imaginary part. Similarly, there are three imaginary types, called `float _Imaginary`, `double _Imaginary`, and `long double _Imaginary`.

Including the `complex.h` header file lets you substitute the word `complex` for `_Complex`, the word `imaginary` for `_Imaginary`, and allows you to use the symbol `I` to represent the square root of -1 .

Other Types

That finishes the list of fundamental data types. For some of you, the list must seem long. Others of you might be thinking that more types are needed. What about a character string type? C doesn't have one, but it can still deal quite well with strings. You will take a first look at strings in Chapter 4.

C does have other types derived from the basic types. These types include arrays, pointers, structures, and unions. Although they are subject matter for later chapters, we have already smuggled some pointers into this chapter's examples. (A *pointer* points to the location of a variable or other data object. The `&` prefix used with the `scanf()` function creates a pointer telling `scanf()` where to place information.)



Summary: The Basic Data Types

Keywords:

The basic data types are set up using eleven keywords: `int`, `long`, `short`, `unsigned`, `char`, `float`, `double`, `signed`, `_Bool`, `_Complex`, and `_Imaginary`.

Signed Integers:

They can have positive or negative values.

`int`: The basic integer type for a given system. C guarantees at least 16 bits for `int`.

`short` or `short int`: The largest `short` integer is no larger than the largest `int` and may be smaller. C guarantees at least 16 bits for `short`.

`long` or `long int`: Can hold an integer at least as large as the largest `int` and possibly larger. C guarantees at least 32 bits for `long`.

`long long` or `long long int`: This type can hold an integer at least as large as the largest `long` and possibly larger. The `long long` type is least 64 bits.

Typically, `long` will be bigger than `short`, and `int` will be the same as one of the two. For example, DOS-based systems for the PC provide 16-bit `short` and `int` and 32-bit `long`, and Windows 95-based systems provide 16-bit `short` and 32-bit `int` and `long`.

You can, if you like, use the keyword `signed` with any of the signed types, making the fact that they are signed explicit.

Unsigned Integers:

They have zero or positive values only. This extends the range of the largest possible positive number. Use the keyword `unsigned` before the desired type: `unsigned int`, `unsigned long`, `unsigned short`. A lone `unsigned` is the same as `unsigned int`.

Characters:

They are typographic symbols such as `A`, `&`, and `+`. By definition, the `char` type uses 1 byte of memory to represent a character. Historically, this character byte has most often been 8 bits, but it can be 16 bits or larger, if needed to represent the base character set.

`char`: The keyword for this type. Some implementations use a signed `char`, but others use an unsigned `char`. C enables you to use the keywords `signed` and `unsigned` to specify which form you want.

Boolean:

Boolean values represent `true` and `false`; C uses `1` for `true` and `0` for `false`.

`_Bool`: The keyword for this type. It is an unsigned int and need only be large enough to accommodate the range 0 through 1.

Real Floating Point:

They can have positive or negative values.

`float`: The basic floating-point type for the system; it can represent at least six significant figures accurately.

`double`: A (possibly) larger unit for holding floating-point numbers. It may allow more significant figures (at least 10, typically more) and perhaps larger exponents than `float`.

`long double`: A (possibly) even larger unit for holding floating-point numbers. It may allow more significant figures and perhaps larger exponents than `double`.

Complex and Imaginary Floating Point:

The imaginary types are optional. The real and imaginary components are based on the corresponding real types:

`float _Complex`

`double _Complex`

`long double _Complex`

`float _Imaginary`

`double _Imaginary`

`long double _Imaginary`

**Summary: How to Declare a Simple Variable**

1. Choose the type you need.
2. Choose a name for the variable using the allowed characters.
3. Use the following format for a declaration statement:

```
type-specifier variable-name;
```

The type-specifier is formed from one or more of the type keywords; here are examples of declarations:

```
int erest;
unsigned short cash;
```

4. You can declare more than one variable of the same type by separating the variable names with commas, for example,

```
char ch, init, ans;
```

5. You can initialize a variable in a declaration statement:

```
float mass = 6.0E24;
```

Type Sizes

Tables 3.3 and 3.4 show type sizes for some common C environments. (In some environments, you have a choice.) What is your system like? Try running the program in Listing 3.8 to find out.

TABLE 3.3 Integer Type Sizes (Bits) for Representative Systems

Type	Macintosh Metrowerks CW (default)	Linux on a PC	IBM PC Windows 98 and Windows NT	ANSI C Minimum
char	8	8	8	8
int	32	32	32	16
short	16	16	16	16
long	32	32	32	32
long long	64	64	64	64

Table 3.4 Floating-point Facts for Representative Systems

Type	Macintosh Metrowerks CW (default)	Linux on a PC	IBM PC Windows 98 and Windows NT	ANSI C Minimum
float	6 digits	6 digits	6 digits	6 digits
	−37 to 38	−37 to 38	−37 to 38	−37 to 37
double	18 digits	15 digits	15 digits	10 digits
	−4931 to 4932	−307 to 308	−307 to 308	−37 to 37
long double	18 digits	18 digits	18 digits	10 digits
	−4931 to 4932	−4931 to 4932	−4931 to 4932	−37 to 37

For each type, the top row is the number of significant digits and the second row is the exponent range (base 10).

LISTING 3.8 The `typesize.c` Program

```

/* typesize.c--prints out type sizes */
#include <stdio.h>
int main(void)
{
    printf("Type int has a size of %u bytes.\n", sizeof(int));
    printf("Type char has a size of %u bytes.\n", sizeof(char));
    printf("Type long has a size of %u bytes.\n", sizeof(long));
    printf("Type double has a size of %u bytes.\n",
           sizeof(double));
    return 0;
}

```

C has a built-in operator called `sizeof` that gives sizes in bytes. (Some compilers require `%lu` instead of `%u` for printing `sizeof` quantities. That's because C leaves some latitude as to the actual unsigned integer type that `sizeof` uses to report its findings. C99 provides a `%z` specifier for this type, and you should use it if your compiler supports it.) The output from Listing 3.8 is as follows:

```

Type int has a size of 4 bytes.
Type char has a size of 1 bytes.
Type long has a size of 4 bytes.
Type double has a size of 8 bytes.

```

This program found the size of only four types, but you can easily modify it to find the size of any other type that interests you. Note that the size of `char` is necessarily 1 byte because C defines the size of one byte in terms of `char`. So, on a system with a 16-bit `char` and a 64-bit `double`, `sizeof` will report `double` as having a size of 4 bytes. You can check the `limits.h` and `float.h` header files for more detailed information on type limits. (The next chapter discusses these two files further.)

Incidentally, notice in the last line how the `printf()` statement is spread over two lines. You can do this as long as the break does not occur in the quoted section or in the middle of a word.

Using Data Types

When you develop a program, note the variables you need and which type they should be. Most likely, you can use `int` or possibly `float` for the numbers and `char` for the characters. Declare them at the beginning of the function that uses them. Choose a name for the variable that suggests its meaning. When you initialize a variable, match the constant type to the variable type.

```

int apples = 3;           /* RIGHT      */
int oranges = 3.0;        /* POOR FORM */

```

C is more forgiving about type mismatches than, say, Pascal. C compilers allow the second initialization, but they might complain, particularly if you have activated a higher warning level. It is best not to develop sloppy habits.

When you initialize a variable of one numeric type to a value of a different type, C converts the value to match the variable. This means you may lose some data. For example, consider the following initializations:

```
int cost = 12.99;           /* initializing an int to a double */
float pi = 3.1415926536;    /* initializing a float to a double */
```

The first declaration assigns 12 to `cost`; when converting floating-point values to integers, C simply throws away the decimal part (*truncation*), instead of rounding. The second declaration loses some precision, because a `float` is guaranteed to represent only the first six digits accurately. Compilers may issue a warning (but don't have to) if you make such initializations. You might have run into this when compiling Listing 3.1.

Many programmers and organizations have systematic conventions for assigning variable names in which the name indicates the type of variable. For example, you could use an `i_` prefix to indicate type `int` and `us_` to indicate `unsigned short`, so `i_smart` would be instantly recognizable as a type `int` variable and `us_verysmart` would be an `unsigned short` variable.

Arguments and Pitfalls

It's worth repeating and amplifying a caution made earlier in this chapter about using `printf()`. The items of information passed to a function, as you may recall, are termed *arguments*. For instance, the function call `printf("Hello, pal.")` has one argument, "Hello, pal.". A series of characters in quotes, such as "Hello, pal.", is called a *string*. We'll discuss strings in Chapter 4. For now, the important point is that one string, even one containing several words and punctuation marks, counts as one argument.

Similarly, the function call `scanf("%d", &weight)` has two arguments, "%d" and `&weight`. C uses commas to separate arguments to a function. The `printf()` and `scanf()` functions are unusual in that they aren't limited to a particular number of arguments. For example, we've used calls to `printf()` with one, two, and even three arguments. For a program to work properly, it needs to know how many arguments there are. The `printf()` and `scanf()` functions use the first argument to indicate how many additional arguments are coming. The trick is that each format specification in the initial string indicates an additional argument. For instance, the following statement has two format specifiers, `%d` and `%d`:

```
printf("%d cats ate %d cans of tuna\n", cats, cans);
```

This tells the program to expect two more arguments, and indeed, there are two more—`cats` and `cans`.

Your responsibility as a programmer is to make sure that the number of format specifications matches the number of additional arguments and that the specifier type matches the value type. C now has a function-prototyping mechanism that checks if a function call has the correct number and correct kind of arguments, but it doesn't work with `printf()` and `scanf()` because they take a variable number of arguments. What happens if you don't live up to the programmer's burden? Suppose, for example, you write a program like that in Listing 3.9.

LISTING 3.9 The `badcount.c` Program

```

/* badcount.c--incorrect argument counts */
#include <stdio.h>
int main(void)
{
    int f = 4;
    int g = 5;
    float h = 5.0f;

    printf("%d\n", f, g);    /* too many arguments */
    printf("%d %d\n",f);    /* too few arguments */
    printf("%d %f\n", h, g); /* wrong kind of values */
    return 0;
}

```

Here's the output from Microsoft Visual C++ 6.0 (Windows 98 PC):

```

4
4 0
0 0.000000

```

Next, here's the output from Linux on a PC:

```

4
4 5
0 0.000000

```

And the following here's the output from Metrowerks CodeWarrior Pro 5 (Macintosh):

```

4
4 2
1075052544 0.000000

```

Note that using `%d` to display a `float` value doesn't convert the `float` value to the nearest `int`; instead, it displays what appears to be garbage. Similarly, using `%f` to display an `int` value doesn't convert an integer value to a floating-point value. Also, the results you get for too few arguments or the wrong kind of argument differ from platform to platform.

None of the compilers we tried raised any objections to this code. Nor were there any complaints when we ran the program. As you can see, the computer doesn't catch this kind of error during runtime, and because the program may otherwise run correctly, you might not notice the errors, either. If a program doesn't print the expected number of values or if it prints unexpected values, check to see whether you've used the correct number of `printf()` arguments. (Incidentally, the UNIX syntax-checking program `lint`, which is much pickier than the UNIX compiler, does mention erroneous `printf()` arguments.)

One More Example

Let's run one more printing example, one that makes use of some of C's special escape characters. In particular, the program in Listing 3.10 shows how backspace (`\b`), tab (`\t`), and carriage return (`\r`) work. These concepts date from when computers used teletype machines for

output and they don't always translate successfully to contemporary graphical interfaces. For example, this listing doesn't work as described on some Macintosh implementations.

LISTING 3.10 The escape.c Program

```

/* escape.c - uses escape characters */
#include <stdio.h>
int main(void)
{
    float salary;
    printf("\aEnter your desired monthly salary:"); /* 1 */
    printf(" $_____ \b\b\b\b\b\b\b\b");          /* 2 */
    scanf("%f", &salary);
    printf("\n\t$%.2f a month is $%.2f a year.", salary,
           salary * 12.0);                          /* 3 */
    printf("\rGee!\n");                             /* 4 */
    return 0;
}

```

What Happens

Let's walk through this program step by step as it would work under an ANSI C implementation. The first `printf()` statement (the one numbered 1) sounds the alert signal (prompted by the `\a`), then prints the following:

Enter your desired monthly salary:

Because there is no `\n` at the end of the string, the cursor is left positioned after the colon.

The second `printf()` statement picks up where the first one stops, so after it is finished, the screen looks as follows:

Enter your desired monthly salary: \$_____

The space between the colon and the dollar sign is there because the string in the second `printf()` statement starts with a space. The effect of the seven backspace characters is to move the cursor seven positions to the left. This backs the cursor over the seven underline characters, placing the cursor directly after the dollar sign. Usually, backspacing does not erase the characters that are backed over, but some implementations may use destructive backspacing, negating the point of this little exercise.

At this point, you type your response, say **2000.00**. Now the line looks like this:

Enter your desired monthly salary: \$2000.00

The characters you type replace the underlined characters, and when you press Enter (or Return) to enter your response, the cursor moves to the beginning of the next line.

The third `printf()` statement output begins with `\n\t`. The newline character moves the cursor to the beginning of the next line. The tab character moves the cursor to the next tab stop on that line, typically, but not necessarily, to column 9. Then the rest of the string is printed. After this statement, the screen looks like this:

```
Enter your desired monthly salary: $2000.00
      $2000.00 a month is $24000.00 a year.
```

Because the `printf()` statement doesn't use the newline character, the cursor remains just after the final period.

The fourth `printf()` statement begins with `\r`. This positions the cursor at the beginning of the current line. Then **Gee!** is displayed there, and the `\n` moves the cursor to the next line. The final appearance of the screen is this:

```
Enter your desired monthly salary: $2000.00
Gee!      $2000.00 a month is $24000.00 a year.
```

Flushing the Output

When does `printf()` actually send output to the screen? Initially, `printf()` statements send output to an intermediate storage area called a *buffer*. Every now and then, the material in the buffer is sent to the screen. The standard C rules for when output is sent from the buffer to the screen are clear: It is sent when the buffer gets full, when a newline character is encountered, or when there is impending input. (Sending the output from the buffer to the screen or file is called *flushing the buffer*.) For instance, the first two `printf()` statements don't fill the buffer and don't contain a newline, but they are immediately followed by a `scanf()` statement asking for input. That forces the `printf()` output to be sent to the screen.

You may encounter an older implementation for which `scanf()` doesn't force a flush, which would result in the program looking for your input without having yet displayed the prompt onscreen. In that case, you can use a newline character to flush the buffer. The code can be changed to look like this:

```
printf("Enter your desired monthly salary:\n");
scanf("%f", &salary);
```

This code works whether or not impending input flushes the buffer. However, it also puts the cursor on the next line, preventing you from entering data on the same line as the prompting string. Another solution is to use the `fflush()` function described in Chapter 13, "File Input/Output."

Key Concepts

C has an amazing number of numeric types. This reflects the intent of C to avoid putting obstacles in the path of the programmer. Instead of mandating, say, that one kind of integer is enough, C tries to give the programmer the options of choosing a particular variety (signed or unsigned) and size that best meets the needs of a particular program.

Floating-point numbers are fundamentally different from integers on a computer. They are stored and processed differently. Two 32-bit memory units could hold identical bit patterns, but if one were interpreted as a `float` and the other as a `long`, they would represent totally different and unrelated values. For example, on a PC, if you take the bit pattern that represents

the `float` number 256.0 and interpret it as a `long` value, you get 113246208. C does allow you to write an expression with mixed data types, but it will make automatic conversions so that the actual calculation uses just one data type.

In computer memory, characters are represented by a numeric code. The ASCII code is the most common in the US, but C supports the use of other codes. A character constant is the symbolic representation for the numeric code used on a computer system—it consists of a character enclosed in single quotes, such as `'A'`.

Summary

C has a variety of data types. The basic types fall into two categories: integer types and floating-point types. The two distinguishing features for integer types are the amount of storage allotted to a type and whether it is signed or unsigned. The smallest integer type is `char`, which can be either signed or unsigned, depending on the implementation. You can use `signed char` and `unsigned char` to explicitly specify which you want, but that's usually done when you are using the type to hold small integers rather than character codes. The other integer types include `short`, `int`, `long`, and `long long` type. C guarantees that each of these types is at least as large as the preceding type. Each of them is a signed type, but you can use the `unsigned` keyword to create the corresponding unsigned types: `unsigned short`, `unsigned int`, `unsigned long`, and `unsigned long long`. Or you can add the `signed` modifier to explicitly state that the type is signed. Finally, there is the `_Bool` type, an unsigned type able to hold the values 0 and 1, representing `false` and `true`.

The three floating-point types are `float`, `double`, and, new with ANSI C, `long double`. Each is at least as large as the preceding type. Optionally, an implementation can support complex and imaginary types by using the keywords `_Complex` and `_Imaginary` in conjunction with the floating-type keywords. For example, there would be a `double _Complex` type and a `float _Imaginary` type.

Integers can be expressed in decimal, octal, or hexadecimal form. A leading 0 indicates an octal number, and a leading 0x or 0X indicates a hexadecimal number. For example, 32, 040, and 0x20 are decimal, octal, and hexadecimal representations of the same value. An l or L suffix indicates a `long` value, and an ll or LL indicates a `long long` value.

Character constants are represented by placing the character in single quotes: `'Q'`, `'8'`, and `'$'`, for example. C escape sequences, such as `'\n'`, represent certain nonprinting characters. You can use the form `'\007'` to represent a character by its ASCII code.

Floating-point numbers can be written with a fixed decimal point, as in 9393.912, or in exponential notation, as in 7.38E10.

The `printf()` function enables you to print various types of values by using conversion specifiers, which, in their simplest form, consist of a percent sign and a letter indicating the type, as in `%d` or `%f`.

Review Questions

You'll find answers to the review questions in Appendix A, "Answers to the Review Questions."

- 1. Which data type would you use for each of the following kinds of data?
 - a. The population of East Simpleton
 - b. The cost of a movie on DVD
 - c. The most common letter in this chapter
 - d. The number of times that the letter occurs in this chapter
- 2. Why would you use a type `long` variable instead of type `int`?
- 3. What portable types might you use to get a 32-bit signed integer, and what would the rationale be for each choice?
- 4. Identify the type and meaning, if any, of each of the following constants:
 - a. `'\b'`
 - b. `1066`
 - c. `99.44`
 - d. `0XAA`
 - e. `2.0e30`
- 5. Dottie Cawm has concocted an error-laden program. Help her find the mistakes.

```
include <stdio.h>
main
(
    float g; h;
    float tax, rate;

    g = e21;
    tax = rate*g;
)
```

- 6. Identify the data type (as used in declaration statements) and the `printf()` format specifier for each of the following constants:

Constant	Type	Specifier
a.	12	
b.	0X3	
c.	'C'	
d.	2.34E07	

Constant	Type	Specifier
e.	'\040'	
f.	7.0	
g.	6L	
h.	6.0f	

7. Identify the data type (as used in declaration statements) and the `printf()` format specifier for each of the following constants (assume a 16-bit `int`):

Constant	Type	Specifier
a.	012	
b.	2.9e05L	
c.	's'	
d.	100000	
e.	'\n'	
f.	20.0f	
g.	0x44	

8. Suppose a program begins with these declarations:

```
int imate = 2;
long shot = 53456;
char grade = 'A';
float log = 2.71828;
```

Fill in the proper type specifiers in the following `printf()` statements:

```
printf("The odds against the %__ were %__ to 1.\n", imate, shot);
printf("A score of %__ is not an %__ grade.\n", log, grade);
```

9. Suppose that `ch` is a type `char` variable. Show how to assign the carriage-return character to `ch` by using an escape sequence, a decimal value, an octal character constant, and a hex character constant. (Assume ASCII code values.)
10. Correct this silly program. (The `/` in C means division.)

```
void main(int) / this program is perfect /
{
    cows, legs integer;
    printf("How many cow legs did you count?\n");
```

```
scanf("%c", legs);
cows = legs / 4;
printf("That implies there are %f cows.\n", cows)
}
```

11. Identify what each of the following escape sequences represents:
 - a. `\n`
 - b. `\\`
 - c. `\"`
 - d. `\t`

Programming Exercises

1. Find out what your system does with integer overflow, floating-point overflow, and floating-point underflow by using the experimental approach; that is, write programs having these problems.
2. Write a program that asks you to enter an ASCII code value, such as 66, and then prints the character having that ASCII code.
3. Write a program that sounds an alert and then prints the following text:
Startled by the sudden sound, Sally shouted, "By the Great Pumpkin, what was that!"
4. Write a program that reads in a floating-point number and prints it first in decimal-point notation and then in exponential notation. Have the output use the following format (the actual number of digits displayed for the exponent depends upon the system):
The input is 21.290000 or 2.129000e+001.
5. There are approximately 3.156×10^7 seconds in a year. Write a program that requests your age in years and then displays the equivalent number of seconds.
6. The mass of a single molecule of water is about 3.0×10^{-23} grams. A quart of water is about 950 grams. Write a program that requests an amount of water, in quarts, and displays the number of water molecules in that amount.
7. There are 2.54 centimeters to the inch. Write a program that asks you to enter your height in inches and then displays your height in centimeters. Or, if you prefer, ask for the height in centimeters and convert that to inches.

INDEX

Symbols

- \Ooo (escape character), 66-67
- , (unary operator), 785
- (decrement operator), 144, 147-149, 159
- ~ (tilde)
 - bitwise unary operator, 789
 - unary operator, bitwise logical operator, 592-593
- = (assignment operator), 132-133, 158, 181-182, 786
- = (assignment operator), 192-195, 786
- == (equal to relational operator), 171, 177, 181-182, 185, 785
- 0 flag (printf() function), 107
- ! (logical NOT operator), 237, 787
- != (not equal to relational operator), 177, 185, 785
- π (pi) constant, 95
- ‘ ‘ (single quotation marks), 66, 85
- _ (underscore)), naming variables, 31
- [] (brackets)
 - arrays, 91, 346
 - declaration modifier, 571-572
 - pointers, 378
- { } (curly braces), 26, 35, 357, 404, 634
 - blocks, 131, 455-456
 - functions, 29
 - if else statements, 233
 - initializers, 531
 - statements, 29
 - structure member declarations, 529
 - while loops, 131
- , (comma), 37, 404
 - initializers, 531
 - operator, 790
 - variables, 56, 339
- “ “ (double quotation marks), 66, 91, 404, 628-629
 - files, 327, 473
 - macros, 619
 - printf() function, 33
- ... (ellipsis), 197, 404, 658
 - parameters, 818
 - variadic macro, 626-627
- (hyphen)
 - arithmetic operator, 785
 - minus sign operator, 134
 - subtraction operator, 134, 159, 785
 - unary operator, 159
- () (parentheses)
 - arguments, 621, 628
 - declaration modifier, 571-572
 - definitions, 628
 - functions, 25, 28, 33, 304
 - operator precedence, 139, 141, 148
 - pointers, 378
 - sizeof operator, 95
 - subexpressions, 225
- ;(semicolon), 35-36, 597, 786, 795
 - assignment statement, 32
 - functions, 304
 - statements, 151, 155
 - structure member declarations, 529
- ? (conditional operator), 244-246, 260, 787
- ? (question mark), 66
- ^ (caret)
 - EXCLUSIVE OR (bitwise binary operator), 789
 - EXCLUSIVE operator (bitwise logical operator), 593-594
- ^= (assignment operator), 786
- . (dot) operator, 532, 541, 563-564
- . (membership operator), 788-789
- . (period), 226, 236
- # (pound sign), 27, 159, 268-270, 616, 640
- < > (angle brackets), 628
- < and > cautionary note, floating-point comparisons, 177

>
 greater than relational operator, 148, 177, 185
 redirection operator, 276-277

>>
 bitwise binary operator, 227, 790
 right shift operator, 597

>= (greater than or equal to relational operator), 177, 185

-> (indirect membership operator), 541, 563-565

< (less than)
 operator, 148
 relational operator, 177, 185, 275, 277, 785
 symbol, 131, 629

<<
 bitwise binary operator, 789
 left shift operator, 596

<= (less than or equal to relational operator), 177, 185, 785

<<= (assignment operator), 786

+ (addition operator), 134, 158, 785

+= (assignment operator), 192-195, 786

++
 arithmetic operator, 785
 unary operator, 366
 increment operator, 144-150, 159

| (OR)
 bitwise binary operator, 789
 bitwise logical operator, 593

| (pipe) operator, 277

|= (assignment operator), 786

|| (logical OR operator), 237, 787

% (percent sign)
 arithmetic operator, 785
 printing, 104
 modulus operator, 142-144, 159

%= (assignment operator), 192-195, 786

%% (conversion specifier), 102

%% (strftime function() format specifier), 842

& (ampersand), 51, 69, 77
 address operator, 335, 368, 540
 pointer-related operator, 788
 unary operator, finding addresses, 330-332

&& (AND operator)
 bitwise binary operator, 789
 bitwise logical operator, 593

&= (assignment operator), 786

&&& (logical AND operator), 237, 787

/ (forward slash, 226

/ (division operator), 137-138, 159, 785

/* (comment symbol), 26-29

/= (assignment operator), 192-195, 786

// symbol, 29

\ (backslash), 34, 66-67, 616

\ " (escape character), 66-67

\ ' (escape character), 66-67

\ ? (escape character), 66

\\ (escape character), 66-67

* (asterisk)
 conversion modifier, 118
 declaration modifier, 571-572
 indirection operator, 334-335, 361
 modifier, 121-123
 multiplication operator, 37, 135-136, 159, 785
 operator, 360, 368
 pointer-related operator, 788
 pointers, 336, 378
 unary operator, 366

