



Stephen G. Kochan

Fourth Edition

Programming in C



FREE SAMPLE CHAPTER

SHARE WITH OTHERS



Programming in C

Fourth Edition

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:

Programming in Objective-C

Stephen G. Kochan

ISBN 978-0-321-96760-2

MySQL

Paul DuBois

ISBN-13: 978-0-321-83387-7

Linux Kernel Development

Robert Love

ISBN-13: 978-0-672-32946-3

Python Essential Reference

David Beazley

ISBN-13: 978-0-672-32978-4

PostgreSQL

Korry Douglas

ISBN-13: 978-0-672-32756-8

C++ Primer Plus

Stephen Prata

ISBN-13: 978-0-321-77640-2

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

Programming in C

Fourth Edition

Stephen G. Kochan

▼ 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

Programming in C, Fourth Edition
Copyright © 2015 by Pearson Education, Inc.

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

ISBN-13: 978-0-321-77641-9

ISBN-10: 0-321-77641-0

Library of Congress Control Number: 2014944082

Printed in the United States of America

First Printing: August 2014

Trademarks

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

Warning and Disclaimer

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

Special Sales

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact international@pearsoned.com.

Acquisitions

Editor

Mark Taber

Managing Editor

Sandra Schroeder

Project Editor

Mandie Frank

Copy Editor

Charlotte Kughen

Indexer

Brad Herriman

Proofreader

Debbie Williams

Technical Editor

Siddhartha Singh

Editorial Assistant

Vanessa Evans

Designer

Chuti Prasertsmith

Compositor

Mary Sudul



For my mother and father



Contents at a Glance

Introduction	1
1 Some Fundamentals	5
2 Compiling and Running Your First Program	11
3 Variables, Data Types, and Arithmetic Expressions	21
4 Program Looping	43
5 Making Decisions	65
6 Working with Arrays	95
7 Working with Functions	119
8 Working with Structures	163
9 Character Strings	193
10 Pointers	233
11 Operations on Bits	277
12 The Preprocessor	297
13 Extending Data Types with the Enumerated Data Type, Type Definitions, and Data Type Conversions	319
14 Working with Larger Programs	331
15 Input and Output Operations in C	345
16 Miscellaneous and Advanced Features	373
17 Debugging Programs	391
18 Object-Oriented Programming	413

A	C Language Summary	427
B	The Standard C Library	471
C	Compiling Programs with gcc	495
D	Common Programming Mistakes	499
E	Resources	505
	Index	509

Table of Contents

Introduction 1

1 Some Fundamentals 5

- Programming 5
- Higher-Level Languages 5
- Operating Systems 6
- Compiling Programs 7
- Integrated Development Environments 10
- Language Interpreters 10

2 Compiling and Running Your First Program 11

- Compiling Your Program 12
- Running Your Program 12
- Understanding Your First Program 13
- Displaying the Values of Variables 15
- Comments 17
- Exercises 19

3 Variables, Data Types, and Arithmetic Expressions 21

- Understanding Data Types and Constants 21
 - The Integer Type `int` 22
 - The Floating Number Type `float` 23
 - The Extended Precision Type `double` 23
 - The Single Character Type `char` 24
 - The Boolean Data Type `_Bool` 24
 - Type Specifiers: `long`, `long long`, `short`, `unsigned`, and `signed` 26
- Working with Variables 29
- Working with Arithmetic Expressions 30
 - Integer Arithmetic and the Unary Minus Operator 33
 - Combining Operations with Assignment: The Assignment Operators 39
 - Types `_Complex` and `_Imaginary` 40
- Exercises 40

4 Program Looping 43

Triangular Numbers 43

The `for` Statement 44

Relational Operators 46

Aligning Output 50

Program Input 51

Nested `for` Loops 53`for` Loop Variants 55The `while` Statement 56The `do` Statement 60The `break` Statement 62The `continue` Statement 62

Exercises 63

5 Making Decisions 65The `if` Statement 65The `if-else` Construct 69

Compound Relational Tests 72

Nested `if` Statements 74The `else if` Construct 76The `switch` Statement 83

Boolean Variables 86

The Conditional Operator 90

Exercises 92

6 Working with Arrays 95

Defining an Array 96

Using Array Elements as Counters 100

Generating Fibonacci Numbers 103

Using an Array to Generate Prime Numbers 104

Initializing Arrays 106

Character Arrays 108

Base Conversion Using Arrays 109

The `const` Qualifier 111

Multidimensional Arrays 113

Variable Length Arrays 115

Exercises 117

7 Working with Functions 119

- Defining a Function 119
- Arguments and Local Variables 123
 - Function Prototype Declaration 124
 - Automatic Local Variables 124
- Returning Function Results 126
- Functions Calling Functions Calling... 130
 - Declaring Return Types and Argument Types 133
 - Checking Function Arguments 135
- Top-Down Programming 137
- Functions and Arrays 137
 - Assignment Operators 141
 - Sorting Arrays 143
 - Multidimensional Arrays 146
- Global Variables 151
- Automatic and Static Variables 155
- Recursive Functions 158
- Exercises 161

8 Working with Structures 163

- The Basics of Structures 163
- A Structure for Storing the Date 164
 - Using Structures in Expressions 166
- Functions and Structures 169
 - A Structure for Storing the Time 175
- Initializing Structures 178
 - Compound Literals 178
- Arrays of Structures 180
- Structures Containing Structures 183
- Structures Containing Arrays 185
- Structure Variants 189
- Exercises 190

9 Character Strings 193

- Revisiting the Basics of Strings 193
- Arrays of Characters 194

Variable-Length Character Strings	197
Initializing and Displaying Character Strings	199
Testing Two Character Strings for Equality	202
Inputting Character Strings	204
Single-Character Input	206
The Null String	211
Escape Characters	215
More on Constant Strings	217
Character Strings, Structures, and Arrays	218
A Better Search Method	221
Character Operations	226
Exercises	229
10 Pointers	233
Pointers and Indirection	233
Defining a Pointer Variable	234
Using Pointers in Expressions	237
Working with Pointers and Structures	239
Structures Containing Pointers	241
Linked Lists	243
The Keyword <code>const</code> and Pointers	251
Pointers and Functions	252
Pointers and Arrays	258
A Slight Digression About Program Optimization	262
Is It an Array or Is It a Pointer?	262
Pointers to Character Strings	264
Constant Character Strings and Pointers	266
The Increment and Decrement Operators Revisited	267
Operations on Pointers	271
Pointers to Functions	272
Pointers and Memory Addresses	273
Exercises	275
11 Operations on Bits	277
The Basics of Bits	277
Bit Operators	278
The Bitwise AND Operator	279

The Bitwise Inclusive-OR Operator	281
The Bitwise Exclusive-OR Operator	282
The Ones Complement Operator	283
The Left Shift Operator	285
The Right Shift Operator	286
A Shift Function	286
Rotating Bits	288
Bit Fields	291
Exercises	295
12 The Preprocessor	297
The <code>#define</code> Statement	297
Program Extendability	301
Program Portability	302
More Advanced Types of Definitions	304
The <code>#</code> Operator	309
The <code>##</code> Operator	310
The <code>#include</code> Statement	311
System Include Files	313
Conditional Compilation	314
The <code>#ifdef</code> , <code>#endif</code> , <code>#else</code> , and <code>#ifndef</code> Statements	314
The <code>#if</code> and <code>#elif</code> Preprocessor Statements	316
The <code>#undef</code> Statement	317
Exercises	318
13 Extending Data Types with the Enumerated Data Type, Type Definitions, and Data Type Conversions	319
Enumerated Data Types	319
The <code>typedef</code> Statement	323
Data Type Conversions	325
Sign Extension	327
Argument Conversion	328
Exercises	329
14 Working with Larger Programs	331
Dividing Your Program into Multiple Files	331
Compiling Multiple Source Files from the Command Line	332

Communication Between Modules	334
External Variables	334
Static Versus Extern Variables and Functions	337
Using Header Files Effectively	339
Other Utilities for Working with Larger Programs	341
The <code>make</code> Utility	341
The <code>cvs</code> Utility	343
Unix Utilities: <code>ar</code> , <code>grep</code> , <code>sed</code> , and so on	343
15 Input and Output Operations in C	345
Character I/O: <code>getchar()</code> and <code>putchar()</code>	346
Formatted I/O: <code>printf()</code> and <code>scanf()</code>	346
The <code>printf()</code> Function	346
The <code>scanf()</code> Function	353
Input and Output Operations with Files	358
Redirecting I/O to a File	358
End of File	361
Special Functions for Working with Files	362
The <code>fopen</code> Function	362
The <code>getc()</code> and <code>putc()</code> Functions	364
The <code>fclose</code> Function	365
The <code>feof</code> Function	367
The <code>fprintf()</code> and <code>fscanf()</code> Functions	367
The <code>fgets()</code> and <code>fputs()</code> Functions	367
<code>stdin</code> , <code>stdout</code> , and <code>stderr</code>	368
The <code>exit</code> Function	369
Renaming and Removing Files	370
Exercises	371
16 Miscellaneous and Advanced Features	373
Miscellaneous Language Statements	373
The <code>goto</code> Statement	373
The <code>null</code> Statement	374
Working with Unions	375
The Comma Operator	378
Type Qualifiers	379
The <code>register</code> Qualifier	379

The volatile Qualifier	379
The restrict Qualifier	379
Command-line Arguments	380
Dynamic Memory Allocation	384
The <code>calloc()</code> and <code>malloc()</code> Functions	385
The <code>sizeof</code> Operator	385
The <code>free</code> Function	387
Exercises	389
17 Debugging Programs	391
Debugging with the Preprocessor	391
Debugging Programs with <code>gdb</code>	397
Working with Variables	400
Source File Display	401
Controlling Program Execution	402
Getting a Stack Trace	406
Calling Functions and Setting Arrays and Structures	407
Getting Help with <code>gdb</code> Commands	408
Odds and Ends	410
18 Object-Oriented Programming	413
What Is an Object Anyway?	413
Instances and Methods	414
Writing a C Program to Work with Fractions	416
Defining an Objective-C Class to Work with Fractions	417
Defining a C++ Class to Work with Fractions	421
Defining a C# Class to Work with Fractions	424
A C Language Summary	427
1.0 Digraphs and Identifiers	427
2.0 Comments	429
3.0 Constants	429
4.0 Data Types and Declarations	432
5.0 Expressions	442
6.0 Storage Classes and Scope	456
7.0 Functions	458
8.0 Statements	460
9.0 The Preprocessor	464

B The Standard C Library 471

- Standard Header Files 471
- String Functions 474
- Memory Functions 475
- Character Functions 476
- I/O Functions 477
- In-Memory Format Conversion Functions 482
- String-to-Number Conversion 483
- Dynamic Memory Allocation Functions 484
- Math Functions 485
- General Utility Functions 493

C Compiling Programs with gcc 495

- General Command Format 495
- Command-Line Options 496

D Common Programming Mistakes 499**E Resources 505**

- The C Programming Language 505
- C Compilers and Integrated Development Environments 506
- Miscellaneous 507

Index 509

About the Author

Stephen G. Kochan has been developing software with the C programming language for more than 30 years. He is the author of several best-selling titles on the C language, including *Programming in C*, *Programming in Objective-C*, and *Topics in C Programming*. He has also written extensively on Unix and is the author or coauthor of *Exploring the Unix System* and *Unix Shell Programming*.

