C FOR PROGRAMMERS
WITH AN INTRODUCTION TO C11
DEITEL® DEVELOPER SERIES
Deitel® Series Page

Deitel® Developer Series
C for Programmers
C++ for Programmers, 2/E
Android™ for Programmers: An App-Driven Approach
C# 2010 for Programmers, 3/E
Dive Into® iOS 6: An App-Driven Approach
Java™ for Programmers, 2/E
JavaScript for Programmers

LiveLessons Video Learning Products
www.deitel.com/books/LiveLessons/
Android® App Development Fundamentals
C++ Fundamentals
C# Fundamentals
iOS® 6 App Development Fundamentals
Java™ Fundamentals
JavaScript Fundamentals
Visual Basic® Fundamentals

Simply Series
Simply C++: An App-Driven Tutorial Approach
Simply Java™ Programming: An App-Driven Tutorial Approach

CourseSmart Web Books
www.deitel.com/books/CourseSmart/
C++ How to Program, 7/E, 8/E & 9/E
Simply C++: An App-Driven Tutorial Approach
Java™ How to Program, 7/E, 8/E & 9/E
Visual Basic® 2012 How to Program
Visual Basic® 2010 How to Program
Visual C#® 2012 How to Program, 5/E
Visual C#® 2010 How to Program, 4/E

How To Program Series
Android™ How to Program
C How to Program, 7/E
C++ How to Program, 9/E
C++ How to Program, Late Objects Version, 7/E
Java™ How to Program, 9/E
Java™ How to Program, Late Objects Version, 8/E
Internet & World Wide Web How to Program, 5/E
Visual Basic® 2012 How to Program
Visual C#® 2012 How to Program, 5/E
Visual C++® 2008 How to Program, 2/E
Small Java™ How to Program, 6/E
Small C++ How to Program, 5/E

To receive updates on Deitel publications, Resource Centers, training courses, partner offers and more, please register for the free Deitel® Buzz Online e-mail newsletter at:
www.deitel.com/newsletter/subscribe.html
and join the Deitel communities on Twitter® @deitel
Facebook® facebook.com/DeitelFan
Google+ gplus.to/deitel
and LinkedIn bit.ly/DeitelLinkedIn
To communicate with the authors, send e-mail to: deitel@deitel.com
For information on government and corporate Dive-Into® Series on-site seminars offered by Deitel & Associates, Inc. worldwide, visit:
www.deitel.com/training/
or write to
delel@deitel.com
For continuing updates on Pearson/Deitel publications visit:
www.deitel.com
www.pearsonhighered.com/deitel/
Visit the Deitel Resource Centers that will help you master programming languages, software development, Android™ and iPhone®/iPad® app development, and Internet- and web-related topics:
www.deitel.com/ResourceCenters.html
In memory of Dennis Ritchie,
creator of the C programming language
and co-creator of the UNIX operating system.

Paul and Harvey Deitel
Trademarks

Deitel, the double-thumbs-up bug and Dive Into are registered trademarks of Deitel and Associates, Inc.

Microsoft and/or its respective suppliers make no representations about the suitability of the information contained in the documents and related graphics published as part of the services for any purpose. All such documents and related graphics are provided "as is" without warranty of any kind. Microsoft and/or its respective suppliers hereby disclaim all warranties and conditions with regard to this information, including all warranties and conditions of merchantability, whether express, implied or statutory, fitness for a particular purpose, title and non-infringement. In no event shall Microsoft and/or its respective suppliers be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of information available from the services.

The documents and related graphics contained herein could include technical inaccuracies or typographical errors. Changes are periodically added to the information herein. Microsoft and/or its respective suppliers may make improvements and/or changes in the product(s) and/or the program(s) described herein at any time. Partial screen shots may be viewed in full within the software version specified.

Throughout this book, trademarks are used. Rather than put a trademark symbol in every occurrence of a trademarked name, we state that we are using the names in an editorial fashion only and to the benefit of the trademark owner, with no intention of infringement of the trademark.
## Contents

### 3 Control Statements: Part I

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Introduction</td>
<td>38</td>
</tr>
<tr>
<td>3.2</td>
<td>Control Structures</td>
<td>38</td>
</tr>
<tr>
<td>3.3</td>
<td>The <code>if</code> Selection Statement</td>
<td>40</td>
</tr>
<tr>
<td>3.4</td>
<td>The <code>if...else</code> Selection Statement</td>
<td>40</td>
</tr>
<tr>
<td>3.5</td>
<td>The <code>while</code> Repetition Statement</td>
<td>43</td>
</tr>
<tr>
<td>3.6</td>
<td>Class Average with Counter-Controlled Repetition</td>
<td>44</td>
</tr>
<tr>
<td>3.7</td>
<td>Class Average with Sentinel-Controlled Repetition</td>
<td>46</td>
</tr>
<tr>
<td>3.8</td>
<td>Nested Control Statements</td>
<td>49</td>
</tr>
<tr>
<td>3.9</td>
<td>Assignment Operators</td>
<td>51</td>
</tr>
<tr>
<td>3.10</td>
<td>Increment and Decrement Operators</td>
<td>52</td>
</tr>
<tr>
<td>3.11</td>
<td>Secure C Programming</td>
<td>55</td>
</tr>
</tbody>
</table>

### 4 Control Statements: Part II

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>58</td>
</tr>
<tr>
<td>4.2</td>
<td>Repetition Essentials</td>
<td>58</td>
</tr>
<tr>
<td>4.3</td>
<td>Counter-Controlled Repetition</td>
<td>59</td>
</tr>
<tr>
<td>4.4</td>
<td><code>for</code> Repetition Statement</td>
<td>60</td>
</tr>
<tr>
<td>4.5</td>
<td><code>for</code> Statement: Notes and Observations</td>
<td>63</td>
</tr>
<tr>
<td>4.6</td>
<td>Examples Using the <code>for</code> Statement</td>
<td>64</td>
</tr>
<tr>
<td>4.7</td>
<td><code>switch</code> Multiple-Selection Statement</td>
<td>67</td>
</tr>
<tr>
<td>4.8</td>
<td><code>do...while</code> Repetition Statement</td>
<td>73</td>
</tr>
<tr>
<td>4.9</td>
<td><code>break</code> and <code>continue</code> Statements</td>
<td>75</td>
</tr>
<tr>
<td>4.10</td>
<td>Logical Operators</td>
<td>77</td>
</tr>
<tr>
<td>4.11</td>
<td>Confusing Equality (==) and Assignment (=) Operators</td>
<td>80</td>
</tr>
<tr>
<td>4.12</td>
<td>Secure C Programming</td>
<td>81</td>
</tr>
</tbody>
</table>

### 5 Functions

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>84</td>
</tr>
<tr>
<td>5.2</td>
<td>Program Modules in C</td>
<td>84</td>
</tr>
<tr>
<td>5.3</td>
<td>Math Library Functions</td>
<td>85</td>
</tr>
<tr>
<td>5.4</td>
<td>Functions</td>
<td>86</td>
</tr>
<tr>
<td>5.5</td>
<td>Function Definitions</td>
<td>87</td>
</tr>
<tr>
<td>5.6</td>
<td>Function Prototypes: A Deeper Look</td>
<td>91</td>
</tr>
<tr>
<td>5.7</td>
<td>Function Call Stack and Stack Frames</td>
<td>94</td>
</tr>
<tr>
<td>5.8</td>
<td>Headers</td>
<td>97</td>
</tr>
<tr>
<td>5.9</td>
<td>Passing Arguments By Value and By Reference</td>
<td>98</td>
</tr>
<tr>
<td>5.10</td>
<td>Random Number Generation</td>
<td>99</td>
</tr>
<tr>
<td>5.11</td>
<td>Example: A Game of Chance</td>
<td>104</td>
</tr>
<tr>
<td>5.12</td>
<td>Storage Classes</td>
<td>107</td>
</tr>
<tr>
<td>5.13</td>
<td>Scope Rules</td>
<td>109</td>
</tr>
<tr>
<td>5.14</td>
<td>Recursion</td>
<td>112</td>
</tr>
<tr>
<td>5.15</td>
<td>Example Using Recursion: Fibonacci Series</td>
<td>116</td>
</tr>
</tbody>
</table>
6 Arrays
6.1 Introduction
6.2 Arrays
6.3 Defining Arrays
6.4 Array Examples
6.5 Passing Arrays to Functions
6.6 Sorting Arrays
6.7 Case Study: Computing Mean, Median and Mode Using Arrays
6.8 Searching Arrays
6.9 Multidimensional Arrays
6.10 Variable-Length Arrays
6.11 Secure C Programming

