Developer’s Library

ESSENTIAL REFERENCES FOR PROGRAMMING PROFESSIONALS

Developer’s Library books are designed to provide practicing programmers with unique, high-quality references and tutorials on the programming languages and technologies they use in their daily work.

All books in the Developer’s Library are written by expert technology practitioners who are especially skilled at organizing and presenting information in a way that’s useful for other programmers.

Key titles include some of the best, most widely acclaimed books within their topic areas:

- **PHP & MySQL Web Development**
  Luke Welling & Laura Thomson
  MySQL
  Paul DuBois
  **Linux Kernel Development**
  Robert Love
  **Python Essential Reference**
  David Beazley
  **PostgreSQL**
  Korry Douglas
  **C++ Primer Plus**
  Stephen Prata

Developer’s Library books are available in print and in electronic formats at most retail and online bookstores, as well as by subscription from Safari Books Online at safari.informit.com

Developer’s Library
informit.com/devlibrary
C Primer Plus

Sixth Edition

Stephen Prata

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City
Contents at a Glance

Preface xxvii

1 Getting Ready 1
2 Introducing C 27
3 Data and C 55
4 Character Strings and Formatted Input/Output 99
5 Operators, Expressions, and Statements 143
6 C Control Statements: Looping 189
7 C Control Statements: Branching and Jumps 245
8 Character Input/Output and Input Validation 299
9 Functions 335
10 Arrays and Pointers 383
11 Character Strings and String Functions 441
12 Storage Classes, Linkage, and Memory Management 511
13 File Input/Output 565
14 Structures and Other Data Forms 601
15 Bit Fiddling 673
16 The C Preprocessor and the C Library 711
17 Advanced Data Representation 773

Appendixes
A Answers to the Review Questions 861
B Reference Section 905
Index 1005
# Table of Contents

**Preface**  xxvii

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  

2. **Language Standards**  7  
   The First ANSI/ISO C Standard  8  
   The C99 Standard  8  
   The C11 Standard  9  

3. **Using C: Seven Steps**  9  
   Step 1: Define the Program Objectives  10  
   Step 2: Design the Program  10  
   Step 3: Write the Code  11  
   Step 4: Compile  11  
   Step 5: Run the Program  12  
   Step 6: Test and Debug the Program  12  
   Step 7: Maintain and Modify the Program  13  
   Commentary  13  

4. **Programming Mechanics**  13  
   Object Code Files, Executable Files, and Libraries  14  
   Unix System  16  
   The GNU Compiler Collection and the LLVM Project  18  
   Linux Systems  18  
   Command-Line Compilers for the PC  19  
   Integrated Development Environments (Windows)  19  
   The Windows/Linux Option  21  
   C on the Macintosh  21
How This Book Is Organized 22
Conventions Used in This Book 22
    Typeface 22
    Program Output 23
    Special Elements 24
Summary 24
Review Questions 25
Programming Exercise 25

2 Introducing C 27
    A Simple Example of C 27
    The Example Explained 28
        Pass 1: Quick Synopsis 30
        Pass 2: Program Details 31
    The Structure of a Simple Program 40
    Tips on Making Your Programs Readable 41
    Taking Another Step in Using C 42
        Documentation 43
        Multiple Declarations 43
        Multiplication 43
        Printing Multiple Values 43
    While You’re at It—Multiple Functions 44
    Introducing Debugging 46
        Syntax Errors 46
        Semantic Errors 47
        Program State 49
    Keywords and Reserved Identifiers 49
    Key Concepts 50
Summary 51
Review Questions 51
Programming Exercises 53

3 Data and C 55
    A Sample Program 55
        What’s New in This Program? 57
    Data Variables and Constants 59
    Data: Data-Type Keywords 59
        Integer Versus Floating-Point Types 60
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Integer</td>
<td>61</td>
</tr>
<tr>
<td>The Floating-Point Number</td>
<td>61</td>
</tr>
<tr>
<td>Basic C Data Types</td>
<td>62</td>
</tr>
<tr>
<td>The int Type</td>
<td>62</td>
</tr>
<tr>
<td>Other Integer Types</td>
<td>66</td>
</tr>
<tr>
<td>Using Characters: Type char</td>
<td>71</td>
</tr>
<tr>
<td>The _Bool Type</td>
<td>77</td>
</tr>
<tr>
<td>Portable Types: stdint.h and inttypes.h</td>
<td>77</td>
</tr>
<tr>
<td>Types float, double, and long double</td>
<td>79</td>
</tr>
<tr>
<td>Complex and Imaginary Types</td>
<td>85</td>
</tr>
<tr>
<td>Beyond the Basic Types</td>
<td>85</td>
</tr>
<tr>
<td>Type Sizes</td>
<td>87</td>
</tr>
<tr>
<td>Using Data Types</td>
<td>88</td>
</tr>
<tr>
<td>Arguments and Pitfalls</td>
<td>89</td>
</tr>
<tr>
<td>One More Example: Escape Sequences</td>
<td>91</td>
</tr>
<tr>
<td>What Happens When the Program Runs</td>
<td>91</td>
</tr>
<tr>
<td>Flushing the Output</td>
<td>92</td>
</tr>
<tr>
<td>Key Concepts</td>
<td>93</td>
</tr>
<tr>
<td>Summary</td>
<td>93</td>
</tr>
<tr>
<td>Review Questions</td>
<td>94</td>
</tr>
<tr>
<td>Programming Exercises</td>
<td>97</td>
</tr>
<tr>
<td>4 Character Strings and Formatted Input/Output</td>
<td>99</td>
</tr>
<tr>
<td>Introductory Program</td>
<td>99</td>
</tr>
<tr>
<td>Character Strings: An Introduction</td>
<td>101</td>
</tr>
<tr>
<td>Type char Arrays and the Null Character</td>
<td>101</td>
</tr>
<tr>
<td>Using Strings</td>
<td>102</td>
</tr>
<tr>
<td>The strlen() Function</td>
<td>103</td>
</tr>
<tr>
<td>Constants and the C Preprocessor</td>
<td>106</td>
</tr>
<tr>
<td>The const Modifier</td>
<td>109</td>
</tr>
<tr>
<td>Manifest Constants on the Job</td>
<td>109</td>
</tr>
<tr>
<td>Exploring and Exploiting printf() and scanf()</td>
<td>112</td>
</tr>
<tr>
<td>The printf() Function</td>
<td>112</td>
</tr>
<tr>
<td>Using printf()</td>
<td>113</td>
</tr>
<tr>
<td>Conversion Specification Modifiers for printf()</td>
<td>115</td>
</tr>
<tr>
<td>What Does a Conversion Specification Convert?</td>
<td>122</td>
</tr>
<tr>
<td>Using scanf()</td>
<td>128</td>
</tr>
</tbody>
</table>
5 Operators, Expressions, and Statements 143

Introducing Loops 144

Fundamental Operators 146
- Assignment Operator: = 146
- Addition Operator: + 149
- Subtraction Operator: – 149
- Sign Operators: – and + 150
- Multiplication Operator: * 151
- Division Operator: / 153
- Operator Precedence 154
- Precedence and the Order of Evaluation 156

Some Additional Operators 157
- The sizeof Operator and the size_t Type 158
- Modulus Operator: % 159
- Increment and Decrement Operators: ++ and -- 160
- Decrementing: -- 164
- Precedence 165
- Don’t Be Too Clever 166

Expressions and Statements 167
- Expressions 167
- Statements 168
- Compound Statements (Blocks) 171

Type Conversions 174
- The Cast Operator 176

Function with Arguments 177

A Sample Program 180

Key Concepts 182

Summary 182

Review Questions 183

Programming Exercises 187
6 C Control Statements: Looping 189

Revisiting the while Loop 190
Program Comments 191
C-Style Reading Loop 192
The while Statement 193
Terminating a while Loop 194
When a Loop Terminates 194
while: An Entry-Condition Loop 195
Syntax Points 195
Which Is Bigger: Using Relational Operators and Expressions 197
What Is Truth? 199
What Else Is True? 200
Troubles with Truth 201
The New _Bool Type 203
Precedence of Relational Operators 205
Indefinite Loops and Counting Loops 207
The for Loop 208
Using for for Flexibility 210
More Assignment Operators: +=, -=, *=, /=, %= 215
The Comma Operator 215
Zeno Meets the for Loop 218
An Exit-Condition Loop: do while 220
Which Loop? 223
Nested Loops 224
Program Discussion 225
A Nested Variation 225
Introducing Arrays 226
Using a for Loop with an Array 228
A Loop Example Using a Function Return Value 230
Program Discussion 232
Using Functions with Return Values 233
Key Concepts 234
Summary 235
Review Questions 236
Programming Exercises 241
7 C Control Statements: Branching and Jumps 245
   The if Statement 246
   Adding else to the if Statement 248
      Another Example: Introducing getchar() and putchar() 250
   The ctype.h Family of Character Functions 252
   Multiple Choice else if 254
   Pairing else with if 257
   More Nested ifs 259
   Let's Get Logical 263
      Alternate Spellings: The iso646.h Header File 265
   Precedence 265
   Order of Evaluation 266
   Ranges 267
   A Word-Count Program 268
   The Conditional Operator: ?: 271
   Loop Aids: continue and break 274
      The continue Statement 274
      The break Statement 277
   Multiple Choice: switch and break 280
      Using the switch Statement 281
      Reading Only the First Character of a Line 283
      Multiple Labels 284
      switch and if else 286
   The goto Statement 287
      Avoiding goto 287
   Key Concepts 291
   Summary 291
   Review Questions 292
   Programming Exercises 296

8 Character Input/Output and Input Validation 299
   Single-Character I/O: getchar() and putchar() 300
   Buffers 301
   Terminating Keyboard Input 302
      Files, Streams, and Keyboard Input 303
   The End of File 304
   Redirection and Files 307
Pointers and Multidimensional Arrays 417
  Pointers to Multidimensional Arrays 420
  Pointer Compatibility 421
  Functions and Multidimensional Arrays 423
Variable-Length Arrays (VLAs) 427
Compound Literals 431
Key Concepts 434
Summary 435
Review Questions 436
Programming Exercises 439

11 Character Strings and String Functions 441
Representing Strings and String I/O 441
  Defining Strings Within a Program 442
  Pointers and Strings 451
String Input 453
  Creating Space 453
  The Unfortunate _gets_() Function 453
  The Alternatives to _getS_() 455
  The _scanf_() Function 462
String Output 464
  The _puts_() Function 464
  The _fputs_() Function 465
  The _printf_() Function 466
The Do-It-Yourself Option 466
String Functions 469
  The _strlen_() Function 469
  The _strcat_() Function 471
  The _strncat_() Function 473
  The _strcmp_() Function 475
  The _strcpy_() and _strncpy_() Functions 482
  The _sprintf_() Function 487
  Other String Functions 489
A String Example: Sorting Strings 491
  Sorting Pointers Instead of Strings 493
  The Selection Sort Algorithm 494
Contents  xv

The ctype.h Character Functions and Strings  495
Command-Line Arguments  497
  Command-Line Arguments in Integrated Environments  500
  Command-Line Arguments with the Macintosh  500
String-to-Number Conversions  500
Key Concepts  504
Summary  504
Review Questions  505
Programming Exercises  508

12 Storage Classes, Linkage, and Memory Management  511
Storage Classes  511
  Scope  513
  Linkage  515
  Storage Duration  516
  Automatic Variables  518
  Register Variables  522
  Static Variables with Block Scope  522
  Static Variables with External Linkage  524
  Static Variables with Internal Linkage  529
  Multiple Files  530
  Storage-Class Specifier Roundup  530
  Storage Classes and Functions  533
  Which Storage Class?  534
A Random-Number Function and a Static Variable  534
Roll 'Em  538
Allocated Memory: malloc() and free()  543
  The Importance of free()  547
  The malloc() Function  548
  Dynamic Memory Allocation and Variable-Length Arrays  548
  Storage Classes and Dynamic Memory Allocation  549
ANSI C Type Qualifiers  551
  The const Type Qualifier  552
  The volatile Type Qualifier  554
  The restrict Type Qualifier  555
  The _Atomic Type Qualifier (C11)  556
  New Places for Old Keywords  557
13 File Input/Output 565

