Code Reading
The Open Source Perspective

Diomidis Spinellis
Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Addison-Wesley was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers discounts on this book when ordered in quantity for bulk purchases and special sales. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact:
International Sales
(317) 581-3793
international@pearsontechgroup.com

Visit Addison-Wesley on the Web: www.awprofessional.com

Library of Congress Cataloging-in-Publication Data
Spinellis, Diomidis.
Code reading / Diomidis Spinellis.
p. cm.
Contents: The open source perspective.
Includes bibliographical references and Index.
1. Computer programming. 2. Coding theory. I. Title.
QA76.6 .S675 2003
005.1--dc21 2002038563

Copyright © 2003 by Pearson Education, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher. Printed in the United States of America. Published simultaneously in Canada.

For information on obtaining permission for use of material from this work, please submit a written request to:

Pearson Education, Inc.
Rights and Contracts Department
75 Arlington Street, Suite 300
Boston, MA 02116
Fax: (617) 848-7047

ISBN 0-201-79940-5

Text printed in the United States at Offset Paperback Manufacturers in Laflin, Pennsylvania.
11th Printing March 2009
# Contents at a Glance

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foreword</td>
<td>xxi</td>
</tr>
<tr>
<td>Preface</td>
<td>xxv</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2 Basic Programming Elements</td>
<td>19</td>
</tr>
<tr>
<td>3 Advanced C Data Types</td>
<td>61</td>
</tr>
<tr>
<td>4 C Data Structures</td>
<td>95</td>
</tr>
<tr>
<td>5 Advanced Control Flow</td>
<td>143</td>
</tr>
<tr>
<td>6 Tackling Large Projects</td>
<td>179</td>
</tr>
<tr>
<td>7 Coding Standards and Conventions</td>
<td>225</td>
</tr>
<tr>
<td>8 Documentation</td>
<td>241</td>
</tr>
<tr>
<td>9 Architecture</td>
<td>267</td>
</tr>
<tr>
<td>10 Code-Reading Tools</td>
<td>339</td>
</tr>
<tr>
<td>11 A Complete Example</td>
<td>379</td>
</tr>
<tr>
<td>A Outline of the Code Provided</td>
<td>399</td>
</tr>
<tr>
<td>B Source Code Credits</td>
<td>403</td>
</tr>
<tr>
<td>C Referenced Source Files</td>
<td>405</td>
</tr>
<tr>
<td>D Source Code Licenses</td>
<td>413</td>
</tr>
<tr>
<td>E Maxims for Reading Code</td>
<td>425</td>
</tr>
<tr>
<td>Bibliography</td>
<td>445</td>
</tr>
<tr>
<td>Index</td>
<td>459</td>
</tr>
<tr>
<td>Author Index</td>
<td>491</td>
</tr>
</tbody>
</table>
## Contents

*Figures* ............................................................... xiii  
*Tables* ................................................................. xix  
*Foreword* .............................................................. xxi  
*Preface* ........................................................................ xxv  

### 1 Introduction ............................................................................ 1  
1.1 Why and How to Read Code .................................................. 2  
  1.1.1 Code as Literature .................................................. 2  
  1.1.2 Code as Exemplar .................................................. 5  
  1.1.3 Maintenance ...................................................... 6  
  1.1.4 Evolution ....................................................... 7  
  1.1.5 Reuse ......................................................... 9  
  1.1.6 Inspections .................................................... 9  
1.2 How to Read This Book .................................................... 10  
  1.2.1 Typographical Conventions .................................. 10  
  1.2.2 Diagrams .................................................. 12  
  1.2.3 Exercises .................................................. 13  
  1.2.4 Supplementary Material .................................. 14  
  1.2.5 Tools ...................................................... 14  
  1.2.6 Outline ................................................... 15  
  1.2.7 The Great Language Debate .............................. 15  
Further Reading ........................................................................ 17  

### 2 Basic Programming Elements ................................................. 19  
2.1 A Complete Program ...................................................... 19  
2.2 Functions and Global Variables .......................................... 25
2.3 while Loops, Conditions, and Blocks ........................................ 28
2.4 switch Statements .................................................................... 32
2.5 for Loops ............................................................................. 34
2.6 break and continue Statements ............................................. 37
2.7 Character and Boolean Expressions ...................................... 39
2.8 goto Statements .................................................................. 43
2.9 Refactoring in the Small .................................................... 45
2.10 do Loops and Integer Expressions .................................... 51
2.11 Control Structures Revisited ........................................... 54
Further Reading ................................................................. 60

3 Advanced C Data Types ......................................................... 61
3.1 Pointers ........................................................................... 61
  3.1.1 Linked Data Structures .................................................. 62
  3.1.2 Dynamic Allocation of Data Structures ......................... 62
  3.1.3 Call by Reference ......................................................... 63
  3.1.4 Data Element Access .................................................... 65
  3.1.5 Arrays as Arguments and Results ................................. 65
  3.1.6 Function Pointers ......................................................... 67
  3.1.7 Pointers as Aliases ....................................................... 70
  3.1.8 Pointers and Strings ..................................................... 72
  3.1.9 Direct Memory Access ................................................ 74
3.2 Structures ........................................................................... 75
  3.2.1 Grouping Together Data Elements ............................... 75
  3.2.2 Returning Multiple Data Elements from a Function ........ 76
  3.2.3 Mapping the Organization of Data ............................... 76
  3.2.4 Programming in an Object-Oriented Fashion ............... 78
3.3 Unions .............................................................................. 80
  3.3.1 Using Storage Efficiently ............................................ 80
  3.3.2 Implementing Polymorphism ...................................... 81
  3.3.3 Accessing Different Internal Representations .............. 82
3.4 Dynamic Memory Allocation ............................................. 84
  3.4.1 Managing Free Memory .............................................. 87
  3.4.2 Structures with Dynamically Allocated Arrays ............ 89
3.5 typedef Declarations ....................................................... 91
Further Reading ................................................................. 93

4 C Data Structures ............................................................. 95
4.1 Vectors ........................................................................ 96
4.2 Matrices and Tables ...................................................... 101
4.3 Stacks ........................................................................ 105
4.4 Queues ....................................................................... 107
4.5 Maps .......................................................................... 111
4.5.1 Hash Tables ............................................................ 113
4.6 Sets ........................................................................... 116
4.7 Linked Lists ............................................................... 117
4.8 Trees .......................................................................... 125
4.9 Graphs ........................................................................ 131
4.9.1 Node Storage .......................................................... 131
4.9.2 Edge Representation ................................................ 134
4.9.3 Edge Storage ........................................................... 137
4.9.4 Graph Properties ...................................................... 139
4.9.5 Hidden Structures .................................................... 139
4.9.6 Other Representations .............................................. 140
Further Reading ................................................................. 140

5 Advanced Control Flow ................................................... 143
5.1 Recursion .................................................................... 143
5.2 Exceptions ................................................................. 150
5.3 Parallelism ................................................................. 154
5.3.1 Hardware and Software Parallelism ......................... 154
5.3.2 Control Models ....................................................... 156
5.3.3 Thread Implementations .......................................... 162
5.4 Signals ....................................................................... 165
5.5 Nonlocal Jumps .......................................................... 169
5.6 Macro Substitution ...................................................... 172
Further Reading ............................................................... 177
6 Tackling Large Projects ............................................... 179

6.1 Design and Implementation Techniques .............................. 179
6.2 Project Organization ............................................... 181
6.3 The Build Process and Makefiles .................................... 189
6.4 Configuration ...................................................... 197
6.5 Revision Control ................................................... 202
6.6 Project-Specific Tools .............................................. 210
6.7 Testing ........................................................... 215
Further Reading ........................................................ 224

7 Coding Standards and Conventions ................................. 225

7.1 File Names and Organization ....................................... 225
7.2 Indentation ........................................................ 228
7.3 Formatting ........................................................ 230
7.4 Naming Conventions ................................................ 234
7.5 Programming Practices ............................................. 237
7.6 Process Standards .................................................. 239
Further Reading ........................................................ 240

8 Documentation ........................................................ 241

8.1 Documentation Types ............................................... 241
8.2 Reading Documentation .............................................. 243
8.3 Documentation Problems .......................................... 254
8.4 Additional Documentation Sources .................................. 256
8.5 Common Open-Source Documentation Formats ................. 260
Further Reading ........................................................ 266

9 Architecture ............................................................ 267

9.1 System Structures .................................................. 268
  9.1.1 Centralized Repository and Distributed Approaches ............. 268
  9.1.2 Data-Flow Architectures .................................... 273
  9.1.3 Object-Oriented Structures ................................... 275
  9.1.4 Layered Architectures ........................................ 279
9.1.5 Hierarchies .................................................. 282
9.1.6 Slicing ...................................................... 283
9.2 Control Models .................................................... 285
  9.2.1 Event-Driven Systems ........................................ 285
  9.2.2 System Manager ............................................. 289
  9.2.3 State Transition .............................................. 291
9.3 Element Packaging ................................................. 292
  9.3.1 Modules ..................................................... 293
  9.3.2 Namespaces ................................................. 296
  9.3.3 Objects ...................................................... 300
  9.3.4 Generic Implementations ................................. 313
  9.3.5 Abstract Data Types ......................................... 318
  9.3.6 Libraries .................................................... 319
  9.3.7 Processes and Filters ......................................... 323
  9.3.8 Components ................................................ 325
  9.3.9 Data Repositories ............................................ 325
9.4 Architecture Reuse .............................................. 328
  9.4.1 Frameworks ................................................ 329
  9.4.2 Code Wizards ................................................ 330
  9.4.3 Design Patterns .............................................. 331
  9.4.4 Domain-Specific Architectures ........................... 333
Further Reading ....................................................... 337

10 Code-Reading Tools ................................................. 339
  10.1 Regular Expressions ............................................. 340
  10.2 The Editor as a Code Browser ................................ 343
  10.3 Code Searching with grep ...................................... 346
  10.4 Locating File Differences ...................................... 355
  10.5 Roll Your Own Tool ............................................ 357
  10.6 The Compiler as a Code-Reading Tool .......................... 360
  10.7 Code Browsers and Beautifiers ............................... 365
  10.8 Runtime Tools .................................................. 370
  10.9 Nonsoftware Tools .............................................. 375
Tool Availability and Further Reading ............................... 376
Contents

11 A Complete Example ................................................ 379
  11.1 Overview .......................................................379
  11.2 Attack Plan .....................................................380
  11.3 Code Reuse .....................................................382
  11.4 Testing and Debugging ...........................................388
  11.5 Documentation ..................................................396
  11.6 Observations ....................................................397

A Outline of the Code Provided ........................................ 399

B Source Code Credits .................................................403

C Referenced Source Files ............................................. 405

D Source Code Licenses ................................................413
  D.1 ACE ............................................................413
  D.2 Apache ........................................................415
  D.3 ArgouML .......................................................416
  D.4 DemoGL .......................................................416
  D.5 hsqldb .........................................................417
  D.6 NetBSD .........................................................418
  D.7 OpenCL .........................................................418
  D.8 Perl ............................................................419
  D.9 qtchat ........................................................422
  D.10 socket ........................................................422
  D.11 vcf ............................................................422
  D.12 X Window System ..............................................423

E Maxims for Reading Code ........................................... 425

Bibliography ........................................................ 445

Index ............................................................... 459

Author Index ........................................................491
We’re programmers. Our job (and in many cases our passion) is to make things happen by writing code. We don’t meet our user’s requirements with acres of diagrams, with detailed project schedules, with four-foot-high piles of design documentation. These are all wishes—expressions of what we’d like to be true. No, we deliver by writing code: code is reality.

So that’s what we’re taught. Seems reasonable. Our job is to write code, so we need to learn how to write code. College courses teach us to to write programs. Training courses tell us how to code to new libraries and APIs. And that’s one of the biggest tragedies in the industry.

Because the way to learn to write great code is by reading code. Lots of code. High-quality code, low-quality code. Code in assembler, code in Haskell. Code written by strangers ten thousand miles away, and code written by ourselves last week. Because unless we do that, we’re continually reinventing what has already been done, repeating both the successes and mistakes of the past.

I wonder how many great novelists have never read someone else’s work, how many great painters never studied another’s brush strokes, how many skilled surgeons never learned by looking over a colleague’s shoulder, how many 767 captains didn’t first spend time in the copilot’s seat watching how it’s really done.

And yet that’s what we expect programmers to do. “This week’s assignment is to write. . . .” We teach developers the rules of syntax and construction, and then we expect them to be able to write the software equivalent of a great novel.

The irony is that there’s never been a better time to read code. Thanks to the huge contributions of the open-source community, we now have gigabytes of source code floating around the ’net just waiting to be read. Choose any language, and you’ll be able to find source code. Select a problem domain, and there’ll be source code. Pick a level, from microcode up to high-level business functions, and you’ll be able to look at a wide body of source code.
Code reading is fun. I love to read others’ code. I read it to learn tricks and to study traps. Sometimes I come across small but precious gems. I still remember the pleasure I got when I came across a binary-to-octal conversion routine in PDP-11 assembler that managed to output the six octal digits in a tight loop with no loop counter.

I sometimes read code for the narrative, like a book you’d pick up at an airport before a long flight. I expect to be entertained by clever plotting and unexpected symmetries. Jame Clark’s gpic program (part of his GNU groff package) is a wonderful example of this kind of code. It implements something that’s apparently very complex (a declarative, device-independent picture-drawing language) in a compact and elegant structure. I came away feeling inspired to try to structure my own code as tidily.

Sometimes I read code more critically. This is slower going. While I’m reading, I’m asking myself questions such as “Why is this written this way?” or “What in the author’s background would lead her to this choice?” Often I’m in this mode because I’m reviewing code for problems. I’m looking for patterns and clues that might give me pointers. If I see that the author failed to take a lock on a shared data structure in one part of the code, I might suspect that the same might hold elsewhere and then wonder if that mistake could account for the problem I’m seeing. I also use the incongruities I find as a double check on my understanding; often I find what I think is a problem, but on closer examination it turns out to be perfectly good code. Thus I learn something.

In fact, code reading is one of the most effective ways to eliminate problems in programs. Robert Glass, one of this book’s reviewers, says, “by using (code) inspections properly, more than 90 percent of the errors can be removed from a software product before its first test." In the same article he cites research that shows “Code-focused inspectors were finding 90 percent more errors than process-focused inspectors.” Interestingly, while reading the code snippets quoted in this book I came across a couple of bugs and a couple of dubious coding practices. These are problems in code that’s running at tens of thousands of sites worldwide. None were critical in nature, but the exercise shows that there’s always room to improve the code we write. Code-reading skills clearly have a great practical benefit, something you already know if you’ve ever been in a code review with folks who clearly don’t know how to read code.

And then there’s maintenance, the ugly cousin of software development. There are no accurate statistics, but most researchers agree that more than half of the time we spend on software is used looking at existing code: adding new functionality, fixing bugs, integrating it into new environments, and so on. Code-reading skills are crucial. There’s a bug in a 100,000-line program, and you’ve got an hour to find it. How do

1http://www.stickyminds.com/se/S2587.asp
you start? How do you know what you’re looking at? And how can you assess the impact of a change you’re thinking of making?

For all these reasons, and many more, I like this book. At its heart it is pragmatic. Rather than taking an abstract, academic approach, it instead focuses on the code itself. It analyzes hundreds of code fragments, pointing out tricks, traps and (as importantly) idioms. It talks about code in its environment and discusses how that environment affects the code. It highlights the important tools of the code reader’s trade, from common tools such as `grep` and `find` to the more exotic. And it stresses the importance of tool building: write code to help you read code. And, being pragmatic, it comes with all the code it discusses, conveniently cross-referenced on a CD-ROM.

This book should be included in every programming course and should be on every developer’s bookshelf. If as a community we pay more attention to the art of code reading we’ll save ourselves both time and pain. We’ll save our industry money. And we’ll have more fun while we’re doing it.

Dave Thomas
The Pragmatic Programmers, LLC
http://www.pragmaticprogrammer.com
This page intentionally left blank
Preface

**What do we ever get nowadays from reading to equal the excitement and the revelation in those first fourteen years?**

—Graham Greene

The reading of code is likely to be one of the most common activities of a computing professional, yet it is seldom taught as a subject or formally used as a method for learning how to design and program.

One reason for this sad situation originally may have been the lack of real-world or high-quality code to read. Companies often protect source code as a trade secret and rarely allow others to read, comment on, experiment with, and learn from it. In the few cases where important proprietary code was allowed out of a company’s closet, it spurred enormous interest and creative advancements. As an example, a generation of programmers benefited from John Lions’s *Commentary on the Unix Operating System* that listed and annotated the complete source code of the sixth-edition Unix kernel. Although Lions’s book was originally written under a grant from AT&T for use in an operating system course and was not available to the general public, copies of it circulated for years as bootleg nth-generation photocopies.

In the last few years, however, the popularity of open-source software has provided us with a large body of code we can all freely read. Some of the most popular software systems used today, such as the *Apache* Web server, the Perl language, the GNU/Linux operating system, the BIND domain name server, and the *sendmail* mail-transfer agent are in fact available in open-source form. I was thus fortunate to be able to use open-source software such as the above to write this book as a primer and reader for software code. My goal was to provide background knowledge and techniques for reading code written by others. By using real-life examples taken out of working, open-source projects, I tried to cover most concepts related to code that are likely to appear
before a software developer’s eyes, including programming constructs, data types, data structures, control flow, project organization, coding standards, documentation, and architectures. A companion title to this book will cover interfacing and application-oriented code, including the issues of internationalization and portability, the elements of commonly used libraries and operating systems, low-level code, domain-specific and declarative languages, scripting languages, and mixed language systems.

This book is—as far as I know—the first one to exclusively deal with code reading as a distinct activity, one worthy on its own. As such I am sure that there will be inevitable shortcomings, better ways some of its contents could have been treated, and important material I have missed. I firmly believe that the reading of code should be both properly taught and used as a method for improving one’s programming abilities. I therefore hope this book will spur interest to include code-reading courses, activities, and exercises into the computing education curriculum so that in a few years our students will learn from existing open-source systems, just as their peers studying a language learn from the great literature.

**Supplementary Material**

Many of the source code examples provided come from the source distribution of NetBSD. NetBSD is a free, highly portable Unix-like operating system available for many platforms, from 64-bit AlphaServers to handheld devices. Its clean design and advanced features make it an excellent choice for both production and research environments. I selected NetBSD over other similarly admirable and very popular free Unix-like systems (such as GNU/Linux, FreeBSD, and OpenBSD) because the primary goal of the NetBSD project is to emphasize correct design and well-written code, thus making it a superb choice for providing example source code. According to its developers, some systems seem to have the philosophy of “if it works, it’s right,” whereas NetBSD could be described as “it doesn’t work unless it’s right.” In addition, some other NetBSD goals fit particularly well with the objectives of this book. Specifically, the NetBSD project avoids encumbering licenses, provides a portable system running on many hardware platforms, interoperates well with other systems, and conforms to open systems standards as much as is practical. The code used in this book is a (now historic) `export-19980407` snapshot. A few examples refer to errors I found in the code; as the NetBSD code continuously evolves, presenting examples from a more recent version would mean risking that those realistic gems would have been corrected.

I chose the rest of the systems I used in the book’s examples for similar reasons: code quality, structure, design, utility, popularity, and a license that would not make my
publisher nervous. I strived to balance the selection of languages, actively looking for suitable Java and C++ code. However, where similar concepts could be demonstrated using different languages I chose to use C as the least common denominator.

I sometimes used real code examples to illustrate unsafe, nonportable, unreadable, or otherwise condemnable coding practices. I appreciate that I can be accused of disparaging code that was contributed by its authors in good faith to further the open-source movement and to be improved upon rather than merely criticized. I sincerely apologize in advance if my comments cause any offense to a source code author. In defense I argue that in most cases the comments do not target the particular code excerpt, but rather use it to illustrate a practice that should be avoided. Often the code I am using as a counterexample is a lame duck, as it was written at a time when technological and other restrictions justified the particular coding practice, or the particular practice is criticized out of the context. In any case, I hope that the comments will be received with good humor, and I openly admit that my own code contains similar, and probably worse, misdeeds.