7 Pointers
7.1 Introduction
7.2 Pointer Variable Definitions and Initialization
7.3 Pointer Operators
7.4 Passing Arguments to Functions by Reference
7.5 Using the const Qualifier with Pointers
   7.5.1 Converting a String to Uppercase Using a Non-Constant Pointer to Non-Constant Data
   7.5.2 Printing a String One Character at a Time Using a Non-Constant Pointer to Constant Data
   7.5.3 Attempting to Modify a Constant Pointer to Non-Constant Data
   7.5.4 Attempting to Modify a Constant Pointer to Constant Data
7.6 Bubble Sort Using Pass-by-Reference
7.7 sizeof Operator
7.8 Pointer Expressions and Pointer Arithmetic
7.9 Relationship between Pointers and Arrays
7.10 Arrays of Pointers
7.11 Case Study: Card Shuffling and Dealing Simulation
7.12 Pointers to Functions
7.13 Secure C Programming

8 Characters and Strings
8.1 Introduction
8.2 Fundamentals of Strings and Characters
8.3 Character-Handling Library
   8.3.1 Functions isdigit, isalpha, isalnum and isxdigit
   8.3.2 Functions islower, isupper, tolower and toupper
   8.3.3 Functions isspace, iscntrl, ispunct, isprint and isgraph
8.4 String-Conversion Functions
   8.4.1 Function \texttt{strtod}  
   8.4.2 Function \texttt{strtol}  
   8.4.3 Function \texttt{strtoul}  
8.5 Standard Input/Output Library Functions
   8.5.1 Functions \texttt{fgets} and \texttt{putchar}  
   8.5.2 Function \texttt{getchar}  
   8.5.3 Function \texttt{sprintf}  
   8.5.4 Function \texttt{scanf}  
8.6 String-Manipulation Functions of the String-Handling Library
   8.6.1 Functions \texttt{strcpy} and \texttt{strncpy}  
   8.6.2 Functions \texttt{strcat} and \texttt{strncat}  
8.7 Comparison Functions of the String-Handling Library  
8.8 Search Functions of the String-Handling Library
   8.8.1 Function \texttt{strchr}  
   8.8.2 Function \texttt{strcspn}  
   8.8.3 Function \texttt{strpbrk}  
   8.8.4 Function \texttt{strrchr}  
   8.8.5 Function \texttt{strspn}  
   8.8.6 Function \texttt{strstr}  
   8.8.7 Function \texttt{strtok}  
8.9 Memory Functions of the String-Handling Library
   8.9.1 Function \texttt{memcpy}  
   8.9.2 Function \texttt{memmove}  
   8.9.3 Function \texttt{memcmp}  
   8.9.4 Function \texttt{memchr}  
   8.9.5 Function \texttt{memset}  
8.10 Other Functions of the String-Handling Library
   8.10.1 Function \texttt{strerror}  
   8.10.2 Function \texttt{strlen}  
8.11 Secure C Programming  

9 Formatted Input/Output

9.1 Introduction  
9.2 Streams  
9.3 Formatting Output with \texttt{printf}  
9.4 Printing Integers  
9.5 Printing Floating-Point Numbers  
9.6 Printing Strings and Characters  
9.7 Other Conversion Specifiers  
9.8 Printing with Field Widths and Precision  
9.9 Using Flags in the \texttt{printf} Format Control String  
9.10 Printing Literals and Escape Sequences  
9.11 Reading Formatted Input with \texttt{scanf}  
9.12 Secure C Programming  

10 Structures, Unions, Bit Manipulation and Enumerations

10.1 Introduction

10.2 Structure Definitions
   10.2.1 Self-Referential Structures
   10.2.2 Defining Variables of Structure Types
   10.2.3 Structure Tag Names
   10.2.4 Operations That Can Be Performed on Structures

10.3 Initializing Structures

10.4 Accessing Structure Members

10.5 Using Structures with Functions

10.6 `typedef`

10.7 Example: High-Performance Card Shuffling and Dealing Simulation

10.8 Unions
   10.8.1 Union Declarations
   10.8.2 Operations That Can Be Performed on Unions
   10.8.3 Initializing Unions in Declarations
   10.8.4 Demonstrating Unions

10.9 Bitwise Operators
   10.9.1 Displaying an Unsigned Integer in Bits
   10.9.2 Making Function `displayBits` More Scalable and Portable
   10.9.3 Using the Bitwise AND, Inclusive OR, Exclusive OR and Complement Operators
   10.9.4 Using the Bitwise Left- and Right-Shift Operators
   10.9.5 Bitwise Assignment Operators

10.10 Bit Fields

10.11 Enumeration Constants

10.12 Secure C Programming

11 File Processing

11.1 Introduction

11.2 Files and Streams

11.3 Creating a Sequential-Access File

11.4 Reading Data from a Sequential-Access File

11.5 Random-Access Files

11.6 Creating a Random-Access File

11.7 Writing Data Randomly to a Random-Access File

11.8 Reading Data from a Random-Access File

11.9 Case Study: Transaction-Processing Program

11.10 Secure C Programming

12 Data Structures

12.1 Introduction

12.2 Self-Referential Structures
12.3 Dynamic Memory Allocation 313
12.4 Linked Lists 314
  12.4.1 Function insert 320
  12.4.2 Function delete 321
  12.4.3 Function printList 322
12.5 Stacks 323
  12.5.1 Function push 327
  12.5.2 Function pop 328
  12.5.3 Applications of Stacks 328
12.6 Queues 329
  12.6.1 Function enqueue 333
  12.6.2 Function dequeue 334
12.7 Trees 335
  12.7.1 Function insertNode 338
  12.7.2 Traversals: Functions inOrder, preOrder and postOrder 339
  12.7.3 Duplicate Elimination 340
  12.7.4 Binary Tree Search 340
12.8 Secure C Programming 340

13 Preprocessor 342
13.1 Introduction 343
13.2 #include Preprocessor Directive 343
13.3 #define Preprocessor Directive: Symbolic Constants 344
13.4 #define Preprocessor Directive: Macros 344
13.5 Conditional Compilation 346
13.6 #error and #pragma Preprocessor Directives 347
13.7 # and ## Operators 348
13.8 Line Numbers 348
13.9 Predefined Symbolic Constants 348
13.10 Assertions 349
13.11 Secure C Programming 349

14 Other Topics 351
14.1 Introduction 352
14.2 Redirecting I/O 352
14.3 Variable-Length Argument Lists 353
14.4 Using Command-Line Arguments 355
14.5 Notes on Compiling Multiple-Source-File Programs 356
14.6 Program Termination with exit and atexit 358
14.7 Suffixes for Integer and Floating-Point Literals 359
14.8 Signal Handling 360
14.9 Dynamic Memory Allocation: Functions calloc and realloc 362
14.10 Unconditional Branching with goto 363
### A Operator Precedence Chart
- Page 365

### B ASCII Character Set
- Page 367

### C Number Systems
- C.1 Introduction 369
- C.2 Abbreviating Binary Numbers as Octal and Hexadecimal Numbers 372
- C.3 Converting Octal and Hexadecimal Numbers to Binary Numbers 373
- C.4 Converting from Binary, Octal or Hexadecimal to Decimal 373
- C.5 Converting from Decimal to Binary, Octal or Hexadecimal 374
- C.6 Negative Binary Numbers: Two’s Complement Notation 376

### D Sorting: A Deeper Look
- D.1 Introduction 379
- D.2 Big O Notation 379
- D.3 Selection Sort 380
- D.4 Insertion Sort 384
- D.5 Merge Sort 387

### E Additional Features of the C Standard
- E.1 Introduction 395
- E.2 Support for C99 396
- E.3 C99 Headers 396
- E.4 Mixing Declarations and Executable Code 397
- E.5 Declaring a Variable in a for Statement Header 397
- E.6 Designated Initializers and Compound Literals 398
- E.7 Type bool 401
- E.8 Implicit int in Function Declarations 402
- E.9 Complex Numbers 403
- E.10 Variable-Length Arrays 404
- E.11 Additions to the Preprocessor 407
- E.12 Other C99 Features 408
  - E.12.1 Compiler Minimum Resource Limits 408
  - E.12.2 The restrict Keyword 409
  - E.12.3 Reliable Integer Division 409
  - E.12.4 Flexible Array Members 409
  - E.12.5 Relaxed Constraints on Aggregate Initialization 410
  - E.12.6 Type Generic Math 410
  - E.12.7 Inline Functions 410
  - E.12.8 return Without Expression 411
  - E.12.9 __func__ Predefined Identifier 411
  - E.12.10 va_copy Macro 411
Welcome to the C programming language. This book presents leading-edge computing technologies for software development professionals.

At the heart of the book is the Deitel signature “live-code approach.” We present concepts in the context of complete working programs, rather than in code snippets. Each code example is followed by one or more sample executions. Read the online Before You Begin section (www.deitel.com/books/cfp/cfp_BYB.pdf) to learn how to set up your computer to run the 130 code examples and your own C programs. All the source code is available at www.deitel.com/books/cfp and www.pearsonhighered.com/deitel. Use the source code we provide to run every program as you study it.

