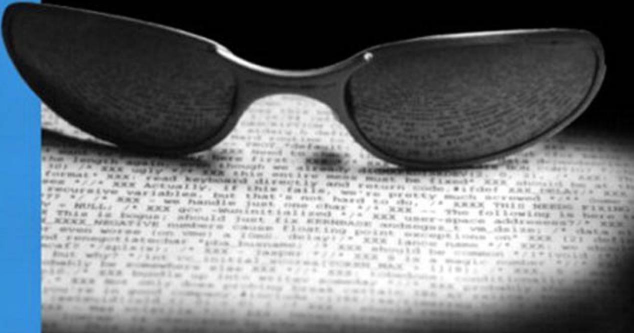
Effective SOFTWARE DEVELOPMENT SERIES

Scott Meyers, Consulting Editor

CODE *Reading*

The Open Source Perspective



Diomidis Spinellis

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.
ISBN 0-201-79940-5 (pbk. : alk. paper)
1. Computer programming. 2. Coding theory. I. Title.
QA76.6 .S675 2003
005.1–dc21 2002038563

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

This page intentionally left blank

Contents

Foreword

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 it 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.[1] 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

---

[1] http://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.

2

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:

```
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

```c
/*
 * Copyright (c) 1989, 1993
 *    The Regents of the University of California.  All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software
 *    must display the following acknowledgement:
 *    This product includes software developed by the University of
 *    California, Berkeley and its contributors.
 * 4. Neither the name of the University nor the names of its contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ''AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 */
#include <sys/cdefs.h>

#ifndef lint
__COPYRIGHT(
"@(#) Copyright (c) 1989, 1993\n\
    The Regents of the University of California.  All rights reserved.\n");

__RCSID("$NetBSD: echo.c,v 1.7 1997/07/20 06:07:03 thorpej Exp $");
#endif /* not lint */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int    main __P((int, char *[]));

int
main(argc, argv)
    int argc;
    char *argv[];
{
    int nflag;

    /* This utility may NOT do getopt(3) option parsing. */
    if (*++argv && !strcmp(*argv, "-n")) {
        ++argv;
        nflag = 1;
    }
    else
        nflag = 0;

    while (*argv) {
        (void)printf("%s", *argv);
        if (*++argv)
            putchar(' ');
    }
    if (!nflag)
        putchar('\n');
    exit(0);
}
```

- **1** Comment (copy right and distribution license), ignored by the compiler. This license appears on most programs of this collection. It will not be shown again.
- Copyright and program version identifiers that will appear as strings in the executable program
- **2** Standard library headers for: `printf`, `exit`, `strcmp`
- Function declaration with macro to hide arguments for pre-ANSI compilers
- **3** The program starts with this function
- **4** Number of arguments to the program
- **4** The actual arguments (starting with the program name, terminated with NULL)
- **5** When true output will not be terminated with a newline
- **6** The first argument is `-n`
- **6** Skip the argument and set `nflag`
- There are arguments to process
- **7** Print the argument
- Is there a next argument? (Advance `argv`)
- Print the separating space
- Terminate output with newline unless `-n` was given
- Exit program indicating success

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[3]

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

while the corresponding Java class method definition is[4]

```
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[5] 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:[6]

```
#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:[7]

```
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:[8]

```
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). ⚠

---

[7] jt4/catalina/src/share/org/apache/catalina/startup/CatalinaService.java:136–143
[8] netbsdsrc/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:[9]

```
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?

---

[9] netbsdsrc/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*,[10] Figure 2.3 contains the main code body,[11] and Figure 2.5 (in Section 2.5) contains the two supplementary functions used.[12]

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

[10]netbsdsrc/usr.bin/expand/expand.c:36–62
[11]netbsdsrc/usr.bin/expand/expand.c:64–151
[12]netbsdsrc/usr.bin/expand/expand.c:153–185