*/ (comment symbol), 26

*/ symbol, 26, 28-29

*= assignment operator, 192-195, 786

A

\a (escape character), 66

"a" mode string, 498

%A (conversion specifier), 101, 118

%A (strftime function() format specifier), 840

"a+" mode string, 498

"a+b" mode string, 498

"ab" mode string, 498

"ab+" mode string, 498

abstract data types. *See* ADTs

access
 random, array elements, 708
 sequential, linked list elements, 708

accessing
 C library, 643-644
 files randomly, 506-510, 518-520
 structure members, 532-533
 structure pointer members, 541

active positions (characters), 66

actual arguments, 160, 309-310, 340

add one.c program, code, 144

addaword.c program, code, 503-504

addemup.c program, code, 152

AddItem() function, 687, 716, 719

addition operator (+), 134, 158, 785

AddNode() function, 717

- addresses
 - address operator (&), 335, 368
 - arrays, 358-361
 - byte addressable, 360
 - finding, & (unary operator), 330-332
 - head pointers, 670
 - pointers, 330, 334-338, 375
 - structures, 540, 543-544
 - transmitting, 337
 - variables, 339
- Adelson-Velskii (AVL trees), 735
- ADTs (abstract data types), 666
 - binary search trees, 713
 - data types, defining abstract to concrete, 677
 - defining, 677
 - integer properties, 676
 - interfaces, 678-688
 - lists, 677-678
 - queues, 689-708
- advice booths in malls, simulating, 702-708
- algorithms, 666
 - recursion, calculating binary equivalents of integers, 323-324
 - selection sort, 435
 - sorting, 652
- Algorithms in C: Fundamentals, Data Structures, Sorting, Searching*, 783
- allocating
 - memory, 475-481, 530, 793
 - structures, comparing in blocks or individually, 669
- alphabetically sorting strings, 432-434
- alphanumeric to integer (atoi() function), 440-441
- altering variables in calling functions, 332-334
- alternate spellings for logical operators, 237-238
- alternative buffers, creating, 512-513
- altnames.c program, code, 71
- American National Standards Institute. *See* ANSI
- ampersand (&), 51, 69, 77
- analyses
 - characters, 226-227
 - checking.c program, 288-289
 - lethead1.c program, 303-305
- anatomy of a C program, 24
- and (&&) logical operator, 237, 787
- AND (&)
 - bitwise binary operator, 789
 - bitwise logical operator, 593
- angle brackets (<>), 628-629
- animals.c program, code, 251-252
- ANSI (American National Standards Institute) C, 18
 - buffered input, 270
 - C library, 643-645
 - C standard, 18
 - const type qualifier, 482-484
 - conversion specifiers, scanf() function, 118
 - files, 494-495, 513
 - fseek() function, portability, 510
 - ftell() function, portability, 510
 - functions
 - arguments, 317-318
 - buffering, 270
 - defining, 339
 - prototyping, 314-318
 - string conversions, 441
 - general utilities library, 648-654
 - I/O functions, 824-827
 - library
 - assert.h header file, 801
 - C99 additions, 801-811
 - complex.h header file, 801-803
 - ctype.h header file, 803-804
 - errno.h header file, 804
 - fcntl.h header file, 805-807
 - inttypes.h header file, 807-808
 - locale.h header file, 808-811
 - stdlib.h header file, 827-833
 - string functions, 417, 430-432
 - string.h header file, 834-837
 - tgmth.h header file, 837-838
 - time.h header file, 838-842
 - Unix I/O functions, 267
 - wchar.h header file, 842-849
 - wctype.h header file, 849-852
 - math functions (standard), 812-816
 - math library, 645-648
 - preprocessor, 615
 - qualifiers, keywords, 486-487
 - restrict type qualifier, 485-486
 - setbuf() function, 270
 - setvbuf() function, 270
 - standard library, 800-811
 - stdio.h header file, 268
 - stdlib.h header file, function prototypes, 648
 - string.h file, 418
 - type qualifiers, 481-482
 - volatile type qualifier, 484-485
- append() function, 518
- append.c program code, 516-517
- appending text files, 498

- arf.c program, code, 372-373
- argc (argument count), 439
- argument values (argv), pointer arrays, 439
- arguments, 81
 - () (parentheses), 621, 628
 - actual, 160, 309-310, 340
 - ANSI C functions, 317-318
 - argc (argument count), 439
 - argv (argument values), 439
 - arrays, argv, 439
 - command-line, 437-440, 497
 - data types, 81-82, 157
 - declaration list, 339
 - #define preprocessor directive, 621-625
 - FILE pointer, 508
 - float, conversion specifications, 106
 - formal, 160, 308
 - fseek() function, 508
 - function-like macros, 621-625
 - functions, 33, 159, 304-307
 - calling, 340
 - defining, 308
 - pointers, 573
 - pound.c program, code, 159-161
 - prototyping, 308-309
 - int, argc (argument count), 439
 - macros, 624-62
 - main() function, 438
 - malloc() function, 475
 - mode, 508
 - offset, 508
 - parameters, comparing, 160
 - passing, 113
 - pfun, 681
 - pointers, 364-366
 - printf() function, 33, 103
 - strings, 415, 437-440, 624-625
 - structure pointers and structure arguments, comparing, 548-549
 - structures, passing as, 544
 - type void functions, 650
 - variables
 - macros, 819
 - stdarg.h header file, 818-824, 658-660
- argv (argument values), pointer arrays, 439
- arithmetic operators, 132, 139, 785
 - = assignment, 133
 - / division, 137-138, 159
 - % modulus, 159
 - * multiplication, 135-136, 159
 - + addition, 134, 158
 - ++ increment, 159
 - decrement, 159
 - subtraction, 134, 159
 - unary, 159
 - binary, 134
 - dyadic, 134
 - exponential growth, 136-137
 - expression trees, 139
 - operator precedence, 138-141
- array2d.c program, 381-383
- arrays
 - [] (brackets), 91, 571-572
 - addresses, 358-361
 - array of, 377
 - binary searches, ordered list elements, 710
 - char, 91, 204, 397
 - characters
 - character pointers and character arrays, comparing, 549-550
 - strings, 90, 204, 404-405
 - compound literals, 387-389
 - concepts, 210
 - constants, 358
 - content protection, const keyword, 370-375
 - copying, mems.c program code, 656-657
 - creating, 476
 - declaring, 345-346
 - designated initializers (C99), 350-351
 - elements
 - indexes, 346
 - inserting, 708
 - random access, 708
 - subscript numbers, 346
 - flexible array members (C99), structures, 554-556
 - for loops, 204-206
 - functions, 361-364, 383
 - gsort() function, 573
 - indices, 204, 352-353
 - initializing, 346-351
 - int, 204
 - linked lists, comparing, 708-710
 - loops, 203-204
 - malloc() function, 476
 - multidimensional, 354-358, 375-383
 - numbers, 204
 - offsets, 204
 - one-dimensional, 358, 390
 - parameters, declaring, 363
 - pointers, 358-367, 375

- program modularity, 206
- qsort() function, 650-653
- queues, 691-694
- ragged, 404
- rectangular, 404
- sizes, 347, 353-354
- storage classes, 348
- storage units, 359
- strings, 91
- of structures, functions, 533-537, 556-557
- subscripts, 204
- talkback.c program, 90
- three-dimensional, 358
- two-dimensional, 355-358
- values, 351-352
- variable-length dynamic memory allocation, 480-481
- variables, comparing declarations, 91
- VLAs (variable-length arrays), 354
 - dynamic memory allocation, 387
 - sizes, 384
 - vararr2d.c program code, 385-386
- Art of Computer Programming (The)*, Volume 1, 783
- ASCII code, 64, 85
- .asm file extension, 17
- assert library, 654-656
- assert.c program code, 655
- assert.h header file, 654-656, 801
- assigned values, 566-567
- assignment operation, pointers, 368
- assignment operators, 132, 158, 181-182, 192-195, 786
- assignment statements, 32, 35, 152, 157, 795
- association rule (operators), 140
- asterisk (*)
 - modifier, 121
 - multiplication, 37
 - pointers, 336, 378
- atan() function, 646
- atan2() function, 646
- atexit() function, 648-650
- atoi() function, 440-441
- auto keyword, 453
- auto specifier, 464
- automated reseeding, 471
- automatic access to C library, 643
- automatic storage class, 453, 488, 793
- automatic storage duration, variables, 452-455
- AVL (Adel'son-Vel'skii) trees, binary search trees, 735

B

- /b (backspace character), 34
- \b (escape character), 66-67, 82
- %B (strftime function() format specifier), 840
- backslash (\), 34, 66-67, 616
- backspace character (/b), 34
- badlimits() function, 285, 288
- balanced binary search trees, 735
- base 2 number system (binary), 588
- base 8 number system (octal), 590-591
- base 10 number system (decimal), 588, 591
- base 16 number system (hexadecimal), 591-592
- basenames of files, 11
- bases.c program, code, 59
- Bell Labs, 1
- benefits to programmers, 3-4
- beta() function, 467
- binary code, integer 7, 54
- binary data, writing to files, 514-515
- binary equivalents of integers, calculating, 323-324
- binary exponents, 590
- binary floating-point numbers, 589-590
- binary form, storing data, 513
- binary fractions, 590
- binary I/O, 513, 518-520
- binary integers, 588
- binary mode and text mode, comparing, 509
- binary numbers, 587-591
- binary operators, 134, 784, 789-790
- binary output, 513
- binary search trees, 711
 - ADTs (abstract data types), 713
 - AVL trees, 735
 - balanced, 735
 - emptying, 725
 - implementing, 716-730
 - interfaces, 713-716
 - items, managing, 716-724, 735
 - Nerfville Pet Club, 735
 - nodes, deleting, 722-723
 - petclub.c program, 731-735
 - programming package, 725-729
 - stringy, 735
 - subtrees, 712, 721-722
 - traversing, 724-725
 - tree.c implementation file program code, 725-729
 - unbalanced, 735
 - words, storing, 712

- binary searches, ordered list elements, 710
- binary tree structures, 561
- binary views (files), 494
- binary.c program, code, 323-324, 598-599
- bit 0 (low-order bit), 588
- bit 7 (high-order bit), 588
- bitmapped graphic images, 666
- bits, 53, 587
 - accessing, 611
 - fields, 601-611
 - high-order (bit 7), 588
 - low-order (bit 0), 588
 - numbers, 588
 - positions, bit fields, 611
 - toggling, 595-596
 - turning on or off, 595
 - values, 588, 596
- bitwise logical operators, 789-790
 - & (AND operator), 593
 - ^ (EXCLUSIVE operator), 593-594
 - | (OR operator), 593
 - ~ (unary operator), 592-593
- binary.c program, code, 598-599
- bit fields, 606-611
- bits, toggling, 595-596
- invert4.c program, code, 600-601
- logical, 593
- masks, 594-595
- programming example, 598-599
- shift operators, 598
- bitwise shift operators, 596-598
- black box viewpoint, functions, 310
- block scope, variables, 450-451, 457-459
- blocks
 - compound statements, 131, 154-156, 454-456, 795
 - statements in functions, { } (curly braces), 29
- body
 - #define preprocessor directive lines, 617
 - functions, 29, 35
- book.c program, 528-529
- books. *See also* reference sources
 - Algorithms in C: Fundamentals, Data Structures, Sorting, Searching*, 783
 - Art of Computer Programming (The)*, Volume 1, 783
 - C: A Reference Manual*, Fourth Edition, 783
 - C Programming FAQs*, 783
 - C Programming Language (The)*, Second Edition, 782
 - C Puzzle Book (The)*, 782
 - C Traps and Pitfalls*, 783
 - C++ Primer Plus*, Fourth Edition, 784
 - C++ Programming Language (The)*, Third Edition, 784
 - conventions, 20-21
 - Elements of Programming Style (The)*, 783
 - International C Standard (The)* (ISO/IEC 9899 1999), 783
 - inventory, 527-529
 - organization, 19-20
 - Reliable Data Structures in C*, 783
 - Standard C Library (The)*, 783
- booksave.c program, 558-561
- Bool keyword, 53, 78, 791
- bool macro, 819
- bool true false are defined macro, 819
- Bool types, 70, 182
- Boole, George, 182
- Boolean support, stdbool.h header file, 819
- Boolean types
 - C and C+, comparing, 868
 - C99, 791
- Boolean values, 78
- Boolean variables, 182
- boolean.c program, code, 182-183
- bore() function, 452
- Borland C/C++, 533
- bottles.c program, code, 147-148
- bounds, array indices, 352-353
- bounds.c program code, 352
- braces, curly ({}), 26, 35, 357, 404, 634
 - blocks, 131, 455-456
 - functions, 29
 - if else statements, 233
 - initializers, 531
 - structure member declarations, 529
 - while loops, 131
- brackets
 - < > (angle), 628-629
 - [], 91, 346, 378
- branching statements. *See* if statements
- break command, 259, 799
- break keyword, 799
- break statements, 246, 249-250, 254-255
- break.c program, code, 249-250
- broken-down time, 838
- buffers
 - alternative, creating, 512-513
 - flushing, 84, 512
 - functions, 270
 - I/O, 269-270, 279-281, 496
 - printf() statements, 84
 - sizes, 269

bugs, debugging programs, 10, 39-43, 654-656
 butler() function, 38-39
 byebye.c program code, 648-649
 byte addressable, 360
 bytes, 53, 295, 520, 587-588

C

C
 advantages, 4-5
 Borland C/C++, 533
 C: A Reference Manual, Fourth Edition, 783
 compilers, 3
 and C++, comparing, 864-869
 design features, 2
 efficiency, 3
 flexibility, 3
 function library, 94
 history, 1
 language standards, 18-19
 library, 643-645
 new elements, 51
 overview, 2
 portability, 3
 power, 3
 preprocessor, 90, 95-101
 programmer benefits, 3-4
 programming mechanics, 7-18
 purpose, 1
 shortcomings, 4
 simple C program, 23-25
 trends, 4-5
 %c conversion specifier, 102, 118
 .c file extension, 11, 16-17, 24, 327
 %c format specifier, 68
 C Programming FAQs, 783
 C Programming Language (The), 18
 C Programming Language (The), Second Edition, 782
 C Puzzle Book (The), 782
 %C specifier, scanf() function, 289
 %c (strftime function() format specifier), 840
 C Traps and Pitfalls, 783
 C++, 4, 533
 C++ and C, comparing, 864-869
 C++ Primer Plus, Fourth Edition, 784
 C++ Programming Language (The), Third Edition, 784
 c.type.h header file, 226-228
 C/C++ Users Journal, 781
 C90 standard, 18
 C99 standard, 19
 additions to ANSI C standard library, 801-811
 Boolean type, 791
 compilers, identifiers, 462
 complex numbers, 792
 compound literals, 552-553
 conversion specifiers, scanf() function, 118
 designated initializers, arrays, 350-351
 flexible array members, 554-556
 imaginary numbers, 792
 Math library, 811-818
 numeric computations, 860-864
 stdbool.h header file, Boolean support, 819
 talkback.c program, code, 89
 wctype.h header file, 849-852
 C9X committee (C99 standard), 19
 caching, 484
 calculations, fathm ft.c program, 37
 calendar time, 839
 calling functions, 33, 38, 161, 303-304
 arguments, 309, 340
 recursion, 318-325
 variables, 332-334
 calloc() function, memory allocation, 479-480
 capitalization, macro function names, 628
 case labels, break or switch statement, vowels.c program
 code, 254-255
 case sensitivity, naming variables, 31
 cast operator, 158-159
 category macros, 809
 cc command (UNIX), 326
 cc compiler (UNIX), 14
 ccommand() function, 278, 440
 centering text, 306
 central processing units (CPUs), 5-6
 char keyword, 77
 char * fgets(char * restrict, int, FILE * restrict), function,
 825
 char * gets(char *), function, 826
 char * setlocale(int category, const char * locale), func-
 tion, 808
 char * strerror(int errnum), function, 836
 char * tmpnam(char *), function, 826
 char * asctime(const struct tm *tmpt), function, 840
 char * ctime(const time_t *ptm) Wed Aug 11 10:48:24
 1999\n\n function, 840
 char * currency_symbol macro, 809
 char * decimal_point macro, 809
 char * getenv(const char * name) function, 831
 char * grouping macro, 809
 char * int_curr_symbol macro, 809

- char *mon_decimal_point macro, 809
- char *mon_grouping macro, 810
- char *mon_thousands_sep macro, 809
- char *negative_sign macro, 810
- char *positive_sign macro, 810
- char *strcat(char * restrict s1, const char * restrict s2) function, 834
- char *strcat(char * s1, const char * s2) function, 431
- char *strchr(const char * s, int c) function, 431
- char *strchr(const char *s, int c) function, 836
- char *strcpy(char * restrict s1, const char * restrict s2) function, 835
- char *strcpy(char * s1, const char * s2) function, 430
- char *strncat(char * restrict s1, const char * restrict s2, size_t n) function, 834
- char *strncat(char * s1, const char * s2, size_t n) function, 431
- char *strncpy(char * restrict s1, const char * restrict s2, size_t n) function, 835
- char *strncpy(char * s1, const char * s2, size_t n) function, 430
- char *strpbrk(const char * s1, const char * s2) function, 431
- char *strpbrk(const char *s1, const char *s2) function, 836
- char *strrchr(const char * s, int c) function, 431
- char *strrchr(const char *s, int c) function, 836
- char *strstr(const char * s1, const char * s2) function, 431
- char *strstr(const char *s1, const char *s2), function, 836
- char *strtok(char * restrict s1, const char * restrict s2), function, 836
- char *thousands_sep macro, 809
- char arrays, 91, 204, 397
- char frac_digits macro, 810
- char int_frac_digits macro, 810
- char int_n_cs_precedes macro, 810
- char int_n_sep_by_space macro, 811
- char int_n_sign_posn macro, 811
- char int_p_cs_precedes macro, 810
- char int_p_sep_by_space macro, 810
- char int_p_sign_posn macro, 811
- char keyword, 53, 791
- char n_cs_precedes macro, 810
- char n_sep_by_space macro, 810
- char n_sign_posn macro, 810
- char p_cs_precedes macro, 810
- char p_sep_by_space macro, 810
- char p_sign_posn macro, 810
- char types, 64-69, 865-866
- characters, 77
 - \ " (escape character), 66-67
 - \ ' (escape character), 66-67
 - \ 000 (escape character), 66-67
 - \ ? (escape character), 66
 - \\ (escape character), 66-67
 - \ a (escape character), 66
 - active positions, 66
 - analyzing, cypher2.c program code, 226-227
 - arrays, loops or strings, 204
 - /b (backspace character), 34
 - \b (escape character), 66-67, 82
 - collating sequences, 423
 - concepts, 124
 - constants, 65, 85
 - #define statement, 98
 - digraphs, 857
 - echoed input, 270
 - escape, 34, 66-68, 82-84
 - evaluating, functions, 227-228
 - extensible wide-character classification functions, 850-851
 - \f (escape character), 66-67
 - format strings, 120-121
 - functions, strings, 435-437
 - getc() function, 499
 - handling, 803-804
 - input, 281-284, 292-295
 - I/O, standard, getc() or putc() function, 499
 - keywords, 791
 - mapping, 227, 616
 - multibyte, 858-860
 - \n (escape character), 66-67
 - /n (newline), 33
 - newline, 271, 432, 616
 - nonprinting, 65-68
 - null, 91-92, 418, 859
 - numeric code, 85
 - printing, 68-69
 - punctuation, counting, 436
 - pushing back to input stream, 512
 - putc() function, 499
 - \r (escape character), 66-67, 82
 - reading first of a line, 254
 - returning to strings, 431
 - scanf() function, specifiers, 289
 - sets, sizes of bytes, 588
 - single-character I/O, 268
 - spelling, 857-858
 - strings, 90-95, 397-405
 - supporting, 856

- /t (tab character), 34
- \t (escape character), 66-67, 82
- testing, functions, 227
- trigraph sequences, 856
- UCNs (universal character names), 858-859
- \v (escape character), 66-67
- whitespace, `scanf( )` function, 124
- wide, 859-860, 868
- wide-characters, 843-852
- word-count programs, 240-244
- \xhh (escape character), 66-67
- `charcode.c` program, code, 68-69
- `chcount.c` program, code, 236
- `checking.c` program, 286-289
- child nodes deleting, 721-722
- circular queues, 692
- classes, storage, 792-793
 - arrays, 348
 - auto specifier, 464
 - automatic, 453-456, 488, 793
 - duration, structure initialization, 531
 - extern specifier, 465
 - files of code, 464
 - functions, 467-468
 - `parta.c` file program, code, 465-466
 - `partb.c` file program, code, 466
 - register, 453, 488, 793
 - register specifier, 465
 - specifiers, 453, 464-465
 - static specifier, 465
 - static, 399, 453, 488, 793
 - `typedef` specifier, 464
 - variables, 449-467
- classifications, wide character classification utilities, 849-852
- `cleanup( )` function, `names3.c` program code, 551-552
- `clock_t` `clock(void)` function, 839
- `clock_t` type, 838
- closing files, `fclose( )` function, 500-501
- clubs (Nerfville Pet Club), binary search trees, 735
- `cmpflt.c` program, code, 177-178
- code
 - `add one.c` program, 144
 - `addaword.c` program, 503-504
 - `addemup.c` program, 152
 - `altnames.c` program, 71
 - `animals.c` program, 251-252
 - `append.c` program, 516-517
 - `arf.c` program, 372-373
 - `array2d.c` program, 381-382
 - ASCII, 64, 85
 - `assert.c` program, 655
 - `bases.c` program, 59
 - binary, integer 7, storing, 54
 - `binary.c` program, 323-324, 598-599
 - `book.c` program, 528
 - `booksave.c` program, 558-560
 - `boolean.c` program, 182-183
 - `bottles.c` program, 147-148
 - `bounds.c` program, 352
 - `break.c` program, 249-250
 - `byebye.c` program, 648-649
 - C source, 9
 - `charcode.c` program, 68-69
 - `chcount.c` program, 236
 - `checking.c` program, 286-288
 - `cmpflt.c` program, 177-178
 - `colddays.c` program, 220-221
 - `compare.c` program, 421
 - `compack.c` program, 422
 - `concrete.c` program, 11
 - `convert.c` program, 157-158, 436, 646-647
 - `copy1.c` program, 425-426
 - `copy2.c` program, 427
 - `copy3.c` program, 428-429
 - `count.c` program, 496-497
 - `cube.c` program, 188
 - `cypher1.c` program, 224
 - `cypher2.c` program, 226-227
 - `day mon1.c` program, 346-347
 - `day mon2.c` program, 349
 - `day mon3.c` program, 361
 - `defines.c` program, 100
 - `designate.c` program, 350
 - `diceroll.c` file program, 472-473
 - `diceroll.h` file program, 473
 - `divide.c` program, 137-138
 - `divisors.c` program, 234-235
 - `doubincl.c` program, 637
 - `dowhile.c` program, 198
 - `dual.c` program, 607-609
 - `dyn arr.c` program, 477
 - EBCDIC, 64
 - `echo eof.c` program, 272-273
 - `echo.c` program, 268, 438
 - `electric.c` program, 228-229
 - `entry.c` program, 198-199
 - `enum.c` program, 567-568
 - error messages, 50
 - `escape.c` program, 83-84
 - executable, 9
 - `factor.c` program, 321-322

fathm ft.c program, 36-37
 fields.c program, 604-605
 file eof.c program, 278
 files of, storage classes, 464
 films1.c program, 667
 films2.c program, 672-673
 films3.c program, 681-682
 flags.c program, 109
 flc.c program, 388-389
 flexmemb.c program, 554-556
 floatcnv.c program, 112-113
 floats.c program, 108-109
 forc99.c program, 455-456
 format.c program, 429-430
 friend.c program, 537-538
 friends.c program, 539-540
 func ptr.c program, 575-577
 funds1.c program, 542
 funds2.c program, 543
 funds3.c program, 544
 funds4.c program, 556-557
 global.c program, 461-462
 glue.c program, 625
 golf.c program, 133
 guess.c program, 279-280
 hello.c program, 440-441
 hiding.c program, 454-455
 hotel.c function support module, 328-329
 hotel.h header files, 329-330
 ifdef.c program, 634
 input.c program, 117, 423-424
 intconv.c program, 111-112
 invert4.c program, 600-601
 join chk.c program, 419-420
 lesser.c program, 310-311
 lethead1.c program, 303
 lethead2.c program, 306-307
 list.c implementation file program, 684-685
 list.h interface header file program, 680
 loccheck.c program, 331
 longstrg.c program, 115-116
 mac arg.c program, 621-622
 mall.c program, 704-706
 manybook.c program, 534
 manydice.c file program, 473-474
 mems.c program, 656-657
 menuette.c program, 293-295
 min sec.c program, 143-144
 misuse.c program, 315
 name1.c program, 407-408
 name2.c program, 408
 name3.c program, 409-410
 nameln2.c program, 551-553
 names.c source file program, 630
 names.h header file program, 629, 636
 names1.c program, 545-546
 names2.c program, 547
 no data.c program, 347-348
 nogo.c program, 420-421
 nogood.c program, 39
 nono.c program, 413
 numeric, characters, 85
 order.c program, 366
 p and s.c program, 405-406
 paint.c program, 244-245
 parrot.c program, 505
 parta.c file program, 465-466
 partb.c file program, 466
 petclub.c program, 731-734
 pizza.c program, 96-98
 pnt add.c program, 359
 post pre.c program, 146
 postage.c program, 193-194
 pound.c program, 159-161
 power.c program, 207-209
 praise1.c program, 92-93
 praise2.c program, 93-95
 predef.c program, 638-639
 preproc.c program, 617
 print1.c program, 57-58
 print2.c program, 62-63
 printout.c program, 102-104
 prntval.c program, 115
 proto1.c program, 316-317
 pseudocode, 172
 pt ops.c program, 367-368
 put out.c program, 412
 put put.c program, 416
 put1.c program, 414
 put2.c program, 415-416
 qsorter.c program, 651-652
 queue.c implementation file program, 698-700
 queue.h interface header file program, 694-695
 r drive1.c driver program, 469
 r drive2.c program, 470
 rain.c program, 355-356
 rand0.c function file program, 468
 randbin.c program, 518-519
 recur.c program, 318-319
 reducto.c program, 501-502
 reverse.c program, 507
 rhodium.c program, 49-50

- rows1.c program, 201-202
- rows2.c program, 202-203
- rules.c program, 140-141
- running.c program, 161-162
- s and r.c program, 469-470
- scan str.c program, 411
- scores.c program, 204-206
- shoe2.c program, 144-145
- shoes1.c program, 129-130
- shoes2.c program, 130-131
- showchar1.c program, 282
- showchar2.c program, 283
- showfpt.c program, 74-75
- simple C program, 23-25
- sizeof.c program, 142
- skip.c program, 246-247
- skip2.c program, 122-123
- somedata.c program, 348-349
- sort str.c program, 432-433
- source
 - compiler instructions, 639-640
 - converting to executable files, 12
 - preprocessing, 27
- squares.c program, 135-136
- starsch.c program, 424-425
- start-up, 12
- stillbad.c program, 41-42
- str cat.c program, 419
- strcnvt.c program, 442
- strings.c program, 109-110, 397-399
- subst.c program, 624
- sum arr1.c program, 363-364
- sum arr2.c program, 364-365
- summing.c program, 170
- swap1.c program, 332-333
- swap2.c program, 333
- swap3.c program, 336-337
- sweetie1.c program, 186
- sweetie2.c program, 187
- t and f.c program, 178-179
- talkback.c program, 89-91
- test.c program, 417-418
- tree.c implementation file, 725-729
- tree.h interface header file program, 714-716
- trouble.c program, 180-182
- truth.c program, 179-180
- two func.c program, 38-39
- typesize.c program, 79-80
- use qc program, 700-701
- useheader.c program, 630-631
- usehotel.c control module, 327-328
- varargs.c program, 659-660
- vararr2d.c program, 385-386
- variadic.c program, 626
- varwid.c program, 121-122
- vowels.c program, 254-255
- warnings, 50
- wheat.c program, 136-137
- when.c program, 174
- while1.c program, 175
- while2.c program, 175-176
- width.c program, 107-108
- wordcnt.c program, 242-244
- zeno.c program, 196-197
- zippo1.c program, 376
- zippo2.c program, 378
- colddays.c program, code, 220-221
- collating sequences, 423
- colors, bit fields, 604-611
- columns
 - creating, nested loops, 202
 - printing, fixed field widths, 123
- combined redirection, 277
- combining tokens (## operator), 625
- Comeau C/C++ compiler, 17
- comma (,), 37, 404
 - initializers, 531
 - operator, 193-197, 235, 790
 - variable declarations, 339
 - variables, separating, 56
- command-line arguments, 437-440, 497
- commands
 - break, 259, 799
 - cc (UNIX), 326
 - continue, 259, 800
 - goto, 260, 800
 - make (UNIX), 326
 - rewind(), 561
- commentary, programs, 10-11
- comments
 - */ symbol, 26, 28-29
 - /* symbol, 26, 28-29
 - // symbol, 29
 - function preconditions or postconditions, 680
 - redirection, 278
 - syntax, 26
 - text, breaking into sequences, 616
 - variables, qualifying, 793
- committees, C9X, C99 standard, 19
- compare.c program, code, 421