Contributing Author, Fourth Edition

Dean Miller is a writer and editor with more than 20 years of experience in both the publishing and licensed consumer products businesses. He is coauthor of the most recent editions of *Sams Teach Yourself C in One Hour a Day*, and *Sams Teach Yourself Beginning Programming in 24 Hours*.

Acknowledgments

I wish to thank the following people for their help in the preparation of various versions of this text: Douglas McCormick, Jim Scharf, Henry Tabickman, Dick Fritz, Steve Levy, Tony Ianinno, and Ken Brown. I also want to thank Henry Mullish of New York University for teaching me so much about writing and for getting me started in the publishing business.

At Pearson, I'd like to thank Mark Taber and my project editor Mandie Frank. Thanks also to my copy editor, Charlotte Kughen, and my technical editor, Siddhartha Singh. Finally, I'd like to thank all the other people from Pearson who were involved on this project, even if I did not work with them directly.

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.

We welcome your comments. 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 and author, as well as your name and phone number or email address.

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 www.informit.com/register for convenient access to any updates, downloads, or errata that might be available for this book.

Introduction

The C programming language was pioneered by Dennis Ritchie at AT&T Bell Laboratories in the early 1970s. It was not until the late 1970s, however, that this programming language began to gain widespread popularity and support. This was because until that time C compilers were not readily available for commercial use outside of Bell Laboratories. Initially, C's growth in popularity was also spurred on in part by the equal, if not faster, growth in popularity of the Unix operating system. This operating system, which was also developed at Bell Laboratories, had C as its "standard" programming language. In fact, well over 90% of the operating system itself was written in the C language!

The enormous success of the IBM PC and its look-alikes soon made MS-DOS the most popular environment for the C language. As C grew in popularity across different operating systems, more and more vendors hopped on the bandwagon and started marketing their own C compilers. For the most part, their version of the C language was based on an appendix found in the first C programming text—*The C Programming Language*—by Brian Kernighan and Dennis Ritchie. Unfortunately, this appendix did not provide a complete and unambiguous definition of C, meaning that vendors were left to interpret some aspects of the language on their own.

In the early 1980s, a need was seen to standardize the definition of the C language. The American National Standards Institute (ANSI) is the organization that handles such things, so in 1983 an ANSI C committee (called X3J11) was formed to standardize C. In 1989, the committee's work was ratified, and in 1990, the first official ANSI standard definition of C was published.

Because C is used around the world, the International Standard Organization (ISO) soon got involved. They adopted the standard, where it was called ISO/IEC 9899:1990. Since that time, additional changes have been made to the C language. The most recent standard was adopted in 2011. It is known as ANSI C11, or ISO/IEC 9899:2011. It is this version of the language upon which this book is based.

C is a "higher-level language," yet it provides capabilities that enable the user to "get in close" with the hardware and deal with the computer on a much lower level. This is because, although C is a general-purpose structured programming language, it was originally designed with systems programming applications in mind and, as such, provides the user with an enormous amount of power and flexibility.

This book proposes to teach you how to program in C. It assumes no previous exposure to the language and was designed to appeal to novice and experienced programmers alike. If you have previous programming experience, you will find that C has a unique way of doing things that probably differs from other languages you have used.

Every feature of the C language is treated in this text. As each new feature is presented, a small *complete* program example is usually provided to illustrate the feature. This reflects the overriding philosophy that has been used in writing this book: to teach by example. Just as a picture is worth a thousand words, so is a properly chosen program example. If you have access to a computer that supports the C programming language, you are strongly encouraged to download and run each program presented in this book and to compare the results obtained on your system to those shown in the text. By doing so, not only will you learn the language and its syntax, but you will also become familiar with the process of typing in, compiling, and running C programs.

You will find that program readability has been stressed throughout the book. This is because I strongly believe that programs should be written so that they can be easily read—either by the author or by somebody else. Through experience and common sense, you will find that such programs are almost always easier to write, debug, and modify. Furthermore, developing programs that are readable is a natural result of a true adherence to a structured programming discipline.

Because this book was written as a tutorial, the material covered in each chapter is based on previously presented material. Therefore, maximum benefit will be derived from this book by reading each chapter in succession, and you are highly discouraged from “skipping around.” You should also work through the exercises that are presented at the end of each chapter before proceeding on to the next chapter.

Chapter 1, “Some Fundamentals,” which covers some fundamental terminology about higher-level programming languages and the process of compiling programs, has been included to ensure that you understand the language used throughout the remainder of the text. From Chapter 2, “Compiling and Running Your First Program,” on, you will be slowly introduced to the C language. By the time Chapter 15, “Input and Output Operations in C,” rolls around, all the essential features of the language will have been covered. Chapter 15 goes into more depth about I/O operations in C. Chapter 16, “Miscellaneous and Advanced Features,” includes those features of the language that are of a more advanced or esoteric nature.

Chapter 17, “Debugging Programs,” shows how you can use the C preprocessor to help debug your programs. It also introduces you to interactive debugging. The popular debugger `gdb` was chosen to illustrate this debugging technique.

Over the last decade, the programming world has been abuzz with the notion of object-oriented programming, or OOP for short. C is not an OOP language; however, several other programming languages that are based on C are OOP languages. Chapter 18, “Object-oriented Programming,” gives a brief introduction to OOP and some of its terminology. It also gives a brief overview of three OOP languages that are based on C, namely C++, C#, and Objective-C.

Appendix A, “C Language Summary,” provides a complete summary of the language and is provided for reference purposes.

Appendix B, “The Standard C Library,” provides a summary of many of the standard library routines that you will find on all systems that support C.

Appendix C, “Compiling Programs with `gcc`,” summarizes many of the commonly used options when compiling programs with GNU’s C compiler `gcc`.

In Appendix D, “Common Programming Mistakes,” you’ll find a list of common programming mistakes.

Finally, Appendix E, “Resources,” provides a list of resources you can turn to for more information about the C language and to further your studies.

This book makes no assumptions about a particular computer system or operating system on which the C language is implemented. The text makes brief mention of how to compile and execute programs using the popular GNU C compiler `gcc`.

This page intentionally left blank

3

Variables, Data Types, and Arithmetic Expressions

The true power of programs you create is their manipulation of data. In order to truly take advantage of this power, you need to better understand the different data types you can use, as well as how to create and name variables. C has a rich variety of math operators that you can use to manipulate your data. In this chapter you will cover:

- The `int`, `float`, `double`, `char`, and `_Bool` data types
- Modifying data types with `short`, `long`, and `long long`
- The rules for naming variables
- Basic math operators and arithmetic expressions
- Type casting

Understanding Data Types and Constants

You have already been exposed to the C basic data type `int`. As you will recall, a variable declared to be of type `int` can be used to contain integral values only—that is, values that do not contain decimal places.

The C programming language provides four other basic data types: `float`, `double`, `char`, and `_Bool`. A variable declared to be of type `float` can be used for storing floating-point numbers (values containing decimal places). The `double` type is the same as type `float`, only with roughly twice the precision. The `char` data type can be used to store a single character, such as the letter '`a`', the digit character '`6`', or a semicolon (`'.'`) (more on this later). Finally, the `_Bool` data type can be used to store just the values `0` or `1`. Variables of this type are used for indicating an on/off, yes/no, or true/false situation. These one-or-the-other choices are also known as binary choices.

In C, any number, single character, or character string is known as a *constant*. For example, the number `58` represents a constant integer value. The character string "`Programming in C`"

`is fun.\n"` is an example of a constant character string. Expressions consisting entirely of constant values are called *constant expressions*. So, the expression

```
128 + 7 - 17
```

is a constant expression because each of the terms of the expression is a constant value. But if `i` were declared to be an integer variable, the expression

```
128 + 7 - i
```

would not represent a constant expression because its value would change based on the value of `i`. If `i` is 10, the expression is equal to 125, but if `i` is 200, the expression is equal to -65.

The Integer Type `int`

In C, an integer constant consists of a sequence of one or more digits. A minus sign preceding the sequence indicates that the value is negative. The values 158, -10, and 0 are all valid examples of integer constants. No embedded spaces are permitted between the digits, and values larger than 999 cannot be expressed using commas. (So, the value 12,000 is not a valid integer constant and must be written as 12000.)

Two special formats in C enable integer constants to be expressed in a base other than decimal (base 10). If the first digit of the integer value is a 0, the integer is taken as expressed in *octal* notation—that is, in base 8. In that case, the remaining digits of the value must be valid base-8 digits and, therefore, must be 0–7. So, to express the value 50 in base 8 in C, which is equivalent to the value 40 in decimal, the notation 050 is used. Similarly, the octal constant 0177 represents the decimal value 127 ($1 \times 64 + 7 \times 8 + 7$). An integer value can be displayed at the terminal in octal notation by using the format characters %o in the format string of a `printf()` statement. In such a case, the value is displayed in octal without a leading zero. The format character %#o does cause a leading zero to be displayed before an octal value.

If an integer constant is preceded by a zero and the letter `x` (either lowercase or uppercase), the value is taken as being expressed in hexadecimal (base 16) notation. Immediately following the letter `x` are the digits of the hexadecimal value, which can be composed of the digits 0–9 and the letters a–f (or A–F). The letters represent the values 10–15, respectively. So, to assign the hexadecimal value FFEF0D to an integer variable called `rgbColor`, the statement

```
rgbColor = 0xFFEF0D;
```

can be used. The format characters %x display a value in hexadecimal format without the leading 0x, and using lowercase letters a–f for hexadecimal digits. To display the value with the leading 0x, you use the format characters %#x, as in the following:

```
printf ("Color is %#x\n", rgbColor);
```

An uppercase x, as in %x or %#x, can be used to display the leading x and the hexadecimal digits that follow using uppercase letters.

Storage Sizes and Ranges

Every value, whether it's a character, integer, or floating-point number, has a *range* of values associated with it. This range has to do with the amount of storage that is allocated to store a particular type of data. In general, that amount is not defined in the language. It typically depends on the computer you're running, and is, therefore, called *implementation- or machine-dependent*. For example, an integer might take up 32 bits on your computer, or perhaps it might be stored in 64. You should never write programs that make any assumptions about the size of your data types. You are, however, guaranteed that a minimum amount of storage will be set aside for each basic data type. For example, it's guaranteed that an integer value will be stored in a minimum of 32 bits of storage, which is the size of a "word" on many computers.

The Floating Number Type `float`

A variable declared to be of type `float` can be used for storing values containing decimal places. A floating-point constant is distinguished by the presence of a decimal point. You can omit digits before the decimal point or digits after the decimal point, but obviously you can't omit both. The values `3.`, `125.8`, and `- .0001` are all valid examples of floating-point constants. To display a floating-point value at the terminal, the `printf` conversion characters `%f` are used.