Acknowledgments

A number of people generously contributed advice, comments, and their time helping to make this book a reality. Addison-Wesley assembled what I consider a dream team of reviewers: Paul C. Clements, Robert L. Glass, Scott D. Meyers, Guy Steele, Dave Thomas, and John Vlissides graciously read the manuscript in a form much rougher than the one you hold in your hands and shared their experience and wisdom through thoughtful, perceptive, and often eye-opening reviews. In addition, Eliza Fragaki, Georgios Chrisoloras, Kleanthis Georgaris, Isidor Kouvelas, and Lorenzo Vicisano read parts of the manuscript in an informal capacity and contributed many useful comments and suggestions. I was also lucky to get advice on the mechanics of the production process from Bill Cheswick, Christine Hogan, Tom Limoncelli, and Antonis Tsolomitis. Furthermore, George Gousios suggested the use of Tomcat as Java open-source software material and explained to me details of its operation, pointed me toward the ant build tool, and clarified issues concerning the use of the DocBook documentation format. Stephen Ma solved the mystery of how vnode pointers end up at the operating system device driver level (see Section 9.1.4). Spyros Oikonomopoulos provided me with an overview of the reverse engineering capabilities of UML-based modeling tools. Panagiotis Petropoulos updated the book references. Konstantina Vassilopoulou advised me on readability aspects of the annotated code listings. Ioanna Grinia, Vasilis Karakoidas, Nikos Korfiatis, Vasiliki
Tangalaki, and George M. Zouganelis contributed their views on the book’s layout. Athan Tolis located the epigram for Chapter 5 in the London Science Museum library. Elizabeth Ryan and the folks at ITC patiently designed and redesigned the book until we could all agree it had the right look.

My editors, Ross Venables and Mike Hendrickson at Addison-Wesley, handled the book’s production with remarkable effectiveness. In the summer of 2001, a week after we first established contact, Ross was already sending the manuscript proposal for review; working with a seven-hour time zone difference, I would typically find any issues I raised near the end of my working day solved when I opened my email in the morning. Their incredible efficiency in securing reviewers, answering my often naive questions, dealing with the book’s contractual aspects, and coordinating the complex production process was paramount in bringing this project to fruition. Later on, Elizabeth Ryan expertly synchronized the Addision-Wesley production team; Chrysta Meadowbrooke diligently copy-edited my (often rough) manuscript, demonstrating an admirable understanding of its technical content; ITC handled the demanding composition task; and Jennifer Lundberg patiently introduced me to the mysteries of book marketing.

The vast majority of the examples used in this book are parts of existing open-source projects. The use of real-life code allowed me to present the type of code that one is likely to encounter rather than simplified toy programs. I therefore wish to thank all the contributors of the open-source material I have used for sharing their work with the programming community. The contributor names of code that appears in the book, when listed in the corresponding source code file, appear in Appendix B.
Basic Programming Elements

What we observe is not nature itself, but nature exposed to our method of questioning.

—Werner Heisenberg

Code reading is in many cases a bottom-up activity. In this chapter we review the basic code elements that comprise programs and outline how to read and reason about them. In Section 2.1 we dissect a simple program to demonstrate the type of reasoning necessary for code reading. We will also have the first opportunity to identify common traps and pitfalls that we should watch for when reading or writing code, as well as idioms that can be useful for understanding its meaning. Sections 2.2 and onward build on our understanding by examining the functions, control structures, and expressions that make up a program. Again, we will reason about a specific program while at the same time examining the (common) control constructs of C, C++, Java, and Perl. Our first two complete examples are C programs mainly because realistic self-standing Java or C++ programs are orders of magnitude larger. However, most of the concepts and structures we introduce here apply to programs written in any of the languages derived from C such as C++, C#, Java, Perl, and PHP. We end this chapter with a section detailing how to reason about a program’s flow of control at an abstract level, extracting semantic meaning out of its code elements.

2.1 A Complete Program

A very simple yet useful program available on Unix systems is `echo`, which prints its arguments on the standard output (typically the screen). It is often used to display
information to the user as in:

```bash
echo "Cool! Let's get to it..."
```

in the NetBSD upgrade script.\(^1\) Figure 2.1 contains the complete source code of `echo`.\(^2\)

As you can see, more than half of the program code consists of legal and administrative information such as copyrights, licensing information, and program version identifiers. The provision of such information, together with a summary of the specific program or module functionality, is a common characteristic in large, organized systems. When reusing source code from open-source initiatives, pay attention to the licensing requirements imposed by the copyright notice (Figure 2.1:1).

C and C++ programs need to include header files (Figure 2.1:2) in order to correctly use library functions. The library documentation typically lists the header files needed for each function. The use of library functions without the proper header files often generates only warnings from the C compiler yet can cause programs to fail at runtime. Therefore, a part of your arsenal of code-reading procedures will be to run the code through the compiler looking for warning messages (see Section 10.6).

Standard C, C++, and Java programs begin their execution from the function (method in Java) called `main` (Figure 2.1:3). When examining a program for the first time `main` can be a good starting point. Keep in mind that some operating environments such as Microsoft Windows, Java applet and servlet hosts, palmtop PCs, and embedded systems may use another function as the program’s entry point, for example, `WinMain` or `init`.

In C/C++ programs two arguments of the `main` function (customarily named `argc` and `argv`) are used to pass information from the operating system to the program about the specified command-line arguments. The `argc` variable contains the number of program arguments, while `argv` is an array of strings containing all the actual arguments (including the name of the program in position 0). The `argv` array is terminated with a NULL element, allowing two different ways to process arguments: either by counting based on `argc` or by going through `argv` and comparing each value against NULL. In Java programs you will find the `argv String` array and its `length` method used for the same purpose, while in Perl code the equivalent constructs you will see are the `@ARGV` array and the `$#ARGV` scalar.

---

\(^1\) `netbsdsrc/distrib/miniroot/upgrade.sh:98`

\(^2\) `netbsdsrc/bin/echo/echo.c:3–80`
Figure 2.1 The Unix echo program.
The declaration of argc and argv in our example (Figure 2.1:4) is somewhat unusual. The typical C/C++ definition of main is:

```c
int main(int argc, char **argv)
```

while the corresponding Java class method definition is:

```java
public static void main(String args[])
```

The definition in Figure 2.1:4 is using the old-style (pre-ANSI C) syntax of C, also known as K&R C. You may come across such function definitions in older programs; keep in mind that there are subtle differences in the ways arguments are passed and the checks that a compiler will make depending on the style of the function definition.

When examining command-line programs you will find arguments processed by using either handcrafted code or, in POSIX environments, the `getopt` function. Java programs may be using the GNU `gnu.getopt` package for the same purpose.

The standard definition of the `echo` command is not compatible with the `getopt` behavior; the single `-n` argument specifying that the output is not to be terminated with a newline is therefore processed by handcrafted code (Figure 2.1:6). The comparison starts by advancing argv to the first argument of `echo` (remember that position 0 contains the program name) and verifying that such an argument exists. Only then is `strcmp` called to compare the argument against `-n`. The sequence of a check to see if the argument is valid, followed by a use of that argument, combined with using the Boolean AND (`&&`) operator, is a common idiom. It works because the `&&` operator will not evaluate its righthand side operand if its lefthand side evaluates to false. Calling `strcmp` or any other string function and passing it a NULL value instead of a pointer to actual character data will cause a program to crash in many operating environments.

Note the nonintuitive return value of `strcmp` when it is used for comparing two strings for equality. When the strings compare equal it returns 0, the C value of false. For this reason you will see that many C programs define a macro `STREQ` to return true when two strings compare equal, often optimizing the comparison by comparing the first two characters on the fly:

```c
#define STREQ(a, b) (*(a) == *(b) && strcmp((a), (b)) == 0)
```

---

3 `netbsdsrc/usr.bin/elf2aout/elf2aout.c:72–73`
4 `jt4/catalina/src/share/org/apache/catalina/startup/Catalina.java:161`
5 `http://www.gnu.org/software/java/packages.html`
6 `netbsdsrc/usr.bin/file/ascmagic.c:45`
Fortunately the behavior of the Java equals method results in a more intuitive reading:

```java
if (isConfig) {
    configFile = args[i];
    isConfig = false;
} else if (args[i].equals("-config")) {
    isConfig = true;
} else if (args[i].equals("-debug")) {
    debug = true;
} else if (args[i].equals("-nonaming")) {

The above sequence also introduces an alternative way of formatting the indentation of cascading if statements to express a selection. Read a cascading if-else if-else sequence as a selection of mutually exclusive choices.

An important aspect of our if statement that checks for the -n flag is that nflag will always be assigned a value: 0 or 1. nflag is not given a value when it is defined (Figure 2.1:5). Therefore, until it gets assigned, its value is undefined: it is the number that happened to be in the memory place it was stored. Using uninitialized variables is a common cause of problems. When inspecting code, always check that all program control paths will correctly initialize variables before these are used. Some compilers may detect some of these errors, but you should not rely on it.

The part of the program that loops over all remaining arguments and prints them separated by a space character is relatively straightforward. A subtle pitfall is avoided by using printf with a string-formatting specification to print each argument (Figure 2.1:7). The printf function will always print its first argument, the format specification. You might therefore find a sequence that directly prints string variables through the format specification argument:

```java
printf(version);
```

Printing arbitrary strings by passing them as the format specification to printf will produce incorrect results when these strings contain conversion specifications (for example, an SCCS revision control identifier containing the % character in the case above).

---

7jt4/catalina/src/share/org/apache/catalina/startup/CatalinaService.java:136–143
8netbsdsrc/sys/arch/mvme68k/mvme68k/machdep.c:347
Even so, the use of `printf` and `putchar` is not entirely correct. Note how the return value of `printf` is cast to `void`. `printf` will return the number of characters that were actually printed; the cast to `void` is intended to inform us that this result is intentionally ignored. Similarly, `putchar` will return `EOF` if it fails to write the character. All output functions—in particular when the program’s standard output is redirected to a file—can fail for a number of reasons:

- The device where the output is stored can run out of free space.
- The user’s quota of space on the device can be exhausted.
- The process may attempt to write a file that exceeds the process’s or the system’s maximum file size.
- A hardware error can occur on the output device.
- The file descriptor or stream associated with the standard output may not be valid for writing.

Not checking the result of output operations can cause a program to silently fail, losing output without any warning. Checking the result of each and every output operation can be inconvenient. A practical compromise you may encounter is to check for errors on the standard output stream before the program terminates. This can be done in Java programs by using the `checkError` method (we have yet to see this used in practice on the standard output stream; even some JDK programs will fail without an error when running out of space on their output device); in C++ programs by using a stream’s `fail`, `good`, or `bad` methods; and in C code by using the `ferror` function:

```c
if (ferror(stdout))
    err(1, "stdout");
```

After terminating its output with a newline, `echo` calls `exit` to terminate the program indicating success (0). You will also often find the same result obtained by returning 0 from the function `main`.

**Exercise 2.1** Experiment to find out how your C, C++, and Java compilers deal with uninitialized variables. Outline your results and propose an inspection procedure for locating uninitialized variables.

**Exercise 2.2** (Suggested by Dave Thomas.) Why can’t the `echo` program use the `getopt` function?

---

9netbsdsrc/bin/cat/cat.c:213–214
Exercise 2.3  Discuss the advantages and disadvantages of defining a macro like STREQ. Consider how the C compiler could optimize strcmp calls.

Exercise 2.4  Look in your environment or on the book’s CD-ROM for programs that do not verify the result of library calls. Propose practical fixes.

Exercise 2.5  Sometimes executing a program can be a more expedient way to understand an aspect of its functionality than reading its source code. Devise a testing procedure or framework to examine how programs behave on write errors on their standard output. Try it on a number of character-based Java and C programs (such as the command-line version of your compiler) and report your results.

Exercise 2.6  Identify the header files that are needed for using the library functions sscanf, qsort, strchr, setjmp, adjacent_find, open, FormatMessage, and XtOwnSelection. The last three functions are operating environment–specific and may not exist in your environment.

2.2 Functions and Global Variables

The program expand processes the files named as its arguments (or its standard input if no file arguments are specified) by expanding hard tab characters (\t, ASCII character 9) to a number of spaces. The default behavior is to set tab stops every eight characters; this can be overridden by a comma or space-separated numeric list specified using the -t option. An interesting aspect of the program’s implementation, and the reason we are examining it, is that it uses all of the control flow statements available in the C family of languages. Figure 2.2 contains the variable and function declarations of expand, figure 2.3 contains the main code body, and figure 2.5 (in Section 2.5) contains the two supplementary functions used.

When examining a nontrivial program, it is useful to first identify its major constituent parts. In our case, these are the global variables (Figure 2.2:1) and the functions main (Figure 2.3), getstops (see Figure 2.5:1), and usage (see Figure 2.5:8).

The integer variable nstops and the array of integers tabstops are declared as global variables, outside the scope of function blocks. They are therefore visible to all functions in the file we are examining.

The three function declarations that follow (Figure 2.2:2) declare functions that will appear later within the file. Since some of these functions are used before they are defined, in C/C++ programs the declarations allow the compiler to verify the arguments.
Basic Programming Elements

```c
#include <sys/cdefs.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <unistd.h>

int nstops;
int tabstops[100];

static void getstops(char *);

int main(int, char *);

static void usage (void);
```

Figure 2.2 Expanding tab stops (declarations).

passed to the function and their return values and generate correct corresponding code. When no forward declarations are given, the C compiler will make assumptions about the function return type and the arguments when the function is first used; C++ compilers will flag such cases as errors. If the following function definition does not match these assumptions, the compiler will issue a warning or error message. However, if the wrong declaration is supplied for a function defined in another file, the program may compile without a problem and fail at runtime.

Notice how the two functions are declared as `static` while the variables are not. This means that the two functions are visible only within the file, while the variables are potentially visible to all files comprising the program. Since `expand` consists only of a single file, this distinction is not important in our case. Most linkers that combine compiled C files are rather primitive; variables that are visible to all program files (that is, not declared as `static`) can interact in surprising ways with variables with the same name defined in other files. It is therefore a good practice when inspecting code to ensure that all variables needed only in a single file are declared as `static`.

Let us now look at the functions comprising `expand`. To understand what a function (or method) is doing you can employ one of the following strategies.

- Guess, based on the function name.
- Read the comment at the beginning of the function.
- Examine how the function is used.
- Read the code in the function body.
- Consult external program documentation.

In our case we can safely guess that the function `usage` will display program usage information and then exit; many command-line programs have a function with the same name and functionality. When you examine a large body of code, you
2.2 Functions and Global Variables

Figure 2.3 Expanding tab stops (main part).
Basic Programming Elements

will gradually pick up names and naming conventions for variables and functions. These will help you correctly guess what they do. However, you should always be prepared to revise your initial guesses following new evidence that your code reading will inevitably unravel. In addition, when modifying code based on guesswork, you should plan the process that will verify your initial hypotheses. This process can involve checks by the compiler, the introduction of assertions, or the execution of appropriate test cases.

The role of `getstops` is more difficult to understand. There is no comment, the code in the function body is not trivial, and its name can be interpreted in different ways. Noting that it is used in a single part of the program (Figure 2.3:3) can help us further. The program part where `getstops` is used is the part responsible for processing the program’s options (Figure 2.3:2). We can therefore safely (and correctly in our case) assume that `getstops` will process the tab stop specification option. This form of gradual understanding is common when reading code; understanding one part of the code can make others fall into place. Based on this form of gradual understanding you can employ a strategy for understanding difficult code similar to the one often used to combine the pieces of a jigsaw puzzle: start with the easy parts.

**Exercise 2.7** Examine the visibility of functions and variables in programs in your environment. Can it be improved (made more conservative)?

**Exercise 2.8** Pick some functions or methods from the book’s CD-ROM or from your environment and determine their role using the strategies we outlined. Try to minimize the time you spend on each function or method. Order the strategies by their success rate.

### 2.3 while Loops, Conditions, and Blocks

We can now examine how options are processed. Although `expand` accepts only a single option, it uses the Unix library function `getopt` to process options. A summarized version of the Unix on-line documentation for the `getopt` function appears in Figure 2.4. Most development environments provide on-line documentation for library functions, classes, and methods. On Unix systems you can use the `man` command and on Windows the Microsoft Developer Network Library (MSDN), while the Java API is documented in HTML format as part of the Sun JDK. Make it a habit to read the documentation of library elements you encounter; it will enhance both your code-reading and code-writing skills.

---

13http://msdn.microsoft.com
NAME
getopt – get option character from command line argument list

SYNOPSIS
#include <unistd.h>
extern char ∗optarg;
extern int optind;
extern int optopt;
extern int opterr;
extern int optreset;
int getopt(int argc, char ∗const ∗argv, const char ∗optstring)

DESCRIPTION
The getopt() function incrementally parses a command line argument list argv and returns the next known option character. An option character is known if it has been specified in the string of accepted option characters, optstring.

The option string optstring may contain the following elements: individual characters, and characters followed by a colon to indicate an option argument is to follow. For example, an option string "x" recognizes an option "–x", and an option string "x:" recognizes an option and argument "–x argument". It does not matter to getopt() if a following argument has leading white space.

On return from getopt(), optarg points to an option argument, if it is anticipated, and the variable optind contains the index to the next argv argument for a subsequent call to getopt(). The variable optopt saves the last known option character returned by getopt().

The variable opterr and optind are both initialized to 1. The optind variable may be set to another value before a set of calls to getopt() in order to skip over more or less argv entries.

The getopt() function returns –1 when the argument list is exhausted, or a non-recognized option is encountered. The interpretation of options in the argument list may be cancelled by the option ‘--’ (double dash) which causes getopt() to signal the end of argument processing and returns –1. When all options have been processed (i.e., up to the first non-option argument), getopt() returns –1.

DIAGNOSTICS
If the getopt() function encounters a character not found in the string optstring or detects a missing option argument it writes an error message to stderr and returns ‘?’. Setting opterr to a zero will disable these error messages. If optstring has a leading ‘:’ then a missing option argument causes a ‘:’ to be returned in addition to suppressing any error messages.

Option arguments are allowed to begin with “–”; this is reasonable but reduces the amount of error checking possible.

HISTORY
The getopt() function appeared 4.3BSD.

BUGS
The getopt() function was once specified to return EOF instead of –1. This was changed by POSIX 1003.2-92 to decouple getopt() from <stdio.h>.

4.3 Berkeley Distribution April 19, 1994 1
Based on our understanding of `getopt`, we can now examine the relevant code (Figure 2.3:2). The option string passed to `getopt` allows for a single option `-t`, which is to be followed by an argument. `getopt` is used as a condition expression in a `while` statement. A `while` statement will repeatedly execute its body as long as the condition specified in the parentheses is true (in C/C++, if it evaluates to a value other than 0). In our case the condition for the `while` loop calls `getopt`, assigns its result to `c`, and compares it with `-1`, which is the value used to signify that all options have been processed. To perform these operations in a single expression, the code uses the fact that in the C language family assignment is performed by an operator (`=`), that is, assignment expressions have a value. The value of an assignment expression is the value stored in the left operand (the variable `c` in our case) after the assignment has taken place. Many programs will call a function, assign its return value to a variable, and compare the result against some special-case value in a single expression. The following typical example assigns the result of `readLine` to `line` and compares it against `null` (which signifies that the end of the stream was reached).