- comparing
 - arguments and parameters, 160
 - arrays and pointers, 401-402
 - binary and text modes, 509
 - buffered and unbuffered input, 269
 - C and C++, 864-869
 - character pointers and character arrays, 549-550
 - continue statements and break statements, 249
 - data display and storage, 69
 - floating-point, < and > cautionary note, 177
 - if statements and if else statements, 223
 - integer types and floating-point types, 54
 - linked lists and arrays, 708-710
 - macros and functions, 627-628
 - recursion and loops, 318
 - strings, 420-424, 431-434
 - strings and characters, 93
 - structure allocation in blocks or individually, 669
 - structure pointers and structure arguments, 548-549
 - variable and array declarations, 91
- comparison expressions (relational expressions), 173, 176-185, 785
- compack.c program, code, 422
- compilations, conditional, 633-638
- compile time substitution, 96
- compilers, 12
 - \ (backslash) and newline characters, 616
 - ; (semicolon), 36
 - array declarations, 345
 - Borland C/C++, floating-point values, 533
 - C, 3
 - C/C++, 17
 - C99 standard, identifiers, 462
 - cc (UNIX), 14
 - characters, mapping, 616
 - DOS
 - command-line, compiling functions, 326-327
 - IBM PCs, 17
 - echo.c file, 270
 - forward declarations, 210
 - functions, 304
 - gcc
 - GNU, 15
 - Linux, 326
 - high-level programming languages, 7
 - instructions, placing in source code, 639-640
 - Macintosh, 327
 - Metrowerks CodeWarrior, 17-18
 - preprocessing source code, 27
 - programs
 - concepts, 44
 - translations for preprocessing, 616
 - text, breaking into sequences, 616
 - UNIX, 3
- compiling
 - C source code, 9
 - error messages, 50
 - executable files, 12
 - functions, 326-327
 - programs
 - files, 270
 - preprocessor, 615
 - source code files, 326
 - on UNIX systems, 14
- complex floating points, 78
- Complex I macro, 801
- complex keyword, 53
- complex macro, 801
- complex numbers, 76, 792, 801-803, 863-864
- complex types, comparing C and C++, 868
- complex.h header file, 76, 801-803, 863-864
- components of computers, 5
- compound literals, 387-389, 552-553
- compound statements (blocks), 154-156, 795
- compression, lossless or lossy, 666
- computations, C99 numeric, 860-864
- computers, 5-6
- concatenating strings, 419-420, 431, 483, 626
- concrete data types, defining, 677
- concrete.c program, code, 11
- condensing files, 501-503
- conditional compilations, 633-638
- conditional expressions, 244-245
- conditional loops, 174-175
- conditional operator (?), 244-246, 260, 787
- conditions, testing, 260
- conio.h header file, 270
- const (char * restrict, const char * restrict, FILE * restrict) function, 825
- const keyword, 481-482, 793
 - arrays, initializing, 347
 - constants, 373-375
 - parameters, 371-373
 - strings, 432
- const modifier, 98, 866
- const type qualifier, 482-484
- constant expressions, WEOF, 850
- constant pi (π), 95

- constants, 51-52
 - arrays, 358
 - C preprocessor, 95-98
 - char type, comparing C and C++, 865-866
 - character strings, 92, 399-400
 - characters, 65, 85
 - compound literals, 387-389
 - const keyword, 373-375
 - const modifier, 98
 - #define statement, 98
 - defines.c program, code, 100
 - defining, 96, 823, 827-828
 - enum, 566
 - EXIT_FAILURE, 827
 - EXIT_SUCCESS, 828
 - extended integers, 855
 - floating-point, 73-74
 - int type, 57, 98
 - integers, 62, 565-566, 823-824
 - INTMAX_MAX, 823
 - INTMAX_MIN, 823
 - INTN_MAX, 823
 - INTN_MIN, 823
 - INTPTR_MAX, 823
 - INTPTR_MIN, 823
 - INT_FASTN_MAX, 823
 - INT_FASTN_MIN, 823
 - INT_LEASTN_MAX, 823
 - INT_LEASTN_MIN, 823
 - long, 62
 - long long, 62
 - manifest, 98-101, 616-621, 631
 - MB_CUR_MAX, 828
 - NULL, 827
 - numerical, 124
 - pointers, 359
 - PTRDIFF_MAX, 824
 - PTRDIFF_MIN, 824
 - RAND_MAX, 828
 - read-only values, 98
 - redefining, 620-621
 - SIG_ATOMIC_MAX, 824
 - SIG_ATOMIC_MIN, 824
 - SIZE_MAX, 824
 - strings, static storage class, 399
 - symbolic, 95-100, 347, 620
 - UINTMAX_MAX, 823
 - UINTN_MAX, 823
 - UINTPTR_MAX, 823
 - UINT_FASTN_MAX, 823
 - UINT_LEASTN_MAX, 823
 - WCHAR_MAX, 824
 - WCHAR_MIN, 824
 - WINT_MAX, 824
 - WINT_MIN, 824
 - writing with int types, 68
- continue command, 259, 800
- continue keyword, 799
- continue statements, 246-249
- contrasting arrays and pointers, 402-403
- control flow, lethead1.c program, 305
- control mode values, floating-point numbers, 861
- control statements, 795
- control strings, 103-104
- conventions
 - books, 20-21
 - functions, naming, 302
- conversions
 - data types, promotions, 156
 - errors, 113
 - floatcnv.c program, code, 112-113
 - format, integer types, 807-808
 - intconv.c program, code, 111-112
 - mismatched, 110-113
 - printf() function, 110-113
 - specifications
 - float arguments, 106
 - printf() function, 101-110
 - scanf() function, 117-119
 - talkback.c program, 90
 - types, 156-157
 - wide-character multibyte conversion functions, 846-849
- convert.c program, code, 157-158, 436, 646-647
- converting
 - numbers to strings, 440
 - strings to numbers, 440-443
- coordinates, polar or rectangular, 662
- copy1.c program, code, 425-426
- copy2.c program, code, 427
- copy3.c program, code, 428-429
- copying
 - arrays, 656-657
 - strings, 430
- CopyToNode() function, 686, 697
- count variable, 305
- count() function, 292-293
- count.c program, code, 496-497
- counting
 - loops, 186
 - punctuation characters, 436
 - words, 240-244

CPUs (central processing units), 5-6

creating

- arrays, 476
- book inventory, 527-529
- data forms, 561-562
- functions, 303
- linked lists, 674-675
- strings, 397-399
- text files, 498
- type names with typedef, 569-571
- user interfaces, 279-284

critic() function, 462

Ctrl+D keyboard shortcut, 276

Ctrl+Z keyboard shortcut, 271, 274, 276

cctype.h header file, 435-437, 803-804

cube.c program, code, 188

curly braces ({ }), 26, 35, 404

- blocks, 131, 455-456
- functions, 29
- if else statements, 233
- statements, 29
- while loops, 131

cypher1.c program, code, 224

cypher2.c program, code, 226-227

D

%d

- conversion specifier, 68, 102, 118, 289
- strftime function() format specifier, 840-841
- symbol group, 34

data

- bits, 53
- bytes, 53
- constants, 52
- data structures. *See* structures
- data-type keywords, 52-55
- displaying and storing, comparing, 69
- forms, creating, 561-562
- global, const type qualifier, 483-484
- hiding, 680, 689
- objects
 - lvalues, 133
 - side effects, sequence points, 153-154
- pointers, 573
- rhodium.c program, code, 49-50
- storing in binary form, 513
- unions, 562-565
- variables, 52
- words, 53-54

data representation, 736-737

- ADTs (abstract data types)
 - films3.c program code, 681-682
 - integer properties, 676
 - interfaces, 678-683, 688-689
 - list.c implementation file program, 684-688
 - list.h interface header file program code, 680
 - lists, 677-678
 - queues, 689-708
- algorithms, 666
- binary search trees, 711
 - ADTs (abstract data types), 713
 - AVL trees, 735
 - balanced, 735
 - emptying, 725
 - implementing, 716-730
 - interfaces, 713-716
 - items, managing, 716-724, 735
 - Nerfville Pet Club, 735
 - nodes, deleting, 722-723
 - petclub.c program, 731-735
 - programming package, 725-729
 - stringy, 735
 - subtrees, 712
 - traversing, 724-725
 - tree.c implementation file program code, 725-729
 - tree.h interface header file program code, 714-716
 - unbalanced, 735
 - words, storing, 712
- bitmapped graphics images, 666
- data hiding, 680, 689
- films1.c program, code, 667-668
- linked lists, 668-676
- linked lists and arrays, comparing, 708-710
- lossless compression, 666
- lossy compression, 666
- malloc() function, 668

data types, 790 data types. *See also* ADTs; arrays

- arguments, 81-82
- Bool types, 70
- Boolean
 - type (C99), 791
 - values, 78
- C9X changes, 854
- char types, 64-69
- characters, 77
- complex floating points, 78
- complex numbers, 76, 792
- defining abstract to concrete, 677

- double types, 72-73
- fastest minimum width types, 70
- float types, 72-73
- floating-points
 - constants, 73-74
 - numbers, 75-76, 791-792
 - types, sizes for systems, 79
 - values, `showfpt.c` program, code, 74-75
 - variables, declaring, 73
- imaginary floating points, 78
- imaginary numbers, 76, 792
- int, 29, 56-64, 68, 79
- `inttypes.h` header file, 70-71
- keywords, 52-55, 77, 790-791
- long double types, 72-73
- long long types, 60
- long types, 60
- minimum width types, 70
- pitfalls, 81-82
- pointer types, 77
- portable types, 70-71
- promotion, 156
- real floating points, 78
- short types, 60
- signed integers, 77
- `sizeof` operator, 80
- sizes, 79
- truncation, 81
- `typesize.c` program, code, 79-80
- unsigned, 60-61, 77
- variables, declaring, 30, 78, 792
- DATE macro, 638
- `dates, time.h` header file, 838-842
- `day mon1.c` program, code, 346-347
- `day mon2.c` program, code, 349
- `day mon3.c` program, code, 361
- debugging programs, 10, 39-43, 654-656
- decimal numbers (base 10 system), 588, 591
- decimal points
 - floating-point numbers, 85
 - numbers, keywords, 53
- declarations
 - arrays, 345-346, 363, 535, 571-572
 - char type variables, 64-65
 - defining, 463
 - external variables, 463, 632
 - floating-point variables, 73
 - functions, 39, 210, 313-317, 467, 571-579
 - identifiers, modifiers, 571-573
 - int type variables, 56
 - integer types, 60
 - modifiers, `()`, `[]`, `*`, 571-572
 - names, modifiers, 571-573
 - one-dimensional arrays, 390
 - parameters, `const` type qualifier, 482-483
 - `pf` pointer (`ToUpper()` function), 573
 - pointers, 335-336, 382-383, 573-579
 - referencing, 463
 - statements, 29-30, 35, 795
 - structure pointers, 540-541
 - structures, 529
 - variables and arrays, comparing, 91
 - variables, 29-32, 78, 339, 792
- decrement operator (`—`), 144, 147-149, 159
- decrementing a pointer operation, 369
- default values, 566
- `#define`
 - preprocessor directive, 616-625, 632-633
 - statement, 98
- `defines.c` program, code, 100
- defining
 - ADTs (abstract data types) queues, 690-691
 - ANSI C functions, 339
 - constants, 96, 823, 827-828
 - data types, abstract to concrete, 677
 - declarations, external variables, 463
 - external variables, 463
 - functions, 308, 339
 - int type variables, 57
 - Item type, 678
 - macros, 633, 843
 - `main()` function, 304
 - `mycomp()` function, 653-654
 - program objectives, 8
 - `starbar()` function, 304
 - strings, 399-400
 - structure variables, 530-531
 - types
 - `stdlib.h` header file, 827
 - `time.h` header file, 838
 - `wchar.h` header file, 842-843
 - union variables, 563
 - variables, statements, 26
- definitions
 - `()` (parentheses), 628
 - functions, 38, 303
 - hiding outer definitions (variables), 454
 - macros (function-like), 621
 - object-like macros, 617
 - projects, 327
 - removing, 632-633
 - `stddef.h` header file, 820

- structure template, header files, 631
 - type, 631, 676
- DeleteAll() function, 725
- DeleteAllNodes() function, 725
- DeleteItem() function, 719, 724
- DeleteNode() function, 724
- deleting
 - items from binary search trees, 720-724
 - definitions, 632-633
 - items from queues, 691, 697
- DeQueue() function, 698
- dereferencing operation, pointers, 368
- dereferencing operator, * (indirection operator), 334-335, 361
- dereferencing uninitialized pointers, 369
- descriptions
 - C library, 644-645
 - functions, 27, 34
- design features, 2
- designate.c program code, 350
- designated initializers, 350-351, 532
- designing programs, 8
- diagnostics, assert.h header file, 801
- dice, rolling, 471-475
- diceroll.c file program, code, 472-473
- diceroll.h file program, code, 473
- diceroll.h header file, roll_n_dice() function, 473
- differencing operation, pointers, 369
- digit(s) conversion modifier, 118
- .digit(s) modifier, printf() function, 105
- digraphs (characters), 857
- direct input, comparing unbuffered input and buffered input, 269
- directives
 - #define preprocessor, 616-627
 - #elif, conditional compilations, 637-638
 - #else, conditional compilations, 633-635
 - #endif, conditional compilations, 633-635
 - #error, 639
 - #if, conditional compilations, 637-638
 - #ifdef, conditional compilations, 633-635
 - #ifndef, conditional compilations, 635-637
 - #include "/usr/biff/p.h", 628
 - #include "hot.h", 628
 - #include "mystuff.h", 628
 - #include <stdio.h>, 628
 - #include preprocessor, 628-632
 - #line, 639
 - #pragma, 639-640
 - #undef preprocessor, 632-633
 - include, 631
 - preprocessor, 27, 632-639
- displaying
 - data, compared with storing, 69
 - hexadecimal numbers, 58-59
 - linked lists, 673
 - octal numbers, 5859
- divide.c program, code, 137-138
- division operator (/), 137-138, 159, 785
- divisors.c program, code, 234-235
- div_t div(int numer, int denom), function, 832
- div_t type, 827
- do keyword, 797
- do while loops, 198-200
- do while statements, 200, 797
- doble indirection, pointer addresses, 375
- documentation, 27-29, 37. *See also* reference sources
- DOS
 - > (redirection operator), 276-277
 - >> operator, 277
 - < (redirection operator), 275-277
 - | (pipe) operator, 277
 - compilers
 - command-line, compiling functions, 326-327
 - IBM PCs, 17
 - Ctrl+Z keyboard shortcut, 276
 - input, redirecting, 275-276, 279
- dot (.)
 - membership operator, 788-789
 - operator, 541, 563-564
 - structure member operator, 532
- doubincl.c program, code, 637
- double acos(double x) function, 645, 813
- double asin(double x) function, 645, 813
- double atan(double x) function, 645, 813
- double atan2(double y, double x) function, 645, 813
- double atof(const char * nptr) function, 828
- double cabs(double complex z) function, 803
- double carg(double complex z) function, 803
- double cbrt(double x) function, 814
- double ceil(double x) function, 646, 814
- double cimag(double complex z) function, 803
- double complex cacos(double complex z) function, 802
- double complex cacosh(double complex z) function, 802
- double complex casin(double complex z) function, 802
- double complex casinh(double complex z) function, 802
- double complex catan(double complex z) function, 802
- double complex catanh(double complex z) function, 802

- double complex `ccos(double complex z)` function, 802
- double complex `ccosh(double complex z)` function, 802
- double complex `cexp(double complex z)` functions, 802
- double complex `clog(double complex z)` function, 803
- double complex `conj(double complex z)` function, 803
- double complex `cpows(double complex z, double complex y)` function, 803
- double complex `cproj(double complex z)` function, 803
- double complex `csin(double complex z)` function, 802
- double complex `csinh(double complex z)` function, 802
- double complex `csqrt(double complex z)` function, 803
- double complex `ctan(double complex z)` function, 802
- double complex `ctanh(double complex z)` function, 802
- double complex function, 803
- double complex type, 76
- double `copysign(double x, double y)` function, 815
- double `cos(double x)` function, 645, 813
- double `cosh(double x)` function, 813
- double `creal(double complex z)` function, 803
- double `difftime(time_t t1, time_t t0)` function, 839
- double `erf(double x)` function, 814
- double `erfc(double x)` function, 814
- double `exp(double x)` function, 646, 813
- double `exp2(double x)` function, 813
- double `expm1(double x)` function, 813
- double `fabs(double x)` function, 646, 814
- double `fdim(double x, double y)` function, 816
- double `floor(double x)` function, 646, 814
- double `fma(double x, double y, double z)` function, 816
- double `fmax(double x, double y)` function, 816
- double `fmin(double x, double y)` function, 816
- double `frexp(double v, int *pt_e)` function, 813
- double `hypot(double x, double y)` function, 814
- double keyword, 53
- double `ldexp(double x, int p)` function, 814
- double `lgamma(double x)` function, 814
- double `log(double x)` function, 814
- double `log(double x)` function, 646
- double `log10(double x)` function, 646, 814
- double `log2(double x)` function, 814
- double `logb(double x)` function, 814
- double `logp1(double x)` function, 814
- double `modf(double x, double *p)` function, 814
- double `nan(const char *tagp)` function, 815
- double `nearbyint(double x)` function, 814
- double `nextafter(double x, double y)` function, 816
- double `nexttoward(double x, long double y)` function, 816
- double `pow(double x, double y)` function, 646, 814
- double quotation marks (“ ”), 66, 91, 404, 628-629
 - files, 473
 - include files, 327
 - macros, 619
 - `printf( )` function, 33
- double recursion, 325
- double remainder(double x, double y) function, 815
- double `remquo(double x, double y, int *quo)` function, 815
- double `rint(double x)` function, 815
- double `round(double x)` function, 815
- double `scalbln(double x, long n)` function, 814
- double `scalbn(double x, int n)` function, 814
- double `sin(double x)` function, 645, 813
- double `sinh(double x)` function, 813
- double `sqrt(double x)` function, 646, 814
- double `strtod(char * restrict npt, char ** restrict ept)` function, 828
- double `tan(double x)` function, 645, 813
- double `tanh(double x)` function, 813
- double `tgamma(double x)` function, 814
- double `trunc(double x)` function, 815
- double types, 72-73, 78, 792
- `dowhile.c` program, code, 198
- drivers
 - `r drive1.c` driver program, code, 469
 - `r drive2.c` program, code, 470
 - testing functions, 310
- `dual.c` program, code, 607-609
- duplicating items in binary search trees, 735
- duration of storage classes, structure initialization, 531
- dyadic operators, 134
- `dyn arr.c` program, code, 477
- dynamic memory allocation, 387, 480-481

E

- `%e`
 - conversion specifier, 102, 118
 - `strftime` function() format specifier, 841
- `eatline( )` function, 579
- EBCDIC code, 64
- `echo eof.c` program, code, 273-274
- `echo.c` program
 - code, 268, 438
 - stopping, 270
- echoing input, 268-270
- `eddyred` file, 502

- editing on UNIX systems, 13-14
- EDOM macro, 804
- efficiency, 3
- EIC (International Electrotechnical Commission), 858
- EILSEQ macro, 804
- electric.c program, code, 228-229
- elements
 - arrays, 346, 708
 - inserting into arrays, 708
 - inserting into linked lists, 709
 - linked lists, sequential access, 708-709
 - ordered lists, binary searches, 710
- Elements of Programming Style (The)*, 783
- `#elif` preprocessor directive, conditional compilations, 637-638
- ellipsis (...), 197, 404, 658
 - parameters, 818
 - variadic macro, 626-627
- else if statements, 228-230
- `#else` preprocessor directive, conditional compilations, 633-635
- else statements and if else statements, pairing, 231-232
- emptying
 - binary search trees, 725
 - queues, 698
- `EmptyTheList()` function, 688
- end of file. *See* EOF
- end recursion (tail recursion), 321-323
- `#endif` preprocessor directive, conditional compilations, 633-635
- `EnQueue()` function, 698
- entry errors, avoiding, 295
- entry-condition loops. *See* for loops; while loops
- entry.c program, code, 198-199
- enum constants, 566
- enum keyword, 565
- enum.c program, code, 567-568
- enumerated types, 565-569
- enumerations, C and C++, comparing, 867-868
- environments
 - IDEs (Integrated development environments), 10, 15-17, 629
 - integrated, command-line arguments, 439
 - programs, running, 10
- EOF (end of file), 499-500, 520
 - Ctrl+Z keyboard shortcut, 271
 - echo eof.c program, 272-274
 - file eof.c program, code, 278
 - `getchar()` function, 272
 - GUIs (graphical user interfaces), 274
 - `int feof(FILE *fp)` function, 515
 - `int ferror(FILE *fp)` function, 515
 - marking, 271-274
 - `scanf()` function, 272
- equal to (==) relational operator, 177, 185, 785
- equality of functions, 325-326
- equality operator (==), 171
- ERANGE macro, 804
- `errno.h` header file, 804
- `#error` preprocessor directive, 639
- errors
 - conversion, 113
 - debugging, 39-43
 - entry, avoiding, 295
 - messages, 50, 639
 - output, 495-496
 - reporting, `errno.h` header file, 804
 - round-off, floating-point numbers, 76
 - semantic, 41-42
 - syntax, 40-41
- escape characters, printing, 82-84
- escape sequences, 34, 66-68, 83
- escape.c program, code, 83-84
- evaluating
 - ADTs (abstract data types) interfaces, 688-689
 - characters, functions, 227-228
 - logical operators, order of, 238-239
- evaluation order (operator precedence), rules.c program
 - code, 140-141
- exact width types, 820-821, 853
- exceptions, floating-point numbers, 861
- EXCLUSIVE operator (^) (bitwise logical operator), 593-594
- EXCLUSIVE OR (^) (bitwise binary operator), 789
- executable code, 9
- executable files, 10-13
- executing. *See* implementing
- .EXE file extension, 17
- `exit()` function, 477, 497, 648-650
- exit-condition loops (do while loops), 198-200
- EXIT_FAILURE constant, 827
- EXIT_SUCCESS constant, 828
- exiting. *See* terminating
- expansions (macro), `#define` preprocessor directive lines, 617
- exponential growth, wheat.c program code, 136-137
- exponential notations, 72, 85
- exponentiating operator (`pow()` function), 132, 206
- exponents, binary, 590
- expressions, 795
 - comma operator, 194
 - concepts, 163

- conditional, 244-245
- constant, WEOF, 850
- full, 153
- if statements, 235
- logical, 787
- logical operators, testing conditions, 260
- operators, logical, 239
- relational, 173-185, 236-240, 786
- statements, 151-158
- subexpressions, 150, 225
- test, 222, 240-241, 248, 253
- trees, 139
- values, 150-151
- extended integer constants, 824
- extended integer types, 852-855
- extended multibyte utilities, wchar.h header file, 842-849
- extensible wide-character classification functions, 850-851
- extensions of files
 - .asm, 17
 - .c, 11, 16-17, 24, 327
 - .EXE, 17
 - .h, 327, 629
 - .o, 14, 326
 - .obj, 17, 326
 - .red, 501
 - .txt, 17
- extern keyword, 459, 463, 467, 793
- extern specifier, 465
- external linkage, 452-453, 459-461, 488
- external storage class (static variables), 457-464, 468-471
- external variables, 459-463, 632

F

- %f conversion specifier, 102, 118
- \f (escape character), 66-67
- %f specifier
 - printf() function, 51
 - scanf() function, 289
- %F (strftime function() format specifier), 841
- fabs() function, 177
- factor.c program, code, 321-322
- false macro, 819
- false values of expressions, 178-180
- false variables, Bool type, 182
- fastest minimum width types, 70, 821-822, 854

- fathm ft.c program, 36-38
- fclose() function, closing files, 500-501
- FE ALL EXCEPT macro, 806
- FE DFL ENV macro, 806
- FE DIVBYZERO macro, 805
- FE DOWNWARD macro, 806
- FE INEXACT macro, 805
- FE INVALID macro, 806
- FE OVERFLOW macro, 806
- FE TONEAREST macro, 806
- FE TOWARDZERO macro, 806
- FE UNDERFLOW macro, 806
- FE UPWARD macro, 806
- fenv t type, 805
- fenv.h header file, 805-807, 861-862
- Feuer, Alan R., 782
- fexcept t type, 805
- fflush() function, 84
- fgetpos() function, 510
- fgets() function, 504
 - newline characters, 432
 - parrot.c program code, 505-506
 - string input, name3.c program code, 409-410
- Fibonacci numbers, 325
- field widths, 123, 410
- fields
 - bit, 601-611
 - structure records, 557
 - structures, 529
- fields.c program, code, 604-605
- FIFO (first in, first out), queues, 690
- FILE * fopen(const char * restrict, const char * restrict) function, 825
- FILE * freopen(const char * restrict, char * restrict, FILE * restrict) function, 825
- FILE * tmpfile(void) function, 826
- file_eof.c program, code, 278
- FILE
 - macro, 638
 - pointer argument, 508
- files
 - “ “ (double quotation marks), 473
 - accessing, 520
 - ANSI C, binary or text view, 494
 - .asm extension, 17
 - assert.h header, 654-656, 801
 - basenames, 11
 - binary, 513-515
 - .c extension, 11, 16-17, 24, 327
 - c.type.h header, 226-228
 - code, storage classes, 464