Floating-point constants can also be expressed in *scientific notation*. The value `1.7e4` is a floating-point value expressed in this notation and represents the value 1.7×10^4 . The value before the letter `e` is known as the *mantissa*, whereas the value that follows is called the *exponent*. This exponent, which can be preceded by an optional plus or minus sign, represents the power of 10 by which the mantissa is to be multiplied. So, in the constant `2.25e-3`, the `2.25` is the value of the mantissa and `-3` is the value of the exponent. This constant represents the value 2.25×10^{-3} , or `0.00225`. Incidentally, the letter `e`, which separates the mantissa from the exponent, can be written in either lowercase or uppercase.

To display a value in scientific notation, the format characters `%e` should be specified in the `printf()` format string. The `printf()` format characters `%g` can be used to let `printf()` decide whether to display the floating-point value in normal floating-point notation or in scientific notation. This decision is based on the value of the exponent: If it's less than `-4` or greater than `5`, `%e` (scientific notation) format is used; otherwise, `%f` format is used.

Use the `%g` format characters for displaying floating-point numbers—it produces the most aesthetically pleasing output.

A *hexadecimal* floating constant consists of a leading `0x` or `0X`, followed by one or more decimal or hexadecimal digits, followed by a `p` or `P`, followed by an optionally signed binary exponent. For example, `0x0.3p10` represents the value $3/16 \times 2^{10} = 0.5$.

The Extended Precision Type `double`

The `double` type is very similar to the `float` type, but it is used whenever the range provided by a `float` variable is not sufficient. Variables declared to be of type `double` can store roughly

twice as many significant digits as can a variable of type `float`. Most computers represent `double` values using 64 bits.

Unless told otherwise, all floating-point constants are taken as `double` values by the C compiler. To explicitly express a `float` constant, append either an `f` or `F` to the end of the number, as follows:

```
12.5f
```

To display a `double` value, the format characters `%f`, `%e`, or `%g`, which are the same format characters used to display a `float` value, can be used.

The Single Character Type `char`

A `char` variable can be used to store a single character.¹ A character constant is formed by enclosing the character within a pair of single quotation marks. So `'a'`, `';'`, and `'0'` are all valid examples of character constants. The first constant represents the letter *a*, the second is a semicolon, and the third is the character zero—which is not the same as the number zero. Do not confuse a character constant, which is a single character enclosed in single quotes, with a character string, which is any number of characters enclosed in double quotes.

The character constant `'\n'`—the newline character—is a valid character constant even though it seems to contradict the rule cited previously. This is because the backslash character is a special character in the C system and does not actually count as a character. In other words, the C compiler treats the character `'\n'` as a single character, even though it is actually formed by two characters. There are other special characters that are initiated with the backslash character. Consult Appendix A, “C Language Summary,” for a complete list.

The format characters `%c` can be used in a `printf()` call to display the value of a `char` variable at the terminal.

The Boolean Data Type `_Bool`

A `_Bool` variable is defined in the language to be large enough to store just the values `0` and `1`. The precise amount of memory that is used is unspecified. `_Bool` variables are used in programs that need to indicate a Boolean condition. For example, a variable of this type might be used to indicate whether all data has been read from a file.

By convention, `0` is used to indicate a false value, and `1` indicates a true value. When assigning a value to a `_Bool` variable, a value of `0` is stored as `0` inside the variable, whereas any nonzero value is stored as `1`.

To make it easier to work with `_Bool` variables in your program, the standard header file `<stdbool.h>` defines the values `bool`, `true`, and `false`. An example of this is shown in Program 5.10A in Chapter 5, “Making Decisions.”

¹ Appendix A discusses methods for storing characters from extended character sets, through special escape sequences, universal characters, and wide characters.

In Program 3.1, the basic C data types are used.

Program 3.1 Using the Basic Data Types

```
#include <stdio.h>

int main (void)
{
    int      integerVar = 100;
    float    floatingVar = 331.79;
    double   doubleVar = 8.44e+11;
    char     charVar = 'W';

    _Bool    boolVar = 0;

    printf ("integerVar = %i\n", integerVar);
    printf ("floatingVar = %f\n", floatingVar);
    printf ("doubleVar = %e\n", doubleVar);
    printf ("doubleVar = %g\n", doubleVar);
    printf ("charVar = %c\n", charVar);

    printf ("boolVar = %i\n", boolVar);

    return 0;
}
```

Program 3.1 Output

```
integerVar = 100
floatingVar = 331.790009
doubleVar = 8.440000e+11
doubleVar = 8.44e+11
charVar = W
boolVar = 0;
```

The first statement of Program 3.1 declares the variable `integerVar` to be an integer variable and also assigns to it an initial value of 100, as if the following two statements had been used instead:

```
int integerVar;
integerVar = 100;
```

In the second line of the program's output, notice that the value of 331.79, which is assigned to `floatingVar`, is actually displayed as 331.790009. In fact, the actual value displayed is dependent on the particular computer system you are using. The reason for this inaccuracy is the particular way in which numbers are internally represented inside the computer. You have probably come across the same type of inaccuracy when dealing with numbers on your pocket calculator. If you divide 1 by 3 on your calculator, you get the result .33333333, with perhaps

some additional 3s tacked on at the end. The string of 3s is the calculator's approximation to one third. Theoretically, there should be an infinite number of 3s. But the calculator can hold only so many digits, thus the inherent inaccuracy of the machine. The same type of inaccuracy applies here: Certain floating-point values cannot be exactly represented inside the computer's memory.

When displaying the values of `float` or `double` variables, you have the choice of three different formats. The `%f` characters are used to display values in a standard manner. Unless told otherwise, `printf()` always displays a `float` or `double` value to six decimal places rounded. You see later in this chapter how to select the number of decimal places that are displayed.

The `%e` characters are used to display the value of a `float` or `double` variable in scientific notation. Once again, six decimal places are automatically displayed by the system.

With the `%g` characters, `printf()` chooses between `%f` and `%e` and also automatically removes from the display any trailing zeroes. If no digits follow the decimal point, it doesn't display that either.

In the next-to-last `printf()` statement, the `%c` characters are used to display the single character 'w' that you assigned to `charVar` when the variable was declared. Remember that whereas a character string (such as the first argument to `printf()`) is enclosed within a pair of double quotes, a character constant must always be enclosed within a pair of single quotes.

The last `printf()` shows that a `_Bool` variable can have its value displayed using the integer format characters `%i`.

Type Specifiers: `long`, `long long`, `short`, `unsigned`, and `signed`

If the specifier `long` is placed directly before the `int` declaration, the declared integer variable is of extended range on some computer systems. An example of a `long int` declaration might be

```
long int factorial;
```

This declares the variable `factorial` to be a `long` integer variable. As with `floats` and `doubles`, the particular accuracy of a `long` variable depends on your particular computer system. On many systems, an `int` and a `long int` have the same range and either can be used to store integer values up to 32-bits wide ($2^{31} - 1$, or 2,147,483,647).

A constant value of type `long int` is formed by optionally appending the letter `L` (upper- or lowercase) onto the end of an integer constant. No spaces are permitted between the number and the `L`. So, the declaration

```
long int numberOfPoints = 131071100L;
```

declares the variable `numberOfPoints` to be of type `long int` with an initial value of 131,071,100.

To display the value of a `long int` using `printf()`, the letter `l` is used as a modifier before the integer format characters `i`, `o`, and `x`. This means that the format characters `%li` can be used to

display the value of a `long int` in decimal format, the characters `%lo` can display the value in octal format, and the characters `%lx` can display the value in hexadecimal format.

There is also a `long long` integer data type, so

```
long long int maxAllowedStorage;
```

declares the indicated variable to be of the specified extended accuracy, which is guaranteed to be at least 64 bits wide. Instead of a single letter `l`, two `ls` are used in the `printf` string to display `long long` integers, as in `"%lli"`.

The `long` specifier is also allowed in front of a `double` declaration, as follows:

```
long double US_deficit_2004;
```

A `long double` constant is written as a floating constant with the letter `l` or `L` immediately following, such as

```
1.234e+7L
```

To display a `long double`, the `L` modifier is used. So, `%Lf` displays a `long double` value in floating-point notation, `%Le` displays the same value in scientific notation, and `%Lg` tells `printf()` to choose between `%Lf` and `%Le`.

The specifier `short`, when placed in front of the `int` declaration, tells the C compiler that the particular variable being declared is used to store fairly small integer values. The motivation for using `short` variables is primarily one of conserving memory space, which can be an issue in situations in which the program needs a lot of memory and the amount of available memory is limited.

On some machines, a `short int` takes up half the amount of storage as a regular `int` variable does. In any case, you are guaranteed that the amount of space allocated for a `short int` will not be less than 16 bits.

There is no way to explicitly write a constant of type `short int` in C. To display a `short int` variable, place the letter `h` in front of any of the normal integer conversion characters: `%hi`, `%ho`, or `%hx`. Alternatively, you can also use any of the integer conversion characters to display `short ints`, due to the way they can be converted into integers when they are passed as arguments to the `printf()` routine.

The final specifier that can be placed in front of an `int` variable is used when an integer variable will be used to store only positive numbers. The declaration

```
unsigned int counter;
```

declares to the compiler that the variable `counter` is used to contain only positive values. By restricting the use of an integer variable to the exclusive storage of positive integers, the accuracy of the integer variable is extended.

An `unsigned int` constant is formed by placing the letter `u` (or `U`) after the constant, as follows:

```
0x00ffU
```

You can combine the letters `u` (or `U`) and `l` (or `L`) when writing an integer constant, so
`20000UL`

tells the compiler to treat the constant `20000` as an `unsigned long`.

An integer constant that's not followed by any of the letters `u`, `U`, `l`, or `L` and that is too large to fit into a normal-sized `int` is treated as an `unsigned int` by the compiler. If it's too small to fit into an `unsigned int`, the compiler treats it as a `long int`. If it still can't fit inside a `long int`, the compiler makes it an `unsigned long int`. If it doesn't fit there, the compiler treats it as a `long long int` if it fits, and as an `unsigned long long int` otherwise.

When declaring variables to be of type `long long int`, `long int`, `short int`, or `unsigned int`, you can omit the keyword `int`. Therefore, the `unsigned` variable `counter` could have been equivalently declared as follows:

```
unsigned counter;
```

You can also declare `char` variables to be `unsigned`.

The `signed` qualifier can be used to explicitly tell the compiler that a particular variable is a signed quantity. Its use is primarily in front of the `char` declaration, and further discussion is deferred until Chapter 13, “More on Data Types.”

Don't worry if the discussions of these specifiers seem a bit esoteric to you at this point. In later sections of this book, many of these different types are illustrated with actual program examples. Chapter 13 goes into more detail about data types and conversions.

Table 3.1 summarizes the basic data types and qualifiers.