This book will give you an informative, challenging and entertaining introduction to C. If you have questions, send an e-mail to deitel@deitel.com—we’ll respond promptly. For book updates, visit www.deitel.com/books/cfp, join our communities on Facebook (www.deitel.com/deitelfan), Twitter (@deitel), Google+ (gplus.to/deitel) and LinkedIn (bit.ly/deitellinkedIn), and subscribe to the Deitel® Buzz Online newsletter (www.deitel.com/newsletter/subscribe.html).

Features

Here are some key features of C for Programmers with an Introduction to C11:

• Coverage of the New C standard. The book is written to the new C standard, often referred to as C11 or simply “the C standard” since its approval in 2011. Support for the new standard varies by compiler. Most of our readers use either the GNU gcc compiler—which supports many of the key features in the new standard—or the Microsoft Visual C++ compiler. Microsoft supports only a limited subset of the features that were added to C in C99 and C11—primarily the features that are also required by the C++ standard. To accommodate all of our readers, we placed the discussion of the new standard’s features in optional, easy-to-use-or-omit sections and in Appendix E, Additional Features of the C Standard. We’ve also replaced various deprecated capabilities with newer preferred versions as a result of the new C standard.

• Chapter 1. We’ve included test-drives that show how to run a command-line C program on Microsoft Windows, Linux and Mac OS X.

• Secure C Programming Sections. We’ve added notes about secure C programming to many of the C programming chapters. We’ve also posted a Secure C Programming Resource Center at www.deitel.com/SecureC/. For more details, see the section A Note About Secure C Programming in this Preface.
• **Focus on Performance Issues.** C is often favored by designers of performance-intensive applications such as operating systems, real-time systems, embedded systems and communications systems, so we focus intensively on performance issues.

• **All Code Tested on Windows and Linux.** We’ve tested every example program using Visual C++ and GNU gcc in Windows and Linux, respectively.

• **Sorting: A Deeper Look.** Sorting is an interesting problem because different sorting techniques achieve the same final result but they can vary hugely in their consumption of memory, CPU time and other system resources—algorithm performance is crucial. We begin our presentation of sorting in Chapter 6 and, in Appendix D, we present a deeper look. We consider several algorithms and compare them with regard to their memory consumption and processor demands. For this purpose, we introduce Big O notation, which indicates how hard an algorithm may have to work to solve a problem. Appendix D discusses the selection sort, insertion sort and recursive merge sort.

• **Debugger Appendices.** We include Visual Studio and GNU gdb debugging appendices.

• **Order of Evaluation.** We discuss subtle order of evaluation issues to help you avoid errors.

• **C++-Style // Comments.** We use the newer, more concise C++-style // comments in preference to C’s older style /* . . . */ comments.

• **C Standard Library.** Section 1.3 references en.cppreference.com/w/c where you can find thorough searchable documentation for the C Standard Library functions.

**A Note About Secure C Programming**

Experience has shown that it’s difficult to build industrial-strength systems that stand up to attacks from viruses, worms, etc. Today, via the Internet, such attacks can be instantaneous and global in scope. Software vulnerabilities often come from easy-to-avoid programming issues. Building security into software from the start of the development cycle can greatly reduce costs and vulnerabilities.

The CERT® Coordination Center (www.cert.org) was created to analyze and respond promptly to attacks. CERT—the Computer Emergency Response Team—publishes and promotes secure coding standards to help C programmers and others implement industrial-strength systems that avoid the programming practices that open systems to attack. The CERT standards evolve as new security issues arise.

Our code conforms to various CERT recommendations as appropriate for a book at this level. If you’ll be building C systems in industry, consider reading two books by Robert Seacord—*The CERT C Secure Coding Standard* (Addison-Wesley Professional, 2009) and *Secure Coding in C and C++* (Addison-Wesley Professional, 2013). The CERT guidelines are available free online at www.securecoding.cert.org. Seacord, a technical reviewer for this book, also provided specific recommendations on each of our new Secure C Programming sections. Mr. Seacord is the Secure Coding Manager at CERT at Carnegie Mellon University’s Software Engineering Institute (SEI) and an adjunct professor in the Carnegie Mellon University School of Computer Science.
The Secure C Programming sections at the ends of Chapters 2–13 discuss many important topics, including testing for arithmetic overflows, using unsigned integer types, new more secure functions in the C standard’s Annex K, the importance of checking the status information returned by standard-library functions, range checking, secure random-number generation, array bounds checking, techniques for preventing buffer overflows, input validation, avoiding undefined behaviors, choosing functions that return status information vs. similar functions that do not, ensuring that pointers are always NULL or contain valid addresses, preferring C functions to preprocessor macros, and more.

Teaching Approach

C for Programmers with an Introduction to C11 contains a rich collection of examples. We focus on good software engineering, stressing program clarity.

Syntax Shading. For readability, we syntax shade the code, similar to the way most IDEs and code editors syntax color code. Our syntax-shading conventions are:

- comments appear like this
- keywords appear like this
- constants and literal values appear like this
- all other code appears in black

Code Highlighting. We place gray rectangles around the key code segments in each source-code program.

Using Fonts for Emphasis. We place the key terms and the index’s page reference for each defining occurrence in bold text for easy reference. We emphasize on-screen components in the bold Helvetica font (e.g., the File menu) and C program text in the Lucida font (for example, int x = 5;).

Objectives. Each chapter includes a list of chapter objectives.

Illustrations/Figures. Abundant charts, tables, line drawings, flowcharts, programs and program outputs are included.

Programming Tips. We include programming tips to help you focus on important aspects of program development. These tips and practices represent the best we’ve gleaned from a combined eight decades of programming and corporate training experience.

Good Programming Practices

The Good Programming Practices call attention to techniques that will help you produce programs that are clearer, more understandable and more maintainable.

Common Programming Errors

Pointing out these Common Programming Errors reduces the likelihood that you’ll make them.

Error-Prevention Tips

These tips contain suggestions for exposing and removing bugs from your programs; many describe aspects of C that prevent bugs from getting into programs in the first place.
We've included an extensive index, which is especially useful when you use the book as a reference. Defining occurrences of key terms are highlighted with a bold page number.

Software Used in C for Programmers with an Introduction to C11

We wrote this book using the free GNU C compiler (gcc.gnu.org/install/binaries.html), which is already installed on most Linux systems and can be installed on Mac OS X, and Windows systems and Microsoft's free Visual Studio Express 2012 for Windows Desktop (www.microsoft.com/express). The Visual C++ compiler in Visual Studio can compile both C and C++ programs. Apple includes the LLVM compiler in its Xcode development tools, which Mac OS X users can download for free from the Mac App Store. Many other free C compilers are available online.

C Fundamentals: Parts I and II LiveLessons Video Training Product

Our C Fundamentals: Parts I and II LiveLessons video training product (available Fall 2013) shows you what you need to know to start building robust, powerful software with C. It includes 10+ hours of expert training synchronized with C for Programmers with an Introduction to C11. For additional information about Deitel LiveLessons video products, visit or contact us at deitel@deitel.com. You can also access our LiveLessons videos if you have a subscription to Safari Books Online (www.safaribooksonline.com).

Acknowledgments

We'd like to thank Abbey Deitel and Barbara Deitel for long hours devoted to this project. We're fortunate to have worked with the dedicated team of publishing professionals at Prentice Hall/Pearson. We appreciate the extraordinary efforts and mentorship of our friend and professional colleague of 17 years, Mark L. Taub, Editor-in-Chief of Pearson Technology Group. Carole Snyder did a marvelous job managing the review process. Chuti Prasertsith designed the cover with creativity and precision. John Fuller does a superb job managing the production of all our Deitel Developer Series books.
Reviewers
We wish to acknowledge the efforts of our reviewers, who under tight deadlines scrutinized the text and the programs and provided countless suggestions for improving the presentation: Dr. John F. Doyle (Indiana University Southeast), Hemanth H.M. (Software Engineer at SonicWALL), Vytautus Leonavicius (Microsoft), Robert Seacord (Secure Coding Manager at SEI/CERT, author of *The CERT C Secure Coding Standard* and technical expert for the international standardization working group for the programming language C) and José Antonio González Seco (Parliament of Andalusia).

Well, there you have it! C11 is a powerful programming language that will help you write high-performance programs quickly and effectively. C11 scales nicely into the realm of enterprise systems development to help organizations build their business-critical and mission-critical information systems. As you read the book, we would sincerely appreciate your comments, criticisms, corrections and suggestions for improving the text. Please address all correspondence to:

deitel@deitel.com

We’ll respond promptly and post corrections and clarifications on:

www.deitel.com/books/cfp

We hope you enjoy working with *C for Programmers with an Introduction to C11* as much as we enjoyed writing it!

Paul and Harvey Deitel

About the Authors

Paul Deitel, CEO and Chief Technical Officer of Deitel & Associates, Inc., is a graduate of MIT, where he studied Information Technology. Through Deitel & Associates, Inc., he has delivered hundreds of programming courses to industry, government and military clients, including Cisco, IBM, Siemens, Sun Microsystems, Dell, Fidelity, NASA at the Kennedy Space Center, the National Severe Storm Laboratory, White Sands Missile Range, Rogue Wave Software, Boeing, SunGard Higher Education, Nortel Networks, Puma, iRobot, Invensys and many more. He and his co-author, Dr. Harvey M. Deitel, are the world’s best-selling programming-language textbook/professional book/video authors.