- complex.h header, 76, 801-803, 863-864
- conio.h header, 270
- ctype.h header, 437, 803-804
- diceroll.c file program, code, 472-473
- diceroll.h file program, code, 473
- echo.c, 270
- eddy.red, 502
- EOF (end of file), 271-274, 499-500, 515, 520
- errno.h header, 804
- executable, 10-13
- .EXE extension, 17
- fclose() function, closing, 500-501
- fenv.h header, 805-807, 861-862
- fgetc() function, 504-506
- file-condensing programs, 501-503
- file inclusion, accessing C library, 643-644
- file scope, variables, 451
- float.h, constants, 98-101
- fprintf() function, addaword.c program code, 503-504
- fputs() function, 504-506
- fscanf() function, addaword.c program code, 503-504
- fseek() function, 506-510
- ftell() function, 506-510
- functions, 304, 326-327
- gets() function, 506
- .h extension, 327, 629
- header, 27, 327-331, 631-632, 815
- hotel.h header, code, 329-330
- I/O, 493-510, 518-520
- include, “ ” (double quotation marks), 327
- #include
 - preprocessor directive, 628-632
 - statement, 26
- input
 - buffered, 496
 - redirecting, 274-276, 279
- inttypes.h header, 70-71, 807-808
- iso646.h header, 857-858
- limits.h, constants, 98-101
- linkers, 12
- list.c, 683-688
- list.h, 680, 683, 688
- locale.h header, 808-811
- low-level I/O, 271
- manydice.c file program, code, 473-474
- math.h header, 811-818, 862-863
- names, resetting, 639
- naming conventions, 11
- .o extension, 14, 326
- .obj extension, 17, 326
- object code, 12-13
- opening, fopen() function, 498-499
- output
 - binary and text, 513
 - buffered, 49, 5126
 - naming, 502
 - redirecting, 274-279
- parta.c file program, code, 465-466
- partb.c file program, code, 466
- pointers, 498
- programs, 270
- queue.c implementation file program code, 698-700
- queue.h, 690, 694-695
- rand0.c function file program, code, 468
- random access, 506-510, 518-520
- .red extension, 501
- redirection, 493-494
- rewind() function, addaword.c program code, 503-504
- scope, variables, 451
- SEEK_CUR mode, 508
- SEEK_END mode, 508
- SEEK_SET mode, 508
- setjmp.h header, 817
- side effects, sequence points, 153-154
- signal.h header, 817-818
- source code, 327, 630-631
- standard, 271, 501
- stdarg.h header, variable arguments, 658-659, 818-824
- stdbool.h header, 819
- stddef.h header, 820
- stderr pointer, 501
- stdin pointer, 501
- stdint.h header, integer types, 820-824
- stdio.h, 24-27, 268, 824-827
- stdlib.h header, 648, 827-833
- stdout pointer, 501
- streams, 271
- string.h header, 94, 418, 834-837
- structure contents, saving, 557-561
- system header, IDEs (integrated development environments), 629
- text, 27, 5135
- tgmath.h header, 837-838
- time.h header, 838-842
- tree.c, 717, 725-729
- tree.h interface header file program code, 714-716
- .txt extension, 17

- wchar.h header, 842-849
- wctype.h header, 849-852
- films1.c program, code, 667
- films2.c program, code, 672-673
- films3.c program, code, 681-682
- finding
 - addresses, & (unary operator), 330-332
 - items, binary search trees, 719-720
- first in, first out (FIFO), queues, 690
- fit() function, test.c program code, 417-418
- fixed decimal points, floating-point numbers, 85
- fixed field widths, 123
- flags, 234
 - + flag (printf() function), 106
 - flag (printf() function), 105-106
 - # flag (printf() function), 107
 - printf() function, 106
 - status, floating-point numbers, 861
- flags.c program, code, 109
- flc.c program, code, 388-389
- flexibility of C, 3
- flexibility of for loops, 188-192
- flexible array members (C99), structures, 554-556
- flexmemb.c program code, 554-556
- float arguments, conversion specifications, 106
- float Complex type, 76
- float keyword, 53
- float strtol(const char * restrict npt, char ** restrict ept)
 - function, 828
- float types, 72-73, 78, 791
- float-point standard, IEC (IEC International Electrotechnical Committee), 861
- float.h files, constants, 98-101
- floatcnv.c program, code, 112-113
- floating-point numbers, 84, 792
 - binary, 589-590
 - comparisons, < and > cautionary note, 177
 - constants, 73-74
 - control mode values, 861
 - division, 137
 - double types, 72-73, 78, 792
 - environment, fenv.h header file, 805-807
 - exceptions, 861
 - exponential notations, 85
 - fixed decimal points, 85
 - float types, 72-73, 78, 791
 - long double types, 72-73, 78, 792
 - overflow, 75
 - pi (π) numbers, storing, 55
 - real numbers, 54
 - relational operators, 177
 - representing, 590
 - round-off errors, 76
 - status flags, 861
 - types, 53-54, 79
 - underflow, 75
 - values, 74-75++, 533
 - variables, 51, 73
- floats.c program, code, 108-109
- flow
 - control, lethead1.c program, 305
 - electric.c program, 229
 - programs, forms of, 169, 253
- flushing buffers, 84, 269, 512
- flushing output, escape.c program, 84
- fonts, book convention, 20
- fopen() function files, opening, 498-499
- for keyword, 796
- for loops, 192
 - arrays, 204-206
 - comma operator, 193-197, 235
 - cube.c program, code, 188
 - flexibility, 188-192
 - structure, 187
 - sweetie2.c program, code, 187
 - variations, 191
- for statements, 192, 796
- forc99.c program, code, 455-456
- formal arguments, 160, 308
- formal parameters, 160, 308-310, 340
- format conversions, integer types, 807-808
- format specifiers, strftime function(), 840-842
- format strings, characters, 120-121
- format.c program, code, 429-430
- formats, free-form, 36
- forms (data), creating, 561-562
- forward declarations, 210
- forward slash (/), 226
- fprintf() function, 503-504
- fputs() function, 413-414, 504-506
- FP_FAST_FMA macro, 812
- FP_FAST_FMAF macro, 812
- FP_FAST_FMAL macro, 812
- FP_ILOGB0 macro, 812
- FP_ILOGBNAN macro, 812
- FP_INFINITE macro, 812
- FP_NAN macro, 812
- FP_NORMAL macro, 812
- FP_SUBNORMAL macro, 812
- FP_ZERO macro, 812
- fractions, binary, 590
- fread() function, 513, 516-517

- free() function, 475-479, 675, 720, 793
- fseek-form formats, 36
- friend.c program, code, 537-538
- friends.c program code, 539-540
- fscanf() function, addaword.c program, code, 503-504
- fseek() function, 506-510
- fseek(file, 0L, SEEK_CUR) function, 510
- fseek(file, 0L, SEEK_END) function, 510
- fseek(file, 0L, SEEK_SET) function, 510
- fseek(file, ftell-pos, SEEK_SET) function, 510
- fsetpos() function, 510
- ftell() function, 506-510
- full expressions, 153
- fully buffered I/O, 269
- func ptr.c program, 575-579
- functions, 25, 301
 - , (comma), 339
 - { } (curly braces), 29
 - () (declaration modifier), 571-572
 - () (parentheses), 25, 33, 304
 - # (pound sign), 159
 - ; (semicolon), 304, 828-833
 - & (unary operator), finding addresses, 330-332
 - aan2(), 646
 - actual arguments, 309-310, 340
 - AddItem(), 687, 716, 719
 - AddNode(), 717
 - addresses
 - finding with & (unary operator), 330-332
 - transmitting, 337
 - ADTs (abstract data types) queue interfaces, 696-700
 - ANSI C
 - arguments, 317-318
 - defining, 339
 - I/O, 824-827
 - prototyping, 314-318
 - standard math functions, 645-646
 - string conversions, 441
 - append(), 518
 - arguments, 33, 159-161, 304-308, 317-318, 339-340
 - arrays
 - const keyword, 371-375
 - content protection, 370-375
 - multidimensional, 382-383
 - pointers, 361-367
 - of structures, 556-557
 - sum arr1.c program code, 363-364
 - VLAs (variable-length arrays), 383-387
 - atan(), 646
 - atexit(), 648-650
 - atoi(), 440-441
 - badlimits(), 288
 - bad_limits(), 285
 - beta(), 467
 - binary I/O, 513
 - binary.c program, code, 323-324
 - black box viewpoint, 310
 - blocks of statements, 29
 - body, 35
 - bore(), 452
 - buffering for IBM PC compatibles, 270
 - butler(), 38-39
 - C function library, 94
 - call statements, 35, 795
 - calling, 33, 304, 309, 318-325, 340
 - calloc(), memory allocation, 479-480
 - calls, 161, 303
 - ccommand(), 278, 440
 - char * fgets(char * restrict, int, FILE * restrict), 825
 - char * gets(char *), 826
 - char * setlocale(int category, const char * locale), 808
 - char * strerror(int errnum), 836
 - char * tmpnam(char *), 826
 - char * asctime(const struct tm *tmpt), 840
 - char * ctime(const time_t *ptm) Wed Aug 11 10:48:24 1999\n0, 840
 - char * getenv(const char * name), 831
 - char * strcat(char * restrict s1, const char * restrict s2), 834
 - char * strcat(char * s1, const char * s2), 431
 - char * strchr(const char * s, int c), 431
 - char * strchr(const char *s, int c), 836
 - char * strcpy(char * restrict s1, const char * restrict s2), 835
 - char * strcpy(char * s1, const char * s2), 430
 - char * strncat(char * restrict s1, const char * restrict s2, size_t n), 834
 - char * strncat(char * s1, const char * s2, size_t n), 431
 - char * strncpy(char * restrict s1, const char * restrict s2, size_t n), 835
 - char * strncpy(char * s1, const char * s2, size_t n), 430
 - char * strpbrk(const char * s1, const char * s2), 431
 - char * strpbrk(const char *s1, const char *s2), 836
 - char * strrchr(const char * s, int c), 431
 - char * strrchr(const char *s, int c), 836

- char *strstr(const char * s1, const char * s2), 431
- char *strstr(const char *s1, const char *s2), 836
- char *strtok(char * restrict s1, const char * restrict s2), 836
- characters
 - cypher2.c program, code, 226-227
 - handling, 803-804
 - in strings, 435-437
- cleanup(), names3.c program code, 551-552
- clock_t clock(void), 839
- compiling, 326-330
- complex numbers, 802-803
- concepts, 340-341
- const (char * restrict, const char * restrict, FILE * restrict), 825
- CopyToNode(), 686, 697
- count(), 292-293
- creating, 303
- critic(), 462
- ctype.h, macros, 437
- declaring, 210
 - by type, 313-314
 - extern keyword, 467
 - header files, 631
 - old style, 314-316
 - old style compared to ANSI C, 317
 - prototypes, 39
- definitions, 38, 303, 308
- DeleteAll(), 725
- DeleteAllNodes(), 725
- DeleteItem(), 719, 724
- DeleteNode(), 724
- DeQueue(), 698
- description documentation, 27
- descriptions, 34
- div_t div(int numer, int denom), 832
- double acos(double x), 645, 813
- double asin(double x), 645, 813
- double atan(double x), 645, 813
- double atan2(double y, double x), 645, 813
- double atof(const char * nptr), 828
- double cabs(double complex z), 803
- double carg(double complex z), 803
- double cbrt(double x), 814
- double ceil(double x), 646, 814
- double cimag(double complex z), 803
- double complex, 803
- double complex cacos(double complex z), 802
- double complex cacosh(double complex z), 802
- double complex casin(double complex z), 802
- double complex casinh(double complex z), 802
- double complex catan(double complex z), 802
- double complex catanh(double complex z), 802
- double complex ccos(double complex z), 802
- double complex ccosh(double complex z), 802
- double complex cexp(double complex z), 802
- double complex clog(double complex z), 803
- double complex conj(double complex z), 803
- double complex cpows(double complex z, double complex y), 803
- double complex cproj(double complex z), 803
- double complex csin(double complex z), 802
- double complex csinh(double complex z), 802
- double complex csqrt(double complex z), 803
- double complex ctan(double complex z), 802
- double complex ctanh(double complex z), 802
- double copysign(double x, double y), 815
- double cos(double x), 645, 813
- double cosh(double x), 813
- double creal(double complex z), 803
- double difftime(time_t t1, time_t t0), 839
- double erf(double x), 814
- double erfc(double x), 814
- double exp(double x), 646, 813
- double exp2(double x), 813
- double expm1(double x), 813
- double fabs(double x), 646, 814
- double fdim(double x, double y), 816
- double floor(double x), 646, 814
- double fma(double x, double y, double z), 816
- double fmax(double x, double y), 816
- double fmin(double x, double y), 816
- double frexp(double v, int *pt_e), 813
- double hypot(double x, double y), 814
- double ldexp(double x, int p), 814
- double lgamma(double x), 814
- double log(double x), 646, 814
- double log10(double x), 646, 814
- double log2(double x), 814
- double logb(double x), 814
- double logp1(double x), 814
- double modf(double x, double *p), 814
- double nan(const char *tagp), 815
- double nearbyint(double x), 814
- double nextafter(double x, double y), 816
- double nexttoward(double x, long double y), 816
- double pow(double x, double y), 646, 814
- double recursion, 325
- double remainder(double x, double y), 815
- double remquo(double x, double y, int *quo), 815
- double rint(double x), 815
- double round(double x), 815

double scalbln(double x, long n), 814
 double scalbn(double x, int n), 814
 double sin(double x), 645, 813
 double sinh(double x), 813
 double sqrt(double x), 646, 814
 double strtod(char * restrict npt, char ** restrict
 ept), 828
 double tan(double x), 645, 813
 double tanh(double x), 813
 double tgamma(double x), 814
 double trunc(double x), 815
 drivers, 310
 eatline(), 579
 employing, 210
 EmptyTheList(), 688
 EnQueue(), 698
 equality of, 325-326
 escape sequences, 34
 exit(), 477, 497, 648-650
 extensible wide-character classification, 850-851
 fabs(), 177
 factor.c program, code, 321-322
 fclose(), closing files, 500-501
 fflush(), 84
 fgetpos(), 510
 fgets(), 409-410, 432, 504-506
 Fibonacci numbers, 325
 FILE * fopen(const char * restrict, const char
 * restrict), 825
 FILE * freopen(const char * restrict, char * restrict,
 FILE * restrict), 825
 FILE * tmpfile(void), 826
 files, 304
 fit() test.c program, code, 417-418
 float strtod(const char * restrict npt, char ** restrict
 ept), 828
 fopen(), opening files, 498-499
 formal arguments, 308
 formal parameters, 308-310, 340
 forward declarations, 210
 fprintf(), addaword.c program code, 503-504
 fputs(), 413-414, 504-506
 fread(), 513, 516-517
 free(), 475-479, 675, 720, 793
 fscanf(), addaword.c program code, 503-504
 fseek(), 506-510
 fseek(file, 0L, SEEK_CUR), 510
 fseek(file, 0L, SEEK_END), 510
 fseek(file, 0L, SEEK_SET), 510
 fseek(file, ftell-pos, SEEK_SET), 510
 fsetpos(), 510
 ftell(), 506-510
 function calls, 38
 function statement, 152
 function-like macros, 617, 621-625
 fwrite(), 513, 516-517
 gamma(), 467
 general utilities, 652
 get choice(), menus, 291-292
 get int(), 288
 getc(), getting characters from files, 499
 getchar(), 16, 223-226, 268-272, 295
 getche(), 270
 getchoice(), menus, 291
 getinfo(), 546, 550
 gets(), 397-399, 407-409, 506
 get_first(), 292
 gobble(), 479
 greatest width integers, 807-808
 gsort(), arrays, 573
 headers, 34
 hotel.c function support module, code, 328-329
 hotel.h header files, code, 329-330
 I/O (input and output), 27, 267-268, 516-517
 identifiers, 30
 imax(), 315-317
 imaxdiv_t imaxdiv(intmax_t numer, intmax_t
 denom), 807
 imin(), 310-313
 InitializeList(), 680, 686
 inline, 640-643, 869
 InOrder(), 724
 int abs(int n), 832
 int atexit(void (*func)(void)), 830
 int atoi(const char * nptr), 828
 int atol(const char * nptr), 828
 int classify(real-floating x), 813
 int fclose(FILE *), 824
 int fegetround(void), 806
 int feholdexcept(fenv_t *envp), 807
 int feof(FILE *), 825
 int feof(FILE *fp), 515
 int ferror(FILE *), 825
 int ferror(FILE *fp), 515
 int fesetround(int round), 806
 int fetestexcept(int excepts), 806
 int fflush(FILE *), 825
 int fflush(FILE *fp), 512
 int fgetc(FILE *), 825
 int fgetpos(FILE * restrict, fpos_t * restrict), 825

- int fmod(double x, double y), 815
- int fprintf(FILE * restrict, const char * restrict, ...), 825
- int fputc(int, FILE *), 825
- int fputs(const char * restrict, FILE * restrict), 825
- int fscanf(FILE * restrict, const char * restrict, ...), 825
- int fseek(FILE *, long, int), 825
- int fsetpos(FILE *, const fpos_t *), 825
- int getc(FILE *), 825
- int getchar(), 826
- int ilogb(double x), 813
- int isalnum(int c), 803
- int isalpha(int c), 803
- int isblank(int c), 803
- int iscntrl(int c), 804
- int isdigit(int c), 804
- int isfin(real-floating x), 813
- int isfinite(real-floating x), 813
- int isgraph(int c), 804
- int isgreater(real-floating x, real-floating y), 816
- int isgreaterequal(real-floating x, real-floating y), 816
- int isless(real-floating x, real-floating y), 816
- int islessequal(real-floating x, real-floating y), 816
- int islessgreater(real-floating x, real-floating y), 816
- int islower(int c), 804
- int isnan(real-floating x), 813
- int isnormal(real-floating x), 813
- int isprint(int c), 804
- int ispunct(int c), 804
- int isspace(int c), 804
- int isunordered(real-floating x, real-floating y), 816
- int isupper(int c), 804
- int iswalnum(wint_t wc), 850
- int iswalpha(wint_t wc), 850
- int iswblank(wint_t wc), 850
- int iswcntrl(wint_t wc), 850
- int iswdigit(wint_t wc), 850
- int iswgraph(wint_t wc), 850
- int iswlower(wint_t wc), 850
- int iswprint(wint_t wc), 850
- int iswpunct(wint_t wc), 850
- int iswspace(wint_t wc), 850
- int iswupper(wint_t wc), 850
- int iswxdigit(wint_t wc), 850
- int isxdigit(int c), 804
- int mblen(const char *s, size_t n), 833
- int mbsinit(const mbstate_t *ps), 847
- int mbtowc(wchar_t *pw, const char, 833
- int memcmp(const void *s1, const void *s2, size_t n), 834
- int printf(const char * restrict, ...), 826
- int putc(int, FILE *), 826
- int putchar(int), 826
- int puts(const char *), 826
- int raise(int sig), 818
- int rand(void), 829
- int remove(const char *), 826
- int rename(const char *, const char *), 826
- int scanf(const char * restrict, ...), 826
- int setjump(jmp_buf env), 817
- int setvbuf(FILE * restrict, char * restrict, int, size_t), 826
- int setvbuf(FILE *fp, char *buf, int mode, size_t size), 512-513
- int signbit(real-floating x), 813
- int snprintf(char * restrict, size_t n, const char * restrict, ...), 826
- int sprintf(char * restrict, const char * restrict, ...), 826
- int sscanf(const char * restrict, const char * restrict, ...), 826
- int strcmp(const char * s1, const char * s2), 431
- int strcmp(const char *s1, const char *s2), 835
- int strcoll(const char *s1, const char *s2), 835
- int strlen(const char * s), 836
- int strncmp(const char * s1, const char * s2, size_t n), 431
- int strncmp(const char *s1, const char *s2, size_t n), 835
- int system(const char *str), 831
- int tolower(int c), 804
- int toupper(int c), 804
- int ungetc(int c, FILE *fp) function, 512
- int ungetc(int, FILE *), 826
- int vfprintf(FILE * restrict, const char * restrict, va_list), 826
- int vprintf(const char * restrict, va_list), 827
- int vsprintf(char * restrict, const char * restrict, va_list), 827
- int vsprintf(char * restrict, size_t n) const char * restrict, va_list), 827
- int wctob(wint_t c), 847
- int wctomb(char *s, wchar_t wc), 833
- interchange(), 332-334
- intmax_t imaxabs(intmax_t j), 807
- intmax_t strtointmax(const char * restrict nptr, char ** restrict endptr, int base), 808

- intmax_t wcstolmax(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base), 808
- InTree(), 719
- ioctl(), 270
- isalnum(), 227
- isalpha(), 226-227
- iscntrl(), 227
- isdigit(), 227
- isgraph(), 227
- islower(), 227, 240
- isprint(), 227
- ispunct(), 227, 436
- isspace(), 227, 241
- isupper(), 227
- ldiv_t ldiv(long numer, long denom), 832
- lesser.c program, code, 310-311
- lethead1.c program, analyzing, 303-305
- lethead2.c program, code, 306-307
- libraries, 802
- library routines, 9, 12
- ListIsEmpty(), 686
- ListIsFull(), 686
- ListItemCount(), 686
- lldiv_t lldiv(long numer, long denom), 832
- localization, 808
- loccheck.c program, code, 331
- long double strtols(const char * restrict npt, char ** restrict ept), 829
- long int lrint(double x), 815
- long int lround(double x), 815
- long labs(int n), 832
- long long int llrint(double x), 815
- long long int llround(double x), 815
- long long llabs(int n), 832
- long long strtoll(const char * restrict npt, char ** restrict ept, int base), 829
- long strtol(const char * restrict npt, char ** restrict ept, int base), 829
- macros, 627-628
- main(), 28, 288
 - arguments, 438
 - defining, 304
 - getinfo() function, 546
 - int, 25
 - int statement, 34
 - interchange() function, variable swapping, 332-334
 - return statements, 34
 - return types, 27
 - void, 25
- makeinfo(), 546-548
- MakeNode(), 717
- malloc(), 475-481, 550-552, 668-670, 793
- math, 815
- memcpy(), 486, 656-657
- memmove(), 486, 656-657
- misuse.c program, code, 315
- modules, 288
- multidimensional arrays, 380-383
- mycomp(), defining, 653-654
- names, 302, 578, 628
- operators, 301
- parameters, 82, 308, 340
- parentheses(), 28
- pointers, 330, 334-339, 573-575, 578-579
- postconditions, 680
- pound(), 159
- pow(), 132, 206
- power(), 209
- preconditions, 680
- printf(), 34, 51, 81-82, 85, 101, 124, 271, 301-302
 - “(double quotation marks), 33
 - * (modifier), 121-123
 - arguments, 33, 103
 - conversions, 101-113
 - .digit(s) modifier, 105
 - exercise, 46
 - %f specifier, 51
 - # flag, 107
 - + flag, 106
 - flag, 106
 - 0 flag, 107
 - flags, 105-106
 - flags.c program, code, 109
 - float arguments, conversion specifications, 106
 - floatcnv.c program, code, 112-113
 - floating-point values, printing, 74
 - floats.c program, code, 108-109
 - functions, calling, 33
 - h modifier, 105
 - hh modifier, 105
 - intcnv.c program, code, 111-112
 - interactive programs, 51
 - j modifier, 105
 - l modifier, 105-106
 - ll modifier, 105
 - long strings, printing, 115-116
 - longstrg.c program, code, 115-116
 - printout.c program, code, 102-104