Communicating with Files 565

What Is a File? 566

The Text Mode and the Binary Mode 566

Levels of I/O 568

Standard Files 568

Standard I/O 568

Checking for Command-Line Arguments 569

The fopen() Function 570

The getc() and putc() Functions 572

End-of-File 572

The fclose() Function 574

Pointers to the Standard Files 574

A Simple-Minded File-Condensing Program 574

File I/O: fprintf(), fscanf(), fgets(), and fputs() 576

The fprintf() and fscanf() Functions 576

The fgets() and fputs() Functions 578

Adventures in Random Access: fseek() and ftell() 579

How fseek() and ftell() Work 580

Binary Versus Text Mode 582

Portability 582

The fgetpos() and fsetpos() Functions 583

Behind the Scenes with Standard I/O 583

Other Standard I/O Functions 584

The int ungetc(int c, FILE *fp) Function 585

The int fflush() Function 585

The int setvbuf() Function 585

Binary I/O: fread() and fwrite() 586

The size_t fwrite() Function 588

The size_t fread() Function 588

The int feof(FILE *fp) and int ferror(FILE *fp) Functions 589

An fread() and fwrite() Example 589
Contents

Unions: A Quick Look 645
   Using Unions 646
   Anonymous Unions (C11) 647
Enumerated Types 649
   enum Constants 649
   Default Values 650
   Assigned Values 650
   enum Usage 650
   Shared Namespaces 652
typedef: A Quick Look 653
Fancy Declarations 655
Functions and Pointers 657
Key Concepts 665
Summary 665
Review Questions 666
Programming Exercises 669

15 Bit Fiddling 673
Binary Numbers, Bits, and Bytes 674
   Binary Integers 674
   Signed Integers 675
   Binary Floating Point 676
Other Number Bases 676
   Octal 677
   Hexadecimal 677
C’s Bitwise Operators 678
   Bitwise Logical Operators 678
   Usage: Masks 680
   Usage: Turning Bits On (Setting Bits) 681
   Usage: Turning Bits Off (Clearing Bits) 682
   Usage: Toggling Bits 683
   Usage: Checking the Value of a Bit 683
   Bitwise Shift Operators 684
   Programming Example 685
   Another Example 688
Bit Fields 690
   Bit-Field Example 692
16 The C Preprocessor and the C Library 711
   First Steps in Translating a Program 712
   Manifest Constants: 
   - Tokens 717
   - Redefining Constants 717
   Using Arguments with 
   - Creating Strings from Macro Arguments: The # Operator 721
   - Preprocessor Glue: The ## Operator 722
   - Variadic Macros: ... and __VA_ARGS__ 723
   Macro or Function? 725
   File Inclusion: 
   - Header Files: An Example 727
   - Uses for Header Files 729
   Other Directives 730
   - The #undef Directive 731
   - Being Defined—The C Preprocessor Perspective 731
   - Conditional Compilation 731
   - Predefined Macros 737
   - #line and #error 738
   - #pragma 739
   - Generic Selection (C11) 740
   Inline Functions (C99) 741
   _Noreturn Functions (C11) 744
The C Library 744
   - Gaining Access to the C Library 745
   - Using the Library Descriptions 746
The Math Library 747
   - A Little Trigonometry 748
   - Type Variants 750
   - The tmath.h Library (C99) 752
### 17 Advanced Data Representation

#### Exploring Data Representation

Beyond the Array to the Linked List

Using a Linked List

Afterthoughts

Abstract Data Types (ADTs)

Getting Abstract

Building an Interface

Using the Interface

Implementing the Interface

Getting Queued with an ADT

Defining the Queue Abstract Data Type

Defining an Interface

Implementing the Interface Data Representation

Testing the Queue

Simulating with a Queue

The Linked List Versus the Array

Binary Search Trees

A Binary Tree ADT

The Binary Search Tree Interface

The Binary Tree Implementation

Trying the Tree

Tree Thoughts
The Conditional Operator 911
Pointer-Related Operators 912
Sign Operators 912
Structure and Union Operators 912
Bitwise Operators 913
Miscellaneous Operators 914
Section III: Basic Types and Storage Classes 915
Summary: The Basic Data Types 915
Summary: How to Declare a Simple Variable 917
Summary: Qualifiers 919
Section IV: Expressions, Statements, and Program Flow 920
Summary: Expressions and Statements 920
Summary: The while Statement 921
Summary: The for Statement 921
Summary: The do while Statement 922
Summary: Using if Statements for Making Choices 923
Summary: Multiple Choice with switch 924
Summary: Program Jumps 925
Section V: The Standard ANSI C Library with C99 and C11 Additions 926
Diagnostics: assert.h 926
Complex Numbers: complex.h (C99) 927
Character Handling: ctype.h 929
Error Reporting: errno.h 930
Floating-Point Environment: fenv.h (C99) 930
Floating-point Characteristics: float.h 933
Format Conversion of Integer Types: inttypes.h (C99) 935
Alternative Spellings: iso646.h 936
Localization: locale.h 936
Math Library: math.h 939
Non-Local Jumps: setjmp.h 945
Signal Handling: signal.h 945
Alignment: stdalign.h (C11) 946
Variable Arguments: stdarg.h 947
Atomics Support: stdatomic.h (C11) 948
Boolean Support: stdbool.h (C99) 948
Common Definitions: stddef.h 948
Integer Types: stdint.h 949
Standard I/O Library: stdio.h 953
General Utilities: stdlib.h 956
__Noreturn: stdnoreturn.h 962
String Handling: string.h 962
Type-Generic Math: tgmath.h (C99) 965
Threads: threads.h (C11) 967
Date and Time: time.h 967
Unicode Utilities: uchar.h (C11) 971
Extended Multibyte and Wide-Character Utilities: wchar.h (C99) 972
Wide Character Classification and Mapping Utilities: wctype.h (C99) 978

Section VI: Extended Integer Types 980
Exact-Width Types 981
Minimum-Width Types 982
Fastest Minimum-Width Types 983
Maximum-Width Types 983
Integers That Can Hold Pointer Values 984
Extended Integer Constants 984

Section VII: Expanded Character Support 984
Trigraph Sequences 984
Digraphs 985
Alternative Spellings: iso646.h 986
Multibyte Characters 986
Universal Character Names (UCNs) 987
Wide Characters 988
Wide Characters and Multibyte Characters 989

Section VIII: C99/C11 Numeric Computational Enhancements 990
The IEC Floating-Point Standard 990
The fenv.h Header File 994
The STDC FP_CONTRACTPragma 995
Additions to the math.h Library 995
Support for Complex Numbers 996

Section IX: Differences Between C and C++ 998
Function Prototypes 999
char Constants 1000
The const Modifier 1000
Structures and Unions 1001
Enumerations 1002
Dedication

To the memory of my father, William Prata.

About the Author

Stephen Prata, now retired, taught astronomy, physics, and programming at the College of Marin in Kentfield, California. 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.

Acknowledgments

I wish to thank Mark Taber at Pearson for getting this project underway and for seeing it through. And I’d like to thank Danny Kalev for his technical help and for suggesting the term “program scope.”
We Want to Hear from You!

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.

You can email or write directly to let us know what you did or didn’t like about this book—as well as what we can do to make our books better.

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

When you write, please be sure to include this book’s title, edition number, and author as well as your name and contact information.

Email:    feedback@developers-library.info
Mail:      Reader Feedback
           Addison-Wesley Developer’s Library
           800 East 96th Street
           Indianapolis, IN 46240 USA

Reader Services

Visit our website and register this book at informit.com/register for convenient access to any updates, downloads, or errata that might be available for this book.
Preface

C was a relatively little-known language when the first edition of *C Primer Plus* appeared in 1984. Since then, the language has boomed, and many people have learned C with the help of this book. In fact, *C Primer Plus* throughout its various editions has sold over 550,000 copies.

As the language has grown from the early informal K&R standard through the 1990 ISO/ANSI standard through the 1999 ISO/ANSI standard to the 2011 ISO/IEC standard, so has this book matured through this, the sixth 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 with changing part of a program to see what the effect is. Modify a program to do something slightly different. See if you can develop an alternative approach. 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.
This page intentionally left blank
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, we 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 platinum.c Program

```c
#include <stdio.h>

int main(void)
{
    float weight;    /* user weight          */
    float value;     /* platinum equivalent */

    printf("Are you worth your weight in platinum?\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 $1700 per ounce */
    /* 14.5833 converts pounds avd. to ounces troy */
    value = 1700.0 * weight * 14.5833;
    printf("Your weight in platinum is worth $%.2f.\n", value);
    printf("You are easily worth that! If platinum prices drop,\n");
    printf("eat more to maintain your value.\n");

    return 0;
}
```

Tip 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 such as 1700.0. 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 1700.0 to the current price of the precious metal platinum. 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 156, not words, such as too much. Entering letters rather than digits causes problems that require an \texttt{if} statement (Chapter 7, “C Control Statements: Branching and Jumps”) to defeat, so please be polite and enter a number. Here is some sample output:

Are you worth your weight in platinum?
Let’s check it out.
Please enter your weight in pounds: 156
Your weight in platinum is worth $3867491.25.
You are easily worth that! If platinum prices drop,
eat more to maintain your value.

Program Adjustments
Did the output for this program briefly flash onscreen and then disappear even though you added the following line to the program, as described in Chapter 2, “Introducing C”?

\begin{verbatim}
getchar();
\end{verbatim}

For this example, you need to use that function call twice:

\begin{verbatim}
getchar();
getchar();
\end{verbatim}

The \texttt{getchar()} function reads the next input character, so the program has to wait for input. In this case, we provided input by typing 156 and then pressing the Enter (or Return) key, which transmits a newline character. So \texttt{scanf()} reads the number, the first \texttt{getchar()} reads the newline character, and the second \texttt{getchar()} causes the program to pause, awaiting further input.

What’s New in This Program?

There are several new elements of C in this program:

\begin{itemize}
\item Notice that the code uses a new kind of variable declaration. The previous examples just used an integer variable type (\texttt{int}), but this one adds a floating-point variable type (\texttt{float}) so that you can handle a wider variety of data. The \texttt{float} type can hold numbers with decimal points.
\item The program demonstrates some new ways of writing constants. You now have numbers with decimal points.
\item To print this new kind of variable, use the \texttt{\%f} specifier in the \texttt{printf}() code to handle a floating-point value. The \texttt{.2} modifier to the \texttt{\%f} specifier fine-tunes the appearance of the output so that it displays two places to the right of the decimal.
\item The \texttt{scanf()} function provides keyboard input to the program. The \texttt{\&weight} tells \texttt{scanf()} to read a floating-point number from the keyboard, and the \texttt{weight} tells \texttt{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 156 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().

```c
/*platinum.c*/

int main(void)
{
    /*platinum.c*/
    int main(void)
    {
    // getting keyboard input
    scanf("-----");
    // displaying program output
    printf("Are you--");
    printf("-----");
    printf("Are you--");
    printf("-----");
    return 0;
}
```

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 types of data are preset before a program is used and keep their values unchanged throughout the life of the program. These are constants. Other types of 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 1700.0? True, the price of platinum 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 types of 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 type keywords 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>
<thead>
<tr>
<th>Original K&amp;R Keywords</th>
<th>C90 K&amp;R Keywords</th>
<th>C99 Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>signed</td>
<td>_Bool</td>
</tr>
<tr>
<td>long</td>
<td>void</td>
<td>_Complex</td>
</tr>
<tr>
<td>short</td>
<td></td>
<td>_Imaginary</td>
</tr>
<tr>
<td>unsigned</td>
<td></td>
<td></td>
</tr>
<tr>
<td>char</td>
<td></td>
<td></td>
</tr>
<tr>
<td>float</td>
<td></td>
<td></td>
</tr>
<tr>
<td>double</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The int keyword provides the basic class of integers used in C. The next three keywords (long, short, and unsigned) and the C90 addition signed are used to provide variations of the basic type, for example, unsigned short int and long long int. 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.

**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 sidebar 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 such as 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).

![8-bit word diagram](image)

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 10 for internal storage. You’ll find more on this topic in Chapter 15. Now, let’s concentrate on the practical differences:

- 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 is 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 \texttt{float} value—more about precision later.

Floating-point operations were once much slower than integer operations. However, today many CPUs incorporate floating-point processors that close the gap.