```java
if ((line = input.readLine()) == null) [...] return errors;
```

It is imperative to enclose the assignment within parentheses, as is the case in the two examples we have examined. As the comparison operators typically used in conjunction with assignments bind more tightly than the assignment, the following expression

```java
c = getopt (argc, argv, "t:" ) != -1
```

will evaluate as

```java
c = (getopt (argc, argv, "t:" ) != -1)
```

thus assigning to `c` the result of comparing the return value of `getopt` against `-1` rather than the `getopt` return value. In addition, the variable used for assigning the result of the function call should be able to hold both the normal function return values and any exceptional values indicating an error. Thus, typically, functions that return characters such as `getopt` and `getc` and also can return an error value such as `-1` or

---

14cocoon/src/java/org/apache/cocoon/components/language/programming/java/Javac.java:106–112
EOF have their results stored in an integer variable, *not a character variable*, to hold the superset of all characters *and* the exceptional value (Figure 2.3:7). The following is another typical use of the same construct, which copies characters from the file stream *pf* to the file stream *active* until the *pf* end of file is reached.\(^{15}\)

```c
while ((c = getc(pf)) != EOF)
   putc(c, active);
```

The body of a *while* statement can be either a single statement or a block: one or more statements enclosed in braces. The same is true for all statements that control the program flow, namely, *if*, *do*, *for*, and *switch*. Programs typically indent lines to show the statements that form part of the control statement. However, the indentation is only a visual clue for the human program reader; if no braces are given, the control will affect only the single statement that follows the respective control statement, regardless of the indentation. As an example, the following code does not do what is suggested by its indentation.\(^{16}\)

```c
for (ntp = nettab; ntp != NULL; ntp = ntp->next) {
    if (ntp->status == MASTER)
        rmnetmachs(ntp);
    ntp->status = NOMASTER;
}
```

The line *ntp-*status = NOMASTER; will be executed for every iteration of the *for* loop and not just when the *if* condition is true.

**Exercise 2.9** Discover how the editor you are using can identify matching braces and parentheses. If it cannot, consider switching to another editor.

**Exercise 2.10** The source code of *expand* contains some superfluous braces. Identify them. Examine all control structures that do not use braces and mark the statements that will get executed.

**Exercise 2.11** Verify that the indentation of *expand* matches the control flow. Do the same for programs in your environment.

**Exercise 2.12** The Perl language mandates the use of braces for all its control structures. Comment on how this affects the readability of Perl programs.

\(^{15}\) netbsdsrc/usr.bin/m4/eval.c:601–602
\(^{16}\) netbsdsrc/usr.sbin/timed/timed/timed.c:564–568
2.4 switch Statements

The normal return values of `getopt` are handled by a `switch` statement. You will find `switch` statements used when a number of discrete integer or character values are being processed. The code to handle each value is preceded by a `case` label. When the value of the expression in the `switch` statement matches the value of one of the `case` labels, the program will start to execute statements from that point onward. If none of the label values match the expression value and a `default` label exists, control will transfer to that point; otherwise, no code within the `switch` block will get executed. Note that additional labels encountered after transferring execution control to a label will not terminate the execution of statements within the `switch` block; to stop processing code within the `switch` block and continue with statements outside it, a `break` statement must be executed. You will often see this feature used to group case labels together, merging common code elements. In our case when `getopt` returns ‘t’, the statements that handle -t are executed, with `break` causing a transfer of execution control immediately after the closing brace of the `switch` block (Figure 2.3:4). In addition, we can see that the code for the `default` switch label and the error return value ‘?’ is common since the two corresponding labels are grouped together.

When the code for a given `case` or `default` label does not end with a statement that transfers control out of the `switch` block (such as `break`, `return`, or `continue`), the program will continue to execute the statements following the next label. When examining code, look out for this error. In rare cases the programmer might actually want this behavior. To alert maintainers to that fact, it is common to mark these places with a comment, such as `FALLTHROUGH`, as in the following example.17

```c
    case 'a':
        fts_options |= FTS_SEEDOT;
        /* FALLTHROUGH */
    case 'A':
        f_listdot = 1;
        break;
```

The code above comes from the option processing of the Unix `ls` command, which lists files in a directory. The option `-A` will include in the list files starting with a dot (which are, by convention, hidden), while the option `-a` modifies this behavior by adding to the list the two directory entries. Programs that automatically verify

---

17 netbsdsrc/bin/ls/ls.c:173–178
source code against common errors, such as the Unix lint command, can use the FALLTHROUGH comment to suppress spurious warnings.

A switch statement lacking a default label will silently ignore unexpected values. Even when one knows that only a fixed set of values will be processed by a switch statement, it is good defensive programming practice to include a default label. Such a default label can catch programming errors that yield unexpected values and alert the program maintainer, as in the following example.\textsuperscript{18}

```
switch (program) {
  case ATQ:
  [...]  
  case BATCH:
    writefile(time(NULL), 'b');
    break;
  default:
    panic("Internal error");
    break;
}
```

In our case the switch statement can handle two getopt return values.

1. ‘t’ is returned to handle the -t option. Optind will point to the argument of -t. The processing is handled by calling the function getstops with the tab specification as its argument.

2. ‘?’ is returned when an unknown option or another error is found by getopt. In that case the usage function will print program usage information and exit the program.

A switch statement is also used as part of the program’s character-processing loop (Figure 2.3:7). Each character is examined and some characters (the tab, the newline, and the backspace) receive special processing.

**Exercise 2.13** The code body of switch statements in the source code collection is formatted differently from the other statements. Express the formatting rule used, and explain its rationale.

**Exercise 2.14** Examine the handling of unexpected values in switch statements in the programs you read. Propose changes to detect errors. Discuss how these changes will affect the robustness of programs in a production environment.

\textsuperscript{18} netbsdsrc/usr.bin/at/at.c:535–561
Exercise 2.15  Is there a tool or a compiler option in your environment for detecting missing break statements in switch code? Use it, and examine the results on some sample programs.

2.5 for Loops

To complete our understanding of how expand processes its command-line options, we now need to examine the getstops function. Although the role of its single cp argument is not obvious from its name, it becomes apparent when we examine how getstops is used. getstops is passed the argument of the -t option, which is a list of tab stops, for example, 4, 8, 16, 24. The strategies outlined for determining the roles of functions (Section 2.2) can also be employed for their arguments. Thus a pattern for reading code slowly emerges. Code reading involves many alternative strategies: bottom-up and top-down examination, the use of heuristics, and review of comments and external documentation should all be tried as the problem dictates.

After setting nstops to 0, getstops enters a for loop. Typically a for loop is specified by an expression to be evaluated before the loop starts, an expression to be evaluated before each iteration to determine if the loop body will be entered, and an expression to be evaluated after the execution of the loop body. for loops are often used to execute a body of code a specific number of times.\(^\text{19}\)

\[\text{for} \ (i = 0; i < \text{len}; i++) \{\]

Loops of this type appear very frequently in programs; learn to read them as “execute the body of code \text{len} times.” On the other hand, any deviation from this style, such as an initial value other than 0 or a comparison operator other than $<$, should alert you to carefully reason about the loop’s behavior. Consider the number of times the loop body is executed in the following examples.

Loop extrknt + 1 times:\(^\text{20}\)

\[\text{for} \ (i = 0; i \leq \text{extrknt}; i++) \]

Loop month - 1 times:\(^\text{21}\)

\[\text{for} \ (i = 1; i < \text{month}; i++) \]

\(^{19}\text{cocoon/src/java/org/apache/cocoon/util/StringUtils.java:85}\)
\(^{20}\text{netbsdsrc/usr.bin/fsplit/fsplit.c:173}\)
\(^{21}\text{netbsdsrc/usr.bin/cal/cal.c:332}\)
Loop nargs times:

for (i = 1; i <= nargs; i++)

Note that the last expression need not be an increment operator. The following line will loop 256 times, decrementing code in the process:

for (code = 255; code >= 0; code--) {

In addition, you will find for statements used to loop over result sets returned by library functions. The following loop is performed for all files in the directory dir.

if ((dd = opendir(dir)) == NULL) 
   return (CC–ERROR);
for (dp = readdir(dd); dp != NULL; dp = readdir(dd)) {

The call to opendir returns a value that can be passed to readdir to sequentially access each directory entry of dir. When there are no more entries in the directory, readdir will return NULL and the loop will terminate.

The three parts of the for specification are expressions and not statements. Therefore, if more than one operation needs to be performed when the loop begins or at the end of each iteration, the expressions cannot be grouped together using braces. You will, however, often find expressions grouped together using the expression-sequencing comma (,) operator.

for (cnt = 1, t = p; cnt <= cnt–orig; ++t, ++cnt) {

The value of two expressions joined with the comma operator is just the value of the second expression. In our case the expressions are evaluated only for their side effects: before the loop starts, cnt will be set to 1 and t to p, and after every loop iteration t and cnt will be incremented by one.

Any expression of a for statement can be omitted. When the second expression is missing, it is taken as true. Many programs use a statement of the form for (;;) to perform an “infinite” loop. Very seldom are such loops really infinite. The following

---

22 netbsdsrc/usr.bin/apply/apply.c:130
23 netbsdsrc/usr.bin/compress/zopen.c:510
24 netbsdsrc/usr.bin/ftp/complete.c:193–198
25 netbsdsrc/usr.bin/vi/vi/vs_smap.c:389
for (; ;) {
    s = (state_t) (*s)();
    quiet = 0;
}

In most cases an “infinite” loop is a way to express a loop whose exit condition(s) cannot be specified at its beginning or its end. These loops are typically exited either by a return statement that exits the function, a break statement that exits the loop body, or a call to exit or a similar function that exits the entire program. C++, C#, and Java programs can also exit such loops through an exception (see Section 5.2).

A quick look through the code of the loop in Figure 2.5 provides us with the possible exit routes.

---

26 netbsdsrc/sbin/init/init.c:540–545
• A bad stop specification will cause the program to terminate with an error message (Figure 2.5:3).
• The end of the tab specification string will break out of the loop.

**Exercise 2.16** The for statement in the C language family is very flexible. Examine the source code provided to create a list of ten different uses.

**Exercise 2.17** Express the examples in this section using while instead of for. Which of the two forms do you find more readable?

**Exercise 2.18** Devise a style guideline specifying when while loops should be used in preference to for loops. Verify the guideline against representative examples from the book’s CD-ROM.

### 2.6 break and continue Statements

A break statement will transfer the execution to the statement after the innermost loop or switch statement (Figure 2.5:7). In most cases you will find break used to exit early out of a loop. A continue statement will continue the iteration of the innermost loop without executing the statements to the end of the loop. A continue statement will reevaluate the conditional expression of while and do loops. In for loops it will evaluate the third expression and then the conditional expression. You will find continue used where a loop body is split to process different cases; each case typically ends with a continue statement to cause the next loop iteration. In the program we are examining, continue is used after processing each different input character class (Figure 2.3:8).

Note when you are reading Perl code that break and continue are correspondingly named last and next.27

```perl
while (<UD>) {
    chomp;
    if ($/0x[\d\w]+\s+\((.*?)\)/ and $wanted eq $1) {
        [...]
        last;
    }
}
```

---

To determine the effect of a `break` statement, start reading the program upward from `break` until you encounter the first `while`, `for`, `do`, or `switch` block that encloses the `break` statement. Locate the first statement after that loop; this will be the place where control will transfer when `break` is executed. Similarly, when examining code that contains a `continue` statement, start reading the program upward from `continue` until you encounter the first `while`, `for`, or `do` loop that encloses the `continue` statement. Locate the last statement of that loop; immediately after it (but not outside the loop) will be the place where control will transfer when `continue` is executed. Note that `continue` ignores `switch` statements and that neither `break` nor `continue` affect the operation of `if` statements.

There are situations where a loop is executed only for the side effects of its controlling expressions. In such cases `continue` is sometimes used as a placeholder instead of the empty statement (expressed by a single semicolon). The following example illustrates such a case.  

```java
for (; *string && isdigit(*string); string++)
    continue;
```

In Java programs `break` and `continue` can be followed by a label identifier. The same identifier, followed by a colon, is also used to label a loop statement. The labeled form of the `continue` statement is then used to skip an iteration of a nested loop; the label identifies the loop statement that the corresponding `continue` will skip. Thus, in the following example, the `continue skip;` statement will skip one iteration of the outermost `for` statement.

```java
skip:
    for ( […] ) {
        if ( ch == limit.charAt(0) ) {
            for (int i = 1 ; i < limlen ; i++) {
                if ( […] )
                    continue skip;
            }
        }
        return ret;
    }
```

---

28 netbsdsrc/usr.bin/error/pi.c:174–175
29 j4/jasper/src/share/org/apache/jasper/compiler/JspReader.java:472–482
Similarly, the labeled form of the `break` statement is used to exit from nested loops; the label identifies the statement that the corresponding `break` will terminate. In some cases a labeled `break` or `continue` statements is used, even when there are no nested loops, to clarify the corresponding loop statement.30

```java
comp : while(prev < length) {
    [...] 
    if (pos >= length || pos == -1) {
        [...] 
        break comp;
    }
}
```

**Exercise 2.19** Locate ten occurrences of `break` and `continue` in the source code provided with the book. For each case indicate the point where execution will transfer after the corresponding statement is executed, and explain why the statement is used. Do not try to understand in full the logic of the code; simply provide an explanation based on the statement’s use pattern.

### 2.7 Character and Boolean Expressions

The body of the `for` loop in the `getstops` function starts with a block of code that can appear cryptic at first sight (Figure 2.5:2). To understand it we need to dissect the expressions that comprise it. The first, the condition in the `while` loop, is comparing `*cp` (the character `cp` points to) against two characters: '0' and '9'. Both comparisons must be true and both of them involve `*cp` combined with a different inequality operator and another expression. Such a test can often be better understood by rewriting the comparisons to bring the value being compared in the middle of the expression and to arrange the other two values in ascending order. This rewriting in our case would yield

```java
while ('0' <= *cp && *cp <= '9')
```

This can then be read as a simple range membership test for a character `c`.

\[
0 \leq c \leq 9
\]
Note that this test assumes that the digit characters are arranged sequentially in ascending order in the underlying character set. While this is true for the digits in all character sets we know, comparisons involving alphabetical characters may yield surprising results in a number of character sets and locales. Consider the following typical example.31

```c
if ('a' <= *s && *s <= 'z')
    *s -= ('a' - 'A');
```

The code attempts to convert lowercase characters to uppercase by subtracting from each character found to be lowercase (as determined by the `if` test) the character set distance from 'a' to 'A'. This fragment will fail to work when there are lowercase characters in character set positions outside the range a...z, when the character set range a...z contains nonlowercase characters, and when the code of each lowercase character is not a fixed distance away from the corresponding uppercase character. Many non-ASCII character sets exhibit at least one of these problems.

The next line in the block (Figure 2.5:2) can also appear daunting. It modifies the variable `i` based on the values of `i` and `*cp` and two constants: 10 and '0' while at the same time incrementing `cp`. The variable names are not especially meaningful, and the program author has not used macro or constant definitions to document the constants; we have to make the best of the information available.

We can often understand the meaning of an expression by applying it on sample data. In our case we can work based on the initial value of `i` (0) and assume that `cp` points to a string containing a number (for example, 24) based on our knowledge of the formatting specifications that `expand` accepts. We can then create a table containing the values of all variables and expression parts as each expression part is evaluated. We use the notation `i'` and `*cp'` to denote the variable value after the expression has been evaluated.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>i</th>
<th>i*10</th>
<th>*cp</th>
<th>*cp-'0'</th>
<th>i'</th>
<th>*cp'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>'2'</td>
<td>2</td>
<td>2</td>
<td>'4'</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>20</td>
<td>'4'</td>
<td>4</td>
<td>24</td>
<td>0</td>
</tr>
</tbody>
</table>

The expression `*cp - '0'` uses a common idiom: by subtracting the ordinal value of '0' from `*cp` the expression yields the integer value of the character digit pointed to by `*cp`. Based on the table we can now see a picture emerging: after the

---

31netbsdsrc/games/hack/hack.objnam.c:352–253
loop terminates, \( i \) will contain the decimal value of the numeric string pointed to by \( cp \) at the beginning of the loop.

Armed with the knowledge of what \( i \) stands for (the integer value of a tab-stop specification), we can now examine the lines that verify the specification. The expression that verifies \( i \) for reasonable values is straightforward. It is a Boolean expression based on the logical OR (\( \text{||} \)) of two other expressions. Although this particular expression reads naturally as English text (print an error message if \( i \) is either less than or equal to zero, or greater than 255), it is sometimes useful to transform Boolean expressions to a more readable form. If, for example, we wanted to translate the expression into the range membership expression we used above, we would need to substitute the logical OR with a logical AND (\( \&\& \)). This can easily be accomplished by using De Morgan’s rules.\(^{32}\)

\[
&!(a \text{ || } b) \iff !a \text{ && } !b \\
&!(a \text{ && } b) \iff !a \text{ || } !b
\]

We can thus transform the testing code as follows:

\[
&i \leq 0 \text{ || } i > 256 \iff \\
&!(i \leq 0) \text{ && } !(i > 256) \\
&!(i > 0) \text{ && } i \leq 256 \\
&!(0 < i \text{ && } i \leq 256) \\
&!(0 < i \leq 256)
\]

In our example we find both the initial and final expressions equally readable; in other cases you may find that De Morgan’s rules provide you a quick and easy way to disentangle a complicated logical expression.

When reading Boolean expressions, keep in mind that in many modern languages Boolean expressions are evaluated only to the extent needed. In a sequence of expressions joined with the \( \&\& \) operator (a conjunction), the first expression to evaluate to false will terminate the evaluation of the whole expression yielding a false result. Similarly, in a sequence of expressions joined with the \( \text{||} \) operator (a disjunction), the first expression to evaluate to true will terminate the evaluation of the whole expression yielding a true result. Many expressions are written based on this short-circuit evaluation property and should be read in the same way. When reading a conjunction, you can always assume that the expressions on the left of the expression you are examining

\(^{32}\)We use the operator \( \iff \) to denote that two expressions are equivalent. This is not a C/C++/C#/Java operator.
are true; when reading a disjunction, you can similarly assume that the expressions on the left of the expression you are examining are false. As an example, the expression in the following if statement will become true only when all its constituent elements are true, and t->type will be evaluated only when t is not NULL.\textsuperscript{33}

\begin{verbatim}
if (t != NULL && t->type != TEOF && interactive && really_exit)
   really_exit = 0;
\end{verbatim}

Conversely, in the following example argv[1] will be checked for being NULL only if argv is not NULL.\textsuperscript{34}

\begin{verbatim}
   return -1;
\end{verbatim}

In both cases, the first check guards against the subsequent dereference of a NULL pointer. Our getstops function also uses short-circuit evaluation when checking that a delimiter specified (i) is larger than the previous one (tabstops[nstops-1]) (Figure 2.5:4). This test will be performed only if at least one additional delimiter specification has been processed (nstops > 0). You can depend on the short-circuit evaluation property in most C-derived languages such as C++, Perl, and Java; on the other hand, Fortran, Pascal, and most Basic dialects will always evaluate all elements of a Boolean expression.

**Exercise 2.20** Locate expressions containing questionable assumptions about character code values in the book’s CD-ROM. Read about the Java Character class test and conversion methods such as isUpper and toLowerCase or the corresponding ctype family of C functions (isupper, tolower, and so on). Propose changes to make the code less dependent on the target architecture character set.

**Exercise 2.21** Find, simplify, and reason about five nontrivial Boolean expressions in the source code base. Do not spend time on understanding what the expression elements mean; concentrate on the conditions that will make the expression become true or false. Where possible, identify and use the properties of short-circuit evaluation.

**Exercise 2.22** Locate and reason about five nontrivial integer or character expressions in the source code base. Try to minimize the amount of code you need to comprehend in order to reason about each expression.

\textsuperscript{33}netbsdsrc/bin/ksh/main.c:606–607
\textsuperscript{34}netbsdsrc/lib/libedit/term.c:1212–1213
static int
gen_init(void)
{
    [...]  
    if (((sigaction(SIGXCPU, &n_hand, &o_hand) < 0) && 
        (o_hand.sa_handler == SIG_IGN) && 
        (sigaction(SIGXCPU, &o_hand, &n_hand) < 0))
        goto out;)
    n_hand.sa_handler = SIG_IGN;
    if ((sigaction(SIGPIPE, &n_hand, &o_hand) < 0) ||
        (sigaction(SIGXFSZ, &n_hand, &o_hand) < 0))
        goto out;
    return(0);  

    out:
    syswarn(1, errno, "Unable to set up signal handler");
    return(-1);
}

Figure 2.6 The goto statement used for a common error handler.

2.8 goto Statements

The code segment that complains about unreasonable tab specifications (Figure 2.5:3) begins with a word followed by a colon. This is a label: the target of a goto instruction. Labels and goto statements should immediately raise your defenses when reading code. They can be easily abused to create "spaghetti" code: code with a flow of control that is difficult to follow and figure out. Therefore, the designers of Java decided not to support the goto statement. Fortunately, most modern programs use the goto statement in a small number of specific circumstances that do not adversely affect the program’s structure.

You will find goto often used to exit a program or a function after performing some actions (such as printing an error message or freeing allocated resources). In our example the exit(1) call at the end of the block will terminate the program, returning an error code (1) to the system shell. Therefore all goto statements leading to the bad label are simply a shortcut for terminating the program after printing the error message. In a similar manner, the listing in Figure 2.6 illustrates how a common error handler (Figure 2.6:4) is used as a common exit point in all places where an error is found (Figure 2.6:1, Figure 2.6:2). A normal exit route for the function, located before the error handler (Figure 2.6:3), ensures that the handler will not get called when no error occurs.

\[\text{netbsdsrc/bin/pax/pax.c:309–412}\]
Basic Programming Elements

You will also find the goto statement often used to reexecute a portion of code, presumably after some variables have changed value or some processing has been performed. Although such a construct can often be coded by using a structured loop statement (for example, for (;;) together with break and continue, in practice the coder’s intent is sometimes better communicated by using goto. A single label, almost invariably named again or retry, is used as the goto target. The example in Figure 2.7, which locates the entry of a specific service in the system’s database while ignoring comments and overly large lines, is a typical case. (Interestingly, the code example also seems to contain a bug. If a partial line is read, it continues by reading the remainder as if it were a fresh line, so that if the tail of a long line happened to look like a service definition it would be used. Such oversights are common targets for computer security exploits.)

Finally, you will find the goto statement used to change the flow of control in nested loop and switch statements instead of using break and continue, which affect only the control flow in the innermost loop. Sometimes goto is used even if the nesting level would allow the use of a break or continue statement. This is used in large, complex loops to clarify where the flow of control will go and to avoid the possibility of errors should a nested loop be added around a particular break or continue statement. In the example in Figure 2.8 the statement goto have_msg is used instead of break to exit the for loop.

Exercise 2.23 Locate five instances of code that use the goto statement in the code base. Categorize its use (try to locate at least one instance for every one of the possible uses we outlined), and argue whether each particular goto could and should be replaced with a loop or other statement.

Figure 2.7 The use of goto to reexecute code.
Exercise 2.24  The function getstops produces the same error message for a number of different errors. Describe how you could make its error reporting more user-friendly while at the same time eliminating the use of the goto statement. Discuss when such source code changes are appropriate and when they should be avoided.

2.9 Refactoring in the Small

The rest of the getstops code is relatively straightforward. After checking that each tab stop is greater than the previous one (Figure 2.5:4), the tab stop offset is stored in the tabstops array. After a single tab stop number has been converted into an integer (Figure 2.5:2), cp will point to the first nondigit character in the string (that is, the loop will process all digits and terminate at the first nondigit). At that point, a series of checks specified by if statements control the program’s operation. If cp points to the end of the tab stop specification string (the character with the value 0, which signifies the end of a C string), then the loop will terminate (Figure 2.5:5). The last if (Figure 2.5:6) will check for invalid delimiters and terminate the program operation (using the goto bad statement) if one is found.

The body of each one of the if statements will transfer control somewhere else via a goto or break statement. Therefore, we can also read the sequence as:

```c
if (*cp == 0)  
  break;
else if (*cp != ',' && *cp != '')
  goto bad;
else
  cp++;
```
This change highlights the fact that only one of the three statements will ever get executed and makes the code easier to read and reason about. If you have control over a body of code (that is, it is not supplied or maintained by an outside vendor or an open-source group), you can profit by reorganizing code sections to make them more readable. This improvement of the code’s design after it has been written is termed refactoring. Start with small changes such as the one we outlined—you can find more than 70 types of refactoring changes described in the relevant literature. Modest changes add up and often expose larger possible improvements.

As a further example, consider the following one-line gem.38

\[
\text{op} = \&(!x \ ? \ (!y \ ? \ \text{upleft} : (y == \text{bottom} \ ? \ \text{lowleft} : \text{left})) : \\
(x == \text{last} \ ? \ (!y \ ? \ \text{upright} : (y == \text{bottom} \ ? \ \text{lowright} : \text{right})) : \\
(!y \ ? \ \text{upper} : (y == \text{bottom} \ ? \ \text{lower} : \text{normal}))))[w->orientation];
\]

The code makes excessive use of the conditional operator \(?:\). Read expressions using the conditional operator like \textit{if} code. As an example, read the expression39

\[
\text{sign} \ ? \ -n : n
\]

as follows:

“If \text{sign} is true, then the value of the expression is \(-n\); otherwise, the value of the expression is \(n\).”

Since we read an expression like an \textit{if} statement, we can also format it like an \textit{if} statement; one that uses \textit{x} \ ? instead of \textit{if} (\(x\)), parentheses instead of curly braces, and \textbf{:} instead of \textbf{else}. To reformat the expression, we used the indenting features of our editor in conjunction with its ability to show matching parentheses. You can see the result in Figure 2.9 (left).

Reading the conditional expression in its expanded form is certainly easier, but there is still room for improvement. At this point we can discern that the \(x\) and \(y\) variables that control the expression evaluation are tested for three different values:

1. \(0\) (expressed as \(!x\ \text{or} \ !y\)
2. \text{bottom} or \text{last}
3. All other values

\[\text{38} \text{netbsdsrc/games/worms/worms.c:419} \]
\[\text{39} \text{netbsdsrc/bin/csh/set.c:852}\]
We can therefore rewrite the expression formatted as a series of cascading if–else statements (expressed using the ?: operator) to demonstrate this fact. You can see the result in Figure 2.9 (right).

The expression’s intent now becomes clear: the programmer is selecting one of nine different location values based on the combined values of x and y. Both alternative formulations, however, visually emphasize the punctuation at the expense
struct options *locations[3][3] = {
    {upleft, upper, upright},
    {left, normal, right},
    {lowleft, lower, lowright},
};

int xlocation, ylocation;

if (x == 0)
    xlocation = 0;
else if (x == last)
    xlocation = 2;
else
    xlocation = 1;

if (y == 0)
    ylocation = 0;
else if (y == bottom)
    ylocation = 2;
else
    ylocation = 1;

op = &(locations[ylocation][xlocation])[w->orientation];

Figure 2.10 Location detection code replacing the conditional expression.

of the semantic content and use an inordinate amount of vertical space. Nevertheless, based on our newly acquired insight, we can create a two-dimensional array containing these location values and index it using offsets we derive from the x and y values. You can see the new result in Figure 2.10. Notice how in the initialization of the array named locations, we use a two-dimensional textual structure to illustrate the two-dimensional nature of the computation being performed. The initializers are laid out two-dimensionally in the program text, the array is indexed in the normally unconventional order [y][x], and the mapping is to integers “0, 2, 1” rather than the more obvious “0, 1, 2”, so as to make the two-dimensional presentation coincide with the semantic meanings of the words upleft, upper, and so on.

The code, at 20 lines, is longer than the original one-liner but still shorter by 7 lines from the one-liner’s readable cascading-else representation. In our eyes it appears more readable, self-documenting, and easier to verify. One could argue that the original version would execute faster than the new one. This is based on the fallacy that code readability and efficiency are somehow incompatible. There is no need to sacrifice code readability for efficiency. While it is true that efficient algorithms and certain optimizations can make the code more complicated and therefore more difficult to follow, this does not mean that making the code compact and unreadable will make it more efficient. On our system and compiler the initial and final versions of the code execute at exactly the same speed: 0.6 µs. Even if there were speed differences, the economics behind software maintenance costs, programmer salaries, and CPU performance most of the time favor code readability over efficiency.
However, even the code in Figure 2.10 can be considered a mixed blessing: it achieves its advantages at the expense of two distinct disadvantages. First, it separates the code into two chunks that, while shown together in Figure 2.10, would necessarily be separated in real code. Second, it introduces an extra encoding (0, 1, 2), so that understanding what the code is doing requires two mental steps rather than one (map “0, last, other” to “0, 2, 1” and then map a pair of “0, 2, 1” values to one of nine items). Could we somehow directly introduce the two-dimensional structure of our computation into the conditional code? The following code fragment\(^{40}\) reverts to conditional expressions but has them carefully laid out to express the computation’s intent.

```c
op = &(
    !y ? (!x ? upleft : x!=last ? upper : upright ) : 
    y!=bottom ? (!x ? left : x!=last ? normal : right ) : 
    (!x ? lowleft : x!=last ? lower : lowright )
)[w->orientation];
```

The above formulation is a prime example on how sometimes creative code layout can be used to improve code readability. Note that the nine values are right-justified within their three columns, to make them stand out visually and to exploit the repetition of “left” and “right” in their names. Note also that the usual practice of putting spaces around operators is eschewed for the case of `!=` in order to reduce the test expressions to single visual tokens, making the nine data values stand out more. Finally, the fact that the whole expression fits in five lines makes the vertical alignment of the first and last parentheses more effective, making it much easier to see that the basic structure of the entire statement is of the form

```c
op = &(<conditional-mess>)[w->orientation];
```

The choice between the two new alternative representations is largely a matter of taste; however, we probably would not have come up with the second formulation without expressing the code in the initial, more verbose and explicit form.

The expression we rewrote was extremely large and obviously unreadable. Less extreme cases can also benefit from some rewriting. Often you can make an expression more readable by adding whitespace, by breaking it up into smaller parts by means of temporary variables, or by using parentheses to amplify the precedence of certain operators.

---

\(^{40}\)Suggested by Guy Steele.
You do not always need to change the program structure to make it more readable. Often items that do not affect the program’s operation (such as comments, the use of whitespace, and the choice of variable, function, and class names) can affect the program’s readability. Consider the work we did to understand the code for the `getstops` function. A concise comment before the function definition would enhance the program’s future readability.

```c
/*
 * Parse and verify the tab stop specification pointed to by cp
 * setting the global variables nstops and tabstops[].
 * Exit the program with an error message on bad specifications.
 */
```

When reading code under your control, make it a habit to add comments as needed.

In Sections 2.2 and 2.3 we explained how names and indentation can provide hints for understanding code functionality. Unfortunately, sometimes programmers choose unhelpful names and indent their programs inconsistently. You can improve the readability of poorly written code with better indentation and wise choice of variable names. These measures are extreme: apply them only when you have full responsibility and control over the source code, you are sure that your changes are a lot better than the original code, and you can revert to the original code if something goes wrong. Using a version management system such as the Revision Control System (RCS), the Source Code Control System (SCCS), the Concurrent Versions System (CVS), or Microsoft’s Visual SourceSafe can help you control the code modifications. The adoption of a specific style for variable names and indentation can appear a tedious task. When modifying code that is part of a larger body to make it more readable, try to understand and follow the conventions of the rest of the code (see Chapter 7). Many organizations have a specific coding style; learn it and try to follow it. Otherwise, adopt one standard style (such as one of those used by the GNU41 or BSD42 groups) and use it consistently. When the code indentation is truly inconsistent and cannot be manually salvaged, a number of tools (such as `indent`) can help you automatically reindent it to make it more readable (see Section 10.7). Use such tools with care: the judicious use of whitespace allows programmers to provide visual clues that are beyond the abilities of automated formatting tools. Applying `indent` to the code example in Figure 2.10 would definitely make it less readable.

Keep in mind that although reindenting code may help readability, it also messes up the program’s change history in the revision control system. For this reason it

41http://www.gnu.org/prep/standards_toc.html
42netbsdsrc/share/misc/style:1–315
is probably best not to combine the reformatting with any actual changes to the program’s logic. Do the reformat, check it in, and then make the other changes. In this way future code readers will be able to selectively retrieve and review your changes to the program’s logic without getting overwhelmed by the global formatting changes. On the flip side of the coin, when you are examining a program revision history that spans a global reindentation exercise using the `diff` program, you can often avoid the noise introduced by the changed indentation levels by specifying the `-w` option to have `diff` ignore whitespace differences.

**Exercise 2.25** Provide five examples from your environment or the book’s CD-ROM where the code structure can be improved to make it more readable.

**Exercise 2.26** You can find tens of intentionally unreadable C programs at the International Obfuscated C Code Contest Web site. Most of them use several layers of obfuscation to hide their algorithms. See how gradual code changes can help you untangle their code. If you are not familiar with the C preprocessor, try to avoid programs with a large number of `#define` lines.

**Exercise 2.27** Modify the position location code we examined to work on the mirror image of a board (interchange the right and left sides). Time yourself in modifying the original code and the final version listed in Figure 2.10. Do not look at the readable representations; if you find them useful, create them from scratch. Calculate the cost difference assuming current programmer salary rates (do not forget to add overheads). If the readable code runs at half the speed of the original code (it does not), calculate the cost of this slowdown by making reasonable assumptions concerning the number of times the code will get executed over the lifetime of a computer bought at a given price.

**Exercise 2.28** If you are not familiar with a specific coding standard, locate one and adopt it. Verify local code against the coding standard.

### 2.10 do Loops and Integer Expressions

We can complete our understanding of the `expand` program by turning our attention to the body that does its processing (Figure 2.3, page 27). It starts with a `do` loop. The body of a `do` loop is executed at least once. In our case the `do` loop body is executed for every one of the remaining arguments. These can specify names of files that are to be tab-expanded. The code processing the file name arguments (Figure 2.3:6) reopens the `stdin` file stream to access each successive file name argument. If no file name arguments are specified, the body of the `if` statement (Figure 2.3:6) will not get

---

43http://www.ioccc.org
executed and expand will process its standard input. The actual processing involves reading characters and keeping track of the current column position. The switch statement, a workhorse for character processing, handles all different characters that affect the column position in a special way. We will not examine the logic behind the tab positioning in detail. It is easy to see that the first three and the last two blocks can again be written as a cascading if-else sequence. We will focus our attention on some expressions in the code.

Sometimes equality tests such as the ones used for nstops (for example, nstops == 0) are mistakenly written using the assignment operator = instead of the equality operator ==. In C, C++, and Perl a statement like the following:

```c
if ((p = q))
    q[-1] = '\n';
```

uses a valid test expression for the if statement, assigning q to p and testing the result against zero. If the programmer intended to test p against q, most compilers would generate no error. In the statement we examined, the parentheses around (p = q) are probably there to signify that the programmer’s intent was indeed an assignment and a subsequent test against zero. One other way to make such an intention clear is to explicitly test against NULL.

```c
if ((p = strchr(name, '=')) != NULL) {
    p++;
}
```

In this case the test could also have been written as if (p = strchr(name, '=')), but we would not know whether this was an intentional assignment or a mistake.

Finally, another approach you may come across is to adopt a style where all comparisons with constants are written with the constant on the lefthand side of the comparison.

```c
if (0 == serconsole)
    serconsinit = 0;
```

When such a style is used, mistaken assignments to constants are flagged by the compiler as errors.

---

44 netbsdsrc/bin/ksh/history.c:313–314
45 netbsdsrc/bin/sh/var.c:507–508
46 netbsdsrc/sys/arch/amiga/dev/ser.c:227–228
When reading Java or C# programs, there are fewer chances of encountering such errors since these languages accept only Boolean values as control expressions in the corresponding flow statements. We were in fact unable to locate a single suspicious statement in the Java code found in the book’s CD-ROM.

The expression `column & 7` used to control the first `do` loop of the loop-processing code is also interesting. The `&` operator performs a bitwise-and between its two operands. In our case, we are not dealing with bits, but by masking off the most significant bits of the `column` variable it returns the remainder of `column` divided by 8. When performing arithmetic, read `a & b` as `a % (b + 1)` when `b = 2^n − 1`. The intent of writing an expression in this way is to substitute a division with a—sometimes more efficiently calculated—bitwise-and instruction. In practice, modern optimizing compilers can recognize such cases and do the substitution on their own, while the speed difference between a division and a bitwise-and instruction on modern processors is not as large as it used to be. You should therefore learn to read code that uses these tricks, but avoid writing it.

There are two other common cases where bit instructions are used as substitutes for arithmetic instructions. These involve the `shift` operators `<<` and `>>`, which shift an integer’s bits to the left or right. Since every bit position of an integer has a value equal to a power of 2, shifting an integer has the effect of multiplying or dividing it by a power of 2 equal to the number of shifted bits. You can therefore think of shift operators in an arithmetic context as follows.

- Read `a << n` as `a * k`, where `k = 2^n`. The following example uses the shift operator to multiply by 4:\[n = ((dp - cp) << 2) + 1; /* 4 times + NULL */\]

- Read `a >> n` as `a / k`, where `k = 2^n`. The following example from a binary search routine uses the right shift operator to divide by 2:\[bp = bp1 + ((bp2 - bp1) >> 1);\]

Keep in mind that Java’s logical shift right operator `>>>` should not be used to perform division arithmetic on signed quantities since it will produce erroneous results when applied on negative numbers.

\[\text{netbsdsrc/bin/csh/str.c:460}\]
\[\text{netbsdsrc/bin/csh/func.c:106}\]
Exercise 2.29 Most compilers provide a facility to view the compiled code in assembly language. Find out how to generate assembly code when compiling a C program in your environment and examine the code generated by your compiler for some instances of arithmetic expressions and the corresponding expressions using bit instructions. Try various compiler optimization levels. Comment on the readability and the code efficiency of the two alternatives.

Exercise 2.30 What type of argument could cause expand to fail? Under what circumstances could such an argument be given? Propose a simple fix.

2.11 Control Structures Revisited

Having examined the syntactic details of the control flow statements we can now focus our attention on the way we can reason about them at an abstract level.

The first thing you should remember is to examine one control structure at a time, treating its contents as a black box. The beauty of structured programming is that the control structures employed allow you to abstract and selectively reason about parts of a program, without getting overwhelmed by the program’s overall complexity.

Consider the following code sequence:

```
while (enum.hasMoreElements()) {
    [...]
    if (object instanceof Resource) {
        [...]
        if (!copy(is, os))
            [...]
    } else if (object instanceof InputStream) {
        [...]
        if (!copy((InputStream) object, os))
            [...]
    } else if (object instanceof DirContext) {
        [...]
    }
}
```

Although we have removed a large part of the 20 lines of code, the loop still appears quite complex. However, the way you should read the above loop is

```
while (enum.hasMoreElements()) {
    // Do something
}
```

---

At that level of abstraction you can then focus on the loop body and examine its functioning without worrying about the control structure in which it is enclosed. This idea suggests a second rule we should follow when examining a program’s flow of control: treat the controlling expression of each control structure as an assertion for the code it encloses. Although the above statement may appear obtuse or trivial, its significance to the understanding of code can be profound. Consider again the while statement we examined. The typical reading of the control structure would be that while \texttt{enum.hasMoreElements()} is true the code inside the loop will get executed. When, however, you examine the loop’s body (in isolation as we suggested above), you can always assume that \texttt{enum.hasMoreElements()} will be true and that, therefore, the enclosed statement

\begin{verbatim}
NameClassPair ncPair = (NameClassPair) enum.nextElement();
\end{verbatim}

will execute without a problem. The same reasoning also applies to if statements. In the code below you can be sure that when \texttt{links.add} is executed the \texttt{links} collection will not contain a next element.\footnote{cocoon/src/java/org/apache/cocoon/Main.java:574–576}

\begin{verbatim}
if (!links.contains(next)) {
    links.add(next);
}
\end{verbatim}

Unfortunately, some control statements taint the rosy picture we painted above. The return, goto, break, and continue statements as well as exceptions interfere with the structured flow of execution. Reason about their behavior separately since they all typically either terminate or restart the loop being processed. This assumes that for goto statements their target is the beginning or the end of a loop body, that is, that they are used as a multilevel break or continue. When this is not the case, all bets are off.

When going over loop code, you may want to ensure that the code will perform according to its specification under all circumstances. Informal arguments are sufficient for many cases, but sometimes a more rigorous approach is needed.

Consider the binary search algorithm. Getting the algorithm right is notoriously difficult. Knuth [Knu98] details how its use was first discussed in 1946, but nobody published a correct algorithm working for arrays with a size different from $2^n - 1$ until 1962. Bentley [Ben86] adds that when he asked groups of professional programmers to implement it as an exercise, only 10% got it right.
Consider the standard C library implementation of the binary search algorithm listed in Figure 2.11. We can see that it works by gradually reducing the search interval stored in the \( \text{lim} \) variable and adjusting the start of the search range stored in \( \text{base} \), but it is not self-evident whether the arithmetic calculations performed are correct under all circumstances. If you find it difficult to reason about the code, the comment that precedes it might help you.

The code below is a bit sneaky. After a comparison fails, we divide the work in half by moving either left or right. If \( \text{lim} \) is odd, moving left simply involves halving \( \text{lim} \): e.g., when \( \text{lim} \) is 5 we look at item 2, so we change \( \text{lim} \) to 2 so that we will look at items 0 & 1. If \( \text{lim} \) is even, the same applies. If \( \text{lim} \) is odd, moving right again involves halving \( \text{lim} \), this time moving the \( \text{base} \) up one item past \( p \): e.g., when \( \text{lim} \) is 5 we change \( \text{base} \) to item 3 and make \( \text{lim} \) 2 so that we will look at items 3 & 4. If \( \text{lim} \) is even, however, we have to shrink it by one before halving: e.g., when \( \text{lim} \) is 4, we still looked at item 2, so we have to make \( \text{lim} \) 3, then halve, obtaining 1, so that we will only look at item 3.

If you—like myself—did not regard the above comment as particularly enlightening or reassuring, you might consider employing more sophisticated methods.

A useful abstraction for reasoning about properties of loops is based around the notions of variants and invariants. A loop invariant is an assertion about the program

---

Figure 2.11 Binary search implementation.
state that is valid both at the beginning and at the end of a loop. By demonstrating that a particular loop maintains the invariant, and by choosing an invariant so that when the loop terminates it can be used to indicate that the desired result has been obtained, we can ensure that an algorithm’s loop will work within the envelope of the correct algorithm results. Establishing this fact, however, is not enough. We also need to ensure that the loop will terminate. For this we use a variant, a measure indicating our distance from our final goal, which should be decreasing at every loop iteration. If we can demonstrate that a loop’s operation decreases the variant while maintaining the invariant, we determine that the loop will terminate with the correct result.

Let us start with a simple example. The following code finds the maximum value in the `depths` array.52