Dr. Harvey Deitel, Chairman and Chief Strategy Officer of Deitel & Associates, Inc., has more than 50 years of experience in the computer field. Dr. Deitel earned B.S. and M.S. degrees in Electrical Engineering (studying computing) from MIT and a Ph.D. in Mathematics (studying computer science) from Boston University. He has extensive industry and college teaching experience, including earning tenure and serving as the Chairman of the Computer Science Department at Boston College before founding Deitel & Associates, Inc., in 1991 with his son, Paul Deitel. Dr. Deitel has delivered hundreds of professional programming seminars to major corporations, academic institutions, government organizations and the military. The Deitels’ publications have earned international recognition, with translations published in traditional Chinese, simplified Chinese, Korean, Japanese, German, Russian, Spanish, French, Polish, Italian, Portuguese, Greek, Urdu and Turkish.
Preface

Corporate Training from Deitel & Associates, Inc.

Deitel & Associates, Inc., founded by Paul Deitel and Harvey Deitel, is an internationally recognized authoring, corporate training and software development organization specializing in computer programming languages, object technology, Android and iOS app development and Internet and web software technology. The company offers instructor-led training courses delivered at client sites worldwide on major programming languages and platforms, including C, C++, Visual C++®, Java™, Visual C#®, Visual Basic®, XML®, Python®, object technology, Internet and web programming, Android™ app development, Objective-C and iOS® app development and a growing list of additional programming and software development courses. The company’s clients include some of the world’s largest companies as well as government agencies, branches of the military, and academic institutions.

Through its 37-year publishing partnership with Prentice Hall/Pearson, Deitel & Associates, Inc., publishes leading-edge programming professional books, college textbooks and LiveLessons video courses. Deitel & Associates, Inc. and the authors can be reached at:

deitel@deitel.com

To learn more about Deitel’s Dive-Into® Series Corporate Training curriculum, visit:

www.deitel.com/training

This book is also available as an e-book to Safari Books Online subscribers at

www.safaribooksonline.com

The last printed page of the book tells you how to get a free 45-day trial subscription to access the e-book.

Individuals wishing to purchase Deitel books and LiveLessons video training can do so through www.deitel.com. Bulk orders by corporations, the government, the military and academic institutions should be placed directly with Pearson. For more information, visit

www.informit.com/store/sales.aspx
Introduction to C Programming

Objectives
In this chapter you’ll:

- Write simple C programs.
- Use simple input and output statements.
- Use the fundamental data types.
- Use arithmetic operators.
- Learn the precedence of arithmetic operators.
- Write simple decision-making statements.
2.1 Introduction

The C language facilitates a structured and disciplined approach to computer-program design. In this chapter we introduce C programming and present several examples that illustrate many important features of C. In Chapters 3 and 4 we present an introduction to structured programming in C. We then use the structured approach throughout the remainder of the text.

2.2 A Simple C Program: Printing a Line of Text

We begin by considering a simple C program. Our first example prints a line of text. The program and its screen output are shown in Fig. 2.1.

```
1 // Fig. 2.1: fig02_01.c
2 // A first program in C.
3 #include <stdio.h>
4
5 // function main begins program execution
6 int main( void )
7 {
8    printf( "Welcome to C!\n" );
9 } // end function main
```

Welcome to C!

Fig. 2.1 | A first program in C.

Comments

This program illustrates several important C features. Lines 1 and 2 begin with //, indicating that these two lines are comments. Comments do not cause the computer to perform any action when the program is run. Comments are ignored by the C compiler and do not cause any machine-language object code to be generated. The preceding comment simply describes the figure number, file name and purpose of the program.

You can also use /*...*/ multi-line comments in which everything from /* on the first line to */ at the end of the last line is a comment. We prefer // comments because they’re shorter and they eliminate common programming errors that occur with /*...*/ comments, especially when the closing */ is omitted.
#include Preprocessor Directive
Line 3

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

is a directive to the C preprocessor. Lines beginning with # are processed by the preprocessor before compilation. Line 3 tells the preprocessor to include the contents of the standard input/output header (<stdio.h>) in the program. This header contains information used by the compiler when compiling calls to standard input/output library functions such as printf (line 8). We explain the contents of headers in more detail in Chapter 5.

Blank Lines and White Space
Line 4 is simply a blank line. You use blank lines, space characters and tab characters (i.e., “tabs”) to make programs easier to read. Together, these characters are known as white space. White-space characters are normally ignored by the compiler.

The main Function
Line 6

```c
int main( void )
```

is a part of every C program. The parentheses after main indicate that main is a function. C programs contain one or more functions, one of which must be main. Every program in C begins executing at the function main. Functions can return information. The keyword int to the left of main indicates that main “returns” an integer (whole-number) value. We’ll explain what this means when we demonstrate how to create your own functions in Chapter 5. For now, simply include the keyword int to the left of main in each of your programs. Functions also can receive information when they’re called upon to execute. The void in parentheses here means that main does not receive any information. In Chapter 14, we’ll show an example of main receiving information.

Good Programming Practice 2.1
Every function should be preceded by a comment describing the purpose of the function.

A left brace, {, begins the body of every function (line 7). A corresponding right brace ends each function (line 9). This pair of braces and the portion of the program between the braces is called a block. The block is an important program unit in C.

An Output Statement
Line 8

```c
printf( "Welcome to C!\n" );
```

instructs the computer to perform an action, namely to print on the screen the string of characters marked by the quotation marks. A string is sometimes called a character string, a message or a literal. The entire line, including the printf function (the “f” stands for “formatted”), its argument within the parentheses and the semicolon (;), is called a statement. Every statement must end with a semicolon (also known as the statement terminator). When the preceding printf statement is executed, it prints the message Welcome to C! on the screen. The characters normally print exactly as they appear between the double quotes in the printf statement.
Escape Sequences

Notice that the characters \n were not printed on the screen. The backslash (\) is called an escape character. It indicates that printf is supposed to do something out of the ordinary. When encountering a backslash in a string, the compiler looks ahead at the next character and combines it with the backslash to form an escape sequence. The escape sequence \n means newline. When a newline appears in the string output by a printf, the newline causes the cursor to position to the beginning of the next line on the screen. Some common escape sequences are listed in Fig. 2.2.

<table>
<thead>
<tr>
<th>Escape sequence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>Newline. Position the cursor at the beginning of the next line.</td>
</tr>
<tr>
<td>\t</td>
<td>Horizontal tab. Move the cursor to the next tab stop.</td>
</tr>
<tr>
<td>\a</td>
<td>Alert. Produces a sound or visible alert without changing the current cursor position.</td>
</tr>
<tr>
<td>&quot;</td>
<td>Backslash. Insert a backslash character in a string.</td>
</tr>
<tr>
<td>&quot;</td>
<td>Double quote. Insert a double-quote character in a string.</td>
</tr>
</tbody>
</table>

Fig. 2.2 | Some common escape sequences.

Because the backslash has special meaning in a string, i.e., the compiler recognizes it as an escape character, we use a double backslash (\") to place a single backslash in a string. Printing a double quote also presents a problem because double quotes mark the boundaries of a string—such quotes are not printed. By using the escape sequence \" in a string to be output by printf, we indicate that printf should display a double quote. The right brace, }, (line 9) indicates that the end of main has been reached.

Good Programming Practice 2.2

Add a comment to the line containing the right brace, }, that closes every function, including main.

We said that printf causes the computer to perform an action. As any program executes, it performs a variety of actions and makes decisions. Section 2.5 discusses decision making. Chapter 3 discusses this action/decision model of programming in depth.

The Linker and Executables

Standard library functions like printf and scanf are not part of the C programming language. For example, the compiler cannot find a spelling error in printf or scanf. When the compiler compiles a printf statement, it merely provides space in the object program for a “call” to the library function. But the compiler does not know where the library functions are—the linker does. When the linker runs, it locates the library functions and inserts the proper calls to these library functions in the object program. Now the object program is complete and ready to be executed. For this reason, the linked program is called an executable. If the function name is misspelled, the linker will spot the error, because it will not be able to match the name in the C program with the name of any known function in the libraries.
2.2 A Simple C Program: Printing a Line of Text

**Good Programming Practice 2.3**

Indent the entire body of each function one level of indentation (we recommend three spaces) within the braces that define the body of the function. This indentation emphasizes the functional structure of programs and helps make programs easier to read.

**Good Programming Practice 2.4**

Set a convention for the size of indent you prefer and then uniformly apply that convention. The tab key may be used to create indents, but tab stops may vary.