![Floating-point representation](image)

**Figure 3.3** Storing the number \( \pi \) in floating-point format (decimal version).

### Basic 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 with a literal value, such as 5 or 2.78, 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 \texttt{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 \texttt{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 \texttt{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 \texttt{int} uses one machine word for storage. Therefore, older IBM PC compatibles, which have a 16-bit word, use 16 bits to store an \texttt{int}. This allows a range in values from \(-32768\) to \(32767\). Current personal computers typically have 32-bit integers and fit an \texttt{int} to that size. Now the personal computer industry is moving toward 64-bit processors that naturally will use even larger integers. ISO C specifies that the minimum range for type \texttt{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:

```c
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:

```c
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 a starting, or initial, 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:

```c
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).
Type int Constants

The various integers (21, 32, 14, and 94) in the last example are integer constants, also called integer literals. 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 the section “long Constants 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;
    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
return 0;
}

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

Doing it right: 10 minus 2 is 8
Doing it wrong: 10 minus 16 is 1650287143

For the first line of output, 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. Not only might the memory contents be different, but different compilers will manage memory locations differently.)

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. Some compilers, however, will use unusual methods of checking and warn you that you might be doing something wrong. Still, it’s best to remember to always 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 4 bits. For example, the hexadecimal digit 3 is 0011 and the hexadecimal digit 5 is 0101. So the hexadecimal value 35 is the bit pattern 0011 0101, and the hexadecimal value 53 is 0101 0011. This correspondence makes it easy to go back and forth between hexadecimal and binary (base 2) notation. 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-ex) 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
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 to 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

```c
/* 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 = 64
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. Here are some points to keep in mind:
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` (introduced in the C99 standard), may use more storage than `long`. At the minimum, it must use at least 64 bits. 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.

```c
long int estime;
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. For example, in the days of Windows 3, an `int` and a `short` were both 16 bits, and a `long` was 32 bits. Later, Windows and Apple systems moved to using 16 bits for `short` and 32 bits for `int` and `long`. Using 32 bits allows integers in excess of 2 billion. Now that 64-bit processors are common, there’s a need for 64-bit integers, and that’s the motivation for the `long long` type.
The most common practice today on personal computers is to set up `long long` as 64 bits, `long` as 32 bits, `short` as 16 bits, and `int` as either 16 bits or 32 bits, depending on the machine’s natural word size. In principle, these four types could represent four distinct sizes, but in practice at least some of the types normally overlap.

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 fifteen using U.S. nomenclature (the short scale or *échelle courte* system), 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 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.

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.)

```c
/* 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 \textit{j} is acting like a car’s odometer. When it reaches its maximum value, it starts over at the beginning. The integer \textit{i} acts similarly. The main difference is that the unsigned \texttt{int} variable \textit{j}, like an odometer, begins at 0, but the \texttt{int} variable \textit{i} begins at -2147483648. Notice that you are not informed that \textit{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. The behavior shown here is typical, but you could encounter something different.

\textbf{long Constants and long long Constants}

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

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

Sometimes you might want the compiler to store a small number as a \texttt{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 \texttt{long} values. To cause a small constant to be treated as type \texttt{long}, you can append an \texttt{l} (lowercase \texttt{L}) or \texttt{L} as a suffix. The second form is better because it looks less like the digit 1. Therefore, a system with a 16-bit \texttt{int} and a 32-bit \texttt{long} treats the integer 7 as 16 bits and the integer 7\texttt{L} as 32 bits. The \texttt{l} and \texttt{L} suffixes can also be used with octal and hex integers, as in 020\texttt{L} and 0x10\texttt{L}.

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

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. So 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

```c
/* 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 %d\n", big, big);
    printf("verybig = %lld and not %ld\n", verybig, verybig);

    return 0;
}
```

Here is the output on one system (results can vary):

```
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 end 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. The value 65537 expressed in binary format as a 32-bit number is 00000000000000010000000000000001. Using the %hd specifier persuaded printf() to look at just the last 16 bits; therefore, 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.

**Tip** 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 U.S. is the ASCII code given in the table on the inside front cover. 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 Macs, 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 110,000 characters. The International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) have 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.

The C language defines a byte to be the number of bits used by type char, so one can have a system with a 16-bit or 32-bit byte and char type.

### Declaring Type char Variables

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

```c
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:

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

A single character 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. Here's another example:

```c
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 omit 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:

```c
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

```c
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 such as ‘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 the table on the inside front cover of this book, you’ll 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:

```c
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>
<thead>
<tr>
<th>Sequence</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\a</td>
<td>Alert (ANSI C).</td>
</tr>
<tr>
<td>\b</td>
<td>Backspace.</td>
</tr>
<tr>
<td>\f</td>
<td>Form feed.</td>
</tr>
<tr>
<td>\n</td>
<td>Newline.</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage return.</td>
</tr>
<tr>
<td>\t</td>
<td>Horizontal tab.</td>
</tr>
<tr>
<td>\v</td>
<td>Vertical tab.</td>
</tr>
<tr>
<td>&quot;</td>
<td>Backslash ().</td>
</tr>
<tr>
<td>'</td>
<td>Single quote (‘).</td>
</tr>
</tbody>
</table>
Chapter 3  Data and C

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;</td>
<td>Double quote (&quot;).</td>
</tr>
<tr>
<td>?</td>
<td>Question mark (?)</td>
</tr>
<tr>
<td>\000</td>
<td>Octal value. (o represents an octal digit.)</td>
</tr>
<tr>
<td>\xhh</td>
<td>Hexadecimal value. (h represents a hexadecimal digit.)</td>
</tr>
</tbody>
</table>

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

```c
char 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 C 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 generalization 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, these 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 sequence 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."
```

Then use this code:

```c
printf("Gramps sez, \"a \ is a backslash.\"\n");
```
The final two forms (\0oo 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:

```c
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 Ctrl+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.
```

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

```
Figure 3.5 Writing constants with the int family.
```

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 I use the ASCII code, and when should I 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 I 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 such as \032 can be embedded in part of a C string, the way \007 was in the first point.

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).
Basic C Data Types

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.

As of 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 1 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, so let’s defer further discussion until then.

Portable Types: stdint.h and inttypes.h
By now you’ve probably noticed that C offers a wide variety of integer types, which is a good thing. And you probably also have noticed that the same type name doesn’t necessarily mean the same thing on different systems, which is not such a good thing. It would be nice if C had types that had the same meaning regardless of the system. And, as of C99, it does—sort of.

What C has done is create more names for the existing types. The trick is to define these new names in a header file called stdint.h. For example, int32_t represents the type for a 32-bit
signed integer. The header file on a system that uses a 32-bit int could define int32_t as an alias for int. A different system, one with a 16-bit int and a 32-bit long, could define the same name, int32_t, as an alias for int. Then, when you write a program using int32_t as a type and include the stdint.h header file, the compiler will substitute int or long for the type in a manner appropriate for your particular system.

The alternative names we just discussed are examples of exact-width integer types; int32_t is exactly 32 bits, no less or no more. It’s possible the underlying system might not support these choices, so the exact-width integer types are optional.

What if a system can’t support exact-width types? C99 and C11 provide a second category of alternative names that are required. This set of names promises 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 16 bits, the int8_t type would not be defined. However, 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 and C11 define 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; intmax_t stands for that type, a type that can hold any valid signed integer value. Similarly, uintmax_t stands for the largest available unsigned type. Incidentally, these types could be bigger than long long and unsigned long because C implementations are permitted to define types beyond the required ones. Some compilers, for example, introduced the long long type before it became part of the standard.

C99 and C11 not only provide these new, portable type names, they also provide assistance 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 an %ld for another? The current standard provides some string macros (a mechanism introduced in Chapter 4) to be used to display the portable types. For example, the inttypes.h header file will define PRIi32 as a string representing the appropriate specifier (d or 1, for instance) for a 32-bit signed value. Listing 3.6 shows a brief example illustrating how to use a portable type and its associated specifier. The inttypes.h header file includes stdint.h, so the program only needs to include inttypes.h.

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)
```c

{ int32_t me32; // me32 a 32-bit signed variable

me32 = 45933945;
printf("First, assume int32_t is int: ");
printf("me32 = %d\n", me32);
printf("Next, let's not make any assumptions.\n");
printf("Instead, use a \"macro\" from inttypes.h: ");
printf("me32 = %” PRId32 “\n", me32);

return 0;
}
```

In the final `printf()` argument, the `PRId32` is replaced by its `inttypes.h` definition of “d”, making the line this:

```
printf("me16 = %” d “\n", me16);
```

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

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

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

First, assume int32_t is int: me32 = 45933945
Next, let’s not make any assumptions.
Instead, use a "macro" from inttypes.h: me32 = 45933945

It’s not the purpose of this section to teach you all about expanded integer types. Rather, its main intent is to reassure you that this level of control over types is available if you need it. Reference Section VI, “Extended Integer Types,” in Appendix B provides a complete rundown of the `inttypes.h` and `stdint.h` header files.

**Note C99/C11 Support**

Even though C has moved to the C11 standard, compiler writers have implemented C99 features at different paces and with different priorities. At the time this book was prepared, some compilers haven’t yet implemented the `inttypes.h` header file and features.

**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.

<table>
<thead>
<tr>
<th>Number</th>
<th>Scientific Notation</th>
<th>Exponential Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000,000,000</td>
<td>1.0\times10^9</td>
<td>1.0e9</td>
</tr>
<tr>
<td>123,000</td>
<td>1.23\times10^5</td>
<td>1.23e5</td>
</tr>
<tr>
<td>322.56</td>
<td>3.2256\times10^2</td>
<td>3.2256e2</td>
</tr>
<tr>
<td>0.000056</td>
<td>5.6\times10^{-5}</td>
<td>5.6e-5</td>
</tr>
</tbody>
</table>

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 such as 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.
Basic C Data Types

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.

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:

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

Floating-Point Constants (Literals)

There are many choices open to you when you write a literal floating-point constant. The basic form of a floating-point literal 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:

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

Then 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 it 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 1 (one) than is `l`. If the floating-point number has no suffix, it is type double.

Since C99, C has 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:

```c
0xa.1fp10
```

The a is 10 in hex, the .1f is 1/16th plus 15/256th (f is 15 in hex), and the p10 is 2^10, or 1024, making the complete value (10 + 1/16 + 15/256) x 1024, or 10364.0 in base 10 notation.

Not all C compilers have added support for this feature.

### 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 hexadecimal format for floating-point numbers, you can use `a` or `A` 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:

```c
0xa.1fp10
```

The `a` is 10 in hex, the `.1f` is 1/16th plus 15/256th (`f` is 15 in hex), and the `p10` is 2^10, or 1024, making the complete value (10 + 1/16 + 15/256) x 1024, or 10364.0 in base 10 notation.

Not all C compilers have added support for this feature.