```c
max = depths[n];
while (n--) {
    if (depths[n] > max)
        max = depths[n];
}
```

If we define \( n_0 \) as the number of elements in the `depths` array (initially held in variable \( n \)), we can formally express the result we want at the end of the loop as

\[
max = \text{maximum}\{\text{depths}[0 : n_0]\}
\]

We use the symbolism \([a : b]\) to indicate a range than includes \( a \) but ends one element before \( b \), that is, \([a : b - 1]\). A suitable invariant can then be

\[
max = \text{maximum}\{\text{depths}[n : n_0]\}
\]

The invariant is established after the first assignment to \( max \), so it holds at the beginning of the loop. Once \( n \) is decremented, it does not necessarily hold, since the range \([n : n_0]\) contains the element at index \( n \), which might be larger than the maximum value held in \( max \). The invariant is reestablished after the execution of the `if` statement, which will adjust \( max \) if the value of the new member of the now extended range is indeed larger than the maximum we had to this point. We have thus shown that the invariant will also be true at the end of every loop iteration and that therefore it will be true when the loop terminates. Since the loop will terminate when \( n \) (which we can consider as our loop’s variant) reaches 0, our invariant can at that point be rewritten in the form

52XFree86-3.3/xc/lib/Xt/GCManager.c:252–256
of the original specification we wanted to satisfy, demonstrating that the loop does indeed arrive at the result we want.

We can apply this reasoning to our binary search example. Figure 2.12 illustrates the same algorithm slightly rearranged so as to simplify reasoning with the invariant.

- We substituted the right shift operations >> with division.
- We factored out the size variable since it is used only to simulate pointer arithmetic without having to know the pointer’s type.
- We moved the last expression of the for statement to the end of the loop to clarify the order of operations within the loop.

A suitable invariant can be the fact that the value we are looking for lies within a particular range. We will use the notation \( R \in [a : b) \) to indicate that the result of the search lies between the array elements \( a \) (including \( a \)) and \( b \) (excluding \( b \)). Since base and \( \text{lim} \) are used within the loop to delimit the search range, our invariant will be \( R \in [\text{base} : \text{base} + \text{lim}) \). We will show that the bsearch function will indeed find the value in the array, if such a value exists, by demonstrating that the invariant is maintained after each loop iteration. Since the comparison function \( \text{compar} \) is always called with an argument from within the invariant’s range (\( \text{base} + \text{lim}/2 \)), and since \( \text{lim} \) (our variant) is halved after every loop iteration, we can be sure that \( \text{compar} \) will eventually locate the value if that value exists.

At the beginning of the bsearch function we can only assert the function’s specification: \( R \in [\text{base0} : \text{base0} + \text{nmemb}) \). However, after Figure 2.12:1 this
can be expressed as $R \in [\text{base} : \text{base} + \text{nmemb})$, and after the for assignment (Figure 2.12:2) as $R \in [\text{base} : \text{base} + \text{lim})$—our invariant. We have thus established that our invariant holds at the beginning of the loop.

The result of the \texttt{compar} function is positive if the value we are looking for is greater than the value at point $p$. Therefore, at Figure 2.12:3 we can say that

$$R \in (p : \text{base} + \text{lim}) \equiv R \in [p + 1 : \text{base} + \text{lim}).$$

If we express the original base value as $\text{base}_{old}$ our original invariant, after the assignment at Figure 2.12:4, is now

$$R \in [\text{base} : \text{base}_{old} + \text{lim}).$$

Given that $p$ was given the value of $\text{base} + \text{lim}/2$, we have

$$\begin{align*}
\text{base} &= \text{base}_{old} + \frac{\text{lim}}{2} + 1 \\
\text{base}_{old} &= \text{base} - \frac{\text{lim}}{2} - 1.
\end{align*}$$

By substituting the above result in the invariant we obtain

$$R \in \left[\text{base} : \text{base} - \frac{\text{lim}}{2} - 1 + \text{lim}\right).$$

When $\text{lim}$ is decremented by one at Figure 2.12:5 we substitute $\text{lim} + 1$ in our invariant to obtain

$$\begin{align*}
R \in \left[\text{base} : \text{base} - \frac{\text{lim} + 1}{2} - 1 + \text{lim} + 1\right) \equiv \\
R \in \left[\text{base} : \text{base} + \text{lim} - \frac{\text{lim} + 1}{2}\right) \equiv \\
R \in \left[\text{base} : \text{base} + \frac{\text{lim}}{2}\right).
\end{align*}$$

By a similar process, in the case where the result of the \texttt{compar} function is negative, indicating that the value we are looking for is less than the value at point $p$, we obtain

$$\begin{align*}
R \in [\text{base} : p) \equiv \\
R \in \left[\text{base} : \text{base} + \frac{\text{lim}}{2}\right).
\end{align*}$$

Note that the invariant is now the same for both comparison results. Furthermore, when $\text{lim}$ is halved at Figure 2.12:7 we can substitute its new value in the invariant to obtain $R \in [\text{base} : \text{base} + \text{lim})$, that is, the invariant we had at the top of the loop. We have thus shown that the loop maintains the invariant and therefore will correctly
locate the value within the array. Finally, when \( \text{lim} \) becomes zero, the range where the value can lie is empty, and it is therefore correct to return \( \text{NULL} \), indicating that the value could not be located.

**Exercise 2.31** Locate five control structures spanning more than 50 lines in the book’s CD-ROM and document their body with a single-line comment indicating its function.

**Exercise 2.32** Reason about the body of one of the above control structures, indicating the place(s) where you use the controlling expression as an assertion.

**Exercise 2.33** Provide a proof about the correct functioning of the insertion sort function\(^53\) found as part of the radix sort implementation in the book’s CD-ROM. **Hint:** The innermost \( \text{for} \) loop just compares two elements; the \( \text{swap} \) function is executed only when these are not correctly ordered.

### Further Reading

Kernighan and Plauger [KP78] and, more recently, Kernighan and Pike [KP99, Chapter 1] provide a number of suggestions to improve code style; these can be used to disentangle badly written code while reading it. Apart from the specific style sheets mentioned in Section 2.9, a well-written style guide is the *Indian Hill C Style and Coding Standard*; you can easily find it on-line by entering its title in a Web search engine. For a comprehensive bibliography on programming style, see Thomas and Oman [TO90]. The now classic article presenting the problems associated with the \( \text{goto} \) statement was written by Dijkstra [Dij68]. The effects of program indentation on comprehensibility are studied in the work by Miara et al. [MMNS83], while the effects of formatting and commenting are studied by Oman and Cook [OC90]. For an experiment of how comments and procedures affect program readability, see Tenny [Ten88]. Refactoring as an activity for improving the code’s design (and readability) is presented in Fowler [Fow00, pp. 56–57]. If you want to see how a language is introduced by its designers, read Kernighan and Ritchie [KR88] (covering C), Stroustrup [Str97] (C++), Microsoft Corporation [Mic01] (C#), and Wall et al. [WCSP00] (Perl). In addition, Ritchie [Rit79] provides an in-depth treatment of C and its libraries, while Linden [Lin94] lucidly explains many of the C language’s finer points.

Invariants were introduced by C. A. R. Hoare [Hoa71]. You can find them also described in references [Ben86, pp. 36–37; Mey88, pp. 140–143; Knu97, p. 17; HT00, p. 116.] A complete analysis of the binary search algorithm is given in Knuth [Knu98].

\(^53\)netbsdsrc/lib/libc/stdlib/radixsort.c:310–330
Symbols

|, 30
*, regular expression, 341
+, regular expression, 340
-, operator, 35
->, operator, 102
.flag, see the flag under the corresponding command
-name, see the name under the corresponding letter
. *, regular expression, 341
\.**, troff command, 262
. , regular expression, 341
/**/, operator, 176
/**/, 231, 261
*/*, 231
/flag, see the flag under the corresponding command
-. , 302
-, regular expression, 340
\], editor command, 343
\., regular expression, 341
<<, operator, 53, 308
<, 261
==, operator, 52
=, operator, 30, 52
>>>., Java operator, 53
>>, operator, 53, 58, 308
>, 261
?, operator, 46, 174
?, regular expression, 340
@(#), revision-id tag, 206
@, 231, 261
@code, Texinfo command, 263
[ ], 341
[... ], 11
[ ], 341
\<=, regular expression, 341
\>=, regular expression, 341
\, regular expression, 341
#*, operator, 176
$*, 259
$*, identifier name, 194
$<, identifier name, 194
$>, identifier name, 194
$?, identifier name, 194
$@, identifier name, 194
$$, identifier name, 194
$, regular expression, 341
$name, see the name under the corresponding letter
%, editor command, 346
%, regular expression, 340
%name, see the name under the corresponding letter
&, operator, 22, 41
&., operator, 53, 64
..., 297
.name, see the name under the corresponding letter
Digits
05.fastfs, sample, 243
16-bit, 238
4GL, 16

A
%A%, revision-id tag, 206
.a, file extension, 227
abort, C library, 168
abstract
class, see class, abstract
data type, 106, 249, 318
machine, 113
Abstract Windowing Toolkit, 286, 329
abstract, Java keyword, 276, 306
abstraction, 91
Acceptor, design pattern, 336
Access, see Microsoft Access
access method, 304
ace, sample, 329, 336, 338, 399
ace_min, 315
ActionGoToEdit.java, sample, 299
active class, 13
Active Object, design pattern, 336
Active Template Library, 325
ActiveX, 325
Ada, 16, 98, 126, 296, 299
Ada keyword
package, 299
use, 299
with, 299
Ada-95, 300
adaptation, 8
Adaptec, 280
Adapter, design pattern, 336
Adaptive Communication Environment, 329, 338
adb.c, sample, 108, 109
adb_direct.c, sample, 108, 109
Addr.i, sample, 308
administrator manual, 242
Adobe FrameMaker, 260
adornment, 276
ADT, 106, 318
aggregation association, 13
agile programming, 8
aha.c, sample, 281
aha_done, 281
aha_finish_ccbs, 281
aha_intr, 281
algebraic specification, 318
algorithms, 245
aliasing, through pointers, 70
a11, pseudo-target, 194
a1loc.c, sample, 217
a1locas.c, sample, 119, 120, 233, 246
Apache, program, xxv, 162, 181, 244, 260, 270, 322, 323
Apache, sample, 399
API, see also under the name of the specific
API (e.g. Win32)
documentation, 249
Java platform documentation, 28, 264
thread, 162
providing OS services, 160
Sun RPC, 272
API Spy, 370
APM, 292
apple, 20
AppleTalk, 186
application, 336
Application programming interface, see API
apply.c, sample, 35
apropos, program, 255
apropos.1, sample, 255
apropos.pdf, sample, 255
.A, troff command, 262
.arc, 139
arch, directory name, 183
arch, sample, 186
archeology, see software archeology
architecture, 180
centralized repository, 268
control models, 285
data-flow, 273
design pattern, 331
distributed, 268
domain-specific, 333
event-driven, 285
framework, 329
hierarchies, 282
layered, 279
non-trivial, 180
object-oriented, 275
packaging, 292
pipes-and-filters, 273
reuse, 328
state transition, 291
system manager, 289
$Archive:$, revision-id tag, 206
args.c, sample, 246
argv, identifier name, 20
ArgoEvent.java, sample, 289
ArgoModuleEvent.java, sample, 289
ArgoUML, 278
array, 96
as argument, 65
associative, 111
data structure, 96
dynamically allocated, 89
index, 111
results, 65
two-dimensional, 111
array.t, sample, 215
as, program, 323
AS, Modula keyword, 299
asc.c, sample, 282
asc.disp.out, 212
asc.intr, 282
ASCII characters, 40
ASCII diagram, 256
ascmagic.c, sample, 22
.asm, file extension, 227
.asp, file extension, 227
assembly code, see symbolic code
assert, C library, 218, 220
assert, Java method, 220
assertion, 28, 55, 56, 217, 219, 220
association navigation, 13
associative array, 111
astosc.c, sample, 213
astosc.out, 213
asymmetric bounds, 100
asynchronous, 143, 165, 167
Asynchronous Completion Token, design
pattern, 336
asynchronous signals, 156
at, program, 160
at.1, sample, 251
at.c, sample, 33
at.pdf, sample, 251
at, sample, 135
ATL, see Active Template Library
ATM, 186
atom.c, sample, 114
atrun.c, sample, 160
attribute, see class, property
$Author:$, revision-id tag, 206
$Author$, revision-id tag, 206
@author, javadoc tag, 263
autoconf, 199, 224
AVL tree, 310
awk, program, 209, 227, 309, 330, 357, 377
.awk, file extension, 227
AWT, see Abstract Windowing Toolkit
B
B+ tree, 319
b4light.c, sample, 108
%B%, revision-id tag, 206
b, Hungarian prefix, 236
.B, troff command, 262
backward-1st, editor command, 346
backward-up-1st, editor command, 346
bad, C++ library, 24
Index

.bas, file extension, 227
base case, 143, 146
base class, see class, base
BaseDirContextTestCase.java, sample, 220
Basic, 42, 309, 354
basic block, 372
basic block coverage analysis, 372
.bat, file extension, 227
beautifier, 365
.Bl, troff command, 262
.bin, directory name, 183
binary search, 55, 113
binary tree, 126
BIND, xxv, 248
bio–doread, 281
biodone, 281
biowait, 281
bit field, 77
bitwise-AND, 53
.Bl, troff command, 262
blackboard system, 270
bless, Perl keyword, 300, 309
block, 28, 31
block.cpp, sample, 297, 301
.bmp, file extension, 227
boolean expression, 39
boss/worker, 156, 157
bpb.h, sample, 77
.br, troff command, 262
.BR, troff command, 262
branch, 203
bread, 281
break, Java keyword, 38, 39, 152
break, keyword, 32, 36, 37, 55, 96, 231
breakpoint, 6, 374
brelse, 281
bremfree, 281
BRIEF, 340
C, 15, 98
C data type, 61
C library
   abort, 168
   assert, 218, 220
   bsearch, 69, 320
   cos, 388
   ctime, 389
   ctype, 42
   EOF, 24
   exit(1), 43
   exit, 24, 36, 168
   ferror, 24
   fgets, 99
   fread, 98
   free, 85, 89
   fwrite, 98, 280
   gets, 98, 99
   isupper, 42
   longjmp, 168, 169
   main, 20, 219
   malloc, 80, 84, 85, 88, 101
   memcpy, 97, 98
   memmove, 97
   buffer, 96
   buffer overflow, 99, 141, 254
   bufinit, 281
   bug tracking database, 209
   bugs, documenting, 251
   Bugs, manual section, 251, 254
   Bugzilla, 209
   build process, 189, 212, 321
   build.bat, sample, 380
   build.properties.sample, sample, 198
   build.xml, sample, 196
   build, directory name, 183
   build, pseudo-target, 194
   bus.h, sample, 176
   bwrite, 281
   bytecode, 334
   .bz2, file extension, 227
   bzip2, program, 227
memset, 97
NULL, 52, 236
printf, 23, 73
qsort, 69, 98
realloc, 86
scanf, 99
setjmp, 169
signal, 168
sin, 388
snprintf, 99
sprintf, 98, 99
stdin, 51
stdio, 280
strcat, 99
strcmp, 22, 234
strcpy, 99
strdup, 66
strlen, 72
strncat, 99
strncpy, 99
tm_year, 390
tolower, 42
tv_sec, 76
tv_usec, 76
vsnprintf, 99
vsprintf, 99
C preprocessor, see preprocessor
C++, 42, 98, 276, 364
C++ keyword, see keyword
C++ library
bad, 24
fail, 24
good, 24
vector, 100
C++ operator, see operator
c++decl, program, 368
C-M-d, editor command, 346
C-M-m, editor command, 346
C-M-p, editor command, 346
C-M-u, editor command, 346
C#, 98, 227, 276, 300, 304
C# keyword, see keyword
C# operator, see operator
%c, revision-id tag, 206
.c, file extension, 227
C, Hungarian prefix, 236
@c, Texinfo command, 263
cache.c, sample, 63
Cache.java, sample, 234
cal.c, sample, 34
Calendar, Java class, 382, 390
call
and return, 285
by name, 172
by reference, 63, 66
diag, 134, 372
case, keyword, 32, 219, 231
cat, program, 244
cat.1, sample, 244, 251
cat.c, sample, 24, 205, 243
cat.pdf, sample, 244, 251
Catalina.java, sample, 22
catalina, sample, 358
CatalinaService.java, sample, 23
catch, Java keyword, 150, 151, 152
cb, program, 367
cb, Hungarian prefix, 236
cbo, Hungarian prefix, 237
cbrowser, 366
c, file extension, 227
cc, program, 323
-E, 361
-S, 362
.C, file extension, 227
cgi, file extension, 227
cgram.y, sample, 130
c, Hungarian prefix, 236
ch, Hungarian prefix, 236
c, file extension, 227
c, Hungarian prefix, 236
change, 252
CHANGELOG.txt, sample, 397
ChangeLog, file name, 226, 252
Changes, file name, 226
character expression, 39
Character, Java class, 42
chatter.h, sample, 305
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>check in</td>
<td>203</td>
</tr>
<tr>
<td>checkError, Java method</td>
<td>24</td>
</tr>
<tr>
<td>checknr.c, sample</td>
<td>238</td>
</tr>
<tr>
<td>chio.c, sample</td>
<td>97</td>
</tr>
<tr>
<td>chk, Hungarian prefix</td>
<td>237</td>
</tr>
<tr>
<td>chown.c, sample</td>
<td>230</td>
</tr>
<tr>
<td>@cindex, Texinfo command</td>
<td>263</td>
</tr>
<tr>
<td>circular buffer</td>
<td>108</td>
</tr>
<tr>
<td>circular list</td>
<td>122</td>
</tr>
<tr>
<td>cksum.c, sample</td>
<td>238</td>
</tr>
<tr>
<td>cl, program</td>
<td>361, 362</td>
</tr>
<tr>
<td>.class, file extension</td>
<td>227</td>
</tr>
<tr>
<td>.class, Java keyword</td>
<td>300</td>
</tr>
<tr>
<td>.classes, directory name</td>
<td>183</td>
</tr>
<tr>
<td>.clean, pseudo-target</td>
<td>194</td>
</tr>
<tr>
<td>Cleanup_Strategies_T.h, sample</td>
<td>256</td>
</tr>
<tr>
<td>client-server systems</td>
<td>269, 338</td>
</tr>
<tr>
<td>closeSe, Unix-specific function</td>
<td>80, 281</td>
</tr>
<tr>