Table 3.1 Basic Data Types

Type	Constant Examples	<code>printf</code> chars
<code>char</code>	'a', '\n'	%c
<code>_Bool</code>	0, 1	%i, %u
<code>short int</code>	—	%hi, %hx, %ho
<code>unsigned short int</code>	—	%hu, %hx, %ho
<code>int</code>	12, -97, 0xFFE0, 0177	%i, %x, %o
<code>unsigned int</code>	12u, 100U, 0xFFFFU	%u, %x, %o
<code>long int</code>	12L, -2001, 0xffffL	%li, %lx, %lo
<code>unsigned long int</code>	12UL, 100ul, 0xffffffffUL	%lu, %lx, %lo
<code>long long int</code>	0xe5e5e5e5LL, 50011	%lli, %llx, &ll0
<code>unsigned long long int</code>	12ull, 0xffffffffULL	%llu, %llx, %ll0
<code>float</code>	12.34f, 3.1e-5f, 0x1.5p10, 0x1P-1	%f, %e, %g, %a

Type	Constant Examples	<code>printf</code> chars
double	12.34, 3.1e-5, 0x.1p3	%f, %e, %g, %a
long double	12.341, 3.1e-51	%Lf, \$Le, %Lg

Working with Variables

Early computer programmers had the onerous task of having to write their programs in the binary language of the machine they were programming. This meant that computer instructions had to be hand-coded into binary numbers by the programmer before they could be entered into the machine. Furthermore, the programmer had to explicitly assign and reference any storage locations inside the computer's memory by a specific number or memory address.

Today's programming languages allow you to concentrate more on solving the particular problem at hand than worrying about specific machine codes or memory locations. They enable you to assign symbolic names, known as *variable names*, for storing program computations and results. A variable name can be chosen by you in a meaningful way to reflect the type of value that is to be stored in that variable.

In Chapter 2, “Compiling and Running Your First Program,” you used several variables to store integer values. For example, you used the variable `sum` in Program 2.4 to store the result of the addition of the two integers 50 and 25.

The C language allows data types other than just integers to be stored in variables as well, provided the proper declaration for the variable is made *before* it is used in the program. Variables can be used to store floating-point numbers, characters, and even *pointers* to locations inside the computer's memory.

The rules for forming variable names are quite simple: They must begin with a letter or underscore (`_`) and can be followed by any combination of letters (upper- or lowercase), underscores, or the digits 0–9. The following is a list of valid variable names.

```
sum
pieceFlag
i
J5x7
Number_of_moves
_sysflag
```

On the other hand, the following variable names are not valid for the stated reasons:

<code>sum\$value</code>	\$ is not a valid character.
<code>piece flag</code>	Embedded spaces are not permitted.
<code>3Spencer</code>	Variable names cannot start with a number.
<code>int</code>	int is a reserved word.

`int` cannot be used as a variable name because its use has a special meaning to the C compiler. This use is known as a reserved name or reserved word. In general, any name that has special significance to the C compiler cannot be used as a variable name. Appendix A provides a complete list of such reserved names.

You should always remember that upper- and lowercase letters are distinct in C. Therefore, the variable names `sum`, `Sum`, and `SUM` each refer to a different variable.

Your variable names can be as long as you want, although only the first 63 characters might be significant, and in some special cases (as described in Appendix A), only the first 31 characters might be significant. It's typically not practical to use variable names that are too long—just because of all the extra typing you have to do. For example, although the following line is valid

```
theAmountOfMoneyWeMadeThisYear = theAmountOfMoneyLeftAttheEndOfTheYear -  
    theAmountOfMoneyAtTheStartOfTheYear;
```

this line

```
moneyMadeThisYear = moneyAtEnd - moneyAtStart;
```

conveys almost as much information in much less space.

When deciding on the choice of a variable name, keep one recommendation in mind—don't be lazy. Pick names that reflect the intended use of the variable. The reasons are obvious. Just as with comments, meaningful variable names can dramatically increase the readability of a program and pay off in the debug and documentation phases. In fact, the documentation task is probably greatly reduced because the program is more self-explanatory.

Working with Arithmetic Expressions

In C, just as in virtually all programming languages, the plus sign (+) is used to add two values, the minus sign (-) is used to subtract two values, the asterisk (*) is used to multiply two values, and the slash (/) is used to divide two values. These operators are known as *binary* arithmetic operators because they operate on two values or terms.

You have seen how a simple operation such as addition can be performed in C. Program 3.2 further illustrates the operations of subtraction, multiplication, and division. The last two operations performed in the program introduce the notion that one operator can have a higher priority, or *precedence*, over another operator. In fact, each operator in C has a precedence associated with it. This precedence is used to determine how an expression that has more than one operator is evaluated: The operator with the higher precedence is evaluated first. Expressions containing operators of the same precedence are evaluated either from left to right or from right to left, depending on the operator. This is known as the *associative* property of an operator. Appendix A provides a complete list of operator precedences and their rules of association.

Program 3.2 Using the Arithmetic Operators

```
// Illustrate the use of various arithmetic operators

#include <stdio.h>

int main (void)
{
    int a = 100;
    int b = 2;
    int c = 25;
    int d = 4;
    int result;

    result = a - b;           // subtraction
    printf ("a - b = %i\n", result);

    result = b * c;           // multiplication
    printf ("b * c = %i\n", result);

    result = a / c;           // division
    printf ("a / c = %i\n", result);

    result = a + b * c;       // precedence
    printf ("a + b * c = %i\n", result);

    printf ("a * b + c * d = %i\n", a * b + c * d);

    return 0;
}
```

Program 3.2 Output

```
a - b = 98
b * c = 50
a / c = 4
a + b * c = 150
a * b + c * d = 300
```

After declaring the integer variables `a`, `b`, `c`, `d`, and `result`, the program assigns the result of subtracting `b` from `a` to `result` and then displays its value with an appropriate `printf()` call.

The next statement

```
result = b * c;
```

has the effect of multiplying the value of `b` by the value of `c` and storing the product in `result`. The result of the multiplication is then displayed using a `printf()` call that should be familiar to you by now.

The next program statement introduces the division operator—the slash. The result of 4, as obtained by dividing 100 by 25, is displayed by the `printf()` statement immediately following the division of `a` by `c`.

On some computer systems, attempting to divide a number by zero results in abnormal termination of the program.² Even if the program does not terminate abnormally, the results obtained by such a division will be meaningless.

In Chapter 5, you see how you can check for division by zero before the division operation is performed. If it is determined that the divisor is zero, an appropriate action can be taken and the division operation can be averted.

The expression

```
a + b * c
```

does not produce the result of 2550 (102×25); rather, the result as displayed by the corresponding `printf()` statement is shown as 150. This is because C, like most other programming languages, has rules for the order of evaluating multiple operations or terms in an expression. Evaluation of an expression generally proceeds from left to right. However, the operations of multiplication and division are given precedence over the operations of addition and subtraction. Therefore, the expression

```
a + b * c
```

is evaluated as

```
a + (b * c)
```

by the C programming language. (This is the same way this expression would be evaluated if you were to apply the basic rules of algebra.)

If you want to alter the order of evaluation of terms inside an expression, you can use parentheses. In fact, the expression listed previously is a perfectly valid C expression. Thus, the statement

```
result = a + (b * c);
```

could have been substituted in Program 3.2 to achieve identical results. However, if the expression

```
result = (a + b) * c;
```

were used instead, the value assigned to `result` would be 2550 because the value of `a` (100) would be added to the value of `b` (2) before multiplication by the value of `c` (25) would take place. Parentheses can also be nested, in which case evaluation of the expression proceeds outward from the innermost set of parentheses. Just be certain you have as many closed parentheses as you have open ones.

2. This happens using the gcc compiler under Windows. On Unix systems, the program might not terminate abnormally, and might give 0 as the result of an integer division by zero and “Infinity” as the result of a float division by zero.

You will notice from the last statement in Program 3.2 that it is perfectly valid to give an expression as an argument to `printf()` without having to first assign the result of the expression evaluation to a variable. The expression

```
a * b + c * d
```

is evaluated according to the rules stated previously as

```
(a * b) + (c * d)
```

or

```
(100 * 2) + (25 * 4)
```

The result of 300 is handed to the `printf()` routine.

Integer Arithmetic and the Unary Minus Operator

Program 3.3 reinforces what you just learned and introduces the concept of integer arithmetic.

Program 3.3 More Examples with Arithmetic Operators

```
// More arithmetic expressions
```

```
#include <stdio.h>

int main (void)
{
    int    a = 25;
    int    b = 2;

    float c = 25.0;
    float d = 2.0;

    printf ("6 + a / 5 * b = %i\n", 6 + a / 5 * b);
    printf ("a / b * b = %i\n", a / b * b);
    printf ("c / d * d = %f\n", c / d * d);
    printf ("-a = %i\n", -a);

    return 0;
}
```

Program 3.3 Output

```
6 + a / 5 * b = 16
a / b * b = 24
c / d * d = 25.000000
-a = -25
```

Extra blank spaces are inserted between `int` and the declaration of `a`, `b`, `c`, and `d` in the first four statements to align the declaration of each variable. This helps make the program more readable. You also might have noticed in each program presented thus far that a blank space was placed around each operator. This, too, is not required and is done solely for aesthetic reasons. In general, you can add extra blank spaces just about anywhere that a single blank space is allowed. A few extra presses of the spacebar prove worthwhile if the resulting program is easier to read.

The expression in the first `printf()` call of Program 3.3 reinforces the notion of operator precedence. Evaluation of this expression proceeds as follows:

1. Because division has higher precedence than addition, the value of `a` (25) is divided by 5 first. This gives the intermediate result of 5.
2. Because multiplication also has higher precedence than addition, the intermediate result of 5 is next multiplied by 2, the value of `b`, giving a new intermediate result of 10.
3. Finally, the addition of 6 and 10 is performed, giving a final result of 16.

The second `printf()` statement introduces a new twist. You would expect that dividing `a` by `b` and then multiplying by `b` would return the value of `a`, which has been set to 25. But this does not seem to be the case, as shown by the output display of 24. It might seem like the computer lost a bit somewhere along the way. The fact of the matter is that this expression was evaluated using integer arithmetic.

If you glance back at the declarations for the variables `a` and `b`, you will recall that they were both declared to be of type `int`. Whenever a term to be evaluated in an expression consists of two integers, the C system performs the operation using integer arithmetic. In such a case, all decimal portions of numbers are lost. Therefore, when the value of `a` is divided by the value of `b`, or 25 is divided by 2, you get an intermediate result of 12 and *not* 12.5 as you might expect. Multiplying this intermediate result by 2 gives the final result of 24, thus explaining the “lost” digit. Don’t forget that if you divide two integers, you always get an integer result. In addition, keep in mind that no rounding occurs, the decimal value is simply dropped, so integer division that ends up with 12.01, 12.5, or 12.99 will end up with the same value—12.

As you can see from the next-to-last `printf()` statement in Program 3.3, if you perform the same operation using floating-point values instead of integers, you obtain the expected result.

The decision of whether to use a `float` variable or an `int` variable should be made based on the variable’s intended use. If you don’t need any decimal places, use an integer variable. The resulting program is more efficient—that is, it executes more quickly on many computers. On the other hand, if you need the decimal place accuracy, the choice is clear. The only question you then must answer is whether to use a `float`, `double`, or `long double`. The answer to this question depends on the desired accuracy of the numbers you are dealing with, as well as their magnitude.