- prntval.c program, code, 115
- return values, 114-115
- skip2.c program, code, 122-123
- space flag, 107
- specifiers, matching, 64
- string output, 414
- strings, converting to numbers, 440
- strings.c program, code, 109-110
- t modifier, 106
- %u specifier, 61
- unsigned int values, 61
- values, printing, 37-38
- varwid.c program, code, 121-122
- warnings, 84
- width.c program, code, 107-108
- z modifier, 106
- programs, adding, 38-39
- proto1.c program, code, 316-317
- prototyping with arguments, 308-309
- putc(), putting characters into files, 499
- putchar(), 223-226, 268, 271, 302
- puts(), 397-399, 412-413, 418
- qsort(), 650-654
- queues, 696-697
- r drive2.c program, code, 470
- raise(), 817
- rand(), 468-471, 475
- rand0() m r drive1.c driver program, code, 469
- rand1(), 475
- random numbers, 468-471
- recur.c program, code, 318-319
- recursion, 318-325
- return keyword, 301
- return types, 340
- return values, 206-210
- reviewing, 301-302
- rewind(), addaword.c program code, 503-504
- rollem(), 473
- roll_n_dice(), 473
- scanf, 118-121
- scanf(), 51, 81, 101, 116, 124, 271, 295
 - & (ampersand), 51, 77
 - * modifier, 121-123
 - %C specifier, 289
 - conversions, 117-119
 - %d specifier, 289
 - EOF (end of files), 272
 - %f specifier, 289
 - format string characters, 120-121
 - input, reading, 119-120
 - input.c program, code, 117
 - interactive programs, 51
 - loops, 171
 - null characters, 92
 - %s specifier, 289
 - skip2.c program, code, 122-123
 - string input, 410-412
 - summing.c program, 171-172
 - varwid.c program, code, 121-122
 - whitespaces, 93, 117, 124
- scope, variables, 451
- SeekItem(), 719
- SeekNode(), 724
- setbuf(), 270
- setjmp.h header file, 817
- setvbuf(), 270, 516
- show(), 575-579
- showinfo(), 548
- showmenu(), 579
- showmovies(), 687
- show_n_char(), 306-308
- sign off(), 649
- signal(), 817-818
- signals, 818
- size_t fread(void * restrict, size_t, size_t, FILE * restrict), 825
- size_t fread(void *ptr, size_t size, size_t nmemb, FILE *fp), 515
- size_t fwrite(const void * restrict, size_t, size_t, FILE * restrict), 825
- size_t fwrite(void *ptr, size_t size, size_t nmemb, FILE *fp), 514-515
- size_t mbrlen(const char * restrict s, size_t n, mbstate_t * restrict ps), 847
- size_t mbrtowc(wchar_t * restrict pwc, const char * restrict s, size_t n, mbstate_t * restrict ps), 847
- size_t mbsrtowcs(wchar_t * restrict dst, const char ** restrict src, size_t len, mbstate_t * restrict ps), 848
- size_t mbstowcs(wchar_t * restrict pwcs, const char *s restrict, size_t n), 833
- size_t strcspn(const char *s1, const char *s2), 836
- size_t strftime(char * restrict s, size_t max, const char * restrict fmt, const struct tm * restrict tmpt), 840
- size_t strlen(const char * s), 431
- size_t strspn(const char *s1, const char *s2), 836
- size_t strxfrm(char * restrict s1, const char * restrict s2, size_t n), 835
- size_t wctomb(char * restrict s, wchar_t wc, mbstate_t * restrict ps), 848

size_t wcsrtombs(char * restrict dst, const wchar_t
 ** restrict src, size_t len, mbstate_t * restrict ps),
 849
 size_t wstombs(char * restrict s, const wchar_t *
 restrict pwcs, size_t n), 833
 sprintf(), 429-430, 440
 sqrt(), 646
 srand1(), 469-470
 standard I/O, 271, 511-515, 518
 starbar(), 303-305
 statements, blocks, 29
 storage classes, 467-468
 strcat(), 419-420, 483
 strchr(), newline characters, 432
 strcmp(), 420-424, 432-434
 strcpy(), 425-427
 strftime(), format specifiers, 840-842
 strings, 414-432, 435-437, 834-836
 strlen(), 90-95, 301, 417-418
 strncat(), join chk.c program code, 419-420
 strncmp(), 420-425
 strncpy(), 425, 428-429
 strtod(), 441
 strtok(), 837
 strtol(), 441-443
 strtoul(), 441
 struct lconv * localeconv(void), 808
 struct tm * gmtime(const time_t * ptm), 840
 struct tm * localtime(const time_t * ptm), 840
 structures, 306
 addresses, 543-544
 compound literals (C99), 552-553
 flexible array members (C99), 554-556
 malloc() function, 550-552
 members, passing, 541-543
 names1.c program code, 545-546
 names2.c program code, 547
 passing as arguments, 544
 pointers, 550-552
 structure pointers and structure arguments,
 comparing, 548-549
 sum(), pointer arguments, 364-365
 sump(), 365
 sun squares(), 288
 sun(), sum arr1.c program code, 363-364
 swap1.c program, code, 332-333
 swap2.c program, code, 333
 swap3.c program, code, 336-337
 tail recursion, 321-323
 terminating, 313
 testing, 310
 time(), 471
 time.h header file, 839-840
 time_t mktime(struct tm * tmpr), 839
 time_t time(time_t * ptm), 839
 ToLeft(), 717-718
 tolower(), 227
 too bad(), 649
 ToRight(), 717-718
 toupper(), 228, 435-437
 ToUpper(), 573
 Traverse(), 681, 687, 724
 trystat(), 458
 two-dimensional, applying to arrays, 383
 type void, 650
 types, int or void, 304
 uintmax_t strtoumax(const char * restrict nptr,
 char ** restrict endptr, int base), 808
 uintmax_t wstoumax(const wchar_t * restrict
 nptr, wchar_t ** restrict endptr, int base), 808
 ungetc(), 512
 Unix I/O, 267
 unsigned long long strtoull(const char * restrict
 npt, char ** restrict ept, int base), 829
 unsigned long strtoul(const char * restrict npt, char
 ** restrict ept, int base), 829
 usehotel.c control module, code, 327-328
 valuelong ftell(FILE *), 825
 values, returning, 26, 310-313, 340
 variables
 altering, 332-334
 arguments, stdarg.h header file, 658-660
 count, 305
 local, 305
 private names, 160
 swapping, 332-334
 void (*signal(int sig, void (*func)(int)))(int), 818
 void *bsearch(const void *key, const void *base,
 size_t nmem, size_t size, int (*comp)
 (const void *, const void *)), 831
 void *calloc(size_t nmem, size_t size), 829
 void *malloc(size_t size), 830
 void *memchr(const void *s, int c, size_t n), 834
 void *memcpy(void * restrict s1, const void *
 restrict s2, size_t n), 834
 void *memmove(void *s1, const void *s2,
 size_t n), 834
 void *memset(void *s, int v, size_t n), 834
 void *realloc(void *ptr, size_t size), 830
 void abort(void), 830
 void clearer(FILE *), 824
 void exit(int status), 831

- void feclearexcept(int excepts), 806
- void fegetenv(fenv_t *envp), 806
- void fegetexceptflag(fexcept_t *flagp, int excepts), 806
- void feraiseexcept(int excepts), 806
- void fesetenv(const fenv_t *envp), 807
- void fesetexceptflag(const fexcept_t *flagp, int excepts), 806
- void feupdateenv(const fenv_t *envp), 807
- void free(void *ptr), 830
- void functions, 304
- void longjmp(jmp_buf env, int val), 817
- void perror(const char *), 826
- void qsort(void *base, size_t nmem, size_t size, int (*comp) (const void *, const void *)), 832
- void rewind(FILE *), 826
- void setbuf(FILE * restrict, char * restrict), 826
- void srand(unsigned int seed), 829
- void _Exit(int status), 831
- wide-characters, 843-852
- funds1.c program, code, 542
- funds2.c program code, 543
- funds3.c program, code, 544
- funds4.c program, code, 556-557
- fwrite() function, 513, 516-517

G

- %G conversion specifier, 102, 118
- %g (strftime function() format specifier), 841
- gamma() function, 467
- gcc compiler
 - GNU, 15
 - Linux, 326
 - Web site, 15
- general utilities library
 - atexit() function, 648-650
 - exit() function, 648-650
 - qsort() function, 650-654
 - stdlib.h header file, 827-833
 - string.h header file, 834-837
 - tgmath.h header file, 837-838
 - time.h header file, 838-842
 - wchar.h header file, 842-849
 - wctype.h header file, 849-852
- get choice() function, menus, 291-292
- get int() function, 288
- getc() function, getting characters from files, 499
- getchar() function, 16, 223-226, 268-272, 295

- getche() function, 270
- getchoice() function, menus, 291
- getinfo() function, 546, 550
- gets() function, 397-399, 407-409, 506
- get_first() function, 292
- global data, const type qualifier, 483-484
- global variables, 451
- global.c program, code, 461-462
- glue.c program, code, 625
- GNU, gcc compiler, 15
- gobble() function, 479
- golf.c program, code, 133
- goto
 - command, 260, 800
 - keyword, 799
 - statements, 257-259
- graphical user interfaces (GUIs), 274
- graphics
 - bitmapped images, 666
 - lossless compression, 666
 - lossy compression, 666
- greater than operator (>), 148, 177, 185, 785
- greater than or equal to (>=) relational operator, 177, 185, 785
- greatest width integer functions, 807-808
- gsort() function, arrays, 573
- guess.c program, code, 279-280
- GUIs (graphical user interfaces), 274

H

- h conversion modifier, 119
- .h file extension, 327, 629
- h modifier, printf() function, 105
- %h (strftime function() format specifier), 841
- handling characters, 803-804
- handling signals, 817-818
- handling strings, 834-837
- Harbison, Samuel P., 783
- hash symbol (#), 268
- header files, 34
 - assert.h, 801
 - complex.h, 801-803, 863-864
 - conio.h, 270
 - ctype.h, 803-804
 - errno.h, 804
 - external variables declarations, 632
 - fenv.h, 805-807, 861-862
 - float.h files

functions

- compiling, 327-330
- declarations, 631
- descriptions, 27
- .h file extension, 327, 629
- #include preprocessor directive, 631-632
- inttypes.h, 807-808
- iso646.h, 857-858
- limits.h files, constants, 98-101
- list.h interface header file program code, 680
- locale.h, 808-811
- macro functions, 631
- manifest constants, 631
- math.h, 815, 862-863
- names.c source file program code, 630
- names.h header file program code, 629
- queue.h interface header file program code, 694-695
- setjmp.h, 817
- signal.h, 817-818
- stdarg.h, variable arguments, 658-660, 818-824
- stdbool.h, 819
- stddef.h, 820
- stdint.h, integer types, 820-824
- stdio.h, 268, 824-827
- stdlib.h, 827-833
- string.h, 834-837
- structure template definitions, 631
- tgmath.h, 837-838
- time.h, 838-842
- tree.h interface header file program code, 714-716
- type definitions, 631
- useheader.c program code, 630-631
- wchar.h, 842-849
- wctype.h, 849-852

hello.c program, code, 440-441

hexadecimal numbers (base 16 system), 58-59, 591-592

hh conversion modifier, 118

hh modifier (printf() function), 105

hiding

- data, 680, 689
- outer definitions (variables), 454

hiding.c program, code, 454-455

high-level programming languages, 6-7

high-order bit (bit 7), 588

history of C, 1

Hoare, C.A.R., 650

hotel.c function support module, code, 328-329

hotel.h header files, code, 329-330

HUGE_VAL macro, 811

HUGE_VALF macro, 811

HUGE_VALL macro, 811

I

- %i conversion specifier, 102, 118
- %l (strtime function() format specifier), 841
- I macro, 801
- IBM PCs
 - compatibles, buffering functions, 270
 - DOS compilers, 17
- identifiers, 30
 - compilers, C99 standard, 462
 - keywords, 43
 - of declarations, modifiers, 571-573
 - preprocessor, 633
 - reserved, 31, 43
 - starbar, 303
- identifying members, arrays of structures, 535-536
- IDEs (integrated development environments), 10, 15-17, 629
- IEC, 861
- if else statements, 222-223, 260
 - ? (conditional operator), 244-245
 - { } (curly braces), 233
 - conditional expressions, 244-245
 - else if statements, 228-230
 - else statements, pairing, 231-232
 - getchar() function, 223-226
 - if statements, comparing, 223
 - multiple choices, 228-230
 - nesting, 232-235
 - putchar() function, 223-226
 - switch statements, when to use, 256
- if keyword, 797
- #if preprocessor directive, conditional compilations, 637-638
- if statements, 235, 260, 797-798
 - colddays.c program, code, 220-221
 - expressions, 235
 - if else statements, comparing, 223
 - if...else if...else sequence, 232
- #ifdef preprocessor directive, conditional compilations, 633-635
- ifdef.c program, code, 634
- #ifndef preprocessor directive, conditional compilations, 635-637
- images, bitmapped graphics, 666
- imaginary floating points, 78
- Imaginary I macro, 801
- imaginary keyword, 53
- imaginary macro, 801
- imaginary numbers, 76, 792
- imax() function, 315-317

- imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom) function, 807
- imin() function, 310-313
- implementation files, 684-688, 698-700, 725-729
- implementing
 - ADTs (abstract data types) queues, interfaces, 683, 691-700
 - binary search trees, 716-730
 - menus, 290-291
- #include "usr/biff/p.h" directive, 628
- #include "hot.h" directive, 628
- #include "mystuff.h" directive, 628
- #include <stdio.h> directive, 628
- #include directive, 631
- include files, " " (double quotation marks), 327
- #include preprocessor directive, 628-632
- include statement, preprocessor directives, 26-27
- including files, accessing C library, 643-644
- including libraries, accessing C library, 644
- increment operator (++), 144-150, 159
- incrementing a pointer operation, pointers, 368
- incrementing int pointers, 370
- indefinite loops, sweetie1.c program, code, 186
- index variable, 452
- indices
 - arrays, 204, 346, 351-353
 - values, changing, 173
- indirect membership operator (->), 565, 789
- indirection operator (*), 334-335, 361, 375
- infinite loops, 146, 175
- INFINITY macro, 811
- init statement, main() function, 27
- InitializeList() function, 680, 686
- initializers
 - designated, 350-351, 532
 - structures, 531
- initializing
 - arrays, 346-351
 - automatic variables, 456
 - char types, 65
 - character string arrays, 400-401
 - external variables, 461
 - int type variables, 56-57
 - structure pointers, 540-541
 - structure variables, 531-532
 - two-dimensional arrays, 357
 - unions, 563
- inline functions, 640-643, 869
- InOrder() function, 724
- input. *See also* I/O
 - buffered, 269-270, 279-281, 496
 - buffered and unbuffered, comparing, 269
 - character, user interfaces, 281-284
 - character and numeric, mixing, 292-295
 - echoing, 268-270
 - files, standard input, 495-496
 - numeric, user interfaces, 281-284
 - reading, scanf() function, 119-120
 - scanf function, 120
 - sign off() function, 649
 - standard, 274-276, 279, 495-496
 - streams, 271, 295, 512, 520
 - strings, 406-412
 - terminating, 268
 - too bad() function, 649
 - unbuffered, 269
 - unechoed, 270
 - validating, 268, 284-290, 295
- input.c program, code, 117, 423-424
- input/output. *See* I/O
- inserting elements, 708-709
- instruction sets, CPUs (central processing units), 6
- instructions for compilers, placing in source code, 639-640
- int abs(int n) function, 832
- int atexit(void (*func)(void)) function, 830
- int atoi(const char * nptr) function, 828
- int atol(const char * nptr) function, 828
- int classify(real-floating x) function, 813
- int fclose(FILE *) function, 824
- int fegetround(void) function, 806
- int feholdexcept(fenv_t *envp) function, 807
- int feof(FILE *) function, 825
- int feof(FILE *fp) function, 515
- int ferror(FILE *) function, 825
- int ferror(FILE *fp) function, 515
- int fesetround(int round) function, 806
- int fetestexcept(int excepts) function, 806
- int fflush(FILE *) function, 825
- int fflush(FILE *fp) function, 512
- int fgetc(FILE *) function, 825
- int fgetpos(FILE * restrict, fpos_t * restrict) function, 825
- int fmod(double x, double y) function, 815
- int fprintf(FILE * restrict, const char * restrict, ...) function, 825
- int fputc(int, FILE *) function, 825
- int fputs(const char * restrict, FILE * restrict) function, 825

- `int fscanf(FILE * restrict, const char * restrict, ...) function`, 825
- `int fseek(FILE *, long, int) function`, 825
- `int fsetpos(FILE *, const fpos_t *) function`, 825
- `int getc(FILE *) function`, 825
- `int getchar( ) function`, 826
- `int ilogb(double x) function`, 813
- `int (integer) types`, 77
 - argument, `argc` (argument count), 439
 - arrays, 204
 - constants, 57, 62, 98
 - data type, 29, 57, 61
 - declaring, 60
 - floating-point types, comparing, 54
 - format conversions, `inttypes.h` header file, 807-808
 - functions, 304
 - hexadecimal numbers, 58-59
 - keywords, 29, 53, 59-60
 - long constants, 62
 - long long constants, 62
 - long long types, 62-64
 - long types, 62-64
 - `main( )` function, 25
 - multiple types, 60-61
 - octal numbers, 58-59
 - overflow, 61-62
 - pointers, incrementing, 370
 - pointer-to-int, 362
 - short types, 62-64
 - signed, 601, 791
 - sizes for systems, 79
 - statement, 25
 - `stdint.h` header file, 820-824
 - unsigned, 62-64, 601, 606
 - values, `printf.c` program, code, 57-58
 - variables, initializing, 56-57
- `int isalnum(int c) function`, 803
- `int isalpha(int c) function`, 803
- `int isblank(int c) function`, 803
- `int iscntrl(int c) function`, 804
- `int isdigit(int c) function`, 804
- `int isfin(real-floating x) function`, 813
- `int isfinite(real-floating x) function`, 813
- `int isgraph(int c) function`, 804
- `int isgreater(real-floating x, real-floating y) function`, 816
- `int isgreaterequal(real-floating x, real-floating y) function`, 816
- `int isless(real-floating x, real-floating y) function`, 816
- `int islessequal(real-floating x, real-floating y) function`, 816
- `int islessgreater(real-floating x, real-floating y) function`, 816
- `int islower(int c) function`, 804
- `int isnan(real-floating x) function`, 813
- `int isnormal(real-floating x) function`, 813
- `int isprint(int c) function`, 804
- `int ispunct(int c) function`, 804
- `int isspace(int c) function`, 804
- `int isunordered(real-floating x, real-floating y) function`, 816
- `int isupper(int c) function`, 804
- `int iswalnum(wint_t wc) function`, 850
- `int iswalpha(wint_t wc) function`, 850
- `int iswblank(wint_t wc) function`, 850
- `int iswcntrl(wint_t wc) function`, 850
- `int iswdigit(wint_t wc) function`, 850
- `int iswgraph(wint_t wc) function`, 850
- `int iswlower(wint_t wc) function`, 850
- `int iswprint(wint_t wc) function`, 850
- `int iswpunct(wint_t wc) function`, 850
- `int iswspace(wint_t wc) function`, 850
- `int iswupper(wint_t wc) function`, 850
- `int iswxdigit(wint_t wc) function`, 850
- `int isxdigit(int c) function`, 804
- `int mblen(const char *s, size_t n) function`, 833
- `int mbsinit(const mbstate_t *ps) function`, 847
- `int mbtowc(wchar_t *pw, const char function`, 833
- `int memcmp(const void *s1, const void *s2, size_t n) function`, 834
- `int printf(const char * restrict, ...) function`, 826
- `int putc(int, FILE *) function`, 826
- `int putchar(int) function`, 826
- `int puts(const char *) function`, 826
- `int raise(int sig) function`, 818
- `int rand(void) function`, 829
- `int remove(const char *) function`, 826
- `int rename(const char *, const char *) function`, 826
- `int scanf(const char * restrict, ...) function`, 826
- `int setjump(jmp_buf env) function`, 817
- `int setvbuf(FILE * restrict, char * restrict, int, size_t) function`, 826
- `int setvbuf(FILE *fp, char *buf, int mode, size_t size) function`, 512-513
- `int signbit(real-floating x) function`, 813
- `int snprintf(char * restrict, size_t n, const char * restrict, ...) function`, 826
- `int sprintf(char * restrict, const char * restrict, ...) function`, 826
- `int sscanf(const char * restrict, const char * restrict, ...) function`, 826

- `int strcmp(const char * s1, const char * s2)` function, 431
- `int strcmp(const char *s1, const char *s2)` function, 835
- `int strcoll(const char *s1, const char *s2)` function, 835
- `int strlen(const char * s)` function, 836
- `int strncmp(const char * s1, const char * s2, size_t n)` function, 431
- `int strncmp(const char *s1, const char *s2, size_t n)` function, 835
- `int system(const char *str)` function, 831
- `int tm hour` (struct tm structure member), 838
- `int tm isdst` (struct tm structure member), 839
- `int tm mday` (struct tm structure member), 838
- `int tm min` (struct tm structure member), 838
- `int tm mon` (struct tm structure member), 839
- `int tm sec` (struct tm structure member), 838
- `int tm wday` (struct tm structure member), 839
- `int tm yday` (struct tm structure member), 839
- `int tm year` (struct tm structure member), 839
- `int tolower(int c)` function, 804
- `int toupper(int c)` function, 804
- `int ungetc(int c, FILE *fp)` function, 512
- `int ungetc(int, FILE *)` function, 826
- `int vprintf(FILE * restrict, const char * restrict, va_list)` function, 826
- `int vprintf(const char * restrict, va_list)` function, 827
- `int vsprintf(char * restrict, const char * restrict, va_list)` function, 827
- `int vsprintf(char * restrict, size_t n) const char * restrict, va_list)` function, 827
- `int wctob(wint_t c)` function, 847
- `int wctomb(char *s, wchar_t wc)` function, 833
- `int16_t` type, 820
- `int32_t` type, 821
- `int64_t` type, 821
- `int8_t` type, 820
- `intconv.c` program, code, 111-112
- integer 7, storing as binary code, 54
- integers, 54
 - binary, 323-324, 588
 - bits, accessing, 611
 - constants, 565-569, 823-824, 855
 - data-type keywords, 53
 - division, 137
 - exact width types, 853
 - extended types, 852-855
 - fastest minimum width types, 854
 - greatest width funtions, 807-808
 - int, 77, 791
 - long, 77, 791
 - long int, 77, 791
 - long long, 77, 791
 - long long int, 7
 - maximum width types, 855
 - minimum width types, 853-854
 - pointer values, 822, 855
 - properties, 676
 - short, 77, 791
 - short int, 77, 791
 - signed, 77, 589, 791
 - truncation, 81
 - types, 59-60
 - unions, 607
 - unsigned, 77, 791
- Integrated Development Environments (IDEs), 10, 15-17, 629
- integrated environments, command-line arguments, 439
- interactive programs, 51
- `interchange( )` function, 332-334
- interfaces
 - ADTs (abstract data types), 678-700
 - binary search trees, 713-716
 - creating, 280-282
 - GUIs (graphical user interfaces), 274
 - user, 279-284, 290-295
- internal linkage, 452-453, 463-464, 488
- International C Standard (The)* (ISO/IEC 9899 1999), 783
- International Electrotechnical Committee (IEC), 858, 861
- International Organization for Standardization (ISO), 858
- `INTMAX_MAX` constant, 823
- `INTMAX_MIN` constant, 823
- `intmax_t imaxabs(intmax_t j)` function, 807
- `intmax_t strtoumax(const char * restrict nptr, char ** restrict endptr, int base)` function, 808
- `intmax_t` type, 822
- `intmax_t wcstoumax(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base)` function, 808
- `INTN_MAX` constant, 823
- `INTN_MIN` constant, 823
- `INTPTR_MAX` constant, 823
- `INTPTR_MIN` constant, 823
- `intptr_t` type, 822
- `InTree( )` function, 719
- `inttypes.h` header file, 70-71, 807-808
- `int_fast16_t` type, 822
- `int_fast32_t` type, 822
- `int_fast64_t` type, 822
- `int_fast8_t` type, 822

- INT_FASTN_MAX constant, 823
 - INT_FASTN_MIN constant, 823
 - int_least16_t type, 821
 - int_least32_t types, 821
 - int_least64_t type, 821
 - int_least8_t type, 821
 - INT_LEASTN_MAX constant, 823
 - INT_LEASTN_MIN constant, 823
 - inventories, book, 527-529
 - invert4.c program, code, 600-601
 - invoking. *See* calling functions
 - I/O (input and output)
 - # (hash symbol), 268
 - buffers, 269-270, 280
 - C preprocessor, 95-98
 - character strings, 91-95
 - checking.c program, 286-289
 - const modifier, 98
 - constants, 95-98
 - echo eof.c program, 273-274
 - echo.c program, 268-270
 - echoing the input, 268
 - end of files, marking, 271-274
 - entry errors, avoiding, 295
 - files, 493-497, 501-510, 518-520
 - float.h file, symbolic constants, 100
 - fread() function, append.c program code, 516-517
 - fully buffered, 269
 - functions, 27, 267-268, 515-518, 824-827
 - fwrite() function, append.c program code, 516-517
 - guess.c program, code, 279-280
 - input
 - buffered, 269
 - redirecting, 274-276, 279
 - terminating, 268
 - unbuffered, 269
 - validating, 284-290, 295
 - keyboards, 270-274
 - library, 824-827
 - limits.h file, symbolic constants, 99
 - line-buffered, 269
 - low-level, 271
 - manifest constants, 98-101
 - menuette.c program, code, 293-295
 - menus, 290-295
 - newline character, checking, 271
 - output, redirecting, 274-279
 - pizza.c program, code, 96-98
 - printf() function, 101-116, 121-124
 - printout.c program, code, 102-104
 - scanf() function, 101, 116-124
 - showchar1.c program, code, 282
 - showchar2.c program, code, 283
 - standard, 271, 496-501, 512-520
 - std stream, 271
 - stdout stream, 271
 - streams, 271
 - strings, options, 414-417
 - talkback.c program, code, 89-91
 - wide-character I/O functions, 843-844
 - ioctl() function, 270
 - isalnum() function, 227
 - isalpha() function, 226-227
 - isctrl() function, 227
 - isdigit() function, 227
 - isgraph() function, 227
 - islower() function, 227, 240
 - ISO (International Organization for Standardization), 858
 - ISO C standard, 18
 - ISO/ANSI C standard, 19
 - ISO/ANSI C90, keywords, 43
 - ISO/ANSI C99, keywords, 43
 - iso646.h header file, 857-858
 - isprint() function, 227
 - ispunct() function, 227, 436
 - isspace() function, 227, 241
 - isupper() function, 227
 - item duplication, binary search trees, 735
 - Item type, 690, 678
 - iterations, 173
- ## J
-
- j (modifier, printf() function), 105
 - %j (strtime function() format specifier), 841
 - join chk.c program, code, 419-420
 - journals, *C/C++ Users Journal*, 781
 - jumps
 - non-local, 817
 - programs, 259-260, 799-800
 - statements (break statements), 246, 249-250, 254-255, 261
- ## K
-
- Kernighan, Brian W., 18, 782-783
 - keyboards
 - input, 51, 270-274
 - logical operators, alternate spellings, 237-238

- shortcuts
 - Ctrl+D, 276
 - Ctrl+Z, 271, 274-276
- output, stdout stream, 271
- keystrokes, 20-21, 270
- keywords
 - ANSI C qualifiers, 486-487
 - auto, 453
 - Bool, 53, 78, 791
 - break, 799
 - char, 53, 77, 791
 - characters, 791
 - complex, 53
 - const, 481-482, 793
 - arrays, initializing, 347
 - constants, 373-375
 - parameters, 371-373
 - strings, 432
 - continue, 799
 - data type, 52-55, 77, 790-791
 - do, 797
 - double, 53
 - enum, 565
 - extern, 459, 463, 467, 793
 - float, 53
 - for, 796
 - goto, 799
 - if, 797
 - imaginary, 53
 - int, 29, 53
 - int data type, 61
 - integer types, 59-60
 - ISO/ANSI C90, 43
 - ISO/ANSI C99, 43
 - long, 53, 59
 - long double, 53
 - register, 453
 - restrict, 485-486, 793
 - restricted, 457
 - return, 301, 311
 - short, 53, 59
 - signed, 53, 60, 77
 - signed integers, 791
 - static, 346, 452, 486-487, 792
 - struct, 529, 540
 - switch, 798
 - typedef, 464
 - unsigned, 53, 59, 791
 - unsigned integers, 791
 - variables, qualifying, 793

- volatile, 481, 484, 793
- while, 795
- Knuth, Donald E., 783
- Koenig, Andrew, 783

L

- l
 - conversion modifier, 119
 - modifier, printf() function, 105-106
- labels (case), vowels.c program code, 254-255
- Landis (AVL trees), 735
- languages
 - programming, 1
 - standards (C), 18-19
- LC ALL macro, 809
- LC COLLATE macro, 809
- LC CTYPE macro, 809
- LC MONETARY macro, 809
- LC NUMERIC macro, 809
- LC TIME macro, 809
- ldiv_t ldiv(long numer, long denom) function, 832
- ldiv_t type, 827
- leaf (nodes), 720
- left shift operator (<<), 596
- lengths of strings, strlen() function, 93-95
- less than (<), 131
 - operator, 148
 - relational operator, 177, 185, 785
- less than or equal to (<=) relational operator, 177, 185, 785
- lesser.c program, code, 310-311
- lethead1.c program, analyzing, 303-305
- lethead2.c program, code, 306-307
- levels of files, 495
- libraries
 - ANSI C, 800
 - assert.h header file, 801
 - C99 additions, 801-811
 - complex.h header file, 801-803
 - ctype.h header file, 803-804
 - errno.h header file, 804
 - fenv.h header file, 805-807
 - inttypes.h header file, 807-808
 - locale.h header file, 808-811
 - stdlib.h header file, 827-833
 - string functions, 417, 430-432
 - string.h header file, 834-837
 - tgmath.h header file, 837-838