Using Multiple `printf`

The `printf` function can print `Welcome to C!` several different ways. For example, the program of Fig. 2.3 produces the same output as the program of Fig. 2.1. This works because each `printf` resumes printing where the previous `printf` stopped printing. The first `printf` (line 8) prints `Welcome` followed by a space, and the second `printf` (line 9) begins printing on the same line immediately following the space.

```c
// Fig. 2.3: fig02_03.c
// Printing on one line with two printf statements.
#include <stdio.h>

// function main begins program execution
int main( void )
{
    printf( "Welcome " );
    printf( "to C!\n" );
} // end function main
```

**Fig. 2.3** | Printing on one line with two `printf` statements.

One `printf` can print several lines by using additional newline characters as in Fig. 2.4. Each time the `\n` (newline) escape sequence is encountered, output continues at the beginning of the next line.

```c
// Fig. 2.4: fig02_04.c
// Printing multiple lines with a single printf.
#include <stdio.h>

// function main begins program execution
int main( void )
{
    printf( "Welcome\nto\nC!\n" );
} // end function main
```

**Fig. 2.4** | Printing multiple lines with a single `printf`.
2.3 Another Simple C Program: Adding Two Integers

Our next program uses the Standard Library function `scanf` to obtain two integers typed by a user at the keyboard, computes the sum of these values and prints the result using `printf`. The program and sample output are shown in Fig. 2.5. [In the input/output dialog of Fig. 2.5, we emphasize the numbers entered by the user in **bold.**]

```c
// Fig. 2.5: fig02_05.c
// Addition program.
#include <stdio.h>

// function main begins program execution
int main( void )
{
    int integer1; // first number to be entered by user
    int integer2; // second number to be entered by user
    int sum; // variable in which sum will be stored

    printf( "Enter first integer\n" ); // prompt
    scanf( "%d", &integer1 ); // read an integer

    printf( "Enter second integer\n" ); // prompt
    scanf( "%d", &integer2 ); // read an integer

    sum = integer1 + integer2; // assign total to sum

    printf( "Sum is %d\n", sum ); // print sum
}
```

The comment in line 2 states the purpose of the program. As we stated earlier, every program begins execution with `main`. The left brace { (line 7) marks the beginning of the body of `main`, and the corresponding right brace } (line 21) marks the end of `main`.

**Variables and Variable Definitions**

Lines 8–10

```c
int integer1; // first number to be entered by user
int integer2; // second number to be entered by user
int sum; // variable in which sum will be stored
```

are definitions. The names `integer1`, `integer2` and `sum` are the names of variables—locations in memory where values can be stored for use by a program. These definitions specify that variables `integer1`, `integer2` and `sum` are of type `int`, which means that they’ll hold `integer` values, i.e., whole numbers such as 7, −11, 0, 31914 and the like.
All variables must be defined with a name and a data type before they can be used in a program. For readers using the Microsoft Visual C++ compiler, note that we’re placing our variable definitions immediately after the left brace that begins the body of main. The C standard allows you to place each variable definition anywhere in main before that variable’s first use in the code. Some compilers, such as GNU gcc, have implemented this capability. We’ll address this issue in more depth in later chapters.

The preceding definitions could have been combined into a single definition statement as follows:

```c
int integer1, integer2, sum;
```

but that would have made it difficult to describe the variables with corresponding comments as we did in lines 8–10.

**Identifiers and Case Sensitivity**
A variable name in C is any valid identifier. An identifier is a series of characters consisting of letters, digits and underscores ( _) that does not begin with a digit. C is case sensitive—uppercase and lowercase letters are different in C, so a1 and A1 are different identifiers.

**Error-Prevention Tip 2.1**
 Avoid starting identifiers with the underscore character ( _ ) to prevent conflicts with compiler-generated identifiers and standard library identifiers.

**Good Programming Practice 2.5**
The first letter of an identifier used as a simple variable name should be a lowercase letter. Later in the text we’ll assign special significance to identifiers that use all capital letters.

**Good Programming Practice 2.6**
Multiple-word variable names can help make a program more readable. Separate the words with underscores as in total_commissions, or, if you run the words together, begin each word after the first with a capital letter as in totalCommissions. The latter style is preferred.

**Syntax Errors**
We discussed what syntax errors are in Chapter 1. Recall that the Microsoft Visual C++ compiler requires variable definitions to be placed after the left brace of a function and before any executable statements. Therefore, in the program in Fig. 2.5, inserting the definition of integer1 after the first printf would cause a syntax error in Visual C++.

**Common Programming Error 2.1**
Placing variable definitions among executable statements causes syntax errors in the Microsoft Visual C++ Compiler.

**Prompting Messages**
Line 12

```c
printf( "Enter first integer\n" ); // prompt
```

displays the literal "Enter first integer" and positions the cursor to the beginning of the next line. This message is called a prompt because it tells the user to take a specific action.
Chapter 2  Introduction to C Programming

The scanf Function and Formatted Inputs

The next statement uses `scanf` (the “f” stands for “formatted”) to obtain a value from the user. The function reads from the standard input, which is usually the keyboard. This `scanf` has two arguments, "%d" and &integer1. The first, the format control string, indicates the type of data that should be entered by the user. The %d conversion specifier indicates that the data should be an integer (the letter d stands for “decimal integer”). The % in this context is treated by `scanf` (and `printf` as we’ll see) as a special character that begins a conversion specifier. The second argument of `scanf` begins with an ampersand (&)—called the address operator—followed by the variable name. The &, when combined with the variable name, tells `scanf` the location (or address) in memory at which the variable integer1 is stored. The computer then stores the value that the user enters for integer1 at that location. The use of ampersand (&) is often confusing to novice programmers or to people who have programmed in other languages that do not require this notation. For now, just remember to precede each variable in every call to `scanf` with an ampersand. Some exceptions to this rule are discussed in Chapters 6 and 7. The use of the ampersand will become clear after we study pointers in Chapter 7.

Good Programming Practice 2.7

Place a space after each comma (,) to make programs more readable.

When the computer executes the preceding `scanf`, it waits for the user to enter a value for variable integer1. The user responds by typing an integer, then pressing the Enter key to send the number to the computer. The computer then assigns this number, or value, to the variable integer1. Any subsequent references to integer1 in this program will use this same value. Functions `printf` and `scanf` facilitate interaction between the user and the computer. Because this interaction resembles a dialogue, it’s often called interactive computing.

Line 15

```
printf( "Enter second integer\n" ); // prompt
```

displays the message Enter second integer on the screen, then positions the cursor to the beginning of the next line. This `printf` also prompts the user to take action.

Line 16

```
scanf( "%d", &integer2 ); // read an integer
```

obtains a value for variable integer2 from the user.

Assignment Statement

The assignment statement in line 18

```
sum = integer1 + integer2; // assign total to sum
```

calculates the total of variables integer1 and integer2 and assigns the result to variable sum using the assignment operator =. The statement is read as, “sum gets the value of integer1 + integer2.” Most calculations are performed in assignments. The = operator
and the + operator are called binary operators because each has two operands. The + operator's two operands are integer1 and integer2. The = operator's two operands are sum and the value of the expression integer1 + integer2.

**Good Programming Practice 2.8**

Place spaces on either side of a binary operator for readability.

**Printing with a Format Control String**

Line 20

```c
printf( "Sum is %d\n", sum ); // print sum
```

calls function printf to print the literal Sum is followed by the numerical value of variable sum on the screen. This printf has two arguments, "Sum is %d\n" and sum. The first argument is the format control string. It contains some literal characters to be displayed, and it contains the conversion specifier %d indicating that an integer will be printed. The second argument specifies the value to be printed. Notice that the conversion specifier for an integer is the same in both printf and scanf—this is the case for most C data types.

**Calculations in printf Statements**

Calculations can also be performed inside printf statements. We could have combined the previous two statements into the statement

```c
printf( "Sum is %d\n", integer1 + integer2 );
```

The right brace, }, at line 21 indicates that the end of function main has been reached.

**Common Programming Error 2.2**

Forgetting to precede a variable in a scanf statement with an ampersand when that variable should, in fact, be preceded by an ampersand results in an execution-time error. On many systems, this causes a "segmentation fault" or "access violation." Such an error occurs when a user's program attempts to access a part of the computer's memory to which it does not have access privileges. The precise cause of this error will be explained in Chapter 7.

**Common Programming Error 2.3**

Preceding a variable included in a printf statement with an ampersand when, in fact, that variable should not be preceded by an ampersand.

### 2.4 Arithmetic in C

Most C programs perform calculations using the C arithmetic operators (Fig. 2.6). The asterisk (*) indicates multiplication and the percent sign (%) denotes the remainder operator, which is introduced below. In algebra, to multiply a times b, we simply place these single-letter variable names side by side, as in ab. In C, however, if we were to do this, ab would be interpreted as a single, two-letter name (or identifier). Therefore, multiplication must be explicitly denoted by using the * operator, as in a * b. The arithmetic operators are all binary operators. For example, the expression 3 + 7 contains the binary operator + and the operands 3 and 7.
Chapter 2  Introduction to C Programming

Integer Division and the Remainder Operator

Integer division yields an integer result. For example, the expression $7/4$ evaluates to 1 and the expression $17/5$ evaluates to 3. C provides the remainder operator, %, which yields the remainder after integer division. The remainder operator is an integer operator that can be used only with integer operands. The expression $x\%y$ yields the remainder after $x$ is divided by $y$. Thus, $7\%4$ yields 3 and $17\%5$ yields 2. We’ll discuss many interesting applications of the remainder operator.