Header files

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

1 Global variables

```
int     nstops;
int     tabstops[100];
```

2 Forward function declarations

```
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

```
int
main(int argc, char *argv)
{
    int c, column;
    int n;

    while ((c = getopt (argc, argv, "t:")) != -1) {
        switch (c) {
        case 't':
            getstops(optarg);
            break;
        case '?': default:
            usage();
        }
    }
    argc -= optind;
    argv += optind;

    do {
        if (argc > 0) {
            if (freopen(argv[0], "r", stdin) == NULL) {
                perror(argv[0]);
                exit(1);
            }
            argc--, argv++;
        }
        column = 0;
        while ((c = getchar()) != EOF) {
            switch (c) {
            case '\t':
                if (nstops == 0) {
                    do {
                        putchar(' ');
                        column++;
                    } while (column & 07);
                    continue;
                }
                if (nstops == 1) {
                    do {
                        putchar(' ');
                        column++;
                    } while (((column - 1) % tabstops[0]) != (tabstops[0] - 1));
                    continue;
                }
                for (n = 0; n < nstops; n++)
                    if (tabstops[n] > column)
                        break;
                if (n == nstops) {
                    putchar(' ');
                    column++;
                    continue;
                }
                while (column < tabstops[n]) {
                    putchar(' ');
                    column++;
                }
                continue;
            case '\b':
                if (column)
                    column--;
                putchar('\b');
                continue;
            default:
                putchar(c);
                column++;
                continue;
            case '\n':
                putchar(c);
                column = 0;
                continue;
            }
        }
    } while (argc > 0);
    exit(0);
}
```

1 Variables local to main
2 Argument processing using `getopt`
3 Process the `-t` option
Switch labels grouped together
4 End of `switch` block
5 At least once
6 Process remaining arguments
7 Read characters until EOF
Tab character
8 Process next character
8
8
8
Backspace
8
All other characters
8
Newline
8
End of `switch` block
End of `while` block
End of do block

Figure 2.3 Expanding tab stops (main part).

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),[13] 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.

[13] http://msdn.microsoft.com

GETOPT (3) UNIX Programmer's Manual GETOPT (3)

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

Figure 2.4 The *getopt* manual page.

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).[14]

```
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

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

will evaluate as

```
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

[14] cocoon/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]

```
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]

```
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]

```
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.[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.

[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.[19]

```
for (i = 0; i < len; i++) {
```

i Loops of this type appear very frequently in programs; learn to read them as "execute the body of code `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:[20]

```
for ( i = 0; i <= extrknt; i++ )
```

Loop `month - 1` times:[21]

```
for (i = 1; i < month; i++)
```

[19] cocoon/src/java/org/apache/cocoon/util/StringUtils.java:85
[20] netbsdsrc/usr.bin/fsplit/fsplit.c:173
[21] netbsdsrc/usr.bin/cal/cal.c:332

Loop `nargs` times:[22]

```
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:[23]

```
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`.[24]

```
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.[25]

```
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

```
static void
getstops(char *cp)
{
    int i;

    nstops = 0;
    for (;;) {
        i = 0;
        while (*cp >= '0' && *cp <= '9')
            i = i * 10 + *cp++ - '0';
        if (i <= 0 || i > 256) {
bad:
            fprintf(stderr, "Bad tab stop spec\n");
            exit(1);
        }
        if (nstops > 0 && i <= tabstops[nstops-1])
            goto bad;
        tabstops[nstops++] = i;
        if (*cp == 0)
            break;
        if (*cp != ',' && *cp != ' ')
            goto bad;
        cp++;
    }
}
```

```
static void
usage(void)
{
    (void)fprintf (stderr, "usage: expand [-t tablist] [file ...]\n");
    exit(1);
}
```

1 Parse tab stop specification
2 Convert string to number
3 Complain about unreasonable specifications
4 Verify ascending order
5 Break out of the loop
6 Verify valid delimiters
7 `break` will transfer control here
8 Print program usage and exit

Figure 2.5 Expanding tab stops (supplementary functions).

example—taken out of *init*, the program that continuously loops, controlling all Unix processes—is an exception.[26]

```
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]

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

[27] perl/lib/unicode/mktables.PL:415–425

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.

i 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.[28]

```
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.[29]

```
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] jt4/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]

```
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

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

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

$$0 \le c \le 9$$

[30]cocoon/src/scratchpad/src/org/apache/cocoon/treeprocessor/MapStackResolver.java:201–244

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]

```
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.

Iteration	`i`	`i*10`	`*cp`	`*cp-'0'`	`i'`	`*cp'`
0	`0`	`0`	`'2'`	`2`	`2`	`'4'`
1	`2`	`20`	`'4'`	`4`	`24`	`0`

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

[31] netbsdsrc/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 (`||`) 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 || b) <=> !a && !b
!(a && b) <=> !a || !b
```

We can thus transform the testing code as follows:

```
i <= 0 || i > 256  <=>
!(!(i <= 0) && !(i > 256)) <=>
!(i > 0 && i <= 256) <=>
!(0 < i && i <= 256) <=>
¬(0 < i ≤ 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 `||` 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 <=> 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`.[33]

```
if (t != NULL && t->type != TEOF && interactive && really_exit)
        really_exit = 0;
```

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

```
if (argv == NULL || argv[1] == NULL || argv[2] == NULL)
    return -1;
```

i 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.

---

[33] netbsdsrc/bin/ksh/main.c:606–607

[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, &o_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);
}
```

1 Failure; exit with an error
2 Failure; exit with an error
3 Normal function exit (success)
4 Common error handling code

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[35] 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.