- time.h header file, 838-842
- Unix I/O functions, 267
- wchar.h header file, 842-849
- wctype.h header file, 849-852
- assert, debugging programs, 654-656
- C library, 94, 643-645
- functions, 802
- general utilities
 - atexit() function, 648-650
 - exit() function, 648-650
 - qsort() function, 650-654
 - stdlib.h header file, 827-833
 - string.h header file, 834-837
 - tgmath.h header file, 837-838
 - time.h header file, 838-842
 - wchar.h header file, 842-849
 - wctype.h header file, 849-852
- inclusion, accessing C library, 644
- I/O, 824-827
- math, 645-648, 862-863
- Math, C99, 811-818
- routines, linkers, 9, 12
- stdarg.h header file, variable arguments, 658-660
- string.h, 656-657
- limitations of C, 4
- limits.h files, constants, 98-101
- LINE macro, 638
- line-buffered I/O, 269
- #line preprocessor directive, 639
- lines, 616-617, 639
- linkages, variables, 449, 452-453, 459-464, 488
- linked lists, 668-676, 708-710
- linkers, 9, 12
- Linux
 - >> (operator), 277
 - | (pipe) operator, 277
 - > (redirection operator), 276-277
 - < (redirection operator), 275-277
 - functions, compiling, 326
 - gcc compiler, 15, 326
 - input, redirecting, 275-276, 279
 - output, redirecting, 275-279
 - programs, 10, 15
- list.c file, 683, 688
- list.c implementation file program, 684-688
- list.h file, 683, 688
- list.h interface header file program code, 680
- listings. *See* code
- ListIsEmpty() function, 686
- ListIsFull() function, 686
- ListItemCount() function, 686
- lists
 - ADTs (abstract data types), 677-678
 - EmptyTheList() function, 688
 - InitializeList() function, 680, 686
 - linked, 668-676, 708-710
 - ListIsEmpty() function, 686
 - ListIsFull() function, 686
 - ListItemCount() function, 686
 - ordered elements, binary searches, 710
 - replacement, #define preprocessor directive lines, 617
 - showmovies() function, 687
 - Traverse() function, 687
- literals, 387-389, 552-553
- ll conversion modifier, 119
- ll modifier, printf() function, 105
- lldiv_t lldiv(long numer, long denom) function, 832
- lldiv_t type, 827
- local time, 839
- local variables, 305
- locale.h header file, 808-811
- localization functions, 808
- localizations, locale.h header file, 808-811
- loccheck.c program, code, 331
- logical expressions, 787
- logical lines, 616-617
- logical operators, 787
 - && (and), 237
 - ! (not), 237
 - || (or), 237
 - alternate spellings, 237-238
 - bitwise, 592-594
 - chcount.c program, code, 236
 - conditions, testing, 260
 - evaluations, order of, 238-239
 - expressions, 239
 - precedence, 238
 - ranges, testing, 240
- long constants, 62
- long double complex type, 76
- long double keyword, 53
- long double strtols(const char * restrict npt, char ** restrict ept) function, 829
- long double types, 72-73, 78, 792
- long int lrint(double x) function, 815
- long int lround(double x) function, 815
- long int (signed integer), 791
- long int types, 77
- long keyword, 53, 59
- long labs(int n) function, 832
- long long constants, 62

- long long int llrint(double x) function, 815
- long long int llround(double x) function, 815
- long long int types, 77
- long long labs(int n) function, 832
- long long (signed integer), 791
- long long strtoll(const char * restrict npt, char ** restrict ept, int base) function, 829
- long long types, 60-63, 77
- long (signed integer), 791
- long strings, printing, 115-116
- long strtol(const char * restrict npt, char ** restrict ept, int base) " function, 829
- long types, 60-64, 77
- longstrg.c program, code, 115-116
- looping statements, sequences, 169
- loops
 - arrays, 203-206
 - break statements, 246, 249-250
 - buffered input, 280
 - char arrays, 204
 - choosing, 200-201
 - concepts, 210
 - conditional, 174-175
 - continue statements, 246-249
 - counting, 186
 - do while, 198-200
 - entry-condition, while statement, 795
 - evaluating expressions, troubleshooting, 181
 - floating-point comparisons, < and > cautionary note, 177
 - for, 187-197, 204-206, 235
 - functions, 206-209
 - indefinite, sweetie1.c program, code, 186
 - indexes, changing values, 173
 - infinite, 146, 175
 - int arrays, 204
 - iterations, 173
 - nested, 201-203
 - null statements, 176
 - pseudocode, 172
 - reading, while loops (summing.c program), 172
 - recursion, comparing, 318
 - relational expressions, 173
 - scanf() function, 171
 - shoes1.c program, code, 129-130
 - shoes2.c program, code, 130-131
 - strings, 204
 - sump() function, 365
 - test expressions, 248

- while (entry-condition loops), 152
 - { } (curly braces), 131
 - Bool type, 182
 - boolean.c program, code, 182-183
 - cmpflt.c program, code, 177-178
 - compound statements (blocks), 154-155
 - conditional loops, 174-175
 - entry.c program, code, 198-199
 - relational expressions, 176-184
 - shoes2.c program, code, 130-131
 - single-character I/O, 268
 - structure, 173
 - summing.c program, 170-172
 - sweetie1.c program, code, 186
 - switch statements, 290
 - t and f.c program, code, 178-179
 - terminating, 173-174
 - trouble.c program, code, 180-182
 - truth.c program, code, 179-180
 - values, processing, 288
 - when.c program, code, 174
 - while statements, 173, 185

- lossless compression, 666
- lossy compression, 666
- low-level files, 495
- low-level I/O, 271
- low-order bit (bit 0), 588
- lvalues, data objects, 133

M

- %m (strftime function() format specifier), 841
- mac arg.c program, code, 621-622
- machine language (numeric instruction code), 6
- machines, collating sequences, 423
- Macintosh
 - command-line arguments, 440
 - compilers, 327
 - Metrowerks CodeWarrior compiler, 17-18
- macros
 - " " (double quotation marks), 619
 - ... (ellipsis) variadic macro, 626-627
 - arguments, 621, 624, 628
 - bool, 819
 - category, 809
 - char *currency_symbol, 809
 - char *decimal_point, 809
 - char *grouping, 809
 - char *int_curr_symbol, 809

char *mon_decimal_point, 809
 char *mon_grouping, 810
 char *mon_thousands_sep, 809
 char *negative_sign, 810
 char *positive_sign, 810
 char *thousands_sep, 809
 char frac_digits, 810
 char int_frac_digits, 810
 char int_n_cs_precedes, 810
 char int_n_sep_by_space, 811
 char int_n_sign_posn, 811
 char int_p_cs_precedes, 810
 char int_p_sep_by_space, 810
 char int_p_sign_posn, 811
 char n_cs_precedes, 810
 char n_sep_by_space, 810
 char n_sign_posn, 810
 char p_cs_precedes, 810
 char p_sep_by_space, 810
 char p_sign_posn, 810
 complex, 801
 Complex I, 801
 ctype.h functions, 437
 DATE, 638
 #define preprocessor directive, 617-622
 defining, 633, 843
 EDOM, 804
 EILSEQ, 804
 ERANGE, 804
 expansion, #define preprocessor directive lines, 617
 false, 819
 FE ALL EXCEPT, 806
 FE DFL ENV, 806
 FE DIVBYZERO, 805
 FE DOWNWARD, 806
 FE INEXACT, 805
 FE INVALID, 806
 FE OVERFLOW, 806
 FE TONEAREST, 806
 FE TOWARDZERO, 806
 FE UNDERFLOW, 806
 FE UPWARD, 806
 fenv.h header file, 805-806
 FILE, 638
 FP_FAST_FMA, 812
 FP_FAST_FMAF, 812
 FP_FAST_FMAL, 812
 FP_ILOGB0, 812
 FP_ILOGBNAN, 812
 FP_INFINITE, 812
 FP_NAN, 812
 FP_NORMAL, 812
 FP_SUBNORMAL, 812
 FP_ZERO, 812
 function-like, 617, 621-625
 functions, 627-628, 631
 HUGE_VAL, 811
 HUGE_VALF, 811
 HUGE_VALL, 811
 I, 801
 imaginary, 801
 Imaginary I, 801
 INFINITY, 811
 LC ALL, 809
 LC COLLATE, 809
 LC CTYPE, 809
 LC MONETARY, 809
 LC NUMERIC, 809
 LC TIME, 809
 LINE, 638
 MATH_ERREXCEPT, 812
 math_errhandling, 812
 MATH_ERRNO, 812
 names, spaces, 628
 NAN, 811
 NULL, 809, 820, 843
 object-like, 617
 offsetof (type, member-designator), 820
 predefined, predef.c program code, 638-639
 program speeds, 628
 SIGABRT, 817
 SIGFPE, 817
 SIGILL, 817
 SIGINT, 817
 signal.h header file, 817
 SIGSEGV, 817
 SIGTERM, 817
 SIG_DFL, 818
 SIG_ERR, 818
 SIG_IGN, 818
 SQUARE, 622
 stdbool.h header file, 819
 STDC, 638
 STDC HOSTED, 638
 STDC VERSION, 638
 stddef.h header file, 820
 struct lconv, 809-811
 TIME, 638
 true, 819
 type va_arg(va_list ap, type), 819
 VA_ARGS variadic, 626-627
 va copy(), 659

- va_end(), 659
- va_start(), 658
- variable arguments, 819
- variadic, 626-627
- va_arg(), 658
- void (*f)(int), 818
- void assert(int exprs), 801
- void va_copy(va_list dest, va_list src), 819
- void va_end(va_list ap), 819
- void va_start(va_list ap, parmN), 819
- WCHAR_MAX, 843
- WCHAR_MIN, 843
- wctrans_t, 850
- wctype.header file, 849-850
- wctype_t, 850
- WEOF, 843
- WEOF constant expression, 850
- wint_t, 849
- main() function, 26-28, 288
 - arguments, 438
 - defining, 304
 - getinfo() function, 546
 - int, 25
 - int statement, 34
 - interchange() function, variable swapping, 332-334
 - return statements, 34
 - void, 25
- maintaining programs, 10
- make command (UNIX), 326
- makeinfo() function, 546-548
- MakeNode() function, 717
- mall advice booth simulation, 702-708
- mall.c program, code, 704-706
- malloc() function, 670, 793
 - arguments, 475
 - arrays, 476
 - data presentation, 668
 - dynamic memory allocation, 480-481
 - memory allocation, 475-478
 - pointers, 476
 - structures, 550-552
- manifest constants, 98-101, 631. *See also* symbolic constants
- manybook.c program, 533-537
- manydice.c file program, code, 473-474
- mapping
 - characters, 616, 227
 - wide characters, 849-852
- marking end of files, 271-274
- masks, bitwise operators, 594-595
- matching printf() function specifiers, 64
- math
 - functions, ANSI C standard, 812-816
 - type-generic, 837-838
- Math library, 645-648, 811-818, 862-863
- math.h header file, 811-818, 862-863
- MATH_ERREXCEPT macro, 812
- math_errhandling macro, 812
- MATH_ERRNO macro, 812
- maximum width types, 822, 855
- mbstate_t type, 843
- MB_CUR_MAX constant, 828
- mechanics of programming, 11-18
- members (structure)
 - . (dot) operator, 532
 - accessing, 532-533
 - arrays of structures, identifying, 535-536
 - declarations, 529
 - flexible array (C99), 554-556
 - functions, 541-542
 - passing, 542-543
 - pointers, accessing, 541
 - struct tm structure, 838-839
- membership (.) operators, 788-789
- memcpy() function, 486, 656-657
- memmove() function, 486, 656-657
- memory
 - addresses, finding, 331
 - allocating, 475-481, 793
 - bits, 53
 - bytes, 53
 - caching, 484
 - char arrays, 204
 - dynamic memory allocation, 387
 - int arrays, 204
 - manybook.c program, 533
 - programs, free() function, 675
 - RAM (random-access memory), 5
 - static, strings, 402
 - structures, 530, 533
 - words, 54
- mems.c program code, 656-657
- menuette.c program, code, 293-295
- menus, 290-295
- messages, error, 50, 639
- Metrowerks CodeWarrior compiler, 17-18
- min sec.c program, code, 143-144
- minimum width types, 70, 821, 853-854
- minus sign (-) operator, 134
- mismatched conversions, printf() function, 110-113
- misuse.c program, code, 315

- mixing character and numeric input, 292-295
- modes
 - binary and text, comparing, 509
 - mode argument, 508
 - postfix, 144
 - strings, `fopen()` function, 498
- modifiable lvalue, 132-133
- modifiers
 - * (`printf()` or `scanf()` function), 121-123
 - const, 98, 866
 - conversions, 104-110, 118-119
 - declarations, 571-573
- modularity, programs, 206
- modules
 - functions, 288
 - hotel.c function support, code, 328-329
 - usehotel.c control, code, 327-328
- modulus operator (%), 142-144, 159
- monitors, printing to, 24
- MONTHS symbolic constant, 347
- MS-DOS, running programs, 10
- multibyte characters, 858-860
- multidimensional arrays, 354
 - one-dimensional, 358, 390
 - functions, 380-383
 - pointers, 375-380
 - rain.c program code, 355-356
 - three-dimensional, 358
 - two-dimensional, 355-358
- multiplication operator (*), 37, 135-136, 159
- mycomp() function, defining, 653-654

N

- `\n` (escape character), 66-67
- `/n` (newline character), 33
- name variables, 339
- name1.c program, code, 407-408
- name2.c program, code, 408
- name3.c program, code, 409-410
- nameln2.c program, code, 551-553
- names
 - declarations, modifiers, 571-573
 - files, 11, 639
 - functions, uses, 578
 - macros, 628
 - tags, structure declarations, 529
 - types, creating with typedef, 569-571
 - UCNs (universal character names), 858-859
 - variables, 31, 160

- names.c source file program code, 630
- names.h header file program, code, 629, 636
- names1.c program code, 545-546
- names2.c program code, 547
- namespaces, shared, 568-569
- naming
 - conventions for files, 11
 - functions, 25, 302
 - output files, 502
 - symbolic constants, 96
 - variables, 31, 462-463
- NAN macro, 811
- Nerfville Pet Club, binary search trees, 735
- nesting
 - if else statements, 232-235
 - if statements, if...else if...else sequence, 232
 - loops, 201-203
 - structures, 537-539
- new C elements, 51
- newline character (`\n`), 33
 - `\` (backslash), 616
 - checking, 271
 - `fgets()` function, 432
 - `strchr()` function, 432
- no data.c program, code, 347-348
- no linkage, 452-453, 488
- nodes
 - `AddItem()` function, 687
 - `CopyToNode()` function, 686, 697
 - deleting binary search trees, 722-723
 - leaf, deleting, 720
 - one-child, deleting, 721
 - two-child, deleting, 722
- nogo.c program, code, 420-421
- nogood.c program, code, 39
- non-local jumps, 817
- nono.c program, code, 413
- nonprinting characters, 65-68
- not (!) logical operator, 237, 787
- not equal to (!=) relational operator, 177, 185
- notations
 - exponential, 72
 - pointers, 367
 - scientific, 72
- null
 - characters, 91-92, 418
 - pointer, 476
 - statements, 176, 795
 - wide characters, 859

NULL

- constant, 827
- macro, 809, 820, 843

num variable, 26, 29

numbering lines, resetting, 639

numbers

- arrays, identifying, 204
- binary, 323-324, 587-591
- bit, 588
- complex, 76, 792, 801-803, 863-864
- converting to strings, 440
- decimal (base 10 system), 588, 591
- decimal points, keywords, 53
- exponential notations, 72
- Fibonacci, 325
- floating-point, 54, 84
 - binary, 589-590
 - control mode values, 861
 - double types, 72-73, 78, 792
 - environment, fenv.h header file, 805-807
 - exceptions, 861
 - exponential notations, 85
 - fixed decimal points, 85
 - float types, 72-73, 78, 791
 - IEC (International Electrotechnical Committee), 861
 - long double types, 72-73, 78, 792
 - overflow, 75
 - pi (π) numbers, storing, 55
 - relational operators, 177
 - round-off errors, 76
 - status flags, 861
 - underflow, 75
- hexadecimal (base 16 system), 58-59, 591-592
- imaginary, 76, 792
- input, validation, 289-290
- integers, 54, 57
- number variable, 452
- number systems
 - base 2 (binary), 588
 - base 8 (octal), 590-591
 - base 10 (decimal), 588, 591
 - base 16 (hexadecimal), 591-592
- octal (base 8 system), 58-59, 590-591
- pi (π) storing in floating-point format, 55
- prime, finding, 234
- random, 468-474
- real, 54
- scientific notations, 72
- sign magnitude, 589
- signed, 589

strings, converting, 440-443

subscript, array elements, 346

truncation, 81

numeric code, characters, 85

numeric computations (C99), 860-864

numeric input, 281-284, 292-295

numeric instruction code (machine language), 6

numerical constants, 124

O

%o conversion specifier, 102, 118

.o file extension, 14, 326

.obj file extension, 17, 326

object code files, 12-13

object-like macros, 617

objectives of programs, defining, 8

objects, 133, 326

octal numbers (base 8 system), 58-59, 590-591

offset argument, 508

offsetof (type, member-designator) macro, 820

offsets, arrays, 204

one's-complement method, binary or signed numbers, 589

one-child nodes, deleting, 721

one-dimensional arrays, 358, 390

opening files (fopen() function), 498-499

operands, 133

? (conditional operator), 246

expressions, values, 150-151

sizeof operator, 142

operations, pointers, 367-370, 511

operators

^ (EXCLUSIVE OR) (bitwise binary operator), 593-594, 789

-, (unary), 785

~ (bitwise unary operator), 592-59, 789

. (dot), 532, 541, 564

/ (division), 137-138, 159, 785

? (conditional), 244-246, 260, 787

|| (or) logical operator, 237

() (parentheses)

precedence, 139-141, 148

sizeof, 95

(operator), strings, 624-626

(operator), tokens, 625

— (arithmetic), 785

— (decrement), 144, 147-149, 159

= (assignment), 132-133, 158, 181-182

== (equal to relational operator), 171, 181-182, 785
 ! (logical NOT operator), 237, 787
 != (unequal relational operator), 785
 -> (operator), 541, 563
 -> (indirect membership), unions, 565
 # (strings), 624-626
 ## (tokens), 625
 % (arithmetic), 785
 % modulus, 142-144, 159
 - (minus sign), 134
 - (subtraction), 134, 159, 785
 - (unary), 159
 | (OR operator), 593
 | (OR) (bitwise binary operator), 789
 | (pipe), 277
 || (logical OR operator), 787
 & (address), 335, 368, 540
 & (AND) (bitwise binary operator), 593, 789
 & (pointer-related operator), 788
 && (logical AND operator), 237, 787
 * (indirection operator), 334-335, 360-361, 368
 * (multiplication), 135-136, 159, 785
 * (pointer-related operator), 788
 * (unary), 366
 + (addition), 134, 158, 785
 ++ (arithmetic), 785
 ++ (increment), 144-150, 159
 ++ (unary), 366
 < (less than), 148
 < (less than relational operator), 785
 < (redirection), 275-277
 << (bitwise binary operator), 789-790
 << (left shift), 596
 <= (less than or equal to relational operator), 785
 > (greater than), 148
 >> (right shift), 277, 597
 arithmetic, 132, 159, 785
 assignment, 192-195, 786
 association rule, 140
 binary, 134, 784, 789-790
 bitwise, 592-601, 606-611, 789-790
 cast, 158-159
 comma (,), 193-196, 235, 790
 concepts, 163
 conditional, 787
 dyadic, 134
 EXCLUSIVE OR (^) (bitwise binary operator), 789
 exponential growth, wheat.c program, code, 136-137
 exponentiating (pow() function), 132, 206

expressions, 139, 150-151
 functions, 301
 indirect membership (->), 789
 logical, 236-240, 260, 592-593, 787
 membership (.), 788-789
 NOT (!) logical operator, 237, 787
 operands, 133
 OR (|) (bitwise binary operator), 789
 OR (||) (logical operator), 787
 pointer-related, 788
 precedence, 138-141, 784-785
 redirection, 276-277
 relational, 148, 176-185, 785
 shift, 596-597
 sign, 788
 sizeof, 80, 95, 142, 159, 790
 structures, 565, 788-789
 (type), 159, 790
 unary, 134, 784, 789-790
 &, 330-332
 *, 366
 ++, 366
 union, 788-789
 OR (|) (bitwise binary operator), 593, 789
 OR (||) (logical operator), 237, 787
 order of evaluation (operator precedence), 140-141, 238-239
 order.c program, code, 366
 ordered lists, elements, binary searches, 710
 organization of book, 19-20
 output, 267. *See also* I/O
 binary, 513
 buffered, 496
 files
 buffers, flushing to, 512
 naming, 502
 standard output, 495-496
 flushing, escape.c program, 84
 standard, 275-279, 495-496
 streams, 520, 271
 strings, 412-414
 text, 513
 overflows, 61-62, 75

P

%p
 conversion specifier, 102, 118
 strftime function() format specifier, 841
 p and s.c program, code, 405-406

- packages
 - programming, binary search tree, 725-729
 - programs, parts of, 683
 - standard I/O, 271
- paint.c program, code, 244-245
- pairing if else statements and else statements, 231-232
- parameter declarations, const type qualifier, 482-483
- parameters, 82
 - ... (ellipsis), 818
 - arguments, comparing, 160
 - arrays, declaring, 363
 - const keyword, 371-373
 - formal, 160, 308-310, 340
 - functions, defining with arguments, 308
 - register variables, 457
- parentheses ()
 - arguments, 621, 628
 - definitions, 628
 - functions, 25, 28, 33, 304
 - operator precedence, 139-141, 148
 - pointers, 378
 - sizeof operator, 95
 - subexpressions, 225
- parrot.c program, 505-506
- parta.c file program, code, 465-466
- partb.c file program, code, 466
- passing
 - arguments, 113
 - structure members, 542-543
 - structures as arguments, 544
- pausing programs, 16
- PCs (IBM), 17, 270
- percent sign (%), printing, 104
- percent symbol group (%d), 34
- period (.), 226, 236
- pet clubs (Nerfville Pet Club), binary search trees, 735
- petclub.c program, 731-735
- pf pointer (ToUpper() function), declaring, 573
- pfun argument, 681
- physical lines, changing to logical lines, 616
- pi (π)
 - constant, 95
 - numbers, storing in floating-point format, 55
- pizza.c program, code, 96-98
- Plauger, P.J., 783
- Plum, Thomas, 783
- plus sign (+) operator, 134
- pnt add.c program, code, 359
- pointer-related operators, 788
- pointer-to-char, 476, 479
- pointer-to-float, 360
- pointer-to-int, 360-362
- pointer-to-void, 476, 479, 868
- pointers, 330
 - & (address operator), 335
 - * (asterisk), 336
 - * (declaration modifier), 571-572
 - * (indirection operator), 334-335
 - * (operator), 360
 - > (operator), 541, 563
 - addresses, double indirection, 375
 - arguments, 364-366
 - argv array, 439
 - arrays, 358-364, 367
 - assignment operation, 368
 - byte addressable, 360
 - character pointers and character arrays, comparing, 549-550
 - character string arrays, 401-403
 - communicating between functions, 336-338
 - const type qualifier, 482-483
 - constants, 359
 - data, 573
 - data objects, 360
 - declaring, 335-336, 382-383, 574
 - decrementing a pointer operation, 369
 - dereferencing operation, 368
 - differencing operation, 369
 - file, 498
 - FILE argument, 508
 - functions, 336-338, 573-579
 - head pointers, 670
 - incrementing a pointer operation, 368
 - int, incrementing, 370
 - malloc() function, 476
 - multidimensional arrays, 375-380
 - notation, 367
 - null, 476
 - operations, 367-370
 - pf (ToUpper() function), declaring, 573
 - properties, 375
 - returning to string location, 431
 - standard files, 501
 - stderr, 501
 - stdin, 501
 - stdout, 501
 - strcpy() function, 427
 - strings, 405-406, 434-435
 - structure pointers and structure arguments, comparing, 548-549
 - structures, 539-541, 550-552
 - sump() function, 365

- taking a pointer address operation, 368
- types, 77
- uninitialized, dereferencing, 369
- unions, `->` (operator), 563
- value-finding (dereferencing) operation, 368
- types, 77
- variables, 339, 359, 367-368
- polar coordinates, 662
- portability, 3
 - `fseek()` and `ftell()` functions, 510
 - types, 70-71
- positions
 - active (characters), 66
 - bit, bit fields, 611
- post `pre.c` program, code, 146
- postage.c program, code, 193-194
- postconditions, functions, 680
- postfix mode, 144
- pound sign (`#`), 27, 159, 268, 270, 616, 640
- `pound()` function, 159
- `pound.c` program, code, 159-161
- `pow()` function, 132, 206
- power of C, 3
- `power()` function, 209
- `power.c` program, code, 207-209
- `#pragma` preprocessor directive, 639-640
- `praise1.c` program, code, 92-93
- `praise2.c` program, code, 93-95
- Prata, Stephen, 784
- precedence
 - `—` (decrement operator), 148-149
 - `++` (increment operator), 148-150
 - `()` (parentheses), 148
 - logical operators, 238
 - operators, 138-141, 784-785
 - relational operators, 183-184
- preconditions, functions, 680
- `predef.c` program code, 638-639
- predefined macros, `predef.c` program code, 638-639
- `preproc.c` program, code, 617
- preprocessing source code, 27
- preprocessor
 - ANSI C standards, 615
 - compiling programs, 615
 - `#define` preprocessor directive, 616-627, 632-633
 - directives, 27, 631-639
 - `#elif` preprocessor directive, 637-638
 - `#else` preprocessor directive, 633-635
 - `#endif` preprocessor directive, 633-635
 - `#error` preprocessor directive, 639
 - functions, inline, 640-643
 - identifiers, 633
 - `#if` preprocessor directive, 637-638
 - `#ifdef` preprocessor directive, 633-635
 - `#ifndef` preprocessor directive, 635-637
 - `#include` preprocessor directive, 628-632
 - `#line` preprocessor directive, 639
 - `#pragma` preprocessor directive, 639-640
 - programs, translations for, 616
 - `#undef` preprocessor directive, 632-633
- prime numbers, finding, 234
- print statements, 35
- `print1.c` program, code, 57-58
- `print2.c` program, code, 62-63
- `printf()` function, 34, 51, 81-82, 85, 101, 124, 271, 301-302
 - `" "` (double quotation marks), 33
 - `#` flag, 107
 - `+` flag, 106
 - `-` flag, 106
 - `0` flag, 107
 - `*` modifier, 121-123
 - arguments, 33, 103
 - characters, 68
 - conversions, 101-107, 110-113
 - `.digit(s)` modifier, 105
 - exercise, 46
 - `%f` specifier, 51
 - `fathm ft.c` programs, 37-38
 - flags, 105-106
 - `flags.c` program, code, 109
 - float arguments, conversion specifications, 106
 - `floatcnv.c` program, code, 112-113
 - floating-point values, printing, 74
 - `floats.c` program, code, 108-109
 - functions, calling, 33
 - `h` modifier, 105
 - `hh` modifier, 105
 - `intconv.c` program, code, 111-112
 - interactive programs, 51
 - `j` modifier, 105
 - `l` modifier, 105-106
 - `ll` modifier, 105
 - long strings, printing, 115-116
 - `longstrg.c` program, code, 115-116
 - `printout.c` program, code, 102-104
 - `prntval.c` program, code, 115
 - return values, 114-115
 - `skip2.c` program, code, 122-123
 - space flag, 107
 - specifiers, matching, 64
 - strings, 414, 440