Arithmetic Expressions in Straight-Line Form

Arithmetic expressions in C must be written in straight-line form to facilitate entering programs into the computer. Thus, expressions such as “a divided by b” must be written as a/b so that all operators and operands appear in a straight line. The algebraic notation

\[
\frac{a}{b}
\]

is generally not acceptable to compilers, although some special-purpose software packages do support more natural notation for complex mathematical expressions.

Parentheses for Grouping Subexpressions

Parentheses are used in C expressions in the same manner as in algebraic expressions. For example, to multiply $a$ times the quantity $b+c$ we write $a*(b+c)$.

Rules of Operator Precedence

C applies the operators in arithmetic expressions in a precise sequence determined by the following rules of operator precedence, which are generally the same as those in algebra:

1. Operators in expressions contained within pairs of parentheses are evaluated first. Parentheses are said to be at the “highest level of precedence.” In cases of nested, or embedded, parentheses, such as

\[
( (a + b) + c )
\]

the operators in the innermost pair of parentheses are applied first.
2. Multiplication, division and remainder operations are applied next. If an expression contains several multiplication, division and remainder operations, evaluation proceeds from left to right. Multiplication, division and remainder are said to be on the same level of precedence.

3. Addition and subtraction operations are evaluated next. If an expression contains several addition and subtraction operations, evaluation proceeds from left to right. Addition and subtraction also have the same level of precedence, which is lower than the precedence of the multiplication, division and remainder operations.

4. The assignment operator (=) is evaluated last.

The rules of operator precedence specify the order C uses to evaluate expressions. When we say evaluation proceeds from left to right, we’re referring to the associativity of the operators. We’ll see that some operators associate from right to left. Figure 2.7 summarizes these rules of operator precedence for the operators we’ve seen so far.

<table>
<thead>
<tr>
<th>Operator(s)</th>
<th>Operation(s)</th>
<th>Order of evaluation (precedence)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( )</td>
<td>Parentheses</td>
<td>Evaluated first. If the parentheses are nested, the expression in the innermost pair is evaluated first. If there are several pairs of parentheses “on the same level” (i.e., not nested), they’re evaluated left to right.</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>Evaluated second. If there are several, they’re evaluated left to right.</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td></td>
</tr>
<tr>
<td>%</td>
<td>Remainder</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>Addition</td>
<td>Evaluated third. If there are several, they’re evaluated left to right.</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>Assignment</td>
<td>Evaluated last.</td>
</tr>
</tbody>
</table>

---

**Fig. 2.7** | Precedence of arithmetic operators.

**Sample Algebraic and C Expressions**

Now let’s consider several expressions in light of the rules of operator precedence. Each example lists an algebraic expression and its C equivalent. The following expression calculates the arithmetic mean (average) of five terms.

Algebra: \( m = \frac{a + b + c + d + e}{5} \)

C: \( m = \frac{( a + b + c + d + e )}{5}; \)

The parentheses are required to group the additions because division has higher precedence than addition. The entire quantity \( ( a + b + c + d + e ) \) should be divided by 5. If the parentheses are erroneously omitted, we obtain \( a + b + c + d + e / 5 \), which evaluates incorrectly as

\[ a + b + c + d + \frac{e}{5} \]

---

1. We use simple examples to explain the order of evaluation of expressions. Subtle issues occur in more complex expressions that you’ll encounter later in the book. We’ll discuss these issues as they arise.
The following expression is the equation of a straight line:

<table>
<thead>
<tr>
<th>Algebra:</th>
<th>( y = mx + b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>C:</td>
<td>( y = m \times x + b );</td>
</tr>
</tbody>
</table>

No parentheses are required. The multiplication is evaluated first because multiplication has a higher precedence than addition.

The following expression contains remainder (\%), multiplication, division, addition, subtraction and assignment operations:

<table>
<thead>
<tr>
<th>Algebra:</th>
<th>( z = pr %q + w/c - y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>C:</td>
<td>( z = p \times r %q + w / x - y );</td>
</tr>
</tbody>
</table>

The circled numbers indicate the order in which C evaluates the operators. The multiplication, remainder and division are evaluated first in left-to-right order (i.e., they associate from left to right) because they have higher precedence than addition and subtraction. The addition and subtraction are evaluated next. They’re also evaluated left to right. Finally, the result is assigned to the variable \( z \).

Not all expressions with several pairs of parentheses contain nested parentheses. For example, the following expression does not contain nested parentheses—instead, the parentheses are said to be “on the same level.”

\[
6 \ 1 \ 2 \ 4 \ 3 \ 5
\]

**Evaluation of a Second-Degree Polynomial**

To develop a better understanding of the rules of operator precedence, let’s see how C evaluates a second-degree polynomial.

| y = a \times x \times x + b \times x + c; |
|--------|-----------------------------|
| 6 \ 1 \ 2 \ 4 \ 3 \ 5 |

The circled numbers under the statement indicate the order in which C performs the operations. There’s no arithmetic operator for exponentiation in C, so we’ve represented \( x^2 \) as \( x \times x \). The C Standard Library includes the `pow` (“power”) function to perform exponentiation. Because of some subtle issues related to the data types required by `pow`, we defer a detailed explanation of `pow` until Chapter 4.

Suppose variables \( a \), \( b \), \( c \) and \( x \) in the preceding second-degree polynomial are initialized as follows: \( a = 2 \), \( b = 3 \), \( c = 7 \) and \( x = 5 \). Figure 2.8 illustrates the order in which the operators are applied.

As in algebra, it’s acceptable to place unnecessary parentheses in an expression to make the expression clearer. These are called **redundant parentheses**. For example, the preceding statement could be parenthesized as follows:

\[
y = (a \times x \times x) + (b \times x) + c;
\]
Executable statements either perform actions (such as calculations or input or output of data) or make decisions (we’ll soon see several examples of these). We might make a decision in a program, for example, to determine whether a person’s grade on an exam is greater than or equal to 60 and whether the program should print the message “Congratulations! You passed.” This section introduces a simple version of C’s if statement that allows a program to make a decision based on the truth or falsity of a statement of fact called a condition. If the condition is true (i.e., the condition is met), the statement in the body of the if statement is executed. If the condition is false (i.e., the condition isn’t met), the body statement isn’t executed. Whether the body statement is executed or not, after the if statement completes, execution proceeds with the next statement after the if statement.

Conditions in if statements are formed by using the equality operators and relational operators summarized in Fig. 2.9. The relational operators all have the same level of precedence and they associate left to right. The equality operators have a lower level of precedence than the relational operators and they also associate left to right. [Note: In C, a condition may actually be any expression that generates a zero (false) or nonzero (true) value.]

### Common Programming Error 2.5
Confusing the equality operator == with the assignment operator. To avoid this confusion, the equality operator should be read “double equals” and the assignment operator should be read “gets” or “is assigned the value of.” As you’ll see, confusing these operators may not cause an easy-to-recognize compilation error, but may cause extremely subtle logic errors.
Figure 2.10 uses six if statements to compare two numbers entered by the user. If the condition in any of these if statements is true, the printf statement associated with that if executes. The program and three sample execution outputs are shown in the figure.

```c
// Fig. 2.10: fig02_10.c
// Using if statements, relational operators, and equality operators.
#include <stdio.h>

int main( void )
{
    int num1; // first number to be read from user
    int num2; // second number to be read from user
    printf( "Enter two integers, and I will tell you\n" );
    printf( "the relationships they satisfy: " );
    scanf( "%d%d", &num1, &num2 ); // read two integers

    if ( num1 == num2 ) {
        printf( "%d is equal to %d\n", num1, num2 );
    } // end if

    if ( num1 != num2 ) {
        printf( "%d is not equal to %d\n", num1, num2 );
    } // end if

    if ( num1 < num2 ) {
        printf( "%d is less than %d\n", num1, num2 );
    } // end if
```

Fig. 2.10 | Using if statements, relational operators, and equality operators. (Part 1 of 2.)
The program uses `scanf` (line 15) to input two numbers. Each conversion specifier has a corresponding argument in which a value will be stored. The first `%d` converts a value to be stored in the variable `num1`, and the second `%d` converts a value to be stored in the variable `num2`.

```c
if ( num1 > num2 ) {
    printf( "%d is greater than %d\n", num1, num2 );
} // end if

if ( num1 <= num2 ) {
    printf( "%d is less than or equal to %d\n", num1, num2 );
} // end if

if ( num1 >= num2 ) {
    printf( "%d is greater than or equal to %d\n", num1, num2 );
} // end if
```

Enter two integers, and I will tell you the relationships they satisfy: 3 7
3 is not equal to 7
3 is less than 7
3 is less than or equal to 7

Enter two integers, and I will tell you the relationships they satisfy: 22 12
22 is not equal to 12
22 is greater than 12
22 is greater than or equal to 12

Enter two integers, and I will tell you the relationships they satisfy: 7 7
7 is equal to 7
7 is less than or equal to 7
7 is greater than or equal to 7

Good Programming Practice 2.9

Although it’s allowed, there should be no more than one statement per line in a program.