In the last `printf()` statement, the value of the variable `a` is negated by use of the unary minus operator. A *unary* operator is one that operates on a single value, as opposed to a binary

operator, which operates on two values. The minus sign actually has a dual role: As a binary operator, it is used for subtracting two values; as a unary operator, it is used to negate a value.

The unary minus operator has higher precedence than all other arithmetic operators, except for the unary plus operator (+), which has the same precedence. So the expression

```
c = -a * b;
```

results in the multiplication of $-a$ by b . Once again, in Appendix A you will find a table summarizing the various operators and their precedences.

The Modulus Operator

A surprisingly valuable operator, one you may not have experience with, is the modulus operator, which is symbolized by the percent sign (%). Try to determine how this operator works by analyzing Program 3.4.

Program 3.4 Illustrating the Modulus Operator

```
// The modulus operator

#include <stdio.h>

int main (void)
{
    int a = 25, b = 5, c = 10, d = 7;

    printf("a = %i, b = %i, c = %i, and d = %i\n", a, b, c, d);
    printf ("a %% b = %i\n", a % b);
    printf ("a %% c = %i\n", a % c);
    printf ("a %% d = %i\n", a % d);
    printf ("a / d * d + a %% d = %i\n",
           a / d * d + a % d);

    return 0;
}
```

Program 3.4 Output

```
a = 25, b = 5, c = 10, and d = 7
a % b = 0
a % c = 5
a % d = 4
a / d * d + a % d = 25
```

The first statement inside `main()` defines and initializes the variables `a`, `b`, `c`, and `d` in a single statement.

For a reminder, before a series of statements that use the modulus operator are printed, the first `printf()` statement prints the values of the four variables used in the program. It's not crucial, but it's a nice reminder to help someone follow along with your program. For the remaining `printf()` lines, as you know, `printf()` uses the character that immediately follows the percent sign to determine how to print the next argument. However, if it is another percent sign that follows, the `printf()` routine takes this as an indication that you really intend to display a percent sign and inserts one at the appropriate place in the program's output.

You are correct if you concluded that the function of the modulus operator `%` is to give the remainder of the first value divided by the second value. In the first example, the remainder after 25 is divided by 5 and is displayed as 0. If you divide 25 by 10, you get a remainder of 5, as verified by the second line of output. Dividing 25 by 7 gives a remainder of 4, as shown in the third output line.

The last line of output in Program 3.4 requires a bit of explanation. First, you will notice that the program statement has been written on two lines. This is perfectly valid in C. In fact, a program statement can be continued to the next line at any point at which a blank space could be used. (An exception to this occurs when dealing with character strings—a topic discussed in Chapter 9, “Character Strings.”) At times, it might not only be desirable, but perhaps even necessary, to continue a program statement onto the next line. The continuation of the `printf()` call in Program 3.4 is indented to visually show that it is a continuation of the preceding program statement.

Turn your attention to the expression evaluated in the final statement. You will recall that any operations between two integer values in C are performed with integer arithmetic. Therefore, any remainder resulting from the division of two integer values is simply discarded. Dividing 25 by 7, as indicated by the expression `a / d`, gives an intermediate result of 3. Multiplying this value by the value of `d`, which is 7, produces the intermediate result of 21. Finally, adding the remainder of dividing `a` by `d`, as indicated by the expression `a % d`, leads to the final result of 25. It is no coincidence that this value is the same as the value of the variable `a`. In general, the expression

```
a / b * b + a % b
```

will always equal the value of `a`, assuming of course that `a` and `b` are both integer values. In fact, the modulus operator `%` is defined to work only with integer values.

As far as precedence is concerned, the modulus operator has equal precedence to the multiplication and division operators. This implies, of course, that an expression such as

```
table + value % TABLE_SIZE
```

will be evaluated as

```
table + (value % TABLE_SIZE)
```

Integer and Floating-Point Conversions

To effectively develop C programs, you must understand the rules used for the implicit conversion of floating-point and integer values in C. Program 3.5 demonstrates some of the simple

conversions between numeric data types. You should note that some compilers might give warning messages to alert you of the fact that conversions are being performed.

Program 3.5 Converting Between Integers and Floats

```
// Basic conversions in C

#include <stdio.h>

int main (void)
{
    float f1 = 123.125, f2;
    int i1, i2 = -150;
    char c = 'a';

    i1 = f1;           // floating to integer conversion
    printf ("%f assigned to an int produces %i\n", f1, i1);

    f1 = i2;           // integer to floating conversion
    printf ("%i assigned to a float produces %f\n", i2, f1);

    f1 = i2 / 100;     // integer divided by integer
    printf ("%i divided by 100 produces %f\n", i2, f1);

    f2 = i2 / 100.0;   // integer divided by a float
    printf ("%i divided by 100.0 produces %f\n", i2, f2);

    f2 = (float) i2 / 100; // type cast operator
    printf "(float) %i divided by 100 produces %f\n", i2, f2);

    return 0;
}
```

Program 3.5 Output

```
123.125000 assigned to an int produces 123
-150 assigned to a float produces -150.000000
-150 divided by 100 produces -1.000000
-150 divided by 100.0 produces -1.500000
(float) -150 divided by 100 produces -1.500000
```

Whenever a floating-point value is assigned to an integer variable in C, the decimal portion of the number gets truncated. So, when the value of `f1` is assigned to `i1` in the previous program, the number 123.125 is *truncated*, which means that only its integer portion, or 123, is stored in `i1`. The first line of the program's output verifies that this is the case.

Assigning an integer variable to a floating variable does not cause any change in the value of the number; the value is simply converted by the system and stored in the floating variable. The second line of the program's output verifies that the value of `i2` (`-150`) was correctly converted and stored in the `float` variable `f1`.

The next two lines of the program's output illustrate two points that must be remembered when forming arithmetic expressions. The first has to do with integer arithmetic, which was previously discussed in this chapter. Whenever two operands in an expression are integers (and this applies to `short`, `unsigned`, `long`, and `long long` integers as well), the operation is carried out under the rules of integer arithmetic. Therefore, any decimal portion resulting from a division operation is discarded, even if the result is assigned to a floating variable (as you did in the program). Therefore, when the integer variable `i2` is divided by the integer constant `100`, the system performs the division as an integer division. The result of dividing `-150` by `100`, which is `-1`, is, therefore, the value that is stored in the `float` variable `f1`.

The next division performed in the previous listing involves an integer variable and a floating-point constant. Any operation between two values in C is performed as a floating-point operation if either value is a floating-point variable or constant. Therefore, when the value of `i2` is divided by `100.0`, the system treats the division as a floating-point division and produces the result of `-1.5`, which is assigned to the `float` variable `f1`.

The Type Cast Operator

The last division operation from Program 3.5 that reads

```
f2 = (float) i2 / 100; // type cast operator
```

introduces the type cast operator. The type cast operator has the effect of converting the value of the variable `i2` to type `float` for purposes of evaluation of the expression. In no way does this operator permanently affect the value of the variable `i2`; it is a unary operator that behaves like other unary operators. Because the expression `-a` has no permanent effect on the value of `a`, neither does the expression `(float) a`.

The type cast operator has a higher precedence than all the arithmetic operators except the unary minus and unary plus. Of course, if necessary, you can always use parentheses in an expression to force the terms to be evaluated in any desired order.

As another example of the use of the type cast operator, the expression

```
(int) 29.55 + (int) 21.99
```

is evaluated in C as

```
29 + 21
```

because the effect of casting a floating value to an integer is one of truncating the floating-point value. The expression

```
(float) 6 / (float) 4
```

produces a result of 1.5, as does the following expression:

```
(float) 6 / 4
```

Combining Operations with Assignment: The Assignment Operators

The C language permits you to join the arithmetic operators with the assignment operator using the following general format: $op=$

In this format, op is any of the arithmetic operators, including $+$, $-$, \times , $/$, and $\%$. In addition, op can be any of the bit operators for shifting and masking, which is discussed later.

Consider this statement:

```
count += 10;
```

The effect of the so-called “plus equals” operator $+=$ is to add the expression on the right side of the operator to the expression on the left side of the operator and to store the result back into the variable on the left-hand side of the operator. So, the previous statement is equivalent to this statement:

```
count = count + 10;
```

The expression

```
counter -= 5
```

uses the “minus equals” assignment operator to subtract 5 from the value of `counter` and is equivalent to this expression:

```
counter = counter - 5
```

A slightly more involved expression is

```
a /= b + c
```

which divides `a` by whatever appears to the right of the equal sign—or by the sum of `b` and `c`—and stores the result in `a`. The addition is performed first because the addition operator has higher precedence than the assignment operator. In fact, all operators but the comma operator have higher precedence than the assignment operators, which all have the same precedence.

In this case, this expression is identical to the following:

```
a = a / (b + c)
```

The motivation for using assignment operators is threefold. First, the program statement becomes easier to write because what appears on the left side of the operator does not have to be repeated on the right side. Second, the resulting expression is usually easier to read. Third, the use of these operators can result in programs that execute more quickly because the compiler can sometimes generate less code to evaluate an expression.

Types _Complex and _Imaginary

Before leaving this chapter it is worthy to note two other types in the language called `_Complex` and `_Imaginary` for working with complex and imaginary numbers.

Support for `_Complex` and `_Imaginary` types has been part of the ANSI C standard since C99, although C11 does make it optional. The best way to know if your compiler supports these types is to examine the summary of data types in Appendix A.

Exercises

- Type in and run the five programs presented in this chapter. Compare the output produced by each program with the output presented after each program in the text.
- Which of the following are invalid variable names? Why?

Int	char	6_05
Calloc	Xx	alpha_beta_routine
floating	_1312	z
ReInitialize	_	A\$

- Which of the following are invalid constants? Why?

123.456	0x10.5	0X0G1
0001	0xFFFF	123L
0xab05	0L	-597.25
123.5e2	.0001	+12
98.6F	98.7U	17777s
0996	-12E-12	07777
1234uL	1.2Fe-7	15,000
1.234L	197u	100U
0XABCDEFL	0abcu	+123

- Write a program that converts 27° from degrees Fahrenheit (F) to degrees Celsius (C) using the following formula:

$$C = (F - 32) / 1.8$$

- What output would you expect from the following program?