- strings.c program, code, 109-110
- t modifier, 106
- %u specifier, 61
- unsigned int values, 61
- utilizing, 104
- values, printing, 37-38
- varwid.c program, code, 121-122
- warnings, 84
- width.c program, code, 107-108
- z modifier, 106
- printf statement, 26
- printf() statement, 83-84
- printing
 - % (percent sign), 104
 - characters, 68-69
 - columns, fixed field widths, 123
 - control strings, 103
 - escape characters, 82-84
 - fathom ft.c program values, 37-38
 - floating-point values, 74-75
 - int type values, 57-58
 - long long types, 62-64
 - long strings, 115-116
 - long types, 62-64
 - short types, 62-64
 - strings, 92, 397-399
 - to screens, 24
 - unsigned types, 62-64
 - values, 37-38
- printout.c program, code, 102-104
- private variable names, 160
- prntval.c program, code, 115
- programmers, C benefits, 3-4
- programming
 - C, steps, 7-8
 - data representation, 665, 736-737
 - ADTs (abstract data types), 676-708
 - algorithms, 666
 - binary search trees, 711-735
 - bitmapped graphics images, 666
 - data hiding, 680, 689
 - films1.c program, code, 667
 - integer properties, 676
 - linked lists, 668-676, 708-710
 - lossless compression, 666
 - lossy compression, 666
 - malloc() function, 668
 - examples, bitwise operators, 598-601
 - exercises, 46-47
 - languages, 1
 - C++, 4
 - high-level, 6-7
 - portability, 3
 - mechanics, 11-18
 - modular, 288
 - packages (binary search trees), 725-729
 - portability, 3
 - steps, 9
- programs
 - add one.c, code, 144
 - addaword.c, code, 503-504
 - addemup.c, code, 152
 - altnames.c, code, 71
 - animals.c, code, 251-252
 - append.c, code, 516-517
 - arf.c program, code, 372-373
 - array2d.c, 381-383
 - assert library, 654-656
 - assert.c, code, 655
 - bases.c, code, 59
 - binary.c, code, 323-324, 598-599
 - book.c, code, 528-529
 - booksave.c, 560-561
 - boolean.c, code, 182-183
 - bounds.c, code, 352
 - break.c, code, 249-250
 - bugs, debugging, 10
 - byebye.c, code, 648-649
 - C code, 9
 - charcode.c, code, 68-69
 - chcount.c, code, 236
 - checking.c, 286-289
 - cmpflt.c, code, 177-178
 - colddays.c, code, 220-221
 - commentary, 10-11
 - comments, 28-29
 - compare.c, code, 421
 - compack.c, code, 422
 - compile time substitution, 96
 - compiling, 326, 615
 - concepts, 43-44
 - concrete.c, code, 11
 - convert.c, code, 157-158, 436, 646-647
 - copy1.c, code, 425-426
 - copy2.c, code, 427
 - copy3.c, code, 428-429
 - count.c, code, 496-497
 - creating (Linux), 15
 - cube.c, code, 188

cypher1.c, code, 224
 cypher2.c, code, 226-227
 day mon1.c, code, 346-347
 day mon2.c, code, 349
 day mon3.c, code, 361
 debugging, 10, 39-43, 654-656
 defines.c, code, 100
 designate.c, code, 350
 designing, 8
 diceroll.c file, code, 472-473
 diceroll.h file, code, 473
 divisors.c program, 234-235
 doubincl.c, code, 637
 dowhile.c, code, 198
 dual.c, code, 607-609
 dyn arr.c, code, 477
 echo eof.c, 272-274
 echo.c, 268-270, 438
 electric.c, code, 228-229
 entry errors, avoiding, 295
 entry.c, code, 198-199
 enum.c, code, 567-568
 error messages, 50
 escape.c, code, 83-84
 factor.c, code, 321-322
 fathm ft.c, 36-38
 fields.c, code, 604-605
 file eof.c, code, 278
 file-condensing, 501-503
 files, 11, 270
 files of code, storage classes, 464
 films1.c, code, 667
 films2.c, code, 672-673
 films3.c, code, 681-682
 flags.c program, code, 109
 flc.c, code, 388-389
 flexmemb.c, code, 554-556
 floatcnv.c, code, 112-113
 floats.c program, code, 108-109
 flow, 169, 253
 forc99.c, code, 455-456
 format.c, code, 429-430
 friend.c, code, 537-538
 friends.c, code, 539-540
 func ptr.c, 575-579
 functions, adding, 38-39
 funds1.c, code, 542
 funds3.c, code, 544
 funds4.c, code, 556-557
 fundsd2.c, code, 543
 global.c, code, 461-462
 glue.c, code, 625
 golf.c, code, 133
 guess.c, code, 279-280
 hello.c, code, 440-441
 hiding.c, code, 454-455
 ifdef.c, code, 634
 input.c, code, 117, 423-424
 intconv.c, code, 111-112
 interactive, 51
 invert4.c, code, 600-601
 I/O functions, 27, 267
 join chk.c, code, 419-420
 jumps, 259-260, 799-800
 keywords, 43
 lesser.c, code, 310-311
 lethead1.c, analyzing, 303-305
 lethead2.c, code, 306-307
 linkers, 9
 list.c implementation file, 684-688
 list.h interface header file, code, 680
 loccheck.c, code, 331
 longstrg.c, code, 115-116
 loops
 shoes1.c program, code, 129-130
 shoes2.c program, code, 130-131
 mac arg.c, code, 621-622
 maintaining, 10
 mall.c, 704-708
 manybook.c, 533-537
 manydice.c file, code, 473-474
 memory, free() function, 675
 mems.c, code, 656-657
 menuette.c, code, 293-295
 misuse.c, code, 315
 modifying, 10
 modularity, 206
 name1.c, code, 407-408
 name2.c, code, 408
 name3.c, code, 409-410
 nameln2.c, code, 551-553
 names.c source file, code, 630
 names.h header file, code, 629, 636
 names1.c program code, 545-546
 names2.c program code, 547
 no data.c, code, 347-348
 nogo.c, code, 420-421
 nogood.c, code, 39
 nono.c, code, 413
 objectives, defining, 8
 order.c, code, 366
 p and s.c, code, 405-406

- packages, parts of, 683
- paint.c, code, 244-245
- parrot.c, 505-506
- parta.c file, code, 465-466
- partb.c file, code, 466
- pausing, 16
- petclub.c, 731-735
- pizza.c, code, 96-98
- pnt add.c, code, 359
- post pre.c, code, 146
- postage.c, code, 193-194
- pound.c, code, 159-161
- power.c, code, 207-209
- praise1.c program, code, 92-93
- praise2.c program, code, 93-95
- predef.c, code, 638-639
- preproc.c, code, 617
- print1.c, code, 57-58
- print2.c, code, 62-63
- printout.c, code, 102-104
- prntval.c, code, 115
- program states, examining, 42-43
- programming exercises, 46-47
- proto1.c, code, 316-317
- pt ops.c, code, 367-368
- put out.c, code, 412
- put put.c, code, 416
- put1.c, code, 414
- put2.c, code, 415-416
- qsorter.c, code, 651-652
- queue.c implementation file, code, 698-700
- queue.h interface header file, code, 694-695
- r drive1.c driver, code, 469
- r drive2.c, code, 470
- rain.c, code, 355-356
- rand0.c function file, code, 468
- randbin.c, 519-520, code, 518-519
- readability, 35-36, 565-569
- recur.c, code, 318-319
- reducto.c, 501-503
- reserved identifiers, 43
- reverse.c, code, 507
- rhodium.c, 49-51
- rows1.c, code, 201-202
- rows2.c, code, 202-203
- rules.c, code, 140-141
- running in environments, 10
- running.c, code, 161-162
- s and r.c, code, 469-470
- scan str.c, code, 411
- scores.c, code, 204-206
- shoes1.c, code, 129-130
- shoes2.c, code, 130-131, 144-145
- showchar1.c, code, 282
- showchar2.c, code, 283
- showfpt.c, code, 74-75
- simple C, 23-25
- sizeof.c, code, 142
- skip.c, code, 246-247
- skip2.c, code, 122-123
- somedata.c, code, 348-349
- sort str.c, code, 432-433
- speeds, macros, 628
- squares.c, code, 135-136
- starsrch.c, code, 424-425
- statements, 151-155
- stillbad.c, code, 41-42
- str cat.c, code, 419
- strcnvt.c, code, 442
- streams, 271
- strings, 399-400
- strings.c, code, 109-110, 397-399
- structure, 34-35
- subst.c, code, 624
- sum arr1.c, code, 363-364
- sum arr2.c, code, 364-365
- summing.c, 170-172
- swap1.c, code, 332-333
- swap2.c, code, 333
- swap3.c, code, 336-337
- sweetie1.c, code, 186
- sweetie2.c, code, 187
- t and f.c, code, 178-179
- talkback.c, code, 89-91
- terminating, 497
- test.c, code, 417-418
- testing, 10
- tracing, 42
- translations for preprocessing, 616
- tree.h interface header file, code, 714-716
- trouble.c, code, 180-182
- truth.c, code, 179-180
- two func.c, code, 38-39
- typesize.c, code, 79-80
- use qc, code, 700-701
- useheader.c, code, 630-631
- varargs.c, code, 659-660
- vararr2d.c, code, 385-386
- variadic.c, code, 626
- varwid.c, code, 121-122
- vowels.c, code, 254-255
- warnings, 50

- wheat.c, code, 136-137
- while loops, 130-131
- while1.c, code, 175
- while2.c, code, 175-176
- width.c program, code, 107-108
- word-count, 240-244
- wordcnt.c, code, 242-244
- writing, 6
- zeno.c, code, 196-197
- zippo1.c, 376-377
- zippo2.c, code, 378
- projects, 16, 327
- promotion, data types, 156
- properties
 - of integers, 676
 - pointers, 375
 - strcpy() function, 427
 - variables, qualifying, 794
- proto1.c program, code, 316-317
- prototyping
 - ANSI C functions, 314-318
 - functions with arguments, 308-309
- pseudocode, 172
- pt ops.c program, code, 367-368
- PTRDIFF_MAX constant, 824
- PTRDIFF_MIN constant, 824
- ptrdiff_t type, 820
- punctuation characters, counting, 436
- pushing characters to input stream, 512
- put out.c program, code, 412
- put put.c program, code, 416
- put1.c program, code, 414
- put2.c program, code, 415-416
- putc() function, putting characters into files, 499
- putchar() function, 223-226, 268, 271, 302
- puts() function, 412-413, 418, 397-399

Q

- qsort() function (quick sort), 650-654
- qsorter.c program code, 651-652
- qualifiers
 - ANSI C types, 481-487
 - const type, 482-484
 - restrict type, 485-486
 - volatile type, 484-485
- qualifying variables, 793-794
- question mark (?), 66
- Queue type, 690

- queue.c implementation file program code, 698-700
- queue.h file, 690
- queue.h interface header file program code, 694-695
- queues
 - ADTs (abstract data types), 689-708
 - arrays, 691
 - circular, 692
 - emptying, 698
 - FIFO (first in, first out), 690
 - items, manipulating, 691, 694-697
- quick sort function (qsort() function), 650-654
- quotation marks
 - “ ” (double), 66, 91, 404, 628-629
 - files, 327, 473
 - macros, 619
 - printf() function, 33
 - ‘ ’ (single), 66, 85

R

- r drive1.c driver program, code, 469
- r drive2.c program, code, 470
- \r (escape character), 66-67, 82
- “r” (mode string), 498
- “r+” (mode string), 498
- %r (strftime function() format specifier), 841
- ragged array, 404
- rain.c program, code, 355-356
- raise() function, 817
- RAM (random-access memory), 5
- rand() function, 468-471, 475
- rand0() function, r drive1.c driver program, code, 469
- rand0.c function file program, code, 468
- rand1() function, 470, 475
- randbin.c program, 518-520
- random access
 - array elements, 708
 - files, 506-510, 518-520
- random numbers, 468-475
- random-access memory (RAM), 5
- RAND_MAX constant, 828
- ranges, testing, 240
- “rb” mode string, 498
- read-only values, constants, 98
- readability of programs, 35-36, 565-569
- reading
 - characters, first of a line, 254
 - input, scanf() function, 119-120
 - strings, 397-399
 - text files, 498

- reading loop (while loops), *summing.c* program, 172
- real floating points, 78
- real numbers, 54
- records, fields, 557
- rectangular arrays, 404
- rectangular coordinates, 662
- recur.c* program, code, 318-319
- recursion
 - algorithms, calculating binary equivalents of integers, 323-324
 - functions, 318-325
 - loops, comparing, 318
- .red file extension, 501
- redefining constants, *#define* preprocessor directive, 620-621
- redirection
 - files, 493-494
 - operators, < or >, 275-277
 - standard input or output, 274-279
- reducto.c* program, 501-503
- reference sources
 - Algorithms in C: Fundamentals, Data Structures, Sorting, Searching*, 783
 - ANSI C I/O functions, 824-827
 - ANSI C library, 800
 - assert.h* header file, 801
 - C99 additions, 801-811
 - complex.h* header file, 801-803
 - ctype.h* header file, 803-804
 - errno.h* header file, 804
 - fenv.h* header file, 805-807
 - inttypes.h* header file, 807-808
 - locale.h* header file, 808-811
 - stdlib.h* header file, 827-833
 - string functions, 417, 430-432
 - string.h* header file, 834-837
 - tgmath.h* header file, 837-838
 - time.h* header file, 838-842
 - Unix I/O functions, 267
 - wchar.h* header file, 842-849
 - wctype.h* header file, 849-852
 - Art of Computer Programming (The)*, Volume 1, 783
 - Boolean support, *stdbool.h* header file, 819
 - C: A Reference Manual*, Fourth Edition, 783
 - C and C++, comparing, 864-869
 - C Programming FAQs*, 783
 - C Programming Language (The)*, Second Edition, 782
 - C Puzzle Book (The)*, 782
 - C Traps and Pitfalls*, 783
 - C++ Primer Plus*, Fourth Edition, 784
 - C++ Programming Language (The)*, Third Edition, 784
 - C/C++ Users Journal*, 781, 860-864
 - C99 numeric computations, 860
 - character support, 856-860
 - data types, 790-792
 - definitions, *stddef.h* header file, 820
 - do while statements, 797
 - Elements of Programming Style (The)*, 783
 - expressions, 795-787
 - extended integer types, 852-855
 - for statements, 796
 - free()* function, 793
 - I/O library, 824-827
 - if statements, 797-798
 - integer types, *stdint.h* header file, 820-824
 - integers (extended types), 852-855
 - International C Standard (The)* (ISO/IEC 9899 1999), 783
 - malloc()* function, 793
 - Math library, 811-818
 - memory, allocated, 793
 - numeric computations (C99), 860-864
 - online, 781-782
 - operators, 784-790
 - program jumps, 799-800
 - Reliable Data Structures in C*, 783
 - Standard C Library (The)*, 783
 - statements, 795
 - stdarg.h* header file, 818-824
 - stdbool.h* header file, 819
 - stddef.h* header file, 820
 - stdint.h* header file, 820-824
 - stdio.h* header file, 824-827
 - stdlib.h* header file, 827-833
 - storage classes, 792-793
 - string.h* header file, 834-837
 - switch statements, 798-799
 - tgmath.h* header file, 837-838
 - time.h* header file, 838-842
 - UCNs (universal character names), 858-859
 - variables, 792-794, 818-824
 - wchar.h* header file, 842-849
 - wctype.h* header file, 849-852
 - Web sites, 781-782
 - while statement, 795-796
- referencing
 - declarations, external variables, 463
 - uninitialized pointers, dereferencing, 369

- register
 - keyword, 453
 - specifier, 465
 - storage class, 453, 488, 793
 - variables, 457
- registers, CPUs (central processing units), 6
- relational
 - expressions, 173, 176-185, 236-240, 786
 - operator (==), 181-182
 - operators, 148, 176-183, 185, 785
- Reliable Data Structures in C*, 783
- removing. *See* deleting
- replacement lists, #define preprocessor directive lines, 617
- reporting errors, errno.h header file, 804
- representing floating-point numbers, 590
- reseeding, automated, 471
- reserved identifiers, 31, 43
- resources. *See* reference sources
- restrict keyword, 485-486, 793
- restricted keyword, 457
- return
 - keyword, 301, 311
 - statements, 34-35
 - types (functions), 27, 340
- return values
 - assigning to variables, 312
 - functions, 206-210
 - printf() function, 114-115
 - scanf() function, 121
 - strcmp() function, 422-424
- returning
 - characters or pointers to strings, 431
 - values, 26, 310-313
- reversal and recursion (functions), 323-324
- reverse.c program code, 507
- rewind()
 - command, 561
 - function, addaword.c program code, 503-504
- rhodium.c program, 49-51
- right shift operator (>>), 597
- Ritchie, Dennis M., 1, 18, 782
- rollem() function, 473
- rolling dice, 471-475
- roll_n_dice() function, 473
- round-off errors, floating-point numbers, 76
- routines of libraries, 9, 12
- rows, creating, 202
- rows1.c program, code, 201-202
- rows2.c program, code, 202-203
- rules (association), operators, 140

- rules.c program code, 140-141
- running programs in environments, 10
- running.c program, code, 161-162
- rvalues, 133

S

- %s
 - conversion specifier, 90, 102, 118
 - specifier, scanf() function, 289
- s and t.c program, code, 469-470
- %S (strftime function() format specifier), 841
- saving structure contents to files, 557-561
- scalar variables (single-valued variables), 346
- scan str.c program, code, 411
- scanf() function, 81, 101, 116, 271, 295
 - & (ampersand), 51, 77
 - * modifier, 121-123
 - %C specifier, 289
 - conversions, 117-119
 - %d specifier, 289
 - EOF (end of files), 272
 - %f specifier, 289
 - format string characters, 120-121
 - input, reading, 119-120
 - input.c program, code, 117
 - interactive programs, 51
 - loops, 171
 - null characters, 92
 - return values, 121
 - %s specifier, 289
 - skip2.c program, code, 122-123
 - string input, 410-412
 - summing.c program, 170-172
 - varwid.c program, code, 121-122
 - whitespace characters, 124
 - whitespaces, 93, 117
- scientific notations, 72
- scope (variables), 449-451, 457-459
- scores.c program, code, 204-206
- screens
 - output, book convention, 20
 - printing to, 24
- searches
 - binary, 710-735
 - sequential, linked list elements, 709
- Sedgewick, Robert, 783
- seeds, automated reseeding, 471
- SEEK_CUR mode (files), 508
- SEEK_END mode (files), 508

- SEEK SET mode (files), 508
- SeekItem() function, 719
- SeekNode() function, 724
- selection sort algorithm, sorting pointers, 435
- selection statements. *See* if statements
- semantic errors, 41-42
- semicolon (;), 35-36, 795
 - assignment statement, 32
 - functions, 304
 - statements, 151, 155
 - structure member declarations, 529
- sequences
 - access, linked list elements, 708
 - points, 153-154
 - searches, linked list elements, 709
 - statements, branching, 169
 - trigraph (characters), 856
- setbuf() function, 270
- setjmp.h header file, 817
- setvbuf() function, 270, 516
- shared namespaces, 568-569
- shift operators (bitwise), 596-598
- shoes1.c program, code, 129-130
- shoes2.c program, code, 130-131, 144-145
- short
 - int types, 77, 791
 - keyword, 53, 59
 - signed integer, 791
 - types, 60-64, 77
- shortcomings of C, 4
- shortcuts, keyboard
 - Ctrl+D, 276
 - Ctrl+Z, 271, 274-276
- show() function, 575-579
- showchar1.c program, code, 282
- showchar2.c program, code, 283
- showfpt.c program, code, 74-75
- showinfo() function, 548
- showmenu() function, 579
- showmovies() function, 687
- show_n_char() function, 306-308
- side effects, sequence points, 153-154
- SIGABRT macro, 817
- SIGFPE macro, 817
- SIGILL macro, 817
- SIGINT macro, 817
- sign off() function, 649
- sign operators, 134, 788
- sign-magnitude (numbers), 589
- signal handling, 817-818
- signal() function, 817-818
- signal.h header file, 817-818
- signals, functions, 818
- signed
 - char types, 69
 - int, bit fields, 601
 - integers, 77, 589, 791
 - keyword, 53, 60, 77
 - numbers, 589
- signs (two's-complement binary numbers), reversing, 589
- SIGSEGV macro, 817
- SIGTERM macro, 817
- SIG_ATOMIC_MAX constant, 824
- SIG_ATOMIC_MIN constant, 824
- SIG_DFL macro, 818
- SIG_ERR macro, 818
- SIG_IGN macro, 818
- simple C program, 23-25
- simple statements, 155, 795
- simulating with ADTs (abstract data types) queues, 702-708
- simulations
- single quotation mark (' '), 66
- single-character I/O, 268
- single-valued variables, 346
- sites. *See* Web sites
- size_t type, 142, 432
- sizeof operator, 80, 95, 142, 159, 790
- sizeof.c program, code, 142
- sizes
 - arrays, 347, 353-35
 - buffers, 269
 - data types, 79-80
 - VLAs (variable-length arrays), 384
- SIZE_MAX constant, 824
- size_t fread(void * restrict, size_t, size_t, FILE * restrict) function, 825
- size_t fread(void *ptr, size_t size, size_t nmemb, FILE *fp) function, 515
- size_t fwrite(const void * restrict, size_t, size_t, FILE * restrict) function, 825
- size_t fwrite(void *ptr, size_t size, size_t nmemb, FILE *fp) function, 514-515
- size_t mbrlen(const char * restrict s, size_t n, mbstate_t * restrict ps) function, 847
- size_t mbrtowc(wchar_t * restrict pwc, const char * restrict s, size_t n, mbstate_t * restrict ps) function, 847
- size_t mbsrtowcs(wchar_t * restrict dst, const char ** restrict src, size_t len, mbstate_t * restrict ps) function, 848

- `size_t` `mbstowcs(wchar_t * restrict pwcs, const char *s restrict, size_t n)` function, 833
- `size_t` `strcspn(const char *s1, const char *s2)` function, 836
- `size_t` `strftime(char * restrict s, size_t max, const char * restrict fmt, const struct tm * restrict tmpt)` function, 840
- `size_t` `strlen(const char * s)` function, 431
- `size_t` `strspn(const char *s1, const char *s2)` function, 836
- `size_t` `strxfrm(char * restrict s1, const char * restrict s2, size_t n)` function, 835
- `size_t` type, 820, 827, 838, 842
- `size_t` `wcrtomb(char * restrict s, wchar_t wc, mbstate_t * restrict ps)` function, 848
- `size_t` `wcsrtombs(char * restrict dst, const wchar_t ** restrict src, size_t len, mbstate_t * restrict ps)` function, 849
- `size_t` `wcstombs(char * restrict s, const wchar_t * restrict pwcs, size_t n)` function, 833
- `skip.c` program, code, 246-247
- `skip2.c` program, code, 122-123
- slashes
 - `\` (backslash), 34, 66-67
 - `/` (forward slash), 226
- software developers, 4
- `somdata.c` program, code, 348-349
- `sort str.c` program, code, 432-433
- sorting
 - algorithms, 652
 - pointers, 435
 - strings, 432-434
- source code
 - compiling, 9, 639-640
 - converting to executable files, 12
 - files, 327, 630-631
 - preprocessing, 27
 - writing, 9
- sources. *See* reference sources
- space flag, `printf()` function, 107
- spaces
 - macro names, 628
 - string input, 407
 - whitespaces, 93
- specifications
 - array sizes, 353-354
 - conversions, 90, 101-110, 117-118
- specifiers
 - `auto`, 464
 - `%C`, 289
 - `%c` format, 68
 - `%d`, 68, 289
 - `extern`, 465
 - `%f`, 51, 289
 - formats, 840-842
 - matching, 64
 - register, 465
 - `%s`, 289
 - static, 465
 - storage classes, 453, 464-465
 - `typedef`, 464
 - `%u`, 61
- speed of programs, macros, 628
- spelling alternatives
 - C and C++, comparing, 868
 - characters, 857-858
 - logical operators, 237-238
- `sprintf()` function, 429-430, 440
- `sqrt()` (square root) function, 646
- `SQUARE` macro, 622
- `squares.c` program, code, 135-136
- `srand1()` function, 469-470
- `strlen()` function, 93-95
- standard ANSI C library, 800-811
- Standard C Library (The)*, 783
- standard error output, files, 495-496
- standard files, pointers, 501
- standard high-level files, 495
- standard I/O
 - binary I/O, 513, 518-520
 - command-line arguments, 497
 - `count.c` program code, 496-497
 - EOF (end of file), 499-500
 - `fclose()` function, 500-501
 - files, pointers, 501
 - `fopen()` function, 498-499
 - functions, 511-512, 515, 518
 - `getc()` function, 499
 - input, 274-276, 279, 495-496
 - input/output header (`stdio.h` file), 24-27, 268, 824-827
 - `int feof(FILE *fp)` function, 515
 - `int ferror(FILE *fp)` function, 515
 - `int fflush(FILE *fp)` function, 512
 - `int setvbuf(FILE *fp, char *buf, int mode, size_t size)` function, 512-513
 - `int ungetc(int c, FILE *fp)` function, 512
 - library, 824-827
 - operation, 511
 - output, 275-279, 495-496
 - package, 271, 498
 - `putc()` function, 499