### Listing 3.7 The `showf_pt.c` Program

```c
/* 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);
    // next line requires C99 or later compliance
    printf("And it's %a in hexadecimal, powers of 2 notation\n", aboat);
```
printf("%f can be written %e\n", abet, abet);
printf("%Lf can be written %Le\n", dip, dip);

return 0;
}

This is the output, provided your compiler is C99/C11 compliant:

32000.000000 can be written 3.200000e+04
And it’s 0x1.f4p+14 in hexadecimal, powers of 2 notation
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:

```c
float toobig = 3.4E38 * 100.0f;
printf("%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 (or some variation on that theme) 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 10 value with four significant digits, such as 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, NaN, or something similar.

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 ← older gcc on Linux
-13584010575872.000000 ← Turbo C 1.5
4008175468544.000000 ← XCode 4.5, Visual Studio 2012, current gcc

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 six or seven 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 fifth digit, and float numbers are precise enough for that.

Floating-Point Representation

The preceding sidebar listed different possible outputs for the same program, depending on the computer system used. The reason is that there are many possible ways to implement floating-point representation within the broad outlines discussed earlier. To provide greater uniformity, the Institute of Electrical and Electronics Engineers (IEEE) developed a standard for floating-point representation and computation, a standard now used by many hardware floating-point units. In 2011 this standard was adopted as the international ISO/IEC/IEEE 60559:2011 standard. This standard is incorporated as an option in the C99 and C11 standards, with the intention that it be supported on platforms with conforming hardware. The final example of output for the floaterr.c program comes from systems supporting this floating-point standard. C support includes tools for catching the problem. See Appendix B, Section V for more details.
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. With C11, the entire complex number package is 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 and the word imaginary for _Imaginary, and it allows you to use the symbol I to represent the square root of –1.

You may wonder why the C standard doesn’t simply use complex as the keyword instead of using _Complex and then adding a header file to define complex as _Complex. The standards committee is hesitant to introduce a new keyword because that can invalidate existing code that uses the same word as an identifier. For example, prior to C99, many programmers had already used, say, struct complex to define a structure to represent complex numbers or, perhaps, psychological conditions. (The keyword struct, as discussed in Chapter 14, “Structures and Other Data Forms,” is used to define data structures capable of holding more than one value.) Making complex a keyword would make these previous uses syntax errors. But it’s much less likely that someone would have used struct _Complex, especially since using identifiers having an initial underscore is supposed to be reserved. So the committee settled on _Complex as the keyword and made complex available as an option for those who don’t have to worry about conflicts with past usage.

Beyond the Basic 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. For instance, 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 11 keywords: int, long, short, unsigned, char, float, double, signed, _Bool, _Complex, and _Imaginary.

Signed Integers:
These 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, old DOS-based systems for the PC provided 16-bit short and int and 32-bit long, and Windows 95–based systems and later provide 16-bit short and 32-bit int and long.

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

Unsigned Integers:
These 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:
These 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:
These 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. Here's an example:
   ```
   char ch, init, ans;
   ```

5. You can initialize a variable in a declaration statement:
   ```
   float mass = 6.0E24;
   ```

**Type Sizes**
What type sizes does your system use? Try running the program in Listing 3.8 to find out.

**Listing 3.8 The typesize.c Program**
```c
/* typesize.c -- prints out type sizes */
#include <stdio.h>
int main(void)
{
    /* c99 provides a %zd specifier for sizes */
```
C has a built-in operator called \texttt{sizeof} that gives sizes in bytes. C99 and C11 provide a \texttt{\%zd} specifier for this type used by \texttt{sizeof}. Noncompliant compilers may require \texttt{\%u} or \texttt{\%lu} instead. Here is a sample output:

- Type int has a size of 4 bytes.
- Type char has a size of 1 byte.
- Type long has a size of 8 bytes.
- Type long long has a size of 8 bytes.
- Type double has a size of 8 bytes.
- Type long double has a size of 16 bytes.

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

Incidentally, notice in the last few lines how a \texttt{printf()} statement can be 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 \texttt{int} or possibly \texttt{float} for the numbers and \texttt{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. Here’s an example:

```c
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:

```c
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`:

```c
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`. 

---

*Arguments and Pitfalls*
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 whether 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

```c
/* badcount.c -- incorrect argument counts */
#include <stdio.h>
int main(void)
{
    int n = 4;
    int m = 5;
    float f = 7.0f;
    float g = 8.0f;

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

    return 0;
}
```

Here’s a sample output from XCode 4.6 (OS 10.8):

```
4
4 1 -706337836
1606414344 1
```

Next, here’s a sample output from Microsoft Visual Studio Express 2012 (Windows 7):

```
4
4 0 0
0 1075576832
```

Note that using `%d` to display a `float` value doesn’t convert the `float` value to the nearest `int`. Also, the results you get for too few arguments or the wrong kind of argument differ from platform to platform and can from trial to trial.

None of the compilers we tried refused to compile this code; although most did issue warnings that something might be wrong. Nor were there any complaints when we ran the program. It is true that some compilers might catch this sort of error, but the C standard doesn’t require them to. Therefore, the computer may not catch this kind of error, 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.

One More Example: Escape Sequences

Let’s run one more printing example, one that makes use of some of C’s special escape sequences for characters. In particular, the program in Listing 3.10 shows how the 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, Listing 3.10 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(" $_______\b\b\b\b\b\b\b");             /* 2 */
  scanf("%f", &salary);
  printf("\n\t$%.2f a month is $%.2f a year.", salary,
        salary * 12.0);                     /* 3 */
  printf("\rGee!\n");                            /* 4 */

  return 0;
}

What Happens When the Program Runs

Let’s walk through this program step by step as it would work under a system in which the escape characters behave as described. (The actual behavior could be different. For instance, XCode 4.6 displays the \a, \b, and \r characters as upside down question marks!)

The first printf() statement (the one numbered 1) sounds the alert signal (prompted by the \a) and 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 underscore 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 4000.00. Now the line looks like this:

Enter your desired monthly salary: $4000.00

The characters you type replace the underscore 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: $4000.00
$4000.00 a month is $48000.00 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.

Here is the final appearance of the screen:

Enter your desired monthly salary: $4000.00
Gee! $4000.00 a month is $48000.00 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:

```c
printf("Enter your desired monthly salary:\n");
```
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 meet 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 U.S., 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 char-
acter 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 the `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, since C90, `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. C99 and C11 provide a third exponential notation using hexadecimal digits and powers of 2, as in 0xa.1fp10.