<td>closed range</td>
<td>100</td>
</tr>
<tr>
<td>closedir, Unix-specific function</td>
<td>319</td>
</tr>
<tr>
<td>CloseHandle, Win32 SDK</td>
<td>163</td>
</tr>
<tr>
<td>.cmd, file extension</td>
<td>227</td>
</tr>
<tr>
<td>cmd, Hungarian prefix</td>
<td>237</td>
</tr>
<tr>
<td>COBOL</td>
<td>16</td>
</tr>
<tr>
<td>cocoon, sample</td>
<td>399</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>5</td>
</tr>
<tr>
<td>as exemplar</td>
<td>2</td>
</tr>
<tr>
<td>as literature</td>
<td>2</td>
</tr>
<tr>
<td>assembly, see symbolic code</td>
<td>365</td>
</tr>
<tr>
<td>beautifier</td>
<td>365</td>
</tr>
<tr>
<td>browser</td>
<td>365</td>
</tr>
<tr>
<td>generation</td>
<td>212, 334</td>
</tr>
<tr>
<td>inspections</td>
<td>9</td>
</tr>
<tr>
<td>languages</td>
<td>15</td>
</tr>
<tr>
<td>object</td>
<td>360, 362</td>
</tr>
<tr>
<td>portability, see portability</td>
<td>336</td>
</tr>
<tr>
<td>reuse</td>
<td>320</td>
</tr>
<tr>
<td>searching</td>
<td>346</td>
</tr>
<tr>
<td>standards, see guidelines</td>
<td>360, 362</td>
</tr>
<tr>
<td>symbolic</td>
<td>360, 362</td>
</tr>
<tr>
<td>wizard</td>
<td>328, 330</td>
</tr>
<tr>
<td>codefind.pl, sample</td>
<td>14</td>
</tr>
<tr>
<td>coding standards</td>
<td>225, see also guidelines</td>
</tr>
<tr>
<td>cohesion</td>
<td>283, 338</td>
</tr>
<tr>
<td>collaboration diagram</td>
<td>276</td>
</tr>
<tr>
<td>collect.c, sample</td>
<td>246</td>
</tr>
<tr>
<td>Column.java, sample</td>
<td>316, 317</td>
</tr>
<tr>
<td>.com, file extension</td>
<td>227</td>
</tr>
<tr>
<td>comma operator</td>
<td>35</td>
</tr>
<tr>
<td>commands.c, sample</td>
<td>122</td>
</tr>
<tr>
<td>commit</td>
<td>203</td>
</tr>
<tr>
<td>Common Object Request Broker Architecture</td>
<td>271</td>
</tr>
<tr>
<td>common, directory name</td>
<td>183</td>
</tr>
<tr>
<td>compile, directory name</td>
<td>183</td>
</tr>
<tr>
<td>compiler</td>
<td>360</td>
</tr>
<tr>
<td>compiler driver</td>
<td>323</td>
</tr>
<tr>
<td>complete.c, sample</td>
<td>35</td>
</tr>
<tr>
<td>complex numbers</td>
<td>309</td>
</tr>
<tr>
<td>component</td>
<td>180, 325</td>
</tr>
<tr>
<td>Component Configurator, design pattern</td>
<td>336</td>
</tr>
<tr>
<td>compress, program</td>
<td>227</td>
</tr>
<tr>
<td>compress.c, sample</td>
<td>107</td>
</tr>
<tr>
<td>computer networks</td>
<td>338</td>
</tr>
<tr>
<td>concurrent systems</td>
<td>289</td>
</tr>
<tr>
<td>Concurrent Versions System, see CVS</td>
<td>336</td>
</tr>
<tr>
<td>condition</td>
<td>28</td>
</tr>
<tr>
<td>conf.c, sample</td>
<td>92, 246</td>
</tr>
<tr>
<td>conf, directory name</td>
<td>183</td>
</tr>
</tbody>
</table>
conference publications, 259
config.cache, file name, 199
config.h.in, file name, 200
config.h, file name, 199, 200, 226
config.h, sample, 199
config.log, file name, 199
config.status, file name, 199
config, sample, 211, 215
config.h.Sh, file name, 226
configuration, 197, 210
configure, file name, 199, 226
confpars.c, sample, 334
conjunction, 41
connected, 139
Connector, design pattern, 337
connector, 278
connector, sample, 278
conservative garbage collector, 88
const, keyword, 64, 360
constant
  array size, 113
documentation, 40
in assertion, 219
in conditional expression, 360
naming of, 234
on left side of comparison, 52
preprocessor defined, 361
constructor, see class, constructor
ContainerBase.java, sample, 151, 286
containers, 318
context diff, 209, 331, 355
ContextConfig.java, sample, 152
continue, Java keyword, 38, 39, 152
continue, keyword, 37, 55, 96
cntrib, directory name, 183
control flow, 143
control flow statements, 25
control model, 156, 285
  boss-worker, 157
event-driven, 285
multiple process, 159
mutual exclusion, 158
state transition, 291
system manager, 289
work crew, 157

conventions, see guidelines
convolve.c, sample, 104
Cookies, directory name, 325
copy constructor, 303
Copying, file name, 226
copyright, 20, 413
CORBA, 271, 325, 329, 337
corruption, 168
cos, C library, 388
coupling, 284, 338
CPAN.pm, sample, 299
cpp, program, 323
.cpp, file extension, 227
CPP, identifier name, 193
crc24.h, sample, 306
crc32.h, sample, 300
CreateEvent, Win32 SDK, 163
CreateMutex, Win32 SDK, 163
CreateThread, Win32 SDK, 162, 163
Critic.java, sample, 299
critical region, 163
cs4231reg.h, sample, 256
.cs, file extension, 227
cscope, program, 365
csh, program, 227
.csh, file extension, 227
ctags, program, 343
ctags.1, sample, 252
ctags.c, sample, 125
ctags.pdf, sample, 252
crtime, C library, 389
ctype, C library, 42
cur, Hungarian prefix, 237
currentThread, Java method, 163
curses.c, sample, 71
cursor, 65, 102
cut.1, sample, 260
cVS, 5, 50, 203, 208, 224, 268, 355
cVS, directory name, 183
cw, Hungarian prefix, 236
CWEB, 263, 266
.cxx, file extension, 227
cycle, in graph, 131
Cygwin, 376
D
%D%, revision-id tag, 206
d, Hungarian prefix, 236
daemon, 216, 324
data
dictionary, 327
element access, 65
flow, 324
grouping, 75
link, 336
organization, 76
repository, 325
structure, 95
structure, dynamic allocation, 62
structure, linked, 62
type, 61, 105, 106, 318
data-flow architecture, 273
data-flow diagram, 275
database triggers, 285
Database.java, sample, 235, 395
DatabaseInformation.java, sample, 228
$Date: $, revision-id tag, 206
date.c, sample, 231
$Date$, revision-id tag, 206
date, Hungarian prefix, 237
Date, Java class, 387, 396
db, 256
db.h, sample, 91
db, sample, 319
DB, Unix-specific function, 319
db_load.c, sample, 334
db_update.c, sample, 247
dbopen, Unix-specific function, 319
dCOM, 271, 337
.Dd, troff command, 262
De Morgan’s rules, 41
dead code, 372
debug level, 217
Debug, directory name, 183
DEBUG, identifier name, 216, 222
derbugger, 373
debugging output, 216
decomposition, 180
Decorator, design pattern, 337
deep copy, 303
def, file extension, 227
default, keyword, 32
DefaultContext.java, sample, 298
defensive programming, 33
deficiencies, 252
defs.h, sample, 91, 172
del, Unix-specific function, 319
delele, keyword, 167, 168, 302, 303
deliver.c, ex, 246
demoGL, sample, 400
depend, pseudo-target, 194
dependency
graph, 189
isolation, 198
relationship, 13
@deprecated, javadoc tag, 263
deque, 121
derived class, see class, derived
design pattern, 331, 336, 337
design specification, 242
Desktop, directory name, 325
DESTROY, Perl keyword, 310
destructor, see class, destructor
/dev/null, file name, 350
Developer Studio, see Microsoft Developer Studio
development process standards, 239
device driver, 155
dgl_dllstartupdialog.cpp, sample, 286
dhcp, sample, 268
diagram
and architecture modeling, 278
ASCII, 256
class, 276, 277
collaboration, 276
data-flow, 275
design, 2
for understanding code, 375
in practice, 337
modeling hierarchy, 283
object, 276
reverse engineering, 278
state transition, 247, 291
UML, 12, 276
dialog box, 239
diff, program, 51, 355, 357
   -b, 356
   -c, 209
   -i, 356
   -w, 51, 356
differences between files, 355
dig.c, sample, 76
   .digit, 227
direct access to memory, 74
directional, 139
directory name
   Cookies, 325
   Desktop, 325
doc, 248
   Documents and Settings, 325
   Favorites, 325
   include/protocols, 269
   My Documents, 325
   net, 186
   src, 181
   Start Menu, 325
   see also Table 6.1, 183
DirHandle.pm, sample, 309
disjunction, 41
disp–asc.c, sample, 212
disp–asc.out, 212
dispatch, 306
Distributed Component Object Model, 271
ditroff, program, 266
division, 53
DLL, 322, 323, see also shared library
DLL hell, 322
DLL.pm, sample, 311
   .d11, file extension, 227
dlutils.c, sample, 323
DNS, 247
do, keyword, 51, 174
doc, directory name, 183, 248
doc, pseudo-target, 194
doc, sample, 248, 402
DocBook, 261, 265, 266
doclets, 265
docmd.c, sample, 120
   {@docRoot}, javadoc tag, 263
documentation, 183, 214, 227, 241
   algorithms, 245
   bugs, 251
   change, 252
   deficiencies, 252
   identifiers, 246
   interfaces, 249
   non-functional requirements, 247
   overview, 243
   requirements, 247
   specifications, 244
   system structure, 245
   test cases, 249
   tools, 214
Documents and Settings, directory name, 325
doexec.c, sample, 222
domain name system, 247
domain-specific
   architectures, 333
   languages, 181, 212, 330
   protocols, 272
   tools, 210
domain.h, sample, 78
Double Checked Locking, design pattern, 337
double-ended queue, 121
doubly linked list, 121
down–1st, editor command, 346
Doxygen, 261
dry-run, 223
   .dsp, file extension, 227
   .dsw, file extension, 227
   .dt, Hungarian prefix, 237
   .Dt, troff command, 262
dump, program, 269
Dump.h, sample, 308
dumpbin, 363
dumpstore.h, sample, 269
   .Dv, troff command, 262
   .dvi, file extension, 227
dw, Hungarian prefix, 236
dynamic
   allocation of data structures, 62
   dispatch, 180, 306
link library, see DLL
linking, 322
memory allocation, 84
memory allocation pool, 168
shared objects, 322

E
%EX%, revision-id tag, 206
EBCDIC, 212
echo.c, sample, 20
Eclipse, 377
ed, program, 170, 355
ed, sample, 171
edge, 62, 125, 131, 134, 137
editor, 343
editor command, 343
edu, directory name, 183
eg, directory name, 183
egrep, program, 161
Eiffel, 126, 276, 296, 300
.el, file extension, 227
e1f.c, sample, 368
e1f2aout.c, sample, 22
Emacs, 227, 229, 340, 344, 345, 365
@emph, Texinfo command, 263
empty statement, 38
encapsulation, 320
.encoding, 227
@end, Texinfo command, 263
endian.h, sample, 92
genie.c, sample, 173, 218
enum, keyword, 234
evelope.c, ex, 246
EOF, C library, 24
Epsilon, 340
eqn, program, 227
.eqn, file extension, 227
eqauls, Java method, 23
.errno.c, sample, 320
error messages, 215, 360
Error, Java class, 150, 151
Errors, manual section, 223
/etc/inetd.conf, file name, 325
etc, directory name, 183
EtherExpress, 76
Ethernet, 279
European Conference on Pattern Languages of Programming, 338
eval.c, sample, 31, 125
event loop, 286
event pump, 286
event-driven architecture, 285
EventObject, Java class, 289
@example, Texinfo command, 263
exception, 36, 98, 150
Exception, Java class, 150
@exception, javadoc tag, 263
exclusive range, 100
exclusive-or, 114
exec, Unix-specific function, 281
execution profiler, 372
execve.pdf, sample, 223
execve, Unix-specific function, 222, 223
exercises, 13
exit, C library, 24, 36, 43, 168
ExitThread, Win32 SDK, 163
expand.c, sample, 25
expat-lite, sample, 322
exponent, 83
export, 299
Exporter, Perl module, 299
expr.c, sample, 130, 149
expression
  boolean, 39
  character, 39
  integer, 51
  pointer, 65, 72
ext2, 281
ext2fs_readwrite.c, sample, 281
ext2fs_vnops.c, sample, 282
ext2fs_write, 281
ExtendableRendererFactory.java, sample, 332
extended linear hashing, 319
Extension Interface, design pattern, 337
extern.h, sample, 92
extern, keyword, 363
External Polymorphism, design pattern, 337
extra-functional property, 269
extreme programming, 8, 10, 17, 378
exuberant ctags, 344
Eye.c, sample, 231

F
%F%, revision-id tag, 206
f771, program, 324
f, Hungarian prefix, 236
fail, C++ library, 24
FALLTHROUGH, 32, 231
FAQ, 260
fast file system, 243
FastDateFormat.java, sample, 220
fault isolation, 323
Favorites, directory name, 325
ferror, C library, 24
ffs.pdf, sample, 243
ffs, sample, 243
ffs_vnops.c, sample, 313
fgets, C library, 99
fgrep, program, 352
field separator, 275
field, class, see class, field
field, table, 101
FIFO, 77, 107
FigEdgeModelElement.java, sample, 302
file differences, 355
file extension
.h, 225
.info, 265
.man, 227
.texi, 265
see also Table 7.2, 227
file name
/dev/null, 350
/etc/inetd.conf, 325
ChangeLog, 226, 252
Changes, 226
config.cache, 199
config.h.in, 200
config.h, 199, 200, 226
config.log, 199
config.status, 199
config.h.SH, 226
configure, 199, 226
Copying, 226
INSTALL, 226
LICENSE, 226
Makefile.in, 200
Makefile.SH, 226
Makefile, 137, 200, 223, 226
MANIFEST, 226
NEWS, 226
NUL, 350
patchlevel.h, 226
README, 226
tags, 343
TODO, 226
version.h, 226
file.h, sample, 79
@file, Texinfo command, 263
filename, 225
filter, 161, 273, 323, 324
final state, 291
finally, Java keyword, 302
finally, Java keyword, 150, 152, 153
find, 14, 353
find.c, sample, 125
First, Hungarian prefix, 236
fix list, 252
fixed-width, 275
FIXME, 233
FIXME, identifier name, 354
.Fl, troff command, 264
floating-point numbers, 97
floppy distribution, 4
flow, 376
fn, Hungarian prefix, 236
folding, 345
for, Java keyword, 38
for, keyword, 34, 96, 118
Fork.C, sample, 166
fork, Unix-specific function, 160, 166, 281
formal practices, 180
formatting, 230
Fortran, 16, 42
fortunes2, sample, 457
forward list traversal, 119
Index

forward-list, editor command, 346

Foundation Classes, see Microsoft
  Foundation Classes

fpu_sqrt.c, sample, 341
fractions, 309
FrameMaker, 227, 260
framework, 328, 329
fread, C library, 98
free memory, 87
free, C library, 85, 89
FreeBSD, 89
FreeBSD documentation project, 261
frexp.c, sample, 83
friend, keyword, 305, 307
.frm, file extension, 227
frm, Hungarian prefix, 237
FROM, Modula keyword, 299
fsplit.c, sample, 34
ftp, program, 7
FTP, 269
ftpd.c, sample, 69
func.c, sample, 53, 296
function, 25, 26
  declaration, 25
  pointer, 67
  return values, 76
function.c, sample, 213
function.h, sample, 213
functional description, 242
functionality addition, 7
Future_Set.h, sample, 305
fwrite.c, sample, 280
fwrite, C library, 98, 280
fxp_intr, 282

G

g substitute flag, 348
g++, program, 323
%G%, revision-id tag, 206
garbage collector, 87, 88, 303
gcc, program, 195, 372
  -a, 372
GCManager.c, sample, 57
gencode.h, sample, 132
generalization relationship, 13, 277
generic code, 313
generic implementation, 313
GENERIC, sample, 211
get, in method name, 304
get, Unix-specific function, 319
GetCurrentThreadId, Win32 SDK, 163
getenv, Unix-specific function, 201
getopt, Unix-specific function, 22, 24, 28,
  238, 264, 273
getopt_Long, Unix-specific function, 238
getProperty, Java method, 387
getpwent.c, sample, 72
gets, C library, 98, 99
getservent.c, sample, 44
gmtime, Java method, 387
Glimpse, 354
global variable, 25
GNATS, 209
GNU, xxv, 4, 8, 14, 22, 89, 225, 261
GNU C compiler, 195, 364, see also gcc
gnu getopt, Java package, 22
good, C++ library, 24
goto, 43, 169
goto, keyword, 43, 55, 159
GoToMatchBraceExtend, editor command, 346
GoToMatchBrace, editor command, 346
gperf, program, 141
gprof, program, 134, 372, 378
gprof.c, sample, 69
gprof.h, sample, 134
gprof.pdf, sample, 246
grammar, 129, 144, 147, 177, 191
graph, 95, 131
graphics algorithm, 105
GraphViz, 12
GregorianCalendar, Java class, 382, 386,
  390
grep, program, 14, 264, 344, 346, 364, 398
  -e, 354
  -i, 354
  -l, 348
  -n, 354
  -v, 351
groff, program, 365
grok, 255
gtkdiff, program, 357
GUI, 236, 274, 285
guidelines
  BSD, 50, 226
coding, 225
data interchange, 129
development process, 239
deviations from, 339
formatting, 230
GNU, 50
indentation, 367
identifier names, 234
indentation, 228
Java code, 226
naming files, 225
portability, 237
  programming practices, 237
  .gz, file extension, 227
gzip, program, 162, 227
$Header: $, revision-id tag, 206
history.c, sample, 52
homedir.c, sample, 86
Host.java, sample, 298
hostctlr.h, sample, 213
hp71c2k.c, sample, 222
.hpp, file extension, 227
hsq1.html, sample, 380
hsq1db, program, 249
hsq1db, sample, 400
hsq1Syntax.html, sample, 397
HTML, 28, 214, 249, 261, 265, 358, 359
HTTP, 244, 269, 271
http_core.c, sample, 271
http_protocol.c, sample, 244, 270
Hungarian naming notation, 225, 235
hunt, 149
hunt.c, sample, 147
Hypersonic SQL database engine, 249
H
  %H%, revision-id tag, 206
  .h, file extension, 225, 227
  h, Hungarian prefix, 236
  hack.objnam.c, sample, 40
  Half-Sync/Half-Async, design pattern, 333, 337
  handler.c, sample, 286
  hardware interrupt, 286
  hash, 309
  hash function, 114
  hash table, 113
  hash.c, sample, 229
  Hashable, Java class, 392, 395
  Hashtable, Java class, 319
  head, 107, 117
  $Header: $, revision-id tag, 206
  header files, 20, 183, 226
  $Header: $, revision-id tag, 206
  headers.c, sample, 246
  heap, 89
  help file, 190, 239
  hidden structures, 139
  hierarchical decomposition, 282
I
  i82365.c, sample, 282
  %I%, revision-id tag, 206
  i, Hungarian prefix, 236
  .I, troff command, 262
  IBM, 227
  IBM 3270, 212
  IBM VM/CMS, 212
  .Ic, troff command, 262
  ICMP, 101
  ico.c, sample, 102, 157
  .ico, file extension, 227
  .icon, file extension, 227
  $Id$, revision-id tag, 206
  IDE, 198
  idealized presentation, 254
  ident, program, 207
  identd.c, sample, 66, 73
  identifier name
    $*$, 194
    $<$, 194
    $>$, 194
    $?$, 194
    @, 194
    $$, 194
argv, 20
buf, 96
CC, 193
CFLAGS, 193
CPP, 193
DEBUG, 216, 222
FIXME, 354
IN, 64
INCLUDES, 193
INSTALL, 193
left, 126
LFLAGS, 193
LIBS, 193
NDEBUG, 220
new, 63
next, 118
NULL, 20, 35, 42, 60, 118
OBJS, 193
OUT, 64
prev, 121
right, 126
SHELL, 193
SRCS, 193
STREQ, 22
usage, 26
xmalloc, 85
XXX, 233, 354

 identifiers, 246
IDL, 261, 273
.idl, file extension, 227
idutils, 344
if..arp.c, sample, 234
if..atm.c, sample, 64
if, keyword, 23, 55, 360
if.cs..isa.c, sample, 230
if..ed.c, sample, 96
if..fxp.c, sample, 282
if..fregexp.h, sample, 76, 77
IIS, see Microsoft Internet Information Server
imake, 190, 195, 212
import, Java keyword, 299
IMPORT, Modula keyword, 299

 IN, identifier name, 64
in..proto..c, sample, 113
inbound..c, sample, 213
include files, see header files
include/protocols, directory name, 269
include, directory name, 183
INCLUDES, identifier name, 193
inclusive range, 100
inconsistent, 231
indent, program, 231, 367