Common Programming Error 2.6

Placing commas (when none are needed) between conversion specifiers in the format control string of a `scanf` statement.
Comparing Numbers
The if statement in lines 17–19

```c
if ( num1 == num2 ) {
    printf( "%d is equal to %d\n", num1, num2 );
}
```

compares the values of variables num1 and num2 to test for equality. If the values are equal, the statement in line 18 displays a line of text indicating that the numbers are equal. If the conditions are true in one or more of the if statements starting in lines 21, 25, 29, 33 and 37, the corresponding body statement displays an appropriate line of text. Indenting the body of each if statement and placing blank lines above and below each if statement enhances program readability.

Common Programming Error 2.7
Placing a semicolon immediately to the right of the right parenthesis after the condition in an if statement.

A left brace, {, begins the body of each if statement (e.g., line 17). A corresponding right brace, }, ends each if statement’s body (e.g., line 19). Any number of statements can be placed in the body of an if statement.2

Good Programming Practice 2.10
A lengthy statement may be spread over several lines. If a statement must be split across lines, choose breaking points that make sense (such as after a comma in a comma-separated list). If a statement is split across two or more lines, indent all subsequent lines. It’s not correct to split identifiers.

Figure 2.11 lists from highest to lowest the precedence of the operators introduced in this chapter. Operators are shown top to bottom in decreasing order of precedence. The equals sign is also an operator. All these operators, with the exception of the assignment operator =, associate from left to right. The assignment operator (=) associates from right to left.

Good Programming Practice 2.11
Refer to the operator precedence chart when writing expressions containing many operators. Confirm that the operators in the expression are applied in the proper order. If you’re uncertain about the order of evaluation in a complex expression, use parentheses to group expressions or break the statement into several simpler statements. Be sure to observe that some of C’s operators such as the assignment operator (=) associate from right to left rather than from left to right.

Some of the words we’ve used in the C programs in this chapter—in particular int and if—are keywords or reserved words of the language. Figure 2.12 contains the C keywords. These words have special meaning to the C compiler, so you must be careful not to use these as identifiers such as variable names.

---
2. Using braces to delimit the body of an if statement is optional when the body contains only one statement. Many programmers consider it good practice to always use these braces. In Chapter 3, we’ll explain the issues.
We mentioned *The CERT C Secure Coding Standard* in the Preface and indicated that we would follow certain guidelines that will help you avoid programming practices that open systems to attacks.

**Avoid Single-Argument `printf`s**

One such guideline is to avoid using `printf` with a single string argument. If you need to display a string that terminates with a newline, use the `puts` function, which displays its string argument followed by a newline character. For example, in Fig. 2.1, line 8

```c
printf( "Welcome to C!\n" );
```

should be written as:

```c
puts( "Welcome to C!" );
```
We did not include \n in the preceding string because puts adds it automatically.

If you need to display a string without a terminating newline character, use printf with two arguments—a "%s" format control string and the string to display. The %s conversion specifier is for displaying a string. For example, in Fig. 2.3, line 8

```c
printf( "Welcome " );
```

should be written as:

```c
printf( "%s", "Welcome " );
```

Although the printfs in this chapter as written are actually not insecure, these changes are responsible coding practices that will eliminate certain security vulnerabilities as we get deeper into C—we’ll explain the rationale later in the book. From this point forward, we use these practices in the chapter examples and you should use them in your own code.

For more information on this issue, see CERT C Secure Coding rule FIO30-C

```c
www.securecoding.cert.org/confluence/display/seccode/
FIO30-C.+Exclude+user+input+from+format+strings
```

In Chapter 6’s Secure C Programming section, we’ll explain the notion of user input as referred to by this CERT guideline.

**scanf and printf, scanf_s and printf_s**

We introduced scanf and printf in this chapter. We’ll be saying more about these in subsequent Secure C Coding Guidelines sections. We’ll also discuss scanf_s and printf_s, which were introduced in C11.
Symbols
\t horizontal-tab escape sequence 22
\& bitwise exclusive OR operator 271
\& bitwise exclusive OR assignment operator 278
__func__ predefined identifier 411
__VA_ARGS__ 408
\_Pragma operator 407
, (comma operator) 62, 65
!, logical negation (NOT) operator 77, 78
!= inequality operator 32
?: conditional operator 41, 54, 119
.dot operator 262
.structure member operator 263
\^ bitwise exclusive OR operator 271
\^ bitwise inclusive OR operator 271
\^ bitwise one's complement 271
\^ bitwise complement operator 271
/\ bitwise AND operator 271
//\ bitwise AND operator 271
\| pipe 352
\| bitwise inclusive OR operator 271
\| bitwise inclusive OR operator 271
\| bitwise complement operator 271
Numerics
0 Conversion specifier 26, 252, 253
5 Conversion specifier 244
% remainder operator 27, 99
% conversion specifier 244
%= remainder assignment operator 54
%c conversion specifier 93, 244, 254
%d conversion specifier 93
%e conversion specifier 242, 253
%f conversion specifier 48, 93
%g conversion specifier 253
%h conversion specifier 93
%H conversion specifier 93
%i conversion specifier 252
%ld conversion specifier 93
%lf conversion specifier 93
%I conversion specifier 93
%l conversion specifier 93
%lu conversion specifier 93
%lu conversion specifier 93
%p conversion specifier 170, 244
%q conversion specifier 36, 196, 244, 254
%s conversion specifier 50, 93, 240
%x conversion specifier 252
+ flag 248
+ flag 247
- minus operator 54
- unary plus operator 54
-- operator 52, 54, 189
++ operator 52, 54, 189
+= addition assignment operator 52, 54
< less than operator 32
< left-shift operator 271
<< left-shift assignment operator 278
= assignment operator 54
== subtraction assignment operator 54
== equality operator 81
> greater than operator 32
> redirect output symbol 353
-> structure pointer operator 262
>> append output symbol 353
>> right-shift operator 271
>>> right shift assignment operator 278
| bitwise inclusive OR operator 271
| pipe 352
|= bitwise inclusive OR assignment operator 278
| 119
|\ bitwise OR operator 77
| bitwise one’s complement 271
|= bitwise complement operator 276
A
a file open mode 291
a.out 7
ab file open mode 291
ab file open mode 291
ab+ file open mode 291
abnormal program termination 360
abort function 349
absolute-value 86
abstraction 87
access privileges 177
access violation 27, 208, 244
action 21, 22, 31, 43
action symbol 39
action/decision model 22
actions 31
active window 429
add an integer to a pointer 188
addition assignment operator (+=) 51
address 320
address of a bit field 282
address operator (&) 26, 99, 134, 169, 172, 182
aggregate data types 180
aggregates 259
alert (\a) 22
algorithm
insertion sort 384
merge sort 387
selection sort 380
aligning 239
American National Standards Committee on Computers and Information Processing
American National Standards Institute (ANSI) 3, 3
amperand (&) 26, 27
AND 270
Android 18
operating system 16, 18
smartphone 18
Annex K 165, 166
ANSI 3
Apache Software Foundation 17
append output symbol >> 353
Apple Inc. 17
Apple Macintosh 18
argc 355
argument 21
argument (of a function) 85
arguments 344
argv 355
arithmetic assignment operators 52
+=, -=, *=, /=, and %= 52
arithmetic conversion rules 92
arithmetic expressions 188
n! 113
n! 113
name 59, 124
name of a control variable 59
name of an array 123
natural logarithm 86
negative value 376
negative binary numbers 368
nested if...else statement 42, 43
nestes parentheses 28, 30
nesting 49

newline (\n) 22, 40, 134, 206, 208, 209, 256
NeXTSTEP operating system 18
nodes 313, 314
non-constant pointer to constant data
177, 178, 179
non-constant pointer to non-constant data
177
nonfatal error 91
NULL 169, 190, 194, 288, 313, 320
null character (\n') 133, 134, 178, 194,
207
NULL pointer 362
null-terminated string 194
Number Systems Appendix 368
numeric codes 225

O
O(1) 379
O(log n) time 392
O(n) time 379
O(n) time 380, 383, 387
object 5
object code 7
object-oriented programming (OOP) 18,
87, 5
object program 22
Objective-C 18
Objective-C programming language 5
octal (base-8) number system 369
octal number 209, 215, 239
off-by-one error 61
offset 190, 300
one's complement 276
one's complement notation 376
ones position 369
open a file 288
open file table 286
Open Handset Alliance 18
open source 17, 18
operand 27
operating system 2, 16, 18
operator precedence 34
operator precedence chart 365
Operator sizeof when applied to an ar-
ray name returns the number of bytes
in the array 185
operators 51
order 38
order of evaluation of operands 118
order of operands of operators 119
OS X 18
outer block 110
out-of-bounds array elements 165
oval symbol 39
overflow 360