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 (sometimes more than one type could be appropriate)?
   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
5. Dottie Cawm has concocted an error-laden program. Help her find the mistakes.

```c
#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:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Type</th>
<th>Specifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>a. 12</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>b. 0X3</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>c. 'C'</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>d. 2.34E07</td>
<td>float</td>
<td></td>
</tr>
<tr>
<td>e. '\040'</td>
<td>char</td>
<td></td>
</tr>
<tr>
<td>f. 7.0</td>
<td>float</td>
<td></td>
</tr>
<tr>
<td>g. 6L</td>
<td>long</td>
<td></td>
</tr>
<tr>
<td>h. 6.0f</td>
<td>float</td>
<td></td>
</tr>
<tr>
<td>i. 0x5.b6p12</td>
<td>long double</td>
<td></td>
</tr>
</tbody>
</table>

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):

<table>
<thead>
<tr>
<th>Constant</th>
<th>Type</th>
<th>Specifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>a. 012</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>b. 2.9e05L</td>
<td>float</td>
<td></td>
</tr>
<tr>
<td>c. 's'</td>
<td>char</td>
<td></td>
</tr>
<tr>
<td>d. 100000</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>e. '\n'</td>
<td>char</td>
<td></td>
</tr>
</tbody>
</table>
f. 20.0f

g. 0x44

h. -40

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. \\n
    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. (You can check the discussion in Chapter 4 of limits.h and float.h to get guidance on the largest and smallest values.)

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, then in exponential notation, and then, if your system supports it, p notation. Have the output use the following format (the actual number of digits displayed for the exponent depends on the system):
   Enter a floating-point value: 64.25
   fixed-point notation: 64.250000
   exponential notation: 6.425000e+01
   p notation: 0x1.01p+6

5. There are approximately $3.156 \times 10^7$ 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 $3.0 \times 10^{-23}$ 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.

8. In the U.S. system of volume measurements, a pint is 2 cups, a cup is 8 ounces, an ounce is 2 tablespoons, and a tablespoon is 3 teaspoons. Write a program that requests a volume in cups and that displays the equivalent volumes in pints, ounces, tablespoons, and teaspoons. Why does a floating-point type make more sense for this application than an integer type?
This page intentionally left blank
Symbols

-/+ (sign operators), 149
[ ] (brackets), 102
    arrays, 384
    empty, 388, 424
%= assignment operator, 214
*= assignment operator, 214, 230
+= assignment operator, 214
-= assignment operator, 214
/= assignment operator, 214
~ bitwise operator, 688
<< (left shift) bitwise operator, 684
>> (right shift) bitwise operator, 684-685
. (period) character, 262
* modifier, printf( ) function, 133-135
! operator, 264
# operator, strings from macro arguments, 721-722
### operator, 722-723
& operator, 367-368
    bitwise, 679
&& operator, 264
    ranges, 267-268
| operator, bitwise, 679-680
|| operator, 264
+ (addition) operator, 149
= (assignment) operator, 146-149, 202
?: (conditional) operator, 272-273
- (decrement) operator, 164-166
== (equality) operator, 191
++ (increment) operator, 160-166
* (indirection) operator, 371-372
* (multiplication) operator, 151-153
. (membership) operator, 912-913
== (relational) operator, 202
+ (sign) operator, 150
- (subtraction) operator, 149-150
*/ symbol, 30, 33-34
/+ symbol, 30, 33-34
* unary operator, 406
++ unary operator, 406
/ (division) operator, 153-154
< (redirection) operator, 308
> (redirection) operator, 308
{ } (braces), 34
  while loop, 146

A
a+b mode, 643
actual arguments, 343-344
add_one.c program, 160-161
addaword.c program, 577-578
addemup.c program, 169-170
AddItem( ) function, 793, 800-801, 833-837
addition (+) operator, 149
AddNode( ) function, 833-835
addresses
  & operator, 367-368
double quotation marks, 465
function pointers, 657
inline functions, 743
pointers, 409
structures, 619-620
variables, 375
addresses.c program, 446-447

ADT (abstract data type), 774, 786-787
  binary search trees, 829
    EmptyTree( ) function, 833
    FullTree( ) function, 833
    InitializeTree( ) function, 833
    interface, 830-832
    Treeitems( ) function, 833
defining, 787
interfaces
  building, 789-793
defining, 805-806
functions, 810-815
implementing, 796-802
using, 793-796
lists, operations, 788
queue, 804
align.c program, 704-705
alignment, C11, 703
allocated memory, 543
calloc( ) function, 548
dynamic, VLAs and, 548-549
free( ) function, 545-548
malloc( ) function, 543-544
storage classes and, 549-551
structures, 605
altnames.c program, 78-79
AND operator, 679
animals.c program, 280-281
anonymous structures, 636-637
anonymous unions, 647
ANSI (American Nation Standards Institute), 8
ANSI C, 8-9, 17
  functions, prototyping, 349-353
  math functions, 748
type qualifiers, 551
  _Atomic, 556-557
const, 552-554
formal parameters, 557
restrict, 555-556
volatile, 554-555
ANSI/ISO C standard, 8-9
a.out file, 17
append.c program, 590-592
Apple, Xcode, 21
arguments, 89-91
actual, 343-344
command-line, 497
   integrated environment, 500, 569-570
#define, 718-722
## operator, 722-723
variadic macros, 723-724
float, conversion, 116
fseek( ) function, 580-581
functions, 340
functions with, 177-180
none, 352-353
passing, 124, 621-622
printf( ) function, 114
unspecified, 352-353
arrays, 226-227, 407. See also VLAs (variable-length arrays)
   [ ] (brackets), 102, 384
      empty, 388
      of arrays, 419
bounds, 390-392
char, 101-102, 227
      in memory, 228
character string arrays, 444-445, 449-451
compound literals, 432
const keyword, 385
   array size, 431
   contents, protecting, 412-417
   creating, 544
days[ ], 385
declaring, 102
   constant expressions, 544
   pointers and, 544
   variable expressions and, 544
description, 101
designated initializers, 388-390
elements, inserting, 824
function pointers, 664
index, 384
initialization, 384-388, 444-445
   multidimensional, 396
int, in memory, 228
linked lists and, 824-828
for loops in, 228-230
members, flexible, 633-636
multidimensional, 393-398
   functions and, 423-427
   pointers and, 417-427
   two-dimensional, 394-398
names, pointer notation, 402
notation, pointers and, 402
parameters, declaring, 403
pointers and, 398
   comparison, 445-447
   differences, 447-449
   parentheses, 420
as queue, 806
ragged, 450
rectangular, 450
size, specifying, 392-393
storage classes, 386
structures, 607-608
  character arrays, 627-628
declaring, 611
  functions, 637-638
  members, 612
of unions, 645
values, assigning, 390
VLAs, dynamic memory allocation and, 548-549
arrchar.c program, 449-450
ASCII code, numbers versus number characters, 75
assembly languages, 3
assert library, 760
  assert( ) function, 760-763
assert.c program, 761-762
assigned values, enumerated types, 650
assignment
  pointers, 409
  void function, 658
assignment operators, 910-911
  =, 146-149, 202
  %=, 214
  *=, 214, 230
  +=, 214
  -=, 214
  /=, 214
assignment statements, 37-38
atan( ) function, 747
atexit() function, 753-755
atoi( ) function, 500-502
Atomic type qualifier, 556-557
auto keyword, 518
automatic access to C library, 745
automatic variables, storage classes, 518-522

B

B language, 1
base 2 system, 674
bases.c program, 66
BASIC, 3
Bell Labs, 1
binary files, 566, 582
binary floating points
  floating-point representation, 676
  fractions, 676
binary integers, 674-675
binary I/O, 610, 593-594
binary numbers
  decimal equivalents, 678
  hexadecimal equivalents, 678
  octal digits, 677
binary operators, 150
binary output, 586
binary searches, 826-827
  trees, 828-829
    adding items, 833-836
    AddItem( ) function, 833-835, 836-837
    AddNode( ) function, 833-835
    ADTs, 829-843
    DeleteAll( ) function, 843
    DeleteItem( ) function, 836-837, 841-842
    DeleteNode( ) function, 841-842
defeating items, 837-839, 841-842
    deleting nodes, 840-841
    emptying, 843
    EmptyTree( ) function, 833
    finding items, 836-837
    FullTree( ) function, 833
    InitializeTree( ) function, 833
interface, 830-832
InTree() function, 836-837
MakeNode() function, 833-835
SeekItem() function, 833-835, 836-837, 841-842
tips, 854-856
ToLeft() function, 835
ToRight() function, 835
traversing trees, 842
TreeItems() function, 833

binary system, 674
binary tree, 644
binary view (files), 567
binary.c program, 359-360
binbit.c program, 686-687
bit fields, 690-692
bitwise operators and, 696-703
example, 692-695
bit numbers, values, 674
bitmaped images, 774
bits, 60, 674
bitwise operators, 683, 913-914
~, 688
binbit.c program, 686-687
bit fields and, 696-703
clearing bits, 682-683
logical, 678-680
masks, 680-681
setting bits, 681-682
shift operators, 684-685
values, checking, 683-684
black-box viewpoint, 345
blank lines, 41
block scope, 514
blocks (compound statements), 171-173
body, functions, 40

book inventory sample, 601-602
arrays of structures, functions, 637-638
book.c program, 602-603
flexible array members, 633-636
manybook.c program, 608-613
structure declaration, 604
initialization, 606
initializers, 607-608
member access, 607
struct keyword, 604
variables, 605-608
structures
address, 619-620
anonymous, 636-637
arrays, 608-613
compound literals and, 631-633
passing as argument, 621-622
passing members, 618-619
pointers to, 626-627
saving contents to file, 639-644
book.c program, 602-603
books
C++, 908
C language, 907
programming, 907
reference, 908
booksave.c program, 640-643
_Bool type, 77, 203-204
Borland C, floating-point values and, 608
Borland C++ Compiler 5.5, 19
bottles.c program, 164-165
bounds, arrays, 390-392
bounds.c program, 391-392
braces ({}), 30, 34
while loop, 146
brackets ([ ]), 102
    arrays, 384
eempty, 424
break statement, 282-283
    loops, 277-279
break.c program, 277-279
buffers, 300-302
    file position indicator and, 584
    input, user interface, 312-314
butler( ) function, 44-45, 177
bytes, 60
C
C language
    operators, 908-909
    reference books, 907
C library
    access
        automatic, 745
        file inclusion, 745
        library inclusion, 745-746
    descriptions, 746-747
C Reference Manual, 8
C++, 4
    books, 908
    C comparison, const keyword, 423
    enumeration, 649
C11 standard, 9
    alignment, 703-705
    generic selection, 740-741
    _Noreturn functions, 744
C99 standard, 8-9
    compound literals, structures and, 631-633
    designated initializers, 388-390
    flexible array members, 633-636
    functions, inline, 741-744
tgmath.h library, 752
calling functions
    arguments, 343-344
    nested calls, 468-469
    variables, altering, 369-371
calloc( ) function, 548
case labels
    enum variables and, 650
    multiple, 284-285
cast operator, type conversions, 176
cc compiler, 17
CDC 6600 computer, 7
char arrays, 101-102, 227
    in memory, 228
char keyword, 60
char type, 71-72, 93, 136
    nonprinting characters, 73-76
    printing characters, 76
    signed, 77
    unsigned, 77
    variables, declaring, 72
character arrays, structures, 627-628
character constants, 94
    initialization, 72-73
character functions, ctype.h, 252-253
character input, mixing with number, 314-317, 327-330
character pointers, structures, 627-628
character string arrays, 444-445, 449-451
character string literals, 442-443
character strings, 101, 227, 441
characters
    null, 459
    reading single, 283
    single-character I/O, 300-301
    versus strings, 103
charcode.c program, 76
chcount.c program, 262-264
checking.c program, 320-323
circular queue, 808
clang command, 18
classes, storage, 511-513
  automatic, 517
  automatic variables, 518-522
  dynamic memory allocation and, 549-551
  functions and, 533-534
  register, 517
  register variables, 522
  scope, 513-515
  static variables, 522-524
  static with external linkage, 517
  static with internal linkage, 517
  static with no linkage, 517

Classic C, 8

clipping bits (bitwise operators), 682-683

code
  executable files, 14-18
  libraries, 14-18
  object code files, 14-18
  source code, 14
  startup, 15
  writing, 11
colldays.c program, 246-248

combination redirection, 309-310

comma format, 136

comma operator, 214-218
  for loop and, 216

command-line
  arguments, 497
    integrated environment, 500
    Macintosh, 500
    standard I/O, 569-570
  compilers, 19
  redirection, 310

comments, 13
  first.c program, 33-34

compare.c program, 476-477

comparisons, pointers, 411

compatibility, pointers, 421-423

compback.c program, 477-479

compft.c program, 198-199

compilers, 3, 11-12
  Borland C++ Compiler 5.5, 19
  cc, 17
  command-line, 19
  GCC, 18
  languages, 7
  linkers, 15
  system requirements, 24
  translation and, 712

compiling

Apple IDE, multiple source code files
  and, 362-363
conditional, 731
  #elif directive, 736-737
  #else directive, 732-733
  #endif directive, 732-733
  #error directive, 738-740
  #if directive, 736-737
  #ifndef directive, 732-733
  #line directive, 738-740
  predefined macros, 737-740