---

[35] netbsdsrc/bin/pax/pax.c:309–412

```
again:
    if ((p = fgets(line, BUFSIZ, servf)) == NULL)      Read a line; return on EOF
        return (NULL);

    if (*p == '#')                                     Comment? Retry
        goto again;
    cp = strpbrk(p, "#\n");
    if (cp == NULL)                                    Incomplete line? Retry
        goto again;
    *cp = '\0';                                        Complete entry
    [...]
    return (&serv);
```

Figure 2.7 The use of `goto` to reexecute code.

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,[36] 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[37] 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.

---

[36] netbsdsrc/lib/libc/net/getservent.c:65–104

[37] netbsdsrc/sys/dev/ic/ncr5380sbc.c:1575–1654

```
for (;;) {
    [...]
    if ((sc->sc_state & NCR_DROP_MSGIN) == 0) {
        if (n >= NCR_MAX_MSG_LEN) {
            ncr_sched_msgout(sc, SEND_REJECT);
            sc->sc_state |= NCR_DROP_MSGIN;
        } else {
            [...]
            if (n == 1 && IS1BYTEMSG(sc->sc_imess[0]))
                goto have_msg;
            [...]
        }
    }
    [...]
}
have_msg:
```

for loop

Exit the for loop

goto target

Figure 2.8 Exiting a loop using the `goto` statement.

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:

```
if (*cp == 0)
    break;
else if (*cp != ',' && *cp != ' ')
    goto bad;
else
    cp++;
```

i 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]

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

i The code makes excessive use of the conditional operator `?:`. Read expressions using the conditional operator like `if` code. As an example, read the expression[39]

```
sign ? -n : n
```

as follows:

> "If `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 `if` statement, we can also format it like an `if` statement; one that uses `x ?` instead of `if (x)`, parentheses instead of curly braces, and `:` instead of `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` or `!y`)
2. `bottom` or `last`
3. All other values

[38] netbsdsrc/games/worms/worms.c:419
[39] netbsdsrc/bin/csh/set.c:852

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

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

Figure 2.9 A conditional expression formatted like an `if` statement (*left*) and like cascading `if–else` statements (*right*).

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];
```

Location map

To store the x, y map offsets

Determine x offset

Determine y offset

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.

```
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

```
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.

```
/*
 * 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 GNU[41] or BSD[42] 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

[41] http://www.gnu.org/prep/standards_toc.html
[42] netbsdsrc/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.[43] 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

[43] http://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:[44]

```
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`.[45]

```
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.[46]

```
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. [i]

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.[47] [i]

  ```
  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.[48] [i]

  ```
  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. ⚠

[47] netbsdsrc/bin/csh/str.c:460
[48] 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.[49]

```
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
}
```

[49] jt4/catalina/src/share/org/apache/catalina/loader/StandardLoader.java:886–905

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 `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 `enum.hasMoreElements()` will be true and that, therefore, the enclosed statement

```
NameClassPair ncPair = (NameClassPair) enum.nextElement();
```

will execute without a problem. The same reasoning also applies to `if` statements. In the code below you can be sure that when `links.add` is executed the `links` collection will not contain a `next` element.[50]

```
if (!links.contains(next)) {
    links.add(next);
}
```

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.

---

[50]cocoon/src/java/org/apache/cocoon/Main.java:574–576