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

```
int main (void)
{
    char c, d;

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

```

    return 0;
}

```

6. Write a program to evaluate the polynomial shown here:

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

for $x = 2.55$.

7. Write a program that evaluates the following expression and displays the results (remember to use exponential format to display the result):

$$(3.31 \times 10^{-8} x - 2.01 \times 10^{-7}) / (7.16 \times 10^{-6} + 2.01 \times 10^{-8})$$

8. To round off an integer i to the next largest even multiple of another integer j , the following formula can be used:

$$\text{Next_multiple} = i + j - i \% j$$

For example, to round off 256 days to the next largest number of days evenly divisible by a week, values of $i = 256$ and $j = 7$ can be substituted into the preceding formula as follows:

$$\begin{aligned}
 \text{Next_multiple} &= 256 + 7 - 256 \% 7 \\
 &= 256 + 7 - 4 \\
 &= 259
 \end{aligned}$$

9. Write a program to find the next largest even multiple for the following values of i and j :

i	j
365	7
12,258	23
996	4

This page intentionally left blank

Index

Numbers

4 × 5 matrices, 113-114

Symbols

preprocessor directive, 470
operator, 310-311
#define preprocessor directive, 465-467
#define statement
 ## operator, 310-311
 preprocessor, 297-311
 program extendibility, 301-302
 program portability, 302-303
#elif preprocessor statement, 316-317
#else preprocessor statement, 315
#endif preprocessor statement, 316
#error preprocessor directive, 467
#if preprocessor directive, 467-468
#if preprocessor statement, 316-317
#ifdef preprocessor directive, 468
#ifdef preprocessor statement, 314-316
#ifndef preprocessor directive, 468
#ifndef preprocessor statement, 314-316
#include preprocessor directive, 468-469
#include preprocessor statement, 311-314
#line preprocessor directive, 469
#pragma preprocessor directive, 469
#undef preprocessor directive, 469-470
#undef preprocessor statement, 317

A

absolute values, integers, calculating, 66, 128-130
absoluteValue() function, 128-134
addEntry() function, 388
Albahari, Joseph and Ben, 508
algorithms, 5
aligning output, 50-51
alphabetic() function, 210
AND bitwise operator, 279-281
ANSI.org, 506
ar command (Unix), 343-344
arguments
 command-line, 380-384
 functions, 123-124
 checking, 133-137
 declaring type, 133-135
arithmetic, integer, 33-39
arithmetic expressions, 21, 30-39
arithmetic operators, 31, 446-447
arrays, 95-96
 base conversions, 109-111
 basic operations, 451
 character, 108-113, 194-196
 character strings, 218-226
 defining, 96-106
 finding minimum value in, 138-141
 functions, 137-151
 changing elements in, 141-143
 generating Fibonacci numbers, 103-104
 illustrating, 186-187
 initializing, 106-108
 multidimensional, 113-115, 436-437
 multidirectional, 146-151
 pointers, 258-270, 453-454
 prime numbers, generating, 104-106

setting, gdb (GNU Debugger), 407-408
single-dimensional, 435
sorting, 143
structures, 180-181, 185-186
using elements as counters, 100-103
variable-length, 115-117, 436
arraySum() function, 262-264
ascending order, arrays, sorting into, 144-145
assembly language program, 9
assignment operators, 39, 449
 changing array elements in functions, 141-143
atoi() function, 393
auto variable, 456
automatic local variables, 124-126, 155-158

B

base conversions, arrays, 109-111
BASIC programming language, 10
bits, 277-278
 fields, 291-295
 operators, 278-291
 Exclusive-OR, 282-283
 left shift, 285-288
 ones complement, 283-285
 right shift, 286-288
 rotating, 288-291
bitwise operators, 448
 AND, 279-281
 Inclusive-OR, 281-282
_Bool data type, 21, 24-28, 433
Boolean variables, 86-90
break statement, 62, 460
breakpoints, listing and deleting, gdb (GNU Debugger), 406

Budd, Timothy, 507
 building programs, 9
 bytes, 277-278

C

C: A Reference Manual, 505
C Programming Language, The, 505
C# 5.0 in a Nutshell: The Definitive Reference, 508
C++ Primer Plus, 507
C++ Programming Language, The, 507
 calculateTriangularNumber() function, 124-125
calculating
 absolute values, integers, 66, 128-130
 square roots, 130-133
calling functions, 121-122, 130-137, 459
 gdb (GNU Debugger), 407-408
cc command, 7, 12
 char *fgets (buffer, i, filePtr) function, 478
 char *getenv (s) function, 493
 char *gets (buffer) function, 480
 char *strcat (s1, s2) function, 474
 char * strchr (s, c) function, 474
 char *strcoll (s1, s2) function, 474
 char *strcpy (s1, s2) function, 474
 char *strerror (n) function, 474
 char *strncat (s1, s2, n) function, 474
 char *strncpy (s1, s2, n) function, 475
 char *strpbrk (s1, s2) function, 475
 char * strrchr (s, c) function, 475
 char *strstr (s1, s2) function, 475
 char * strtok (s1, s2) function, 475
 char data type, 21, 28, 433
 char variable, 24
character arrays, 108-113

character constants, 430-431
 character functions, 476-477
 character I/O (input/output), 346
 character strings, 16, 193-194
 arrays, 194-196, 218-226
 character operations, 226-229
 concatenating, 200-201
 concatenation, 431-432
 constants, 431-432
 pointers, 266-267
 strings, 217-218
 counting characters in, 198-199
 displaying, 199-202
 escape characters, 215-217
 initializing, 199-202
 inputting, 204-206
 null string, 211-214
 pointers, 264-266
 reading, 205
 single-character input, 206-211
 structures, 218-226
 testing for equality, 202-204
 variable-length, 197-214
characters, diagraphs, 427
classes, storage, 456
closing files, fclose() function, 365-367
code. See also programs
Code::Blocks IDE, 507
CodeWarrior, 507
comma operator, 378, 451
command-line arguments, 380-384
commands
 cc, 7, 12
 gcc, 495, 7, 506
 command-line options, 496-497
 compiling programs, 495-497
 general format, 495

gdb (GNU Debugger), 410-411
 Unix, 343-344
comments, 17-19, 429
compilers, 6
 gcc command, 495-497
 errors, 8-9
common mistakes, programming, 499-503
communication, between modules, 334-340
compilers, 6-7, 506-507
compiling
 debug code, 393-394
 multiple source files from command line, 332-334
 programs, 7-10, 11-12
 conditional compilation, 314-317
 gcc command, 495-497
_Complex data type, 40
compound literals, 454-455
 initializing structures, 178-179
compound relational tests, if statements, 72-74
compound statements, 460
computers
 instruction set, 5
 operating systems, 6-7
concat() function, 197-201
concatenating character strings, 197-201, 431-432
conditional compilation, preprocessor, 314-317
conditional operators, 90-92, 449-450
const keyword, 442
 pointers, 251-252
const variable, 111-113
constant expressions, 445-446
constants, 21-28
 character, 217-218, 430-431
 character string, 431-432
 pointers, 266-267
 defined names, #define statement, 297-311
 enumeration, 432
 floating-point, 430
 integer, 22, 429-430
 wide character, 431
continue statement, 62-63, 460
conversion characters
 printf() function, 348
 scanf() function, 353
conversion modifiers, scanf() function, 353
conversions
 data type, 319, 325-328
 floating-point, 36-38
 integer-point, 36-38
convertNumber() function, 152-154
copying files, 365-366
copyString() function, 264-266, 269-270
counters, arrays, using elements as, 100-103
countWords() function, 210-211
Cox, Brad, 417
cvs utility, 343
CygWin, 506

D

data, formatting
 printf() function, 346-353
 scanf() function, 353-358
data types, 21-28, 433-434
 _Bool, 24-26, 433
 char, 24, 433
 _Complex, 40

conversions, 319, 325-328, 455-456
derived, 435-440
double, 23, 433
double_Complex, 433
enumerated, 319, 441
 extending data types, 319-323
extending, 319
 data type conversions, 325-328
 enumerated data types, 322
 typedef statement, 323-325
float, 23, 433
float_Complex, 433
_Imaginary, 40
int, 433
long double, 433
long double_Complex, 433
long int, 433
short int, 433
specifiers, 26-28
type definitions, 319
unsigned char, 433
unsigned int, 433
void, 433

dates, storing, structures, 164-169

debug code, compiling in, 393-394

DEBUG macro, 395-396

debugging programs, 391

 gdb (GNU Debugger), 397-411
 preprocessor, 391-397

declarations, 432-435

 function calls, 459
 variables, for loops, 56

decrement operators, 267-270, 448-449

defined names, constants, #define statement, 297-311

defining

 arrays, 96-106

functions, 119-122, 458
header files, 471-473
pointer variables, 234-237

derived data types, 435-440

 multidimensional arrays, 436-437
 pointers, 440
 single-dimensional arrays, 435
 structures, 437-439
 unions, 439-440
 variable-length arrays, 436

diagraphs, 427

directives, preprocessor, 465-470

 #error, 467
 #if, 467-468
 #endif, 468
 #ifndef, 468
 #include, 468-469
 #line, 469
 #pragma, 469
 #undef, 469-470

displayConvertedNumber() function, 152

displaying

 character strings, 199-202
 variable values, 15-17

do statement, 60-63, 461

double acos (x) function, 486

double acosh (x) function, 487

double asin (x) function, 487

double asinh (x) function, 487

double atan (x) function, 487

double atan2 (y, x) function, 487

double atanh (x) function, 487

double atof (s) function, 483

double carg (z) function, 491

double ceil (x) function, 487

double cimag (z) function, 492

double complex cabs (z) function, 491
 double complex cacos (z) function, 491
 double complex cacosh (z) function, 491
 double complex casin (z) function, 491
 double complex casinh (z) function, 491
 double complex catan (z) function, 491
 double complex catanh (z) function, 492
 double complex ccos (z) function, 492
 double complex ccosh (z) function, 492
 double complex cexp (z) function, 492
 double complex clog (z) function, 492
 double complex conj (z) function, 492
 double complex cpow (y, z) function, 492
 double complex cproj (z) function, 492
 double complex creal (z) function, 492
 double complex csin (z) function, 492
 double complex csinh (z) function, 492
 double complex csqrt (z) function, 492
 double complex ctan (z) function, 492
 double complex ctanh (z) function, 492
 double copysign (x, y) function, 487
 double cos (r) function, 487
 double cosh (x) function, 487
 double data type, 21-24, 28, 433
 double erf (x) function, 487
 double erfc (x) function, 487
 double exp (x) function, 487
 double expm1 (x) function, 487
 double fabs (x) function, 488
 double fdim (x, y) function, 488
 double floor (x) function, 488
 double fma (x, y, z) function, 488
 double fmax (x, y) function, 488
 double fmin (x, y) function, 488
 double fmod (x, y) function, 488
 double frexp (x, exp) function, 488
 double ldexp (x, n) function, 488
 double lgamma (x) function, 488
 double log (x) function, 488
 double log1p (x) function, 489
 double log2 (x) function, 489
 double log10 (x) function, 489