DOS command-line, multiple source code files
  and, 362
header files, multiple source code files
  and, 363-367
Linux systems, multiple source code files
  and, 362-32
modules, 14
Unix and, 16-18
Unix systems, multiple source code files and, 362
Windows, multiple source code files and, 362-363
complex types, 85
complit.c program, 632-633
compound literals, 431
arrays, 432-434
structures and, 631-633
compound statements (blocks), 171-173
conditional compilation, 731
#elif directive, 736-737
#else directive, 732-733
#endif directive, 732-733
#error directive, 738-740
#if directive, 736-737
#ifdef directive, 732-733
#line directive, 738-740
macros, predefined, 737-40
conditional operators, 911-912
conditional (?:) operator, 272-273
const keyword, 109, 148
arrays, 385
protecting, 415-417
sizes and, 431
C++ compared to C, 423
constants created, 716
formal parameters, 413-415
constant type qualifier, 552
global data, 553-554
parameter declarations, 552-553
pointers and, 552-553
constants, 57-59
character constants, initialization, 72-73
enum keyword, 649-650
expressions, array declaration, 544
floating-point, 81-82
int, 64
long, 68
long long, 68
manifest, 109-110
#define directive, 713
preprocessor and, 106-112
redefining, 717-718
string constants, 442-443
double quotation marks, 465
symbolic, 106-111
when to use, 716
contents of arrays, protecting, 412-417
continue statement, loops, 274-277
control strings, 115-114
scanf( ), 128
conversion specifiers, 112-113
mismatched conversions, 122-124
modifiers, 116-121, 129
conversions. See also type conversions
string-to-number, 500-503
copy1.c program, 482-484
copy2.c program, 484-485
copy3.c program, 486-487
CopyToNode( ) function, 799
count.c program, 569
counting loops, 207-208
CPU (central processing unit), 5
c-type.h
character functions, 252-253, 495
strings, 495
Cygwin, 19
cypher1.c program, 250-252
data keywords, 59-60

data objects, 147

data representation, 773-774
  films1.c program, 775-777
  interfaces
    building, 789-793
    defining, 805-806
    implementing, 796-802, 806-810
    using, 793-796

data types, 35. See also ADT (abstract data type)
  basic, 87
  _Bool, 203-204
  int, 62-65
  mismatches, 89
  size_t, 158

day_mon1.c program, 385-386

day_mon2.c program, 387-388

day_mon3.c program, 401

days[ ] array, 385

directives
  parameters, 403
  pointers, 544
  variable expressions, 544
  pointers, 372-373
  declaring variables, 37, 57, 102
    char type, 72
    int, 63
  decrement (--) operator, 164-166
  decrementing pointers, 410-411
  #define statement, 109, 136
    arguments, 718
  ## operator, 722-723
  function-like macros, 718
  mac_arg.c program, 719-721
    strings from macro arguments, 721-722
  variadic macros, 723-724
  enumerations instead, 701
  manifest constants, 713

directives

design features, 2

designated initializers, 388-390

designing the program, 11

dice rolling example, 538-543

diceroll.c file, 539-540

diceroll.h file, 540

differencing between pointers, 411

directives
  #elif, 736-737
  #else, 732-733
  #endif, 732-733
  first.c program, 31-32
#if, 736-737
#elif directive, 736-737
#else directive, 732-733
#else if statement, 253-257
else if statement, 253-257

#else directive, 732-733
endif directive, 732-733

disks, 5
displaying linked lists, 783-784
division (/ ) operator, 153-154
divisors.c program, 261-262
dlls (dynamic link libraries), 20
do while loop, 220-223
documentation
class, 13
fathm_ft.c program, 43
doubincl.c program, 735
double keyword, 60
double quotation marks, 465
macros and, 716
double type, 80-81
do_while.c program, 221
dualview.c program, 697-703
dummy( ) function, 663
dynamic memory allocation
storage classes and, 549-551
vlas and, 431, 458-459
dyn_arr.c program, 545-547

e

eatline( ) function, 664
echo.c program, 300
echo_eof.c program, 305-306
editors, Unix systems, 16
efficiency, 3
electric.c program, 255-257
elements
arrays, 824
linked lists, 824

#error directive, 738-740
 erase sequences, 73, 91, 94
echo.c program, 91
printf( ) function, 91-92
escape.c program, 91
EXCLUSIVE OR operator, 680
executable files, 14-18
execution, smooth, 325
exit( ) function, 570, 753-755
exit-condition loop, 220-223
EXIT_FAILURE macro, 570
EXIT_SUCCESS macro, 570
expressions, 167-168
  generic selection, 740-741
  logical, 911
  relational, 910
  false, 199-203
  true, 199-203
  values, 168
extern keyword, 536
external linkage, 515

file inclusion
  C library, 745
  #include directive, 726-730
file I/O
  fgets( ) function, 578-579
  fprintf( ) function, 576-578
  fputs( ) function, 578-579
  fscanf( ) function, 576-578
file-condensing program, 574-576
filenames, 14
files, 303
  binary, 566, 582
  binary view, 567
  description, 566
  EOF (end-of-file), 304-306
  executable, 14-18
  object code, 14-18
  portability, 582-583
  redirection, 307
  size, 566
  source code, 14
  structure contents, saving, 639-644
  text, 566
  versus binary, 582
  binary mode, 567
  text mode, 567
  text view, 567
films1.c program, 775-777
films3.c program, 794-796
first.c program, 28
  [ ] (braces), 34
  comments, 33-34
  data types, 35
  declarations, 34-35
  directives, 31-32
  header files, 31-32
  main( ) function, 32-33
name choices, 36
return statement, 40
stdio.h file, 31
fit( ) function, 470-471
flags.c program, 120
flc.c program, 433-434
flexibility of C, 3
flexible arrays, 633-636
flexmemb.c program, 634-636
float argument, conversion, 116
float keyword, 60
float type, 80-81
floating points, binary
   binary fractions, 676
   floating-point representation, 676
floating-point constants, 81-82
floating-point numbers, 61-57, 93
   overflow, 83-84
   round-off errors, 84
   underflow, 83-84
floating-point representation, 84
floating-point types, 94
   integer comparison, 60
floating-point values
   Borland C and, 608
   printing, 82-83
floating-point variables, declaring, 81
flushing output, 92-93
fopen( ) function, 570-572, 579, 584
for keyword, 209
for loop, 208-209
   arrays and, 228-230
   comma operator and, 216
   flexibility, 210-214
   selecting, 223-224
   structure, 209
for_cube.c program, 209-210
FORTRAN, 7
fprintf( ) function, 576-578
fputs( ) function, 578-579
   string input, 456-460
   string output, 465-466
fractional parts, 61
fractions, binary, 676
fread( ) function, 586-639
   example, 589-590
free( ) function, 545-547, 802
   importance of, 547-548
friend.c program, 615-618
fscanf( ) function, 576-578
fseek( ) function, 579-582
fsetpos( ) function, 583
ftell( ) function, 579-582
ftoa( ) function, 503
FullTree( ) function, 833
func_ptr.c program, 660-664
function declarations, 45
function pointers, 657
   addresses, 657
   ToUpper( ) function, 657-658
function scope, 514
function-like macros, 718, 731
functions, 4
   { } (braces), 34
   AddItem( ), 793, 800-801, 833-835
   AddNode( ), 833-835
   ANSI C, prototyping, 349-353
   arguments, 177-180, 340-342
      formal parameters, 342-343
      none, 352-353
   prototyping function with, 343
   unspecified, 352-353
arrays
multidimensional, 423-427
of structures, 637-638
assert(), 760-763
atan(), 747
atoi(), 500
black-box viewpoint, 345
body, 34, 40
butler(), 44-45, 177
calling
altering variables, 369-371
with argument, 343-344
nested calls, 468-469
calloc(), 548
character, ctype.h, 252-253
creating, 337-340
DeleteAll(), 843
DeleteElem(), 841-842
DeleteNode(), 841-842
description, 335
Dummy(), 663
eatline(), 664
EmptyTheList(), 793
EmptyTree(), 833
exit(), 570, 753-755
fclose(), 574
feof(), 589
ferror(), 589
fflush(), 585
fgetpos(), 583
fgets(), 456-461, 578-579
fit(), 470-471
fopen(), 570-572, 579, 584
fputs(), 456-460, 465-466, 578-579
fread(), 586-590, 639
free(), 545-548, 802
fseek(), 579-582
fsetpos(), 583
ftell(), 579-582
ftoa(), 503
FullTree(), 833
fwrite(), 586-590, 639
getc(), 572
getchar(), 20, 250-252
c250-sectio( ), 325-327
getinfo(), 624
get_long(), 322
c250-getnights(), 366
c250-gets(), 453-455, 460-461
c250-gets_s(), 460-461
headers, 40
imax(), 350-351
imin(), 345-348
InitializeList(), 793, 800
InitializeTree(), 833
inline, 725, 741-744
InOrder(), 842
input, 584
isalnum() function, 254
isblank(), 254
iscntrl(), 254
isdigit(), 254
isgraph(), 254
islower(), 254
isspace(), 254
isupper(), 254
isxdigit(), 254
itoa( ), 503
itobs(), 687
ListIsEmpty(), 800
ListIsFull(), 800-801
ListItemCount( ), 800-801
versus macros, 725-726
main( ), 30-33, 232, 337-340
makeinfo( ), 624-626
MakeNode( ), 833-835
malloc( ), 543-544, 628-631, 777
math library, 748
memcpy( ), 763-765
memmove( ), 763-765
menu( ), 366
mult_array( ), 415-416
multiple, 44-45
mycomp( ), 758-760
names, 336
uses, 664
_Noreturn (C11), 744
pointers
arrays of, 664
communication and, 373-375
declaring, 658
pound( ), 179
pow( ), 230
power( ), 233
printf( ), 30-31, 38-39
multiple values, 43-44
print_name( ), 352-353
prototyping
ANSI C, 349-353
arguments and, 343
scope, 514-515
put1( ), 467
put2( ), 468
putc( ), 572
putchar( ), 250-252
puts( ), 442, 453-455, 464-465, 471
qsort( ), 657, 755-758
rand( ), 534, 819-820
rand0( ), 535
recursive, 353-355
returns, 356
statements, 356
variables, 355
return values, 233-234
rewind( ), 577, 643
rfact( ), 358
scanf( ), 58, 128-129
SeekItem( ), 833-835, 841-842
setvbuf( ), 584-586
s_gets( ), 461-462, 592
show( ), 659
show_array( ), 416
show_bstr( ), 687
showmenu( ), 663-664
show_n_char( ), 340-344
sprintf( ), 487-489
sqrt( ), 660, 747
srand( ), 536-538, 542, 820
starbar( ), 337-340
storage classes, 533-534
strcat( ), 471-473, 489
strchr( ), 490, 664
strcmp( ), 475-480, 489
strcpy( ), 482-485, 489
strlen( ), 101-105, 469-471, 490
strncpy( ), 482-489
strncmp( ), 489
strncpy( ), 482-489
strpbrk( ), 490
strstr( ), 490
strtod( ), 503
strtol( ), 503
strtoul( ), 503
structure, 339
sum( ), 402
sump( ), 405
time( ), 538, 654, 820
to_binary( ), 360
ToLeft( ), 835
ToLower( ), 663
tolower( ), 253
ToRight( ), 835
ToUpper( ), 657-659, 663
toupper( ), 253
Transpose( ), 663
Traverse( ), 793, 801, 842
Treenitems( ), 833
types, 348-349
ungetc( ), 585
up_and_down( ), 354-355
uses, 336
values, return keyword, 345-348
VLAs, two-dimensional argument, 428
void, 658
funds1.c program, 618-619
funds2.c program, 620
funds3.c program, 621-622
funds4.c program, 637-638
fwrite( ) function, 586-588, 639
example, 589-590
generic selection, 740-741
getc( ) function, 572
getchar( ), 28
end-of-file, 304
single-character I/O and, 300-301
getchar( ) function, 20, 250-252
get_choice( ) function, 325-327
getinfo( ) function, 624
get_long( ) function, 322
getnights( ) function, 366
gets( ) function, 453-455
string input, 460-461
getsputs.c program, 453-455
gets_s( ) function, string input, 460-461
goto statement, 287, 290
guess.c program, 312-314

header files
  compiling, multiple source code files
  and, 363-367
  example, 727
  first.c program, 31-32
  IDEs, 726
  #include directive, 726-727
  multiple inclusions, 727-728
  uses, 729-730
headers, functions, 40
hello.c program, 500-502
hexadecimal numbers, 65-66, 94, 677-678
  binary equivalents, 678
hotel.h, 365-366

G

gcc command, 18
GCC compiler, 18
general utilities library
  atexit( ) function, 753-755
  exit( ) function, 753-755
  qsort( ) function, 755-758
_Generic keyword, 740-741

H
IDE (integrated development environments), header files, 726
identifiers
  entity, 512
  reserved, 49-50
IDEs (integrated development environments), 19-21
#if directive, 736-737
if else pairings, 257-259
if else statement, 248-249, 291
  ?: (conditional) operator, 272-273
  switch statement comparison, 286-287
if statement, 246-248, 291
  if else comparison, 249
#ifdef directive, 732-733
ifdef.c program, 732-733
#ifndef directive, 733-735
images, bitmapped, 774
imaginary types, 85
imax( ) function, 350-351
imin( ) function, 345-348
#include directive
  C library file inclusion, 745-746
  file inclusion, 726-730
#include statement, 30-31
increment (++) operator, 160-164
incrementing pointers, 410
indefinite loops, 207-208
indexes, arrays, 384
indirect membership operator, 913
initialization
  arrays, 384-388
    multidimensional, 396
  character string arrays, 444-445
  structures, 606
  unions, 645
  variables, 63
InitializeList( ) function, 793, 800
InitializeTree( ) function, 833
inline definition, 744
inline functions, 725, 741-744
inline keyword, 744
InOrder( ) function, 842
input
  buffered, 301
  character, mixing with numeric, 314-317
  functions, 584
  keyboard, 304
    terminating, 302-306
  numbers, 323-324
  numeric mixed with character input, 314-317, 327-330
  redirection, 307-308
string
  buffer overflow, 455
  fgets( ) function, 456-460
  fputs( ) function, 456-460
  gets( ) function, 453-455
  gets_s( ) function, 460-461
  long, 455
  scanf( ) function, 462-463
  s_gets( ) function, 461-462
  space creation, 453
  user interface, 312-314
    numeric mixed with character, 314-317
validation, 299-300, 317-324
int arrays, in memory, 228
int constants, 64
int keyword, 60
<table>
<thead>
<tr>
<th><strong>int type</strong>, 30, 34, 62</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>constants</strong>, 75</td>
</tr>
<tr>
<td><strong>hexadecimal numbers</strong>, 65-66</td>
</tr>
<tr>
<td><strong>long</strong>, 66-67</td>
</tr>
<tr>
<td><strong>multiple</strong>, 67-68</td>
</tr>
<tr>
<td><strong>octal numbers</strong>, 65-66</td>