```
void *
bsearch(key, base0, nmemb, size, compar)
    register const void *key;
    const void *base0;
    size_t nmemb;
    register size_t size;
    register int (*compar) __P((const void *, const void *));
{
    register const char *base = base0;
    register int lim, cmp;
    register const void *p;

    for (lim = nmemb; lim != 0; lim >>= 1) {
        p = base + (lim >> 1) * size;
        cmp = (*compar)(key, p);
        if (cmp == 0)
            return ((void *)p);
        if (cmp > 0) {    /* key > p: move right */
            base = (char *)p + size;
            lim--;
        } /* else move left */
    }
    return (NULL);
}
```

Item to search for
Start of element array
Number of elements
Size of each element
Function to compare two elements
Locate a point in the middle
Compare element against key
Found; return its position
Adjust base upwards
Not sure why this is needed
Not found

Figure 2.11 Binary search implementation.

Consider the standard C library implementation of the binary search algorithm listed in Figure 2.11.[51]. We can see that it works by gradually reducing the search interval stored in the `lim` variable and adjusting the start of the search range stored in `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 `lim` is odd, moving left simply involves halving `lim`: e.g., when `lim` is 5 we look at item 2, so we change `lim` to 2 so that we will look at items 0 & 1. If `lim` is even, the same applies. If `lim` is odd, moving right again involves halving `lim`, this time moving the `base` up one item past `p`: e.g., when `lim` is 5 we change `base` to item 3 and make `lim` 2 so that we will look at items 3 and 4. If `lim` is even, however, we have to shrink it by one before halving: e.g., when `lim` is 4, we still looked at item 2, so we have to make `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

[51] netbsdsrc/lib/libc/stdlib/bsearch.c

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]

```
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

$$\texttt{max} = \text{maximum}\{\texttt{depths}[0 : n_0)\}$$

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

$$\texttt{max} = \text{maximum}\{\texttt{depths}[\texttt{n} : n_0)\}$$

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 $[\texttt{n} : n_0)$ 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

[52] XFree86-3.3/xc/lib/Xt/GCManager.c:252–256

```
    register const char *base = base0;
    for (lim = nmemb; lim != 0;) {
        p = base + lim  / 2;
        cmp = (*compar)(key, p);
        if (cmp == 0)
            return ((void *)p);
        if (cmp > 0) {
            /*Key > p: move right*/

            base = p + 1;


            lim--;

        } /* else move left */

        lim /= 2;
    }
    return (NULL);
}
```

1 R in [base, base + nmemb)
2 R in [base, base + lim)
3 R in (p, base + lim)
R in [p + 1, base + lim)
4 base = base_old + lim / 2 + 1
base_old = base - lim / 2 - 1
R in [base, base_old + lim)
R in [base, base - lim / 2 - 1 + lim)
5 R in [base, base - (lim + 1) / 2 - 1 + lim + 1)
R in [base, base + lim - (lim + 1) / 2)
R in [base, base + lim / 2)
6 R in [base, p)
R in [base, base + lim / 2)
7 R in [base, base + lim)

Figure 2.12 Maintaining the binary search invariant.

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 `lim` are used within the loop to delimit the search range, our invariant will be $R \in [\texttt{base} : \texttt{base} + \texttt{lim})$. 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 `compar` is always called with an argument from within the invariant's range ($\texttt{base} + \texttt{lim}/2$), and since `lim` (our variant) is halved after every loop iteration, we can be sure that `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: $R \in [\texttt{base0} : \texttt{base0} + \texttt{nmemb})$. However, after Figure 2.12:1 this can be expressed as $R \in [\texttt{base} : \texttt{base} + \texttt{nmemb})$, and after the `for` assignment (Figure 2.12:2) as $R \in [\texttt{base} : \texttt{base} + \texttt{lim})$—our invariant. We have thus established that our invariant holds at the beginning of the loop.

The result of the `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

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

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

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

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

$$\begin{aligned} \texttt{base} &= \texttt{base}_{old} + \frac{\texttt{lim}}{2} + 1 \Leftrightarrow \\ \texttt{base}_{old} &= \texttt{base} - \frac{\texttt{lim}}{2} - 1. \end{aligned}$$

By substituting the above result in the invariant we obtain

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

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

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

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

$$R \in [\texttt{base} : \texttt{p}) \equiv$$
$$R \in \left[\texttt{base} : \texttt{base} + \frac{\texttt{lim}}{2}\right).$$

Note that the invariant is now the same for both comparison results. Furthermore, when `lim` is halved at Figure 2.12:7 we can substitute its new value in the invariant to obtain $R \in [\texttt{base} : \texttt{base} + \texttt{lim})$, 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 `lim` becomes zero, the range where the value can lie is empty, and it is therefore correct to return `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 `for` loop just compares two elements; the `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 `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

Index

Symbols

D

P

Q

R

S