- size_t fread(void *ptr, size_t size, size_t nmemb, FILE *fp) function, 515
- size_t fwrite(void *ptr, size_t size, size_t nmemb, FILE *fp) function, 514-515
- standard math functions (ANSI C), 812-816
- standards
 - ANSI C, 18
 - C language, 18-19
 - C90, 18
 - C99, 19, 118
 - data types, C9X changes, 854
 - ISO C, 18
 - ISO/ANSI C standard, 19
 - ISO/ANSI C90, keywords, 43
 - ISO/ANSI C99, keywords, 43
- starbar identifier, 303
- starbar()
 - function, 304-305
 - statement, 304
- starsrch.c program, code, 424-425
- start-up code, 12
- statements
 - { } (curly braces), 26, 29
 - ; (semicolon), 36, 151, 155, 795
 - addemup.c program, code, 152
 - assignment, 32, 35, 152, 157, 795
 - blocks, 29, 131, 795
 - break, 249-255
 - compound, 154-156, 795
 - continue, 246-249
 - control, 795
 - declaration, 29-30, 35, 795
 - #define, 98
 - do while, 200, 797
 - else, pairing with if else statements, 231-232
 - else if, 228-230
 - expression statements, 151
 - for, 192, 796
 - function, 152
 - function call, 35, 795
 - goto, 257-261
 - if, 220-223, 232, 235, 260, 797-798
 - if else, 222-235, 244-245, 256, 260
 - #include, 26-27
 - init, main() function, 27
 - int, 25
 - jump (break and continue statements), 246-255
 - null, 176, 795
 - print, 35
 - printf, 26
 - printf(), 83-84
 - return, 34-35
 - selection, 260
 - sequences, 169
 - side effects, sequence points, 153-154
 - simple, 155, 795
 - star(), 304
 - structured, 152
 - switch, 250-256, 290, 798-799
 - test expressions, 253
 - type conversions, 156-158
 - variables, defining, 26
 - void, 25
 - while, 152, 173-176, 185, 290, 421, 795-796
- states of programs, examining, 42-43
- static
 - keyword, 346, 452, 486-487, 792
 - memory, strings, 402
 - specifier, 465
 - storage class, 399, 452-453, 488
 - variables, 457-464, 468-471
- static with external linkage storage class, 793
- static with internal linkage storage class, 793
- static with no linkage storage class, 793
- static.c program, The (code listing), 458
- status flags, floating-point numbers, 861
- stdarg.h header file, variable arguments, 658-660, 818-824
- stdbool.h header file, 819
- STDC FP CONTRACT, 862
- STDC HOSTED macro, 638
- STDC macro, 638
- STDC VERSION macro, 638
- stddef.h header file, 820
- stderr file pointer, 501
- stdin
 - file pointer, 501
 - stream, 271
- stdint.h header file, integer types, 820-824
- stdio.h file, 24-27, 268, 824-827
- stdlib.h header file, 648, 827-833
- stdout
 - file pointer, 501
 - stream, 271
- Steele, Guy L., 783
- stillbad.c program, code, 41-42
- storage classes, 792
 - arrays, 348
 - automatic, 453, 488, 793
 - block scope, 450-451
 - duration, structure initialization, 531
 - dynamic memory allocation, 480-481

- file scope, 451
- files of code, 464
- functions, 451, 467-468
- memory allocation, 475-480
- parta.c file program, code, 465-466
- parth.c file program, code, 466
- register, 793
- register class, 453, 488
- specifiers, 453, 464-465
- static class, 453, 488
- static with external linkage, 793
- static with internal linkage, 793
- static with no linkage, 793
- variables, 449-467
- storage duration, 449, 452-455
- storage units, arrays, 359
- storage-class specifiers, 464-465
- storing
 - data, 69, 513
 - words in binary search trees, 712
- str cat.c program, code, 419
- strcat() function, 419-420, 483
- strchr() function, newline characters, 432
- strcmp() function, 420-424, 432-434
- strcnvt.c program, code, 442
- strcpy() function, 425-427
- streams
 - bytes, 295
 - input, 289-290, 512, 520
 - output, 520
 - standard I/O package, functions, 271
 - stdin, 271
 - stdout, 271
- strftime function(), format specifiers, 840-842
- string comparison (strcmp() function), 420-424, 432-434
- string concatenation (strcat() function), 419-420, 483
- string constants (string literals), defining character strings, 399-400
- string lengths (strlen() function), 90, 301, 417-418
- string.h header file, 94, 418, 834-837
- string.h library, 656-657
- stringizing macro arguments, 624
- strings, 81
 - arguments, 415, 437-440
 - arrays, 91
 - atoi() function, 441
 - char *strcat(char * s1, const char * s2) function, 431
 - char *strchr(const char * s, int c) function, 431
 - char *strcpy(char * s1, const char * s2) function, 430
 - char *strncat(char * s1, const char * s2, size_t n) function, 431
 - char *strncpy(char * s1, const char * s2, size_t n) function, 430
 - char *strpbrk(const char * s1, const char * s2) function, 431
 - char *strrchr(const char * s, int c) function, 431
 - char *strstr(const char * s1, const char * s2) function, 431
 - characters, 90, 397-399
 - arrays, 204
 - comparing, 93
 - functions, 435-437
 - returning, 431
 - string arrays, 400-405
 - string constants, 92, 399-400
 - comparing, 421-423, 431
 - concatenating, 431, 626
 - const keyword, 432
 - constants, static storage class, 399
 - control, 103-104
 - conversions, ANSI C functions, 441
 - convert.c program, code, 436
 - converting to numbers, 440-443
 - copying, 430
 - creating, 397-399, 624-625
 - ctype.h, 435-437
 - #define statement, 98
 - defining, 399
 - fit() function, 417-418
 - format, characters, 120-121
 - functions, 834-836
 - ANSI C library, 430-432
 - character, 435-437
 - convert.c program, code, 436
 - ctype.h, 435-437
 - fit(), test.c program, code, 417-418
 - gets, 408
 - puts(), null character, 418
 - sprintf(), 429-430
 - sprintf(), format.c program code, 429-430
 - strcat(), 419-420
 - strcmp(), 420-424
 - strcpy(), 425-427
 - strlen(), 417-418
 - strncat(), 419-420
 - strncmp(), 420-425
 - strncpy(), 425, 428-429

- handling, 834-837
- I/O, options, 414-417
- input, 406-411
- int strcmp(const char * s1, const char * s2) function, 431
- int strncmp(const char * s1, const char * s2, size_t n) function, 431
- lengths of (strlen() function), 93-95
- long, printing, 115-116
- loops, 204
- mode, fopen() function, 498
- output, 412-414
- pointers, 405-406, 431, 434-435
- praisel.c program, code, 92-93
- printing, 92, 397-399
- punctuation characters, counting, 436
- puts() function, null character, 418
- reading, 397-399
- size_t type, 432
- size_t strlen(const char * s) function, 431
- sorting alphabetically, 432-434
- sources, 426
- sprintf() function, 429-430
- static memory, 402
- strcat() function, 419-420
- strcmp() function, 420-424
- strcpy() function, 425-427
- strings.c program, code, 397-399
- strlen() function, 417-418
- strncat() function, 419-420
- strncmp() function, 420-425
- strncpy() function, 425, 428-429
- strtod() function, 441
- strtol() function, 441-443
- strtoul() function, 441
- targets, 426
- toupper() function, string characters, 435, 437
- wide-character string utilities, 845-846
- strings.c program, code, 109-110, 397-399
- stringy, binary search trees, 735
- strlen() function, 90, 301, 417-418
- strncat() function, join chk.c program, code, 419-420
- strncmp() function, 420-425
- strncpy() function, 425, 428-429
- Stroustrup, Bjarne, 784
- strtod() function, 441
- strtok() function, 837
- strtol() function, 441-443
- strtoul() function, 441
- struct keyword, 529, 540
- struct lconv * localeconv(void) function, 808
- struct lconv macros, 809-811
- struct lconv structure members, 810
- struct tm *gmtime(const time_t *ptm) function, 840
- struct tm *localtime(const time_t *ptm) function, 840
- struct tm structure members, 838-839
- struct tm type, 838, 843
- structured statement, 152
- structures, 548
 - . (dot) operator, 541
 - addresses, 540, 543-544
 - allocating in blocks or individually, comparing, 669
 - arrays, 533-537, 556-557
 - binary trees, 561
 - book inventory, 527-529
 - Borland C/C++, 533
 - C and C++, comparing, 867
 - character pointers and character arrays, comparing, 549-550
 - compound literals (C99), complit.c program code, 552-553
 - contents, saving to files, 557-561
 - data forms, creating, 561-562
 - declarations, 529
 - do while loops, 199
 - fields, 529
 - flexible array members (C99), 554-556
 - for loops, 187
 - functions, 306
 - initializers, 531
 - malloc() function, 550-552
 - members, 529, 532-533, 541-543
 - memory, allocating, 530
 - names1.c program code, 545-546
 - names2.c program code, 547
 - nested, 537-539
 - operators, 565, 788-789
 - passing as arguments, 544
 - pointers, 539-541, 550-552
 - programs, 34-35
 - records, fields, 557
 - struct lconv, members, 810
 - struct tm structure members, 838-839
 - structure pointers and structure arguments, comparing, 548-549
 - template definitions, header files, 631
 - unions, 562-565, 607
 - variables variables, 527, 530-532
 - while loops, 173
- styles, while loops, 155
- subexpressions, 150, 225

- subnormal (floating-point values), 75
- subscripts
 - arrays, 204, 351
 - numbers, array elements, 346
- subst.c program, code, 624
- substitutions, compile time, 96
- subtraction operator (-), 134, 159, 785
- subtrees
 - binary search trees, 712
 - child nodes, deleting, 721-722
- sum arr1.c program, code, 363-364
- sum arr2.c program, code, 364-365
- sum() function, 363-365
- summing.c program, 170-172
- Summit, Steve, 783
- sump() function, 365
- sun squares() function, 288
- swap1.c program, code, 332-333
- swap2.c program, code, 333
- swap3.c program, code, 336-337
- swapping variables in calling functions, 332-334
- sweetie1.c program, code, 186
- sweetie2.c program, code, 187
- switch
 - keyword, 798
 - statements, 250-256, 290, 798-799
- symbolic constants, 95-100, 347, 620. *See also* manifest constants
- symbols. *See* Symbols (index section)
- syntax
 - comments, 26
 - errors, 40-41
 - points, 175
- system header files, IDEs (integrated development environments), 629
- systems
 - book convention, 21
 - floating-point type sizes, 79
 - integer type sizes, 79
 - Linux, 15
 - UNIX, 13-14

T

- t and f.c program, code, 178-179
- \t (escape character), 66-67, 82
- t (modifier), printf() function, 106
- %t (strftime function()) format specifier, 841
- /t (tab character), 34

- tables
 - creating, nested loops, 202
 - operators, high to low precedence, 784-785
- tag names, structure declarations, 529
- tail recursion, 321-323
- taking a pointer address operation, 368
- talkback.c program, 89-91
- targets, 16, 426
- tasks, menus, 290
- templates, structure, 529, 631
- terminating
 - functions, 313
 - input, 268
 - keyboard input, 270-274
 - programs, 497
 - while loops, 173-174
- test.c program, code, 417-418
- testing
 - ADTs (abstract data types) queues, 700-702
 - characters, functions, 227
 - conditions, 260
 - functions, 310
 - programs, 10
 - ranges, 240
- tests
 - conditions, concepts, 210
 - expressions, 222, 240
 - loops, 248
 - non-whitespaces, detecting, 241
 - switch statements, 253
 - whitespace, detecting, 241
- text, 306, 513, 616
 - files, 275, 498
 - mode and binary mode, comparing, 509
 - views, files, 494
- tgmath.h header file, 837-838
- Thompson, Ken, 1
- three-dimensional arrays, 358
- tilde (~)
 - bitwise unary operator, 789
 - unary operator, bitwise logical operator, 592-593
- time, 838-842
- TIME macro, 638
- time() function, 471
- time.h header file, 838-842
- time_t mktime(struct tm *tmptr) function, 839
- time_t time(time_t *ptm) function, 839
- time_t type, 838
- tooggling bits, 595-596
- tokens, 616, 620, 625
- ToLeft() function, 717-718

- tolower() function, 227
- too bad() function, 649
- ToRight() function, 717-718
- toupper() function, 228, 435-437
- ToUpper() function, 573
- tracing programs, 42
- transformations, wide-character functions, 851-852
- translations, program preprocessing, 616
- Traverse() function, 681, 687, 724
- traversing binary search trees, 724-725
- tree.c file, 717
- tree.c implementation file code, 725-729
- tree.h interface header file program code, 714-716
- trees
 - AVL, 735
 - binary, 561, 711-735
 - expression, 139
- trends of C, 4
- trigraph sequences (characters), 856
- trouble.c program, code, 180-182
- troubleshooting, debugging programs, 10, 39-43, 654-656
- true macro, 819
- true values of expressions, 178-180
- true variables, Bool type, 182
- truncating text files, 498
- truncation, 81, 137
- truth.c program, code, 179-180
- trystat() function, 458
- turning bits on or off, 595
- two-child nodes, deleting, 722
- two-dimensional arrays, 355, 358, 390
- two-dimensional functions, applying to arrays, 383
- two func.c program, code, 38-39
- two's-complement method, 589
- .txt file extension, 17
- (type) operators, 159, 790
- type-generic math, 837-838
- typedef
 - constants, 823-824
 - exact width types, 820-821
 - fastest minimum width types, 821-822
 - integers, 822-824
 - Item type, 690
 - keyword, 464
 - maximum width types, 822
 - minimum width types, 821
 - Queue type, 690
 - specifier, 464
 - types, creating names, 569-571
- typeface, book convention, 20
- types. *See also* ADTs
 - Boolean (C99), 791, 868
 - char constants, comparing C and C++, 865-866
 - clock_t, 838
 - complex, comparing C and C++, 868
 - conversions, 156-158
 - data, defining abstract to concrete, 677
 - defining, 631, 676
 - stdlib.h header file, 827
 - time.h header file, 838
 - wchar.h header file, 842-843
 - div_t, 827
 - double (floating-point numbers), 792
 - enumerated, 565-569
 - exact width, 820-821, 853
 - extended integers, 852-855
 - fastest minimum width, 821-822, 854
 - fenv_t, 805
 - fenv.h header file, 805
 - fexcept_t, 805
 - float (floating-point numbers), 791
 - functions, 304, 313-314
 - int8_t, 820
 - int16_t, 820
 - int32_t, 821
 - int64_t, 821
 - integers, 820-824, 852-855
 - intmax_t, 822
 - intptr_t, 822
 - int_fast8_t, 822
 - int_fast16_t, 822
 - int_fast32_t, 822
 - int_fast64_t, 822
 - int_least8_t, 821
 - int_least16_t, 821
 - int_least32_t, 821
 - int_least64_t, 821
 - Item, 678, 690
 - ldiv_t, 827
 - lldiv_t, 827
 - long double (floating-point numbers), 792
 - maximum width, 822, 855
 - mbstate_t, 843
 - minimum width, 821, 853-854
 - names, creating with typedef, 569-571
 - ptrdiff_t, 820
 - qualifiers, ANSI C, 481-486
 - Queue, 690
 - size_t, 142, 820, 827, 838, 842

stddef.h header file, 820
 struct tm, 838, 843
 time_t, 838
 ToUpper() function, 573
 typedef, 569-571
 uint8_t, 821
 uint16_t, 821
 uint32_t, 821
 uint64_t, 821
 uintmax_t, 822
 uintptr_t, 822
 uint_fast8_t, 822
 uint_fast16_t, 822
 uint_fast32_t, 822
 uint_fast64_t, 822
 uint_least8_t, 821
 uint_least16_t, 821
 uint_least32_t, 821
 uint_least64_t, 821
 va_arg(va_list ap, type) macro, 819
 void (*)(int) macros, 818
 void functions, 650
 wchar_t, 820, 827, 842
 wctrans_t, 850
 wctype.header file, 849-850
 wctype_t, 850
 wint_t, 842, 849
 typesize.c program, code, 79-80

U

%u
 conversion specifier, 102, 118
 specifier, printf() function, 61
 %U (strftime function() format specifier), 841
 UCNs (universal character names), 858-859
 uint8_t type, 821
 uint16_t type, 821
 uint32_t type, 821
 uint64_t type, 821
 UINTMAX_MAX constant, 823
 uintmax_t strtoumax(const char * restrict nptr, char **
 restrict endptr, int base) function, 808
 uintmax_t type, 822
 uintmax_t wcstoumax(const wchar_t * restrict nptr,
 wchar_t ** restrict endptr, int base) function, 808
 UINTN_MAX constant, 823
 UINTPTR_MAX constant, 823
 uintptr_t type, 822

uint_fast8_t type, 822
 uint_fast16_t type, 822
 uint_fast32_t type, 822
 uint_fast64_t type, 822
 UINT_FASTN_MAX constant, 823
 uint_least8_t type, 821
 uint_least16_t type, 821
 uint_least32_t type, 821
 uint_least64_t type, 821
 UINT_LEASTN_MAX constant, 823
 unary operators, 134, 784, 789-790
 & (ampersand), finding addresses, 330-332
 *, 366
 ++, 366
 -, 159, 785
 ~ (bitwise logical operator), 592-593
 unbalanced binary search trees, 735
 unbuffered input and buffered input, comparing, 269
 #undef preprocessor directive, 632-633
 underflows, floating-point numbers, 75
 underscore (_), naming variables, 31
 unechoed input, 270
 unequal (!=) relational operator, 785
 ungetc() function, 512
 uninitialized pointers, dereferencing, 369
 unions, 562
 . (dot operator), 563-564
 -> (indirect membership operator), 565
 as a structure, 607
 as an integer, 607
 C and C++, comparing, 867
 initializing, 563
 operators, 788-789
 variables, defining, 563
 United States Postal Service Web site, 193
 universal character names (UCNs), 858-859
 Unix
 >> (operator), 277
 > (redirection operator), 276-277
 < (redirection operator), 275-277
 | (pipe) operator, 277
 buffering functions, 270
 C compilers, 3
 cc command, 326
 cc compiler, 14
 compiling on, 14
 Ctrl+D keyboard shortcut, 276
 echo eof.c program, 273
 editing on, 13-14
 functions, compiling, 326
 I/O functions, 267

- input, redirecting, 275-276, 279
- ioctl() function, 270
- make command, 326
- output, redirecting, 275-279
- programs, running, 10
- UNIX Primer Plus, Third Edition*, 277
- unsigned
 - char types, 69
 - int, bit fields, 601, 606
 - int types, 60
 - int values, printf() function, 61
 - integers, 77, 791
 - keyword, 53, 59, 791
 - long long types, 61
 - short types, 60
 - types, 61-64
- unsigned long long strtoull(const char * restrict npt, char ** restrict ept, int base) function, 829
- unsigned long strtoul(const char * restrict npt, char ** restrict ept, int base) function, 829
- updating text files, 498
- use qc program code, 700-701
- useheader.c program code, 630-631
- usehotel.c control module, code, 327-328
- user interfaces
 - buffered input, 279-281
 - character input, 281-284
 - creating, 279
 - menus, 290-295
 - numeric input, 281-284
- utilities, wide character, 842-852. *See also* ANSI C, library; general utilities library

V

- \w (escape character), 66-67
- %V (strftime function() format specifier), 841
- VA_ARGS variadic macro, 626-627
- va_copy() macro, 659
- va_end() macro, 659
- validating input, 268, 284-290, 295
- value variables, 339
- value-assigning operators, = (assignment operator), 132
- value-finding (dereferencing) operation, pointers, 368
- va_along ftell(FILE *) function, 825
- values
 - arguments, 818-824
 - argv (argument values), 439
 - arrays, 351-352

- assigned, 566-567
- bits, 588, 596, 601
- default, 566
- expressions, 150-151, 178-180
- fathm ft.c programs, printing, 37-38
- floating-point, showfpt.c program, code, 74-75
- from functions, returning, 310-313
- functions, return types, 340
- indexes, changing, 173
- int type, printf.c program, code, 57-58
- lvalues, data objects, 133
- modifiable lvalue, 133
- pointer (integers), 822, 855
- printing, 37-38
- return
 - printf() function, 114-115
 - scanf() function, 121
 - strcmp() function, 422-424
- returning, 26
- rvalues, 133
- unsigned int, printf() function, 61
- variables, 26, 42-43
- varargs.c program code, 659-660
- vararr2d.c program code, 385-386
- variable-length arrays, dynamic memory allocation, 480-481
- variable-length arrays (VLAs), 354, 383-387, 480
- variables, 52
 - , (comma), 56, 339
 - actual arguments, 160
 - addresses, 339
 - altering in calling functions, 332-334
 - arguments, 658-660
 - arrays, comparing declarations, 91
 - automatic, 452-456
 - blocks, 450-454
 - Boolean, 182
 - char types, declaring, 64-65
 - comma separated lists, 308
 - const keyword, 793
 - count, 305
 - declaring, 29-32, 77-78
 - defining, statements, 26
 - external, 452, 459-463, 632
 - fathm ft.c program, 37
 - file scope, 451
 - flags, 234
 - floating-point, 51, 73
 - formal arguments or parameters, 160
 - function scope, 451
 - global, 451

- hiding.c program, code, 454-455
- identifiers, 30
- index, 452
- int type, 56-57
- internal linkage, 452
- linkage, 449, 452
- local, 305
- names, 31, 339
- no linkage, 452
- num, 26, 29
- number, 452
- pointers, 334-339, 359, 367-369
- private names, 160
- qualifying, 793-794
- recursion, 320
- register, 457
- restrict keyword, 793
- return values, assigning, 312
- scope, 449-451
- single-values, 346
- static, 452-453, 457-464, 468-471
- storage classes, 449-467, 792-793
- structures, 527, 530-532
- swapping in calling functions, 332-334
- true and false, 182
- unions, defining, 563
- values, 26, 42-43, 339
- volatile keyword, 793
- variadic
 - functions, variable arguments, 658-660
 - macros, 626-627
- variadic.c program code, 626
- varwid.c program, code, 121-122
- va_arg() macro, 658
- va_start() macro, 658
- viewpoints (black box), functions, 310
- views, binary or text, 494
- VLAs (variable-length arrays), 354, 383-387, 480
- VMS, running programs, 10
- void (*f)(int) macros, 818
- void (*signal(int sig, void (*func)(int)))(int) function, 818
- void (main() function), 25
- void *bsearch(const void *key, const void *base, size_t nmem, size_t size, int (*comp)(const void *, const void *)) function, 831
- void *calloc(size_t nmem, size_t size) function, 829
- void *malloc(size_t size) function, 830
- void *memchr(const void *s, int c, size_t n) function, 834

- void *memcpy(void * restrict s1, const void * restrict s2, size_t n) function, 834
- void *memmove(void *s1, const void *s2, size_t n) function, 834
- void *memset(void *s, int v, size_t n) function, 834
- void *realloc(void *ptr, size_t size) function, 830
- void abort(void) function, 830
- void assert(int exprs) macro, 801
- void clearer(FILE *) function, 824
- void exit(int status) function, 831
- void feclearexcept(int excepts) function, 806
- void fegetenv(fenv_t *envp) function, 806
- void fegetexceptflag(fexcept_t *flagp, int excepts) function, 806
- void feraiseexcept(int excepts) function, 806
- void fesetenv(const fenv_t *envp) function, 807
- void fesetexceptflag(const fexcept_t *flagp, int excepts) function, 806
- void feupdateenv(const fenv_t *envp) function, 807
- void free(void *ptr) function, 830
- void functions, 304
- void longjmp(jmp_buf env, int val) function, 817
- void perror(const char *) function, 826
- void qsort(void *base, size_t nmem, size_t size, int (*comp)(const void *, const void *)) function, 832
- void rewind(FILE *) function, 826
- void setbuf(FILE * restrict, char * restrict) function, 826
- void srand(unsigned int seed) function, 829
- void statement, 25
- void type, functions, 304
- void va_copy(va_list dest, va_list src) macro, 819
- void va_end(va_list ap) macro, 819
- void va_start(va_list ap, parmN) macro, 819
- void _Exit(int status) function, 831
- volatile
 - keyword, 481, 484, 793
 - type qualifier, 484-485
- vowels.c program, code, 254-255

W

- “w” mode string, 498
- %w (strftime function() format specifier), 841
- “w+” mode string, 498
- “w+b” mode string, 498
- warnings, error messages, 50
- “wb” mode string, 498
- “wb+” mode string, 498
- wchar.h header file, 842-849

- WCHAR_MAX constant or macro, 824
- WCHAR_MIN
 - constant, 824
 - macro, 843
- wchar_t type, 820, 827, 842
- wctrans_t type, 850
- wctype.h header file, 849-852
- wctype_t type, 850
- Web sites
 - gcc compiler, 15
 - reference sources, 781-782
 - United States Postal Service, 193
- WEOF
 - constant expression, 850
 - macro, 843
- wheat.c program, code, 136-137
- while keyword, 795-797
- while loops (entry-condition loops), 152
 - { } (curly braces), 131
 - compound statements (blocks), 154-155
 - conditional loops, 174-175
 - entry.c program, code, 198-199
 - relational expressions or operators, 176-183
 - shoes2.c program, code, 130-131
 - single-character I/O, 268
 - structure, 173
 - summing.c program, 170-172
 - sweetie1.c program, code, 186
 - switch statements, 290
 - syntax points, 175
 - terminating, 173-174
 - values, processing, 288
 - when.c program, code, 174
 - while statements, 173, 185
 - while1.c program, code, 175
 - while2.c program, code, 175-176
- while statements, 152, 185, 795-796
 - abbreviating, 421
 - accessing menus, 290
 - entry-condition loops, 795
 - keywords, 795
 - while loops, 173-176
- while1.c program, code, 175
- while2.c program, code, 175-176
- whitespaces, 93
 - characters, scanf() function, 124
 - non-whitespaces, test to detect, 241
 - scanf() function, 117
 - test to detect, 241
 - text, breaking into sequences, 616
- wide characters, 860
 - classification, 849-852
 - extensible classification functions, 850-851
 - I/O functions, 843-844
 - mapping utilities, 849-852
 - multibyte conversion functions, 846-849
 - null wide, 859
 - string utilities, 845-846
 - support, comparing C and C++, 868
 - transformation functions, 851-852
 - utilities, 842-849
- width.c program, code, 107-108
- widths
 - exact width types, 820-821, 853
 - fastest minimum width types, 70, 821-822, 854
 - greatest width integer functions, 807-808
 - maximum width types, 822, 855
 - minimum width types, 70, 821, 853-854
- Windows
 - compilers, compiling functions, 327
 - IDEs (Integrated Development Environments), 15-17
- WINT_MAX constant, 824
- WINT_MIN constant, 824
- wint_t type, 842, 849
- word-count programs, 240-244
- wordcnt.c program, code, 242-244
- words, 53-54, 712
- writing
 - binary data to files, 514-515
 - C source code, 9
 - constants with int types, 68
 - programs, 6
 - text files, 498

X-Y-Z

- %X (conversion specifier), 102, 118
- %x (strftime function() format specifier), 841
- \xhh (escape character), 66-67
- %Y (strftime function() format specifier), 841
- z (modifier, printf() function), 106
- %z (strftime function() format specifier), 842
- Zeno, 196
- zeno.c program, code, 196-197
- zippo1.c program, 376-377
- zippo2.c program, code, 378