</tr>
<tr>
<td><strong>printing int values</strong>, 64</td>
</tr>
<tr>
<td><strong>short</strong>, 66-67</td>
</tr>
<tr>
<td><strong>unsigned</strong>, 66-67</td>
</tr>
<tr>
<td><strong>variable declaration</strong>, 63</td>
</tr>
<tr>
<td><strong>intconv.c program</strong>, 122-123</td>
</tr>
</tbody>
</table>

**integers**, 61

- **binary**, 674-675
- **floating-point type comparison**, 60
- **mixing with floating types**, 124
- **overflow**, 69
- **pointers**, 410
  - **subtracting**, 410
- **properties**, 787
- **signed**, 675-676
- **union as**, 697

**integrated environment, command-line arguments**, 500

**interactive programs**, 58

**interchange() function**, 369-371

**interfaces**
- **binary search tree**, 830-832
- **building, ADTs and**, 789-793
- **defining**, 805-806
- **functions, implementing**, 810-815
- **implementing**, 796-802
- **using**, 793-796

**intermediate files**, 14

**internal linkage**, 515

**InTree() function**, 836-837

**inttypes.h**, 78

**inword flag**, 269-270

**I/O (input/output)**

- **file I/O**
  - **fprintf() function**, 576-578
  - **fscanf() function**, 576-578
  - **file-condensing program**, 574-576
  - **functions**, 299-300
  - **levels**, 568
  - **single character**, 300-301
  - **standard**, 568-569
    - **command-line arguments**, 569-570
    - **end-of-file**, 572-573
    - **fclose() function**, 574
    - **fopen() function**, 570-572
    - **getc() function**, 572
    - **pointers to files**, 574
    - **putc() function**, 572

**I/O package**, 32

- **isalnum() function**, 254
- **isalpha() function**, 254
- **isblank() function**, 254
- **iscntrl() function**, 254
- **isdigit() function**, 254
- **isgraph() function**, 254
- **islower() function**, 254
- **isupper() function**, 254
- **isxdigit() function**, 254
- **itoa() function**, 503
- **itobs() function**, 687
### J

**jove editor, 16**

### K

**keyboard input, 304**

**keystrokes, 23**

**keywords, 49-50**
- for, 209
- auto, 518
- char, 60
- const, 109, 385
  - C/C++ comparison, 423
  - formal parameters, 413-415
  - protecting arrays, 415-417
- data types, 59-60
- double, 60
- enum, 649
  - constants, 649-650
  - usage, 650-652
  - values, 650
- extern, 536
- float, 60
- Generic, 740-741
  - inline, 744
- int, 34, 60
- long, 60
- return, 230, 345-348
- short, 60
- struct, 604
- typedef, 158, 653, 654-656
  - #define statement and, 654
  - location, 653
  - variable names, 653-654
- unsigned, 60
- void, 178

### L

**labels, case, 284-285**

**languages**
- Classic C, 8
- compilers, 7
- high-level, 6
- K&R C, 8
- standards, 7-9

**length of strings, 101**

**lesser.c program, 345-348**

**lethead1.c program, 337-340**

**libraries, 14-18**
- assert, 760
  - assert( ) function, 760-763
- C library
  - automatic access, 745
  - descriptions, 746-747
  - file inclusion, 745
  - library inclusion, 745-746
- general utilities
  - atexit( ) function, 753-755
  - exit( ) function, 753-755
  - qsort( ) function, 755-758
- math, 747
  - ANSI C standard functions, 748
  - tgmath.h library, 752
  - trigonometry, 747-750
  - types, 750-752

**library inclusion (C library), 745-746**

**limitations of C, 4**

**#line directive, 738-740**

**linkage, 515-516**
- external, static variables, 524-529
- internal, static variables, 529-530
- variable scope and, 515
linked lists, 779-780
  arrays comparison, 824-828
  creating, 784-785
  displaying lists, 783-784
  elements, inserting, 824
  films2.c program, 781-785
  list memory, freeing, 785-786
  searches, 826
  several items, 781
  two items, 780
Linux systems, 18-19
  compiling, multiple source code files and, 362-32
  redirection, 307-311
  Windows/Linux option, 21
list.c program, 796-802
list.h header file, 791-793
ListIsEmpty() function, 800
ListIsFull() function, 800-801
ListItemCount() function, 800-801
lists
  ADTS, operations, 788
  linked
    arrays and, 824-828
    creating, 784-785
    displaying, 783-784
    freeing list memory, 785-786
    ordered, 826
literals, 81-82
  character string literals, 442-443
  compound, 431
    arrays, 432-434
    structures and, 631-633
  string literals, storage, 512
LLVM Project, 18
loccheck.c program, 367-368
logical operators, 264, 911
  alternative spellings, 265
  bitwise, 678-680
  order of evaluation, 266
  precedence, 265-266
  relational expressions, 291
long constants, 68
long double type, 80-81
long int type, 66-67
  printing, 70
long keyword, 60
long long constants, 68
long long int type, printing, 70
long strings, printing, 126-128
loops
  break statement, 277-279
  continue statement, 274-277
  counting, 207-208
  do while, 220-223
  entry condition, 195
  for, 210-214
  indefinite, 207-208
  introduction, 144-146
  nested, 224-226
  selecting, 223-224
  tail recursion and, 356-358
  while, 144, 190-191, 195
  terminating, 194-195
mac_arg.c program, 719-721
machine language, 6
Macintosh
  command-line arguments, 500
  Xcode, 21
macros
arguments, strings from, 721-722
containing macros, 715
double quotation marks and, 716
empty macros, 731
EXIT_FAILURE, 570
EXIT_SUCCESS, 570
function-like macros, 718, 731
versus functions, 725-726
object-like macros, 714, 731
predefined, 737-740
SQUARE, 719-720
strings, 715
tokens, 717
va_arg(), 766
va_copy(), 767
va_end(), 766
variadic, 723-724
va_start(), 766
mail.c program, 820-824
main() function, 30, 32-33, 232, 337-340
makeinfo() function, 624, 626
MakeNode() function, 833-835
malloc() function, 543-544
data representation, 777
new structures, 779
pointers, 628-631
structures, 628-631
VLAs and, 548-549
manifest constants, 109-110
#define preprocessor directive, 713
manybook.c program, 608-613
manydice.c file, 541-542
masks, bitwise operators, 680-681
math library, 747
  ANSI C standard functions, 748
tgmath.h library, 752
  trigonometry, 747-750
types, 750-752
membership operator (), 912
memcpy() function, 763-765
memmove() function, 763-765
memory, 5-6
allocated, 543
calloc() function, 548
dynamic, VLAs and, 548-549
free() function, 545-548
malloc() function, 543-544
storage classes and, 549-551
for a structure, 605
dynamic allocation, VLAs, 431
list, freeing, 785-786
storage classes, 511-513
structures and, 608
menu() function, 366
menuette.c program, 328-330
menus, 324
tasks, 324
MinGW, 19
min.sec.c program, 159
miscellaneous operators, 914
misuse.c program, 350-351
mode strings, fopen() function, 571
modifiers, declarations, 655-656
mod_str.c program, 495-497
modules, compiling, 14
modulus operator, 159-160
mult_array() function, 415-416
multidimensional arrays, 393-398
functions and, 423-427
pointers and, 417-427
two-dimensional, 394-396
  initializing, 397-398
multiplication (*) operator, 151-153
mycomp() function, 758-760

N
names1.c program, 622-624
names2.c program, 624-626
names3.c program, 629-631
names.h header file, 735
namespaces, shared, 652-653
names_st.h header file, 727
naming, 36
arrays, pointer notation, 402
functions, 336
uses of names, 664
pointer variables, 371
pointers, arrays and, 402
variables, 375
typedef, 653-654
nested function calls, 468-469
nested if statement, 259-262
nested loops, 224-226
nested structures, 613-615
newline character
preprocessor directives, 713
stripping, 603
no_data.c program, 386
nogood.c program, 46-49
nono.c program, 465
nonprinting characters, 73-76
_Noreturn functions (C11), 744
Notepad, 19
null character, 101, 459
scanf() function, 103
null pointer, 459
num variable, 30, 34

number input, mixing with character, 314-317, 327-330
numbers, 6
binary, octal digits, 677
bits, values, 674
decimal points, 57
decimal system, 674
floating-point, 57-61
hexadecimal, 65-66, 94, 677-678
input, 323-324
octal, 65-66, 94
order number bases, 676-678

O
object code files, 14-18
object-like macros, 714, 731
octal numbers, 65-66, 94, 677
one's complement, 679
online resources, 905-906
operators
#, 713
##, 722
AND, 679
+ (addition), 149
= (assignment), 146-149, 202
?: (conditional), 272-273
-- (decrement), 164-166
== (equality), 191
++ (increment), 166
* (indirection), 371-372
. (membership), 912
* (multiplication), 151-153
== (relational), 202
-/+(sign operators), 150
- (subtraction), 149-150
/ (division), 153-154
< (redirection), 308
> (redirection), 308
arithmetic, 908
assignment, 910-911
  %=, 215
  *=, 215
  +=, 215
  -=, 215
  /=, 215
binary, 150
bitwise, 913-914
  binbit.c program, 686-687
  bit fields and, 696-703
  clearing bits, 682-683
  logical, 678-680
  masks, 680-681
  setting bits, 681-682
  shift operators, 684-685
  toggling bits, 683
  value checking, 683-684
C, 908
comma operator, 214-218
conditional, 911-912
EXCLUSIVE OR, 680
indirect membership, 913
logical, 264, 911
  alternative spellings, 265
  order of evaluation, 266
  precedence, 265-266
miscellaneous, 914
modulus, 159-160
OR, 679-680
pointer-related, 912
precedence, 154-155
  increment/decrement, 165-166
  logical operators, 265-266
  order of evaluation, 155-157, 266
  relational, 197, 910
  expressions, 910
  precedence, 205
sign, 912
sizeof, 158, 388
structure, 617-618, 647, 912-913
structure pointer, 913
unary, 150
  *, 406
  ++, 406
union, 912-913
OR operator, 679-680
order number bases, 676-678
order of operator evaluation, 155-157
  logical operators, 266
order.c program, 406-407
ordered lists, 826
output, 23
  binary, 586
  disappearing, 28, 57
  printf() function, 92-93
  redirection, 308-309
string
  fputs() function, 465-466
  printf() function, 466
  puts() function, 464-465
text, 586

P
paint.c program, 272-273
parameters
  arrays, declaring, 403
  const type qualifier, 552-553
  formal parameters
    const keyword, 413-415
    function arguments, 342-343
  pointers, 404-407
parentheses, pointers to arrays, 420
precedence of operators 1027

parta.c file, 532
partb.c file, 532-533
passing
  arguments, 124
  pointers, 412
  structure members, 618-619
  structures, as arguments, 621-622
period () character, 262
peripherals, 5
petclub.c program, 849-854
pizza.c program, 108
platinum.c program, 56-58
pnt_add.c program, 399-400
pointer-related operators, 912
pointers, 371, 407
  * (indirection) operator, 371-372
  addresses, 409
  arrays, 398
    comparison, 445-447
    declaration, 544
    differences, 447-449
    names, 402
    notation and, 402
    multidimensional, 417-427
    parentheses, 420
  assignment and, 409
  comparisons, 411
  compatibility, 421-423
  const type qualifier, 552-553
  constants
    as function parameter, 416
    value changes and, 415
  declaring, 372-373
  to functions, 658
  decrementing, 410-411
differencing, 411
  function, arrays of, 664
  function communication, 373-375
  function pointers, 657
    addresses, 657
    ToUpper( ) function, 657-658
  incrementing, 410
  integers, 410
  subtracting, 410
  malloc( ) function, 628-631
  null, 459
  operations, 408-412
  parameters, 404-407
  passing, 412
  standard files (I/O), 574
  strcpy( ) function, 485
  strings, sorting, 493
  strings and, 451-452
  structures, 626-627
    character pointers, 627-628
    declaring, 617
    initializing, 617
    member acces, 617-618
    uninitialized, dereferencing, 411
  value finding, 409
  variables, names, 371
portability, 3, 582-583
postage.c program, 216
postfix, 163-164
pound( ) function, 179
pow( ) function, 230
power( ) function, 233
power.c program, 231-233
praise1.c program, 102
praise2.c program, 104-105
precedence of operators, 154-155
  increment/decrement, 165-166
  logical operators, 265-266
  order of evaluation, 155-157
  relational operators, 205
predefined macros, 737-740
prefix, 163-164
preproc.c program, 713-718
preprocessor
  constants and, 106-112
directives, newline character, 713
  identifiers and, 731
print_name( ) function, 352-353
print1.c program, 64-65
print2.c program, 70-71
printf( ) function, 30-31, 38-39
  * modifier, 133-135
  %f specifier, 57
  arguments, 89-91, 114
  conversion specifications, 112-113
    mismatched conversions, 122-124
  modifiers, 116-121
escape sequences, 91-92
flags, 118
multiple values, 43-44
output, 92-93
return value, 126
usage tips, 135-136
printing
  char type and, 76
  floating-point values, 82-83
  int values, 64
  long long types, 70
  long types, 70
  short types, 70
  strings, 102-103
    long strings, 126-128
  unsigned types, 70
printout.c program, 112-114
prntval.c program, 126
program jumps, 290
program state, 49
programmers, 3
programming
  books, 907
code, writing, 11
  commenting, 13
  compiling, 11-12
debugging, 12
design, 11
maintenance, 13
objectives, 10
running the program, 12
seven steps,
testing, 12
programs
  readability, 41-42
  structure, 40
protecting array contents, 412-417
proto.c program, 351-352
prototyping functions
  ANSI C, 349-353
    arguments and, 343
    scope, 514-515
ptr_ops.c program, 408-409
put1( ) function, 467
put2( ) function, 468
putc( ) function, 572
putchar( ) function, 250-252
  single-character I/O and, 300-301
put_out.c program, 464-465
put_put.c program, 468-469
puts( ) function, 442
  null character and, 471
  string input, 453-455
  string output, 464-465
qsort( ) function, 657, 755-758
queue abstract data type, 804
  array as queue, 806
circular queue, 808
  interface, defining, 805-806
  simulations, 818-824
testing queue, 815-817
queue.c implementation file, 813-815
queue.h interface header file, 809-810