P
packets in a computer network 329
padding 280
page layout software 206
parameter 87
parameter list 89
parameter of a function 88
parameter passing 176
parameter types 184
parent node 335
parentheses () 28, 34
pass-by-reference 138, 139, 168, 172,
174, 176, 180, 182
pass-by-value 172, 174, 176, 180
passing an array 139
passing an array element 139
Passing arrays and individual array ele-
ments to functions 140
percent sign (%) 27
performance 4
performance requirements 109
PHP 5
pipe symbol (|) 352
piping 352
pointer 168, 170, 172
pointer arithmetic 188, 191
pointer arrow (->) operator 262
pointer comparisons 190
pointer expression 190
pointer notation 174, 190, 193
pointer parameter 173
pointer subscripting 191
pointer to a function 199
pointer to pointer (double indirection) 320
pointer to the structure 262
pointer to void (void *) 190, 313
pointer variable 181, 182
pointer/offset notation 190
pointer/subscript notation 191
poll 129
polynomial 30, 31
pop 323
pop off a stack 94
portability 4
Portability Tips overview xviii
portable 4
portable code 4
portable programs 2
position number 123
positional notation 369
positional value 369, 370
positional values in the decimal number
system 370
postdecrement 52
postfix increment and decrement opera-
tors 52
postincrement 52
postorder 336
postOrder traversal 339, 340
pow (power) function 30, 66, 67, 86
power 86
#pragma 347
#pragma processor directive 347
precedence 28, 124, 170
precedence of arithmetic operators 29, 34
precision 48, 67, 239, 239, 241
predecrement operator 52
predefined symbolic constants 348
predicate function 320
prefix increment and decrement operators
52
preincrement operator 52
preincrementing 53
preincrementing vs. postincrementing 53
preorder 336
preOrder traversal 339
preprocess phase 5
preprocessor 7, 97
preprocessor directive 7, 343, 346
principle of least privilege 109, 142, 176, 180, 184, 185
print characters 210
print debugger command 440
printf 239
printf function 21
Printing a string one character at a time using a non-constant pointer to constant data 178
printing character 212
Printing positive and negative numbers with and without the + flag 248
probability 99
Processing a queue 329
program execution stack 94
Program to simulate the game of craps 104
programmer-defined function 84
programmer-defined maximum function 90
prompt 25
pseudo-random numbers 102
push 323, 327
push onto a stack 94
putchar 217, 287
puts 219
puts function 35
quadratic run time 380
queue 168, 259, 329
queue 329
probability 99, 103
RAND_MAX 99
rand 99
remainder operator (%) 27, 99
repetition statement 38, 43
replacement text 128, 344
requirements 109
reserved word 34
restrict 409
restricted pointer 409
return 172
return from a function 85
return key 7
return statement 88, 90
return type 184
return value type 88
return without expression 411
reusable software 5
Richards, Martin 2
right brace (}) 40
right child 335
right justification 239
right justified 67, 245
right subtree 335
right-justifying integers 245
Right-justifying integers in a field 245
Ritchie, D. 2
Rolling a six-sided die 6000 times 100
root node of a binary tree 335
rounded 49
rounding 113, 239
rounding toward negative infinity 409
rounding toward zero 409
rows 155
rules of operator 28
run debugger command 438
rvalue ("right value") 80
r file open mode 291
r+ file open mode 291, 292
radians 86
raise 360
rand 99
RAND_MAX 99, 103
random number 98
random number generation 195
random number 98
randomizing 102
range checking 82
rb file open mode 291
r+ file open mode 291
readability 34, 60
record 180, 287
rectangle symbol 39
recursion 112, 119
recursion step 113
recursive call 113, 114
recursive calls to method fibonacci 118
recursive definition 113
recursive evaluation 114
recursive function 112
vs. iteration 119
red breakpoint circle: solid 428
redirect input from a file 352
redirect input or output 239
redirect input symbol < 352
redirect output symbol > 353
redundant parentheses 30
register 108
reinventing the wheel 4, 84
relational operators 31
reliable integer division 409
remainder 86
remainder operator (%) 27, 99
repetition statement 38, 43
replacement text 128, 344
requirements 109
reserved word 34
restrict 409
restricted pointer 409
return 172
return from a function 85
return key 7
return statement 88, 90
return type 184
return value type 88
return without expression 411
reusable software 5
Richards, Martin 2
right brace (}) 40
right child 335
right justification 239
right justified 67, 245
right subtree 335
right-justifying integers 245
Right-justifying integers in a field 245
Ritchie, D. 2
Rolling a six-sided die 6000 times 100
root node of a binary tree 335
rounded 49
rounding 113, 239
rounding toward negative infinity 409
rounding toward zero 409
rows 155
rules of operator 28
run debugger command 438
rvalue ("right value") 80
S
savings account example 65
scalable 128
scalar 139
scalars 182
scaling factor 99
scaling factor 99, 104
scan characters 252
scan set 254
scanf 239
scanf function 26
scientific notation 165
scientific notation 241
scope 346
Scoping example 110
screen 8
search functions of the string handling library 225
search key 150
searching 149, 151
searching a binary tree 340
searching strings 221
second-degree polynomial 31
secondary storage device 7
seed 102
seed the rand function 102
SEEK_CUR 302
SEEK_END 302
SEEK_SET 300, 302
segmentation fault 27, 177
segmentation violations 360
selection sort algorithm 380, 381, 383
selection statement 40
selection structure 38
self-referential structure 260
semicolon (;) 21, 34
sentinel-controlled repetition 58
sentinel value 46, 48
sequence structure 38, 39
sequential access file 287
sequential execution 38
sequential file 287
set debugger command 442
<stdio.h> 98
shell prompt on Linux 8
shift 99
Shifted, scaled integers produced by 1 + rand() % 6 99
shifting value 104
short int 73, 92
short-circuit evaluation 78
sibling 335
de side effect 98, 109, 119
SIGABRT 360
SIGFPE 360
SIGILL 360
SIGINT 360
signal 360
Signal handling 360
signal handling library 360
signal value 46
<stdio.h> 98, 360
signed decimal integer 240
SIGSEV 360
SIGTERM 360
simple condition 77
simulation 99, 99, 195
sin function 86
sine 86
single quote (') character 244
single-selection statement 39
single-entry/single-exit control statements 40
single-subscribed array 177, 184
sinking sort 142
size_t 126, 222, 225
sizeof operator 185, 261, 313, 346
small circle symbol 39
smartphone 18
software engineering 76, 109, 184
Software Engineering Observations overview 87
software reusability 4, 87, 184, 357
Solution Configurations 427
sort algorithms
insertion sort 384
merge sort 387
selection sort 380
sort key 379
sorting 412
sorting data 379
SourceForge 17
space 256
space flag 256
space 256
space 256
space 256
space 256
space 256
space 256
space 256
space 256
space 256
space 256
space 256
space 256
space 256
space 256
space 256
space 256
space 256
}
u or U for an unsigned int 359
union 268, 269
union operator sizeof 185
unconditional branch 363
#undef 348
﻿#define preprocessor directive 346
underscore (_) 25
union 268, 269
UNIX 2, 70
unnamed bit field 280
unnamed bit field with a zero width 282
unresolved references 357
unsafe macro 350
unsigned 102
unsigned decimal integer 240
unsigned hexadecimal integer 240
unsigned int 93
unsigned integer 270
unsigned long 216
unsigned long long int 114, 115, 116
unsigned octal integer 240
unsigned short 93
uppercase letter 97
usual arithmetic conversion rules 92
utility function 98
va_copy macro 411
va_end 355
va_list 354
validate data 82
value 124
variable 24
variable arguments header stdarg.h 353
variable initialization 194
variable-length argument list 353, 354
variable-length array (VLA) 162, 405
vertical spacing 60
vertical tab (\v) 209
vi 6
Visual C# programming language 5, 5
Visual Studio 396
Visual C++ programming language 5
Visual Studio 6
   Quick Info box 429
void * (pointer to void) 190, 232, 313
w file open mode 291
w+ file open mode 291
w+ file update mode 291
watch debugger command 446
Watch window (Visual Studio debugger) 430, 431
wb file open mode 291
wb+ file open mode 291
wchar.h 396
wctype.h 396
while repetition statement 43
white-space character 21, 40, 256
white-space string literals separated 134
width of a bit field 279, 282
Wikipedia 5
Windows 16, 360
Windows operating system 17
Windows Phone 7 16
worst-case runtime for an algorithm 379
Wozniak, Steve 17
writing to a file 289
X
x 244
Xerox PARC (Palo Alto Research Center) 17
Y
yellow arrow in break mode 429
Z
0 (zero) flag 250
zeroth element 123