indentation, 228
index.html, sample, 249, 379
inetd, program, 327
inetd.conf, sample, 327
infblock..c, sample, 125
.info, file extension, 227, 265
information hiding, 296
information-hiding, 304
inheritance, 13, 180, 306
inheritance hierarchy, 300
.ini, file extension, 202
init..c, sample, 36
init, Java method, 20
init..main..c, sample, 290
initial state, 291
initialization files, 202
inline, keyword, 177
inode, 114
inspections, 9
INSTALL, file name, 226
INSTALL, identifier name, 193
install, pseudo-target, 194
installation instructions, 242
instance, variable, see class, property
integer expression, 51
integer..h, sample, 309
integrability, 269
integrated development environment, see IDE
Intel, 76, 98
Interceptor, design pattern, 337
interface, 13, 106, 108, 317
see IDL, 273
interface, Java keyword, 277
INTERFACE, Modula keyword, 299
interfaces, 249
intermediate files, 324
internal representation, 82
International Conference on Pattern Languages of Programming, 338
Internet Information Server, see Microsoft Internet Information Server
Internet Worm, 141
interpreter state, 336
interrupt, 105, 155, 156, 159
INTR, 281
intro, sample, 245
introductory guide, 242
invariant, 56
IOCCC, 361
ioconf.c, 211
ioctl, Unix-specific function, 80
iostream, 401
IP, 269
.IP, troff command, 262
IPX, 78
.IR, troff command, 262
ISA, Perl keyword, 311
isapnpres.c, sample, 125
isLeapYear, Java method, 386
ISO, 186
isolation of dependencies, see dependency isolation
issue-tracking database, 259
isupper, C library, 42
isUpper, Java method, 42
.I, troff command, 262
@item, Texinfo command, 263
Iterator, design pattern, 337

J
Jade, 265
jam, program, 196
jamfile, 196
.jar, file extension, 227
JasperLogger.java, sample, 162
Java, 42, 98, 276, 365
API, 214
SDK, 214
Java class
Calendar, 382, 390
Character, 42
Date, 387, 396
Error, 150, 151
EventObject, 289
Exception, 150
GregorianCalendar, 382, 386, 390
Hashable, 392, 395
Hashtable, 319
java.lang.Exception, 150
RuntimeException, 151
Stack, 319
String, 20
System, 201
TestCase, 220
Throwable, 150
Thread, 162, 163
Vector, 319
Java interface
Runnable, 162
Java keyword
abstract, 276, 306
break, 38, 39, 152
catch, 150, 151, 152
class, 300
continue, 38, 39, 152
finalize, 302
finally, 150, 152, 153
for, 38
implements, 277, 301
import, 299
interface, 277
package, 283, 298
private, 226, 276, 304
protected, 226, 276, 304
public, 226, 276, 304
return, 152
static, 276, 303, 332
synchronized, 163, 164, 165
this, 302, 309
throws, 151, 154
try, 150, 152, 153
see also keyword
Java method
assert, 220
checkError, 24
currentThread, 163
equals, 23
getProperty, 201
getTime, 387
init, 20
isLeapYear, 386
isUpper, 42
length, 20
main, 220
notify, 163, 164
run, 162
setUp, 220
sleep, 163
start, 162
stop, 163
suite, 220
tearDown, 220
toLowerCase, 42
wait, 163, 164
yield, 163
Java operator
see also operator
>>> 53
Java package
gnu.getopt, 22
java.util, 80, 319
Math, 388
Java server page, 325
Java Virtual Machine, 279
java.util, Java package, 80, 319
.java, file extension, 227
JavaBeans, 325
Javac.java, sample, 30
javadoc, 214, 231, 261, 263, 265, 382
JCT, 227
.jcl, file extension, 227
jdbcConnection.java, sample, 304
jdbcDatabaseMetaData.java, sample, 305
jdbcPreparedStatement.java, sample, 214
JDBCStore.java, sample, 150
jobs, shell command, 146
jobs.c, sample, 120, 146, 147
join, 203
jot, program, 374
journal publications, 259
JSP, 325
.jsp, file extension, 227
JspReader.java, sample, 38, 153
Jt4, sample, 401
JUnit, 220, 224
$JustDate: $, revision-id tag, 206
JVM, see Java Virtual Machine

K
K&R C, 22
kbd.out, 213
kern_descric.c, sample, 230
kern_1km.c, sample, 322
kern_synch.c, sample, 256
kernel modules, 322
key.c, sample, 112, 129
keyword
break, 32, 36, 37, 55, 96, 231
case, 32, 219, 231
const, 64, 360
continue, 37, 55, 96
default, 32
delete, 167, 168, 302, 303
do, 51, 174
enum, 234
extern, 363
for, 34, 96, 118
friend, 305, 307
goto, 43, 55, 159
if, 23, 55, 360
inline, 177
namespace, 283, 293, 297
new, 168, 302, 303
operator, 307
pragma, 360
private, 304
protected, 304
public, 304
return, 36, 159, 169
sig_atomic.t, 168
static, 26, 66, 294, 296, 303, 363
struct, 62, 75, 92, 101, 145
switch, 32, 212, 231, 360
template, 315
this, 302, 309
throw, 154
typedef, 91, 92, 296
union, 80, 145
using, 298
virtual, 306
void, 317
volatile, 77, 168
while, 30, 174, 360
keywords, 205

L
%L%, revision-id tag, 206
.1, file extension, 227
1, Hungarian prefix, 236
label, 43
language, 15, 129
block-structured, 105
markup, 105
.language-code, 227
large integers, 309
large projects, 179
laser printer, 264, 375
last, program, 255
last.1, sample, 255
last.pdf, sample, 255
Last, Hungarian prefix, 236
last, Perl keyword, 37
LaTeX, 228, 261, 265, 266, 359
Law of Demeter, 338
layered architecture, 279
1b1, Hungarian prefix, 237
Id, program, 323
LDAPTransformer.java, sample, 345
Leader/Followers, design pattern, 337
left, identifier name, 126
length, Java method, 20
Leve1Down, editor command, 346
Leve1Up, editor command, 346
lex, program, 194, 227
lex.c, sample, 119, 122, 291
lexical analysis, 333
LFLAGS, identifier name, 193
1ib, directory name, 183
.1ib, file extension, 227
1ibc, sample, 320
libcrypt, sample, 320
libcurses, sample, 320
libedit, sample, 320
libkvm, sample, 320
libpcap, 132
libpcap, sample, 320
library, 180, 227, 319
Library.html, sample, 397
Library.java, sample, 381, 382, 392, 393, 395
LIBS, identifier name, 193
LICENSE, file name, 226
license, 20, 413
LifecycleException.java, sample, 151
LITE, 105
Like.java, sample, 317
Lim, Hungarian prefix, 236
line counting, 372
@1ink], javadoc tag, 263
linked data structure, 62
linked list, 95, 117, 167, 335
lint, 33, 129, 231, 361, 377
1nt1.h, sample, 129
Linux, xxv, 4, 8, 14, 183, 225
Linux documentation project, 266
Lisp, 227
1ist.c, sample, 107
listing, 360, 361
literar programming, 17, 266
localtime.c, sample, 111, 112
locate, program, 186
lock, 203
$Locker$, revision-id tag, 206
$Log$, revision-id tag, 206
log4j, 216
$Log$, revision-id tag, 206
$Logfile$: revision-id tag, 206
logging output, 216
logical AND, 41
logical OR, 41
longjmp, C library, 168, 169
lookup table, 111, 212
loop, 28, 34, 37, 51, 54
loop invariant, 56, 98
lorder.sh, sample, 259
lower bound, 100
lpd, sample, 268
lpr, program, 160
ls.c, sample, 32, 90
ls.h, sample, 90
lsearch.c, sample, 317
lst, Hungarian prefix, 237
LXR, 366
M
-M
gcc option, 195
M--., editor command, 343
M-x outline-mode, editor command, 345
m4, program, 227, 256
.m4, file extension, 227
%M, revision-id tag, 206
Mac, Hungarian prefix, 236
machdep.c, sample, 23, 74
Macintosh, 98
macro, 104, 172
macro.c, ex, 246
main program and subroutine, 268
main.c, sample, 42, 170, 286
Main.java, sample, 55
main, C library, 20, 219
main, directory name, 183
main, Java method, 220
maintenance
branch, 203
code beautifier, 367
cost, 17
documentation, 252
formatting changes, 50, 367
management, 17
organization, 183
reason for reading code, 6
.mak, file extension, 227
make, program, 137, 139, 186, 192, 224
-n, 196
make.h, sample, 137, 139
makedepend, program, 212
makedepend, sample, 212
makefile, 192
Makefile.in, file name, 200
Makefile.nt, sample, 137, 192
Makefile.SH, file name, 226
Makefiletmpl, sample, 195, 240
makefile.win, sample, 196
Makefile, file name, 137, 200, 223, 226
Makefile, sample, 198, 223, 240
makewhatis, program, 274
makewhatis.sh, sample, 161, 274
malloc.c, sample, 80, 81, 173
malloc, C library, 80, 84, 85, 88, 101
man, program, 28, 260, 265
-t, 265
man, directory name, 183
.man, file extension, 227
MANIFEST, file name, 226
mantissa, 83
manual page, 183, 190
map, 95, 111, 113
MapStackResolver.java, sample, 39
marketing material, 260
markup language, 260
marshalling, 273
Martian, 256
master/slave, 157
math.c, sample, 106
Math.PI, Java method, 388
Math, Java package, 388
mathematical theorems, 3, 17
matrix, 95, 101, 309
mbuf.9, sample, 249
mbuf.pdf, sample, 249
md2.h, sample, 306
mdef.h, sample, 256
mdoc, macro package, 260
me, macro package, 227
.me, file extension, 227
member function, see class, method
member variable, see class, property
memcpy, C library, 97, 98
memmove, C library, 97
memory access, 74
memory leak, 85, 87, 303
memory management, 155
memset, C library, 97
Menu.cc, sample, 364
message, 300
meta-characters, 340
method, see class, method
MFC, see Microsoft Foundation Classes
Microsoft
  Access, 357
  C/C++ compiler, 364, see also cl, program
  C#, see C#
  Developer Studio, 227
  DOS, see MS-DOS
  Foundation Classes, 286, 329
  Internet Information Server, 327
  Macro Assembler, 361
  MSDN, 28, 327
  .NET platform, 15, 271
  OLE automation, 359
  SDK element, see Win32 SDK
  Visual Basic, 227, 235, 277, 300, 304, 325, 344, 357
  Visual Basic for Applications, 235
  Visual Source Safe, 203
  Visual Studio, 346, 356, 372
  Windows, 8, 20, 160, 189, 190, 202, 216, 323, 370, 376
  Windows Explorer, 186
  Windows Installer, 190, 240
  Windows NT, 137, 227
  Windows SDK, see Win32 SDK
  Windows SDK source, 235
  Windows Services for Unix, 377
  Windows, resource files, 183
  Word, 260, 345, 359
middleware, 271
midnight commander, program, 186
MIF, 265
  .mif, file extension, 227
  Min_Max.h, sample, 315
  minCurve.c, sample, 103
  minurbs.c, sample, 103
misc.c, sample, 85, 166
mivaltree.c, sample, 125
  .mk, file extension, 227
mkastods.c, sample, 212
mkastosc.c, sample, 213
mkdep, program, 195
mkdsstoas.c, sample, 212
mkhits.c, sample, 213
MKS Toolkit, 377
mktables.PL, sample, 37
mm, macro package, 227
  .mm, file extension, 227
MMDF, 160
mn, Hungarian prefix, 237
mod_so.c, sample, 323
Model-View-Controller, 329
modifications, 7
$Modtime: $, revision-id tag, 206
Modula, 293
Modula keyword
  AS, 299
  FROM, 299
  IMPORT, 299
  INTERFACE, 299
  MODULE, 299
  Modula-3, 299
  module, 293, 297
  MODULE, Modula keyword, 299
  modulo division, 53
Monitor Object, design pattern, 337
more, sample, 11, 168
Motif, 286, 329
mount_nfs.c, sample, 336
move.c, sample, 216
mp, Hungarian prefix, 236
MS-DOS, 77, 227, 370
MSDN, see Microsoft MSDN
msdosfs_write, 281
msdosfs_vnop.c, sample, 281
muldi3.c, sample, 245
multiplayer games, 268
multiple inheritance, 277
multiple precision floating-point numbers, 309
mutithread.c, sample, 162
mutithread.h, sample, 162
mutithreaded, 168
mutex, 159
mutex_clear, Unix-specific function, 162
mutual exclusion, 159
mutual recursion, 147
mv.c, sample, 87
MVC, 329
My Documents, directory name, 325

N

n, Hungarian prefix, 236
name mangling, 364
$Name$, revision-id tag, 206
named, program, 126
namespace, 296
namespace pollution, 296
namespace, keyword, 283, 293, 297
ncr.intr, 282
ncr.c, sample, 282
ncr5380sbc.c, sample, 44
NDEBUG, identifier name, 220
ne2100.c, sample, 218
.NET, see Microsoft .NET
net, directory name, 186
net, sample, 337
netatalk, sample, 186
NETBIOS, 279
NetBSD, xxvi, 183
netbsdsrc, sample, 400
netinet, sample, 186, 337
netiso, sample, 186, 337
netnatm, sample, 186, 337
netpbm, package, 275
network, 336
Network File System, 272
Network Information Center, 269
Network Information System, 272
network interface, 76
network time protocol, 240
new, identifier name, 63
new, keyword, 168, 302, 303
new, Perl identifier, 309
NEWS, file name, 226
Next, Hungarian prefix, 236
next, identifier name, 118
next, Perl keyword, 37
NFS, 272
nfs.unops.c, sample, 281
nfbsd, sample, 268
nfsspec.write, 281

N11, Hungarian prefix, 236
NIS, 272
nm, program, 363
.Nm, troff command, 262
nmake, program, 192
node, 62, 125, 131
@node, Texinfo command, 263
nodetypes, sample, 145
noise, 350
$NoKeywords:$, revision-id tag, 206
non-function requirements, 247
non-software tools, 375
nonexistent, sample, 223
notify, blackboard operation, 270
notify, Java method, 163, 164
NOTREACHED, 232, 291
.nr, file extension, 227
ns_validate.c, sample, 84
NTP, 240
ntp_io.c, sample, 201
NULL, file name, 350
NULL, C library, 52, 236
NULL, identifier name, 20, 35, 42, 60, 118
nullfs, sample, 330
NumberGuessBean.java, sample, 325
numguess.jsp, sample, 325
nvi, program, 294, 365

O
.o, file extension, 227
obj, directory name, 183
.obj, file extension, 227
object, 78, 300
browser, 300
code, 319, 360, 362
diagram, 276
request broker, 271
see also class
Object Lifetime Manager, design pattern, 337
Object Management Group, 271
object-oriented, 78, 180
object-oriented architecture, 275
OBJJS, identifier name, 193
Observer, design pattern, 337
off-by-one errors, 100, 141
ofisa.c, sample, 89
ofw.c, sample, 159
.ok, file extension, 227
OLE automation, see Microsoft OLE automation
OMG, 271
on-line messaging, 268
.Op, troff command, 262, 264
open range, 100
open-source software, xxv, 3, 16, 20
as scientific communication vehicle, 3
contributing, 5, 260
languages used, 15
reorganising code, 46
searching, 260
source browsers, 365
see also individual project names
open, Unix-specific function, 281
open1.h, sample, 306
OpenCl, sample, 401
opendir, Unix-specific function, 35, 319
OpenJade, 265
OpenNt, 376
operating system, 155, 338
specific code, 183
operation, see class, method
operator
||, 41
., 35
->, 102
/**/, 176
<=, 53, 308
=, 52
=, 53, 52
>>, 53, 58, 308
?:, 46, 174
##, 176
&&, 22, 41
&, 53, 64
sizeof, 73, 85, 97
operator overloading, 180, 305, 307
operator, keyword, 307
opt, Hungarian prefix, 237
optimize.c, sample, 133
options.c, sample, 201, 312
or, logical, 41
ORB, 271
org, directory name, 183
os.cpp, sample, 315
os/2, 193, 227
os/32, 377
os2thread.h, sample, 162
os, directory name, 183
.os, troff command, 262
osi, 336
OUT, identifier name, 64
outline mode, 345
output.c, sample, 70
outwit, package, 377
overview, 243

P
%P%, revision-id tag, 206
p, Hungarian prefix, 236
.Pa, troff command, 262
package, 300
package, Ada keyword, 299
package, Java keyword, 283, 298
package, Perl keyword, 299, 309
packaging abstraction, 292
abstract data type, 318
component, 325
data repository, 325
filter, 323
generic implementation, 313
library, 319
module, 293
namespace, 296
object, 300
process, 323
see also the individual terms
page.h, sample, 256
pair programming, 10
parallelism, 154
@param, javadoc tag, 263
parse tree, 129, 334
parse.c, sample, 122, 124, 216
parseaddr.c, ex, 246
parser generator, 130, 228
Parser.java, sample, 334
parsing, 129, 147, 333
Pascal, 42, 293
patch, program, 137
patchlevel.h, file name, 226
pattern, see design pattern
pax.c, sample, 43
pax.h, sample, 312
pb, Hungarian prefix, 237
pcic.intr, 282
PclText.c, sample, 104
PDF, 261
perfect hash function, 141
Perkin-Elmer, 377
Perl, 42, 98, 201, 214, 227, 276, 300, 309, 330, 348, 349, 377
perl
-e, 349
-i, 349
-p, 349
inheritance, 311
Perl classes, 309
Perl identifier
new, 309
self, 309
Perl keyword
bless, 300, 309
DESTROY, 310
ISA, 311
last, 37
next, 37
package, 299, 309
use, 299
see also keyword
Perl module
Exporter, 299
Perl operator, see operator
perl, sample, 401
perlbug.PL, sample, 224
perlguts.pod, sample, 249
perlguts.pod, sample, 249
pgp.s2k.cpp, sample, 306
pgp.s2k.h, sample, 306
physical, 336
physical boundary, 293
pl.c, sample, 38
pic, program, 227
.pic, file extension, 227
pic, Hungarian prefix, 237
pickmove.c, sample, 97, 125
ping.c, sample, 64
pipeline, 157, 161, 274, 324
pipes-and-filters architecture, 273
pippen.pl, sample, 309
pk.subr.c, sample, 256
.pl, file extension, 227
.pm, file extension, 227
pmap.c, sample, 218
PMC.Ruser.cpp, sample, 368
.png, file extension, 227
pod, documentation format, 214
.pod, file extension, 227
pod, sample, 214
pointer, 61, 62, 84, 102
aliasing, 70
and strings, 72
to function, 67
poll, 156, 286
poll, Unix-specific function, 80
polymorphic functions, 316
polymorphism, 79
pon.c, sample, 383, 384, 386, 387, 389
pop, 105
POP-3, 269
portability, 97, 98, 237, 360
Makefile, 196
POSIX, 15, 201, 280, 319
postcondition, 218
postincrement operator, 315
PostInputStream.java, sample, 300
postp.me, sample, 246
Postscript, 228, 261, 265, 359
Power PC, 98
pr.c, sample, 102
practices, 237
formal, 180
pragma, keyword, 360
precondition, 218
Index 481

preen.c, sample, 62
prep, program, 372
preprocessor, 172, 181, 296, 315, 350, 360, 361, see also header files
presentation, 336
pretty-printer, 359, 368
prev, Hungarian prefix, 236
prev, identifier name, 121
print-icmp.c, sample, 101
print.c, sample, 64
printf.c, sample, 364, 374
printf, C library, 23, 73
private, 304
private, Java keyword, 226, 276, 304
Proactor, design pattern, 337
process, 155, 323
in UML diagrams, 13
process standards, 239
process.c, sample, 335
processor architecture, 183
program listing, 360, 361
program slice, 283
program state, 336
programming practices, 237
project organization, 181
ProjectBrowser.java, sample, 286
protected, 304
protected, Java keyword, 226, 276, 304
ps, program, 87, 323
.ps, file extension, 228
pseudo-target, 194, 195
.psp, file extension, 227
pthread_cond_signal, Unix-specific function, 163, 164
pthread Cond_wait, Unix-specific function, 163, 164
pthread_cond_destroy, Unix-specific function, 163
pthread_cond_init, Unix-specific function, 163
pthread_create, Unix-specific function, 162, 163
pthread_exit, Unix-specific function, 163
pthread_mutex_destroy, Unix-specific function, 162, 163
pthread_mutex_init, Unix-specific function, 163
pthread_mutex_lock, Unix-specific function, 163
pthread_mutex_unlock, Unix-specific function, 163