 double logb (x) function, 489
 double modf (x, ipart) function, 489
 double nan (s) function, 490
 double nearbyint (x) function, 490
 double nextafter (x, y) function, 490
 double nexttoward (x, ly) function, 490
 double pow (x, y) function, 490
 double remainder (x, y) function, 490
 double remquo (x, y, quo) function, 490
 double rint (x) function, 490
 double round (x) function, 490
 double scalbln (x, n) function, 490
 double scalbn (x, n) function, 490
 double sin (r) function, 490
 double sinh (x) function, 490
 double sqrt (x) function, 490
 double strtod (s, end) function, 483-484
 double tan (r) function, 490
 double tanh (x) function, 490
 double tgamma (x) function, 490
 double trunc (x) function, 490
 double_Complex data type, 433
 dynamic memory allocation, 384-389
 free() function, 387-389
 dynamic memory allocation functions, 484-485

E

else-if statement, 76-83
 enumerated data types, 319, 441
 extending data types, 319-323

enumeration constants, 432
 equality, character strings, testing for, 202-204
`equalStrings()` function, 202-204, 219
 errors, code, 8-9
 escape characters, character strings, 215-217
`exchange()` function, 254-255
 Exclusive-OR operator, 282-283
 executing, programs, 9-10
`exit()` function, 369-370
 expressions, 442
 arithmetic, 21, 30-39
 arrays, basic operations, 451
 compound literals, 454-455
 constant, 445-446
 data types, conversion of, 455-456
 for loops, multiple, 55
 operators, 443-445
 arithmetic, 446-447
 assignment, 449
 bitwise, 448
 comma, 451
 conditional, 449-450
 decrement, 448-449
 increment, 448-449
 logical, 447
 relational, 447-448
 `sizeof`, 450-451
 type cast, 450
 pointers, 237-238
 basic operations, 452-454
 structures
 basic operations, 452
 using in, 166-169
 extendibility, programs, `#define` statement, 301-302

extending data types, 319
 data type conversions, 325-328
 enumerated data types, 319-323
 external variables, 334-337, 456
 versus static, 337-339

F

`factorial()` function, 158-159
 factorials, calculating recursively, 158-159
`fclose()` function, 365-367
`fgets()` function, 367-368
 Fibonacci numbers, generating, 103-104
 fields
 bits, 291-295
 for loops, omitting, 55
`FILE *fopen (fileName, accessMode) function, 478`
`FILE *freopen (fileName, accessMode, filePtr) function, 479`
`FILE *tmpfile (void) function, 482`
`FILE pointers, I/O (input/output), 368-369`
 files
 closing, `fclose()` function, 365-367
 copying, 365-366
 header, 471-473
 using effectively, 339-340
 Makefile, 341-342
 naming, 7
 opening, 362-364
 programs, dividing into multiple, 331-334
 reading and writing entire lines of data from and to, 367-368
 reading single character from, 364
 redirecting I/O (input/output) to, 358-362
 removing, 370-371

renaming, 370
 specifying lists and dependencies, 341-342
 system include, 313-314
findEntry() function, 257
flags, printf(), 347
float data type, 21-23, 28, 433
float strtod (s, end) function, 484
float_Complex data type, 433
floating-point constants, 430
floating-point conversions, 36-38
fopen() function, 362-364
for loops
 nested, 53-55
 variants, 55-56
for statement, 44-51, 461
formatted I/O (input/output), 346-358
 printf() function, 346-353
formatting data
 printf() function, 346-353
 scanf() function, 353-358
FORTRAN (FORmula TRANslation) language, 6
fprintf() function, 367, 392, 482
fputs() function, 367-368
free() function, 387-389
fscanf() function, 367, 482
functions, 119
 absoluteValue(), 128-134
 addEntry(), 388
 alphabetic(), 210
 arguments, 123-124
 checking, 133-137
 declaring type, 133-135
 arrays, 137-151
 changing elements in, 141-143
 multidirectional, 146-151
arraySum(), 262-264
atoi(), 393
automatic local variables, 124-126
calculateTriangularNumber(), 124-125
calling, 121-122, 130-137, 459
 gdb (GNU Debugger), 407-408
 character, 476-477
 concat(), 197-201
 convertNumber(), 152-154
 copyString(), 264-266, 269-270
 countWords(), 210-211
 defining, 119-122, 458
 displayConvertedNumber(), 152
 dynamic memory allocation, 484-485
 equalStrings(), 202-204, 219
 exchange(), 254-255
 exit(), 369-370
 external versus static, 337-339
 factorial(), 158-160
 fclose(), 365-367
 fgets(), 367-368
 findEntry(), 257
 fopen(), 362-364
 fprintf(), 367, 392, 482
 fputs(), 367-368
 free(), 387-389
 fscanf(), 367, 482
 gcd(), 125-128
 general utility, 492-494
 getc(), 364
 getchar(), 207-208, 346, 362
 getNumberAndBase(), 151
 global variables, 151-155
 I/O (input/output), 477-482
 isLeapYear(), 172

lookup(), 221-226
main(), 13, 119-120, 126
math, 485-492
memory, 475-476
 in-memory format conversion, 482-483
minimum(), 138-141
numberOfDays(), 169-172
perror(), 371
 pointers, 252-258, 272-273, 460
print_list(), 252
printf(), 13-17, 31-32, 119, 123, 137, 269, 345, 358
 formatting data, 346-353
printMessage(), 120-121
process(), 393-394
 prototype declaration, 124
putc(), 364
putchar(), 346
readLine(), 208-210
 recursive, 158-160
remove(), 370-371
 return types, declaring, 133-135
 returning results, 126-130
rotate(), 288-291, 314-315
scanf(), 59, 119, 177, 204-206, 358, 392
 formatting data, 353-358
shift, 286-288
sort(), 143
sprintf(), 482-483
squareRoot(), 131-134, 137, 155-156
sscanf(), 482-483
 static versus external, 337-339
 storage classes, 456
string, 474-475
stringLength(), 198, 217
 string-to-number, 483-484

structures, 169-177
test(), 254
 top-down programming, 137
variables
 automatic, 155-158
 static, 155-158
 writing, 120

G

gcc command, 7, 495, 506
 command-line options, 496-497
 compiling programs, 495-497
 general format, 495
gcd() function, 125-128
gdb (GNU Debugger)
 calling functions, 407-408
 commands, 410-411
 help, 408-409
 controlling program execution, 402-406
 debugging programs, 397-411
 working with variables, 400-401
 listing and deleting breakpoints, 406
 obtaining stack trace, 406-407
 setting arrays and structures, 407-408
 source file display, 401
general utility functions, 492-494
generating
 Fibonacci numbers, arrays, 103-104
 prime numbers, arrays, 104-106
getc() function, 364
getchar() function, 207-208, 346, 362
getNumberAndBase() function, 151
global variables, functions, 151-155
goto statement, 373-374, 461
grep command (Unix), 343-344

H

Harbison, Samuel P. III, 505

header files, 471-473

 using effectively, 339-340

help command (GDB), 408-409

I

identifiers, 428

 predefined, 470

IDEs (Integrated Development Environments), 10, 506-507

if (freopen ("inputData," "r", stdin) == NULL) function, 479

if statements, 65-83, 461-462

 compound relational tests, 72-74

 else-if, 76-83

 if-else, 69-72

 nested, 74-76

if-else statement, 69-72

illustrating

 arrays and structures, 186-187

 pointers, 235

 structures, 165

_Imaginary data type, 40

increment operators, 267-270, 448-449

indirection, pointers, 233-234

initializing

 arrays, 106-108

 character strings, 199-202

 structures, 178-179

in-memory format conversion functions, 482-483

input, program, 51-56

inputting character strings, 204-206

instances, OOP (object-oriented programming), 414-416

instruction set, computers, 5

int abs (n) function, 493

int atoi (s) function, 483

int atol (s) function, 483

int atoll (s) function, 483

int data type, 28, 433

int fclose (filePtr) function, 478

int perror (filePtr) function, 478

int fflush (filePtr) function, 478

int fgetc (filePtr) function, 478

int fgetpos (filePtr, fpos) function, 478

int fpclassify (x) function, 485

int fprintf (filePtr, format, arg1, arg2, argn) function, 479

int fputc (c, filePtr) function, 479

int fputs (buffer, filePtr) function, 479

int fscanf (filePtr, format, arg1, arg2, argn) function, 480

int fseek (filePtr, offset, mode) function, 480

int fsetpos (filePtr, fpos) function, 480

int getc (filePtr) function, 480

int getchar (void) function, 480

int hypot (x, y) function, 488

int ilogb (x) function, 488

int isfin (x) function, 486

int isgreater (x, y) function, 486

int isgreaterequal (x, y) function, 486

int isinf (x) function, 486

int islessequal (x, y) function, 486

int islessgreater (x, y) function, 486

int isnan (x) function, 486

int isnormal (x) function, 486

int isunordered (x, y) function, 486

int printf (format, arg1, arg2, argn) function, 481

int putc (c, filePtr) function, 481

int putchar(c) function, 481

int puts (buffer) function, 481
int remove (fileName) function, 481
int rename (fileName1, fileName2) function, 481
int scanf (format, arg1, arg2, argn) function, 482
int signbit (x) function, 486
int strcmp (s1, s2) function, 474
int strncmp (s1, s2, n) function, 475
int system (s) function, 494
int tolower(c) function, 477
int toupper(c) function, 477
int ungetc (c, filePtr) function, 482
integer arithmetic, 33-39
integer constants, 22, 429-430
integer-point conversions, 36-38
integers
 absolute values, calculating, 128-130
 if statements, 66
 base conversions, 109-111
Integrated Development Environments (IDEs), 10, 506-507
interpreters, 10
Introduction to Object-Oriented Programming, The, 507
I/O (input/output), 345
 character, 346
 exit() function, 369-370
 fclose() function, 365-367
 fgets() function, 367-368
 FILE pointers, 368-369
 fopen() function, 362-364
 formatted, 346-358
 printf() function, 346-353
 fprintf() function, 367
 fputs() function, 367-368
 fscanf() function, 367
 getc() function, 364

putc() function, 364
redirecting to files, 358-362
remove() function, 370-371
rename() function, 370
I/O (input/output) functions, 477-482
isLeapYear() function, 172

J-K

JavaScript,

Kernighan, Brian W., 505

keywords

const, 442
pointers, 251-252
restrict, 442

Kochan, Stephen, 508

L

languages, programming, 5-6

interpreters, 10
left shift operator, 285-288
Liberty, Jesse, 507
linked lists, pointers, 243-251
Linux, 7
literals, compound, 454-455

 initializing structures, 178-179

logical operators, 447
long double data type, 433
long double strtold (s, end) function, 484
long double_Complex data type, 433
long ftell (filePtr) function, 480
long int data type, 26-28, 433
long int labs (l) function, 493
long int lrint (x) function, 489
long int lround (x) function, 489
long int strtol (s, end, base) function, 484

long long int data type, 27-28
long long int llabs (ll) function, 493
long long int llrint (x) function, 489
long long int llround (x) function, 489
long long int strtoll (s, end, base)
 function, 484
lookup() function, 221-226
loops, for
 nested, 53-55
 variants, 55-56

M

Mac OS X, 7
macros, DEBUG, 395-396
main() routine, 119-120, 126, 137
make utility, 341-342
Makefile, 341-342
math functions, 485-492
memory addresses, pointers, 273-275
memory allocation, dynamic, 384-389
memory functions, 475-476