quotation marks, double, 465
ragged arrays, 450
rain.c program, 395-396
RAM (random access memory), 5
rand( ) function, 534, 819, 820
rand0( ) function, 535
randbin.c program, 593-594
random access
  binary I/O, 593-594
  fgetpos( ) function, 583
  fopen( ) function, 579
  fseek( ) function, 579-582
  fsetpos( ) function, 583
  ftell( ) function, 579-582
ranges, && operator, 267-268
readability, 41-42
rectangular arrays, 450
rect_pol.c program, 749-750
recur.c program, 354-355
recursion, 353-355
  Fibonacci numbers and, 360
  pros/cons, 360-361
  returns, 356
reversal and, 358-360
  statements, 356
tail recursion, 356-358
up_and_down( ) function, 354-355
variables, 355
redefining constants, 717-718
redirection, 307
  < operator, 308
  > operator, 308
  combination, 309-310
  command-line, 310
  input, 307-308
  output, 308-309
re ducto.c program, 574-576
reference books, 908
register variables, storage classes, 522
relational expressions
  false, 199-203
  logical operator and, 291
  true, 199-203
relational operators, 197, 910
  ==, 191
  expressions, 910
  precedence, 205
repeat.c program, 498-499
reserved identifiers, 49-50
resources
  books
    C++, 907
    C language, 907
    programming, 907
    reference, 908
    online, 905-906
restrict type qualifier, 555-556
return keyword, 230, 345-348
return statement, 40
return values
  functions, 233-234
  printf( ) function, 126
  scanf( ) function, 133
reversal, recursion and, 358-360
reverse.c program, 579-580
rewind( ) function, 577, 643
rfact( ) function, 358
Ritchie, Dennis, 1
routines, library routines, 15
rows1.c program, 224-225
running.c program, 180-181
  S
samples, book inventory, 601-602
scanf( ) function, 58, 128-129
  arguments, 89-91
  conversion specifiers, 129
  format string, regular characters, 132-133
  input, 129-132
  null character, 103
  return value, 133
  while loop and, 191-193
scope
  block, 514
  function, 514
  function prototypes, 514-515
  linkage, 515-516
  storage classes, 513-515
scores_in.c program, 228-230
searches
  binary, 826-827
  binary search trees, 828-829
    adding items, 833-836
    AddItem( ) function, 833-837
  AddNode( ) function, 833-835
ADT, 829-843
DeleteAll( ) function, 843
DeleteItem( ) function, 836-837, 841-842
DeleteNode( ) function, 841-842
deleting items, 837-842
deleting nodes, 840-841
emptying, 843
EmptyTree( ) function, 833
finding items, 836-837
FullTree( ) function, 833
InitializeTree( ) function, 833
interface, 830-832
InTree( ) function, 836-837
MakeNode( ) function, 833-835
SeekItem( ) function, 833-837, 841-842
tips, 854-856
ToLeft( ) function, 835
ToRight( ) function, 835
traversing trees, 842
TreeItems( ) function, 833
linked lists, 826
SeekItem( ) function, 833-837, 841-842
selection sort algorithm, 494-495
semantic errors, 47-48
sequence points, statements, 170-171
setting bits (bitwise operators), 681-682
setvbuf( ) function, 584-586
s_gets( ) function, 592
  string input, 461-462
shared namespaces, 652-653
shift operators (bitwise), 684-685
short int type, 66-67
  printing, 70
short keyword, 60
show() function, 659
show_array() function, 416
show_bstr() function, 687
showchar2.c program, 316-317
showf_pt.c program, 82-83
showmenu() function, 663, 664
show_n_char() function, 340-344
side effects, statements, 170-171
sign operators, 912
sign operators (+/-), 150
signed integers, 675-676
signed types, 93
char, 77
simulations, queue package, 818-824
single-character I/O, 300-301
single-character reading, 283
sizeof operator, 158, 388
sizeof.c program, 158
size_t type, 158
skip2.c program, 134-135
skippart.c program, 274-276
somedata.c program, 387
sort_str.c program, 491-493
sorting, strings, 491
   pointers, 493
   selection sort algorithm, 494-495
source code
   files, 14
text files, 19
two or more files when compiling, 361-367
sprintf() function, 487-489
sqrt() function, 660, 747
SQUARE macro, 719-720
srand() function, 536-538, 542, 820
standard files (I/O), 568
   pointers to, 574
standard I/O, 568-569
   binary, random access and, 593-594
   command-line arguments, 569-570
   end-of-file, 572-573
fclose() function, 574
feof() function, 589
ferror() function, 589
fflush() function, 585
fopen() function, 570-572, 584
fread() function, 586-589
   example, 589-590
fwrite() function, 586-588
   example, 589-590
getc() function, 572
putc() function, 572
setvbuf() function, 584-586
ungetc() function, 585
starbar() function, 337-340
starsrch.c program, 481
startup code, 15
statements, 168-170
   assignment, 37-38
   break, 277-279, 282-283
   compound (blocks), 171-173
   continue, 274-277
   declarations, 34-35
   #define, 109
   else if, 253-257
goto, 287-290
if, 246-248, 291
if-else, 248-249, 272-273, 291
#include, 30-31
recursive functions, 356
return, 40
sequence points, 170-171
static class qualifier, 557
static variables, 534
  storage classes, 522-524
    external linkage, 524-529
    internal linkage, 529-530
stdarg.h file, variadic macros, 765-768
stdin stream, 307
stdio.h file, 31
  pointers to standard files, 574
storage, 5
  bit fields, 692-695
  numbers, 6
  string literals, 512
storage classes, 511-513
  arrays and, 386
  automatic, 517
  dynamic memory allocation, 549-551
  functions and, 533-534
  linkage, 515-516
  multiple files, 530
  register, 517
  scope, 513-515
  selecting, 534
  specifiers, 530-531
  static w/ external linkage, 517
  static w/ internal linkage, 517
  static w/ no linkage, 517
  storage duration, 516-517
  variables
    automatic, 518-522
    register, 522
  static with block scope, 522-524
  static with external linkage, 524-529
  static with internal linkage, 529-530
storage duration, 516-517
strcat() function, 471-473, 489
strchr() function, 490, 495-497, 664
strcmp() function, 475-480, 489
strcvt.c program, 502-503
strcpy() function, 482-484, 489
  pointsers, 485
  properties, 484-485
streams, 303
string functions
  sprintf(), 487-489
  strcat(), 471-473, 489
  strchr(), 490
  strcmp(), 475-480, 489
  strcpy(), 482-484, 489
  properties, 484-485
  strlen(), 469-471, 490
  strncat(), 473-474, 489
  strncpy(), 489
  strncpy(), 482-489
  strstr(), 490
  strspn(), 490
string input
  buffer overflow, 455
  fgets() function, 456-460
  fputs() function, 456-460
  gets() function, 453-455
  gets_s() function, 460-461
  long, 455
  scanf() function, 462-463
  s_gets() function, 461-462
  space creation, 453
string literals, storage, 512

string output
  puts( ) function, 465-466
  printf( ) function, 466
  puts( ) function, 464-465

strncf.c program, 121

string.h library
  memcpy( ) function, 763-765
  memmove( ) function, 763-765

strings, 102-103
  character string arrays, 444-445, 449-451
  character string literals, 442-443
  character strings, 101, 227, 441
  versus characters, 103
  constants, 442-443
    double quotation marks, 465
  control strings, 115-114
  defining, within program, 442-452
  displaying, 442
  length, 101
  long strings, printing, 126-128
  from macro arguments, 721-722
  macros, 715
  mode strings, fopen( ) function, 571
  pointers and, 451-452
  printing, 102-103
    long strings, 126-128
  puts( ) function, 442
  regular characters, 132-133
  sorting, 491
    pointers, 493
    selection sort algorithm, 494-495

strings1.c program, 442

string-to-number conversions, 500-503

strlen( ) function, 101, 103-105, 469-471, 490

strncat( ) function, 473-474, 489

strncmp( ) function, 489

strncpy( ) function, 482-489

strpbrk( ) function, 490

strptr.c program, 443

strstr( ) function, 490

strtod( ) function, 503

strtol( ) function, 503

strtoul( ) function, 503

struct keyword, 604

structure declaration
  initialization, 606
  initializers, 607-608
  member access, 607
  memory allocation, 605
  struct keyword, 604
  variables, defining, 605-608

structure operators, 912-913

structure pointer operator, 913

structures
  address, 619-620
  allocating in a block, 778
  anonymous, 636-637
  arrays, 607
    declaring, 611
    functions, 637-638
    members, 612
  arrays of, 608
  binary tree, 644
  character arrays, 627-628
  character pointers, 627-628
  compound literals and, 631-633
  malloc( ) function, 628-631
  members, passing, 618-619
  memory and, 608
  nested, 613-615
1034  structures
operator, 617-618
operators, 647
passing as argument, 621-622
pointers to, 615-616, 626-627
declaring, 617
initializing, 617
member access, 617-618
saving contents to file, 639-644
union as, 697
subst.c program, 722
subtraction (-) operator, 149-150
sum( ) function, 402
structure addresses, 619-620
sum_arr1.c program, 403-404
sum_arr2.c program, 405-407
summing.c program, 190-191
sump( ) function, 405
swap3.c program, 373-375
sweetie1.c program, 207-208
sweetie2.c program, 208
switch statement, 280-283, 291
if else statement comparison, 286-287
symbolic constants, 106-111
when to use, 716
symbols
*/ , 30, 33-34
/* , 30
syntax errors, 46-47
syntax points, while loop, 195-197
system requirements, 24

T
tail recursion, 356-358
talkback.c program, 100
tasks, 324
terminating while loop, 194-195
test_fit.c program, 470-471
testing programs, 12
text files, 566
  versus binary, 582
  binary mode, 567
  text mode, 567
  versus word process files, 19
text output, 586
text view (files), 567
tgmath.h library, 752
Thompson, Ken, 1
time( ) function, 538, 654, 820
to_binary( ) function, 360
toggling bits (bitwise operators), 683
tokens
  macros, 717
  translation and, 712-713
ToLeft( ) function, 835
ToLower( ) function, 663
toupper() function, 253
ToRight( ) function, 835
ToUpper() function, 657-659, 663
toupper() function, 253, 495-497
tracing, 48
translation
  compiler and, 712
  newline character and, 712
  tokens, 712-713
  whitespace characters, 713
Transpose( ) function, 663
Traverse( ) function, 793, 801, 842
tree.c implementation file, 843-849
tree.h header file, 830-832
TreeItems( ) function, 833
trigonometry, math library and, 747-750
trouble.c program, 201-203
two-dimensional array, 394-396
   initializing, 397-398

two_func.c program, 44-45

type conversions, 174-176
   cast operator, 176

type portability, 116

type qualifiers, ANSI C
   _Atomic, 556-557
   const, 552-554
   formal parameters, 557
   restrict, 555-556
   volatile, 554-555

type sizes, 86-88

typedef keyword, 158, 655-656
   #define statement and, 654
   location, 653
   variables, names, 653-654

typeface in book, 22

types
   enumerated, 649
   math library, 750-752

U

unary operators, 150
   &, 354
   *, 406
   ++, 406

#undef directive, 731

ungetc( ) function, 585

union operators, 912-913

unions
   anonymous, 647
   arrays of, 645
   initializing, 645
   as integer, 697
   as structure, 697

templates, tags and, 645
   uses, 646-647

Unix systems
   compiling, multiple source code files
   and, 362
   editors, 16
   file size, 566
   filenaming, 16
   redirection, 307-311

unsigned int type, 66-67
   printing, 70

unsigned keyword, 60

unsigned types, char, 77

unspecified arguments, 352-353

up_and_down( ) function, 354-355

usehotel.c
   control module, 363-364
   function support module, 364-365

use_q.c program, 816-817

user interface
   input
      buffered, 312-314
      numeric mixed with character,
      314-317
      menus, 324
      tasks, 324

V

-v option, 18

va_arg( ) macro, 766

va_copy( ) macro, 767

va_end( ) macro, 766

va_start( ) macro, 766

validation, input, 299-300, 317-324

va_list type variable, 765-766
values
arrays, assigning, 390
bit numbers, 674
bitwise operators, 683-684
changing, pointers to constants, 415
expressions, 168
pointers and, 409
return keyword, 345-348
variables, 375
varargs.c program, 767-768
vararr2d.c program, 429-431
variables, 59
addresses, 375
automatic, storage classes, 518-522
calling functions, altering, 369-371
declaring, 37, 57, 102
char type, 72
floating-point, 81
int, 63
expressions, array declaration, 544
floating-point, declaring, 81
initialization, 63
names, 375
typedef, 653-654
num, 30, 34
pointers
declaring, 372-373
names, 371
recursion, 355
register, storage classes, 522
static, 534
with block scope, 522-524
with external linkage, 524-529
with internal linkage, 529-530
structure, defining, 605-608
values, 375
variadic macros, 723-724
stdarg.h file, 765-768
varwld.c program, 133-134
vi editor, 16
Visual Studio, 20-21
VLAs (variable-length arrays), 427
dynamic memory allocation, 431, 548-549
functions, two-dimensional VLA argument, 428
malloc( ) function, 548
restrictions, 428
size, 428
support for, 428
void, 17
void function, assignment statements, 658
void keyword, 178
volatile type qualifier, 554-555
vowels.c program, 284-285

W
when.c program, 194-195
where.c program, 550-551
while loop, 144, 190-191
compound statement and, 172
conditions, 146
entry condition loop, 195
scanf( ) function, 191-193
selecting, 223-224
structure, 193
syntax points, 195-197
terminating, 194-195
while statement, 145, 170, 193
whitespace, 137
scanf( ) function, 129
translation and, 713
width.c program, 116-119
Win32 Console Application, 20
Windows Notepad, 19
Windows/Linux option, 21
word processor files versus text files, 19
wordcnt.c program, 270-271
word-counting program, 268-271
words, 60

X-Y-Z

X Window System, text editor, 16
Xcode, 21

zippo1.c program, 418-419
zippo2.c program, 420-421