pthread_self, Unix-specific function, 163
pthread_yield, Unix-specific function, 163
pty.c, sample, 73
public, 304
public, Java keyword, 226, 276, 304
publications, 259
PulseEvent, Win32 SDK, 163
purenum, sample, 401
push, 105
put, Unix-specific function, 319
px_intr, 282
px.c, sample, 282
@pxref, Texinfo command, 263
.py, file extension, 228
Python, 228, 276, 357, 377
Q
qry, Hungarian prefix, 237
qsort.c, sample, 374
qsort, C library, 69, 98
qtchat, sample, 401
qualifier, 77, 235
quarks.c, sample, 158
queue, 95, 107
queue.c, sample, 246
queue.h, sample, 319
Queue.java, sample, 164
queue, Unix-specific function, 319
quicksort, 143
quot.c, sample, 101
R
%R, revision-id tag, 206
race condition, 166
radius, sample, 126
radixsort.c, sample, 60
random.c, sample, 71
range, 100
Rational Rose, 279
rayshade, package, 320
.rb, file extension, 228
.RB, troff command, 262
rc, directory name, 183
.rc, file extension, 228
rcp.c, sample, 161, 175
RCS, 50, 203, 208, 224, 268, 327, 349, 355
RCS, directory name, 183
$RCSfile$, revision-id tag, 206
regcomp.c, sample, 174, 175
regedit, program, 328
regexp.c, sample, 233
regression testing, 203, 222
regular expression, 13, 291, 322, 340, 377
["\], 341
[\ ], 341
and editors, 340
and grep, 346
building blocks, 340
class classes, 342
eliminating noise, 350
listing matching files, 348
locating definitions, 341
metacharacters, 342
Perl syntax, 344, 352
replacements, 348
starting with dash, 354
relational database, 101, 269, 327, 377, see also SQL
release, 203
Release, directory name, 183
ReleaseMutex, Win32 SDK, 163
Remote Method Invocation, 271, 337
Remote Procedure Call, 82, 271, 337
RemoteAddrValve.java, sample, 277
remove, blackboard operation, 270
ReportEvent, Win32 SDK, 216
repository, 203
RequestFilterValve.java, sample, 276
requirements, 247
requirements specification, 4, 241, 242
res, directory name, 183
.res, file extension, 228
resource files, 183
resource script, 228
return, Java keyword, 152
@return, javadoc tag, 263
return, keyword, 36, 159, 169
reuse
and architecture, 180
and code reading, 1, 9, 10
and design patterns, 331
and libraries, 319
bibliographic references, 17
example, 382
exercise, 357
file differences, 355
of architecture, 328
of data transformations, 273
of generic implementations, 318
of leaky code, 87
of search results, 348
through processes, 159
reverse engineering
modeling tools, 275, 276, 278
references, 338
slicing, 283
$Revision: $, revision-id tag, 206
revision control, 202
revision control system, 183, 203, 252, 254, 259, 263, 268, 327
$Revision$, revision-id tag, 206
rexx, 227
RFC, 269
rfc2068.txt, sample, 244
rfc793.txt, sample, 247
rg, Hungarian prefix, 236
right, identifier name, 126
ring buffer, 108
rmdir.c, sample, 368
RMI, 271, 337
rnd.c, sample, 109
rnd.h, sample, 297
.roff, file extension, 227
room.c, sample, 70
round-robin, 289
round-trip engineering modeling, 279
round.c, sample, 219
route, program, 336
route.c, sample, 336
route, sample, 336
routed, program, 269, 336
routed.h, sample, 269
routed, sample, 336
RPC, 82, 271, 337
rpc_msg.h, sample, 82
RPM, 190, 240
RTF, 261, 265, 359
Ruby, 228, 276
run-time configuration, 200
run-time tool, 370
run, Java method, 162
runnable, Java interface, 162
RuntimeException, Java class, 151
rup.c, sample, 86, 118
rusers_proc.c, sample, 91
rwhod, program, 269
rwhod.h, sample, 269
\S
Perl/sed command, 348
S/Key, 87
%\%, revision-id tag, 206
.s, file extension, 227
@samp, Texinfo command, 263
save.c, sample, 98
sbdsp.c, sample, 282
sbdsp_intr, 282
scanf, C library, 99
SCCS, 23, 50, 203, 207, 224
SCCS, directory name, 183
schema, 327
Scoped Locking, design pattern, 337
SCSI, 281
scsi_done, 281
scsi_base.c, sample, 281
sd.c, sample, 281
sdbm, sample, 322
sdstrategy, 281
@section, Texinfo command, 263
sed, program, 162, 205, 228, 335, 348, 357, 377
.sed, file extension, 228
sed, sample, 343
@see, javadoc tag, 263
seekdir, Unix-specific function, 319
selection, 23, 32, 47
Selection.c, sample, 88
selective display, 345
self, Perl identifier, 309
semantic analysis, 334
semaphore, 163
sendmail, xxv, 160, 245
sendmail.pdf, sample, 245
seq, Unix-specific function, 319
Sequence_T.cpp, sample, 252
ser.c, sample, 52
@serial, javadoc tag, 263
@serialData, javadoc tag, 263
@serialField, javadoc tag, 263
server.cpp, sample, 330
ServerConnection.java, sample, 164
Service Configurator, design pattern, 337
services, 216
servlet, 20
session, 336
set, 95, 116
set foldenable, editor command, 345
set, in method name, 304
set-selective-display, editor command, 345
set.c, sample, 46, 116
setjmp, C library, 169
setUp, Java method, 220
SGML, 261
sh, program, 227
 -c, 161
.sh, file extension, 227
.Sh, troff command, 262
.SH, troff command, 262
shallow copy, 303
.shar, file extension, 227
shared library, 202, 227, 322, see also DLL sharing, 88
shell, 146
SHELL, identifier name, 193
shift, 53, 117
short-circuit evaluation, 41, 174
shutdown, 160
sigatomic.t, keyword, 168
signal, Unix-specific function, 167
SIGCHLD, 166
.SIGCONT, 165
.SIGFPE, 165
.SIGILL, 165
.SIGINT, 165, 168
signal, 150, 165
signal handler, 165
.signal, C library, 168
.signal, Unix-specific function, 167
signature, 358
SIGSEGV, 165
SIGWINCH, 168
silence, 350
sin, C library, 388
@since, javadoc tag, 263
Singleton, design pattern, 332, 337
sizeof, operator, 73, 85, 97
skeyinit, 87
skeyinit.c, sample, 87
skipjack.h, sample, 297
Sleep(0), Win32 SDK, 163
sleep, 281
sleep, Java method, 163
sleep, Unix-specific function, 163
Sleep, Win32 SDK, 163
slicing, 6, 283
slicing criterion, 283
smail, package, 160
Smalltalk, 276, 300, 365
smart pointers, 308
smbfs, 252
SMTP, 269
smtp.c, sample, 308
SNA, 78
snake.c, sample, 75
snprintf, C library, 99
.so, file extension, 227
social processes, 3, 17
Socket.pm, sample, 312
socket, sample, 401
sockunix.c, sample, 303
software archeology, 17
software maintenance, see maintenance software process, 179
software requirements specification, see software requirements specification
Solaris, 8
SONET, 279
sort, program, 275, 352, 357, 374
source code control system, see revision control system
source code tree, 181
source examples, see example source
Source-Navigator, 366
$Source$, revision-id tag, 206
SourceForge Bug Tracker, 209
SourceForge.net, 16
sox, package, 275
spec_strategy, 281
spec_vnops.c, sample, 281
specifications, 244
spglyph.c, sample, 219
split, 203
sprintf, C library, 98, 99
Spy++, 372
SQL, 6, 269, 327
SQLTransformer.java, sample, 153
src, directory name, 181, 183
src, sample, 245, 275, 329
SRCS, identifier name, 193
stack, 89, 95, 99, 105
stack pointer, 106
Stack, Java class, 319
standard document, 259
standard template library, see STL
StandardContext.java, sample, 201
StandardLoader.java, sample, 54, 151
standards, see guidelines
StandardWrapperValve.java, sample, 217
Start Menu, directory name, 325
start, Java method, 162
state machine, 291
state transition, 291
state transition diagram, 291
$State$, revision-id tag, 206
static, Java keyword, 276, 303, 332
static, keyword, 26, 66, 294, 296, 363
stdin, C library, 51
stdio.h, sample, 297
stdio, C library, 166, 280
StdString.h, sample, 309
STL, 80, 100, 318
stop, Java method, 163
storage efficiency, 80, 322
str.c, sample, 53
strace, program, 370
Strategized Locking, design pattern, 337
Strategy Bridge, design pattern, 337
strcat, C library, 99
strcmp, C library, 99
strcspsn.c, sample, 232
strdup, C library, 66
stream editor, 228, 348, see also sed, program
STREQ, identifier name, 22
strftime, 3, sample, 251
strftime.pdf, sample, 251
string, 72
String, Java class, 20
StringUtils.java, sample, 34
strlen.c, sample, 72
strlen, C library, 72
strncat, C library, 99
strncpy, C library, 99
@strong, Texinfo command, 263
struct, keyword, 62, 75, 92, 101, 145
structured programming, 54
stub, 272
stubs.c, sample, 121, 122, 125
style guide, see guidelines
style, sample, 50
suffix, 225
.SUFFIX, makefile command, 194
suite, Java method, 220
Sun, 8
supplementary material, 14
Swing, 286, 329
switch, keyword, 32, 212, 231, 360
symbol table, 124
symbolic code, 227, 360, 362
symbolic name, 203
synchronization, 162
synchronized, Java keyword, 163, 164, 165
sys_write, 280
sys, sample, 165, 295, 346, 353, 368
sys_generic.c, sample, 280
syscalls.master, sample, 280, 355
syslog, Unix-specific function, 216
system manager, 289
system specification, see requirements specification
system structure, 245, 268
System V, 162
System, Java class, 201
systime.c, sample, 217
sz, Hungarian prefix, 236

T
%T%, revision-id tag, 206
\t, 25
  .t, file extension, 228
T, Hungarian prefix, 236
tab, 25, 228
table, 95, 101
tables.c, sample, 114
tag, 203, 235, 354
tags, file name, 343
tail, 107
tail recursion, 147
talk, program, 268
talkd.h, sample, 268
TAO_Singleton.h, sample, 332
tape.c, sample, 67, 154
tar, program, 82
tar.h, sample, 82
  .tar, file extension, 227
task manager, 87, 220
tbl, program, 227
  .tbl, file extension, 227
tbl, Hungarian prefix, 237
Tcl/Tk, 98, 201, 228, 330
  .tcl, file extension, 228
TCP, 77, 269
  tcp.h, sample, 77
TCP/IP, 186, 247, 279
tcp_fsm.h, sample, 246
tcpdump, program, 132, 249, 371
tcpdump.8, sample, 249
tcpdump.pdf, sample, 249
tearDown, Java method, 220
tee, program, 275
telldir, Unix-specific function, 319
template, keyword, 315
temporary files, 274
term.c, sample, 42
Test, 228
test case, 28, 220, 249, 259
test harness, 220, 374
Test scripts, 183
test specification, 242
test suite, 220
test, directory name, 183
  .test, file extension, 228
test, pseudo-target, 194
TEST, sample, 214, 215
TestCase, Java class, 220
testing, 214, 215
TestRunner, 220
tex, C library, 390
TeX, 227, 228, 260, 263, 265, 266
  .tex, file extension, 228
  .texi, file extension, 228, 265
Texinfo, 227, 228, 239, 260
  .TH, troff command, 262
THE, 337
this, 302, 309
this, Java keyword, 302
this, keyword, 309
thread, 66, 155, 162, 216
Thread Pool, design pattern, 337
Thread-per Request, design pattern, 337
Thread-per Session, design pattern, 337
Thread-Safe Interface, design pattern, 337
thread.h, sample, 162
Thread, Java class, 162, 163
tiered architectures, 269
throw, keyword, 154
Throwable, Java class, 150
throws, Java keyword, 151, 154
@throws, javadoc tag, 263
tiered architectures, 269
time, Hungarian prefix, 237
timed, program, 269
timed.c, sample, 31
timed.h, sample, 269
TLD, 183
tm_year, C library, 390
TMPfunc.out, 213
tmr, Hungarian prefix, 237
tn3270, 212
TODO, 233
TODO, file name, 226
Together ControlCenter, 249, 279
token, 129, 333
Tokenizer.java, sample, 391, 392
tolower, C library, 42
toLowerCase, Java method, 42
Tomcat, 277, 278
tool, 183
compiler, 360
non-software, 375
roll your own, 357
run-time, 370
tools, 14, 210, 275, 339
tools, directory name, 183
top, program, 87
topological sort, 131, 192
trace, 203, 370
trace.c, sample, 67
Trace.java, sample, 394
tracing statement, 216
transaction monitor, 269
transport, 336
traversal, 128, 139
traverse
circular list, 122
singly linked list, 117
tree, 128
tree, 95, 125
AVL, 130
binary, 126
parse, 129
tree.c, sample, 127, 128
tree.h, sample, 126
troff, program, 227, 260, 263, 274, 359
try, Java keyword, 150, 152, 153
tsleep, 165
ts, program, 131, 376
ts.c, sample, 131, 139
t, sample, 376
tty noise, 385
tty.c, sample, 70
tv, C library, 76
tv.h, C library, 76
two-tier, 269
txt, Hungarian prefix, 237
type field, 82
typedef, keyword, 91, 92, 296
types.h, sample, 92
typographical conventions, 10
tzfile.h, sample, 386

U
%U%, revision-id tag, 206
UDP/IP, 279
ufs_strategy, 281
ufs_vnodes.c, sample, 281
ufsspec_write, 281
UML, 203, 249, 275, 291, 332, 378, see also
diagram
undo, 105
undocumented features, 254
Unicode, 97, 248
Unified Modeling Language, see UML
uninitialized variables, 23
union_write, 281
union, keyword, 80, 145
union_vnodes.c, sample, 281
uniq, program, 352, 357
Unix, 160, 371
Unix-specific function
alloca, 88
close, 80, 281
closedir, 319
DB, 319
dbopen, 319
de1, 319
exec, 281
execve, 222, 223
fork, 160, 166, 281
get, 319
getenv, 201
getopt, 22, 24, 28, 238, 264, 273
getopt_long, 238
ioctl, 80
mutex_clear, 162
open, 281
opendir, 35, 319
poll, 80
pthread_cond_signal, 163, 164
pthread_cond_wait, 163, 164
pthread_cond_destroy, 163
pthread_cond_init, 163
pthread_create, 162, 163
Index

pthread_exit, 163
pthread_mutex_destroy, 162, 163
pthread_mutex_init, 163
pthread_mutex_lock, 163
pthread_mutex_unlock, 163
pthread_self, 163
pthread_yield, 163
put, 319
queue, 319
read, 80, 281
readdir, 35, 319
seekdir, 319
seq, 319
sigaction, 167
signal, 167
sleep, 163
syslog, 216
telldir, 319
wait, 166
write, 80, 166, 280, 281
unix.kbd, sample, 213
upgrade.sh, sample, 20
upper bound, 100
usage, identifier name, 26
use, Ada keyword, 299
use, Perl keyword, 299
user documentation, 242
user reference manual, 242
using, keyword, 298
util.c, sample, 7, 161
util_win32.c, sample, 234
utils.h, sample, 296
WIN, 376

V
v_increment.c, sample, 294
valves, sample, 277
var.c, sample, 52, 233
@var, Texinfo command, 263
variable, 25
variant, 56
.vbp, file extension, 227
vcf, sample, 401
vcfLicense.txt, sample, 457
vector, 95, 96
vectord.s, sample, 281
vector, C++ library, 100
Vector, Java class, 319
version, 203
version control system, see revision control system
version.c, sample, 73
version.h, file name, 226
@version, javadoc tag, 263
vertex, 62, 131
vfs_bio.c, sample, 281
vfs_subr.c, sample, 79, 234
vfs_vnops.c, sample, 280
vgrind, program, 368
vi, program, 253, 340, 344, 385
vi, sample, 294, 347, 353
viewres.c, sample, 96
vim, program, 229, 233, 344, 345, 365
virtual
  function table, 313
  machine, 113, 279
  method, 79, 282
  virtual, keyword, 306
virus, 99
Visitor, design pattern, 337
Visual . . . , see Microsoft . . .
vixie-security.pdf, sample, 248
vixie-security.ps, sample, 248
vm86.c, sample, 176
vm_glue.c, sample, 289
vm_swap.c, sample, 174
VMS, 227
vn_write, 280
vnode.h, sample, 313
vnode_if.c, sample, 313
vnode_if.h, sample, 313
vnode_if.src, sample, 280, 281, 313
void, keyword, 317
volatile, keyword, 77, 168
VOP_STRATEGY, 281
VOP_WRITE, 280
vs_smap.c, sample, 35
vsnprintf, C library, 99
vstrprintf, C library, 99
VT-100, 222
vtbl, 313
vttest, 222
vttest, sample, 222

W
%W%, revision-id tag, 206
w, Hungarian prefix, 236
wait, Java method, 163, 164
wait, Unix-specific function, 166
WaitForSingleObject, Win32 SDK, 163
wakeup, 281
WARDirContextTestCase.java, sample, 220
warning messages, 26, 360
wc.c, sample, 283, 372
wd80x3.c, sample, 355
Web browser, 186
WebServer.java, sample, 299
what, program, 206, 207
while, keyword, 30, 174, 360
wildmat.c, sample, 99
Win32, 15, 376
Win32 SDK
CloseHandle, 163
CreateEvent, 163
CreateMutex, 163
CreateThread, 162, 163
ExitThread, 163
GetCurrentThreadId, 163
PulseEvent, 163
RegCloseKey, 201
RegOpenKey, 201
RegQueryValueEx, 201
ReleaseMutex, 163
ReportEvent, 216
Sleep, 163
WaitForSingleObject, 163
WinMain, 20
win32thread.c, sample, 162
win32thread.h, sample, 162
xdiff, program, 357
xmessage, sample, 329
XML, 191, 196, 261, 263, 327
XMLByteStreamCompiler.java, sample, 299
Xpoll.h, sample, 116, 117
.Xr, troff command, 262
xref.c, sample, 63, 126
xref.h, sample, 63
@xref, Texinfo command, 263
Xrm.c, sample, 117
XrmUniqueQuark, 158

window.c, sample, 126
window, sample, 268, 294
Windows . . . , see Microsoft Windows . . .
WinDump, 371
WinMain, Win32 SDK, 20
with, Ada keyword, 299
Word, see Microsoft Word
work crew, 156, 157
$Workfile:$, revision-id tag, 206
World Wide Web, 268
worm, 99
worms.c, sample, 46
Wrapper Facade, design pattern, 337
write, blackboard operation, 270
write, Unix-specific function, 80, 166, 280, 281

X
X Window System, 158, 162, 190, 195, 201, 212, 322
X Window System library
XtGetApplicationResources, 201
X.25, 256
xargs, program, 353
.xbm, file extension, 227
xcalc.c, sample, 201
xdiff, program, 357
xdryp.c, sample, 272
Xev, 372
xf86bcache.c, sample, 125
xfontsel.c, sample, 286
XFree86-3.3, sample, 402
xmalloc, identifier name, 85
xman.ad, sample, 286
xmessage, sample, 329
XML, 191, 196, 261, 263, 327
XMLByteStreamCompiler.java, sample, 299
Xpoll.h, sample, 116, 117
.Xr, troff command, 262
xref.c, sample, 63, 126
xref.h, sample, 63
@xref, Texinfo command, 263
Xrm.c, sample, 117
XrmUniqueQuark, 158
<table>
<thead>
<tr>
<th>Xserver, sample, 268</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xt, 286, 329</td>
</tr>
<tr>
<td>Xt, sample, 313</td>
</tr>
<tr>
<td>XtGetApplicationResources, X-Windows library, 201</td>
</tr>
<tr>
<td>Xthreads.h, sample, 162</td>
</tr>
<tr>
<td>Xtransam.c, sample, 158</td>
</tr>
<tr>
<td>xxdiff, program, 357</td>
</tr>
<tr>
<td>xxgdb, program, 374</td>
</tr>
<tr>
<td>XXX, identifier name, 233, 354</td>
</tr>
<tr>
<td>yield, Java method, 163</td>
</tr>
<tr>
<td>yp_first.c, sample, 272</td>
</tr>
<tr>
<td>ypserv.c, sample, 272</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>%Z%, revision-id tag, 206</td>
</tr>
<tr>
<td>.Z, file extension, 227</td>
</tr>
<tr>
<td>Zc, editor command, 345</td>
</tr>
<tr>
<td>zic.c, sample, 334</td>
</tr>
<tr>
<td>Zm, editor command, 345</td>
</tr>
<tr>
<td>Zo, editor command, 345</td>
</tr>
<tr>
<td>zombie, 166</td>
</tr>
<tr>
<td>zopen.c, sample, 35, 65, 74</td>
</tr>
<tr>
<td>Zr, editor command, 345</td>
</tr>
<tr>
<td>zsh, program, 353</td>
</tr>
<tr>
<td>zutil.h, sample, 296</td>
</tr>
</tbody>
</table>