methods, OOP (object-oriented
 programming), 414-416
Microsoft Windows, 7
MinGW (Minimalist GNU for Windows), 506
minimum() function, 138-141
mistakes, programming, 499-503
modifiers, type, 442
modular operator, 35-36
modules, communication between,
 334-340
multibyte characters, character
 strings, 432
multidimensional arrays, 113-115,
 436-437
multidirectional arrays, 146-151
multiple expressions, for loops, 55

N

naming files, 7
nested if statements, 74-76
nested loops, for, 53-55
Newton-Raphson Iteration Technique,
 131-133
null statement, 374-375, 462
null string, 211-214
numberOfDays() function, 169-172
numbers
 Fibonacci, generating, 103-104
 prime, generating, 104-106
numread = fread (text, sizeof (char), 80,
 in_file) function, 479

O

objects, OOP (object-oriented
 programming), 413-414
omitted variable, 456
omitting fields, for loops, 55
ones complement operator, 283-285
OOP (object-oriented programming)
 defining
 C# class to work with fractions,
 424-426
 C++ class, 421-424
 objective-C class, 417-421
 instances, 414-416
 methods, 414-416
 objects, 413-414
 writing program to work with
 fractions, 416
OpenGroup.org, 506
opening files, fopen() function, 362-364
operating systems, 6-7
operations, pointers, 271-272

operators, 443-445

##, 310
 arithmetic, 31, 446-447
 assignment, 39, 141-143, 449
 bit, 278-291
 Exclusive-OR, 282-283
 left shift, 285-288
 ones complement, 283-285
 right shift, 286-288
 bitwise, 448
 AND, 279-281
 Inclusive-OR, 281-282
 comma, 378, 451
 conditional, 90-92, 449-450
 decrement, 267-270, 448-449
 increment, 267-270, 448-449
 logical, 447
 modular, 35-36
 relational, 46-50, 447-448
 sizeof, 450-451
 type cast, 38-39, 450
 unary minus, 33-39
optimization, programs, 262
output, aligning, 50-51

P**perror() function, 371****Petzold, Charles, 507****Plauger, P. J., 505****pointers, 233, 440**

arrays, 258-270, 453-454
 basic operations, 452-454
 character strings, 264-266
 constant, 266-267
 expressions, 237-238
 functions, 252-258, 272-273, 460
 illustrating, 235

indirection, 233-234
 keyword const, 251-252
 linked lists, 243-251
 memory addresses, 273-275
 operations, 271-272
 structures, 239-251, 453-454
 variables, defining, 234-237
portability, programs, #define statement, 302-303

Prata, Stephen, 507**precision modifiers, printf() function, 347****predefined identifiers, 470****preprocessor, 297, 464**

conditional compilation, 314-317
 debugging programs, 391-397
 directives, 465-470
 #define, 465-467
 #error, 467
 #if, 467-468
 #endif, 468
 #ifndef, 468
 #include, 468-469
 #line, 469
 #pragma, 469
 #undef, 469-470

statements

#define statement, 297-311
 #elif statement, 316-317
 #else statement, 315
 #endif, 316
 #if statement, 316-317
 #ifdef, 314-316
 #include statement, 311-314
 #undef statement, 317-
 trigraph sequences, 464

prime numbers, generating, arrays, 104-106

- print_list() function, 252**
 - printf() function, 13-17, 31-32, 119, 123, 137, 269, 345, 358**
 - conversion characters, 348
 - flags, 347
 - formatting data, 346-353
 - illustrating formats, 349-351
 - precision and width modifiers, 347
 - type modifiers, 347
 - printMessage() function, 120-121**
 - process() function, 393-394**
 - program looping, 43**
 - do statement, 60-63
 - program input, 51-56
 - for statement, 44-51
 - triangular numbers, 43-44
 - while statement, 56-60
 - programming**
 - common mistakes, 499-503
 - OOP (object-oriented programming), 413
 - defining C# class to work with fractions, 424-426
 - defining C++ class, 421-424
 - defining objective-C class, 417-421
 - instances, 414-416
 - methods, 414-416
 - writing program to work with fractions, 416
 - top-down, 137
 - Programming C# 3.0, 507**
 - Programming in Objective-C, 508**
 - Programming in the Key of C#, 507**
 - programming languages, 5-6**
 - interpreters, 10
 - OOP (object-oriented programming), objects, 413-414
 - programs, 5**
 - building, 9
 - comments, 17-19, 429
 - compiling, 7-12
 - conditional compilation, 314-317
 - gcc command, 495-497
 - debugging, 391
 - gdb (GNU Debugger), 397-411
 - preprocessor, 391-397
 - dividing into multiple files, 331-334
 - executing, 9-10
 - extendibility, #define statement, 301-302
 - first, assessing, 13-15
 - forcing termination, exit() function, 369-370
 - optimization, 262
 - portability, #define statement, 302-303
 - running, 11, 12-13
 - prototype declaration, functions, 124**
 - putc() function, 364**
 - putchar() function, 346**
-
- Q-R**
-
- qualifiers, type, 379**
 - ranges, integers, 23**
 - reading character strings, 205**
 - readLine() function, 208-210**
 - recursive functions, 158-160**
 - register qualifier, 379**
 - register variable, 456**
 - relational operators, 46-50, 447-448**
 - remove() function, 370-371**
 - rename() function, 370-371**
 - renaming files, rename() function, 370-371**
 - restrict keyword, 442**
 - restrict qualifier, 379-380**
 - results, functions, returning, 126-130**

return statement, 462-463
 return types, functions, declaring, 133-135
 returning function results, 126-130
 right shift operator, 286-288
 Ritchie, Dennis M., 505
 rotate() function, 288-291, 314-315
 rotating bits, 288-291
 running programs, 11-13

S

scanf() function, 59, 119, 177, 204-206, 358, 392
 conversion characters, 353
 conversion modifiers, 353
 formatting data, 353-358
 sed command (Unix), 343-344
 shift functions, 286-288
 short int data type, 27-28, 433
 single-character input, character strings, 206-211
 single-dimensional arrays, 435
 size_t fread (buffer, size, n, filePtr) function, 479
 size_t fwrite (buffer, size, n, filePtr) function, 480
 size_t strcspn (s1, s2) function, 474
 size_t strlen (s) function, 474
 size_t strspn (s1, s2) function, 475
 size_t strxfrm (s1, s2, n) function, 475
 sizeof operator, 450-451
 sort() function, 143
 sorting arrays, 143
 source file display, gdb (GNU Debugger), 401
 source files, compiling multiple from command line, 332-334
 specifiers, data types, 26-28
 sprintf() function, 482-483

square roots, calculating, 130-133
 squareRoot() function, 131-134, 137, 155-156
 sscanf() function, 482-483
 stack traces, obtaining, gdb (GNU Debugger), 406-407
Standard C Library, The, 505
 standard header files, 471-473
 statements, 460
 break, 62, 460
 compound, 460
 continue, 62-63, 460
 debug, adding with preprocessor, 391-392
 #define, preprocessor, 297-311
 do, 60-63, 461
 for, 44-51, 461
 goto, 373-374, 461
 if, 65-83, 461-462
 compound relational tests, 72-74
 else-if, 76-83
 if-else, 69-72
 nested, 74-76
 #include, preprocessor, 311-314
 null, 374-375, 462
 preprocessor
 #elif, 316-317
 #else, 315
 #endif, 316
 #if, 316-317
 #ifndef, 314-316
 #endif, 314-316
 #undef, 317
 return, 462-463
 switch, 83-86, 463-464
 typedef, 441
 extending data types, 323-325
 while, 56-60, 464

- static variables, 456**
 - versus external, 337-339
 - stderr FILE pointer, 368-369**
 - stdin FILE pointer, 368-369**
 - stdout FILE pointer, 368-369**
 - Steele, Guy L. Jr., 505**
 - storage classes, 456**
 - functions, 456
 - variables, 456
 - storage sizes, integers, 23**
 - storing dates, structures, 164-169**
 - string functions, 474-475**
 - StringLength() function, 198, 217**
 - strings, character, 193-194**
 - arrays, 218-226
 - arrays of characters, 194-196
 - character operations, 226-229
 - concatenating, 200-201
 - constant strings, 217-218
 - counting characters in, 198-199
 - displaying, 199-202
 - escape characters, 215-217
 - initializing, 199-202
 - inputting, 204-206
 - null string, 211-214
 - pointers, 264-267
 - reading, 205
 - single-character input, 206-211
 - structures, 218-226
 - testing for equality, 202-204
 - variable-length, 197-214
 - string-to-number routines, 483-484**
 - Stroustrup, Bjarne, 421, 507**
 - structures, 163-164, 437-439**
 - arrays, 180-181, 185-186
 - basic operations, 452
 - character strings, 218-226
 - containing structures, 183-185
 - functions, 169-177
 - illustrating, 165, 186-187
 - initializing, 178-179
 - pointers, 239-251
 - setting, gdb (GNU Debugger), 407-408
 - storing dates, 164-169
 - storing time, 175-177
 - using in expressions, 166-169
 - variants, 189
 - switch statement, 83-86, 463-464**
 - system ("mkdir /usr/tmp/data") function, 494**
 - system include files, 313-314**
-
- T**
- tables, prime numbers, generating, 86-87**
 - test() function, 254**
 - testing character strings for equality, 202-204**
 - tests, compound relational, if statements, 72-74**
 - text editors, 7**
 - top-down programming, 137**
 - triangular numbers, 43-44**
 - trigraph sequences, preprocessor, 464**
 - type cast operator, 38-39, 450**
 - type definitions, 319**
 - type modifiers, 442**
 - printf() function, 347
 - type qualifiers, 379-380**
 - typedef statement, 441**
 - extending data types, 323-325
-
- U**
- unary minus operators, 33-39**
 - unions, 375-378, 439-440**

universal character names, 428

Unix, 7

Unix utilities, 343-344

unsigned char data type, 433

unsigned int constant, 27-28

unsigned int data type, 433

unsigned long int strtoul (s, end, base) function, 484

unsigned long long int strtoull (s, end, base) function, 484

unsigned specifier, 278

utilities

cvs, 343

make, 341-342

Unix, 343-344

V

values, variables, displaying, 15-17

variable-length arrays, 115-117, 436

variable-length character strings, 197-214

variables, 21, 29-30

auto, 456

automatic local, 124-126, 155-158

_Bool, 24-26

Boolean, 86-90

char, 24

declaring, for loops, 56

displaying values of, 15-17

extern, 456

external, 334-337

versus static, 337-339

global, functions, 151-155

omitted, 456

pointers, defining, 234-237

register, 456

static, 155-158, 456

versus external, 337-339

storage classes, 456

volatile, 442

working with, gdb (GNU Debugger), 400-401

variants

for loops, 55-56

structures, 189

vim text editor, 7

Visual Studio, 506

void *calloc (n, size) function, 484

void *malloc (size) function, 485

void *memchr (m1, c, n) function, 476

void *memcmp (m1, m2, n) function, 476

void *memcpy (m1, m2, n) function, 476

void *memmove (m1, m2, n) function, 476

void *memset (m1, c, n) function, 476

void *realloc (pointer, size) function, 485

void clearerr (filePtr) function, 477

void data type, 433

void exit (n) function, 493

void free (pointer) function, 485

void perror (message) function, 481

void qsort (arr, n, size, comp_fn) function, 493-494

void rewind (filePtr) function, 481

void srand (seed) function, 494

volatile qualifier, 379

volatile variable, 442

W-Z

while statement, 56-60, 464

wide character constants, 431

wide character string constants, 432

width modifiers, printf() function, 347

writing functions, 120